

# Verification of the L4 Kernel Memory Allocator

Harvey Tuch   Gerwin Klein   Michael Norrish



## **Abstract**

This proof document contains the Isabelle/HOL scripts together with the code and invariants for the formal verification of the L4 kernel memory allocator.

# Contents

1	Source Code and Invariants	2
2	Data Abstraction Predicates	9
3	Shared Lemmas for Non-Separation Case	30
4	Shared Lemmas for Separation Logic Case	33
5	kalloc without separation logic	41
6	kalloc with separation logic	49
7	kfree without separation logic	66
8	kfree with separation logic	93

## 1 Source Code and Invariants

```
typedef unsigned int u64_t;
typedef unsigned int u32_t;
typedef unsigned short u16_t;
typedef unsigned char u8_t;

typedef signed int s64_t;
typedef signed int s32_t;
typedef signed short s16_t;
typedef signed char s8_t;

typedef u32_t word_t;

typedef void* addr_t;

word_t *kmem_free_list;

void init (void * start, void * end);
void free (void * address, word_t size);

void * alloc (word_t size);
void * alloc_aligned (word_t size, word_t alignment, word_t mask);

void init(void * start, void * end)
{
    kmem_free_list = 0;

    free(start, (word_t)end - (word_t)start);
}
```

```

/** FNSPEC
free_spec:
 $\forall s. \Gamma \vdash$ 
 $\{ \{ s. (\exists \text{start}. \text{lift}_g (\text{abs } \tau_h \tau_d) \text{kmem\_free\_list\_addr} = \text{Some start} \wedge$ 
 $\text{free\_set}_h \tau_h \tau_d \text{start NULL F}) \wedge$ 
 $\text{KMC} \leq \text{ptr\_val } \tau_{\text{address}} \wedge$ 
 $\text{F} \cap \{ \text{ptr\_val } \tau_{\text{address}} \dots \text{ptr\_val } \tau_{\text{address} + \text{max } \tau_{\text{size}} \text{KMC}} \} = \{ \} \wedge$ 
 $\text{KMC} \text{ udvd } \tau_{\text{size}} \wedge$ 
 $\text{ptr\_val } \tau_{\text{address}} \notin \text{F} \wedge$ 
 $\models_t (\text{ptr\_coerce } \tau_{\text{address}} :: \text{word32 ptr}) \wedge 4 \text{ dvd unat } (\text{ptr\_val } \tau_{\text{address}}) \wedge$ 
 $\text{ptr\_val } \tau_{\text{address}} < \text{ptr\_val } \tau_{\text{address}} + \text{max } \tau_{\text{size}} \text{KMC} \wedge$ 
 $\text{ptr\_val } \text{kmem\_free\_list\_addr} \notin \{ \text{ptr\_val } \tau_{\text{address}} \dots \text{ptr\_val } \tau_{\text{address} + \text{max } \tau_{\text{size}} \text{KMC}} \} \wedge$ 
 $\text{ptr\_val } \text{kmem\_free\_list\_addr} \notin \text{F} \} \}$ 
 $\text{PROC free}(\tau_{\text{address}}, \tau_{\text{size}})$ 
 $\{ \{ \text{free\_set}_h \tau_h \tau_d (\text{lift } \tau_h \text{kmem\_free\_list\_addr}) \text{NULL} (\text{free } \tau_{\text{address}} \tau_{\text{size}} \text{F}) \wedge$ 
 $\tau_{\text{size}} = \text{max } \tau_{\text{size}} \text{KMC} \} \}$ 
*/
void free(void * address, word_t size)
{
word_t* p;
word_t* prev, *curr;

size = size >= 1024 ? size : 1024;

/** AUXUPD: (True, ptr_tags (unat  $\tau_{\text{size}} \text{div unat KMC}$ ) (ptr_val  $\tau_{\text{address}}$ )  $\tau_d$ ) */

for (p = (word_t*)address;
p < ((word_t*)((word_t)address) + size - (1024));
p = (word_t*) *p)
/** INV:
 $\{ \{ \text{F} \cap \{ \text{ptr\_val } \tau_{\text{address}} \dots \text{ptr\_val } \tau_{\text{address} + \text{max } \tau_{\text{size}} \text{KMC}} \} = \{ \} \wedge$ 
 $\text{F} \cap \text{chunks } \tau_{\text{address}} (\text{ptr\_val } \tau_{\text{address}} + \tau_{\text{size}} - \text{KMC}) = \{ \} \wedge$ 
 $\text{ptr\_val } \text{kmem\_free\_list\_addr} \notin \text{F} \wedge$ 
 $\text{ptr\_val } \tau_p < \text{ptr\_val } \tau_{\text{address}} + \tau_{\text{size}} \wedge$ 
 $\text{KMC} \leq \text{ptr\_val } \tau_{\text{address}} \wedge$ 
 $\text{KMC} \text{ udvd } \tau_{\text{size}} \wedge$ 
 $\text{KMC} \text{ udvd } (\text{ptr\_val } \tau_p - \text{ptr\_val } \tau_{\text{address}}) \wedge$ 
 $\text{ptr\_coerce } \tau_{\text{address}} \leq \tau_p \wedge$ 
 $\text{ptr\_val } \tau_p \notin \text{F} \wedge \text{ptr\_val } \tau_{\text{address}} \notin \text{F} \wedge$ 
 $\text{free\_set}_h \tau_h \tau_d (\text{ptr\_coerce } \tau_{\text{address}}) \tau_p (\text{chunks } \tau_{\text{address}} (\text{ptr\_val } \tau_p - \text{KMC})) \wedge$ 
 $(\exists \text{start}. \text{lift}_g (\text{abs } \tau_h \tau_d) \text{kmem\_free\_list\_addr} = \text{Some start} \wedge$ 
 $\text{free\_set}_h \tau_h \tau_d \text{start NULL F}) \wedge$ 
 $\text{F} \cap \text{chunks } \tau_{\text{address}} (\text{ptr\_val } \tau_p - \text{KMC}) = \{ \} \wedge$ 
 $(\forall q \in \text{chunks } \tau_{\text{address}} (\text{ptr\_val } \tau_{\text{address}} + (\text{max } \tau_{\text{size}} \text{KMC} - \text{KMC})).$ 
 $\models_t (\text{Ptr } q :: \text{word32 ptr})) \wedge$ 
 $\text{ptr\_val } \tau_p \in \text{chunks } \tau_{\text{address}} (\text{ptr\_val } \tau_{\text{address}} + (\text{max } \tau_{\text{size}} \text{KMC} - \text{KMC})) \wedge$ 
 $\tau_{\text{address}} = \tau_{\text{address}} \wedge$ 
 $\tau_{\text{size}} = \text{max } \tau_{\text{size}} \text{KMC} \wedge$ 
 $\models_t \text{kmem\_free\_list\_addr} \wedge$ 
 $\models_t (\text{ptr\_coerce } \tau_{\text{address}} :: \text{word32 ptr}) \wedge$ 
 $\models_t \tau_p$ 
 $\} \}$ 
*/
*p = (word_t) p + (1024);

for (prev = (word_t*) &kmem_free_list, curr = kmem_free_list;
curr && (address > (void *)curr);
prev = curr, curr = (word_t*) *curr)

```

```

/** INV:
  { F  $\cap$  chunks  $\hat{address}$  (ptr_val  $\hat{address}$  +  $\hat{size}$  - KMC) = {}  $\wedge$ 
    ptr_val kmem_free_list_addr  $\notin$  F  $\wedge$ 
    ptr_val  $\hat{address}$  +  $\hat{size}$  - KMC  $\leq$  ptr_val  $\hat{p}$   $\wedge$ 
    KMC  $\leq$  ptr_val  $\hat{address}$   $\wedge$ 
    ptr_val  $\hat{p}$  < ptr_val  $\hat{address}$  +  $\hat{size}$   $\wedge$ 
    ptr_coerce  $\hat{address}$   $\leq$   $\hat{p}$   $\wedge$ 
    KMC udvd (ptr_val  $\hat{p}$  - ptr_val  $\hat{address}$ )  $\wedge$ 
    KMC udvd  $\hat{size}$   $\wedge$ 
    lift  $\hat{t}_h$   $\hat{prev}$  = ptr_val  $\hat{curr}$   $\wedge$ 
    (ptr_val  $\hat{prev}$   $\in$  F  $\vee$   $\hat{prev}$  = ptr_coerce (kmem_free_list_addr :: word32 ptr))  $\wedge$ 
    (ptr_val  $\hat{curr}$   $\in$  F  $\vee$   $\hat{curr}$  = NULL)  $\wedge$ 
    ptr_val  $\hat{p}$   $\notin$  F  $\wedge$  ptr_val  $\hat{address}$   $\notin$  F  $\wedge$ 
    free_set_h  $\hat{t}_h$   $\hat{t}_d$  (ptr_coerce  $\hat{address}$ )  $\hat{p}$  (chunks  $\hat{address}$  (ptr_val  $\hat{p}$  - KMC))  $\wedge$ 
    ( $\exists$  start. lift_g (abs  $\hat{t}_h$   $\hat{t}_d$ ) kmem_free_list_addr = Some start  $\wedge$ 
    free_set_h  $\hat{t}_h$   $\hat{t}_d$  start NULL F)  $\wedge$ 
    F  $\cap$  chunks  $\hat{address}$  (ptr_val  $\hat{p}$  - KMC) = {}  $\wedge$ 
     $\hat{address}$  =  $\hat{s}$ address  $\wedge$ 
     $\hat{size}$  = max  $\hat{s}$ size KMC  $\wedge$ 
     $\models_t$  kmem_free_list_addr  $\wedge$ 
     $\models_t$  (ptr_coerce  $\hat{address}$  :: word32 ptr)  $\wedge$ 
     $\models_t$   $\hat{p}$ 
  }
*/
;

*prev = (word_t) address; *p = (word_t) curr;
}

/** FNSPEC
sep_free_spec:
 $\forall s. \Gamma \vdash$ 
  { s. (sep_free_set_h kmem_free_list_addr NULL F  $\wedge^*$ 
    sep_cut (ptr_val  $\hat{address}$ ) (max  $\hat{size}$  KMC)) (abs  $\hat{t}_h$   $\hat{t}_d$ )  $\wedge$ 
    KMC  $\leq$  ptr_val  $\hat{address}$   $\wedge$  KMC udvd ptr_val  $\hat{address}$   $\wedge$  KMC udvd  $\hat{size}$   $\wedge$ 
    ptr_val  $\hat{address}$  < ptr_val  $\hat{address}$  + max  $\hat{size}$  KMC }
  { sep_free_set_h kmem_free_list_addr NULL (free  $\hat{s}$ address (max  $\hat{s}$ size KMC) F) (abs  $\hat{t}_h$ 
 $\hat{t}_d$ ) }
*/
void sep_free(void * address, word_t size)
{
  word_t* p;
  word_t* prev, *curr;

  size = size >= 1024 ? size : 1024;

  /** AUXUPD: (True,ptr_tag (ptr_coerce  $\hat{address}$ ::word32 ptr)  $\hat{t}_d$ ) */

  for (p = (word_t*)address;
    p < (((word_t*)((word_t)address) + size - (1024)));
    p = (word_t*) *p)
  /** INV:
    { (sep_free_set_h kmem_free_list_addr NULL F  $\wedge^*$  (c_guard  $\vdash_s$   $\hat{p}$ )  $\wedge^*$  rest_of  $\hat{p}$  KMC  $\wedge^*$ 
      sep_cut (ptr_val  $\hat{p}$  + KMC) ( $\hat{size}$  - (ptr_val  $\hat{p}$  - ptr_val  $\hat{address}$ ) - KMC)  $\wedge^*$ 
      sep_free_set (ptr_coerce  $\hat{address}$ )  $\hat{p}$  (chunks  $\hat{address}$  (ptr_val  $\hat{p}$  - KMC)))
      (abs  $\hat{t}_h$   $\hat{t}_d$ )  $\wedge$ 
    }
  */
}

```

```

    ptr_val `address < ptr_val `address + `size ∧
    ptr_val `address ≤ ptr_val `p ∧
    KMC udvd (ptr_val `p - ptr_val `address) ∧
    KMC udvd ptr_val `p ∧
    KMC udvd `size ∧
    `address = saddress ∧
    ptr_val `p ≤ ptr_val `address + `size - KMC ∧
    `size = max ssize KMC ∧ KMC ≤ ptr_val `address }
*/
{
  *p = (word_t) p + (1024);
  /** AUXUPD: (True, ptr_tag (Ptr (ptr_val `p + 1024)::word32 ptr) `t_d) */
}

for (prev = (word_t*) &kmem_free_list, curr = kmem_free_list;
     curr && (address > (void *)curr);
     prev = curr, curr = (word_t*) *curr)
/** INV:
  { (∃G H. (sep_free_set_h kmem_free_list_addr `curr G * sep_free_set `curr NULL H *
    c_guard ⊢s `p * rest_of `p KMC *
    sep_free_set (ptr_coerce `address) `p
      (chunks `address (ptr_val `address + (`size - KMC - KMC)))
    ) (abs `t_h `t_d) ∧
    (`prev = ptr_coerce kmem_free_list_addr ∨ ptr_val `prev ∈ G) ∧
    F = G ∪ H) ∧
    ptr_val `address ≤ ptr_val `address + (`size - KMC) ∧
    KMC ≤ ptr_val `address ∧
    ptr_val `p = ptr_val `address + `size - KMC ∧
    KMC udvd ptr_val `p ∧
    `address = saddress ∧
    `size = max ssize KMC ∧ KMC udvd `size ∧
    (`prev ↔ ptr_val `curr) (abs `t_h `t_d) }
*/
;

*prev = (word_t) address; *p = (word_t) curr;
}

/** FNSPEC
alloc_spec:
∀σ. Γ ⊢
  {σ. free_set_h `t_h `t_d (ptr_coerce kmem_free_list_addr) NULL F ∧
   aligned F ∧
   KMC udvd `size }
  `alloc_ret := PROC alloc(`size)
  { (`alloc_ret ≠ NULL →
    free_set_h `t_h `t_d (ptr_coerce kmem_free_list_addr) NULL
      (alloc `alloc_ret (max σsize KMC) F)) ∧
    (`alloc_ret = NULL → (`t_h, `t_d) = (σt_h, σt_d)) }
*/
void * alloc(word_t size)
{
  word_t* prev;
  word_t* curr;
  word_t* tmp;
  word_t i;

```

```

size = size >= 1024 ? size : 1024;

for (prev = (word_t*) &kmem_free_list, curr = kmem_free_list;
    curr;
    prev = curr, curr = (word_t*) *curr)
/** INV:  $\{ \{ (\exists G H.$ 
    ptr_val  $\dot{\text{prev}} \in G \wedge$ 
    free_set $_h \dot{\text{t}}_h \dot{\text{t}}_d (\text{ptr\_coerce } \text{kmem\_free\_list\_addr}) \dot{\text{curr}} G \wedge$ 
    free_set $_h \dot{\text{t}}_h \dot{\text{t}}_d \dot{\text{curr}} \text{NULL } H \wedge$ 
     $G \cap H = \{ \} \wedge$ 
     $F = G \cup H) \wedge$ 
     $\dot{\text{t}}_h = \sigma \text{t}_h \wedge \dot{\text{t}}_d = \sigma \text{t}_d \wedge$ 
     $\dot{\text{size}} = \max \sigma \text{size } \text{KMC} \wedge$ 
     $\text{KMC } \text{udvd } \sigma \text{size} \wedge$ 
     $1 \leq \dot{\text{size}} \text{ div } \text{KMC} \wedge$ 
    lift  $\dot{\text{t}}_h \dot{\text{prev}} = \text{ptr\_val } \dot{\text{curr}} \wedge$ 
    aligned F  $\} \}$  */
{
    if (!(word_t) curr & (size - 1))
    {
        tmp = (word_t*) *curr;
        for (i = 1; tmp && (i < (size / (1024))); i++)
            /** INV:  $\{ \{ (\dot{\text{tmp}} \neq \text{NULL} \longrightarrow \text{no\_wrap } \dot{\text{curr}} \dot{\text{i}} \wedge ($ 
                 $\exists G H. \text{ptr\_val } \dot{\text{prev}} \in G \wedge$ 
                free_set $_h \dot{\text{t}}_h \dot{\text{t}}_d (\text{ptr\_coerce } \text{kmem\_free\_list\_addr}) \dot{\text{curr}} G \wedge$ 
                free_set $_h \dot{\text{t}}_h \dot{\text{t}}_d \dot{\text{curr}} \dot{\text{tmp}} (\text{chunks } \dot{\text{curr}} (\text{ptr\_val } \dot{\text{curr}} + (\dot{\text{i}} - 1) * \text{KMC})) \wedge$ 
                free_set $_h \dot{\text{t}}_h \dot{\text{t}}_d \dot{\text{tmp}} \text{NULL } H \wedge$ 
                 $G \cap (\text{chunks } \dot{\text{curr}} (\text{ptr\_val } \dot{\text{curr}} + (\dot{\text{i}} - 1) * \text{KMC})) = \{ \} \wedge$ 
                 $H \cap (\text{chunks } \dot{\text{curr}} (\text{ptr\_val } \dot{\text{curr}} + (\dot{\text{i}} - 1) * \text{KMC})) = \{ \} \wedge$ 
                 $G \cap H = \{ \} \wedge$ 
                 $F = G \cup (\text{chunks } \dot{\text{curr}} (\text{ptr\_val } \dot{\text{curr}} + (\dot{\text{i}} - 1) * \text{KMC})) \cup H) \wedge$ 
                 $(\dot{\text{tmp}} = \text{NULL} \longrightarrow ($ 
                 $\exists G H. \text{ptr\_val } \dot{\text{prev}} \in G \wedge$ 
                free_set $_h \dot{\text{t}}_h \dot{\text{t}}_d (\text{ptr\_coerce } \text{kmem\_free\_list\_addr}) \dot{\text{curr}} G \wedge$ 
                free_set $_h \dot{\text{t}}_h \dot{\text{t}}_d \dot{\text{curr}} \text{NULL } H \wedge$ 
                 $G \cap H = \{ \} \wedge$ 
                 $F = G \cup H) \wedge$ 
                 $\dot{\text{t}}_h = \sigma \text{t}_h \wedge \dot{\text{t}}_d = \sigma \text{t}_d \wedge$ 
                 $\text{KMC } \text{udvd } \sigma \text{size} \wedge$ 
                 $\dot{\text{size}} = \max \sigma \text{size } \text{KMC} \wedge$ 
                 $1 \leq \dot{\text{i}} \wedge$ 
                 $\dot{\text{i}} \leq \dot{\text{size}} \text{ div } \text{KMC} \wedge$ 
                lift  $\dot{\text{t}}_h \dot{\text{prev}} = \text{ptr\_val } \dot{\text{curr}} \wedge$ 
                 $\dot{\text{curr}} \neq \text{NULL} \wedge$ 
                aligned F  $\} \}$  */
            {

                if ((word_t) tmp != ((word_t) curr + (1024)*i))
                {
                    tmp = 0;
                    break;
                };
                tmp = (word_t*) *tmp;
            }
        if (tmp)
        {
            *prev = (word_t) tmp;

```

```

    for (i = 0; i < (size / sizeof(word_t)); i++)
        /** INV:  $\{ ($ 
            free_set_h  $\hat{t}_h \hat{t}_d$  (ptr_coerce kmem_free_list_addr) NULL
                (alloc (ptr_coerce  $\hat{curr}$ ) (max  $\sigma$ size KMC) F))  $\wedge$ 
            chunks  $\hat{curr}$  (ptr_val  $\hat{curr}$  + ( $\hat{size}$  - KMC))  $\cap$ 
            alloc (ptr_coerce  $\hat{curr}$ ) (max  $\sigma$ size KMC) F = {}  $\wedge$ 
            aligned F  $\wedge$ 
            KMC udvd  $\hat{size}$   $\wedge$ 
            KMC udvd ptr_val  $\hat{curr}$   $\wedge$ 
            ptr_val  $\hat{curr}$   $\leq$  ptr_val  $\hat{curr}$  + ( $\hat{size}$  - KMC)  $\wedge$ 
             $\hat{curr} \neq$  NULL  $\wedge$ 
             $\hat{size} = \max \sigma$ size KMC  $\}$  */
            curr[i] = 0;

        return curr;
    }
}

return 0;
}

/** FNSPEC
sep_alloc_spec:
 $\forall \sigma. \Gamma \vdash$ 
 $\{ \sigma. (\text{sep\_free\_set\_h kmem\_free\_list\_addr NULL F})^{\text{sep}} \wedge \text{KMC udvd } \hat{size} \}$ 
 $\hat{sep\_alloc\_ret} := \text{PROC sep\_alloc}(\hat{size})$ 
 $\{ (\hat{sep\_alloc\_ret} \neq \text{NULL} \longrightarrow ($ 
    sep_free_set_h kmem_free_list_addr NULL
        (alloc  $\hat{sep\_alloc\_ret}$  (max  $\sigma$ size KMC) F)  $\wedge^*$ 
        zero_block ((ptr_coerce  $\hat{sep\_alloc\_ret}$ )::word32 ptr) (unat (max  $\sigma$ size KMC div 4))
    )^{\text{sep}} \wedge
 $(\hat{sep\_alloc\_ret} = \text{NULL} \longrightarrow (\text{sep\_free\_set\_h kmem\_free\_list\_addr NULL F})^{\text{sep}}) \}$ 
*/
void * sep_alloc(word_t size)
{
    word_t* prev;
    word_t* curr;
    word_t* tmp;
    word_t i;

    size = size >= 1024 ? size : 1024;

    for (prev = (word_t*) &kmem_free_list, curr = kmem_free_list;
        curr;
        prev = curr, curr = (word_t*) *curr)
        /** INV:  $\{ (\exists G H.$ 
            (sep_free_set_h kmem_free_list_addr  $\hat{curr}$  G  $\wedge^*$ 
            sep_free_set  $\hat{curr}$  NULL H) (abs  $\hat{t}_h \hat{t}_d$ )  $\wedge$ 
            ( $\hat{prev} = \text{ptr\_coerce kmem\_free\_list\_addr} \vee \text{ptr\_val } \hat{prev} \in G) \wedge$ 
            F = G  $\cup$  H)  $\wedge$ 
             $\hat{t}_h = \sigma \hat{t}_h \wedge \hat{t}_d = \sigma \hat{t}_d \wedge$ 
            sep_free_set_h kmem_free_list_addr NULL F (abs  $\hat{t}_h \hat{t}_d$ )  $\wedge$ 
             $\hat{size} = \max \sigma$ size KMC  $\wedge$ 
            KMC udvd  $\sigma$ size  $\wedge$ 

```

```

1 ≤ `size div KMC ∧
(`prev ↦ ptr_val `curr) (abs `t_h `t_d) } */
{
if (!(word_t) curr & (size - 1))
{
tmp = (word_t*) *curr;
for (i = 1; tmp && (i < (size / (1024))); i++)
/** INV: { ( `tmp ≠ NULL → no_wrap `curr `i) ∧ (
    ∃G H. (
    sep_free_set_h kmem_free_list_addr `curr G ∧*
    sep_free_set `curr `tmp (chunks `curr (ptr_val `curr + (`i - 1)*KMC)) ∧*
    sep_free_set `tmp NULL H) (abs `t_h `t_d) ∧
    (`prev = ptr_coerce kmem_free_list_addr ∨ ptr_val `prev ∈ G) ∧
    F = G ∪ (chunks `curr (ptr_val `curr + (`i - 1)*KMC)) ∪ H) ∧
    (`tmp = NULL → (
    ∃G H. (
    sep_free_set_h kmem_free_list_addr `curr G ∧*
    sep_free_set `curr NULL H) (abs `t_h `t_d) ∧
    (`prev = ptr_coerce kmem_free_list_addr ∨ ptr_val `prev ∈ G) ∧
    F = G ∪ H)) ∧
    `t_h =  $\sigma$ t_h ∧ `t_d =  $\sigma$ t_d ∧
    sep_free_set_h kmem_free_list_addr NULL F (abs `t_h `t_d) ∧
    KMC udvd  $\sigma$ size ∧
    `size = max  $\sigma$ size KMC ∧
    1 ≤ `i ∧
    `i ≤ `size div KMC ∧
    `curr ≠ NULL ∧
    (`prev ↦ ptr_val `curr) (abs `t_h `t_d) } */
}

if ((word_t) tmp != ((word_t) curr + (1024)*i))
{
tmp = 0;
break;
};
tmp = (word_t*) *tmp;
}
if (tmp)
{
*prev = (word_t) tmp;

for (i = 0; i < (size / sizeof(word_t)); i++)
/** INV: { ((
    sep_free_set_h kmem_free_list_addr NULL
    (alloc (ptr_coerce `curr) (max  $\sigma$ size KMC) F)) ∧*
    sep_cut (ptr_val `curr + `i*4) (`size - `i*4) ∧*
    zero_block `curr (unat `i)) (abs `t_h `t_d) ∧
    KMC udvd `size ∧
    KMC udvd ptr_val `curr ∧
    `i ≤ `size div 4 ∧
    ptr_val `curr ≤ ptr_val `curr + (`size - KMC) ∧
    `curr ≠ NULL ∧
    `size = max  $\sigma$ size KMC } */ {
/** AUXUPD: (ptr_safe (`curr +p `i) `t_d, ptr_tag (`curr +p `i) `t_d) */
curr[i] = 0;
}
}

```

```

        return curr;
    }
}
return 0;
}

```

## 2 Data Abstraction Predicates

```

theory kmalloc
  imports CTranslation
begin

syntax
  heap_update :: 'a ptr ⇒ 'a ⇒ heap_mem ⇒ heap_mem
  ([_ := _, _] 100)

syntax
  lift_typ_heap :: 'a typ_heap ⇒ 'a ptr ⇒ 'a option (★'(_,_'))

syntax
  lft_c :: heap_mem ⇒ heap_typ_desc ⇒ 'a ptr ⇒ 'a option (★'(_,_,_'))

translations
  lft_c h d p == lift_typ_heap (TypHeap.abs h d) p

types free_set = word32 set

constdefs KMC ≡ 1024::word32

consts
  list :: word32 typ_heap ⇒ word32 ptr ⇒ word32 ptr ⇒ word32 list ⇒ bool
primrec
  list h s e [] = (s = e)
  list h s e (p#ps) =
    (s = Ptr p ∧ s ≠ e ∧ (∃s'. h s = Some s' ∧ list h (Ptr s') e ps))

consts block :: word32 ptr ⇒ word32 ptr ⇒ heap_assert

consts
  sep_list :: word32 ptr ⇒ word32 ptr ⇒ word32 list ⇒ heap_assert
primrec
  sep_list s e [] = (λh. (s = e) ∧ □ h)
  sep_list s e (p#ps) =
    (λh. (s = Ptr p ∧ s ≠ e) ∧ (∃s'. (block s s' ∧* sep_list s' e ps) h))

consts
  ptr_tags :: nat ⇒ word32 ⇒ heap_typ_desc ⇒ heap_typ_desc
primrec
  ptr_tags 0 p d = d
  ptr_tags (Suc n) p d = ptr_tag (Ptr p::word32 ptr) (ptr_tags n (p + KMC) d)

```

```

lemma sep_list_start:
  [[ sep_list s e fs h; s ≠ e ]] ⇒ ptr_val s ∈ set fs
  by (induct fs) auto

lemma list_start:
  [[ list h s e fs; s ≠ e ]] ⇒ ptr_val s ∈ set fs
  by (induct fs) auto

lemma list_start':
  list hp s e fs ⇒ fs = [] ∨ ptr_val s = hd fs
  by (induct fs) auto

lemma list_empty [simp]:
  ∧s. list hp s s ps = (ps = [])
  by (induct ps) auto

lemma sep_list_empty [simp]:
  ∧s. sep_list s s ps = (λhp. ps = [] ∧ □ hp)
  by (induct ps) auto

defs
  block_def: block s s' ≡ λh. KMC udvd (ptr_val s) ∧ (s ↔ ptr_val s') h ∧ sep_cut (ptr_val s)
  KMC h

lemma block_sep_cut:
  block x y s ⇒ sep_cut (ptr_val x) KMC s
  by (clarsimp simp: block_def)

lemma dom_exact_sep_list [simp]:
  dom_exact (sep_list s s' ps)
  apply(subgoal_tac ∀s s' x y x' y'. sep_list x y ps s ∧ sep_list x' y' ps s' → dom s = dom s')
  apply(rule dom_exactI)
  apply fast
  apply(induct_tac ps)
  apply clarsimp
  apply(drule sep_empD)+
  apply simp
  apply clarsimp
  apply(drule sep_conjD, clarsimp)+
  apply(drule_tac x=s1 in spec)
  apply(drule_tac x=s1' in spec)
  apply(erule impE)
  apply fast
  apply clarsimp
  apply(drule block_sep_cut, clarsimp)+
  apply(drule sep_cut_dom, clarsimp)+
  done

lemma block_dom:
  block x y s ⇒ ptr_val x ∈ dom s
  apply(clarsimp simp: block_def)
  apply(drule sep_map'_dom)
  apply force
  done

lemma block_block_same [simp]:

```

```

    (block s s' ^* block s s'a) = sep_false
  apply rule
  apply clarsimp
  apply(drule sep_conjD, clarsimp simp: block_def)
  apply(drule sep_map'_dom)+
  apply(clarsimp simp: heap_disj_def)
  apply fast
  done

lemma block_sep_conj_sep_list_same [simp]:
  (block s s' ^* sep_list s s'a ps) = ( $\lambda h$ . block s s' h  $\wedge$  s=s'a  $\wedge$  ps=[])
  apply(case_tac ps)
  apply clarsimp
  apply rule
  apply rule
  apply clarsimp
  apply clarsimp
  apply clarsimp
  apply rule
  apply clarsimp
  apply(subst (asm) exists_left)
  apply(subst (asm) sep_conj_exists)
  apply clarsimp
  apply(subst (asm) sep_conj_assoc [symmetric])
  apply simp
  done

lemma sep_list_append [simp]:
   $\wedge s0$ . sep_list s0 s2 (ps1@ps2) =
    ( $\lambda hp$ .  $\exists s1$ . (sep_list s0 s1 ps1  $\wedge^*$  sep_list s1 s2 ps2) hp  $\wedge$  ptr_val s2  $\notin$  set ps1)
  apply (induct ps1)
  apply clarsimp
  apply clarsimp
  apply rule
  apply rule
  apply clarsimp
  apply(subst (asm) exists_left)
  apply(subst (asm) sep_conj_exists)
  apply clarsimp
  apply(rule_tac x=x in exI)
  apply rule
  apply clarsimp
  apply(subst (asm) sep_conj_com) back back
  apply(subst (asm) sep_conj_assoc [symmetric])
  apply(subst (asm) block_sep_conj_sep_list_same)
  apply simp
  apply(subst sep_conj_com)
  apply(subst sep_conj_exists)
  apply clarsimp
  apply rule
  apply(rule_tac x=s' in exI)
  apply force
  apply clarsimp
  apply clarsimp
  apply rule
  apply clarsimp
  apply(subst (asm) exists_left)
  apply(subst (asm) sep_conj_exists)

```

```

apply clarsimp
apply(rule_tac x=x in exI)
apply(subst exists_left)
apply(subst sep_conj_exists)
apply(rule_tac x=s1 in exI)
apply simp
done

lemma list_split [rule_format]:
   $\forall s. \text{list hp s e ps} \longrightarrow \text{distinct ps} \longrightarrow (\text{p} \in \text{set ps} \vee \text{Ptr p} = \text{e}) \longrightarrow$ 
   $(\exists \text{ps1 ps2. ps} = \text{ps1@ps2} \wedge \text{list hp s (Ptr p) ps1} \wedge \text{list hp (Ptr p) e ps2})$ 
  apply (induct ps)
  apply simp
  apply (cases Ptr p = e)
  apply simp
  apply clarsimp
  apply (erule allE, erule (1) impE)
  apply clarsimp
  apply (rule_tac x=a#ps1 in exI)
  apply (rule_tac x=ps2 in exI)
  apply simp
  apply (cases e)
  apply simp
  apply (drule list_start, fastsimp)
  apply fastsimp
  done

constdefs
  free_set :: word32 typ_heap  $\Rightarrow$  word32 ptr  $\Rightarrow$  word32 ptr  $\Rightarrow$  free_set  $\Rightarrow$  bool
  free_set hp s e F  $\equiv$   $\exists \text{ps. list hp s e ps} \wedge \text{distinct ps} \wedge F = \text{set ps}$ 

constdefs
  sep_free_set :: word32 ptr  $\Rightarrow$  word32 ptr  $\Rightarrow$  free_set  $\Rightarrow$  heap_assert
  sep_free_set s e F  $\equiv$   $(\lambda h. \exists \text{ps. sep\_list s e ps h} \wedge F = \text{set ps})$ 

lemma sep_free_set_block':
  sep_free_set (Ptr x) y {x} =  $(\lambda s. \text{block (Ptr x) y s} \wedge \text{Ptr x} \neq \text{y})$ 
  apply(clarsimp simp: sep_free_set_def)
  apply rule+
  apply clarsimp
  apply(case_tac ps)
  apply simp
  apply clarsimp
  apply(case_tac list)
  apply clarsimp
  apply clarsimp
  apply(subst (asm) exists_left)
  apply(subst (asm) sep_conj_exists)
  apply clarsimp
  apply(subst (asm) sep_conj_assoc [symmetric])
  apply(subst (asm) sep_conj_com) back
  apply(subst (asm) sep_conj_assoc [symmetric])+
  apply(subst (asm) block_block_same)
  apply simp
  apply clarsimp
  apply(rule_tac x=[x] in exI)
  apply clarsimp
  done

```

```

lemma sep_free_set_block:
  sep_free_set x y {ptr_val x} = (λs. block x y s ∧ x ≠ y)
apply (case_tac x)
apply (clarsimp simp: sep_free_set_block')
done

lemma sep_free_set_blockD:
  [[ block x y s; x ≠ y; x = Ptr z ]] ⇒ sep_free_set x y {z} s
  by (simp add: sep_free_set_block')

constdefs
  chunks :: 'a ptr ⇒ word32 ⇒ word32 set
  chunks a b ≡
  {x. ptr_val a ≤ x ∧ x ≤ b ∧ (∃n ≥ 0. uint x = uint (ptr_val a) + n * uint KMC)}

constdefs
  chunks' :: 'a ptr ⇒ word32 ⇒ word32 set
  chunks' a b ≡
  {x. ptr_val a ≤ x ∧ x < b ∧ (∃n. x = ptr_val a + n*KMC)}

constdefs
  free :: unit ptr ⇒ word32 ⇒ free_set ⇒ free_set
  free p sz F ≡ F ∪ chunks p (ptr_val p + (sz - KMC))

constdefs
  alloc :: unit ptr ⇒ word32 ⇒ free_set ⇒ free_set
  alloc p sz F ≡ F - chunks p (ptr_val p + (sz - KMC))

declare uint_number_of [simp]

lemma free_set_start:
  [[ free_set hp s e F; s ≠ e ]] ⇒ ptr_val s ∈ F
  by (clarsimp simp: free_set_def list_start)

lemma sep_free_set_start:
  [[ sep_free_set s e F hp; s ≠ e ]] ⇒ ptr_val s ∈ F
  by (clarsimp simp: sep_free_set_def sep_list_start)

lemma list_append [simp]:
  ∧s0. distinct (ps1@ps2) ⇒
    list hp s0 s2 (ps1@ps2) =
    (∃s1. list hp s0 s1 ps1 ∧ list hp s1 s2 ps2 ∧ ptr_val s2 ∉ set ps1)
  apply (induct ps1)
  apply simp
  apply (rule iffI)
  apply clarsimp
  apply (rule_tac x=s1 in exI)
  apply (cases ps2)
  apply fastsimp
  apply fastsimp
  apply auto
  done

lemma free_set_unionI:
  [[ free_set hp s t F0; free_set hp t e F1; F0 ∩ F1 = {}; ptr_val e ∉ F0 ]] ⇒
  free_set hp s e (F0 ∪ F1)

```

```

apply (unfold free_set_def)
apply clarsimp
apply (rule_tac x=ps@psa in exI)
apply fastsimp
done

lemma sep_free_set_unionI:
  [[ (sep_free_set s t F0  $\wedge$ * sep_free_set t e F1) hp; ptr_val e  $\notin$  F0 ] ]  $\implies$ 
    sep_free_set s e (F0  $\cup$  F1) hp
apply (unfold sep_free_set_def)
apply (subst (asm) sep_conj_exists)
apply clarsimp
apply (subst (asm) exists_left)
apply (subst (asm) sep_conj_exists)
apply clarsimp
apply (rule_tac x=x@xa in exI)
apply clarsimp
apply (rule_tac x=t in exI)
apply simp
done

lemma list_end:
   $\wedge$ s. list hp s e ps  $\implies$  ptr_val e  $\notin$  set ps
  by (induct ps) auto

lemma sep_list_end:
   $\wedge$ s hp. sep_list s e ps hp  $\implies$  ptr_val e  $\notin$  set ps
apply (induct ps)
apply auto
apply (drule sep_conjD, clarsimp)
apply force
done

lemma free_set_end:
  free_set hp s e F  $\implies$  ptr_val e  $\notin$  F
  by (auto simp: free_set_def list_end)

lemma sep_free_set_end:
  sep_free_set s e F hp  $\implies$  ptr_val e  $\notin$  F
  by (auto simp: sep_free_set_def sep_list_end)

lemma free_set_split_elem:
  [[ free_set hp s e F; p  $\in$  F ] ]  $\implies$ 
     $\exists$ F0 F1 p'.
  F = F0  $\cup$  F1  $\cup$  {p}  $\wedge$  F0  $\cap$  F1 = {}  $\wedge$  p  $\notin$  F0  $\wedge$  p  $\notin$  F1  $\wedge$  p  $\neq$  p'  $\wedge$ 
  hp (Ptr p) = Some p'  $\wedge$ 
  free_set hp s (Ptr p) F0  $\wedge$ 
  free_set hp (Ptr p') e F1
  apply (clarsimp simp: free_set_def)
  apply (subst (asm) in_set_conv_decomp_first)
  apply clarsimp
  apply (rule_tac x=set ys in exI)
  apply (rule_tac x=set zs in exI)
  apply clarsimp
  apply rule
  apply clarsimp
  apply (drule (1) list_start)
  apply fastsimp

```

```

apply fastsimp
done

lemma sep_free_set_split_elem:
  [[ sep_free_set s e F hp; p ∈ F ]] ==>
  ∃F0 F1 p'.
  F = F0 ∪ F1 ∪ {p} ∧ p ≠ p' ∧
  (sep_free_set s (Ptr p) F0 ∧*
   block (Ptr p) (Ptr p') ∧*
   sep_free_set (Ptr p') e F1) hp
apply (clarsimp simp: sep_free_set_def)
apply (subst (asm) in_set_conv_decomp_first)
apply clarsimp
apply(rule_tac x=set ys in exI)
apply(rule_tac x=set zs in exI)
apply clarsimp
apply(subst (asm) exists_left)
apply(subst (asm) sep_conj_exists)
apply clarsimp
apply(case_tac x)
apply clarsimp
apply(rule_tac x=word in exI)
apply rule
  apply clarsimp
  apply(subst (asm) sep_conj_com) back
  apply(subst (asm) sep_conj_assoc [symmetric])
  apply(subst (asm) block_sep_conj_sep_list_same)
  apply simp
apply(erule sep_conj_impl)
  apply simp
  apply(subst sep_conj_exists)
  apply(rule_tac x=ys in exI)
  apply clarsimp
  apply(subst exists_left)
  apply(subst sep_conj_exists)
  apply(rule_tac x=zs in exI)
  apply clarsimp
done

lemma sep_free_set_split_elem_eq:
  [[ p ∈ F; ptr_val e ≠ p; ptr_val e ∉ F ]] ==> sep_free_set s e F = (λhp. ∃F0 F1 p'. F = F0 ∪
  F1 ∪ {p} ∧ p ≠ p' ∧
  (sep_free_set s (Ptr p) F0 ∧* block (Ptr p) (Ptr p') ∧* sep_free_set (Ptr p') e F1) hp)
apply rule
apply rule
  apply(erule (1) sep_free_set_split_elem)
  apply simp
  apply(erule exE)+
  apply clarsimp
  apply(subst insert_def)
  apply(subst Un_assoc [symmetric])
  apply(subst Un_commute)
  apply(subst Un_assoc)+
  apply(rule sep_free_set_unionI)
  apply(subst (asm) sep_conj_assoc [symmetric])+
  apply(subst (asm) sep_conj_com)
  apply(subst (asm) sep_conj_assoc)+
  apply(erule sep_conj_impl)

```

```

  apply fast
  apply(rule sep_free_set_unionI)
  apply(erule sep_conj_impl)
  apply simp
  apply(rule_tac y=Ptr p' in sep_free_set_blockD)
  apply simp+
done

```

```

lemma lift_heap_update_ptr_coerce [simp]:
  lift ([ptr_coerce p :: word32 ptr] := v, h] (p :: word32 ptr ptr) = Ptr v
  apply (cases p)
  apply (simp only: lift_def heap_update_def)
  apply (simp only: h_val_def)

  apply (subgoal_tac size_of TYPE(word32 ptr) = length (to_bytes v))
  apply(subgoal_tac ptr_val ((ptr_coerce ((Ptr word)::word32 ptr ptr))::word32 ptr) = ptr_val
p)
  apply (simp only:)
  apply (subst heap_list_update)
  apply(simp add: len addr_card)
  apply (simp add: from_bytes_ptr_to_bytes_ptr)
  apply simp
  apply(simp add: len)
done

```

```

lemma lift_typ_heap_ptr_coerce_upd:
  d  $\models_t$  (p :: word32 ptr ptr)  $\implies$ 
  lift $_{\tau}^c$  (heap_update (ptr_coerce p :: word32 ptr) v h, d) =
  (lift $_{\tau}^c$  (h, d) (p  $\mapsto$  Ptr v) :: word32 ptr typ_heap)
  apply (rule ext)
  apply (unfold lift $_{\tau}$ _if)
  apply auto
  apply(simp only: heap_update_def h_val_def)
  apply(subgoal_tac ptr_val ((ptr_coerce p)::word32 ptr) = ptr_val p)
  apply(simp only:)
  apply(subgoal_tac size_of TYPE(word32 ptr) = size_of TYPE(word32))
  apply(simp only:)
  apply(subst heap_list_update_to_bytes)
  apply(simp add: to_bytes_word from_bytes_ptr length_word_rsplitt_exp' word_rcat_rsplitt flen_def)
  apply simp
  apply simp
  apply (simp add: lift_def h_val_def heap_update_def)
  apply (subst heap_list_update_disjoint_same)
  apply(drule (1) h_t_valid_neq_disjoint)
  apply simp
  apply(simp add: len)
  apply (simp add: typ_tag_ptr typ_size_def)
done

```

```

lemma lift_typ_heap $_c$ _ptr_coerce_same:
  d  $\models_t$  (p :: word32 ptr ptr)  $\implies$ 
  lift $_{\tau}^c$  (heap_update (ptr_coerce p :: word32 ptr) v h, d) =
  (lift $_{\tau}^c$  (h, d) :: word32 typ_heap)
  apply (rule ext)
  apply (unfold lift $_{\tau}$ _if)

```

```

apply clarsimp
apply(simp add: h_val_def heap_update_def)
apply(subst heap_list_update_disjoint_same)
  apply(frule (1) h_t_valid_neq_disjoint) back
  apply(clarsimp simp: h_t_valid_def valid_footprint_def typ_tag_ptr typ_tag_word)
  apply(simp add: len)
apply simp
done

lemma chunks_empty [simp]:
  chunks a (ptr_val a) = {ptr_val a}
  by (force simp: chunks_def word_less_def)

lemma free_set_empty [simp]:
  free_set h s s F = (F = {})
  by (simp add: free_set_def)

lemma sep_free_set_empty [simp]:
  sep_free_set s s F = ( $\lambda h. \square h \wedge F = \{\}$ )
  by (simp add: sep_free_set_def)

declare max_ac [simp]
lemmas strip = conjE exE

lemma lift_typ_heap_Some_the:
  d  $\models_t$  (p::word32 ptr)  $\implies$  Some (the (lift $_{\tau^c}$  (h,d) p)) = (lift $_{\tau^c}$  (h,d) p)
  by (clarsimp simp: lift $_{\tau}$ _if h_val_def from_bytes_word)

lemmas ktyp_simps = typ_simps lift_typ_heap_Some_the

lemmas typ_simps = typ_simps lift_heap_update_same lift_heap_update_same_type
  typ_tag_word typ_tag_ptr lift_typ_heap_ptr_coerce_upd
  lift_typ_heap_c_ptr_coerce_same

lemma Un_Int_empty [simp]:
  ((A  $\cup$  B)  $\cap$  C = {}) = (A  $\cap$  C = {}  $\wedge$  B  $\cap$  C = {})
  by blast

declare uint_ge_0 [simp del]

lemma KMC_neq_0 [simp]: KMC  $\neq$  0 by (simp add: KMC_def)
lemma KMC_gr_0 [simp]: 0 < KMC by (simp add: KMC_def word_less_alt)
lemma KMC_ge_0 [simp]: 0  $\leq$  KMC by (simp add: KMC_def word_le_def)

lemma chunks_add:
  [ ptr_val a  $\leq$  p; p < p + KMC;
    ( $\exists n \geq 0. \text{uint } p = \text{uint } (\text{ptr\_val } a) + n * \text{uint } KMC$ ) ]  $\implies$ 
  chunks a (p + KMC) = chunks a p  $\cup$  {p + KMC}
  apply (simp add: chunks_def)
  apply (rule set_ext)
  apply (rule iffI)
  apply (clarsimp simp add: linorder_not_less NULL_ptr_val KMC_def)
  apply (simp add: word_le_def word_less_alt max_def split: split_if_asm)
  apply (simp add: uint_plus_if split: split_if_asm)
  apply (rule word_uint_Rep_eqD)
  apply(subst uint_plus_if)
  apply (simp add: flen_def)

```

```

  apply (rule word_uint_Rep_eqD)
  apply clarsimp
  apply (unfold flen_def)
  apply clarsimp
apply clarsimp
apply (erule disjE)
  apply clarsimp
  apply rule
  apply simp
  apply(rule_tac x=n+1 in exI)
  apply(clarsimp simp: uint_plus_if split: split_if_asm)
  apply rule
  apply(simp add: zadd_zmult_distrib)
  apply clarsimp
  apply(simp add: KMC_def)
  apply (simp add: word_le_def word_less_alt max_def split: split_if_asm)
  apply(clarsimp simp: uint_plus_if split: split_if_asm)
  apply(simp add: flen_def)
apply clarsimp
apply (simp add: KMC_def)
done

lemma chunks_NULL [simp]:
  a ≠ NULL  $\implies$  0  $\notin$  chunks a b
  by (case_tac a) (simp add: chunks_def)

lemma in_chunks2:
  ptr_val a < ptr_val a + b  $\implies$  ptr_val a  $\in$  chunks a (ptr_val a + b)
  by (fastsimp simp: chunks_def)

lemma udvd_expand:
   $\llbracket K \text{ udvd } (p-a); a \leq p \rrbracket \implies \exists n \geq 0. \text{ uint } p = \text{ uint } a + n * \text{ uint } K$ 
  by (fastsimp simp add: udvd_def uint_sub_if word_le_def)

lemma udvdK:
   $\llbracket (p::'a::fl \text{ word}) < a + s; a \leq a + s; K \text{ udvd } s; K \text{ udvd } (p - a); a \leq p; 0 < K \rrbracket \implies p + K \leq a + s$ 
  by (fast dest: udvd_incr2_K)

lemma udvd_plus':
   $\llbracket K \text{ udvd } p - a; a \leq p; p \leq p+K \rrbracket \implies K \text{ udvd } p + K - a$ 
  apply (clarsimp simp add: udvd_def)
  apply (rule_tac x=n+1 in exI)
  apply clarsimp
  apply (simp add: uint_sub_if word_le_def)
  apply (fold word_le_def)
  apply (simp add: uint_plus_simple)
  apply (simp add: ring_distrib)
  done

lemma udvd_incr2:
   $\llbracket (p::'a::fl \text{ word}) < a + s; a \leq a + s; K \text{ udvd } s; K \text{ udvd } (p - a); a \leq p; 0 < K \rrbracket \implies p \leq p + K$ 
  by (fast dest: udvd_incr2_K)

lemma chunks_add_udvd:
   $\llbracket \text{ ptr\_val } a \leq p; p < p + \text{ KMC}; \text{ KMC udvd } (p - \text{ ptr\_val } a) \rrbracket \implies$ 
  chunks a (p + KMC) = chunks a p  $\cup$  {p + KMC}

```

```

by (drule (1) udvd_expand) (rule chunks_add)

constdefs
  align :: word32 set  $\Rightarrow$  word32 set
  align F  $\equiv$  F  $\cap$  {p. KMC udvd p}

constdefs
  aligned :: word32 set  $\Rightarrow$  bool
  aligned P  $\equiv$  align P = P

lemma align_elem:
  (p  $\in$  align P) = (p  $\in$  P  $\wedge$  KMC udvd p)
  by (simp add: align_def)

lemma alignedD:
   $\llbracket$  p  $\in$  P; aligned P  $\rrbracket \Longrightarrow$  KMC udvd p
  by (auto simp: aligned_def align_elem)

lemma KMC_max:
  KMC udvd p  $\Longrightarrow$  KMC udvd max p KMC
  apply (simp add: max_def)
  apply (clarsimp simp: udvd_def KMC_def)
  done

declare word_neq_0_conv [simp]

lemma chunks_add_udvd2:
   $\llbracket$  ptr_val a  $\leq$  p; KMC  $\leq$  p; KMC udvd p - ptr_val a  $\rrbracket \Longrightarrow$ 
  chunks a p = chunks a (p-KMC)  $\cup$  {p}
  apply (simp add: chunks_def udvd_def)
  apply clarsimp
  apply (rule set_ext)
  apply (rule iffI)
  apply (clarsimp simp: linorder_not_le)
  apply (simp add: word_le_def uint_sub_if word_less_alt)
  apply (rule word_uint_Rep_eqD)
  apply clarsimp
  apply (subgoal_tac uint p = n * uint KMC + uint (ptr_val a))
  prefer 2
  apply simp
  apply clarsimp
  apply (subgoal_tac (n - 1) * uint KMC < na * uint KMC)
  prefer 2
  apply (simp add: left_diff_distrib)
  apply (simp add: mult_less_cancel_right)
  apply (simp add: mult_le_cancel_right)
  apply (erule impE)
  apply (simp (no_asm) add: KMC_def)
  apply arith
  apply clarsimp
  apply (subgoal_tac uint p = n * uint KMC + uint (ptr_val a))
  prefer 2
  apply (simp add: word_le_def uint_sub_if)
  apply (erule disjE)
  apply (clarsimp simp add: KMC_def)
  apply clarsimp
  apply (simp add: word_le_def word_less_alt split: split_if_asm)
  apply (simp add: mult_le_cancel_right)

```

```

apply (clarsimp simp add: uint_sub_if)
apply (subgoal_tac na * uint KMC ≤ (n - 1) * uint KMC)
  prefer 2
  apply (simp add: left_diff_distrib)
apply (simp add: mult_le_cancel_right)
done

lemma chunks_empty:
  assumes k ≤ ptr_val a 0 < k
  shows chunks a (ptr_val a - k) = {}
proof -
  from prems have ptr_val a - k < ptr_val a
    by (simp add: word_less_alt word_le_def uint_sub_if)
  thus ?thesis by (auto simp: chunks_def)
qed

lemma chunks_end_nowrap [simp]:
  [[ k ≤ p; 0 < k ]] ⇒ p ∉ chunks a (p - k)
  by (clarsimp simp: chunks_def linorder_not_less [symmetric]
      word_less_alt uint_sub_if)

lemma KMC_udvd_0 [simp]: KMC udvd 0
  by (fastsimp simp add: udvd_def)

lemma udvd_minus:
  [[ k udvd x; k ≤ x ]] ⇒ k udvd x - k
  apply (clarsimp simp add: udvd_def)
  apply (case_tac n=0)
  apply (fastsimp simp add: uint_0_iff)
  apply (rule_tac x=n - 1 in exI)
  apply (simp add: word_le_def uint_sub_if int_distrib)
  done

lemma max_size_KMC:
  1 ≤ (max p KMC) div KMC
apply(auto simp: max_def word_le_nat_alt)
apply(subst word_arith_nat_defs)
apply(subst word_unat_eq_norm)
apply clarsimp
apply(subst mod_if)
apply clarsimp
apply rule
  apply clarsimp
  apply(subst div_if)
  apply(simp add: KMC_def)
  apply simp
apply clarsimp
apply(erule notE)
apply(subgoal_tac unat p div unat KMC < unat p)
  apply(insert unat_lt2p [of p])
  apply(simp )
  apply(rule div_less_dividend)
  apply(simp add: KMC_def)
  apply(simp add: KMC_def)
apply(subst word_arith_nat_defs)
apply(subst word_unat_eq_norm)
apply clarsimp
done

```

```

lemma chunks_start:
  ptr_val a ≤ b ⇒ ptr_val a ∈ chunks a b
  by (force simp: chunks_def)

lemma unroll_KMC:
  ptr_val curr + i * KMC = ptr_val curr + (i - 1) * KMC + KMC
  apply simp
  apply(subst left_diff_distrib)
  apply(simp add: mult_ac)
  done

lemma dvd_alignD:
  [| k dvd unat b; k dvd (unat ((b::'a::fl word) + of_nat c)); k dvd 2 ^ fl_of TYPE('a) |] ⇒ k
  dvd c
  apply(subst (asm) word_unat_Rep_inverse [symmetric]) back
  apply(subst (asm) Abs_fnat_homs)
  apply(subst (asm) word_unat_eq_norm)
  apply(drule dvd_mod_imp_dvd)
  apply assumption
  apply(thin_tac ?x) back
  apply(drule (1) dvd_diff) back
  apply clarsimp
  done

lemma h_val_update_id'_align:
  [| ptr_aligned (p::'a ptr); ptr_aligned (q::'b::mem_type ptr); align_of TYPE('a::mem_type) = align_of
  TYPE('b::mem_type); align_of TYPE('a) = size_of TYPE('a); align_of TYPE('b) = size_of TYPE('b);
  ptr_val p ≠ ptr_val q |] ⇒ liftτc (heap_update p v h,d) q = liftτc (h,d) q
  apply(clarsimp simp: liftτ_if_split: option.splits)
  apply(simp add: h_val_def)
  apply(rule_tac f=from_bytes in arg_cong)
  apply(simp add: heap_update_def)
  apply(rule heap_list_update_disjoint_same)
  apply(simp add: len)
  apply(rule ccontr)
  apply(drule intvl_inter)
  apply(simp add: intvl_def)
  apply(erule disjE)
  apply(clarsimp simp: ptr_aligned_def)
  apply(drule (1) dvd_alignD)
  apply(drule sym) back
  apply(simp only:)
  apply(subst fl_of32)
  apply(subst addr_card [symmetric])
  apply(rule align)
  apply(clarsimp simp: dvd_def)
  apply(clarsimp simp: ptr_aligned_def)
  apply(drule (1) dvd_alignD)
  apply(drule sym) back
  apply(simp only:)
  apply(subst fl_of32)
  apply(subst addr_card [symmetric])
  apply(rule align)
  apply(clarsimp simp: dvd_def)
  done

lemma udvd_dvd:

```

```

x udvd y  $\implies$  (unat x dvd unat y)
by (simp add: udvd_iff_dvd)

lemma dvd_udvd:
  (unat x dvd unat y)  $\implies$  x udvd y
  by (simp add: udvd_iff_dvd)

lemma ptr_aligned:
  [[ aligned F; x  $\in$  F; size_of TYPE('a::c_type) dvd unat KMC ]]  $\implies$ 
    ptr_aligned ((Ptr x)::'a ptr)
  apply (clarsimp simp: aligned_def align_def)
  apply (subgoal_tac x  $\in$  {p. KMC udvd p})
  apply (clarsimp simp: ptr_aligned_def)
  apply (simp add: KMC_def)
  apply (subgoal_tac (1024::nat) dvd 232)
  apply (rule dvd_trans)
  apply (rule align_size_of)
  apply (erule dvd_trans)
  apply (drule udvd_dvd)
  apply simp
  apply simp
  apply fast
done

lemma ptr_aligned_KMC:
  KMC udvd ptr_val p  $\implies$  ptr_aligned (p::word32 ptr)
  apply (unfold ptr_aligned_def)
  apply (case_tac p)
  apply (clarsimp simp: align_of_def typ_info_word aalign_def)
  apply (drule udvd_dvd)
  apply (rule_tac n=unat KMC in dvd_trans)
  apply (simp add: KMC_def)
  apply simp
done

lemma udvd_4:
  4 udvd y * (4::word32)
  apply (rule dvd_udvd)
  apply (subst word_arith_nat_defs)
  apply (subst word_unat_eq_norm)
  apply (rule dvd_mod)
  apply simp
  apply simp
done

lemma h_val_update_id'_align_F:
  [[ ptr_val p  $\notin$  F; ptr_aligned p; aligned F; size_of TYPE('a) dvd unat KMC;
    align_of TYPE('a) = align_of TYPE('b::c_type); align_of TYPE('a) = size_of TYPE('a);
    align_of TYPE('b) = size_of TYPE('b) ]]  $\implies$ 
    lift $\tau$ c (heap_update p (v::'a::mem_type) h,d) |(Ptr 'F) = (lift $\tau$ c (h,d) |(Ptr 'F)::'a::mem_type
  typ_heap)
  apply (rule ext)
  apply (case_tac x  $\in$  Ptr ' F)
  apply clarsimp
  apply (rule h_val_update_id'_align, assumption)
  apply (erule (1) ptr_aligned)
  apply simp+

```

```

apply clarsimp
apply clarsimp
done

```

```

lemma KMC_4 [simp]:
  4 dvd unat KMC
  by (simp add: KMC_def)

```

```

lemma aligned_sub:
  [[ aligned F; G  $\subseteq$  F ]]  $\implies$  aligned G
  by (force simp: aligned_def align_def)

```

```

lemma aligned_chunks:
  [[ ptr_val (p::word32 ptr)  $\leq$  ptr_val p + (k - KMC); KMC udvd i * word_of_int 4; KMC udvd k; KMC
 $\leq$  k; i < k div word_of_int 4 ]]
   $\implies$  ptr_val (p +p i)  $\in$  chunks p (ptr_val p + (k - KMC))

```

```

apply(subst chunks_def)
apply clarsimp
apply rule
  apply(frule div_lt_mult)
  apply(simp add: word_less_def word_le_def)
  apply(simp add: no_plus_overflow_uint)
  apply(drule (2) udvd_minus_le')
  apply(drule div_lt_uint'')
  apply simp
  apply(rule_tac y=uint (ptr_val p) + uint (k - KMC) in order_le_less_trans)
  apply simp
  apply(subst (asm) word_le_def) back
  apply(simp add: mult_ac)
  apply simp
  apply rule
  apply(frule div_lt_mult)
  apply(simp add: word_less_def word_le_def)
  apply(drule (2) udvd_minus_le')
  apply(drule div_lt_uint'')
  apply simp
  apply(simp add: mult_ac)
  apply(drule word_le_exists') back back
  apply clarsimp
  apply(subst add_assoc [symmetric])
  apply(simp add: no_plus_overflow_uint)
  apply(simp add: flen_def)
  apply(subst (asm) uint_plus_if)
  apply(clarsimp split: split_if_asm)
  apply(subst (asm) add_assoc [symmetric])
  apply(subst uint_plus_if)
  apply(clarsimp split: split_if_asm)
  apply(erule notE)
  apply(simp add: flen_def)
  apply(subgoal_tac 0  $\leq$  uint z)
  apply simp
  apply(rule uint_ge_0)
  apply(erule notE)
  apply(simp add: flen_def)
  apply(rule udvd_expand)
  apply simp
  apply(clarsimp simp: udvd_def)
  apply(rule_tac x=n in exI)

```

```

apply rule
  apply simp
  apply(simp add: mult_ac)
apply(frulc div_lt_mult, simp add: word_le_def word_less_def)
apply(simp add: no_plus_overflow_uint)
apply(drulc (2) udvd_minus_le')
  apply(drulc div_lt_uint'')
  apply simp
apply(rule_tac y=uint (ptr_val p) + uint (k - KMC) in order_le_less_trans)
  apply simp
  apply(subst (asm) word_le_def) back
  apply(simp add: mult_ac)
apply simp
done

lemma KMC_udvd_plus:
  [[ KMC udvd ptr_val (p +p i); KMC udvd ptr_val (p::word32 ptr) ]] ==> KMC udvd i*word_of_int 4
apply(case_tac p)
apply simp
apply(drulc udvd_dvd)+
apply(rule dvd_udvd)
apply(erule dvd_alignD) back
  apply simp
apply(simp add: flen_def KMC_def)
done

lemma alignedD2:
  [[ aligned P; p ∈ P ]] ==> KMC udvd p
  by (auto simp: aligned_def align_elem)

lemma lt_max_KMC:
  0 < unat (max KMC p)
  by (simp add: max_def word_le_nat_alt KMC_def)

lemma max_KMC_div_times:
  KMC udvd p ==> KMC * (max p KMC div KMC) = max p KMC
apply(drulc udvd_dvd)
apply(subst word_arith_nat_defs)
apply(subst max_lt)
apply(subst dvd_mult_div_cancel)
  apply simp
  apply(simp add: max_def)
apply simp
done

lemma udvd_le:
  a udvd b ==> a ≤ b ∨ b=0
apply(clarsimp simp: udvd_def)
apply(subst (asm) word_less_def)
apply(subst word_le_def)
apply(subst (asm) word_le_def)
apply(subst (asm) word_uint_Rep_inject [symmetric])
applyclarsimp
apply(subgoal_tac 1 * uint a ≤ n * uint a)
  apply simp
  apply(rule mult_mono)
  apply simp+
  apply(rule uint_ge_0)

```

done

lemma index\_not\_NULL:

$\llbracket i < k \text{ div } 4; \text{ptr\_val } x \leq \text{ptr\_val } x + (k - \text{KMC}); \text{KMC udvd } k; \text{KMC udvd ptr\_val } x; x \neq \text{NULL} \rrbracket$   
 $\implies$

$((x::\text{word32 ptr}) +_p i) \neq \text{NULL}$   
apply(frule udvd\_le)  
apply(erule disjE)  
prefer 2  
apply(clarsimp simp: word\_less\_nat\_alt unat\_div)  
apply(case\_tac x, clarsimp)  
apply(simp add: no\_plus\_overflow\_unat)  
apply(subst word\_less\_nat\_alt)  
apply simp  
apply(subst word\_arith\_nat\_defs)  
apply(subst word\_unat\_eq\_norm)  
apply(subst mod\_less)  
prefer 2  
apply(simp add: word\_less\_nat\_alt)  
apply(thin\_tac 0 < word)  
apply(drule udvd\_dvd)+  
apply(clarsimp simp: dvd\_def)  
apply(drule div\_lt\_mult)  
apply(simp add: word\_less\_nat\_alt)  
apply(subst (asm) word\_less\_nat\_alt)  
apply simp  
apply(subst (asm) unat\_sub, simp)  
apply simp  
apply(rule\_tac j=unat KMC \* kaa + unat KMC \* ka in less\_le\_trans)  
apply simp  
apply(subgoal\_tac unat KMC \* ka - unat KMC = unat KMC \* (ka - 1))  
apply(subgoal\_tac  $\exists z. 2^{\text{flen word}} = \text{unat KMC} * z$ )  
apply clarsimp  
apply(subgoal\_tac unat KMC \* (kaa + (ka - Suc 0)) < unat KMC \* z)  
apply clarsimp  
apply(subst (asm) diff\_add\_assoc [symmetric])  
apply(simp add: word\_le\_nat\_alt)  
apply(subgoal\_tac kaa + ka  $\leq$  z)  
apply(drule\_tac k=unat KMC in mult\_le\_mono2)  
apply(simp add: flen\_def)  
apply(subst (asm) add\_mult\_distrib2, simp)  
apply arith  
apply(subst add\_mult\_distrib2, simp)  
apply(rule\_tac x=2<sup>flen word</sup> div unat KMC in exI)  
apply simp  
apply(subst mult\_div\_cancel)  
apply(clarsimp simp: flen\_def KMC\_def)  
apply(subst diff\_mult\_distrib2)  
apply simp  
done

lemma udvd\_KMC:

KMC udvd y \* KMC  
apply(rule dvd\_udvd)  
apply(subst word\_arith\_nat\_defs)  
apply(subst word\_unat\_eq\_norm)  
apply(rule dvd\_mod)  
apply simp

```

apply(simp add: KMC_def)
done

lemma chunks_ptr_coerce [simp]:
  chunks (ptr_coerce p) q = chunks p q
  by (clarsimp simp: chunks_def)

lemma chunks_end_KMC:
  ptr_val p ≤ ptr_val p + i*KMC ⇒ ptr_val p + i*KMC ∈ chunks p (ptr_val p + i*KMC)
apply(clarsimp simp: chunks_def)
apply(rule udvd_expand)
  apply clarsimp
  apply(rule udvd_KMC)
apply simp
done

lemma lt_add_KMC:
  [ Ptr (p + KMC) ≠ NULL; KMC udvd p ] ⇒ p < p + KMC
apply(subst word_less_def)
apply rule
  apply(subst word_le_def)
  apply(simp add: udvd_def)
  apply clarsimp
  apply(simp add: uint_plus_if_split: split_if_asm)
  apply rule
    apply clarsimp
    apply clarsimp
    apply(erule notE)

apply(subgoal_tac n * uint KMC + uint KMC = uint KMC * n + uint KMC * 1)
  prefer 2 apply simp
apply(simp only:)
apply(subst ring_distrib [symmetric])
apply clarsimp
apply(subgoal_tac ∃k. 2^flen p = uint KMC * k)
  apply clarsimp
  apply(subgoal_tac n < k)
    apply(rule_tac k=uint KMC in zmult_zless_mono2)
    apply(drule zless_imp_add1_zle)
    apply(subst (asm) order_le_less) back
    apply clarsimp
    apply(subst (asm) word_less_def)
    apply(subst (asm) word_le_def)
    apply(subst (asm) word_uint_Rep_inject [symmetric])
    apply(simp add: uint_plus_if_split: split_if_asm)
    apply clarsimp
    apply(simp add: mult_ac ring_distrib)
    apply clarsimp
    apply(simp add: mult_ac ring_distrib)
    apply(simp add: KMC_def)
    apply(insert uint_lt2p [of p])
    apply(simp add: flen_def)
    apply(simp add: mult_ac KMC_def)
    apply(rule_tac x=2^flen p div uint KMC in exI)
    apply(simp add: flen_def KMC_def)
apply clarsimp
done

```

```

lemma max_KMC_div_times':
  [ [ KMC udvd p; k udvd KMC ]  $\implies$  (p div k) * k = p ]
  apply (drule udvd_dvd)
  apply (subst word_arith_nat_defs)
  apply (subst unat_div)
  apply (subst nat_mult_commute)
  apply (subst dvd_mult_div_cancel)
  apply (drule udvd_dvd)
  apply (erule (1) dvd_trans)
  apply (drule udvd_dvd)
  apply simp
done

lemma KMC_dvd_4 [simp]:
  4 udvd KMC
  by (clarsimp simp: udvd_def KMC_def)

lemma intvl_chunks:
  [ [ y  $\in$  {ptr_val x..+unat k}; ptr_val x  $\leq$  ptr_val x + (k - KMC); KMC udvd k ]  $\implies$ 
     $\exists m n. m \in \text{chunks } x \text{ (ptr\_val } x + (k - \text{KMC})) \wedge n < \text{KMC} \wedge y = m + n$  ]
  apply (frule udvd_le)
  apply (erule disjE)
  prefer 2
  apply (drule intvlD, clarsimp)
  apply (drule intvlD, clarsimp simp: chunks_def)
  apply (rule_tac x=ptr_val x + (of_nat ka div KMC) * KMC in exI)
  apply rule
  apply (subst no_plus_overflow_unat)
  apply (subst (asm) no_plus_overflow_unat)
  apply (rule le_less_trans)
  prefer 2
  apply fast
  apply simp
  apply (subst word_arith_nat_defs)
  apply (subst word_unat_eq_norm)
  apply (subst mod_less)
  apply (subst word_arith_nat_defs)
  apply (subst word_unat_eq_norm)
  apply (subst mod_less)
  apply (rule le_less_trans)
  apply (rule div_le_dividend)
  apply (rule unat_lt2p)
  apply (rule le_less_trans)
  apply (rule div_mult_le)
  apply (rule unat_lt2p)
  apply (subst word_arith_nat_defs)
  apply (subst word_unat_eq_norm)
  apply (subst mod_less)
  apply (rule le_less_trans)
  apply (rule div_le_dividend)
  apply (rule unat_lt2p)
  apply (subst unat_sub, simp)
  apply (subst nat_mult_commute)
  apply (subst mult_div_cancel)
  apply (rule diff_mod_le)
  apply (subst word_unat_eq_norm)
  apply (subst mod_less)

```

```

    apply(erule less_trans)
    apply(rule unat_lt2p)
    apply simp
    apply(erule udvd_dvd)
  apply rule
    apply(subst word_le_nat_alt)
    apply(subst word_arith_nat_defs)
    apply(subst word_unat_eq_norm)
    apply(subgoal_tac unat (of_nat ka div KMC * KMC) = ka - ka mod unat KMC)
    prefer 2
    apply(subst word_arith_nat_defs)
    apply(subst word_unat_eq_norm)
    apply(subst mod_less)
    apply(subst word_arith_nat_defs)
    apply(subst word_unat_eq_norm)
    apply(subst mod_less)
    apply(rule le_less_trans)
    apply(rule div_le_dividend)
    apply(rule unat_lt2p)
    apply(rule le_less_trans)
    apply(rule div_mult_le)
    apply(rule unat_lt2p)
    apply(subst word_arith_nat_defs)
    apply(subst word_unat_eq_norm)
    apply(subst mod_less)
    apply(rule le_less_trans)
    apply(rule div_le_dividend)
    apply(rule unat_lt2p)
    apply(subst word_unat_eq_norm)
    apply(subst mod_less)
    apply(erule less_trans)
    apply(rule unat_lt2p)
    apply(subgoal_tac ka = ka div unat KMC * unat KMC + ka mod unat KMC)
    apply arith
    apply simp
  apply simp
  apply(subst mod_less) back
    apply(subst (asm) no_plus_overflow_unat)
    apply(rule le_less_trans)
    prefer 2
    apply(simp add: flen_def)
    apply simp
    apply(subst unat_sub, simp)
    apply(erule diff_mod_le)
    apply(erule udvd_dvd)
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_eq_norm)
  apply(subst mod_less) back
    apply(subst (asm) no_plus_overflow_unat)
    apply(rule le_less_trans)
    prefer 2
    apply(simp add: flen_def)
    apply simp
    apply(subst unat_sub, simp+)+
  apply(erule diff_mod_le)
  apply(erule udvd_dvd)
  apply rule
  apply(rule udvd_expand)

```

```

apply clarsimp
apply(rule dvd_udvd)
apply(subst word_arith_nat_defs)
apply(subst word_unat_eq_norm)
apply(subst mod_less)
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_eq_norm)
  apply(subst mod_less)
  apply(rule le_less_trans)
  apply(rule div_le_dividend)
  apply(rule unat_lt2p)
  apply(rule le_less_trans)
  apply(rule div_mult_le)
  apply(rule unat_lt2p)
apply simp
apply(subst (asm) no_plus_overflow_unat)
apply(subst no_plus_overflow_unat)
apply(subgoal_tac unat (of_nat ka div KMC * KMC) ≤ unat (k - KMC))
  apply arith
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_eq_norm)
  apply(subst mod_less)
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_eq_norm)
  apply(subst mod_less)
  apply(rule le_less_trans)
  apply(rule div_le_dividend)
  apply(rule unat_lt2p)
  apply(rule le_less_trans)
  apply(rule div_mult_le)
  apply(rule unat_lt2p)
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_eq_norm)
  apply(subst mod_less)
  apply(rule le_less_trans)
  apply(rule div_le_dividend)
  apply(rule unat_lt2p)
  apply(subst word_unat_eq_norm)
  apply(subst mod_less)
  apply(erule less_trans)
  apply(rule unat_lt2p)
  apply(subst nat_mult_commute)
  apply(subst mult_div_cancel)
  apply(subst unat_sub, simp)
  apply(erule diff_mod_le)
  apply(erule udvd_dvd)
apply(rule_tac x=of_nat ka mod KMC in exI)
apply rule
  apply(subst word_less_nat_alt)
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_eq_norm)
  apply(subst mod_less) back
  apply(rule less_trans)
  apply(rule mod_less_divisor)
  apply(simp add: KMC_def)
  apply(rule unat_lt2p)
  apply(rule mod_less_divisor)
  apply(simp add: KMC_def)

```

```

apply clarsimp
apply(subst word_mod_div_equality)
  apply simp+
done

```

```

lemma udvd_trans:
  [ a udvd b; b udvd c ]  $\implies$  a udvd c
apply(unfold udvd_def)
apply(erule exE)+
apply(rule_tac x=na*n in exI)
apply clarsimp
apply(rule mult_nonneg_nonneg)
  apply simp+
done

```

```

lemma lift_ptr_coerce:
  [ lift h ((ptr_coerce p)::word32 ptr) = Ptr (v::word32) ]  $\implies$  lift h (p::word32 ptr) = v
apply(auto simp: lift_def h_val_def)
apply(simp add: from_bytes_ptr from_bytes_word)
done

```

```

lemma lift_ptr_coerce':
  [ lift h ((ptr_coerce p)::word32 ptr) = ptr_val (v::word32 ptr) ]  $\implies$  lift h (p::word32 ptr) = v
apply(auto simp: lift_def h_val_def)
apply(simp add: from_bytes_ptr from_bytes_word)
done

```

end

### 3 Shared Lemmas for Non-Separation Case

```

theory non_sep_common imports kmalloc begin

```

```

lemma list_restrict:
   $\forall s S hp. \text{Ptr}'\text{set } ps \subseteq S \longrightarrow \text{list } (hp|'S) s e ps = \text{list } hp s e ps$ 
  by (induct ps) auto

```

```

lemma list_restrict_dom:
   $\text{list } (hp|'(\text{Ptr}'\text{set } ps)) s e ps = \text{list } hp s e ps$ 
  by (simp add: list_restrict)

```

```

lemma list_neq_None:
   $\bigwedge s. [ \text{list } h s e ps; \text{ptr\_val } p \in \text{set } ps ] \implies h p \neq \text{None}$ 
  by (induct ps) auto

```

```

lemma free_set_neq_None:
  [ free_set hp s e F; ptr_val p  $\in$  F ]  $\implies$  hp p  $\neq$  None
  by (auto simp add: free_set_def list_neq_None iff del: not_None_eq)

```

```

lemma free_set_restrict_dom:
   $\text{free\_set } (h|'(\text{Ptr}'F)) s e F = \text{free\_set } h s e F$ 
  by (force simp add: free_set_def list_restrict_dom)

```

```

lemma free_set_split:
  [[ free_set hp s e F; p ∈ F ∨ Ptr p = e ]] ⇒
  (∃F0 F1. F = F0 ∪ F1 ∧ F0 ∩ F1 = {} ∧ free_set hp s (Ptr p) F0 ∧ free_set hp (Ptr p) e F1)
  apply (unfold free_set_def)
  apply clarsimp
  apply (drule (2) list_split)
  apply clarsimp
  apply (rule exI)
  apply (rule exI)
  apply (rule conjI)
  apply (rule refl)
  apply fastsimp
  done

lemma free_set_end':
  [[ free_set hp s e F; p ∈ F ]] ⇒ p ≠ ptr_val e
  by (force simp: free_set_end)

lemma free_set_split':
  [[ free_set hp s e F; p ∈ F ]] ⇒
  (∃F0 F1. F = F0 ∪ F1 ∧ p ∈ F1 ∧ F0 ∩ F1 = {} ∧ free_set hp s (Ptr p) F0 ∧ free_set hp (Ptr
p) e F1)
  apply(frult (1) free_set_end')
  apply (unfold free_set_def)
  apply clarsimp
  apply(drult (1) list_split)
  apply fast
  apply clarsimp
  apply (rule exI)
  apply (rule exI)
  apply (rule conjI)
  apply (rule refl)
  apply rule
  apply clarsimp
  apply(force dest: list_start)
  apply fastsimp
  done

lemma list_cons [rule_format]:
  ∀s s'. list hp s e ps → hp s = Some s' → s ≠ e
  →
  (∃ps'. ps = ptr_val s#ps' ∧ list hp (Ptr s') e ps')
  by (induct ps) auto

lemma list_next:
  ∧s p. [[ list hp s e ps; p ∈ set ps ]] ⇒
  ∃p'. hp (Ptr p) = Some p' ∧ (p' ∈ set ps ∨ p' = ptr_val e)
  by (induct ps) (auto dest: list_start)

lemma free_set_next:
  [[ free_set hp s e F; p ∈ F ]] ⇒
  (∃p'. hp (Ptr p) = Some p' ∧ (p' ∈ F ∨ p' = ptr_val e))
  by (auto simp: free_set_def dest: list_next)

lemma free_set_heap_update [simp]:
  assumes F: ptr_val p ∉ F

```

```

shows free_set (hp(p := v)) s e F = free_set hp s e F
proof -
  from F
  have hp(p := v)|'(Ptr'F) = hp|'(Ptr'F) (is ?h = ?h')
    by (cases p) auto
  hence free_set ?h s e F = free_set ?h' s e F by simp
  thus ?thesis by (simp add: free_set_restrict_dom del: restrict_fun_upd)
qed

```

```

lemma list_next:
 $\bigwedge s p. \llbracket \text{list hp } s \text{ e } ps; p \in \text{set } ps \rrbracket \implies$ 
 $\exists p'. \text{hp (Ptr } p) = \text{Some } p' \wedge (p' \in \text{set } ps \vee p' = \text{ptr\_val } e)$ 
by (induct ps) (auto dest: list_start)

```

```

lemma free_set_next:
 $\llbracket \text{free\_set hp } s \text{ e } F; p \in F \rrbracket \implies$ 
 $(\exists p'. \text{hp (Ptr } p) = \text{Some } p' \wedge (p' \in F \vee p' = \text{ptr\_val } e))$ 
by (auto simp: free_set_def dest: list_next)

```

```

lemma list_singleton:
 $s \neq e \implies \text{list (h(s := Some (ptr\_val } e)) s \text{ e [ptr\_val } s]$ 
by simp

```

```

lemma free_set_singleton:
 $\llbracket s \neq e; v = \text{ptr\_val } e \rrbracket \implies$ 
 $\text{free\_set (h(s := Some } v)) s \text{ e \{ptr\_val } s}$ 
by (auto simp: free_set_def intro: list_singleton)

```

```

syntax
free_set_h :: heap_mem  $\times$  heap_typ_desc  $\Rightarrow$  word32 ptr  $\Rightarrow$  word32 ptr  $\Rightarrow$  free_set

```

```

translations
free_set_h h s e F == free_set (lift $_{\tau^c}$  h) s e F

```

```

lemma free_set_valid:
 $\llbracket \text{ptr\_val } p \in F; \text{free\_set}_h (h,d) s \text{ e } F \rrbracket \implies d \models_t (p :: \text{word32 ptr})$ 
by (auto dest: free_set_neq_None simp: lift $_{\tau}$ _if split: split_if_asm)

```

```

lemma free_set_h_singleton:
 $\llbracket d \models_t p; \text{lift } h \text{ } p = \text{ptr\_val } q; \text{ptr\_val } p \neq \text{ptr\_val } q; p' = \text{ptr\_val } p \rrbracket \implies \text{free\_set}_h (h,d) p$ 
 $q \{p'\}$ 
apply (unfold free_set_def)
apply auto
apply (rule_tac x=[ptr_val p] in exI)
apply clarsimp
apply (case_tac p, case_tac q)
apply clarsimp
apply (simp add: ktyp_simps)
done

```

```

lemma free_set_h_NULL:
free_set_h h x y G  $\implies 0 \notin G$ 
apply (clarsimp)
apply (subgoal_tac ptr_val (NULL::word32 ptr)  $\in$  G)
prefer 2
apply simp
apply (case_tac h)

```

```

apply(drule free_set_valid)
  apply force
apply simp
done

end

```

## 4 Shared Lemmas for Separation Logic Case

```

theory sep_common imports kmalloc begin

declare ucast_1 [simp]

constdefs no_wrap :: word32 ptr  $\Rightarrow$  word32  $\Rightarrow$  bool
  no_wrap p i  $\equiv$  ptr_val p  $\leq$  ptr_val p + (i - 1)*KMC

lemma no_wrapD:
  no_wrap p i  $\implies$  ptr_val p  $\leq$  ptr_val p + (i - 1)*KMC
  by (simp add: no_wrap_def)

lemma no_wrap:
  no_wrap p i  $\implies$  ptr_val p  $\leq$  ptr_val p + KMC * (i - 1)
  apply(clarsimp simp: no_wrap_def)
  apply(case_tac i=1)
  apply clarsimp
  apply(simp add: mult_ac)
done

consts zero_block :: 'a::{c_type,zero} ptr  $\Rightarrow$  nat  $\Rightarrow$  heap_assert
primrec
  zero_block p 0 =  $\square$ 
  zero_block p (Suc n) = (((p +p ((of_nat n)::word32))  $\mapsto$  0)  $\wedge^*$  zero_block p n)

lemma zero_block_ptr_aligned [rule_format]:
   $\forall s. \text{zero\_block } p \ n \ s \longrightarrow 0 < n \longrightarrow \text{ptr\_aligned } p$ 
  apply(induct_tac n)
  apply simp
  apply clarsimp
  apply(case_tac n)
  apply clarsimp
  apply(case_tac p, clarsimp)
  apply(rule c_guard_ptr_aligned)
  apply(rule sep_map'_g)
  apply(erule sep_map_sep_map')
  apply(erule impE)
  apply(drule sep_conjD, clarify)+
  apply fast
  apply clarsimp
done

constdefs sep_free_set_h :: word32 ptr ptr  $\Rightarrow$  word32 ptr  $\Rightarrow$  free_set  $\Rightarrow$  heap_assert
  sep_free_set_h s e F  $\equiv$   $\lambda h. \exists x. ((\text{ptr\_coerce } s \mapsto \text{ptr\_val } x) \wedge^* \text{sep\_free\_set } x \ e \ F) \ h$ 

lemma sep_free_set_hD:
  sep_free_set_h s e F h  $\implies$   $\exists x. ((\text{ptr\_coerce } s \mapsto \text{ptr\_val } x) \wedge^* \text{sep\_free\_set } x \ e \ F) \ h$ 
  by (simp add: sep_free_set_h_def)

```

```

lemma sep_free_set_h_empty [simp]:
  sep_free_set_h s e {} = (ptr_coerce s  $\mapsto$  ptr_val e)
  by (simp add: sep_free_set_h_def sep_free_set_def)

lemma sep_free_set_h_end:
  sep_free_set_h s e F hp  $\implies$  ptr_val e  $\notin$  F
  apply (drule sep_free_set_hD, clarsimp)
  apply (drule sep_conjD, clarsimp)
  apply (drule sep_free_set_end, fast)
  done

lemma sep_free_set_h_unionI:
   $\llbracket$  (sep_free_set_h s t F0  $\wedge^*$  sep_free_set t e F1) hp; ptr_val e  $\notin$  F0  $\rrbracket \implies$ 
  sep_free_set_h s e (F0  $\cup$  F1) hp
  apply (simp add: sep_free_set_h_def)
  apply (subst (asm) sep_conj_com)
  apply (subst (asm) sep_conj_exists)
  apply clarsimp
  apply (rule_tac x=x in exI)
  apply (subst (asm) sep_conj_assoc [symmetric])+
  apply (subst (asm) sep_conj_com) back
  apply (erule (1) sep_conj_impl)
  apply (erule (1) sep_free_set_unionI)
  done

lemma sep_free_set_h_split_elem_eq:
   $\llbracket$  p  $\in$  F; ptr_val e  $\neq$  p; ptr_val e  $\notin$  F  $\rrbracket$ 
 $\implies$  sep_free_set_h s e F =
  ( $\lambda$ hp.  $\exists$ F0 F1 p'.
    F = F0  $\cup$  F1  $\cup$  {p}  $\wedge$ 
    p  $\neq$  p'  $\wedge$ 
    (sep_free_set_h s (Ptr p) F0  $\wedge^*$ 
     kmalloc.block (Ptr p) (Ptr p')  $\wedge^*$  sep_free_set (Ptr p') e F1)
    hp)
  apply (unfold sep_free_set_h_def)
  apply (subst sep_free_set_split_elem_eq)
  apply simp+
  apply (subst sep_conj_assoc [symmetric])+
  apply (subst sep_conj_com) back back back back
  apply (subst sep_conj_exists)
  apply clarsimp
  apply (subst sep_conj_com)
  apply (subst sep_conj_exists)
  apply (subst sep_conj_exists)
  apply clarsimp
  apply (subst sep_conj_com)
  apply (subst sep_conj_exists)
  apply clarsimp
  apply rule
  apply fast
  done

constdefs rest_of :: word32 ptr  $\Rightarrow$  word32  $\Rightarrow$  heap_assert
  rest_of x y  $\equiv$  sep_cut (ptr_val x + 4) (y - 4)

lemma block_intvl_split_disj:
  {ptr_val x.. $+$ 4}  $\cap$  {ptr_val x + 4.. $+$ unat (KMC - 4)} = {}
  apply (rule ccontr)

```

```

apply(drule intvl_inter)
apply(clarsimp simp: intvl_def)
apply(erule disjE)
  prefer 2
  apply clarsimp
  apply(subgoal_tac (4::word32) = of_nat 4)
  apply(simp only:)
  apply(subst (asm) word_unat_norm_eq_iff [symmetric])
  apply clarsimp
  apply simp
apply clarsimp
apply(subst (asm) word_unat_Rep_inject [symmetric])
apply clarsimp
apply(subst (asm) word_arith_nat_defs)
apply(subst (asm) word_unat_eq_norm)
apply simp
apply(subst (asm) word_unat_eq_norm)
apply(subst (asm) mod_if)
apply(split split_if_asm)
  apply(clarsimp simp: KMC_def)
apply(erule notE)
apply(erule less_trans)
apply(rule unat_lt2p)
done

lemma block_intvl_split:
  {ptr_val x..+unat KMC} = {ptr_val x..+4} ∪ {ptr_val x + 4..+unat (KMC - 4)}
apply rule
  apply clarsimp
  apply(drule intvlD, clarsimp)
  apply(case_tac k < 4)
  apply(rule intvlI)
  apply simp
  apply(erule notE)
  apply(subgoal_tac ptr_val x + of_nat k = (ptr_val x + 4) + of_nat (k - 4))
  apply(simp only:)
  apply(rule intvlI)
  apply simp
  apply(simp add: KMC_def)
  apply simp
apply clarsimp
apply rule
  apply clarsimp
  apply(drule intvlD, clarsimp)
  apply(rule intvlI)
  apply(simp add: KMC_def)
apply clarsimp
apply(drule intvlD, clarsimp)
apply(subgoal_tac ptr_val x + 4 + of_nat k = ptr_val x + of_nat (k + 4))
  apply(simp only:)
  apply(rule intvlI)
  apply(simp add: KMC_def)
  apply simp
done

lemma block_alt:
  block x y = (λs. KMC udvd ptr_val x ∧ ((x ↦ ptr_val y) ∧* rest_of x KMC) s)
apply(clarsimp simp: block_def rest_of_def)

```

```

apply rule+
  apply clarsimp+
  apply(clarsimp simp: sep_map'_def)
  apply(drule sep_conjD, clarsimp)
  apply(frerule sep_map_dom)
  apply(frerule sep_cut_dom)
  apply(frerule heap_disj_dom)
  apply(clarsimp simp: block_intvl_split)
  apply(subgoal_tac dom s0 = {ptr_val x + 4..unat (KMC - 4)})
  prefer 2
  apply(insert block_intvl_split_disj [of x])
  apply blast
  apply(erule_tac s1=s0 in sep_conjI)
  apply(clarsimp simp: sep_cut_def sep_cut'_def)
  apply(simp add: heap_disj_com)
  apply(simp add: heap_merge)
apply clarsimp
apply rule
  apply(subst sep_map'_unfold)
  apply(erule sep_conj_impl)
  apply(erule sep_map_sep_map')
  apply simp
apply(drule sep_conjD, clarsimp simp: sep_cut_def sep_cut'_def heap_disj_def block_intvl_split)
apply(frerule sep_map_dom)
apply clarsimp
done

```

```

lemma sep_list_dom [rule_format]:
   $\forall x s. \text{sep\_list } x y \text{ ps } s \longrightarrow z \in \text{set ps} \longrightarrow z \in \text{dom } s$ 
apply(induct_tac ps)
  apply simp
apply clarsimp
apply rule
  apply(drule sep_conjD, clarsimp)
  apply(drule block_dom, clarsimp)
apply clarsimp
apply(drule sep_conjD, clarsimp)
apply(drule_tac x=s' in spec)
apply(drule_tac x=s1 in spec)
apply clarsimp
apply(subst heap_merge_com)
  apply(simp add: heap_disj_com)
apply force
done

```

```

lemma sep_free_set_dom:
   $\text{sep\_free\_set } x y F s \Longrightarrow F \subseteq \text{dom } s$ 
apply(unfold sep_free_set_def)
apply clarsimp
apply(drule (1) sep_list_dom)
apply force
done

```

```

lemma sep_free_set_h_dom':
   $\text{sep\_free\_set\_h } x y F s \Longrightarrow F \cup \{\text{ptr\_val } x\} \subseteq \text{dom } s$ 
apply(drule sep_free_set_hD)
apply clarsimp
apply(drule sep_conjD, clarsimp)

```

```

apply(drule sep_free_set_dom)
apply(drule sep_map_dom)
apply rule
  apply(rule disjI2)
  apply(clarsimp simp: intvl_def)
  apply(rule_tac x=0 in exI)
  apply simp
apply fast
done

lemma sep_free_set_h_dom:
  sep_free_set_h x y F s  $\implies$  F  $\subseteq$  dom s
apply(drule sep_free_set_h_dom')
apply fast
done

lemma sep_free_set_h_prev:
  [ ptr_val p  $\in$  G; (p  $\leftrightarrow$  ptr_val y) s; (sep_free_set_h x y G  $\wedge^*$  P) s; ptr_val y  $\neq$  ptr_val p;
  ptr_val y  $\notin$  G ]
   $\implies$  (sep_free_set_h x p (G - {ptr_val p})  $\wedge^*$  block p y  $\wedge^*$  P) s
apply(subst (asm) sep_free_set_h_split_elem_eq)
  apply fast
  apply simp
  apply simp
apply(subst (asm) sep_conj_exists)+
apply clarsimp
apply(rule ccontr)
apply(erule notE) back
apply(subgoal_tac (block p (Ptr xb)  $\wedge^*$  sep_true) s)
  apply(simp add: block_def)
  apply clarsimp
  apply(subgoal_tac (p  $\leftrightarrow$  xb) s)
    apply(drule (1) sep_map'_inj)
    apply clarsimp
    apply(subgoal_tac ptr_val p  $\notin$  xa)
      apply simp
      apply clarsimp
    apply(drule sep_conjD, clarsimp)+
    apply(drule sep_free_set_h_dom)+
    apply(subgoal_tac ptr_val p  $\in$  dom s1'')
      apply(unfold heap_disj_def, clarsimp)
      apply fast
    apply(drule sep_cut_dom) back
    apply(clarsimp simp: intvl_def)
    apply(rule_tac x=0 in exI)
    apply(simp add: KMC_def)
  apply(erule sep_map'_conjE2)
  apply clarsimp
apply(subst sep_conj_true [symmetric])
apply simp
apply(subst (asm) sep_conj_com)
apply(subst (asm) sep_conj_assoc)+
apply(subst sep_conj_com)
apply(erule (1) sep_conj_impl)
apply simp
done

lemma sep_free_set_prev_rev:

```

```

[[ ptr_val p ∈ G; (p ↔ ptr_val y) s; (sep_free_set x p (G - {ptr_val p}) ∧* block p y ∧* P)
s;
  ptr_val y ≠ ptr_val p; ptr_val y ∉ G ]]
⇒ (sep_free_set x y G ∧* P) s
apply(subst sep_free_set_split_elem_eq)
  apply fast
  apply simp
  apply simp
apply(subst sep_conj_exists)+
apply clarsimp
apply(rule_tac x=G - {ptr_val p} in exI)
apply(rule_tac x={} in exI)
apply clarsimp
apply rule
  apply fast
apply(rule_tac x=ptr_val y in exI)
apply clarsimp
done

```

```

lemma sep_free_set_h_prev_rev:
[[ ptr_val p ∈ G; (p ↔ ptr_val y) s; (sep_free_set_h x p (G - {ptr_val p}) ∧* block p y ∧*
P) s;
  ptr_val y ≠ ptr_val p; ptr_val y ∉ G ]]
⇒ (sep_free_set_h x y G ∧* P) s
apply(simp add: sep_free_set_h_def)
apply(subst (asm) sep_conj_assoc [symmetric])+
apply(subst (asm) sep_conj_com) back
apply(subst (asm) sep_conj_exists)
apply clarsimp
apply(subst sep_conj_com)
apply(subst sep_conj_exists)
apply(rule_tac x=xa in exI)
apply(subst sep_conj_assoc)
apply(erule (1) sep_free_set_prev_rev)
  apply simp+
done

```

```

lemma sep_free_set_h_empty_sep_map':
[[ (ptr_coerce x ↔ ptr_val y) s; (sep_free_set_h x y G ∧* P) s ]] ⇒ G = {}
apply(subst (asm) sep_free_set_h_def)
apply(subst (asm) sep_conj_exists, clarsimp)
apply(subgoal_tac (ptr_coerce x ↔ ptr_val xa) s)
  apply(drule (1) sep_map'_inj)
  apply simp
apply(subst sep_map'_unfold)
apply(subst (asm) sep_conj_com)
apply(subst (asm) sep_conj_com)
apply(subst (asm) sep_conj_assoc)+
apply(erule sep_conj_impl)
  apply(erule sep_map_sep_map')
apply simp
done

```

```

lemma sep_heap_update_block:
(block p v' ∧* R) (lift_state (h,d)) ⇒
  (block p v ∧* R) (lift_state (heap_update p (ptr_val v) h,d))
apply(clarsimp simp: block_alt)

```

```

apply(subst (asm) sep_conj_assoc [symmetric])+
apply(subst (asm) sep_conj_com) back
apply(subst sep_conj_assoc [symmetric])+
apply(subst sep_conj_com)
apply(subst sep_conj_com) back
apply(rule sep_heap_update_global)
apply(erule sep_conj_impl)
  apply simp+
done

```

```

lemma sep_map'_lift_ptr_coerce:
  (((ptr_coerce i)::word32 ptr ptr)  $\hookrightarrow_g$  Ptr x) (lift_state (h,d))  $\implies$  CTypes.lift h (i::word32 ptr) = (x::word32)
apply(unfold sep_map'_def)
apply(drule sep_conjD, clarsimp)
apply(drule_tac s0=s1 in lift_typ_heap_heap_merge_sep_map)
apply(drule sym, simp)
apply(subst (asm) liftτ)
apply(drule liftτ_lift)
apply(erule lift_ptr_coerce)
done

```

```

lemma sep_free_set_singletonD:
  [[ sep_free_set p q F h; p  $\neq$  q ]]  $\implies$  ( $\exists r. (p \hookrightarrow \text{ptr\_val } (r::\text{word32 ptr})) h \wedge$ 
    (sep_free_set p r {ptr_val p}  $\wedge^*$ 
      sep_free_set r q (F - {ptr_val p})) h)  $\wedge$  ptr_val p  $\in$  F
apply(frule sep_free_set_start, simp)
apply(drule_tac p=ptr_val p in sep_free_set_split_elem, assumption)
apply clarsimp
apply(rule_tac x=Ptr p' in exI)
apply rule
  apply(subst sep_map'_unfold)
  apply(erule sep_conj_impl)
  apply(clarsimp simp: block_def)
  apply simp
apply(subst (asm) sep_conj_com)
apply(subgoal_tac ptr_val p  $\notin$  F1)
apply(clarsimp simp: sep_free_set_block)
apply clarsimp
apply(drule sep_conjD, clarsimp)
apply(drule_tac p=ptr_val p in sep_free_set_split_elem, assumption)
apply clarsimp
apply(drule sep_conjD, clarsimp)+
apply(drule block_dom)+
apply(clarsimp simp: heap_disj_def)
apply fast
done

```

```

lemma sep_free_set_singleton:
  p  $\neq$  q  $\implies$  sep_free_set p q F = ( $\lambda s. \text{ptr\_val } p \in F \wedge$ 
    ( $\exists r. (\text{sep\_free\_set } p (\text{Ptr } r) \{ \text{ptr\_val } p \} \wedge^*$ 
      sep_free_set (Ptr r) q (F - {ptr_val p})) s))
apply rule
  apply rule
  apply(drule (1) sep_free_set_singletonD)

```

```

apply clarsimp
apply(case_tac r, clarsimp)
apply fast
apply clarsimp
apply(drule sep_free_set_unionI)
apply simp
apply simp
apply(subgoal_tac insert (ptr_val p) F = F)
apply simp
apply fast
done

```

```

lemma sep_free_set_h_NULL [simp]:
  sep_free_set_h NULL y F = sep_false
  by (simp add: sep_free_set_h_def)

```

```

lemma sep_list_aligned [rule_format]:
   $\forall x s. \text{sep\_list } x \ y \ ps \ s \longrightarrow \text{aligned } (\text{set } ps)$ 
apply(induct_tac ps)
  apply(clarsimp simp: aligned_def align_def)
  apply(clarsimp simp: aligned_def align_def)
  apply(drule sep_conjD, clarsimp)
  apply(erule impE)
  apply force
  apply(subgoal_tac a  $\in$  Collect (op udvd KMC))
  apply fast
  apply(clarsimp simp: block_def)
done

```

```

lemma sep_free_set_aligned:
  sep_free_set x y F s  $\implies$  aligned F
  apply(clarsimp simp: sep_free_set_def)
  apply(erule sep_list_aligned)
done

```

```

lemma block_NULL [simp]:
  block NULL y = sep_false
  apply(rule ext)
  apply(clarsimp simp: block_def)
done

```

```

lemma sep_list_NULL [rule_format]:
   $\forall x s. \text{sep\_list } x \ y \ ps \ s \longrightarrow 0 \notin \text{set } ps$ 
apply(induct_tac ps)
  apply simp
  apply clarsimp
  apply(drule sep_conjD, clarsimp)
  apply(erule impE)
  apply fast
  apply simp
  apply(subgoal_tac 0  $\neq$  a)
  apply simp
  apply clarify
  apply simp
done

```

```

lemma sep_free_set_NULL:

```

```

sep_free_set x y F s  $\implies$  0  $\notin$  F
by (clarsimp simp: sep_free_set_def dest!: sep_list_NULL)

lemma sep_free_set_h_NULL:
  sep_free_set_h x y F s  $\implies$  0  $\notin$  F
  by (clarsimp simp: sep_free_set_h_def sep_conj_def dest!: sep_free_set_NULL)

end

```

## 5 kalloc without separation logic

```
theory kalloc imports non_sep_common sep_common begin
```

```
declare ucast_1 [simp]
```

```

lemma free_set_h_term:
   $\llbracket$  free_set_h (h,d) x y F; ptr_val p  $\in$  F; lift h p  $\notin$  F  $\rrbracket \implies$ 
    Ptr (lift h p) = y
  apply (drule (1) free_set_split_elem)
  apply (clarsimp simp: typ_simps)
  apply (drule (1) free_set_start)
  apply simp
done

```

```

lemma free_set_h_nself_loop:
   $\llbracket$  free_set_h (h,d) x y F; p  $\in$  F  $\rrbracket \implies$  p  $\neq$  lift h (Ptr p)
  apply (drule (1) free_set_split_elem)
  apply (clarsimp simp: typ_simps)
done

```

```

lemma free_set_h_singletonD:
   $\llbracket$  free_set_h (h,d) p q F; p  $\neq$  q  $\rrbracket \implies$  free_set_h (h,d) p (Ptr (lift h p))
    {ptr_val p}  $\wedge$  free_set_h (h,d) (Ptr (lift h p)) q (F - {ptr_val p})  $\wedge$ 
    ptr_val p  $\in$  F
  apply (frule free_set_start, simp)
  apply (drule_tac p=ptr_val p in free_set_split_elem, assumption)
  apply (clarsimp simp: typ_simps)
  apply (rule free_set_h_singleton)
  apply (simp add: typ_simps)+
done

```

```

lemma free_set_h_slide:
   $\llbracket$  free_set_h (h,d) x y G; free_set_h (h,d) y z H; G  $\cap$  H = {}; y  $\neq$  z;
    ptr_val z  $\notin$  G  $\rrbracket \implies \exists G'. \text{ptr\_val } y \in G' \wedge$ 
    free_set_h (h,d) x (Ptr (lift h y)) G'  $\wedge$ 
    ( $\exists H'. \text{free\_set\_h } (h,d) (\text{Ptr } (\text{lift } h \ y)) \ z \ H' \wedge$ 
    G'  $\cap$  H' = {}  $\wedge$  G  $\cup$  H = G'  $\cup$  H')
  apply (rule_tac x=G  $\cup$  {ptr_val y} in exI)
  apply rule
  apply simp
  apply rule
  apply (erule free_set_unionI)
  apply (frule free_set_h_singletonD, simp, force)
  apply (drule free_set_start, simp, fast)
  apply clarsimp

```

```

apply(frule_tac p=y in free_set_h_term, drule free_set_start, fastsimp, fast)
  apply fast
  apply(case_tac z, clarsimp)
apply(frule free_set_h_singletonD, assumption+)
apply(rule_tac x=H - {ptr_val y} in exI)
apply clarsimp
apply rule
  apply fastsimp+
done

```

```

lemma free_set_h_slide':
  [[ free_set_h (h,d) x y G; free_set_h (h,d) y NULL H; G ∩ H = {}; y ≠ NULL ]]
    ⇒ ∃ G'. ptr_val y ∈ G' ∧
      free_set_h (h,d) x (Ptr (lift h y)) G' ∧
      (∃ H'. free_set_h (h,d) (Ptr (lift h y)) NULL H' ∧
        G' ∩ H' = {} ∧ G ∪ H = G' ∪ H')
apply(frule free_set_h_NULL)
apply(simp add: free_set_h_slide)
done

```

```

lemma free_set_adjust:
  [[ free_set h x y E; free_set h y z F; free_set h z a G;
    the (h prev) = ptr_val y; ptr_val prev ∈ E; E ∩ G = {}; ptr_val a ∉ E ]]
    ⇒ free_set (h(prev ↦ ptr_val z)) x a (E ∪ G)
apply(case_tac z=a)
  apply(drule_tac p=ptr_val prev in free_set_split_elem, assumption)
  apply clarsimp
  apply(rule ccontr)
  apply(erule notE)
  apply(subgoal_tac insert (ptr_val prev) F0 = F0 ∪ {ptr_val prev})
  prefer 2
  apply simp
  apply(simp only:)
  apply(rule free_set_unionI)
  apply simp+
  apply(rule free_set_singleton)
  apply simp+
apply(rule_tac t=z in free_set_unionI)
  apply(drule_tac p=ptr_val prev in free_set_split_elem, assumption)
  apply clarsimp
  apply(rule ccontr)
  apply(erule notE) back
  apply(subgoal_tac insert (ptr_val prev) F0 = F0 ∪ {ptr_val prev})
  prefer 2
  apply simp
  apply(simp only:)
  apply(rule free_set_unionI)
  apply simp+
  apply(rule free_set_singleton)
  apply(drule free_set_start, clarsimp, fast)
  apply simp+
  apply(drule free_set_start, clarsimp, fast)
  apply(subgoal_tac ptr_val prev ∉ G)
  apply simp
  apply fast+
done

```

```

lemma h_t_valid_c_guard_ptr_coerce:
  d  $\models$  ((ptr_coerce p)::word32 ptr)  $\implies$  c_guard (p::word32 ptr ptr)
apply(drule h_t_valid_c_guard)
apply(clarsimp simp: c_guard_def ptr_aligned_def)
done

lemma list_same [rule_format]:
   $\forall x \text{ ys. list h x y xs} \longrightarrow \text{list h x y ys} \longrightarrow \text{xs=ys}$ 
apply(induct_tac xs)
  apply simp
apply clarsimp
apply(case_tac ys)
  apply simp
  apply clarsimp
done

lemma free_set_same:
  [ free_set h x y F; free_set h x y G ]  $\implies$  F=G
apply(clarsimp simp: free_set_def)
apply(drule (1) list_same)
apply simp
done

install_C_file kmemory.ii

locale alloc_imp = alloc_impl +
  constrains sep_alloc_invs_body :: globals myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char list,
c_errortype) com
  and alloc_invs_body :: globals myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char list, c_errortype)
com
  and sep_free_invs_body :: unit myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char list, c_errortype)
com
  and free_invs_body :: unit myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char list, c_errortype)
com

lemma (in alloc_imp)
  alloc_spec  $\Gamma$  kmem_free_list_addr
apply (unfold alloc_spec_def)
apply (rule conjI)
  apply (rule kmemory_axioms)
  apply (unfold alloc_spec_axioms_def)
  apply (hoare_rule ProcNoRec1)
  apply (hoare_rule anno=alloc_invs_body  $\sigma$  F in annotateI)
  prefer 2
  apply (subst alloc_invs_body_def)
  apply (simp add: whileAnno_def)
  apply (subst alloc_invs_body_def)
  apply (fold KMC_def max_def)
  apply vcg

— Pre  $\implies$  I0

apply clarsimp
apply (rule context_conjI)
  apply (cut_tac kmem_free_list_addr_ptrok)
  apply (simp add: okptr_def)
apply rule
  apply (cut_tac kmem_free_list_addr_ptrok)

```

```

  apply (simp add: okptr_def)
  apply rule
  apply (drule free_seth_singletonD, fastsimp)
  apply (rule_tac x={ptr_val kmem_free_list_addr} in exI)
  apply force
  apply (frule free_set_end)
  apply clarsimp
  apply (rule max_size_KMC)

— I0 ⇒ I1

  apply rule
  apply clarsimp
  apply rule
  apply (frule free_set_start, fastsimp, drule (1) free_set_valid,
    rule c_guard_ptr_aligned, erule h_t_valid_c_guard)
  apply rule
  apply rule
  apply (clarsimp simp: no_wrap_def)
  apply (rule_tac x=G in exI)
  apply clarsimp
  apply (frule free_set_start, simp)
  apply (frule_tac p=lift t_h curr and F=H in free_set_split)
  apply clarify
  apply (rule ccontr)
  apply (drule (2) free_seth_term)
  apply clarsimp
  apply (erule exE | erule conjE)+
  apply (frule free_seth_singletonD, clarsimp,
    drule (1) free_seth_nself_loop, case_tac curr, clarsimp, clarsimp)
  apply (rule_tac x=F1 in exI)
  apply force
  apply clarsimp
  apply rule
  apply (frule free_set_start, fastsimp, drule (1) free_set_valid,
    rule c_guard_ptr_aligned, erule h_t_valid_c_guard)
  apply (clarsimp simp: free_seth_slide')

— I1 ⇒ I1', I1 ⇒ I0' (when a suitable chunk hasn't been found)

  apply simp
  apply rule
  apply (clarsimp simp: mult_ac)
  apply (case_tac tmp=curr)
  apply rule
  apply simp
  apply (frule free_set_start, force,
    drule (1) free_set_valid, rule c_guard_ptr_aligned,
    erule h_t_valid_c_guard)
  apply rule
  prefer 2
  apply simp
  apply (rule_tac x=G ∪ {ptr_val curr} in exI)
  apply rule
  apply clarsimp
  apply rule
  apply (rule free_set_unionI, assumption)
  apply (frule free_seth_singletonD, clarsimp, clarsimp, clarsimp)

```

```

    apply(frule (1) free_set_start)
    apply blast
    apply clarsimp
    apply(frule_tac p=curr in free_seth_term, drule free_set_start, fastsimp, fast)
    apply blast
    apply(clarsimp dest!: free_seth_NULL)
    apply(frule free_seth_singletonD, assumption+, simp)
    apply(rule_tac x=H - {ptr_val curr} in exI)
    apply force
  apply rule
  apply(frule free_set_start, force,
        drule (1) free_set_valid, rule c_guard_ptr_aligned,
        erule h_t_valid_c_guard)
  apply rule
  prefer 2
  apply simp
  apply(rule_tac x=G ∪ {ptr_val curr} in exI)
  apply rule
  apply clarsimp
  apply rule
  apply(rule free_set_unionI, assumption)
  apply(frule free_seth_singletonD, simp, fastsimp)
  apply clarsimp
  apply(drule free_set_start, simp, fast)
  apply clarsimp
  apply(drule free_seth_singletonD) back
  apply simp
  apply clarsimp
  apply(drule free_set_start) back back back
  apply clarsimp
  apply(drule free_set_start) back
  apply(clarify, simp)
  apply(clarsimp dest!: free_seth_NULL)
  apply clarsimp
  apply blast
  apply clarsimp
  apply blast
  apply(rule_tac x=((chunks curr (ptr_val curr +(i - 1)*KMC)) - {ptr_val curr}) ∪ H in exI)
  apply(clarsimp simp: mult_ac)
  apply rule
  apply(rule free_set_unionI)
  apply(drule free_seth_singletonD, simp, force)
  apply assumption
  apply fastsimp
  apply clarsimp
  apply(clarsimp dest!: free_seth_NULL)
  apply(subgoal_tac ptr_val curr ∈ chunks curr (ptr_val curr + KMC * (i - 1)))
  apply force
  apply(rule chunks_start)
  apply(erule no_wrap)
  apply rule
  apply rule
  apply(erule exE | erule conjE)+
  apply clarsimp
  apply(frule free_set_start, simp)
  apply(drule (1) free_set_valid, rule c_guard_ptr_aligned,
        erule h_t_valid_c_guard)
  apply rule

```



```

apply(subst chunks_add)
  apply(clarsimp simp: no_wrap_def)
  apply(rule lt_add_KMC)
  apply(subst unroll_KMC [symmetric])
  apply(case_tac tmp, clarsimp simp: mult_ac)
  apply(erule alignedD2)
  apply(clarsimp simp: no_wrap_def)
  apply(drule chunks_end_KMC)
  apply(simp add: mult_ac)
  apply(rule udvd_expand)
  apply clarsimp
  apply(rule udvd_KMC)
  apply(clarsimp simp: no_wrap_def)
  apply(subst unroll_KMC [symmetric])
  apply simp
  apply(frule no_wrapD)
  apply(simp (no_asm) only: no_wrap_def)
  apply(erule order_trans)
  apply(subst unroll_KMC) back
  apply(simp add: mult_ac)
  apply(rule order_less_imp_le)
  apply(rule lt_add_KMC)
  apply(subst mult_ac)
  apply(subst unroll_KMC [symmetric])
  apply(case_tac tmp, clarsimp)
  apply(erule alignedD2)
  apply clarsimp
  apply(frule no_wrapD)
  apply(drule chunks_end_KMC)
  apply(simp add: mult_ac)
apply rule
  prefer 2
  apply(erule (1) inc_i)
apply clarsimp
apply(rule_tac x=G in exI)
apply clarsimp
apply(rule_tac x=chunks curr (ptr_val curr + (i - 1)*KMC) ∪ H in exI)
apply rule
  apply(rule free_set_unionI, assumption+)
  apply fast
  apply(clarsimp dest!: free_set_h_NULL)
apply(simp add: mult_ac)
apply blast

```

—  $I_1 \implies I_2, I_1 \implies I_0'$

```

apply rule
  apply simp
  apply rule
  apply rule
  apply(clarsimp dest!: free_set_h_NULL)
apply rule
  apply simp
  apply(erule exE | erule conjE)+
  apply clarsimp
  apply(drule (1) free_set_valid)
  apply(rule c_guard_ptr_aligned)
  apply(erule h_t_valid_c_guard)

```

```

apply rule
  apply simp
  apply(erule exE | erule conjE)+
  apply clarsimp
  apply(frule (1) free_set_valid)
  apply(clarsimp simp: typ_simps)
  apply(rule ccontr)
  apply(simp add: mult_ac)
  apply(frule free_set_h_NULL)
  apply(drule (5) free_set_adjust)
  apply clarsimp
  apply(subgoal_tac alloc (ptr_coerce curr) (max size KMC)
    (G  $\cup$  chunks curr (ptr_val curr + KMC * (max size KMC div KMC - 1))  $\cup$  H) = G  $\cup$  H)
  apply simp
  apply(clarsimp simp: alloc_def)
  apply(subgoal_tac KMC * (max size KMC div KMC - 1) = max size KMC - KMC)
  apply clarsimp
  apply blast
  apply(subst right_diff_distrib)
  apply clarsimp
  apply(rule max_KMC_div_times)
  apply simp
apply rule
  apply(clarsimp simp: alloc_def)
apply rule
  apply(erule KMC_max)
apply rule
  apply clarsimp
  apply(drule_tac p=ptr_val curr in alignedD2)
  apply clarsimp
  apply(rule chunks_start)
  apply(simp add: no_wrap_def)
  apply simp
  apply(clarsimp simp: no_wrap_def mult_ac)
  apply(subgoal_tac KMC * (max size KMC div KMC - 1) = max size KMC - KMC)
  apply clarsimp
  apply(subst right_diff_distrib)
  apply clarsimp
  apply(rule max_KMC_div_times)
  apply simp
apply rule
  apply simp
  apply(erule exE | erule conjE)+
  apply clarsimp
  apply rule
  apply(frule free_set_start, simp, drule (1) free_set_valid,
    rule c_guard_ptr_aligned, erule h_t_valid_c_guard)
  apply rule
  apply(clarsimp simp: free_set_h_slide')
  apply simp

```

—  $I_2 \implies I_2'$

```

apply clarsimp
  apply(subst free_set_restrict_dom [symmetric])
  apply(subst h_val_update_id'_align_F)
  apply(case_tac KMC udvd ptr_val (curr +p i))

```

```

    apply(drule_tac i=i in aligned_chunks)
      apply(erule (1) KMC_udvd_plus)
      apply assumption
      apply(clarsimp simp: word_le_def max_def)
      apply simp
      apply(clarsimp simp: alloc_def)
      apply(subgoal_tac aligned (alloc (ptr_coerce curr) (max size KMC) F))
      apply clarsimp
      apply(drule (1) alignedD2)
      apply simp
      apply(erule aligned_sub)
      apply(clarsimp simp: alloc_def)
      apply(rule ptr_aligned_plus)
      apply(erule ptr_aligned_KMC)
      apply(erule aligned_sub)
      apply(clarsimp simp: alloc_def)
      apply simp
      apply fast
      apply(simp add: align_of_def size_of_def typ_info_word aalign_def typ_size_def)
      apply(simp add: align_of_def size_of_def typ_info_word aalign_def typ_size_def)
      apply(simp add: free_set_restrict_dom)
      apply rule
      apply(rule index_not_NULL)
      apply assumption+
      apply(rule ptr_aligned_plus)
      apply(erule ptr_aligned_KMC)

```

—  $I_2 \implies \text{Post}$

```

      apply(clarsimp simp: alloc_def)+
      done

```

end

## 6 kalloc with separation logic

```

theory sep_kalloc imports non_sep_common sep_common begin

```

```

install_C_file memsafe kmemory.ii

```

```

locale sep_alloc_imp = sep_alloc_impl +
  constrains sep_alloc_invs_body :: globals myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char list,
c_errortype) com
  and alloc_invs_body :: globals myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char list, c_errortype)
com
  and sep_free_invs_body :: unit myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char list, c_errortype)
com
  and free_invs_body :: unit myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char list, c_errortype)
com

```

```

lemma sep_free_set_slide:

```

```

  [ (sep_free_set_h x y G  $\wedge^*$  sep_free_set y z H) (lift_state (h,d)); y  $\neq$  z;
    ptr_val z  $\notin$  G ]  $\implies$   $\exists$ Ga Ha.
    (sep_free_set_h x (Ptr (CTypes.lift h y)) Ga  $\wedge^*$ 
      sep_free_set (Ptr (CTypes.lift h y)) z Ha) (lift_state (h,d))  $\wedge$ 

```

```

      (y = ptr_coerce x ∨ ptr_val y ∈ Ga) ∧
      (G ∪ H) = (Ga ∪ Ha)
apply(subgoal_tac ptr_val y ∉ G)
prefer 2
apply(subst (asm) sep_free_set_h_def)
apply(drule sep_conjD, clarsimp)+
apply(frule (1) sep_free_set_start)
  apply(drule sep_free_set_dom)+
apply(unfold heap_disj_def)
apply simp
apply blast
apply(rule_tac x=G ∪ {ptr_val y} in exI)
apply(rule_tac x=H - {ptr_val y} in exI)
apply rule
apply(subst (asm) sep_free_set_singleton)
  apply assumption
apply clarsimp
apply(subst (asm) sep_conj_com)
apply(subst (asm) sep_conj_exists)
apply clarsimp
apply(subst (asm) sep_conj_com)
apply(subst (asm) sep_conj_assoc)
apply(frule sep_map'_conjE1)
  apply(clarsimp simp: sep_free_set_block block_def)
  apply fast
apply(simp add: sep_map'_lift_ptr_coerce)
apply(subgoal_tac xa ∉ G)
  prefer 2
  apply clarsimp
  apply(drule sep_conjD, clarsimp)+
  apply(frule sep_free_set_start) back
  apply clarsimp+
  apply(drule sep_free_set_dom)+
  apply(drule sep_free_set_h_dom)
  apply(unfold heap_disj_def)
  apply simp
  apply blast
apply(subst (asm) sep_conj_assoc [symmetric])+
apply(erule sep_conj_impl)
  apply(subgoal_tac insert (ptr_val y) G = G ∪ {ptr_val y})
  apply(simp only:)
  apply(rule sep_free_set_h_unionI)
  apply(simp add: sep_map'_lift)+
apply(drule sep_conjD, clarsimp)+
apply(drule sep_free_set_start)
  apply simp
apply fast
done

```

lemma zero\_block\_unroll:

```

  n < x div 4 ⇒ zero_block p (unat (n + 1)) =
    zero_block p (unat n) ∧* ((p +p n) ↦ 0)
apply(subst word_arith_nat_defs)
apply(subst word_unat_eq_norm)
apply(subst mod_if)
apply(clarsimp split: split_if_asm)
apply(erule notE)
apply(clarsimp simp: word_less_def word_le_nat_alt)

```

```

apply(clarsimp simp: unat_div)
apply(subst (asm) div_if)
  apply simp
apply(clarsimp split: split_if_asm)
apply(erule order_le_less_trans)
apply clarsimp
apply(rule order_le_less_trans)
  apply(rule div_le_dividend)
apply(insert unat_lt2p [of x])
apply simp
done

lemma sep_list_Some [rule_format]:
   $\forall x s. \text{sep\_list } x \ y \ ps \ s \longrightarrow s \ k = \text{Some } z \longrightarrow$ 
   $(\exists p. p \in \text{set } ps \wedge (\exists r < \text{KMC}. k = p + r))$ 
apply(induct_tac ps)
  apply clarsimp
  apply(drule sep_empD)
  apply simp
apply clarsimp
apply(drule sep_conjD, clarsimp)
apply(erule disjE)
  apply(clarsimp simp: block_def)
  apply(drule sep_cut_dom)
  apply(subgoal_tac  $k \in \{a..+\text{unat } \text{KMC}\}$ )
    prefer 2
    apply fast
  apply(drule intvlD, clarsimp)
  apply(rule_tac x=a in exI)
  apply(clarsimp simp: word_less_def)
  apply rule
    apply(simp add: word_le_nat_alt KMC_def)
    apply(subst word_unat_eq_norm)
    apply simp
  apply clarsimp
  apply(subst (asm) word_unat_Rep_inject [symmetric],
    subst (asm) word_unat_eq_norm)
  apply(clarsimp simp: KMC_def)
apply(drule_tac x=s' in spec)
apply(drule_tac x=s1 in spec)
apply clarsimp
apply(rule_tac x=p in exI)
apply clarsimp
done

lemma sep_free_set_Some:
   $[\text{sep\_free\_set } x \ y \ F \ s; s \ k = \text{Some } z] \Longrightarrow \exists p \ r. p \in F \wedge r < \text{KMC} \wedge k = p + r$ 
apply(clarsimp simp: sep_free_set_def)
apply(erule (1) sep_list_Some)
done

lemma sep_list_Some2 [rule_format]:
   $\forall x s. \text{sep\_list } x \ y \ ps \ s \longrightarrow p \in \text{set } ps \longrightarrow r < \text{KMC} \longrightarrow s \ (p + r) \neq \text{None}$ 
apply(induct_tac ps)
  apply simp
apply clarsimp
apply rule
  apply clarsimp

```

```

apply(drule sep_conjD, clarsimp simp: block_def sep_cut_def sep_cut'_def)
apply(subgoal_tac p + r ∈ dom s₀)
  apply force
  apply(clarsimp simp: intvl_def)
  apply(rule_tac x=unat r in exI)
  apply(simp add: word_less_nat_alt)
apply clarsimp
apply(drule sep_conjD, clarsimp)
apply(drule_tac x=s' in spec)
apply(drule_tac x=s₁ in spec)
apply clarsimp
apply(subst heap_merge_com)
  apply(simp add: heap_disj_com)
apply simp
done

```

lemma sep\_free\_set\_Some2:

```

[[ sep_free_set x y F s; p ∈ F; r < KMC ]] ==> s (p + r) ≠ None
apply(clarsimp simp: sep_free_set_def)
apply(drule (2) sep_list_Some2)
apply simp
done

```

lemma sep\_free\_set\_cut:

```

[[ sep_free_set x y (chunks x (ptr_val x + (k - KMC))) s;
  ptr_val x ≤ ptr_val x + (k - KMC); KMC ≤ k;
  KMC udvd k ]] ==> sep_cut (ptr_val x) k s
apply(clarsimp simp: sep_cut_def sep_cut'_def)
apply rule
  apply(thin_tac ?x ≤ ?y)
  apply clarsimp
  apply(drule (1) sep_free_set_Some)
  apply(clarsimp simp: chunks_def intvl_def)
  apply(subgoal_tac unat p = unat (ptr_val x) + nat n * unat KMC)
  prefer 2
  apply(drule_tac f=nat in arg_cong)
  apply(subst (asm) nat_add_distrib)
    apply(rule uint_ge_0)
  apply(rule mult_nonneg_nonneg)
  apply simp
  apply(rule uint_ge_0)
  apply(subst (asm) nat_mult_distrib)
  apply simp
  apply(fold unat_def)
  apply simp
  apply(rule_tac x=nat n*unat KMC + unat r in exI)
  apply(rule context_conjI)
  apply simp
  apply(subst word_unat_Rep_inject [symmetric])
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_norm_Rep [symmetric])
  apply simp
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_eq_norm)
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_eq_norm)
  apply(subst word_unat_eq_norm)
  apply simp

```

```

    apply(subst mod_add1_eq)
    apply(subst mod_add1_eq, subst mod_mult1_eq', force)
  apply simp
  apply(frule (1) order_trans)
  apply(subst (asm) word_le_nat_alt) back back
  apply(simp)
  apply(subst (asm) word_arith_nat_defs) back back back
  apply(subst (asm) word_unat_eq_norm)
  apply(subst (asm) mod_less)
  apply simp
  apply(subst (asm) no_plus_overflow_unat)
  apply(simp add: flen_def)
  apply simp
  apply(subst (asm) unat_def, subst (asm) uint_sub_if)
  apply(simp split: split_if_asm)
  apply(subst (asm) nat_diff_distrib)
    apply(rule uint_ge_0)
    apply simp
  apply(fold unat_def)
  apply(subst (asm) word_less_nat_alt)
  apply(rule_tac j=nat n* unat KMC + unat KMC in less_le_trans)
    apply arith
  apply(subst (asm) le_diff_conv2)
    apply(subst (asm) word_le_nat_alt, force)
  apply simp
  apply(subst (asm) word_le_def, force)
  apply clarsimp
  apply(drule (2) intvl_chunks)
  apply clarsimp
  apply(drule (2) sep_free_set_Some2)
  apply simp
done

instance globals_ext_type :: (type) heap_state_typ' ..

defs (overloaded)
  hs_mem_globals [simp]: hs_mem_' (s::('g globals_ext_type)) ≡ t_h_' s
  hs_mem_update_globals [simp]: hs_mem_'_update d (s::'g globals_ext_type) ≡ t_h_'_update d s
  hs_htd_globals [simp]: hs_htd_' (s::('g globals_ext_type)) ≡ t_d_' s
  hs_htd_update_globals [simp]: hs_htd_'_update d (s::'g globals_ext_type) ≡ t_d_'_update d s

instance globals_ext_type :: (type) heap_state_typ
  by intro_classes auto

lemma (in kmemory)
  mem_safe (sep_alloc_invs_body s F) Γ
  apply(unfold sep_alloc_invs_body_def)
  apply(subst mem_safe_restrict)
  apply(rule intra_mem_safe)
  apply(clarsimp simp: whileAnno_def)+
  apply auto
done

lemma sep_cut_ptr_safe:
  [[ (P ^* sep_cut (ptr_val p + i * 4) (x - i*4)) (lift_state (h,d)); i < x div 4;
    KMC udvd x ]] ==> ptr_safe ((p::word32 ptr) +p i) d
  apply(drule sep_conjD, clarsimp simp: ptr_safe_def sep_cut_def sep_cut'_def)
  apply(subgoal_tac {ptr_val p + i * 4..+unat (x - i * 4)} ⊆ dom d)

```

```

apply(subgoal_tac xa ∈ {ptr_val p + i * 4..+unat (x - i * 4)})
  apply(drule (1) subsetD)
  apply(erule domD)
apply(simp add: mult_ac)
apply(drule intvlD, clarsimp)
apply(rule intvlI)
apply(frerule div_lt_mult)
  apply(simp add: word_less_def word_le_def)
apply(subst unat_def)
apply(subst word_arith_alts)
apply(subst int_word_uint)
apply(subst mod_pos_pos_trivial)
  apply(simp add: word_less_def word_le_def)
  apply(insert uint_lt2p [of x])
  apply(insert uint_ge_0 [of i*4])
  apply simp
apply(subst nat_diff_distrib)
  apply(rule uint_ge_0)
  apply(simp add: word_less_def word_le_def)
apply(fold unat_def)
apply(erule less_le_trans)
apply(thin_tac ?P < 2^ ?Q)+
apply(subst (asm) word_less_nat_alt) back
apply(subgoal_tac unat (i * 4) = unat i * 4)
  prefer 2
  apply(frerule order_less_imp_le)
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_eq_norm)
  apply(subst mod_less)
  apply(drule div_lt'')
  apply simp
  apply simp
  apply simp
  apply(subgoal_tac 4 udvd x)
  apply(drule udvd_dvd, simp, clarsimp simp: dvd_def, arith)
  apply(rule_tac b=KMC in udvd_trans)
  apply simp+
apply(clarsimp simp: lift_state_def)
apply(drule_tac x=xaa in fun_cong)
apply(auto split: option.splits)
apply(subst (asm) heap_merge_com)
  apply(simp add: heap_disj_com)
apply(drule sym)
apply simp
apply(drule domD, clarsimp)
done

```

lemma inter\_sub:

$\llbracket G \subseteq A; H \subseteq B; A \cap B = \{\} \rrbracket \implies G \cap H = \{\}$   
 by fast

lemma (in sep\_alloc\_imp)

```

  sep_alloc_spec Γ kmem_free_list_addr
apply (unfold sep_alloc_spec_def sep_alloc_spec_axioms_def)
apply (rule conjI)
  apply(rule kmemory_axioms)
apply (hoare_rule ProcNoRec1)
apply (hoare_rule anno = sep_alloc_invs_body σ F in annotateI)

```

```

prefer 2
apply (subst sep_alloc_invs_body_def)
apply (simp add: whileAnno_def)
apply (subst sep_alloc_invs_body_def)
apply (fold KMC_def max_def)
apply (unfold sep_app_def)
apply vcg

— Pre  $\implies$  I0

apply clarsimp
apply rule
  apply clarsimp
apply rule
  apply (drule sep_free_set_hD)
  apply clarsimp
  apply (rule ptr_aligned_coerceI)
  apply (rule c_guard_ptr_aligned)
  apply (rule sep_map'_g)
  apply (erule sep_map'_conjE2)
  apply (erule sep_map_sep_map')
apply rule
  apply (rule_tac x={ } in exI)
  apply clarsimp
  apply (drule sep_free_set_hD)
  apply clarsimp
  apply (subgoal_tac lift t_h (kmem_free_list_addr) = x)
  apply simp
  apply (rule lift_ptr_coerce')
  apply (rule sep_map'_lift)
  apply (erule sep_map'_conjE2)
  apply (erule sep_map_sep_map')
apply rule
  apply (rule max_size_KMC)
  apply (drule sep_free_set_hD)
  apply clarsimp
  apply (subgoal_tac lift t_h kmem_free_list_addr = x)
  apply simp
  apply (erule sep_map'_conjE2)
  apply (erule sep_map_sep_map')
  apply (rule lift_ptr_coerce')
  apply (rule sep_map'_lift)
  apply (erule sep_map'_conjE2)
  apply (erule sep_map_sep_map')

```

— I<sub>0</sub>  $\implies$  I<sub>1</sub>

```

apply rule
  apply clarsimp
  apply rule
    apply (subst (asm) sep_free_set_singleton)
    apply simp
    apply clarsimp
    apply (drule sep_conjD, clarsimp)+
    apply (drule sep_free_set_singletonD)
    apply clarsimp
    apply (drule sep_free_set_singletonD)
    apply clarify

```

```

    apply(clarsimp dest!: sep_free_set_h_NULL)
  apply clarsimp
  apply clarsimp
  apply(rule c_guard_ptr_aligned)
  apply(rule sep_map'_g)
  apply fast
  apply rule
    apply(clarsimp simp: no_wrap_def)
  apply(rule_tac x=G in exI)
  apply(rule_tac x=H - {ptr_val curr} in exI)
  apply rule
    apply(subst sep_conj_com)
    apply(rule sep_lift_exists)
    apply clarsimp
    apply(erule (1) sep_conj_impl)
    apply(drule (1) sep_free_set_singletonD)
    apply clarsimp
    apply(case_tac r)
    apply clarsimp
    apply fast
  apply(drule sep_conjD, clarsimp)+
  apply(drule (1) sep_free_set_start)
  apply fast
  apply clarsimp
  apply rule
    apply(drule sep_conjD, clarsimp)+
    apply(drule sep_free_set_singletonD)
    apply simp
    apply clarsimp
    apply(rule c_guard_ptr_aligned)
    apply(rule sep_map'_g)
    apply fast
  apply rule
    apply(rule sep_free_set_slide, simp+)
    apply(clarsimp dest!: sep_free_set_h_NULL)
  apply(rule sep_map'_lift_rev)
  apply simp
  apply(erule sep_map'_any_conjE2)
  apply(drule sep_free_set_singletonD)
  apply simp
  apply clarsimp

```

—  $I_1 \implies I_1'$ ,  $I_1 \implies I_0'$  (when a suitable chunk hasn't been found)

```

  apply simp
  apply rule
    apply(clarsimp simp: mult_ac)
    apply(case_tac tmp=curr)
    apply clarsimp
    apply rule
      apply(drule sep_conjD, clarsimp)+
      apply(drule sep_free_set_singletonD)
      apply simp
      apply clarsimp
      apply(rule c_guard_ptr_aligned)
      apply(erule sep_map'_g)
    apply rule
      apply(rule sep_free_set_slide, simp+)

```

```

  apply(clarsimp dest!: sep_free_set_h_NULL)
apply rule
  apply simp
apply(rule sep_map'_lift_rev)
  apply simp
apply(erule sep_map'_any_conjE2)
apply(drule sep_free_set_singletonD)
  apply simp
  apply clarsimp
apply(thin_tac sep_free_set_h ?x ?y ?z ?s)
apply rule
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_singletonD, simp, clarsimp,
    rule c_guard_ptr_aligned, erule sep_map'_g)
apply rule
  prefer 2
  apply(rule, simp)
  apply(rule sep_map'_lift_rev)
  apply simp
  apply(erule sep_map'_any_conjE2)
  apply(erule sep_map'_any_conjE2)
  apply(drule sep_free_set_singletonD)
  apply simp
  apply clarsimp
apply(rule_tac x=G  $\cup$  {ptr_val curr} in exI)
apply(rule_tac x=((chunks curr (ptr_val curr +(i - 1)*KMC)) -
  {ptr_val curr})  $\cup$  H in exI)

apply clarsimp
apply rule
  apply(subst (asm) sep_free_set_singleton) back
  apply simp
  apply clarsimp
  apply(subst (asm) exists_left)
  apply(subst (asm) sep_conj_com)
  apply(subst (asm) sep_conj_assoc)+
  apply(subst (asm) sep_conj_exists)
  apply clarsimp
  apply(subst (asm) sep_conj_com)
  apply(subst (asm) sep_conj_assoc [symmetric])+
  apply(subst (asm) sep_conj_com)
  apply(subst (asm) sep_conj_assoc)+
  apply(frule sep_map'_conjE1)
  apply(clarsimp simp: sep_free_set_block block_def)
  apply fast
  apply(simp add: sep_map'_lift)
  apply(subgoal_tac  $x \notin G$ )
  prefer 2
  apply(drule sep_conjD, clarsimp)+
  apply(frule sep_free_set_start) back back
  apply clarsimp
  apply(frule sep_free_set_start)
  apply(clarify, simp, clarsimp dest!: sep_free_set_h_NULL)
  apply clarsimp
  apply(drule sep_free_set_dom)+
  apply(drule sep_free_set_h_dom)
  apply(unfold heap_disj_def)
  apply simp
  apply(subgoal_tac  $x \in \text{dom } s_0 \wedge x \in \text{dom } s_0'$ )

```

```

    apply blast
  apply blast
  apply clarsimp
  apply(drule sep_free_set_dom)+
  apply(drule sep_free_set_h_dom)
  apply simp
  apply(subgoal_tac x ∈ dom s0 ∧ x ∈ dom s1)
    apply blast
  apply rule
    apply(erule (1) subsetD)
  apply(erule subsetD)
  apply simp
  apply(subgoal_tac (sep_free_set_h kmem_free_list_addr curr G ∧*
    sep_free_set tmp NULL H ∧*
    sep_free_set curr (Ptr x) {ptr_val curr} ∧*
    sep_free_set (Ptr x) tmp
    (chunks curr (ptr_val curr + KMC * (i - 1)) - {ptr_val curr}))
    (lift_state (t_h,t_d)) =
    ((sep_free_set_h kmem_free_list_addr curr G ∧*sep_free_set curr (Ptr x) {ptr_val curr}))
  ∧*(
    sep_free_set tmp NULL H ∧*

    sep_free_set (Ptr x) tmp
    (chunks curr (ptr_val curr + KMC * (i - 1)) - {ptr_val curr})))
    (lift_state (t_h,t_d)))
  prefer 2
  apply simp
  apply(simp only:)
  apply(erule sep_conj_impl)
  apply(drule sep_free_set_h_unionI)
    apply clarsimp
    apply(clarsimp simp: mult_ac)
  apply(thin_tac ?P = True)
  apply(rule sep_free_set_unionI)
  apply(simp add: mult_ac)
  apply(clarsimp simp: mult_ac dest!: sep_free_set_NULL sep_conjD)
  apply(clarsimp simp: mult_ac)
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_start, simp, fast)
  apply rule
  apply rule
  apply(erule exE)+
  apply clarsimp
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_singletonD)
  apply simp
  apply clarsimp
  apply(rule c_guard_ptr_aligned, erule sep_map'_g)
  apply rule
  prefer 2
  apply rule
  apply(thin_tac sep_free_set_h ?x ?y ?z ?s)
  apply clarsimp
  apply(rule_tac x=G in exI)
  apply(rule_tac x=H - {ptr_val tmp} in exI)
  apply clarsimp
  apply rule
  apply(simp add: add_ac mult_ac ring_distrib)

```

```

apply(subst unroll_KMC)
apply(subst chunks_add)
  apply(simp add: mult_ac)
  apply(erule no_wrap)
  apply(rule lt_add_KMC)
  apply(clarsimp simp: unroll_KMC [symmetric])
  apply(case_tac tmp, clarsimp simp: mult_ac)
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_aligned)+
  apply(erule alignedD2, clarsimp simp: no_wrap_def,
    drule chunks_end_KMC, force simp: mult_ac)
  apply(rule udvd_expand)
  apply clarsimp
  apply(rule udvd_KMC)
  apply(simp add: mult_ac no_wrap_def)
apply(subst (asm) sep_free_set_singleton)
  apply simp
  apply clarsimp
  apply(subst (asm) sep_conj_assoc [symmetric])+
  apply(subst (asm) sep_conj_com) back
  apply(subst (asm) sep_conj_exists)
  apply clarsimp
  apply(subgoal_tac x  $\notin$  chunks curr (ptr_val curr + KMC * (i - 1)))
  prefer 2
  apply(drule sep_conjD, clarsimp)+
  apply(frule sep_free_set_start) back back
  apply clarify
  apply(clarsimp dest!: sep_free_set_NULL)
  apply(drule sep_free_set_dom)+
  apply(subgoal_tac x  $\in$  dom s0'  $\wedge$  x  $\in$  dom s1)
  apply(unfold heap_disj_def, simp)
  apply blast
  apply(thin_tac ?P = {})+
  apply(rule, blast)
  apply simp
  apply blast
  apply(subst (asm) sep_conj_com)
  apply(subst (asm) sep_conj_assoc)+
  apply(frule sep_map'_conjE1)
  apply(case_tac tmp)
  apply(clarsimp simp: sep_free_set_block' block_def)
  apply fast
  apply(case_tac tmp, clarsimp)
  apply(simp add: sep_map'_lift)
  apply(erule (1) sep_conj_impl)
  apply(erule (1) sep_conj_impl)
  apply simp
  apply(subst (asm) sep_conj_com)
  apply(drule sep_free_set_unionI)
  apply clarsimp
  apply(clarsimp simp: mult_ac unroll_KMC)
  apply(subgoal_tac chunks curr (ptr_val curr + i*KMC) =
    chunks curr (ptr_val curr + (i - 1)*KMC)  $\cup$  {ptr_val curr + i*KMC})
  apply(clarsimp simp: mult_ac)
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_start)
  apply simp
  apply clarsimp

```

```

    apply fast
  apply(subst unroll_KMC)
  apply(subst chunks_add)
    apply(clarsimp simp: no_wrap_def)
  apply(rule lt_add_KMC)
    apply(subst unroll_KMC [symmetric])
  apply(case_tac tmp, clarsimp simp: mult_ac)
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_aligned)+
  apply(erule alignedD2, clarsimp simp: no_wrap_def,
    drule chunks_end_KMC, force simp: mult_ac)
  apply(rule udvd_expand)
  apply clarsimp
  apply(rule udvd_KMC)
  apply(clarsimp simp: no_wrap_def)
  apply(subst unroll_KMC [symmetric])
  apply simp
prefer 2
  apply(thin_tac sep_free_set_h ?x ?y ?z ?s)
  apply clarsimp
  apply(frule no_wrapD)
  apply(simp (no_asm) only: no_wrap_def)
  apply(erule order_trans)
  apply(subst unroll_KMC) back
  apply(simp add: mult_ac)
  apply(rule order_less_imp_le)
  apply(rule lt_add_KMC)
  apply(subst mult_ac)
  apply(subst unroll_KMC [symmetric])
  apply(case_tac tmp, clarsimp)
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_aligned)+
  apply(erule alignedD2, frule no_wrapD,
    drule chunks_end_KMC, force simp: mult_ac)
  apply rule
  prefer 2
  apply(erule (1) inc_i)
  apply clarsimp
  apply(thin_tac sep_free_set_h ?x ?y ?z ?s)
  apply(rule_tac x=G in exI)
  apply(rule_tac x=chunks curr (ptr_val curr + (i - 1)*KMC) ∪ H in exI)
  apply rule
  apply(erule (1) sep_conj_impl)
  apply(rule sep_free_set_unionI)
  apply simp
  apply(clarsimp dest!: sep_conjD sep_free_set_NULL)
  apply blast

```

—  $I_1 \implies I_2, I_1 \implies I_0'$

```

  apply rule
  apply(thin_tac sep_free_set_h ?x ?y ?z ?s)
  apply simp
  apply rule
  apply rule
  apply clarsimp
  apply rule
  apply simp

```

```

apply(erule exE | erule conjE)+
apply clarsimp
apply(rule c_guard_ptr_aligned, erule sep_map'_g)
apply rule
  apply simp
  apply(erule exE | erule conjE)+
  apply(clarsimp simp: mult_ac)
  apply(erule sep_map'_ptr_safe)
apply rule
  apply simp
  apply(erule exE | erule conjE)+
  apply(clarsimp simp: mult_ac)
  apply(subgoal_tac alloc (ptr_coerce curr) (max size KMC)
    ((G  $\cup$  chunks curr (ptr_val curr + KMC *
      (max size KMC div KMC - 1))  $\cup$  H)) = (G  $\cup$  H))
  apply simp
  apply(erule disjE)
  apply clarsimp
  apply(subst sep_free_set_h_def)
  apply(subst sep_conj_com)
  apply(subst sep_conj_exists)
  apply(rule_tac x=tmp in exI)
  apply(subst sep_conj_assoc)+
  apply(rule sep_heap_update_global', simp)
  apply(frule (1) sep_free_set_h_empty_sep_map')
  apply simp
  apply(subst (asm) sep_conj_com)
  apply(subst (asm) sep_conj_assoc)+
  apply(erule sep_conj_impl)
  apply(erule sep_map_tagd)
  apply simp
  apply(erule (1) sep_conj_impl)
  apply(subgoal_tac KMC * (max size KMC div KMC - 1) = max size KMC - KMC)
  apply clarsimp
  apply(erule sep_free_set_cut)
  apply(simp add: no_wrap_def mult_ac)
  apply(drule div_le_mult)
  apply simp+
  apply(erule KMC_max)
  apply(subst right_diff_distrib)
  apply clarsimp
  apply(rule max_KMC_div_times)
  apply simp
  apply(subst sep_free_set_h_split_elem_eq)
  apply fast
  apply(rule ccontr)
  apply clarify
  apply(clarsimp dest!: sep_free_set_h_NULL sep_conjD)
  apply(clarsimp dest!: sep_free_set_h_NULL sep_free_set_NULL sep_conjD)
  apply(subst exists_left)
  apply(subst sep_conj_exists)+
  apply clarsimp
  apply(rule_tac x=G - {ptr_val prev} in exI)
  apply(rule_tac x=H in exI)
  apply clarsimp
  apply rule
  apply fast
  apply(rule_tac x=ptr_val tmp in exI)

```

```

apply clarsimp
apply rule
  apply clarsimp
  apply(drule sep_conjD, clarsimp)+
  apply(frule sep_free_set_start)
  apply clarsimp
  apply(drule sep_free_set_h_dom)
  apply(drule sep_free_set_dom)+
  apply(unfold heap_disj_def, simp)
  apply(subgoal_tac ptr_val tmp ∈ dom s₀' ∧ ptr_val tmp ∈ dom s₀)
  apply blast
  apply(thin_tac ?P = {})+
  apply(rule, blast)
  apply blast
  apply(rule sep_heap_update_block)
apply clarsimp
apply(subst (asm) sep_conj_com)
apply(subst (asm) sep_conj_assoc)+
apply(frule (1) sep_free_set_h_prev)
  apply simp
  apply(subst sep_conj_com)
  apply fast
  apply clarsimp
  apply(drule sep_conjD, clarsimp)+
  apply(frule sep_free_set_start) back
  apply clarsimp
  apply(frule sep_free_set_start)
  apply clarsimp
  apply(drule sep_free_set_dom)+
  apply(drule sep_free_set_h_dom)+
  apply(unfold heap_disj_def, clarsimp)
  apply(subgoal_tac ptr_val tmp ∈ dom s₀ ∧ ptr_val tmp ∈ dom s₀')
  apply blast
  apply blast
  apply(drule sep_free_set_dom)+
  apply(drule sep_free_set_h_dom)+
  apply clarsimp
  apply(subgoal_tac ptr_val prev ∈ dom s₀ ∧ ptr_val prev ∈ dom s₁')
  apply blast
  apply blast
  apply clarsimp
  apply(drule sep_conjD, clarsimp)+
  apply(frule sep_free_set_start) back
  apply clarsimp
  apply(frule sep_free_set_start)
  apply clarsimp
  apply(drule sep_free_set_dom)+
  apply(drule sep_free_set_h_dom)+
  apply(unfold heap_disj_def, clarsimp)
  apply(subgoal_tac ptr_val tmp ∈ dom s₀ ∧ ptr_val tmp ∈ dom s₀')
  apply blast
  apply blast
  apply(drule sep_free_set_dom)+
  apply(drule sep_free_set_h_dom)+
  apply clarsimp
  apply(subgoal_tac ptr_val curr ∈ dom s₀ ∧ ptr_val curr ∈ dom s₁')
  apply blast
  apply blast

```

```

apply simp
apply(thin_tac (?P  $\wedge$ * ?Q) ?R)
apply(subst (asm) sep_conj_com)
apply(subst (asm) sep_conj_assoc)+
apply(erule (1) sep_conj_impl)
apply simp
apply(subst (asm) sep_conj_assoc [symmetric])+
apply(subst (asm) sep_conj_com) back
apply(erule sep_conj_impl)
apply(subgoal_tac KMC * (max size KMC div KMC - 1) = max size KMC - KMC)
  apply clarsimp
  apply(erule sep_free_set_cut)
    apply(simp add: no_wrap_def mult_ac)
    apply(drule div_le_mult)
      apply simp+
      apply(erule KMC_max)
  apply(subst right_diff_distrib)
  apply clarsimp
  apply(rule max_KMC_div_times)
  apply simp
  apply clarsimp
  apply(subst sep_conj_com)
  apply(erule (1) sep_conj_impl)
  apply simp
  apply(clarsimp simp: alloc_def)
  apply(subgoal_tac KMC * (max size KMC div KMC - 1) = max size KMC - KMC)
  apply clarsimp
  apply(subgoal_tac chunks curr (ptr_val curr + (max size KMC - KMC))  $\cap$ 
    insert (ptr_val prev) (G  $\cup$  H) = {})
  apply fast
  apply clarsimp
  apply rule
  apply clarsimp
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_dom)+
  apply(unfold heap_disj_def, simp)
  apply(subgoal_tac ptr_val prev  $\in$  dom s0)
  prefer 2
  apply(drule sep_free_set_h_dom')
  apply(erule disjE, clarsimp)
  apply(fast dest: subsetD)
  apply(subgoal_tac ptr_val prev  $\in$  dom s1')
  apply blast
  apply blast
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_dom)+
  apply(simp add: heap_disj_def)
  apply(subgoal_tac chunks curr (ptr_val curr + (max size KMC - KMC))  $\cap$  G =
    {})
  prefer 2
  apply(rule inter_sub)
  apply assumption
  apply(drule sep_free_set_h_dom)
  apply assumption
  apply fast
  apply(subgoal_tac chunks curr (ptr_val curr + (max size KMC - KMC))  $\cap$  H =
    {})
  prefer 2

```

```

    apply(rule inter_sub)
      apply assumption+
      apply fast
      apply fast
    apply(subst right_diff_distrib)
    apply clarsimp
    apply(rule max_KMC_div_times)
    apply simp
  apply rule
    apply(erule KMC_max)
  apply rule
    apply simp
    apply(erule exE)+
    apply(erule conjE)+
    apply(drule sep_conjD, clarsimp)+
    apply(drule sep_free_set_aligned)+
    apply(erule alignedD2, rule chunks_start, simp add: no_wrap_def, simp)
  apply(clarsimp simp: no_wrap_def mult_ac)
  apply(subgoal_tac KMC * (max size KMC div KMC - 1) = max size KMC - KMC)
    apply clarsimp
    apply(subst right_diff_distrib)
    apply clarsimp
    apply(rule max_KMC_div_times)
    apply simp
  apply rule
  apply simp
  apply rule
    apply(erule exE | erule conjE)+
    apply(drule sep_conjD, clarsimp)+
    apply(drule sep_free_set_singletonD)
    apply simp
    apply clarsimp
    apply(rule c_guard_ptr_aligned, erule sep_map'_g)
  apply rule
    apply clarsimp
    apply(rule sep_free_set_slide, simp+)
    apply(clarsimp dest!: sep_free_set_h_NULL sep_conjD)
  apply rule
    apply simp
  apply(rule sep_map'_lift_rev)
    apply simp
  apply(erule exE)+
  apply clarsimp
  apply(erule sep_map'_any_conjE2)
  apply(drule sep_free_set_singletonD)
    apply simp
  apply clarsimp

```

—  $I_2 \implies I_2'$

```

  apply clarsimp
  apply rule
    apply(subst (asm) sep_conj_assoc [symmetric])+
    apply(erule (2) sep_cut_ptr_safe)
  apply rule
    apply(erule (1) index_not_NULL)
      apply simp
    apply clarsimp

```

```

apply assumption
apply rule
  apply(rule ptr_aligned_plus)
  apply(drule sep_conjD, clarsimp)+
  apply(case_tac i=0)
  apply clarsimp
  apply(erule ptr_aligned_KMC)
  apply(erule zero_block_ptr_aligned)
  apply clarsimp
  apply(subst (asm) word_less_def, subst (asm) word_le_nat_alt,
    clarify, subst (asm) word_unat_Rep_inject [symmetric], force)
apply(subst zero_block_unroll)
  apply(simp)
apply simp
apply(subst sep_conj_assoc [symmetric])+
apply(subst sep_conj_com)
apply(subst sep_conj_assoc)+
apply rule
  apply(rule sep_heap_update_global')
  apply(rule ptr_tag_sep_cut')
  apply simp
  apply(erule (1) sep_conj_impl)
  apply(erule (1) sep_conj_impl)
  apply(drule_tac x=4 in sep_cut_split)
  prefer 2
  apply(clarsimp simp: ring_eq_simps)
  apply(erule sep_conj_impl)
  apply(simp add: sep_cut_def)
  apply assumption
  apply(frule div_lt_mult)
  apply(simp add: word_less_def word_le_def)
  apply(drule inc_le)
  apply(drule div_le_mult)
  apply(simp add: word_less_def word_le_def)
  apply(drule div_le_mult)
  apply(simp add: word_less_def word_le_def)
  apply(simp)
  apply(erule (1) le_minus)
  apply(rule_tac a=0 and s=max size KMC in udvd_incr2)
  apply simp
  apply(simp add: word_le_nat_alt)
  apply(rule_tac b=KMC in udvd_trans)
  apply simp+
  apply(rule udvd_4)
  apply(simp add: word_le_nat_alt)
  apply(simp add: word_less_def word_le_def)
  apply(unfold c_guard_def)
  apply rule
  apply(rule ptr_aligned_plus)
  apply(drule sep_conjD, clarsimp)+
  apply(case_tac i=0)
  apply clarsimp
  apply(erule ptr_aligned_KMC)
  apply(erule zero_block_ptr_aligned)
  apply clarsimp
  apply(subst (asm) word_less_def, subst (asm) word_le_nat_alt,
    clarify, subst (asm) word_unat_Rep_inject [symmetric], force)
  apply(erule (1) index_not_NULL)

```

```

    apply simp
    apply clarsimp
    apply assumption
    apply(erule inc_le)

```

—  $I_2 \implies \text{Post}$

```

prefer 2
  apply clarsimp
  apply clarsimp
  apply(subst (asm) max_KMC_div_times')
  apply(clarsimp simp: max_def split: split_if_asm)
  apply(simp add: KMC_def udvd_def)
  apply(subst (asm) max_KMC_div_times')
  apply(clarsimp simp: max_def split: split_if_asm)
  apply(simp add: KMC_def udvd_def)
  apply simp
done

end

```

## 7 kfree without separation logic

```

theory kfree imports non_sep_common sep_common begin

```

```

lemma free_set_heapE:
  [[ free_set h' s e F; h|(Ptr'F) = h'|'(Ptr'F) ]]  $\implies$  free_set h s e F
  apply (subst free_set_restrict_dom [symmetric])
  apply (simp add: free_set_restrict_dom)
  done

```

```

lemma chunks_intvl_disjoint:
  assumes p1:  $p < p + x$ 
  assumes p2:  $p + x \leq p + x + \text{of\_nat } y$ 
  assumes y:  $y < 2^{\text{flen } p}$ 
  shows  $\text{chunks } a \ p \cap \{p + x..+y\} = \{\}$ 
proof -
  { fix z
    assume  $z \in \{p + x ..+ y\}$ 
    then obtain k where  $z: z = p + x + \text{of\_nat } k$  and  $k: k < y$ 
      by (auto simp: intvl_def)
    from k y
    have  $\text{of\_nat } k \leq (\text{of\_nat } y :: \text{word32})$ 
      by (simp add: word_le_nat_alt unat_of_nat flen_def)
    with p2 z
    have  $p + x \leq z$  by (auto intro: word_plus_mono_right2)
    with p1 have  $p < z$  by simp
    hence  $z \notin \text{chunks } a \ p$  by (simp add: chunks_def linorder_not_less [symmetric])
  }
  thus ?thesis by blast
qed

```

```

lemma intvl_nowrap:
  assumes x:  $x \leq x + \text{of\_nat } y$ 
  assumes y:  $y < 2^{\text{flen } x}$ 
  shows  $\{x ..+ y\} \subseteq \{x ..< x + \text{of\_nat } y\}$ 

```

```

apply (insert prems)
apply (clarsimp simp add: intvl_def)
apply (rule conjI)
  apply (erule word_plus_mono_right2)
  apply (simp add: word_le_nat_alt unat_of_nat flen_def)
apply (simp (no_asm) add: order_less_le)
apply (rule conjI)
  apply (rule word_plus_mono_right)
  apply (simp add: word_le_nat_alt unat_of_nat flen_def)
  apply assumption
apply clarsimp
apply (subst (asm) word_unat_Rep_inject [symmetric])
apply (simp add: unat_of_nat flen_def)
done

```

lemma in\_chunks:

```

[[ ptr_val a ≤ ptr_val a + s; KMC udvd s; KMC ≤ s ]] ⇒
ptr_val a + s - KMC ∈ chunks a (ptr_val a + s)
apply (subgoal_tac ptr_val a ≤ ptr_val a + (s - KMC))
  prefer 2
  apply (subgoal_tac uint (ptr_val a) + uint s - uint KMC < 2^32)
    apply (simp add: no_plus_overflow_uint)
    apply (simp add: uint_sub_if word_le_def flen_def)
  apply (subgoal_tac uint (ptr_val a) + uint s < 2^32)
    apply (simp add: KMC_def)
    apply (simp add: no_plus_overflow_uint flen_def)
  apply (simp add: chunks_def)
  apply (rule conjI)
  apply (simp add: add_ac word_sub_def)
  apply (rule conjI)
  apply (subgoal_tac ptr_val a + (s - KMC) < ptr_val a + s)
    apply (simp add: add_ac word_sub_def)
  apply (subst plus_le_left_cancel_nowrap)
    apply assumption
    apply assumption
  apply (simp add: word_le_def word_less_alt uint_sub_if)
  apply (simp add: KMC_def)
  apply (clarsimp simp add: udvd_def)
  apply (rule_tac x=n - 1 in exI)
  apply simp
  apply (case_tac n=0)
    apply (simp add: word_le_def KMC_def)
  apply (simp add: ring_distrib)
  apply (subgoal_tac uint (ptr_val a + (s - KMC)) = uint (ptr_val a) + (n - 1) * uint KMC)
    apply (simp add: add_ac word_sub_def)
  apply (simp add: uint_plus_simple)
  apply (simp add: word_le_def uint_sub_if)
  apply (simp add: int_distrib)
done

```

lemma udvd\_less\_le: [[ a ≤ p; k udvd p - a; p ≠ a ]] ⇒ k ≤ p

```

apply (clarsimp simp add: udvd_def)
apply (simp add: word_le_def uint_sub_if word_less_alt)
apply (case_tac n = 0)
  apply simp
  apply (subgoal_tac uint k ≤ n * uint k)
    apply (insert uint_range [of a])

```

```

    apply arith
  apply (simp add: mult_compare_simps)
done

lemma chunks_add_udvd3:
  [[ ptr_val a ≤ p; KMC udvd p - ptr_val a ]] ==>
  chunks a p = (if p = ptr_val a then {p} else chunks a (p-KMC) ∪ {p})
  apply clarsimp
  apply (subgoal_tac KMC ≤ p)
    apply (drule (2) chunks_add_udvd2)
    apply simp
  apply (erule (2) udvd_less_le)
done

lemmas free_set_insertI = free_set_unionI [where ?F1.0={p}, simplified, standard]

lemma sub_wrap [simp]:
  fixes x :: 'a::fl word
  shows (x ≤ x - z) = (z = 0 ∨ x < z)
  apply (clarsimp simp: word_le_def word_less_alt uint_sub_if uint_0_iff)
  apply (insert uint_range [of z])
  apply (unfold flen_def)
  apply arith
done

lemma chunks_decr_end:
  k ≤ p ==> chunks a (p - k) ⊆ chunks a p
  apply (clarsimp simp add: chunks_def)
  apply (rule conjI)
    prefer 2
    apply (rule_tac x=n in exI)
    apply simp
  apply (simp add: word_le_def uint_sub_if)
  apply (insert uint_range [of k])
  apply arith
done

lemma free_set_heapupdate_ignore:
  ptr_val x ∉ F ==> free_set (h(x ↦ v)) s e F = free_set h s e F
  by (cases x) (auto elim: free_set_heapE)

lemma udvd_expand_iff:
  a ≤ p ==> K udvd p - a = (∃n. 0 ≤ n ∧ uint p = uint a + n * uint K)
  by (auto simp add: udvd_def word_le_def uint_sub_if intro!: udvd_expand)

lemma udvd_decr:
  fixes p :: 'a::fl word
  assumes less: p < q
  assumes 1: uint p = uint a + n * uint K
  assumes 2: uint q = uint a + n' * uint K
  assumes nowrap: K ≤ q
  shows p ≤ q - K
proof -
  from less 1 2
  have uint a + n * uint K < uint a + n' * uint K
    by (simp add: word_less_alt)

```

```

hence n * uint K < n' * uint K by simp
hence n < n' using uint_ge_0 [of K]
  by (simp add: mult_less_cancel_right)
hence n ≤ n' - 1 by simp
hence n * uint K ≤ (n' - 1) * uint K
  by (simp add: mult_le_cancel_right)
hence uint a + n * uint K ≤ uint a + n' * uint K - uint K
  by (simp add: left_diff_distrib)
thus ?thesis using 1 2 nowrap
  by (simp add: word_le_def uint_sub_if)
qed

```

lemma chunks\_add\_end:

```

assumes le: ptr_val a + s - KMC ≤ p
assumes gt: p < ptr_val a + s
assumes ap: ptr_val a ≤ p
assumes kp: KMC ≤ p
assumes dvdp: KMC udvd p - ptr_val a
assumes dvds: KMC udvd s
shows chunks a (ptr_val a + (s - KMC)) = insert p (chunks a (p - KMC))

```

proof -

```

from gt ap have [simp]: ptr_val a ≤ ptr_val a + s by simp
from kp gt have [simp]: KMC ≤ ptr_val a + s by simp
have chunks a (ptr_val a + (s - KMC)) = chunks a p (is ?chunks_a = ?chunks_p)

```

proof

```

{ fix x
  assume x ∈ ?chunks_a
  then obtain n where chunks_a:
    ptr_val a ≤ x
    x ≤ ptr_val a + (s - KMC)
    0 ≤ n
    uint x = uint (ptr_val a) + n * uint KMC
    by (auto simp: chunks_def)
  hence x ≤ ptr_val a + s - KMC
    by (simp add: word_sub_def add_ac)
  with le have x ≤ p by simp
  with chunks_a
  have x ∈ ?chunks_p by (auto simp: chunks_def)
}

```

thus ?chunks\_a ⊆ ?chunks\_p by blast

next

```

{ fix x
  assume x ∈ ?chunks_p
  then obtain n where chunks_p:
    ptr_val a ≤ x
    x ≤ p
    0 ≤ n
    uint x = uint (ptr_val a) + n * uint KMC
    by (auto simp: chunks_def)
  with gt
  have x < ptr_val a + s by simp
  moreover
  from dvds
  obtain n' where uint s = n' * uint KMC
    by (auto simp add: udvd_def)
  ultimately
  have x ≤ ptr_val a + s - KMC using chunks_p
    by (auto simp add: uint_plus_simple intro: udvd_decr)
}

```

```

    hence x ≤ ptr_val a + (s - KMC)
      by (simp add: word_sub_def add_ac)
    with chunks_p
    have x ∈ ?chunks_a by (auto simp: chunks_def)
  }
  thus ?chunks_p ⊆ ?chunks_a by blast
qed
with ap kp dvdp show ?thesis by (simp add: chunks_add_udvd2)
qed

```

```

lemma ptr_tag_h_t_valid_diff:
  fixes p :: 'a::mem_type ptr
  fixes q :: 'b::c_type ptr
  shows [| d,g' ⊢t p; d,g ⊢t (ptr_coerce q::'c::mem_type ptr); ptr_val p ≠ ptr_val q;
    ptr_val p ∉ {ptr_val q+1..+size_of TYPE('b) - 1}|] ⇒
    ptr_tag q d,g' ⊢t p
  apply (simp add: h_t_valid_def ptr_tag_def ptr_set_def
    ptr_clear_def valid_footprint_def)
  apply clarsimp
  done

```

```

lemma intvl_plus:
  x ≠ y ⇒ (x ∈ {y+1..+n}) = (x ∈ {y..+ n+1})
  apply (simp add: intvl_def)
  apply (rule iffI)
  apply clarsimp
  apply (rule_tac x=k+1 in exI)
  apply clarsimp
  apply clarsimp
  apply (rule_tac x=k - 1 in exI)
  apply simp
  apply (case_tac k)
  apply simp
  apply simp
  done

```

```

lemma ptr_tag_h_t_valid_pq:
  fixes p :: 'a::mem_type ptr
  fixes q :: 'b::mem_type ptr
  shows [| d,g' ⊢t p; ptr_val p ≠ ptr_val q;
    ptr_val p ∉ {ptr_val q+1..+size_of TYPE('b) - 1};
    ptr_val q ∉ {ptr_val p+1..+size_of TYPE('a) - 1}|] ⇒
    ptr_tag q d,g' ⊢t p
  apply (simp add: h_t_valid_def ptr_tag_def ptr_set_def
    ptr_clear_def valid_footprint_def)
  done

```

```

lemma ptr_tag_h_t_valid_diff2:
  fixes p :: 'a::mem_type ptr
  fixes q :: 'b::mem_type ptr
  shows [| ptr_tag q d,g' ⊢t p; ptr_val p ≠ ptr_val q;
    ptr_val p ∉ {ptr_val q + 1..+size_of TYPE('b) - Suc 0}|] ⇒
    d,g' ⊢t p
  apply (simp add: ptr_tag_def ptr_set_def ptr_clear_def)

```

```

apply (clarsimp simp add: h_t_valid_def)
apply (clarsimp simp add: valid_footprint_def)
apply (simp split: split_if_asm)
apply (frule_tac x=y in spec)
apply (erule disjE)
  prefer 2
  apply clarsimp
apply (subgoal_tac ptr_val q ∈ {ptr_val p+1 ..+size_of TYPE('a) - Suc 0})
  apply clarsimp
  apply (drule_tac x=ptr_val q in spec)
  apply clarsimp
apply (thin_tac ∀x. ?P x)
apply (case_tac y = ptr_val p)
  apply simp
apply (case_tac y = ptr_val q)
  apply simp
apply (simp add: intvl_plus)
apply (cases p, cases q)
apply (rename_tac w1 w2)
apply clarsimp
apply (clarsimp simp add: intvl_def)
apply (rename_tac k1 k2)
apply (rule_tac x=k1 - k2 in exI)
apply (rule conjI)
  prefer 2
  apply arith
apply (subgoal_tac w2 = w1 + of_nat k1 - of_nat k2)
  prefer 2
  apply simp
apply hypsubst
apply simp
apply (fold word_sub_def)
apply (case_tac k2 ≤ k1)
  apply simp
apply (simp add: linorder_not_le)
apply (erule_tac x=k2 - k1 in allE)
apply simp
done

```

```

lemma ptr_tag_h_t_valid_diff_s:
  fixes p :: 'a::mem_type ptr
  fixes q :: 'b::c_type ptr
  shows [| (lift_state (h,d)),g ⊢s (ptr_coerce q::'c::mem_type ptr);
          (lift_state (h,d)),g' ⊢s p;
          ptr_val p ≠ ptr_val q;
          ptr_val p ∉ {ptr_val q+1..+size_of TYPE('b) - 1}] ==>
    lift_state (h,ptr_tag q d),g' ⊢s p
  by (auto simp: h_t_s_valid intro: ptr_tag_h_t_valid_diff)

```

```

lemma liftτ_ptr_tagI:
  fixes p :: 'a::mem_type ptr
  fixes q :: 'b::c_type ptr
  shows
  [| liftτ g (h,d) p = Some v;
    d ⊢t (ptr_coerce q::'c::mem_type ptr);
    ptr_val p ≠ ptr_val q;
    ptr_val p ∉ {ptr_val q+1..+size_of TYPE('b) - 1}
  |]

```

```

] ==>
liftτ g (h, (ptr_tag q d)) p = Some v
by (auto simp: liftτ_if intro: ptr_tag_h_t_valid_diff split: split_if_asm)

lemma liftτ_ptr_tag_eq:
  fixes p :: 'a::mem_type ptr
  fixes q :: 'b::mem_type ptr
  shows
  [[ d ⊨t (ptr_coerce q::'c::mem_type ptr);
    ptr_val p ≠ ptr_val q;
    ptr_val p ∉ {ptr_val q+1..+size_of TYPE('b) - 1} ]] ==>
  liftτ g (h, (ptr_tag q d)) p = liftτ g (h, d) p
  by (fastsimp simp: liftτ_if dest: ptr_tag_h_t_valid_diff2 ptr_tag_h_t_valid_diff)

lemma liftτ_ptr_tag_eq2:
  fixes p :: 'a::mem_type ptr
  fixes q :: 'b::mem_type ptr
  shows
  [[ ptr_val p ≠ ptr_val q;
    ptr_val p ∉ {ptr_val q+1..+size_of TYPE('b) - 1};
    ptr_val q ∉ {ptr_val p+1..+size_of TYPE('a) - 1} ]] ==>
  liftτ g (h, (ptr_tag q d)) p = liftτ g (h, d) p
  by (fastsimp simp: liftτ_if dest: ptr_tag_h_t_valid_diff2 ptr_tag_h_t_valid_pq)

lemma liftτc_guarded:
  liftτc h (p::'a::mem_type ptr) = Some v ==> p ≠ NULL ∧ ptr_aligned p
  apply (case_tac h, simp)
  apply (drule liftτ_h_t_valid)
  apply (auto simp: h_t_valid_c_guard c_guard_ptr_aligned c_guard_NULL)
  done

lemma c_guard_coerce_NULL:
  d ⊨t ptr_coerce p ==> p ≠ NULL
  by (simp add: h_t_valid_def c_guard_def)

lemmas guard_simps =
  liftτc_guarded h_t_valid_c_guard
  c_guard_NULL c_guard_ptr_aligned c_guard_coerce_NULL

lemma intvl_sub_plus:
  x ∈ {p ..+ q} ==> x ∈ {p - 1 ..+ q + 1}
  by (rule intvl_plus_sub_Suc) simp

lemma free_set_singletonI:
  [[d ⊨t p; h p = Some (ptr_val q); ptr_val p ≠ ptr_val q; p' = ptr_val p]]
  ==> free_set h p q {p'}
  apply (simp add: free_set_def)
  apply (rule_tac x=[ptr_val p] in exI)
  apply auto
  done

lemma chunks_next:
  [[ p ∈ chunks a b; p + KMC ≤ b; p ≤ p + KMC ]] ==> p + KMC ∈ chunks a b
  apply (clarsimp simp add: chunks_def)
  apply (rule conjI)

```

```

    apply (erule (1) order_trans)
  apply (rule_tac x=n+1 in exI)
  apply (simp add: ring_distrib uint_plus_simple)
done

lemma ptr_tag_restrict:
  fixes p :: 'a::mem_type ptr
  shows ptr_tag p d |' (-{ptr_val p ..+ size_of TYPE('a)}) = d |' (-{ptr_val p ..+ size_of TYPE('a')})
  apply (rule ext)
  apply (simp add: ptr_tag_def ptr_set_def ptr_clear_def restrict_map_def)
  apply clarsimp
  apply (fast dest: intvl_plus_sub_Suc)
done

lemma restrict_map_mono [trans]:
  [[ f |' A = g |' A; B  $\subseteq$  A ]  $\implies$  f |' B = g |' B
  apply (rule ext)
  apply (simp add: restrict_map_def)
  apply clarsimp
  apply (drule (1) subsetD)
  apply (drule_tac x=x in fun_cong)
  apply simp
done

lemma KMC_subset:
  x  $\in$  {p..+4}  $\implies$  x  $\in$  {p..+unat KMC}
  apply (clarsimp simp add: intvl_def KMC_def)
  apply (rule_tac x=k in exI)
  apply simp
done

lemma ptr_tags_restrict:
   $\bigwedge$ p d. ptr_tags n p d |' (-{p ..+ n * unat KMC}) = d |' (-{p ..+ n * unat KMC})
proof (induct n)
  case 0
  show ?case
  by (fastsimp intro: restrict_map_mono ptr_tag_restrict simp: KMC_subset)
next
  case (Suc n)
  hence ptr_tags n (p+KMC) d |' (- {p+KMC..+n * unat KMC}) = d |' (- {p+KMC..+n * unat KMC}) by blast
  also
  have -{p..+unat KMC + (n * unat KMC)}  $\subseteq$  -{p+KMC..+n * unat KMC}
  apply (clarsimp simp add: intvl_def)
  apply (rule_tac x=k + unat KMC in exI)
  apply simp
  done
  finally
  have pK: ptr_tags n (p + KMC) d |' (- {p..+unat KMC + (n * unat KMC)}) = d |' (- {p..+unat KMC + (n * unat KMC)})
  .

  from ptr_tag_restrict [of (Ptr p::word32 ptr)]
  have ptr_tag (Ptr p::word32 ptr) (ptr_tags n (p + KMC) d) |' (-{p ..+ 4}) = (ptr_tags n (p +
KMC) d) |' (-{p ..+ 4})
  by simp
  also
  have -{p..+unat KMC + n * unat KMC}  $\subseteq$  -{p ..+ 4}
  apply (clarsimp simp: intvl_def KMC_def)

```

```

    apply (rule_tac x=k in exI)
    apply (rule conjI)
      apply simp
      apply (rule trans_less_add1)
      apply simp
    done
  finally
  show ?case using pK by simp
qed

lemma restrict_apply [trans]:
  [[ f |' A = g |' A; x ∈ A ]] ⇒ f x = g x
  apply (simp add: restrict_map_def)
  apply (drule_tac x = x in fun_cong)
  apply simp
  done

lemma ptr_tags_valid:
  fixes q :: 'a::mem_type ptr
  assumes q: d,g ⊨t q
  assumes intvl: {ptr_val q ..+ size_of TYPE('a)} ⊆ -{p ..+ n * unat KMC}
  shows ptr_tags n p d,g ⊨t q
proof -
  from q
  have foot: valid_footprint d (ptr_val (q::'a ptr)) (typ_tag TYPE('a)) (size_of TYPE('a))
    and d: d (ptr_val q) = Some (Some (typ_tag TYPE('a)))
    and g: g q
    by (auto simp: h_t_valid_def valid_footprint_def)

  have r:
    ptr_tags n p d |' (-{p ..+ n * unat KMC}) = d |' (-{p ..+ n * unat KMC})
    by (rule ptr_tags_restrict)

  with foot intvl
  have (∀y. y ∈ {ptr_val q + 1..+size_of TYPE('a) - Suc 0} →
    ptr_tags n p d y = Some None)
    apply (simp add: valid_footprint_def)
    apply clarsimp
    apply (erule_tac x=y in alle)
    apply clarsimp
    apply (drule_tac x=y in restrict_apply)
      apply (drule intvl_plus_sub_Suc)
      apply (erule (1) subsetD)
    apply simp
  done
  moreover {
    note r
    also
    have ptr_val q ∈ {ptr_val q ..+ size_of TYPE('a)}
      by (rule intvl_self, rule sz_nzero)
    with intvl
    have ptr_val q ∈ -{p ..+ n * unat KMC} by blast
    finally
    have ptr_tags n p d (ptr_val q) = Some (Some (typ_tag TYPE('a)))
      using d by simp
  }
  ultimately
  show ?thesis using g by (auto simp: h_t_valid_def valid_footprint_def)

```

qed

lemma word\_uminus0: - (x::'a::fl word) = 0 - x by simp

lemma unat\_uminus:

```
0 < x  $\implies$  unat (-x::'a::fl word) = 2flen x - unat x
apply (subst word_uminus0)
apply (subst unat_sub_if)
apply (clarsimp simp: flen_def unat_0_iff)
done
```

lemma unat\_uminus1:

```
unat (-1::'a::fl word) = 2flen of TYPE('a) - 1
```

proof -

```
have uminus: (-1::'a::fl word) = 0 - 1 by simp
show ?thesis
  apply (subst uminus)
  apply (subst unat_sub_if)
  apply (clarsimp simp: flen_def)
done
```

qed

lemma word\_cases [cases type]:

```
fixes x :: 'a :: fl word
shows  $\llbracket P\ 0; \bigwedge x. 0 < x \implies P\ x \rrbracket \implies P\ x$ 
by (cases x = 0) auto
```

lemma intvl\_shift:

```
fixes x :: 'a :: fl word
shows  $\llbracket 0 < y; y < 2^{\text{flen}}\ x \rrbracket \implies \{x+1 \dots y - \text{Suc}\ 0\} = \{x \dots y\} - \{x\}$ 
apply (rule equalityI)
apply clarsimp
apply (simp add: intvl_def)
apply (rule conjI)
apply clarsimp
apply (rule_tac x=k+1 in exI)
apply clarsimp
apply arith
apply clarsimp
apply (cases y)
apply simp
apply (simp add: flen_def)
apply (subgoal_tac of_nat k = (0 - 1::'a::fl word))
prefer 2
apply (subst eq_diff_eq)
apply (simp add: add_ac)
apply clarsimp
apply (subst (asm) of_nat_eq)
apply (subst (asm) unat_uminus1)
apply (clarsimp simp add: flen_def)
apply arith
apply (clarsimp simp: intvl_def)
apply (rule_tac x=k - 1 in exI)
apply simp
apply (case_tac k)
  apply simp
apply simp
```

done

```
lemma max_size_flen [simp]:  
  size_of TYPE('a::mem_type) < 2^flen (x::word32)  
  by (insert max_size) (simp add: flen_def addr_card)
```

```
lemma of_nat_bounded:  
   $\exists q. \text{of\_nat } p = (\text{of\_nat } q :: 'a::\text{fl word}) \wedge q \leq p \wedge q < 2^{\text{fl\_of TYPE('a)}}$   
  apply (rule_tac x=unat (of_nat p) in exI)  
  apply (subst word_unat_Rep_inverse)  
  apply (simp add: unat_of_nat mod_le_dividend)  
  done
```

```
lemma word_of_nat_minus:  
   $b < a \implies (\text{of\_nat } a :: 'a::\text{fl word}) - \text{of\_nat } b = \text{of\_nat } (a - b)$   
  by (rule word_unat_Rep_eqD) simp
```

```
lemma intvl_disjoint:  
  fixes q :: 'a :: fl word  
  assumes q:  $q \notin \{p ..+ x\}$   
  assumes p:  $p \notin \{q ..+ y\}$   
  assumes y:  $0 < y$   
  shows  $\{q ..+ y\} \subseteq -\{p ..+ x\}$   
proof clarify  
  from q  
  have q:  $\forall k. q = p + \text{of\_nat } k \implies x \leq k$  by (simp add: intvl_def linorder_not_less)  
  from p  
  have p:  $\forall k. p = q + \text{of\_nat } k \implies y \leq k$  by (simp add: intvl_def linorder_not_less)
```

```
  fix z  
  assume z  $\in \{q ..+ y\}$   
  then obtain zq' where  
    z = q + of_nat zq' zq' < y  
    by (auto simp: intvl_def)  
  moreover  
  from of_nat_bounded [of zq']  
  obtain zq where  
    of_nat zq' = (of_nat zq :: 'a::fl word)  
    zq  $\leq$  zq' zq < 2^fl_of TYPE('a)  
    by blast  
  ultimately  
  have  
    zq: z = q + of_nat zq and  
    zqy: zq < y and  
    zqf: zq < 2^fl_of TYPE('a)  
    by auto
```

```
  assume z  $\in \{p ..+ x\}$   
  then obtain zp' where  
    zp: z = p + of_nat zp' and  
    zpx: zp' < x  
    by (auto simp: intvl_def)  
  moreover  
  from of_nat_bounded [of zp']  
  obtain zp where  
    of_nat zp' = (of_nat zp :: 'a::fl word)
```

```

    zp ≤ zp' zp < 2^fl_of TYPE('a)
    by blast
ultimately
have
  zp: z = p + of_nat zp and
  zpx: zp < x and
  zpf: zp < 2^fl_of TYPE('a)
  by auto

from zp zq
have pq: p + of_nat zp = q + of_nat zq by simp

{
  assume zp = zq
  with pq p
  have ∀k. of_nat k = (0::'a::fl word) → y ≤ k by simp
  hence of_nat 0 = (0::'a::fl word) → y ≤ 0 ..
  with y
  have False by simp
}
moreover {
  from pq have q - p = of_nat zp - of_nat zq
    by (simp add: ring_eq_simps)
  hence q = p + (of_nat zp - of_nat zq)
    by (simp add: ring_eq_simps)
  also
  assume zq < zp
  hence (of_nat zp :: 'a::fl word) - of_nat zq = of_nat (zp - zq)
    by (rule word_of_nat_minus)
  finally
  have q = p + ... .
  with q
  have x ≤ zp - zq by blast
  with zpx
  have False by arith
}
moreover {
  from pq have p - q = of_nat zq - of_nat zp
    by (simp add: ring_eq_simps)
  hence p = q + (of_nat zq - of_nat zp)
    by (simp add: ring_eq_simps)
  also
  assume zp < zq
  hence (of_nat zq :: 'a::fl word) - of_nat zp = of_nat (zq - zp)
    by (rule word_of_nat_minus)
  finally
  have p = q + ... .
  with p
  have y ≤ zq - zp by blast
  with zqy
  have False by arith
}
ultimately
show False by (blast intro: linorder_cases)
qed

```

```

lemma h_t_valid_coerce_disjoint:
  fixes q :: 'a :: mem_type ptr
  assumes q: d,g ⊨t q
  assumes p: d,g' ⊨t (ptr_coerce p :: 'c::c_type ptr)
  assumes x: ptr_val q ∉ {ptr_val p ..+ x} 0 < x
  shows {ptr_val q ..+ size_of TYPE('a)} ⊆ - {ptr_val p ..+ x}
proof -
  from p q x
  have ptr_val p ∉ {ptr_val q ..+ size_of TYPE('a)}
    apply (clarsimp simp: h_t_valid_def valid_footprint_def intvl_shift sz_nzero)
    apply (erule_tac x=ptr_val p in allE, erule impE, rule conjI, assumption)
    apply clarsimp
    apply (erule notE)
    apply (rule intvl_self)
    apply simp
    apply clarsimp
  done
  moreover
  have 0 < size_of TYPE('a) by (rule sz_nzero)
  ultimately
  show ?thesis using x by - (rule intvl_disjoint)
qed

```

```

lemma chunks_start_plus:
  a ∈ chunks (Ptr p) q ⇒ a ∈ chunks (Ptr (p + KMC)) q ∨ a = p
  apply (simp add: chunks_def)
  apply clarsimp
  apply (case_tac n=0)
  apply simp
  apply (rule conjI)
  apply (simp add: word_le_def)
  apply (simp add: uint_plus_if)
  apply (rule conjI)
  apply (clarsimp simp add: KMC_def)
  apply (clarsimp simp add: KMC_def)
  apply (insert uint_range [of p])
  apply arith
  apply (erule_tac x=n - 1 in exI)
  apply (simp add: ring_eq_simps)
  apply (simp add: order_le_less)
  apply (simp add: order_le_less [symmetric])
  apply (simp add: uint_plus_if)
  apply (insert uint_range [of a])
  apply (simp (no_asm_use) add: KMC_def flen_def)
  apply arith
  done

```

```

lemma KMC_dvd_flen [simp]:
  uint KMC dvd 2flen (w::word32)
  by (simp add: KMC_def flen_def)

```

```

lemma zdvd_zdiffD2:
  [[k dvd m - n; k dvd m] ⇒ k dvd (n::int)
  apply (clarsimp simp: dvd_def)
  apply (erule_tac x = kaa-ka in exI)
  apply (simp add: ring_eq_simps)
  done

```

```

lemma zdvd_zaddD:
  [[k dvd m + n; k dvd n] ==> k dvd (m::int)
  apply (clarsimp simp: dvd_def)
  apply (rule_tac x = ka-kaa in exI)
  apply (simp add: ring_eq_simps)
  done

lemma KMC_not_dvd3:
  [[ k < 3; uint KMC dvd n ] ==> uint KMC dvd n - 1 - uint (of_nat k::word32) = False
  apply (cases k)
  apply (clarsimp simp add: KMC_def dest!: zdvd_zdiffD2)
  apply (simp add: zdvd_iff_zmod_eq_0)
  apply (case_tac nat)
  apply (clarsimp simp add: KMC_def dest!: zdvd_zaddD)
  apply (clarsimp simp add: KMC_def dest!: zdvd_zaddD)
  done

lemma word_plus_mono_right_less:
  assumes less: y < z
  assumes sane: x ≤ x+z
  shows x + y < x + (z :: 'a::fl word)
proof -
  from sane
  have unat x + unat z < 2^flen x by (simp add: no_plus_overflow_unat)
  moreover
  with less
  have unat x + unat y < 2^flen x by (simp add: flen_def word_less_nat_alt)
  ultimately
  show ?thesis using less by (simp add: word_less_nat_alt unat_plus_if)
qed

lemma KMC1 [simp]:
  1 < KMC by (simp add: KMC_def)

lemma unat_mult_no_overflow:
  fixes a :: 'a::fl word
  shows unat a * unat b < 2^fl_of TYPE('a) ==> unat (a * b) = unat a * unat b
  apply (subst word_arith_nat_defs)
  apply (subst unat_of_nat)
  apply simp
  done

lemma unat_mult_div:
  fixes k::'a::fl word
  assumes k: k ≠ 0
  assumes n: n < 2^fl_of TYPE('a) div unat k
  shows unat (of_nat n * k) = n * unat k
proof -
  note n
  also have 2^fl_of TYPE('a) div unat k ≤ 2^fl_of TYPE('a) by simp
  finally have n < 2^fl_of TYPE('a) .
  hence unat (of_nat n::'a word) = n by (simp add: unat_of_nat)
  moreover {
    from k
    have 0 < unat k by (simp add: word_less_nat_alt)
    with n

```

```

    have n * unat k < 2 ^ fl_of TYPE('a) div unat k * unat k
      by (rule mult_less_mono1)
    also note div_mult_le
    finally
    have n * unat k < 2 ^ fl_of TYPE('a) .
  }
ultimately
show ?thesis using n by (simp add: unat_mult_no_overflow)
qed

lemma ptr_tags_chunks:
 $\bigwedge p. \llbracket p \neq 0; 4 \text{ udvd } p; p \leq p + \text{of\_nat } n * \text{KMC}; n < (2^{\text{flen } p}) \text{ div unat KMC} \rrbracket \implies$ 
 $\forall q \in \text{chunks } (\text{Ptr } p) (p + \text{of\_nat } n * \text{KMC}). \text{ptr\_tags } (\text{Suc } n) p \text{ d,c\_guard} \models_t (\text{Ptr } q :: \text{word32 ptr})$ 
proof (induct n)
case 0
thus ?case
  apply (clarsimp simp add: KMC_def chunks_def)
  apply (rule ptr_tag_h_t_valid)
  apply (simp add: c_guard_def)
  apply (simp add: ptr_aligned_def)
  apply (drule udvd_dvd)
  apply simp
done
next
case (Suc n')
have 0: p  $\neq$  0
  and 4: 4 udvd p
  and p: p  $\leq$  p + of_nat (Suc n') * KMC
  and n: Suc n' < 2 ^ flen p div unat KMC .
from p n
have unat p + Suc n' * unat KMC < 2^flen p
  apply -
  apply (subst (asm) no_plus_overflow_unat)
  apply (subst (asm) unat_mult_div)
  apply simp
  apply (simp add: flen_def)
  apply assumption
done
hence unat p + unat KMC < 2^flen p by simp
hence pK: p  $\leq$  p + KMC
  by (simp add: no_plus_overflow_unat)
from pK
have p + KMC  $\neq$  0
  by (simp add: word_less_nat_alt unat_plus_simple KMC_def)
moreover
from 4 pK
have 4 udvd (p+KMC)
  by (fastsimp simp: unat_plus_simple dvd_add KMC_def
    intro: dvd_udvd dest: udvd_dvd)
moreover
have distrib: KMC + of_nat n' * KMC = of_nat (Suc n') * KMC
  by (simp add: ring_eq_simps)
from p pK n
have p + KMC  $\leq$  p + (KMC + of_nat n' * KMC)
  apply -

```

```

    apply (rule word_plus_mono_right)
      prefer 2
      apply (simp add: ring_eq_simps)
    apply (subst distrib)
    apply (subst word_le_nat_alt)
    apply (subst unat_mult_div)
      apply simp
      apply (simp add: flen_def)
    apply simp
  done
hence  $p + \text{KMC} \leq p + \text{KMC} + \text{of\_nat } n' * \text{KMC}$  by (simp add: add_ac)
moreover
from n
have  $n' < 2^{\text{flen } (p+\text{KMC})} \text{ div unat KMC}$  by (simp add: flen_def)
ultimately
have  $\forall q \in \text{chunks } (\text{Ptr } (p+\text{KMC}) :: 'a \text{ ptr}) (p + \text{KMC} + \text{of\_nat } n' * \text{KMC}). \text{ptr\_tags } (\text{Suc } n') (p+\text{KMC})$ 
d  $\models_t (\text{Ptr } q :: \text{word32 ptr})$ 
  by (rule Suc)
with  $pK \ 0 \ 4 \ p \ n$ 
show ?case
apply clarsimp
apply (drule chunks_start_plus)
apply (simp add: ring_distrib)
apply (erule disjE)
  apply (drule_tac  $x=q$  in bspec)
  apply (simp add: ring_distrib add_ac)
  apply (rule ptr_tag_h_t_valid_pq)
  apply assumption
  apply (clarsimp simp add: chunks_def)
  apply simp
  apply (subgoal_tac  $p + \text{KMC} \leq q$ )
    prefer 2
    apply (simp add: chunks_def add_ac)
  apply (clarsimp simp add: intvl_def)
  apply (subgoal_tac  $p + (1 + \text{of\_nat } k) < p + \text{KMC}$ )
    apply (simp add: add_ac)
  apply (rule word_plus_mono_right_less)
  apply (case_tac k)
    apply simp
  apply (case_tac nat)
    apply (simp add: KMC_def)
    apply (simp add: KMC_def)
  apply (simp add: add_ac)
  apply simp
  apply (clarsimp simp add: chunks_def uint_plus_simple)
  apply (subgoal_tac  $\neg(\text{uint KMC dvd } (\text{uint } q - \text{uint } p))$ )
    prefer 2
    apply (drule intvld)
  apply (thin_tac  $\text{uint } q = ?t$ )
  apply clarsimp
  apply (simp add: uint_plus_if_split: split_if_asm)
    apply (simp add: KMC_not_dvd3 [of _ 0, simplified])
    apply (simp add: KMC_not_dvd3)
    apply (simp add: ring_eq_simps(13) [symmetric] KMC_not_dvd3)
  apply (subgoal_tac  $\text{uint KMC dvd } (2^{\text{flen } q} - (1 + \text{uint } (\text{of\_nat } k))) + 2^{\text{flen } (q + 1)}$ )
    apply (drule zdvd_zaddD)
    apply simp
  apply (simp add: ring_eq_simps(13) [symmetric])

```

```

    apply (subst (asm) KMC_not_dvd3, (auto)[3])
    apply (simp add: add_ac ring_eq_simps)
    apply (subgoal_tac uint q - uint p = uint KMC + n * uint KMC)
    prefer 2
    apply (simp add: uint_plus_simple)
    apply (subgoal_tac uint KMC dvd (uint KMC + n * uint KMC))
    prefer 2
    apply (rule zdvd_zadd)
    apply simp
    apply simp
    apply simp
    apply simp
    apply (rule ptr_tag_h_t_valid)
    apply (simp add: c_guard_def ptr_aligned_def)
    apply (drule udvd_dvd)
    apply simp
    done
qed

```

```

lemma ptr_tags_chunksI:
   $\bigwedge p. \llbracket p \neq 0; 4 \text{ udvd } p; p \leq p + \text{of\_nat } n * \text{KMC}; n < (2^{\text{flen}} p) \text{ div unat KMC}; q = p + \text{of\_nat } n * \text{KMC}; m = \text{Suc } n \rrbracket \implies$ 
   $\forall q \in \text{chunks } (\text{Ptr } p) \text{ } q. \text{ptr\_tags } m \text{ } p \text{ } d, \text{c\_guard} \models_t (\text{Ptr } q :: \text{word32 ptr})$ 
  apply (drule ptr_tags_chunks)
  apply assumption+
  apply simp
  done

```

```

lemma ptr_tags_applyI:
   $q \notin \{p..+ n * \text{unat KMC}\} \implies \text{ptr\_tags } n \text{ } p \text{ } d \text{ } q = d \text{ } q$ 
  apply (insert ptr_tags_restrict [of n p d])
  apply (drule fun_cong [of _ _ q])
  apply simp
  done

```

```

lemma unat_flen:
   $\text{unat } (x::'a::\text{fl word}) < 2^{\text{flen}} (y::'a::\text{fl word})$ 
  apply (insert unat_lt2p [of x])
  apply (simp add: flen_def)
  done

```

```

lemma max_div_gr0:
   $0 < b \implies 0 < \max a \text{ } b \text{ div } (b::\text{nat})$ 
  apply (clarsimp simp add: max_def linorder_not_le)
  apply (rule less_le_trans)
  prefer 2
  apply (rule div_le_mono)
  apply (drule order_less_imp_le, assumption)
  apply simp
  done

```

```

lemma max_unat_distrib [simp]:
   $\text{unat } (\max a \text{ } b) = \max (\text{unat } a) (\text{unat } b)$ 
  by (simp add: max_def word_le_nat_alt)

```

```

lemma ring_mult_distrib1:
  fixes a :: 'a :: ring_1
  shows  $a * b + b = (a + 1) * b$ 

```

```

by (simp add: ring_eq_simps)

lemma dvd_div_mult_eq:
  fixes a :: nat
  shows  $\llbracket b \text{ dvd } a; 0 < b \rrbracket \implies (a \text{ div } b = c) = (a = c * b)$ 
  apply (rule iffI)
  apply (drule dvd_mult_div_cancel)
  apply (simp add: mult_ac)
  apply simp
  done

lemma ptr_tags_valid_eq:
  fixes q :: 'a::mem_type ptr
  shows
 $\{ \text{ptr\_val } q.. \text{size\_of } \text{TYPE('a)} \} \subseteq - \{ p..+n * \text{unat } \text{KMC} \} \implies$ 
 $\text{ptr\_tags } n \text{ p } d, g \models_t q = d, g \models_t q$ 
  apply (case_tac n=0)
  apply simp
  apply (simp add: h_t_valid_def valid_footprint_def)
  apply (subst ptr_tags_applyI)
  apply clarsimp
  apply (drule Compl_anti_mono)
  apply simp
  apply (drule (1) subsetD)
  apply simp
  apply (rule iffI)
  apply clarsimp
  apply (erule allE, erule impE, assumption)
  apply (subst (asm) ptr_tags_applyI)
  apply (clarsimp simp: intvl_shift sz_nzero max_size_flen)
  apply (drule (1) subsetD)
  apply simp
  apply simp
  apply clarsimp
  apply (subst ptr_tags_applyI)
  prefer 2
  apply simp
  apply (clarsimp simp: intvl_shift sz_nzero max_size_flen)
  apply (drule (1) subsetD)
  apply simp
  done

lemma overfl1:
  fixes a :: 'a::fl word
  fixes c :: 'a::fl word
  fixes x :: 'a::fl word
  assumes x:  $\text{unat } x = \text{unat } a + n * \text{unat } c$ 
  assumes c:  $0 < c$ 
  shows  $\text{unat } a + n * \text{unat } c = \text{unat } (a + \text{of\_nat } n * c)$ 
proof -
  have  $\text{unat } x < 2^{\text{flen } x}$  by (rule unat_flen)
  also note x
  finally
  have anc:  $\text{unat } a + n * \text{unat } c < 2^{\text{flen } x}$  .

  from c
  have  $0 < \text{unat } c$  by (simp add: word_less_nat_alt)

```

```

hence n ≤ n * unat c by simp
also
from anc
have nc: n * unat c < 2^flen x by arith
finally
have n < 2^flen x .
hence unat (of_nat n :: 'a::fl word) = n
  by (simp add: unat_of_nat flen_def)

with nc
have unat (of_nat n * c) = n * unat c
  by (simp add: unat_mult_no_overflow flen_def)
with anc
show ?thesis
  by (simp add: unat_plus_if flen_def)
qed

lemma overfl2:
  fixes x :: 'a::fl word
  fixes a :: 'a::fl word
  assumes n: unat a div k = Suc n
  shows n < 2 ^ flen x div k
proof -
  have unat a < 2 ^ flen a by (rule unat_flen)
  hence unat a ≤ 2 ^ flen a by simp
  hence unat a div k ≤ 2 ^ flen a div k by (rule div_le_mono)
  also note n
  finally
  show ?thesis by (simp add: flen_def)
qed

lemma overfl3:
  assumes w: w < w + s
  assumes s: unat s div unat k = Suc n
  assumes d: unat k dvd unat s
  assumes k: 0 < unat k
  shows w ≤ w + of_nat n * k
proof -
  from w
  have w ≤ w + s by simp
  moreover {
    note dvd_mult_div_cancel [OF d]
    with s
    have unat s = Suc n * unat k
      by (simp add: add_ac mult_ac)
    hence n * unat k ≤ unat s by simp
    moreover
    then have n: n * unat k < 2 ^ flen k
      by (rule order_le_less_trans) (rule unat_flen)
    moreover
    from k have n ≤ n * unat k by simp
    with n have n < 2 ^ flen k
      by - (rule order_le_less_trans)
    ultimately
    have of_nat n * k ≤ s
      by (simp add: word_le_nat_alt unat_word_ariths unat_of_nat flen_def)
  }

```

```

ultimately
show ?thesis by (rule word_plus_mono_right2)
qed

lemma KMC_gr0 [simp]: 0 < unat KMC by (simp add: KMC_def)

install_C_file kmemory.ii

locale free_imp = free_impl +
  constrains
    sep_alloc_invs_body :: globals globals_scheme myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars,
char list, c_errortype) com
    and alloc_invs_body :: globals globals_scheme myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char
list, c_errortype) com
    and sep_free_invs_body :: globals myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char list, c_errortype)
com
    and free_invs_body :: globals myvars  $\Rightarrow$  word32 set  $\Rightarrow$  (globals myvars, char list, c_errortype)
com

lemmas lift_heap_same_type'' [simp] = lift_heap_update_same_type [where g'=c_guard]

lemma (in free_imp)
  free_spec  $\Gamma$  kmem_free_list_addr
  apply (unfold free_spec_def free_spec_axioms_def)
  apply (rule conjI)
  apply (rule kmemory_axioms)
  apply (hoare_rule ProcNoRec1)
  apply (hoare_rule anno = free_invs_body s F in annotateI)
  prefer 2
  apply (subst free_invs_body_def)
  apply (simp add: whileAnno_def)
  apply (subst free_invs_body_def)
  apply (fold KMC_def max_def)
  apply vcg

  — Pre  $\Rightarrow$  I0

  apply (clarsimp simp add: KMC_max chunks_empty)
  apply (subgoal_tac chunks address (ptr_val address + max size KMC)  $\subseteq$ 
    {ptr_val address..ptr_val address + max size KMC})
  prefer 2
  apply (clarsimp simp add: chunks_def)
  apply (subgoal_tac chunks address (ptr_val address + max size KMC - KMC)  $\subseteq$ 
    {ptr_val address..ptr_val address + max size KMC})
  prefer 2
  apply (subgoal_tac chunks address (ptr_val address + (max size KMC - KMC))  $\subseteq$ 
    chunks address (ptr_val address + max size KMC))
  apply (simp add: word_add_ac word_sub_def)
  apply (clarsimp simp add: chunks_def)
  apply (rule conjI)
  prefer 2
  apply (rule_tac x=n in exI)
  apply simp
  apply (erule order_trans)
  apply (rule word_plus_mono_right)
  apply (clarsimp simp add: max_def word_le_def uint_sub_if)
  apply (erule order_less_imp_le)

```

```

apply (rule conjI)
apply blast
apply (subgoal_tac {ptr_val address..+unat (max size KMC) div unat KMC * unat KMC}  $\subseteq$ 
{ptr_val address..ptr_val address+max size KMC})

prefer 2
apply (rule subset_trans)
apply (rule intvl_nowrap)
apply simp
apply (subst max_unat_distrib [symmetric])
apply (subst unat_div [symmetric])
apply simp
apply (rule word_plus_mono_right2)
apply (erule order_less_imp_le)
apply (rule word_div_mult_le)
apply (rule le_less_trans)
apply (rule div_mult_le)
apply (simp add: unat_flen)
apply clarsimp
apply (drule order_less_imp_le, erule order_trans)
apply (rule word_plus_mono_right)
apply (subst max_unat_distrib [symmetric])
apply (subst unat_div [symmetric])
apply simp
apply (rule word_div_mult_le)
apply simp
apply (subgoal_tac ptr_tags (unat (max size KMC) div unat KMC) (ptr_val address) t_d  $\models_t$ 
kmem_free_list_addr)
prefer 2
apply (subgoal_tac t_d  $\models_t$  kmem_free_list_addr)
prefer 2
apply (simp add: typ_simps)
apply (rule ptr_tags_valid, assumption)
apply (rule h_t_valid_coerce_disjoint)
apply assumption
apply assumption
apply (rule notI)
apply (drule (1) subsetD)
apply simp
apply (simp add: max_div_gr0)
apply (rule conjI)
apply (rule_tac x=start in exI)
apply (rule conjI)
apply (simp add: lift $\tau$ _if split: split_if_asm)
apply (rule free_set_heapE, assumption)
apply (simp add: restrict_map_def)
apply (rule ext)
apply clarsimp
apply (simp (no_asm) add: lift $\tau$ _if)
apply (subst ptr_tags_valid_eq)
apply (rule h_t_valid_coerce_disjoint)
apply (rule free_set_valid)
apply simp
apply assumption
apply simp
apply simp
apply (rule notI)
apply (drule (1) subsetD)
apply blast

```

```

apply (simp add: max_div_gr0)
apply (subst ptr_tags_valid_eq)
apply (rule h_t_valid_coerce_disjoint)
  apply (rule free_set_valid)
  apply simp
  apply assumption
  apply simp
  apply simp
  apply (rule notI)
  apply (drule (1) subsetD)
  apply blast
  apply (simp add: max_div_gr0)
apply simp
apply (rule context_conjI)
apply (case_tac address)
apply simp
apply (case_tac max (unat size) (unat KMC) div unat KMC)
  apply simp
  apply (subgoal_tac 0 < max (unat size) (unat KMC) div unat KMC)
  apply simp
  apply (rule max_div_gr0)
  apply simp
apply (rule ptr_tags_chunksI)
  apply (simp add: KMC_def)
  apply (rule order_less_le_trans)
  prefer 2
  apply assumption
  apply simp
  apply (rule dvd_udvd)
  apply simp
  prefer 4
  apply assumption
  apply simp
  apply (erule overfl3)
  apply simp
  apply (clarsimp simp add: max_def)
  apply (erule udvd_dvd)
  apply (simp add: KMC_def)
  apply simp
  apply (rule overfl2)
  apply (subst (asm) max_unat_distrib [symmetric])
  apply assumption
  apply simp
  apply (subst diff_eq_eq)
  apply (subst (asm) dvd_div_mult_eq)
  apply (clarsimp simp add: max_def)
  apply (erule udvd_dvd)
  apply (simp add: KMC_def)
  apply (simp add: add_ac)
  apply (subgoal_tac unat KMC + nat * unat KMC = unat (KMC + of_nat nat * KMC))
  apply simp
  apply (rule word_unat_Rep_eqD)
  apply simp
  apply (subst overfl1)
  apply (subst (asm) max_unat_distrib [symmetric], assumption)
  apply (simp add: KMC_def)
  apply (rule refl)
  apply (rule conjI)

```

```

apply (simp add: chunks_def)
apply (rule conjI)
  apply (rule word_plus_mono_right2)
    apply (erule order_less_imp_le)
  apply (clarsimp simp add: max_def)
  apply (drule not_leE)
  apply (rule word_sub_le)
  apply simp
  apply force
apply simp
apply (drule_tac x=ptr_val address in bspec)
  apply (rule chunks_start)
  apply (rule word_plus_mono_right2)
    apply (erule order_less_imp_le)
  apply (clarsimp simp add: max_def)
  apply (drule not_leE)
  apply (rule word_sub_le)
  apply simp
apply simp

```

—  $I_0 \implies I_0'$

```

apply (clarsimp simp add: typ_simps)
apply (simp add: ptr_less_def ptr_less_def' ptr_le_def ptr_le_def')
apply (simp add: h_t_valid_c_guard c_guard_ptr_aligned c_guard_NULL)
apply (rule context_conjI)
  apply (drule udvd_expand)
  apply simp
  apply clarsimp
  apply (erule word_less_add_right)
  apply (rule word_le_plus_either)
  apply (rule disjI2)
  apply (simp add: max_def)
  apply (clarsimp simp add: linorder_not_le)
  apply (erule order_less_imp_le)
  apply simp
  apply (subgoal_tac ptr_val p < ptr_val p + KMC)
  prefer 2
  apply (erule word_less_nowrapI)
  apply (rule word_le_plus_either)
    apply (rule disjI2)
    apply (clarsimp simp add: max_def linorder_not_le)
    apply (erule order_less_imp_le)
  apply simp
  apply simp
  apply (rule conjI)
  apply (erule udvd_plus')
  apply simp
  apply simp
  apply (rule context_conjI)
  apply simp
  apply (subgoal_tac ptr_val p + KMC ≤ ptr_val p + KMC + KMC)
  prefer 2
  apply (erule udvd_incr2)
    apply simp
    apply simp
  apply (erule udvd_plus')
  apply simp

```

```

    apply simp
    apply simp
    apply simp
  apply (subgoal_tac chunks address (ptr_val p)  $\cap$  {ptr_val p + KMC.. $+1024$ } = {})
  prefer 2
  apply (rule chunks_intvl_disjoint)
    apply simp
    apply (simp add: KMC_def add_ac)
    apply (simp add: flen_def)
  apply (subgoal_tac {ptr_val p + KMC.. $+1024$ }  $\subseteq$  {ptr_val address .. ptr_val address + max size
KMC})
  prefer 2
  apply (subgoal_tac {ptr_val p + KMC.. $+1024$ }  $\subseteq$  {ptr_val address ..< ptr_val address + max
size KMC})
    apply (erule subset_trans)
    apply clarsimp
    apply (erule order_less_imp_le)
    apply (rule subset_trans)
    apply (rule intvl_nowrap)
      apply (simp add: KMC_def add_ac)
      apply (simp add: flen_def)
    apply simp
    apply (rule disjI2)
    apply (rule udvdK)
      apply assumption
      apply simp
      apply (simp add: KMC_def add_ac)
      apply (simp add: max_def KMC_def)
      apply (fold KMC_def)
      apply (erule udvd_plus')
      apply simp
      apply simp
      apply simp
      apply simp
    apply (subgoal_tac ptr_val p + KMC  $\in$  {ptr_val address.. $+1024$ } + max size KMC})
    prefer 2
    apply clarsimp
    apply (erule order_less_imp_le)
  apply (rule context_conjI)
  apply blast
  apply (rule conjI)
  apply (subst chunks_add_udvd3)
    apply assumption
    apply assumption
    apply simp
    apply (rule conjI)
    apply clarsimp
    apply (case_tac p, case_tac address)
    apply (simp add: free_set_singletonI)
    apply (rule impI)
    apply (frule (2) udvd_less_le)
    apply (rule_tac t=p in free_set_insertI)
      apply (rule free_set_heapE, assumption)
      apply (rule ext)
      apply (clarsimp simp add: restrict_map_def)
      apply (simp add: free_set_singletonI)
    apply simp
  apply clarsimp

```

```

apply (subgoal_tac chunks address (ptr_val p - KMC)  $\subseteq$  {ptr_val address .. ptr_val p - KMC})
apply (drule (1) subsetD)
apply simp
apply (subgoal_tac uint (ptr_val p + KMC)  $\leq$  uint (ptr_val p - KMC))
  prefer 2
  apply (simp add: word_le_def)
apply (subgoal_tac uint (KMC)  $\leq$  uint (ptr_val p))
  prefer 2
  apply (simp add: word_le_def)
  apply (simp add: uint_sub_if)
  apply (subst (asm) uint_plus_simple)
  apply (simp add: word_le_def word_less_alt)
  apply (simp add: KMC_def)
  apply (simp (no_asm) add: chunks_def)
  apply clarsimp
apply (rule conjI)
  apply (simp add: chunks_add_udvd3)
  apply clarsimp
  apply (subgoal_tac ptr_val p + KMC  $\in$  chunks address (ptr_val address + (max size KMC - KMC)))
    apply blast
  apply (erule chunks_next)
  apply (rule udvdK)
    apply (simp add: add_ac word_sub_def)
    apply (simp add: add_ac word_sub_def)
    apply (erule udvd_minus)
    apply (simp add: max_def linorder_not_le)
  apply fastsimp
  apply simp
  apply simp
  apply simp
  apply simp

```

—  $I_0 \implies I_1$

```

apply (clarsimp simp add: typ_simps guard_simps)
apply (rule conjI)
  apply (case_tac p)
  apply (simp add: ptr_less_def)
  apply (case_tac start = NULL)
  apply simp
  apply simp
  apply (erule (1) free_set_start)

```

—  $I_1 \implies I_1'$

```

apply clarsimp
apply (drule (1) free_set_next)
apply (clarsimp simp add: typ_simps guard_simps)

```

—  $I_1 \implies \text{Post}$

```

apply (simp add: guard_simps)
apply (elim strip)

```

```

apply (subgoal_tac  $t_d \models_t \text{prev} \vee t_d \models_t (\text{ptr\_coerce prev} :: \text{word32 ptr ptr})$ )
  prefer 2
  apply (case_tac ptr_val prev  $\in$  F)
  apply (rule disjI1)

```

```

    apply (erule (1) free_set_valid)
  apply fastsimp

apply (rule conjI)
  apply (case_tac t_d  $\models_t$  prev)
  apply (simp add: guard_simps)
  apply clarsimp
apply (rule conjI)
  apply (case_tac t_d  $\models_t$  prev)
  apply (simp add: guard_simps)
  apply clarsimp
  apply (simp add: h_t_valid_def c_guard_def ptr_aligned_def)
  apply (simp add: ptr_le_def ptr_le_def' ptr_less_def ptr_less_def')

apply (subgoal_tac  $KMC \leq ptr\_val\ p$ )
  prefer 2
  apply (erule (1) order_trans)
  apply (subgoal_tac  $address \neq NULL$ )
  prefer 2
  apply (simp add: guard_simps)
  apply (case_tac t_d  $\models_t$  prev)
  apply (simp add: typ_simps del: fun_upd_apply)
  apply (case_tac  $prev = ptr\_coerce\ kmem\_free\_list\_addr$ )
  apply simp
  apply (subgoal_tac  $t\_d \models_t kmem\_free\_list\_addr$ )
  apply (simp add: h_t_valid_def valid_footprint_def typ_simps)
  apply (simp add: typ_simps)
  apply (frule_tac  $p=ptr\_val\ prev$  and  $F=F$  in free_set_split_elem)
  apply simp
  apply (elim strip)
  apply simp
  apply (elim strip)
  apply (subgoal_tac  $(F0 \cup F1) \cap chunks\ address\ (ptr\_val\ p - KMC) = \{\}$ )
  prefer 2
  apply simp
  apply (subgoal_tac  $free\_set\ (lift_{\tau^c}\ (t\_h, t\_d)\ (prev \mapsto ptr\_val\ address, p \mapsto ptr\_val\ curr))$ )
start NULL
   $(F0 \cup (\{ptr\_val\ prev\} \cup chunks\ address\ (ptr\_val\ p - KMC)) \cup \{ptr\_val\ p\}$ 
 $\cup F1)$ 
  apply (subgoal_tac  $F0 \cup (\{ptr\_val\ prev\} \cup chunks\ address\ (ptr\_val\ p - KMC)) \cup \{ptr\_val\ p\}$ 
 $\cup F1 =$ 
     $free\ address\ (max\ size\ KMC)\ (insert\ (ptr\_val\ prev)\ (F0 \cup F1))$ )
  apply simp
  apply (clarsimp simp add: free_def Un_ac)
  apply (simp add: chunks_add_end)
  apply (rule_tac  $t=curr$  in free_set_unionI)
  prefer 2
  apply (simp add: typ_simps)
  apply (rule_tac  $t=p$  in free_set_unionI)
  prefer 2
  apply (subgoal_tac  $p \neq curr$ )
  apply (simp add: free_set_singleton)
  apply (case_tac  $curr = NULL$ )
  apply (simp add: guard_simps)
  apply simp
  apply blast
  apply (rule_tac  $t=prev$  in free_set_unionI)
  apply simp

```

```

    apply (rule_tac t=ptr_coerce address in free_set_unionI)
      apply simp
      apply (rule free_set_singleton, clarsimp+)[1]
      apply (simp add: free_set_heapupdate_ignore)
      apply simp
      apply simp
      apply simp
      apply simp
      apply simp
      apply (case_tac curr = NULL)
      apply clarsimp
      apply (fastsimp dest!: free_set_end)
      apply (subgoal_tac ptr_val curr ∈ F1)
      prefer 2
      apply (rule free_set_start, assumption)
      apply simp
      apply (rule conjI)
      apply clarsimp
      apply blast
      apply blast
      apply (simp add: Un_ac Int_ac Un_Int_distrib Int_Un_distrib)
      apply simp
      apply (rule conjI)
      apply clarsimp
      apply (case_tac address)
      apply simp
      apply (rule conjI)
      apply clarsimp
      apply (drule free_set_end)+
      apply clarsimp
      apply clarsimp
      apply (drule free_set_end)+
      apply clarsimp

  apply (simp add: typ_simps)
  apply (subgoal_tac ptr_val prev ∉ F)
  prefer 2
  apply (rule notI)
  apply (drule (1) free_set_valid)
  apply (erule (1) notE)
  apply (simp add: typ_simps)
  apply (subgoal_tac free_set ((liftτc (t_h,t_d))(p ↦ ptr_val curr)) (ptr_coerce address) NULL
    (chunks address (ptr_val p - KMC) ∪ {ptr_val p} ∪ F))
  apply (subgoal_tac chunks address (ptr_val p - KMC) ∪ {ptr_val p} ∪ F = free address (max size
    KMC) F)
  apply simp
  apply (clarsimp simp add: free_def Un_ac insert_commute)
  apply (simp add: chunks_add_end)
  apply (rule_tac t=start in free_set_unionI)
  apply (rule_tac t=p in free_set_unionI)
  apply (simp add: typ_simps)
  apply (rule free_set_singleton)
  apply (case_tac start)
  apply clarsimp
  apply simp
  apply simp

```

```

    apply (case_tac start = NULL)
      apply clarsimp
    apply clarsimp
    apply (case_tac curr = NULL)
      apply simp
    apply simp
    apply blast
    apply (simp add: typ_simps)
    apply (simp add: Un_ac Int_ac)
  apply (subgoal_tac p ≠ NULL)
  prefer 2
  apply (simp add: guard_simps)
  apply (case_tac p)
  apply simp
done

```

end

## 8 kfree with separation logic

```
theory sep_kfree imports non_sep_common sep_common begin
```

```
install_C_file kmemory.ii
```

```
locale sep_free_imp = sep_free_impl +
  constrains
```

```

    sep_alloc_invs_body :: globals globals_scheme myvars ⇒ word32 set ⇒ (globals myvars,
char list, c_errortype) com
    and alloc_invs_body :: globals globals_scheme myvars ⇒ word32 set ⇒ (globals myvars, char
list, c_errortype) com
    and sep_free_invs_body :: globals myvars ⇒ word32 set ⇒ (globals myvars, char list, c_errortype)
com
    and free_invs_body :: globals myvars ⇒ word32 set ⇒ (globals myvars, char list, c_errortype)
com

```

```
lemmas free_set_insertI = free_set_unionI [where ?F1.0={p}, simplified, standard]
```

```
lemma max_le [simp]:
```

```

  y ≤ max x (y::word32)
  by (auto simp: max_def)

```

```
lemma ptr_plus_KMC [simp]:
```

```

  Ptr (ptr_val p + KMC) = (p::word32 ptr) +p 256
  by (case_tac p, simp add: KMC_def)

```

```
lemma udvd_plus:
```

```

  [ K udvd p; p ≤ p + K ] ⇒ K udvd p + K
  apply(rule dvd_udvd)
  apply(drule udvd_dvd)
  apply(subst word_arith_nat_defs)
  apply(subst word_unat_eq_norm)
  apply(subst mod_less)
  apply(subst (asm) no_plus_overflow_unat)
  apply(simp add: flen_def)
  apply(clarsimp simp: dvd_def)

```

```

apply(rule_tac x=k + 1 in exI)
apply clarsimp
done

lemma plus_minus_minus:
  [[ a < b + c - d; d udvd a - b; d udvd c; b ≤ a; d ≤ c; b ≤ b + c ]] ⇒
    d ≤ c - (a - b) - (d::word32)
apply(subgoal_tac unat a < unat b + unat c - unat d)
prefer 2
apply(subst (asm) word_less_nat_alt)
apply(subst (asm) unat_sub)
  apply(subst word_le_nat_alt)
  apply(subst (asm) unat_plus_simple, simp)
  apply(simp add: word_le_nat_alt)
  apply(subst (asm) unat_plus_simple)
  apply simp+
apply(thin_tac ?P < ?Q)
apply(subst word_le_nat_alt)
apply(subst unat_sub)
  apply(subst word_le_nat_alt)
  apply(subst unat_sub)
  apply(subst word_le_nat_alt)
  apply(subst unat_sub)
  apply simp
  apply arith
  apply(subst unat_sub)
  apply simp
  apply(simp add: word_le_nat_alt)
apply(subst unat_sub)
  apply(subst word_le_nat_alt)
  apply(subst unat_sub)
  apply simp
  apply arith
  apply(subst unat_sub)
  apply simp
  apply(drule udvd_dvd)+
  apply(subst (asm) unat_sub)
  apply simp
  apply simp
  apply(clarsimp simp: dvd_def)
  apply(subgoal_tac unat d * ka - (unat d * k + unat d) =
    unat d * (ka - (k + 1)))
  prefer 2
  apply(simp add: diff_mult_distrib2)
  apply clarsimp
  apply(subgoal_tac (unat a < unat b + unat d * ka - unat d) =
    (unat d < unat d * ka - (unat a - unat b)))
  apply simp
  apply(subst (asm) diff_mult_distrib2 [symmetric])
  apply clarsimp
  apply arith
  apply(thin_tac unat a - unat b = ?P)
  apply simp
  apply(subst diff_diff_right)
  apply(simp add: word_le_nat_alt)
  apply arith
done

```

```

lemma plus_minus:
  [[ a < a + b; c ≤ a; c udvd a; c udvd b ]] ⇒ a ≤ a + b - (c::word32)
apply(subst word_le_nat_alt)
apply(subst unat_sub)
  apply simp
apply(frule order_less_imp_le)
apply(subst (asm) unat_plus_simple, simp)
apply(frule order_less_imp_le)
apply(subst (asm) word_less_nat_alt)
apply(subst (asm) unat_plus_simple, simp)
apply(drule udvd_dvd)+
apply(clarsimp simp: dvd_def)
apply(subgoal_tac unat c * k + unat c * ka - unat c = unat c * (k + ka - 1))
  apply simp
apply(simp add: nat_distrib)
done

lemma c_guard_ptr_coerce:
  c_guard ((ptr_coerce p)::word32 ptr) ⇒ c_guard (p::word32 ptr ptr)
  by (auto simp: c_guard_def ptr_aligned_def)

lemma sep_list_NULL [rule_format,simp]:
  ∀x s. sep_list x y ps s → 0 ∉ set ps
apply(induct_tac ps)
  apply simp
  apply clarsimp
  apply(drule sep_conjD, clarsimp)
  apply rule
  apply(clarsimp simp: block_def)
  apply(drule sep_map'_g)
  apply(drule c_guard_NULL)
  apply clarsimp+
done

lemma sep_free_set_NULL [simp]:
  sep_free_set x y F s ⇒ 0 ∉ F
  by (clarsimp simp: sep_free_set_def)

lemma sep_free_set_H_NULL [simp]:
  sep_free_set_h x y F s ⇒ 0 ∉ F
  apply(clarsimp simp: sep_free_set_h_def)
  apply(drule sep_conjD, clarsimp)
done

lemma sep_free_set_sep_map':
  [[ (sep_free_set x y F ∧* P) s; F ≠ {} ]] ⇒ ∃z. (x ↔ z) s
apply(subgoal_tac ptr_val x ∈ F)
  prefer 2
  apply(drule sep_conjD, clarsimp)+
  apply(frule sep_free_set_start)
  apply clarsimp
  apply assumption
  apply(subgoal_tac ptr_val x ≠ ptr_val y)
  prefer 2
  apply clarsimp
  apply(subgoal_tac ptr_val y ∉ F)
  prefer 2
  apply(drule sep_conjD, clarsimp)+

```

```

  apply(drule sep_free_set_end)
  apply simp
  apply(subst (asm) sep_free_set_split_elem_eq)
    apply fast
    apply simp+
  apply(subst (asm) sep_conj_com)
  apply(subst (asm) sep_conj_exists)+
  apply(erule exE)+
  apply clarsimp
  apply(subst (asm) sep_conj_com)
  apply(subst (asm) sep_conj_exists)+
  apply(erule exE)+
  apply clarsimp
  apply(subst (asm) block_alt)
  apply clarsimp
  apply(rule_tac x=xb in exI)
  apply(erule sep_map'_conjE2)
  apply(erule sep_map'_conjE2)
  apply(erule sep_map'_conjE1)
  apply(erule sep_map_sep_map')
done

lemma (in sep_free_imp)
  shows sep_free_spec  $\Gamma$  kmem_free_list_addr
  apply (unfold sep_free_spec_def sep_free_spec_axioms_def)
  apply (rule conjI)
    apply (rule kmemory_axioms)
  apply (hoare_rule ProcNoRec1)
  apply (hoare_rule anno = sep_free_invs_body s F in annotateI)
  prefer 2
  apply (subst sep_free_invs_body_def)
  apply (simp add: whileAnno_def)
  apply (subst sep_free_invs_body_def)
  apply (fold KMC_def max_def)
  apply (unfold sep_app_def)
  apply vcg
  apply simp

— Pre  $\implies$  I0

apply rule
  apply(erule chunks_empty)
  apply simp
apply rule
  prefer 2
  apply rule
    apply(clarsimp simp: max_def)
    apply(rule dvd_udvd)
    apply(simp add: KMC_def)
    apply(erule (2) plus_minus)
    apply(clarsimp simp: max_def)
    apply(rule dvd_udvd)
  apply simp
apply(rule ptr_tag_sep_cut')
  prefer 2
  apply(simp add: c_guard_def)
  apply rule
    apply(rule ptr_aligned_KMC, simp)

```

```

apply clarsimp
apply simp
apply(subst sep_conj_com)
apply(subst sep_conj_assoc)+
apply(erule (1) sep_conj_impl)
apply(simp add: rest_of_def)
apply(drule_tac x=KMC in sep_cut_split)
  apply simp
apply(subst (asm) sep_conj_com)
apply(erule (1) sep_conj_impl)
apply(drule_tac x=4 in sep_cut_split)
  apply(simp add: KMC_def)
apply(simp add: sep_cut_def)

```

—  $I_0 \implies I_0'$

```

apply clarsimp
apply rule
  apply(rule c_guard_NULL, rule tagd_g)
  apply(erule (2) sep_conj_impl)
apply rule
  apply(rule c_guard_ptr_aligned, rule tagd_g)
  apply(erule (2) sep_conj_impl)
apply rule
  prefer 2
  apply rule
    apply(case_tac p, clarsimp simp: ptr_less_def)
    apply(drule plus_minus_no_overflow)
    apply simp
    apply(subst add_commute)
    apply(rule le_no_overflow)
    apply simp
    apply(subst no_plus_overflow_unat)
    apply(subst (asm) no_plus_overflow_unat)
    apply(simp add: flen_def KMC_def)
  apply(simp add: KMC_def)
  apply rule
    apply(rule udvd_plus')
    apply simp+
    apply(case_tac p, clarsimp simp: ptr_less_def)
    apply(drule plus_minus_no_overflow)
    apply simp+
  apply rule
    apply(erule udvd_plus)
    apply(case_tac p, clarsimp simp: ptr_less_def)
    apply(drule plus_minus_no_overflow)
    apply simp+
  apply(erule le_plus)
    apply(simp add: ptr_less_def)
    apply(drule (3) plus_minus_minus)
    apply(subst max_def, clarsimp simp: word_le_nat_alt)
    apply simp
    apply(simp add: diff_minus add_ac)
    apply(clarsimp simp: ptr_less_def)
    apply(drule plus_minus_no_overflow)
    apply simp+
  apply(subst sep_conj_assoc [symmetric])+
  apply(subst sep_conj_com)

```

```

apply(subst sep_conj_assoc)+
apply(rule ptr_tag_sep_cut')
prefer 2
apply(simp (no_asm) add: c_guard_def)
apply rule
  apply(rule ptr_aligned_plus)
  apply(rule c_guard_ptr_aligned, rule tagd_g)
  apply(erule (2) sep_conj_impl)
  apply(case_tac p, clarify, simp add: ptr_less_def)
  apply(drule plus_minus_not_NULL)
  apply(rule le_no_overflow)
  apply simp+
  apply(simp add: KMC_def)
apply simp
apply(subst sep_conj_com) back back back
apply(subst sep_conj_assoc [symmetric])+
apply(subst sep_conj_com) back back back
apply(subst sep_conj_assoc)+
apply(rule_tac p=p in sep_free_set_prev_rev)
  apply(simp add: KMC_def)
  apply(subst chunks_add_udvd2)
  apply(case_tac p, clarsimp simp: ptr_less_def)
  apply(simp add: KMC_def)+
  apply(simp add: sep_map'_def)
  apply(subst sep_conj_com)
  apply(rule sep_heap_update_global')
  apply(erule (1) sep_conj_impl, simp)
  apply(subst block_alt)
  apply clarsimp
  apply(subgoal_tac (rest_of p KMC  $\wedge^*$ 
    sep_free_set_h kmem_free_list_addr NULL F  $\wedge^*$ 
    rest_of (p +p 256) KMC  $\wedge^*$ 
    p  $\mapsto$  (ptr_val p + 1024)  $\wedge^*$ 
    sep_cut' (ptr_val p + 1024) 4  $\wedge^*$ 
    sep_free_set (ptr_coerce address) p
    (chunks address (ptr_val p + 1024 - KMC) - {ptr_val p})  $\wedge^*$ 
    sep_cut (ptr_val p + 1024 + KMC)
    (max size KMC - (ptr_val p + 1024 - ptr_val address) - KMC)) =
    (p  $\mapsto$  (ptr_val p + (1024::word32))  $\wedge^*$  rest_of p KMC  $\wedge^*$ 
    sep_free_set_h kmem_free_list_addr (Ptr (0::word32)) F  $\wedge^*$ 
    rest_of (p +p (256::word32)) KMC  $\wedge^*$ 
    sep_cut' (ptr_val p + (1024::word32)) (4)  $\wedge^*$ 
    sep_free_set (ptr_coerce address) p
    (chunks address (ptr_val p + (1024::word32) - KMC) -
    {ptr_val p})  $\wedge^*$ 
    sep_cut (ptr_val p + (1024::word32) + KMC)
    (max size KMC - (ptr_val p + (1024::word32) - ptr_val address) -
    KMC)))
  prefer 2
  apply simp
  apply(simp only: KMC_def)
  apply(rule sep_heap_update_global')
  apply(thin_tac ?P = ?Q)
  apply(erule (1) sep_conj_impl)
  apply(erule (1) sep_conj_impl)
  apply(erule (1) sep_conj_impl)
  apply(simp add: KMC_def)
  apply(erule sep_conj_impl)

```

```

apply(subst chunks_add_udvd2)
  apply(simp add: KMC_def)+
apply(subgoal_tac ptr_val p  $\notin$  (chunks address (ptr_val p - 1024)))
  apply simp
apply(rule chunks_end_nowrap)
  apply simp
  apply(simp add: word_less_nat_alt)
apply(drule_tac x=1024 in sep_cut_split)
  apply(clarsimp simp: ptr_less_def)
  apply(erule (3) plus_minus_minus)
  apply(clarsimp simp: max_def word_le_nat_alt)
  apply simp+
apply(subst sep_conj_assoc [symmetric])+
apply(erule sep_conj_impl)
  apply(drule_tac x=4 in sep_cut_split)
  apply simp
  apply(erule sep_conj_impl)
  apply(simp add: sep_cut_def)
  apply(simp add: rest_of_def)
  apply simp
apply clarsimp
apply simp
apply(rule chunks_end_nowrap)
  apply(simp add: KMC_def)
  apply(subst word_le_nat_alt)
  apply(subgoal_tac ptr_val p  $\leq$  ptr_val p + 1024)
  apply(subst (asm) unat_plus_simple, simp)
  apply(rule plus_minus_no_overflow)
  apply(simp add: ptr_less_def)
  apply(subst word_le_nat_alt)
  apply(subgoal_tac ptr_val address  $\leq$  ptr_val address + max size 1024)
  apply(subst (asm) unat_plus_simple, simp)
  apply(subst max_def)
  apply(clarsimp simp: word_le_nat_alt)
  apply simp
apply simp

```

—  $I_0 \implies I_1$

```

apply rule
  apply(rule c_guard_NULL)
  apply(rule c_guard_ptr_coerce)
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_hD, clarsimp)
  apply(rule sep_map'_g)
  apply(erule sep_map'_conjE2)
  apply(erule sep_map_sep_map')
apply rule
  apply(rule c_guard_ptr_aligned)
  apply(rule c_guard_ptr_coerce)
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_hD, clarsimp)
  apply(rule sep_map'_g)
  apply(erule sep_map'_conjE2)
  apply(erule sep_map_sep_map')
apply rule
  apply(rule_tac x={ } in exI)
  apply(rule_tac x=F in exI)

```

```

apply(subst (asm) sep_free_set_h_def)
apply(subst (asm) sep_conj_exists)
apply clarify
apply(frule sep_map'_conjE1)
  apply(simp add: sep_map'_def)
  apply(erule sep_conj_impl, simp)
  apply fast
apply(drule sep_map'_lift)
apply(drule lift_ptr_coerce')
apply simp
apply(erule (1) sep_conj_impl)+
apply(simp add: ptr_less_def add_ac)
apply clarsimp
apply rule
  apply(simp add: add_ac diff_minus)
apply rule
  apply(simp add: ptr_less_def)
apply rule
  apply(simp add: ptr_less_def)
apply(subst (asm) sep_free_set_h_def)
apply simp
apply(subgoal_tac (c_guard  $\vdash_s$  p  $\wedge^*$ 
  rest_of p KMC  $\wedge^*$ 
  sep_free_set (ptr_coerce address) p (chunks address (ptr_val p - KMC))  $\wedge^*$ 
  ( $\lambda$ h.  $\exists$ x. (sep_free_set x NULL F  $\wedge^*$  ptr_coerce kmem_free_list_addr  $\mapsto$  ptr_val x) h)  $\wedge^*$ 
  sep_cut (ptr_val p + KMC) (max size KMC - (ptr_val p - ptr_val address) - KMC)) =
  (( $\lambda$ h.  $\exists$ x. (sep_free_set x NULL F  $\wedge^*$  ptr_coerce kmem_free_list_addr  $\mapsto$  ptr_val x) h)  $\wedge^*$ 
c_guard  $\vdash_s$  p  $\wedge^*$ 
  rest_of p KMC  $\wedge^*$ 
  sep_free_set (ptr_coerce address) p (chunks address (ptr_val p - KMC))  $\wedge^*$ 
  sep_cut (ptr_val p + KMC) (max size KMC - (ptr_val p - ptr_val address) - KMC)))
  prefer 2
  apply simp
apply(simp only:)
apply(subst (asm) sep_conj_exists)
apply(thin_tac ?P = ?Q)
apply clarify
apply(subst (asm) sep_conj_com)
apply(subst (asm) sep_conj_assoc)+
apply(frule sep_map'_conjE1)
  apply(erule sep_map_sep_map')
apply(frule sep_map'_lift)
apply simp

—  $I_1 \implies I_1'$ 

apply clarsimp
apply rule
  apply(drule sep_conjD, clarsimp)+
  apply(frule sep_free_set_start)
  apply clarsimp
  apply(drule (1) sep_free_set_split_elem)
  apply clarsimp
  apply(drule sep_conjD, clarsimp)+
  apply(clarsimp simp: block_def)
  apply(rule c_guard_ptr_aligned)
  apply(rule sep_map'_g)
  apply fast

```

```

apply rule
apply(subgoal_tac ptr_val curr  $\notin$  G)
  prefer 2
  apply(subst (asm) sep_free_set_h_def)
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_end, force)
apply(rule_tac x=G  $\cup$  {ptr_val curr} in exI)
apply(rule_tac x=H - {ptr_val curr} in exI)
apply rule
  apply(subst (asm) sep_free_set_singleton)
  apply assumption
  apply clarsimp
  apply(subst (asm) sep_conj_assoc [symmetric])+
  apply(subst (asm) sep_conj_com) back back back
  apply(subst (asm) sep_conj_assoc)+
  apply(subst (asm) sep_conj_exists)
  apply clarsimp
  apply(subgoal_tac ((c_guard  $\vdash_s$  p  $\wedge^*$ 
    ((rest_of p KMC)  $\wedge^*$ 
      ((sep_free_set_h kmem_free_list_addr curr G)  $\wedge^*$ 
        ((sep_free_set curr (Ptr x) {(ptr_val curr)})  $\wedge^*$ 
          ((sep_free_set (Ptr x) NULL (H - {(ptr_val curr)}))  $\wedge^*$ 
            (sep_free_set (ptr_coerce address) p
              (chunks address ((ptr_val address) + ((-2 * KMC) + (max size KMC)))))))))) =
        (((sep_free_set curr (Ptr x) {(ptr_val curr)})  $\wedge^*$ 
          ((rest_of p KMC)  $\wedge^*$ 
            ((sep_free_set_h kmem_free_list_addr curr G)  $\wedge^*$ 
              ( c_guard  $\vdash_s$  p  $\wedge^*$ 
                ((sep_free_set (Ptr x) NULL (H - {(ptr_val curr)}))  $\wedge^*$ 
                  (sep_free_set (ptr_coerce address) p
                    (chunks address ((ptr_val address) + ((-2 * KMC) + (max size KMC))))))))))))))
    prefer 2
    apply simp
  apply(simp only:)
  apply(thin_tac ?P = ?Q) back
  apply(frule sep_map'_conjE1)
  apply(clarsimp simp: sep_free_set_block block_def)
  apply fast
  apply(simp add: sep_map'_lift)
  apply(erule (1) sep_conj_impl)+
  apply(subgoal_tac x  $\notin$  G)
  prefer 2
  apply clarsimp
  apply(drule sep_conjD, clarsimp)+
  apply(frule sep_free_set_start) back
  apply(clarify, simp)
  apply clarsimp
  apply(drule sep_free_set_dom)+
  apply(drule sep_free_set_h_dom)
  apply(subst (asm) heap_disj_def)
  apply simp
  apply blast
  apply(subst (asm) sep_conj_assoc [symmetric])+
  apply(subst sep_conj_assoc [symmetric])+
  apply(erule sep_conj_impl)
  prefer 2
  apply assumption
  apply(erule sep_conj_impl)

```

```

    apply(subgoal_tac insert (ptr_val curr) G = G ∪ {ptr_val curr})
    apply(simp only:)
    apply(rule sep_free_set_h_unionI)
    apply(simp add: sep_map'_lift)+
  apply(drule sep_conjD, clarsimp)+
  apply(drule sep_free_set_start)
  apply simp
  apply fast
  apply(subgoal_tac (c_guard ⊢s p ∧*
    rest_of p KMC ∧*
    sep_free_set_h kmem_free_list_addr curr G ∧*
    sep_free_set curr NULL H ∧*
    sep_free_set (ptr_coerce address) p
    (chunks address (ptr_val address + (-2 * KMC + max size KMC)))) =
  (
    sep_free_set curr NULL H ∧* c_guard ⊢s p ∧*
    rest_of p KMC ∧*
    sep_free_set_h kmem_free_list_addr curr G ∧*

    sep_free_set (ptr_coerce address) p
    (chunks address (ptr_val address + (-2 * KMC + max size KMC))))))
  prefer 2
  apply simp
  apply(simp only:)
  apply(thin_tac ?P = ?Q) back
  apply(frule sep_free_set_sep_map')
  apply(clarsimp simp: sep_free_set_def)
  apply(clarsimp simp: sep_map'_lift)

```

—  $I_1 \implies \text{Post}$

```

  apply(clarsimp simp: free_def)
  apply rule
  apply(rule c_guard_NULL)
  apply(erule sep_map'_g)
  apply rule
  apply(rule c_guard_ptr_aligned)
  apply(erule sep_map'_g)
  apply rule
  apply(rule c_guard_NULL)
  apply(erule tagd_g)
  apply rule
  apply(rule c_guard_ptr_aligned)
  apply(erule tagd_g)
  apply(subst Un_assoc)
  apply(rule_tac t=ptr_coerce address in sep_free_set_h_unionI)
  apply(case_tac p, clarsimp)
  apply(subst chunks_add_udvd2)
  apply simp
  apply(simp add: word_le_nat_alt)
  apply simp
  apply(rule udvd_minus)
  apply simp
  apply(subst max_def, clarsimp simp: KMC_def word_le_nat_alt)
  apply(subst sep_free_set_split_elem_eq)
  apply(subgoal_tac ptr_val address + (max size KMC - KMC) ∈ H ∪
    (chunks address (ptr_val address + (max size KMC - KMC) - KMC) ∪
    {ptr_val address + (max size KMC - KMC)}))
  prefer 2

```

```

    apply fast
  apply fast
  apply (clarsimp simp: word_less_nat_alt word_le_nat_alt KMC_def)
  apply clarsimp
  apply rule
  apply (clarsimp simp: word_less_nat_alt word_le_nat_alt KMC_def)
  apply rule
  apply (drule sep_conjD, clarsimp)+
  apply (simp add: add_ac)
  prefer 2
  apply (drule sep_conjD, clarsimp)
  apply (drule sep_conjD, clarsimp)
  apply (drule sep_conjD, clarsimp)
  apply (subst sep_conj_com)
  apply (subst sep_conj_exists)
  apply (subst sep_conj_exists)
  apply (rule_tac x=chunks address (-2 * KMC + (ptr_val address + max size KMC))
    in exI)
  apply clarsimp
  apply (rule_tac x=H in exI)
  apply rule
  apply (force simp: add_ac)
  apply (subst sep_conj_com)
  apply (subst sep_conj_exists)
  apply (rule_tac x=ptr_val curr in exI)
  apply clarsimp
  apply (subst block_alt)
  apply clarsimp
  apply rule
  apply (clarsimp simp: add_ac diff_minus)
  apply (drule sep_conjD, clarsimp)+
  apply (frule sep_free_set_start)
  apply clarsimp
  apply (drule sep_free_set_dom)+
  apply (drule tagd_dom_p)
  apply (drule (1) subsetD)
  apply (subst (asm) heap_disj_def)
  apply simp
  apply fast
  apply rule
  apply (simp add: add_ac diff_minus)
  apply (subgoal_tac (sep_free_set curr NULL H  $\wedge^*$ 
    sep_free_set_h kmem_free_list_addr (ptr_coerce address) G  $\wedge^*$ 
    rest_of (Ptr (ptr_val address + (max size KMC - KMC))) KMC  $\wedge^*$ 
    Ptr (ptr_val address + (max size KMC - KMC))  $\mapsto$  ptr_val curr  $\wedge^*$ 
    sep_free_set (ptr_coerce address) (Ptr (ptr_val address + (max size KMC - KMC)))
    (chunks address (-2 * KMC + (ptr_val address + max size KMC)))) =
    (
      Ptr (ptr_val address + (max size KMC - KMC))  $\mapsto$  ptr_val curr  $\wedge^*$  sep_free_set curr NULL
    H  $\wedge^*$ 
    sep_free_set_h kmem_free_list_addr (ptr_coerce address) G  $\wedge^*$ 
    rest_of (Ptr (ptr_val address + (max size KMC - KMC))) KMC  $\wedge^*$ 
    sep_free_set (ptr_coerce address) (Ptr (ptr_val address + (max size KMC - KMC)))
    (chunks address (-2 * KMC + (ptr_val address + max size KMC))))))
  prefer 2
  apply simp
  apply (simp only: diff_minus add_ac)
  apply (rule sep_heap_update_global')

```

```

apply(thin_tac ?P = ?Q)
apply(erule disjE) back
  apply clarsimp
  apply(subst sep_free_set_h_def)
  apply(subst sep_conj_com)
  apply(subst sep_conj_assoc)+
  apply(subst sep_conj_exists)
  apply(drule (1) sep_free_set_h_empty_sep_map')
  apply simp
  apply(rule_tac x=ptr_coerce address in exI)
  apply simp
  apply(subst sep_conj_com)
  apply(subst sep_conj_assoc)+
  apply(rule sep_heap_update_global')
    apply simp+
  apply(subst sep_conj_com)
  apply(subst sep_conj_assoc)+
  apply(erule (1) sep_conj_impl)
  apply simp
  apply(erule sep_conj_impl)
    apply(erule sep_map_tagd)
  apply(erule (1) sep_conj_impl)+
  apply simp
apply clarsimp
apply(subst sep_conj_com)
apply(subst sep_conj_assoc)+
apply(rule sep_free_set_h_prev_rev, assumption)
  apply simp
  apply(subst sep_map'_def)
  apply(rule sep_heap_update_global')
    apply simp+
  apply(subst (asm) sep_map'_def)
  apply simp
  apply(erule (1) sep_conj_impl)
  apply(erule sep_map_tagd)
prefer 2
apply clarsimp
apply(case_tac address, clarsimp)
apply(drule sep_conjD, clarsimp)+
apply(frule sep_free_set_start) back
  apply clarsimp
  apply(drule tagd_dom_p)
  apply(drule sep_free_set_h_dom)
  apply(drule (1) subsetD)
  apply(subst (asm) heap_disj_def)
  apply simp
  apply fast
  apply(drule sep_free_set_h_dom)
  apply(drule sep_free_set_dom)+
  apply(drule (1) subsetD)
  apply(drule (1) subsetD)
  apply(subst (asm) heap_disj_def)
  apply simp
  apply fast
prefer 2
apply clarsimp
apply(drule sep_conjD, clarsimp)+
apply(frule sep_free_set_start) back

```

```

    apply clarsimp
    apply(case_tac address, clarsimp)
    apply(drule sep_free_set_h_dom)
    apply(drule (1) subsetD) back
    apply(drule tagd_dom_p)
    apply(subst (asm) heap_disj_def)
    apply simp
    apply fast
    apply(drule sep_free_set_h_dom)
    apply(drule sep_free_set_dom)+
    apply(drule (1) subsetD) back
    apply(drule (1) subsetD)
    apply(subst (asm) heap_disj_def)
    apply simp
    apply fast
  apply(frule (2) sep_free_set_h_prev)
    apply clarsimp
    apply(drule sep_conjD, clarsimp)+
    apply(frule sep_free_set_start)
      apply clarsimp
      apply(drule sep_free_set_h_dom)
      apply(drule sep_free_set_dom)+
      apply(drule (1) subsetD)
      apply(drule (1) subsetD)
      apply(subst (asm) heap_disj_def)
      apply simp
      apply fast
    apply clarsimp
    apply(drule sep_conjD, clarsimp)+
    apply(frule sep_free_set_start)
      apply clarsimp
      apply(drule sep_free_set_h_dom)
      apply(drule sep_free_set_dom)+
      apply(drule (1) subsetD) back
      apply(drule (1) subsetD)
      apply(subst (asm) heap_disj_def)
      apply simp
      apply fast
  apply simp
  apply(subst ptr_val_ptr_coerce [symmetric], rule sep_heap_update_block)
  apply(erule (1) sep_conj_impl)+
  apply simp
done

end

```