



# CS 3204 Operating Systems

## Lecture 21 Godmar Back



## Announcements


- Project 2 due **Wed March 22**:
- You should have read all documentation by now & probably implemented steps suggested in Section 4.2 (including basic syscall framework)
- Key parts:
  - argument passing and proper synchronization between parent & child on startup
  - exit()/wait() synchronization
  - other syscalls, including file descriptor management (a bit repetitive)
  - robustness: verifying user addresses – you can do this last, but think it through first
  - robustness: proper termination of misbehaved processes
- Midterm **Fri March 24**
- Project 3 page table design document due **Mon March 27**
  - More specific information to follow
- Reminder: minimum requirement to pass the class is a working project 2
  - reread section on *Grading* in Syllabus
  - “working” means a >95% score as reported by “make grade”
  - project 2 is required for projects 3 & 4
- Reading assignments:
  - Stallings Chapter 7.1-7.4, 8.1-8.2



CS 3204 Spring 2006    3/13/2006    2


# Virtual Memory

## Page Tables & TLB



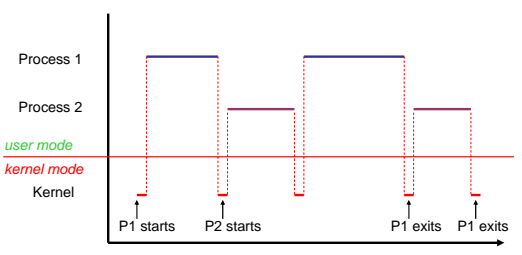
## Goals for Virtual Memory

- Virtualization
  - Maintain illusion that each process has entire memory to itself
  - Allow processes access to more memory than is really in the machine (or: sum of all memory used by all processes > physical memory)
- Protection
  - make sure there’s no way for one process to access another process’s data



CS 3204 Spring 2006    3/13/2006    4


## Context Switching



Process 1  
Process 2

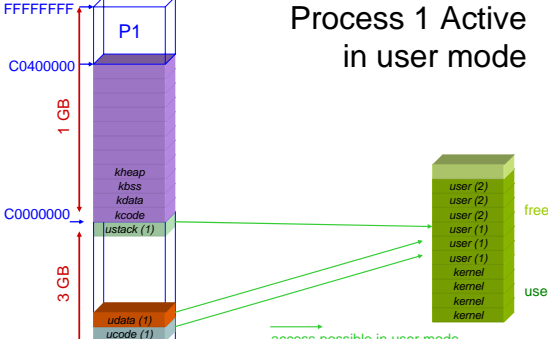
user mode  
kernel mode  
Kernel

P1 starts    P2 starts    P1 exits    P1 exits



CS 3204 Spring 2006    3/13/2006    5

## Process 1 Active in user mode



FFFFFFF  
C0400000  
1 GB  
C0000000  
3 GB  
0

P1


heap  
kbss  
kdata  
kcode  
ustack(1)

udata(1)  
ucode(1)

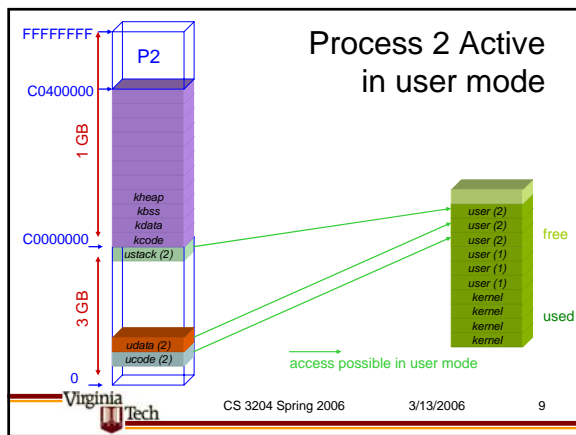
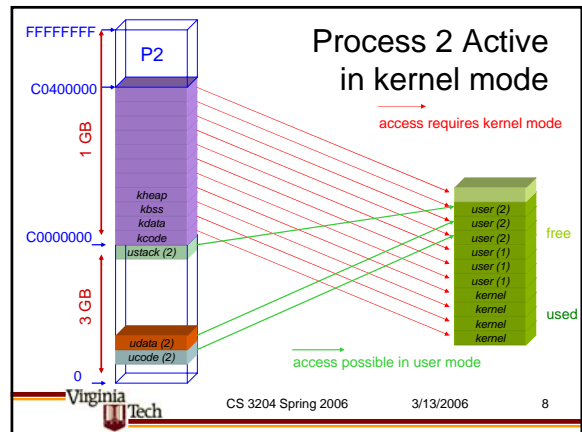
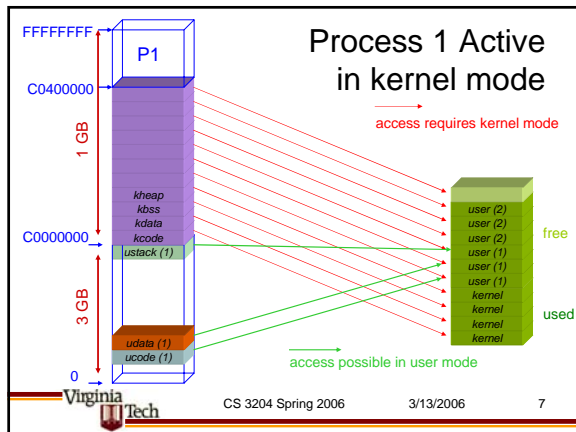
user(2)  
user(2)  
user(1)  
user(1)  
kernel  
kernel

free  
used

access possible in user mode



CS 3204 Spring 2006    3/13/2006    6



### Page Tables

- How are the arrows in previous pictures represented?
  - Page Table: mathematical function "Trans"

Trans:  
 $\{ \text{Process Ids} \} \times \{ \text{Virtual Addresses} \} \times \{ \text{user, kernel} \} \times \varphi(\{ \text{read, write, execute} \})$   
 $\rightarrow \{ \text{Physical Addresses} \} \cup \{ \text{INVALID} \}$

- Typically have
  - $\text{Trans}(p_i, v_a, \text{user}, *) = \text{Trans}(p_i, v_a, \text{kernel}, *)$
  - OR
  - $\text{Trans}(p_i, v_a, \text{user}, *) = \text{INVALID}$
  - User virtual addresses can be accessed in kernel mode

Virginia Tech CS 3204 Spring 2006 3/13/2006 10

### Sharing

- We get user-level sharing between processes  $p_1$  and  $p_2$  if
  - $\text{Trans}(p_1, v_a, \text{user}, *) = \text{Trans}(p_2, v_a, \text{user}, *)$
- Shared physical address doesn't need to be mapped at same virtual address, could be mapped at  $v_a$  in  $p_1$  and  $v_b$  in  $p_2$ :
  - $\text{Trans}(p_1, v_a, \text{user}, *) = \text{Trans}(p_2, v_b, \text{user}, *)$
- Can also map with different permissions: say  $p_1$  can read & write,  $p_2$  can only read
  - $\text{Trans}(p_1, v_a, \text{user}, \{ \text{read, write} \}) = \text{Trans}(p_2, v_b, \text{user}, \{ \text{read} \})$
- In Pintos (and many OS) the kernel virtual address space is shared among all processes & mapped at the same address:
  - $\text{Trans}(p_i, v_{a_i}, \text{kernel}, *) = \text{Trans}(p_k, v_{b_i}, \text{kernel}, *)$  for all processes  $p_i$  and  $p_k$  and  $v_a$  in  $[0xC0000000, 0xFFFFFFFF]$

Virginia Tech CS 3204 Spring 2006 3/13/2006 11

### Per-Process Page Tables

- Can either keep track of all mappings in a single table, or can split information between tables
  - one for each process
  - mathematically: a projection onto a single process
- For each process  $p_i$  define a function  $PTrans_i$  as
  - $PTrans_i(v_a, *, *) = \text{Trans}(p_i, v_a, \text{user}, *)$
- Implementation: associate representation of this function with PCB, e.g. per-process hash table
  - Entries are called "page table entries" or PTEs

Virginia Tech CS 3204 Spring 2006 3/13/2006 12

## Per-Process Page Tables (2)

- Common misconception
  - “User processes use ‘user page table’ and kernel uses ‘kernel page table’” – as if those were two tables
- Not so: mode switch (interrupt, system call) does not change the page table that is used
  - It only activates those entries that require kernel mode within the current process’s page table
- Consequence: kernel code also cannot access user addresses that aren’t mapped

## Non-Resident Pages

- When implementing virtual memory, some of a process’s pages can be swapped out
  - Or may not yet have been faulted in
- Need to record that in page table:

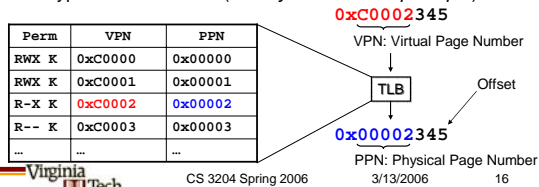
Trans (with paging):  
 $\{ \text{Process Ids} \} \times \{ \text{Virtual Addresses} \} \times \{ \text{user, kernel} \} \times \varphi(\{ \text{read, write, execute} \})$   
 →  $\{ \text{Physical Addresses} \} \cup \{ \text{INVALID} \} \cup \{ \text{Some Location On Disk} \}$

## Implementing Page Tables

- Many, many variants possible
- Done in combination of hardware & software
  - Hardware part: dictated by architecture
  - Software part: up to OS designer
    - Machine-dependent layer that implements architectural constraints (what hardware expects)
    - Machine-independent layer that manages page tables
- Must understand how TLB works first

## TLB: Translation Look-Aside Buffer

- Virtual-to-physical translation is part of every instruction (why not only load/store instructions?)
  - Thus must execute at CPU pipeline speed
- TLB caches a number of translations in fast, fully-associative memory
  - typical: 95% hit rate (*locality of reference principle*)



## TLB Management

- Note: on previous slide example, TLB entries did not have a process id
  - As is true for x86
- Then: if process changes, some or all TLB entries may become invalid
  - X86: flush entire TLB on process switch (refilling is expensive!)
  - NB: “flush” here means discard
- Some architectures store process id in TLB entry (MIPS)
  - Flushing (some) entries only necessary when process id reused

## Address Translation & TLB

