

# OS161 Memory Management

## Dumbvm and kmalloc

- Memory management of OS161 consists of two elements:
  - **Kmalloc** allows memory allocation of kernel side
  - **Dumbvm** memory manager user side
- Memory manager of OS161 use **Contiguous allocation**
- However, it also uses pagination (minimum amount of available memory is page(s))
  - Allocating by page multiples (4096 byte frame, 4K)

OS161 usa un meccanismo di ***contiguous allocation***.

Ha anche la ***paginazione***, ma nel senso che la memoria è suddivisa in frames da **4KB**.

Un processo può avere bisogno di 10 pagine, ma devono essere allocate in 10 frames **contigui**.

## Dumbvm and kmalloc

- Allocation of memory is done in two levels:
  - `getppages` (`dumbvm.c`): calls `ram_stealmem` (in mutual exclusion) (more generic abstraction)
  - `ram_stealmem` (`ram.c`): allocates contiguous RAM starting at `firstpaddr`, that is increased (containing codes architecture dependent)
- Allocator is common to (for both user and kernel)
  - User memory: `as_prepare_load` calls `getppages` for 2 user segments and a stack
  - Dynamic kernel memory: `kmalloc` is based on `alloc_kpages`, that calls `getppages`

L'allocazione di memoria in OS161 è effettuata su due livelli:

- `getppages` — una funzione di alto livello che è un wrapper per `ram_stealmem`
- `ram_stealmem` — una funzione di più basso livello (contiene codice che dipende dall'architettura su cui OS161 viene eseguito) che effettivamente alloca memoria contigua partendo dal `firstpaddr` (il primo indirizzo **fisico** disponibile, il quale viene aggiornato a seguito di ciascuna allocazione)

Il sistema di allocazione della memoria è comune a user e kernel space.

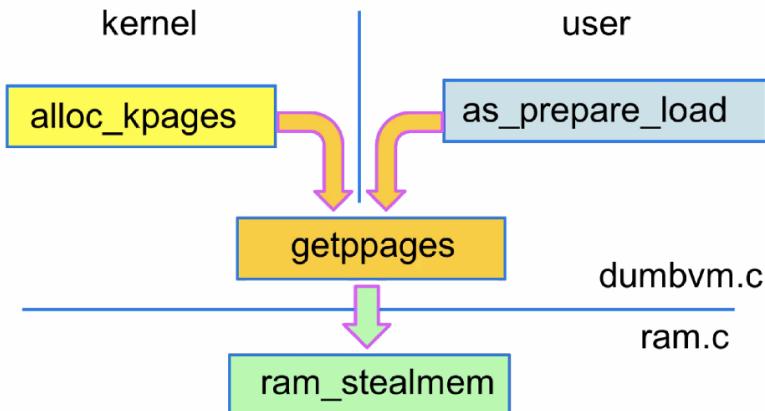
A livello user viene usata la funzione

`as_prepare_load` , che chiama `getppages` per allocare 2 segmenti user e uno stack (ciò di cui ha bisogno un programma utente per eseguire: 2 segmenti per caricarvi il codice e i dati del programma, e lo stack).

A livello kernel viene usata la funzione

`kmalloc` , la quale chiama `alloc_kpages` , che a sua volta chiama `getppages` .

## Dumbvm.c (alloc)



Memory manager architecture of OS161:  
**ram\_stealmem** for memory management at low level (architecture dependent)  
**getppages** for memory management in `dumbvm.c`  
Allocation function for user side (**as\_prepare\_load**) and kernel side (**alloc\_kpages**)

La gestione della memoria viene chiesta dallo user space, ma anche dal kernel space.

L'OS riceve richieste di allocazione di memoria da entrambe le parti.

In OS161 ci sono 2 sistemi che rispondono alle richieste di kernel o user per allocare memoria:

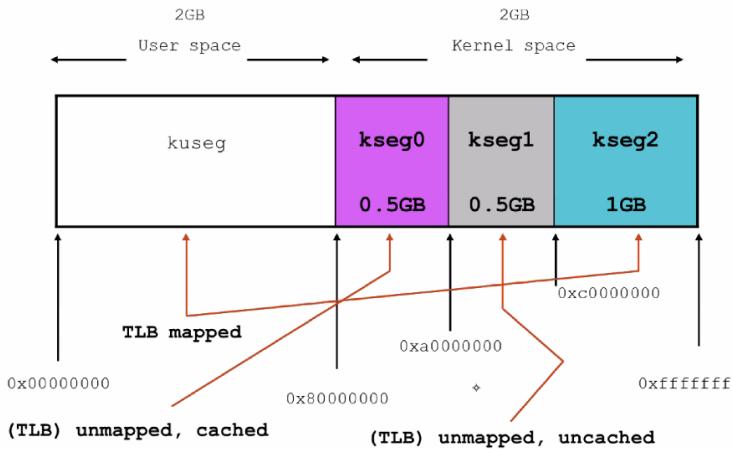
- `dumbvm.c`, in cui troviamo la funzione `getppages`
- `ram.c`, in cui troviamo la funzione `ram_stealmem`

OS161 risponderà a tali richieste: prima tramite `dumbvm.c` e poi `ram.c`.

Se volessimo implementare un altro meccanismo di paging lo faremmo su `dumbvm.c`.

Se volessimo implementare OS161 su un'altra architettura (noi usiamo MIPS), dovremmo riscrivere `ram.c` (che è a basso livello e contiene codice dipendente dall'architettura).

# MIPS VIRTUAL ADDRESS SPACE



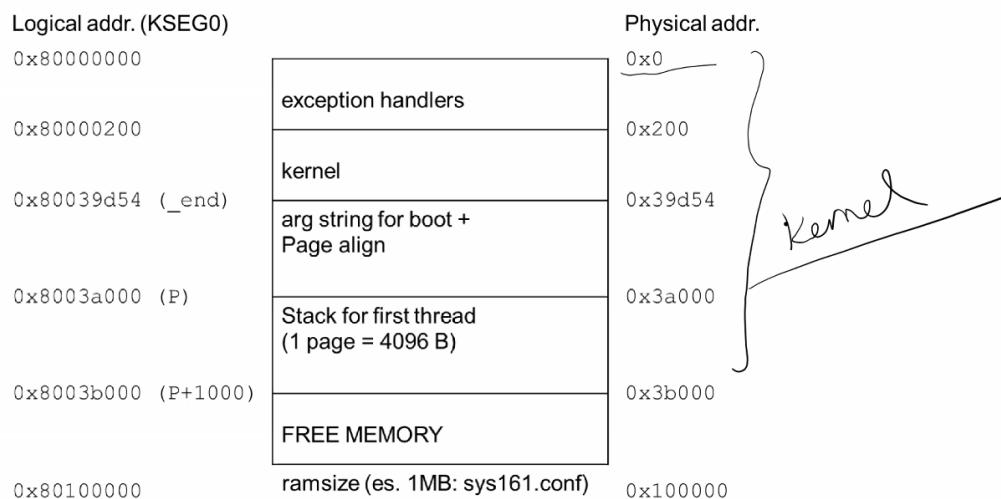
OS161 is running on Mips, a 32 bits processor and addresses at 32 bits. Therefore, logical address space of 4GB.

OS161 è basato su MIPS, su processore a 32 bit e dunque indirizzi di 32 bit, dunque spazio d'indirizzamento da 4GB.

Questi 4GB vengono divisi in 4 parti:

- User Space
- Kernel space
  - kseg0 → contiene strutture dati usate per l'esecuzione di kernel threads
  - kseg1 → usato per mappare dispositivi I/O
  - kseg2 → non viene usato

## Kernel loader (sys161: start.S)



What happens during the boot time of Os161?

Using part of the memory for loading the kernel of OS161, starting from address 0. Starting from exception handlers, kernel, the memory space for the argument of command line of kernel (it is empty right now), stack for the first process that will be created, and free physical memory).

In kseg0 carichiamo il kernel.

Dopo che finisce lo spazio allocato per il kernel, inizia lo spazio libero che può essere usato per allocare processi.

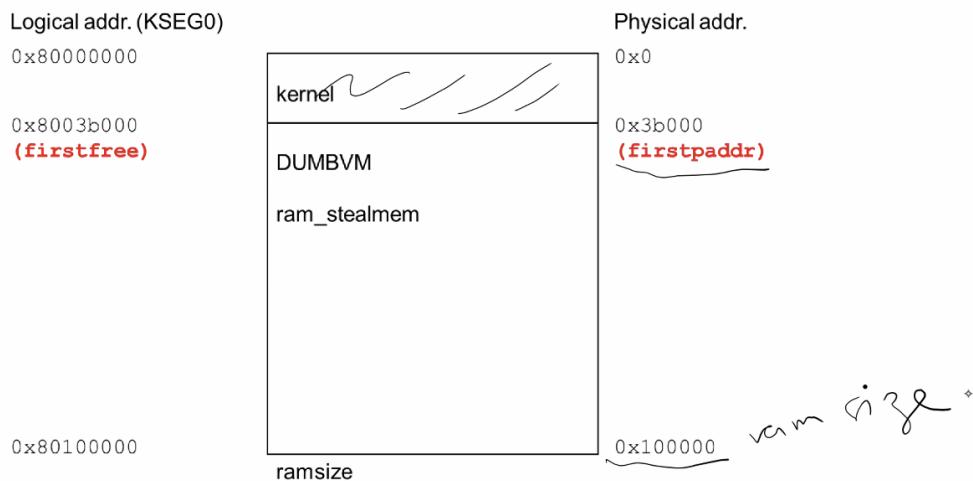
## Kernel loader (sys161: start.S)

Logical addr. (KSEG0)	Physical addr.
0x80000000	0x0
0x80000200	0x200
0x80039d54 (_end)	0x39d54
0x8003a000 (P)	0x3a000
0x8003b000 <b>(firstfree)</b>	0x3b000 <b>(firstpaddr)</b>
0x80100000	0x100000

At the end of the boot phase, OS saves the first available physical address (Free Memory)

OS161 salva il primo indirizzo di memoria disponibile.

# Dumbvm



Complete layout of physical memory:

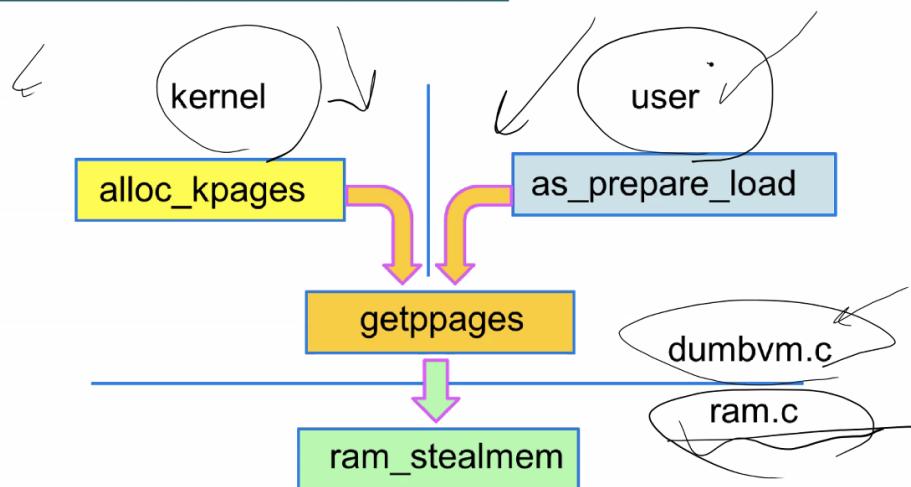
Starting from the first free address (first physical address), arriving to the dimension of the memory.

The first free address depends on the size of the kernel while the last depends on the ram size.

La memoria libera viene gestita tramite `dumbvm.c` e `ram.c`

Vediamo una overview di questi due sistemi.

## Dumbvm.c (alloc)



Memory manager architecture of OS161:

**ram\_stalmem** for memory management at low level (architecture dependent)

**getppages** for memory management in dumbvm.c

Allocation function for user side (**as\_prepare\_load**) and kernel side (**alloc\_kpages**)

Quando arriva una richiesta di allocazione di memoria, i 2 sistemi si occupano di gestirla.

## ram\_bootstrap

## ram\_bootstrap

```
void
ram_bootstrap(void) {
    /* Get size of RAM. */
    size_t ramsize = mainbus_ramsize(); if
    (ramsize > 512*1024*1024) {
        ramsize = 512*1024*1024;
    }
    lastpaddr = ramsize;
    /* Get first free virtual address from where start.S saved
     * it. Convert to physical address. */
    firstpaddr = firstfree - MIPS_KSEG0;
}
```

During the boot, the **ram\_bootstrap** is called which is initializing different parameters such as **ramsize** using function **mainbus\_ramsize**, saving **lastpaddr** as the ram size, and calculating the first free address with respect to KSEG0.

Dobbiamo sapere **dove inizia** la memoria libera e dove **finisce**.

Quando il kernel viene caricato, viene lanciata un'altra funzione (  
`ram_bootstrap(void)`) che chiede al controller della RAM quant'è la dimensione della RAM (`mainbus_ramsize()`) e la memorizza nella variabile `ramsize`.  
Poi ricava l'indirizzo fisico di dove inizia la memoria libera.  
Ciò viene fatto sottraendo al primo indirizzo logico disponibile  
`firstfree` la costante `MIPS_KSEG0`, che equivale a 0x80000000.

## ram\_stalmem

## ram\_stalmem (kern/arch/mips/vm/ram.c)

```
paddr_t ram_stalmem(unsigned long npages) {
    paddr_t paddr;
    size_t size = npages * PAGE_SIZE;
    if(firstpaddr + size > lastpaddr) {
        return 0;
    }
    paddr = firstpaddr;
    firstpaddr += size;
    return paddr;
}
```

number  
ram.c  
↓  
ram-stalmem

In **ram.c**, the most important function is **ram\_stalmem** that allows to ask for memory.

**Npages** is the number of pages to ask the OS.

Calculating the size as the multiply of number of pages and page size.

La dimensione totale che un processo chiede quando deve essere allocato viene calcolata tramite numero di pagine necessarie \* page\_size (4KB).

## ram\_stalmem (kern/arch/mips/vm/ram.c)

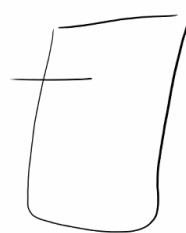
```
paddr_t ram_stalmem(unsigned long npages) {
    paddr_t paddr;
    size_t size = npages * PAGE_SIZE;
    if(firstpaddr + size > lastpaddr) {
        return 0;
    }
    paddr = firstpaddr;
    firstpaddr += size;
    return paddr;
}
```

Checking whether the first free address + the requested size is more than the last physical address. If yes, retuning 0 which shows failing in memory allocation

Se la memoria richiesta è più della memoria disponibile ritorna 0 (errore).

## **ram\_stalmem** (kern/arch/mips/vm/ram.c)

```
paddr_t ram_stalmem(unsigned long npages) {  
    paddr_t paddr;  
    size_t size = npages * PAGE_SIZE;  
    if(firstpaddr + size > lastpaddr) {  
        return 0;  
    }  
    paddr = firstpaddr;  
    firstpaddr += size;  
    return paddr;  
}
```



Checking whether the first free address + the requested size is more than the last physical address. If not, successful memory allocation, returning the address of the first available physical free memory.

Ram.c is regarding the architecture dependent part. For example, for changing from continues allocation to another one. this module should be changed.

Altrimenti assegna quella zona di memoria al processo, e quindi aggiorna l'*inizio* della memoria libera (spostandosi più avanti) e ritorna l'indirizzo di dove inizia lo spazio appena allocato.

## **getppages**

## Getppages (kern/arch/mips/vm/dumbvm.c)

```
static paddr_t
getppages(unsigned long npages) {
    paddr_t addr;
    spinlock_acquire(&stealmem_lock);
    addr = ram_stalmem(npages);
    spinlock_release(&stealmem_lock);
    return addr;
}
```

Getppages is for asking numbers of physical pages, asking for page number and returning pointer to the zone of the memory.

dumbvm.c è un wrapper di ram.c

Quindi chiama funzionalità di ram\_stalmem ma aggiunge anche altre funzionalità:

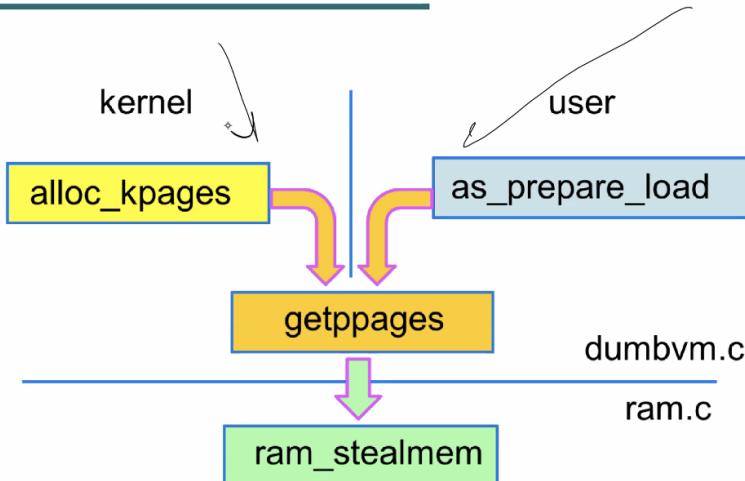
## Getppages (kern/arch/mips/vm/dumbvm.c)

```
static paddr_t
getppages(unsigned long npages) {
    paddr_t addr;
    spinlock_acquire(&stealmem_lock);
    addr = ram_stalmem(npages);
    spinlock_release(&stealmem_lock);
    return addr;
}
```

Spinlock allows the mutual exclusion in case of multiple process, to avoid corrupting of ram\_stalmem function.

Ad esempio utilizza uno spinlock — fa una mutual exclusion per dire che se processo 1 arriva prima e ha accesso, prende il lock e finché non finisce `ram_stealmem()` nessun altro processo può accedere a `ram_stealmem()`.

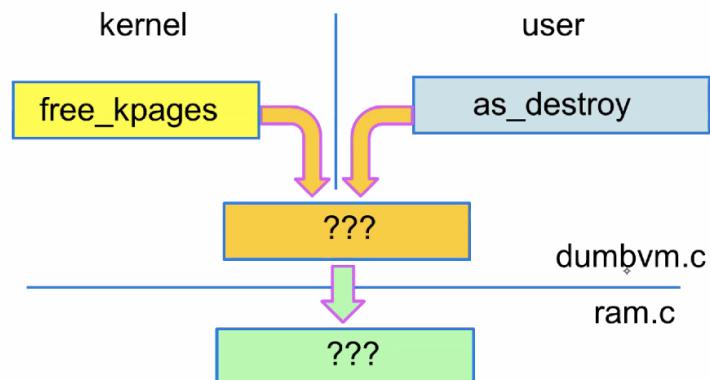
## Dumbvm.c (alloc)



Memory manager architecture of OS161:  
**ram\_stealmem** for memory management at low level (architecture dependent)  
**getppages** for memory management in dumbvm.c  
Allocation function for user side (**as\_prepare\_load**) and kernel side (**alloc\_kpages**)

Fino a qui, è tutto implementato su OS161.

## Dumbvm.c – Not implemented >> to do



The function for allocation of memory is implemented BUT the functions for releasing memory is not implemented.

**free\_kpages** and **as\_destroy** (equivalent of `alloc_kpages` and `as_prepare_load`) is defined but EMPTY.

**Your Job to solve this issue.**

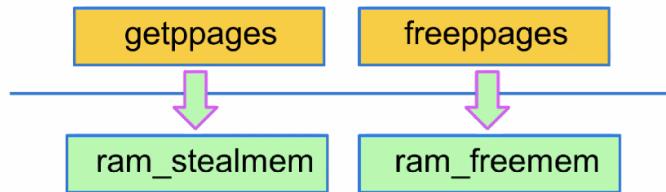
Da qui inizia il nostro lavoro: la parte di **liberare** la memoria manca.

OS161 attualmente sa come allocare memoria, ma non sa come liberarla.

## De-alloc (free) in ram.c

(solution A1)

- freeppages just an interface to ram\_freemem
- Data structure (free-list or bitmap) and **memory management in ram.c**



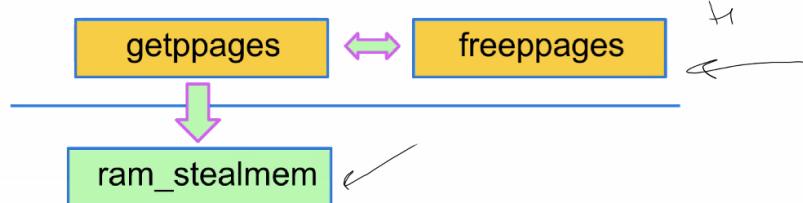
Defining the equivalent system of **ram\_stalmem** and **getppages**,  
Defining a **freeppages** that we be called by **free\_kpages** and **as\_destroy** (high level)  
that will call **ram\_freemem** for freeing memory.

Ad esempio possiamo creare 2 nuove funzioni: freeppages e ram\_freemem.

## De-alloc (free) in ram.c

(solution A2)

- Memory not returned to RAM
- Data structure (free-list o bitmap) and **memory management in dumbvm.c**
- Freepages coordinates with getppages

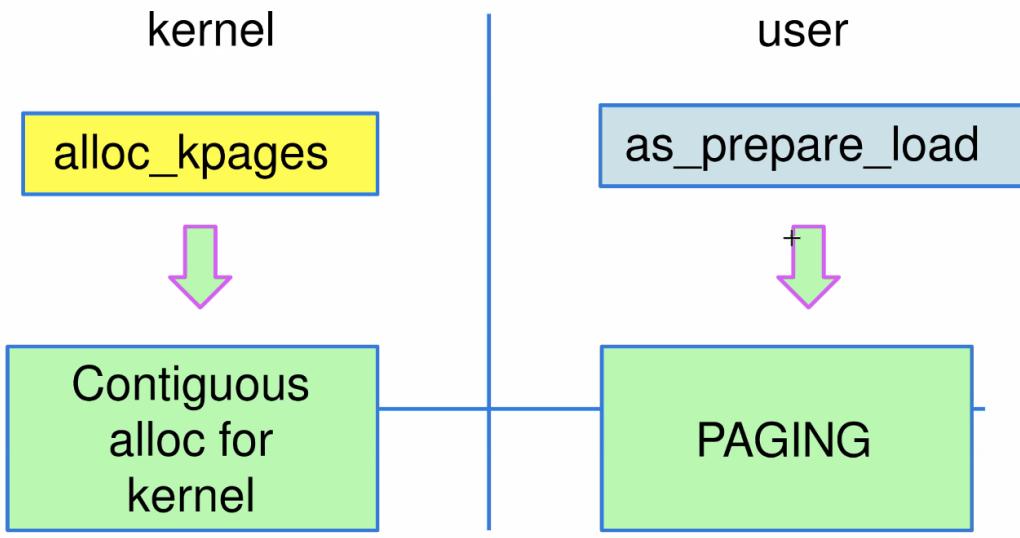


Managing the liberation of memory at low level but just at the dumbvm.c level.  
Every time that we are asking for memory allocation, before asking ram\_stalmem,  
looking if the memory page that has been already allocated, is free now.  
Vantages: working with one system call

Oppure possiamo lavorare solo high-level e lasciare ram\_stalmem inalterato.  
Questo è ciò che faremo in laboratorio (perchè è più semplice).

## Paging in user space

(solution B)



Tutto quello che esiste già, eliminiamo e ripartiamo da 0.

Perchè OS161 usa meccanismo di memoria contigua, quindi l'idea sarebbe implementare il meccanismo di paging vero e proprio e quindi riscrivere tutto. Soluzione più difficile, ovviamente.

## Solution A2

# Proposed Solution

## De-alloc in dumbvm (sol. A2)

- Contiguous allocation(by pages) common to kernel and user
- Allocator in dumbvm: keep track (using a bitmap) of previously freed pages. In order to alloc
  - When calling `getppages`, first search among (previously) freed pages (**an interval of contiguous free pages**)
  - If not found, call `ram_stealmem`
- Bitmap implemented as an array of char (for simplicity)
  - `freeRamFrames[i] = 1/0` (free/allocated): free=**FREED!** (by `freepages`)
- In order to free we need to know
  - Pointer (or index) to first page in interval
  - Size, i.e. number of (contiguous) pages to free
- We need a table to store sizes (number of pages in allocated intervals) for each alloc performed
  - `void free_kpages(vaddr_t addr)`: table needed as only pointer passed
  - `void as_destroy(struct addrspace *as)`: table not needed ad size is stored in address space
- `allocSize[i] = /* number of pages allocated starting at i-th */`

# Global Variables and Test Function

```
static struct spinlock freemem_lock = SPINLOCK_INITIALIZER;

static unsigned char *freeRamFrames = NULL;
static unsigned long *allocSize = NULL;
static int nRamFrames = 0;

static int allocTableActive = 0;

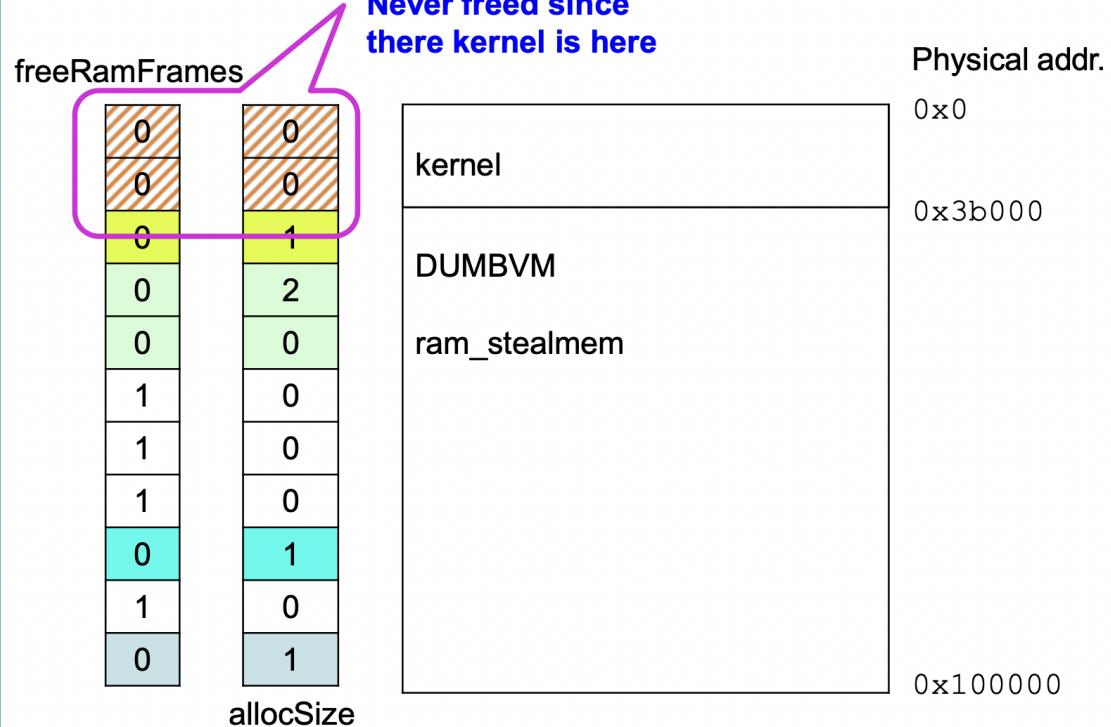
static int isTableActive () {
    int active;
    spinlock_acquire(&freemem_lock);
    active = allocTableActive;
    spinlock_release(&freemem_lock);
    return active;
}
```

# Global Variables and Test Function

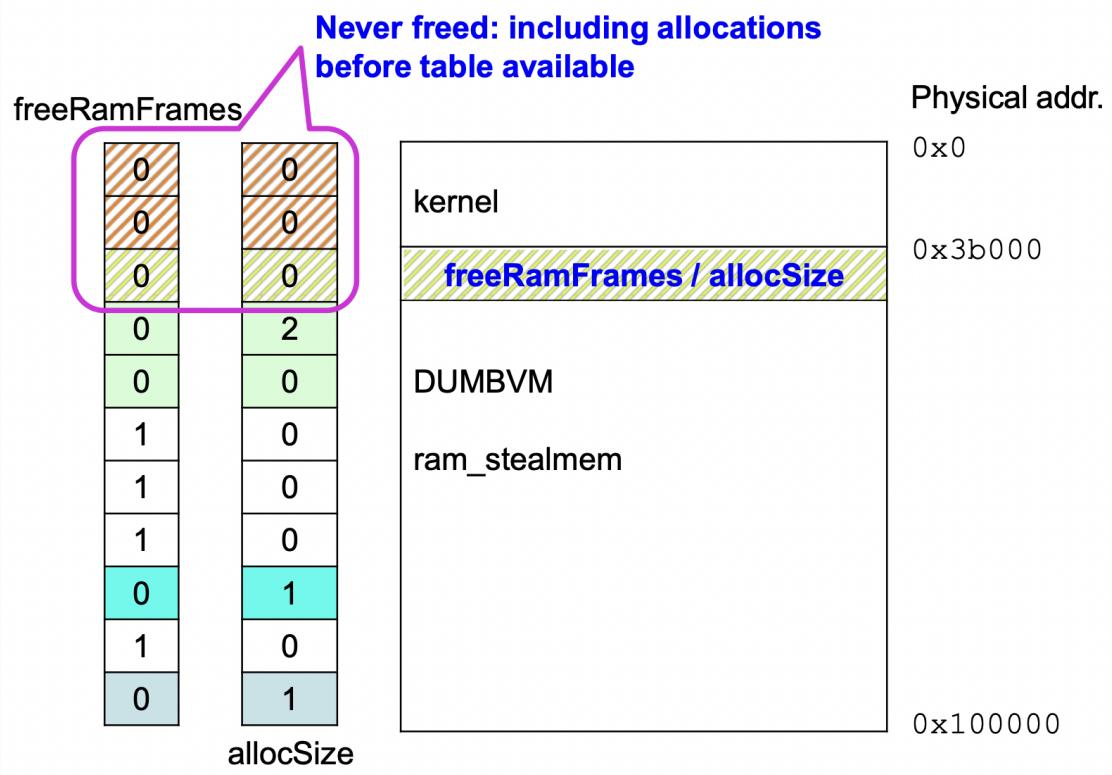
```
static struct spinlock freemem_lock = SPINLOCK_INITIALIZER;  
  
static unsigned char *freeRamFrames = NULL;  
static unsigned long *allocSize = NULL;  
static int nRamFrames = 0;  
  
static int allocTableActive = 0;  
  
static int isTableActive ()  
{  
    int active;  
    spinlock_acquire  
    active = allocT  
    spinlock_release  
    return active;  
}
```

Dynamic arrays as RAM size known at Boot  
(depends on da sys161.conf) Alternative: over-dimensioned static arrays!

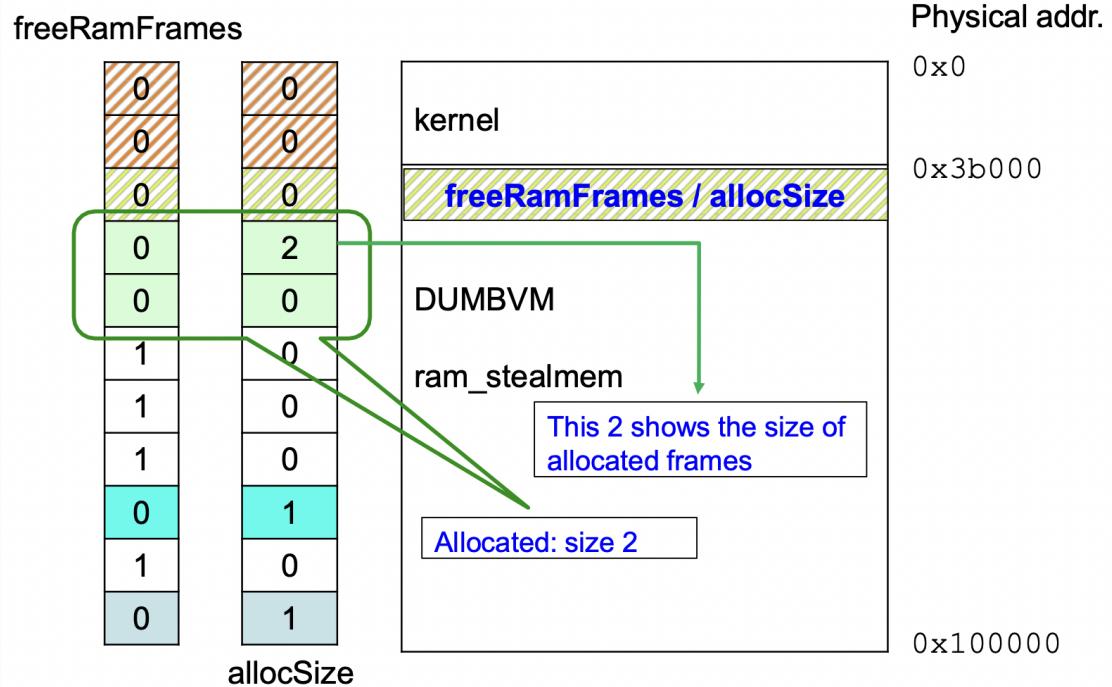
## Dumbvm



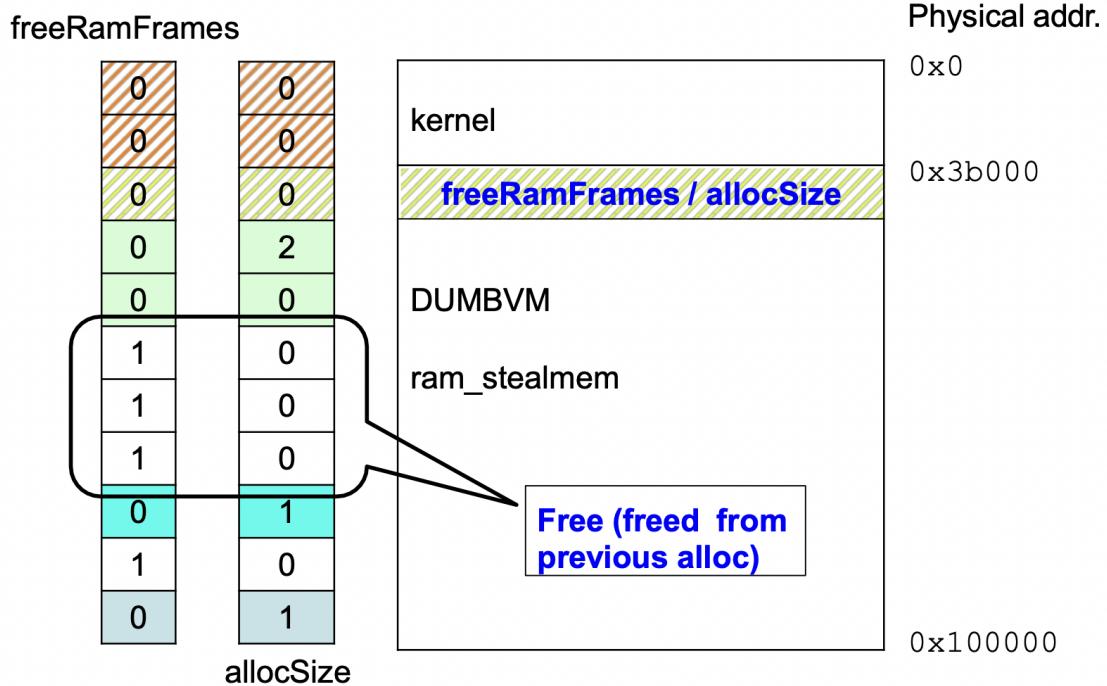
# Dumbvm



# Dumbvm



# Dumbvm



free\_kpages & as\_destroy

## Free\_kpages & as\_destroy

```
void free_kpages(vaddr_t addr){  if (isTableActive())  
{  
    paddr_t paddr = addr - MIPS_KSEG0; /* converting the  
    logical addr to physical*/  
    long first = paddr/PAGE_SIZE; /*determining number of page  
    in RAM*/  
    KASSERT(nRamFrames>first); /* check if the first address  
    is not bigger than ram*/  
    freepages(paddr, allocSize[first]); /* call freepages*/  
}  
}  
void as_destroy(struct addrspace *as){  dumbvm_can_sleep();  
    freepages(as->as_pbase1, as->as_npaged1);  
    freepages(as->as_pbase2, as->as_npaged2);  
    freepages(as->as_stackpbase, DUMBVM_STACKPAGES);  
    kfree(as);  
}
```

Implementazione delle due funzioni per liberare memoria, rispettivamente per il kernel e user space.

La funzione per il kernel,

`free_kpages`, innanzitutto ricava l'indirizzo fisico dall'indirizzo logico passato come parametro.

Da questo, ne ricava il corrispondente frame in RAM cui "punta".

Dopo di che, chiama

`freepages`.

## Free\_kpages & as\_destroy

```
void free_kpages(vaddr_t addr){  if (isTableActive())
{
    paddr_t  paddr  =  addr - MIPS_KSEG0;
    long first = paddr/PAGE_SIZE;
    KASSERT(nRamFrames>first);
    freeppages(paddr, allocSize[first]);
}
void as_destroy(struct addrspace *as){
    dumbvm_can_sleep();
    freeppages(as->as_pbase1, as->as_npaged1); /* returning the first segment
with the physical address and also the dimension(n page)*/
    freeppages(as->as_pbase2, as->as_npaged2); /* returning the second segment
with the physical address and also the dimension(n page)*/
    freeppages(as->as_stackpbase, DUMBVM_STACKPAGES); /* returning the stack
user with the dimension of DUMBVM_STACKPAGES which is a constant*/
    kfree(as);
}
```

La funzione per lo user, `as_destroy`, chiama tre volte `freeppages`, passando come parametri gli indirizzi dei segmenti allocati per il processo user (dunque segmento code, segmento data, segmento stack e relative dimensioni in pagine).

Dopo di che chiama

`kfree` per deallocare la struttura dell'address space del processo user.

# Initialization

```
void vm_bootstrap(void) {
    int i;
    nRamFrames = ((int)ram_getsize(z)) / PAGE_SIZE; /*total number of the frames*/
    freeRamFrames = kmalloc(sizeof(unsigned char) * nRamFrames);
    allocSize = kmalloc(sizeof(unsigned long) * nRamFrames);
    /*two vectors for bitmap*/
    if (freeRamFrames == NULL || allocSize == NULL) {
        /* reset to disable this vm management */
        freeRamFrames = allocSize = NULL;
        return;
    } /* initializing the two vectors*/
    for (i=0; i<nRamFrames; i++) {
        freeRamFrames[i] = (unsigned char)0; allocSize[i] = 0; /* 0 is not available*/
    }
    spinlock_acquire(&freemem_lock);
    allocTableActive = 1;
    spinlock_release(&freemem_lock);
}
```

L'inizializzazione del sistema della gestione della memoria deve quindi includere l'inizializzazione dei vettori che tengono conto di quali frames liberi abbiamo in RAM.

In particolare

`freeRamFrames` è una bitmap che associa ad ogni posizione del vettore un frame in ram, con un valore che può essere 0 (frame occupato) oppure 1 (frame libero).

`allocSize` è invece un vettore grande quanto `freeRamFrames`, e tiene traccia del numero di pagine allocate a partire dall'indice i-esimo.

Ad esempio, se avessimo allocato 2 pagine a partire dal frame associato all'indice 10, in

`allocSize[10]` troveremmo il valore 2; invece in `freeRamFrames[10]` e `freeRamFrames[11]` troveremmo degli 0, a segnare tali frames come occupati, appunto.

## getppages

```
Static paddr_t getppages(unsigned long npages) {
    paddr_t addr;
    /* try freed pages first */
    Addr = getfreepages(npages);
    if (addr == 0) { /* if there is not any memory available,
        then call stealmem*/
        spinlock_acquire(&stealmem_lock);
        addr = ram_stealmem(npages);
        spinlock_release(&stealmem_lock);
    }
    if (addr != 0 && isTableActive()) { /* if there is available
        memory, then allocsize*/
        spinlock_acquire(&freemem_lock);
        allocSize[addr/PAGE_SIZE] = npages;
        spinlock_release(&freemem_lock);
    }
    return addr;
}
```

Quando viene fatta una richiesta di allocazione di memoria, adesso `getppages` prima di chiamare `ram_stealmem` fa una prova con `getfreepages`, una "nuova" funzione che controlla se sono disponibili dei frames che erano stati precedentemente allocati (e magari resi disponibili al termine del processo o thread che li stava usando).

Ricorda che

`ram_stealmem` non fa altro che allocare memoria partendo dal primo indirizzo fisico libero disponibile (il quale viene aggiornato a seguito di ciascuna allocazione), senza tenere conto appunto se "prima" di tale indirizzo fisico si è liberato dello spazio.

Per questo intercediamo prima provando a chiamare `getfreepages`, che tramite i vettori che abbiamo implementato si occupa di gestire esattamente questo meccanismo.

## getfreepages

```

134     static paddr_t
135     getfreepages(unsigned long npages) {
136         paddr_t addr;
137         long i, first, found, np = (long)npages;
138
139         if (!isTableActive()) return 0;
140         spinlock_acquire(&freemem_lock);
141         for (i=0,first=-1; i<nRamFrames; i++) {
142             if (freeRamFrames[i]) {
143                 if (i==0 || !freeRamFrames[i-1])
144                     first = i; /* set first free in an interval */
145                 if (i-first+1 >= np) {
146                     found = first;
147                     break;
148                 }
149             }
150         }
151
152         if (found>=0) {
153             for (i=found; i<found+np; i++) {
154                 freeRamFrames[i] = (unsigned char)0;
155             }
156             allocSize[found] = np;
157             addr = (paddr_t) found*PAGE_SIZE;
158         }
159         else {
160             addr = 0;
161         }
162
163         spinlock_release(&freemem_lock);
164
165         return addr;
166     }

```

La funzione `getfreepages` prende come parametro il numero di pagine che vorremmo allocare.

Inizia quindi a scorrere il vettore

`freeRamFrames` finchè non troviamo una posizione settata ad 1, cioè frame libero.

Quando questo viene trovato, viene memorizzato l'indice come

`first`, cioè abbiamo trovato la prima posizione a partire dalla quale potremmo trovare dei frames liberi.

Dobbiamo vedere però, se a partire da tale posizione, abbiamo tanti frames liberi quanti ce ne servono (`npages`).

Questo viene fatto tramite il secondo if, che ad ogni iterazione del for controlla se l'indice che stiamo esaminando è maggiore o uguale a

`np`, vale a dire se abbiamo trovato `np` frames liberi.

Se è così, allora usciamo dal ciclo e segniamo questi frames come ora occupati (li stiamo prendendo noi), e segniamo

`np` nella posizione `found` di `allocSize` (cioè segniamo quante pagine stiamo allocando a partire dal frame di indice `found`).

Dopo di che calcoliamo l'indirizzo da restituire, e lo restituiamo.

---