Access to High Memory

Introduction to the kernel functions
‘kmap()’ and ‘kunmap()

1GB ram?: too many ‘pages’!

- Some hardware needs virtual addresses (e.g., network and video display cards)
- CPU also needs some virtual addresses (e.g., for interprocessor communication)

- So not enough virtual addresses remain when system has a gigabyte or more of ram

Linux kernel's 'solution'

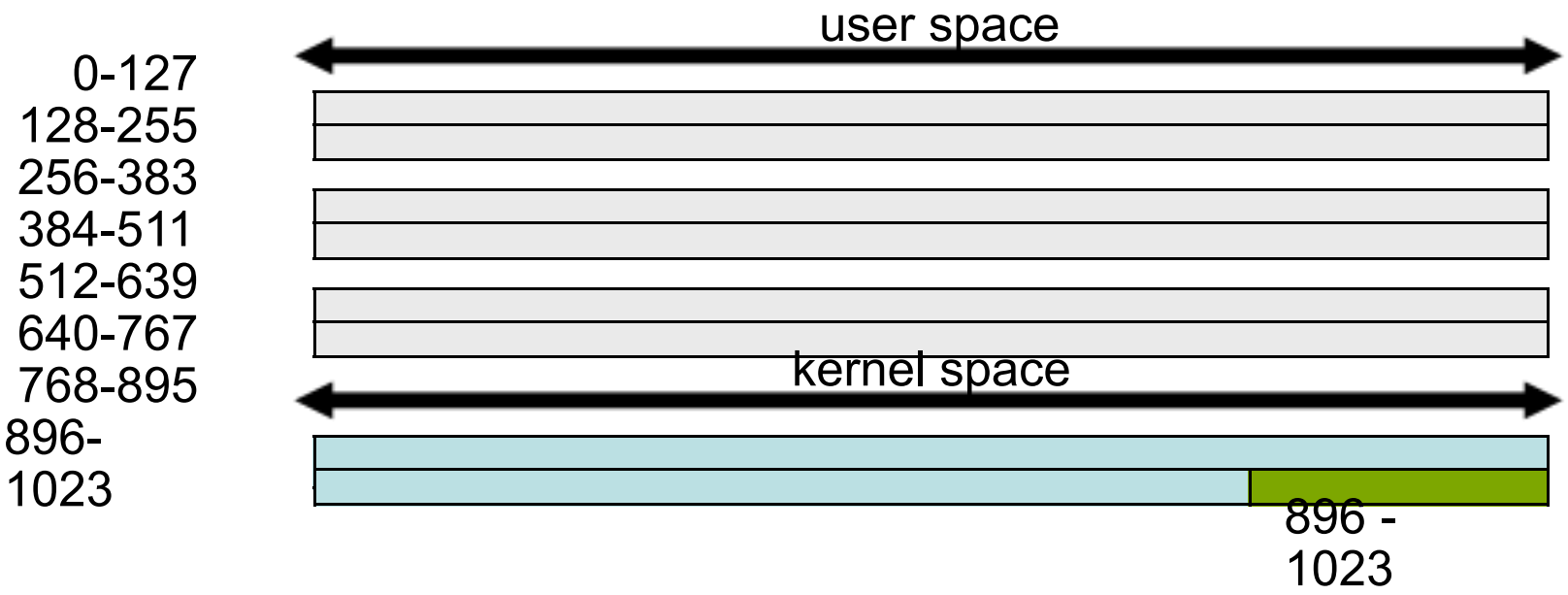
- The kernel-space addresses are 'shared'
- The high-memory page-frames are made visible to the CPU (i.e., are 'mapped') only while access to them is actually needed
- Then these page-frames are 'unmapped' so that other page-frames can 'reuse' the same virtual address
- But sharing only affects 'high memory'

kernel mapping function

- ‘kmap()’ can create new page-mappings
- It works on a page-by-page basis
- ‘kunmap()’ can invalidate these mappings
- Prototypes: #include <linux/highmem.h>
- void * kmap(struct page * page);
- void kunmap(struct page * page);

Visualizing kernel-space

- Memory below 896MB is ‘directly mapped’
- ‘phys_to_virt()’ just adds 0xC0000000
- ‘virt_to_phys()’ subtracts 0xC0000000
- Master Page Directory provides a template



Recall ‘mem_map[ ]’ array

```
unsigned long    num_physpages;
struct page *mem_map;
struct page *highmen_start_page;

highmem_start_page = &mem_map[ 896 ];

struct page { … ; void * virtual; };
```

How ‘kmap()’ works

```
void * kmap( struct page * page )
{
        if ( page < highmem_start_page )
            return  page->virtual;
        return  kmap_high( page );
}
```

How does ‘kmap_high()’ work?

- Kernel first performs a quick check:
- Is the high page is already ‘mapped’?
- If so, just return its existing virtual address
- If not, find an ‘unused’ virtual address
- Set page-table entry that ‘maps’ the page
- BUT…

… there might not be any ‘unused’ entries!

Details of ‘kmap_high()’

```
void kmap_high( struct page * page )
{
   unsigned long    vaddr;
   spin_lock( &kmap_lock );
   vaddr = (unsigned long)page->virtual;
   if ( !vaddr ) vaddr = map_new_virtual( page );
   ++pkmap_count[ (vaddr - PKMAP_BASE)>>12 ];
   spin_unlock( &kmap_lock );
   return    (void *)vaddr;
}
```

Task maybe will sleep

- If a new kernel mapping cannot be set up (because all addresses are already in use) then the ‘map_new_virtual()’ function will ‘block’ the task (i.e., will put it to sleep)

How process gets blocked

```
DECLARE_WAITQUEUE( wait, current );

current->state = TASK_UNINTERRUPTIBLE;
add_wait_queue(&pkmap_map_wait,&wait);
spin_unlock( &kmap_lock );
schedule();
remove_wait_queue(&pkmap_map_wait,&wait);
spin_lock( &kmap_lock );
if ( page->virtual ) return  page->virtual;
```

Purpose of ‘kunmap()’

- Programming sequence:

```
void  * vaddr = kmap( page );      // get address
   …
/* use this virtual address to access the page */
   …
kunmap( page );           // release the address
```

- Now kernel can awaken ‘blocked’ tasks

Reference counters

- Kernel maintains an array of counters
  static int pkmap_count[LAST_PKMAP];
- One counter for each ‘high memory’ page
- Counter values are 0, 1, or more than 1
- =0: page is not mapped
- =1: page not mapped now, but used to be
- =n >1: page was mapped (n-1) times

Flushing the TLB

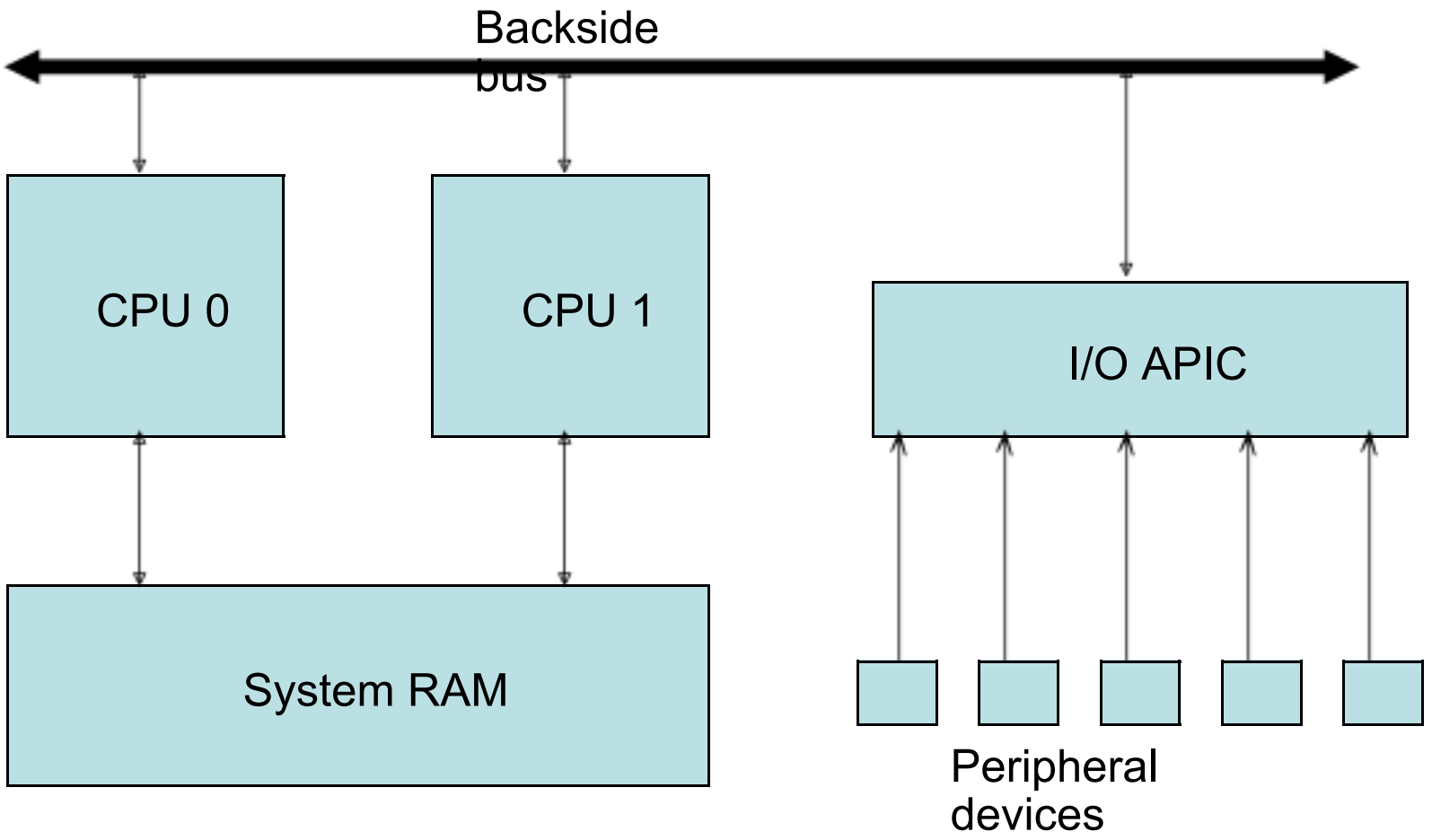
- ‘kunmap(page)’ decrerments refcount
- When refcount == 1, mapping isn’t needed
- But CPU still has ‘cached’ that mapping
- So the mapping must be ‘invalidated’
- With multiple CPUs, all of them must do it

Special CPU instruction

- Pentium has a special instruction:
  invlpg m ; invalidates one TLB entry

- But it only affects CPU that executes it.
- In SMP systems, aother CPUs may still have ‘stale’ physical-to-virtual address-mapping cached in their TLB
- Prevents other CPU from uptodate access

Solution: notify other CPUs



What about interrupts?

- Interrupt-handlers can’t risk using ‘kmap()’
- Yet may need access to high memory data
- Data might not be ‘mapped’ when needed!
- So an ‘emergency’ mechanism is required

Temporary Mappings

- A small number of ‘windows’ are reserved
- These are ‘emergency’ page-table entries
- Just five or six such entries (per CPU)
- Interrupt-handlers can use these safely
- A temporary page-mapping is set up fast
- No checking for other possible users
- No assurance that mapping will persist

‘kmap_atomic()’

- May be safely called by interrupt-handlers
- No need to do ‘unmapping’
- No TLB clashes with other CPUs
- You can experiment with ‘kmap_atomic()’
- But use ‘kmap()/kunmap()’ for project #3