

# *CS5460: Operating Systems*

## ***Project 3 Tutorial -- some hints***



# *Goal*

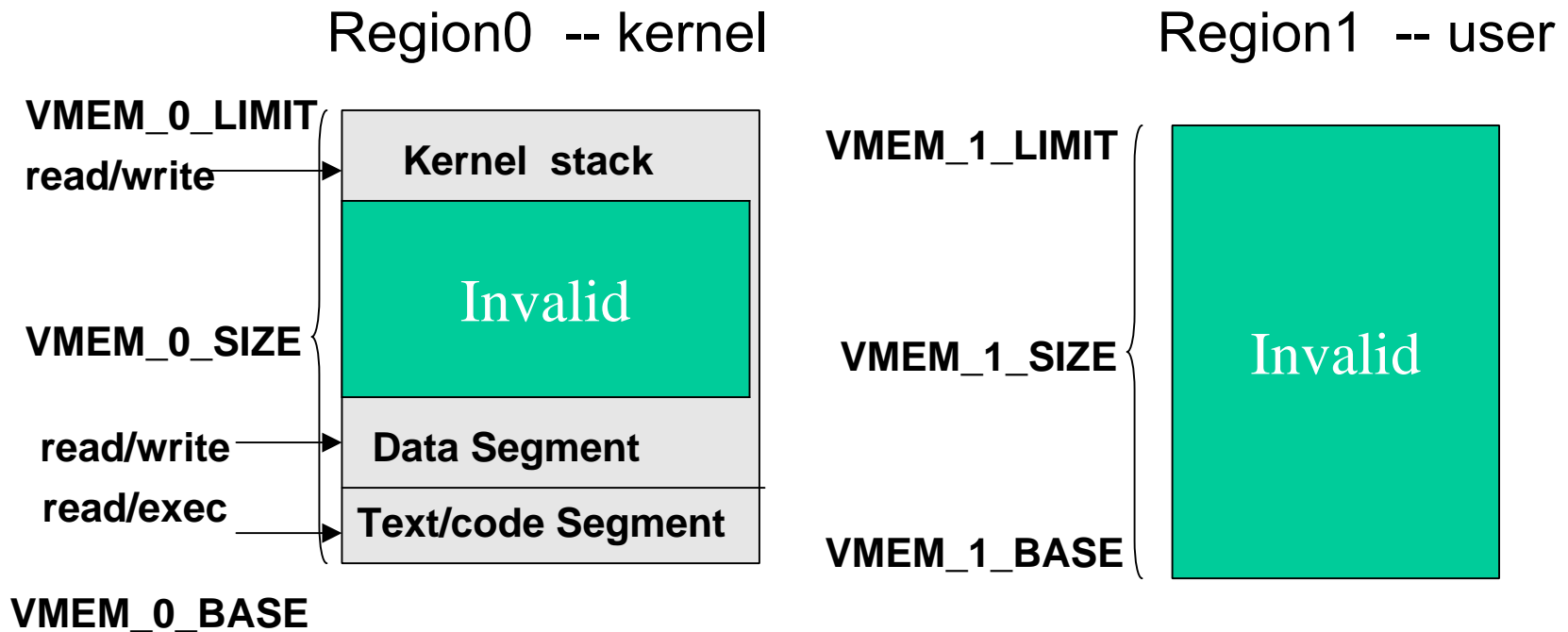
- **Support virtual memory**
- **Load and execute multiple independent user-level program**

# *KernelStart-- boot Yalnix*

- 1. Initialize memory**
- 2. Initialize IO**
- 3. Set up Trap Handler table**
- 4. Initialize PCB data structures**
- 5. Call '*LoadProgram*' to load program into memory, initialize processes**
- 6. Context switch to some process and run**

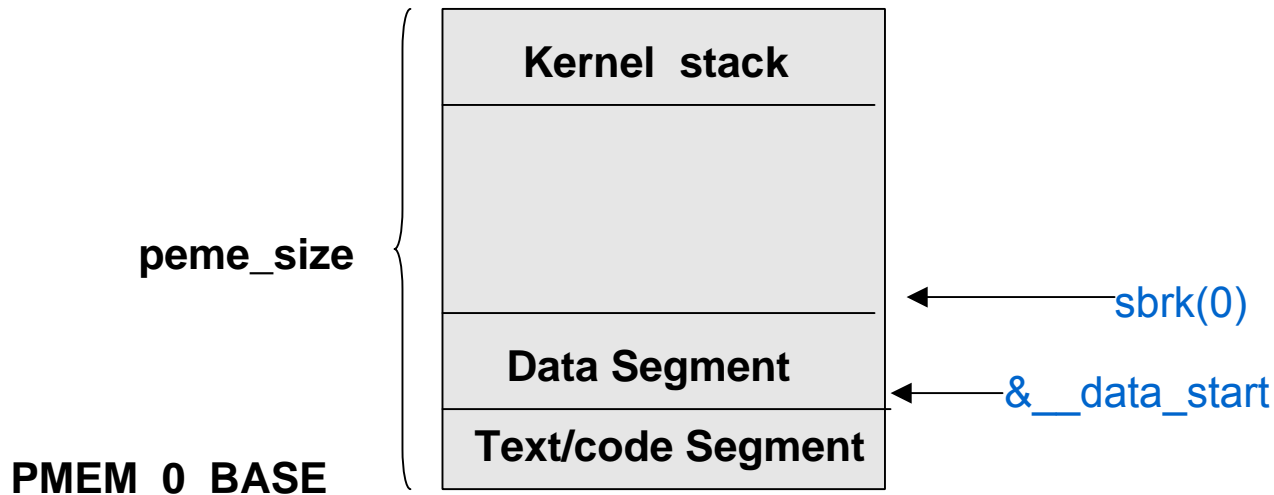
# Yalnix Virtual Memory Organization

- Paged memory scheme using page tables
- Two regions (two modes)
- Initial states



# *Yalrix Physical Memory Organization*

- **PMEM\_BASE == VEME\_0\_BASE !**
- **Initial states**
- **Need manage free frames**



# *Translate Addresses into Page Numbers*

- **Page numbers in regions**
  - Why? Page table sizes
  - How? `VMEM_0_SIZE >> PAGESHIFT`
- **Page number and Page index in the kernel stack**
  - Fixed size
  - Every process has one kernel stack
  - Why? A Process need loads its kernel stack pages into the kernel page table upon context switching
  - How?
    - » Kernel stack page number  
(`KERNEL_STACK_MAXSIZE >> PAGESHIFT`)
    - » Kernel stack start-page index (`KERNEL_STACK_BASE >> PAGESHIFT`)
    - » More: allocate physical frames for kernel stack pages during the PCB construction

# *Find the boundaries*

- **Macros: UP\_TO\_PAGE(x), DOWN\_TO\_PAGE(x)**
  - Round the address  $x$  to the page boundary.
  - If  $x$  is on a page boundary, it returns  $x$ .
  
- **Example: last data page Index**
  - `UP_TO_PAGE(sbrk(0)) >> PAGESHIFT - 1`

# *Memory Initialization*

- Page tables are defined in region 0
- Things to consider
  1. *Initialize the page table for region 0*
    - Code (Text) pages, data pages, stack pages
    - set related register(REG\_PTBR0, REG\_PTLR0)
  2. *Initialise the page table for the current user process (optional) (REG\_PTBR1, REG\_PTLR1)*
  3. *Initialize free frames pool*
    - Stack, list, queue ....
  4. *Enable virtual memory*
    - REG\_VM\_ENABLE, REG\_TLB\_FLUSH

# *I/O Initialization*

- **Kernel must buffer the I/O data**
- **I/O buffers cannot have fixed size – buffers are not restricted by the available memory**
- **Things to consider**
  - Read/write(may not need) buffers for each terminal
  - Read/write waiting queues for each terminal
  - Synchronization mechanism for the waiting queues

# *Set up Trap Handler Table*

**You already know  
Or  
you are in trouble 😊**

# ***PCB Initialization***

- **What should be kept? things to consider:**
  - **Pid**
  - **Status**
  - **Timeout (used for Delay system call)**
  - **Kernel context**
  - **User context**
  - **User Page table**
  - **Kernel stack frame numbers**
  - **User break**
  - **Pointers to previous, next PCB (link PCBs as a bi-direction list)**
- **PCB queues**

# *Loading Programs*

- **Template: load.templates**
- **Read the instructions and customize it for your use**
- **What could happen when loading a program?**
  - Find out the code segment, data segment, stack sizes
  - allocates the pages for user text, data and stack segments
  - Fill the user page table and make it effective
  - Copy data into allocated frames
  - Update pc, sp, brk, data\_start
  - Restore the original user page table

# Context Switching

- **Where and when?**

- End of the KernelStart()
- The Clock Trap Handler
- Anywhere you need.

- **Kernel Context Switching**

- Where you can copy the kernel context

`KernelContext *MyKCS(KernelContext *, void *, void *)`

- What could happen in your call-back?

- » Load and flush kernel stack pages
- » Save current user context to the PCB to be switched to
- » More

- **User Context Switching**

- Save anything you need
- Load and flush the user page table
- Update PCB queue
- Update the current user context

# *SetKernelBrk()*

- **Allocate or deallocate memory in kernel space**
  - Check the same page
  - Check the available virtual and physical memory
  - Allocate frames and set the appropriate page entries
  - Release frames and update page entries on deallocation
- ***Report error on special cases***
  - *Out of virtual memory*
  - *Out of physical memory*

# System Calls

- System call is triggered via TRAP\_KERNEL

Switch (conext->code)  
case ...

- Get parameters from UserContext->regs[]
- Return value set to UserContext->regs[0]
- In project 3, we need to implement
  - TtyRead
  - TtyWrite
  - Brk
  - Exit
  - Delay

# *TtyRead/TtyWrite & Tty Trap Handlers*

- **Things to consider**
  - Get the tty id from `UserContext->Code`
  - Check the validity of any data buffer!
  - Send in the unit of `TERMINAL_MAX_LINE`, receive is restricted by the parameter “len”
  - Copy data to or from the kernel memory space
  - Hardware operations: `TtyReceive()`, `TtyTransmit()`
- **Producer and consumer**
  - The receive trap handler and tty read system call
  - Tty write system call and `TtyTransmit()`
- **Synchronization**

# *Brk*

- **Things to consider**
  - **Check address validity**
    - » You need ubrk: where is it stored?
    - » You need user stack base
  - **Other things noted in SetKernelBrk()**

# *Delay and Exit*

- **Delay**

- You may need a delay queue for this
- Check the time very clock interrupt

- **Exit**

- Release resources
- A special context switching scenario

# *Global variables*

- **Kernel break**
- **System clock**
- ***Process id counter***
- ***PCB queues***
- ***IO buffers***
- ***Free memory page pool***
- ***...***

# *Testing*

- Write some small programs
- Compile it with the help of Makefile.usr
- Run the program
  - **Yalnix prog1 prog2 ...**

# Q & A

