

CS162
Operating Systems and
Systems Programming
Lecture 11

Thread Scheduling (con't)
Protection: Address Spaces

February 23, 2010

Ion Stoica

<http://inst.eecs.berkeley.edu/~cs162>

Review: Last Time

- **Scheduling**: selecting a waiting process from the ready queue and allocating the CPU to it
- **FCFS Scheduling**:
 - Run threads to completion in order of submission
 - Pros: Simple (+)
 - Cons: Short jobs get stuck behind long ones (-)
- **Round-Robin Scheduling**:
 - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
 - Pros: Better for short jobs (+)
 - Cons: Poor when jobs are same length (-)

2/23/10

CS162 ©UCB Spring 2010

Lec 11.2

Goals for Today

- Finish discussion of Scheduling
- Kernel vs User Mode
- What is an Address Space?
- How is it Implemented?

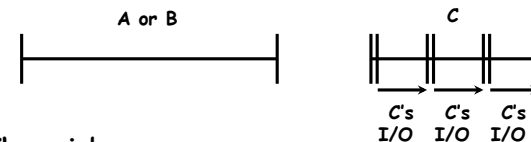
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

2/23/10

CS162 ©UCB Spring 2010

Lec 11.3

Example to illustrate benefits of SRTF

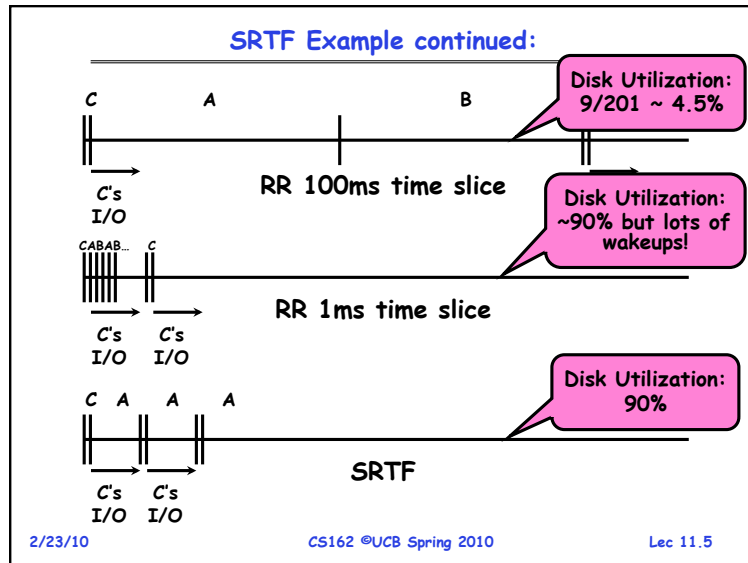


- Three jobs:
 - A, B: both CPU bound, run for week
 - C: I/O bound, loop 1ms CPU, 9ms disk I/O
 - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
 - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
 - Easier to see with a timeline

2/23/10

CS162 ©UCB Spring 2010

Lec 11.4



Review: SRTF Further discussion

- **Starvation**
 - SRTF can lead to starvation if many small jobs!
 - Large jobs never get to run
- **Somehow need to predict future**
 - How can we do this?
 - Some systems ask the user
 - » When you submit a job, have to say how long it will take
 - » To stop cheating, system kills job if takes too long
 - But: Even non-malicious users have trouble predicting runtime of their jobs
- **Bottom line, can't really know how long job will take**
 - However, can use SRTF as a yardstick for measuring other policies
 - Optimal, so can't do any better
- **SRTF Pros & Cons**
 - Optimal (average response time) (+)
 - Hard to predict future (-)
 - Unfair (-)

2/23/10 CS162 ©UCB Spring 2010 Lec 11.6

Predicting the Length of the Next CPU Burst

- **Adaptive:** Changing policy based on past behavior
 - CPU scheduling, in virtual memory, in file systems, etc
 - Works because programs have predictable behavior
 - » If program was I/O bound in past, likely in future
 - » If computer behavior were random, wouldn't help
- **Example: SRTF with estimated burst length**
 - Use an estimator function on previous bursts:
 - Let $t_{n-1}, t_{n-2}, t_{n-3}, \dots$ be previous CPU burst lengths.
 - Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \dots)$
 - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
 - For instance, **exponential averaging**
 - $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$
 - with $(0 < \alpha \leq 1)$

CPU burst (t_i)	6	4	6	4	13	13	13	...
guess* (τ_i)	10	8	6	6	5	9	11	12

2/23/10 CS162 ©UCB Spring 2010 Lec 11.7

Multi-Level Feedback Scheduling

- **Another method for exploiting past behavior**
 - First used in CTSS
 - **Multiple queues, each with different priority**
 - » Higher priority queues often considered "foreground" tasks
 - **Each queue has its own scheduling algorithm**
 - » e.g. foreground - RR, background - FCFS
 - » Sometimes multiple RR priorities with quantum increasing exponentially (highest: 1ms, next: 2ms, next: 4ms, etc)
- **Adjust each job's priority as follows (details vary)**
 - Job starts in highest priority queue
 - If timeout expires, drop one level
 - If timeout doesn't expire, push up one level (or to top)

2/23/10 CS162 ©UCB Spring 2010 Lec 11.8

Scheduling Details

- Result approximates SRTF:
 - CPU bound jobs drop like a rock
 - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
 - **Fixed priority scheduling:**
 - » serve all from highest priority, then next priority, etc.
 - **Time slice:**
 - » each queue gets a certain amount of CPU time
 - » e.g., 70% to highest, 20% next, 10% lowest
- **Countermeasure:** user action that can foil intent of the OS designer
 - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
 - Of course, if everyone did this, wouldn't work!
- Example of Othello program:
 - Playing against competitor, so key was to do computing at higher priority the competitors.
 - » Put in printf's, ran much faster!

2/23/10

CS162 ©UCB Spring 2010

Lec 11.9

Administrivia

- Midterm I coming up in two weeks!
 - Tuesday 3/9, 3:30-6:30 (this room)
 - Should be 2 hour exam with extra time
 - Closed book, one page of hand-written notes (both sides)
- No class on day of Midterm
 - I will post extra office hours for people who have questions about the material (or life, whatever)
- Midterm Topics
 - Everything up to (and including) Thursday (3/4)
 - History, Concurrency, Multithreading, Synchronization, Protection/Address Spaces/TLBs

2/23/10

CS162 ©UCB Spring 2010

Lec 11.10

Scheduling Fairness

- What about fairness?
 - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
 - » long running jobs may never get CPU
 - » In Multics, shut down machine, found 10-year-old job
 - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
 - **Tradeoff: fairness gained by hurting avg response time!**
- How to implement fairness?
 - Could give each queue some fraction of the CPU
 - » What if one long-running job and 100 short-running ones?
 - » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
 - Could increase priority of jobs that don't get service
 - » What is done in UNIX
 - » This is ad hoc—what rate should you increase priorities?
 - » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority⇒Interactive jobs suffer

2/23/10

CS162 ©UCB Spring 2010

Lec 11.11

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
 - Give each job some number of lottery tickets
 - On each time slice, randomly pick a winning ticket
 - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
 - To approximate SRTF, short running jobs get more, long running jobs get fewer
 - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
 - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses



2/23/10

CS162 ©UCB Spring 2010

Lec 11.12

Lottery Scheduling Example

Lottery Scheduling Example

- Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

- What if too many short jobs to give reasonable response time?
 - » In UNIX, if load average is 100, hard to make progress
 - » One approach: log some user out

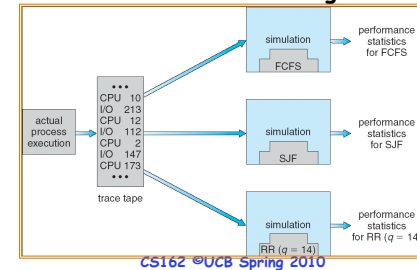
2/23/10

CS162 ©UCB Spring 2010

Lec 11.13

How to Evaluate a Scheduling algorithm?

- Deterministic modeling
 - takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
 - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
 - Build system which allows actual algorithms to be run against actual data. Most flexible/general.



2/23/10

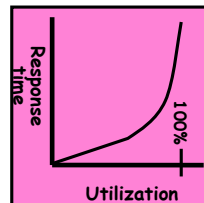
CS162 ©UCB Spring 2010

Lec 11.14

A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
 - When there aren't enough resources to go around
- When should you simply buy a faster computer?
 - (Or network link, or expanded highway, or ...)
 - One approach: Buy it when it will pay for itself in improved response time

- » Assuming you're paying for worse response time in reduced productivity, customer angst, etc...
- » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization \Rightarrow 100%



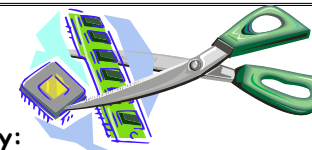
- An interesting implication of this curve:
 - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
 - Argues for buying a faster X when hit "knee" of curve

2/23/10

CS162 ©UCB Spring 2010

Lec 11.15

Virtualizing Resources



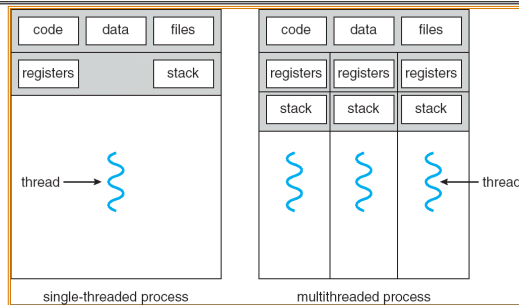
- Physical Reality: Different Processes/Threads share the same hardware
 - Need to multiplex CPU (Just finished: scheduling)
 - Need to multiplex use of Memory (Today)
 - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
 - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
 - Consequently, cannot just let different threads of control use the same memory
 - » Physics: two different pieces of data cannot occupy the same locations in memory
 - Probably don't want different threads to even have access to each other's memory (protection)

2/23/10

CS162 ©UCB Spring 2010

Lec 11.16

Recall: Single and Multithreaded Processes



- **Threads encapsulate concurrency**
 - "Active" component of a process
- **Address spaces encapsulate protection**
 - Keeps buggy program from trashing the system
 - "Passive" component of a process

2/23/10

CS162 ©UCB Spring 2010

Lec 11.17

Important Aspects of Memory Multiplexing

- **Controlled overlap:**
 - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
 - Conversely, would like the ability to overlap when desired (for communication)
- **Translation:**
 - Ability to translate accesses from one address space (virtual) to a different one (physical)
 - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
 - Side effects:
 - » Can be used to avoid overlap
 - » Can be used to give uniform view of memory to programs
- **Protection:**
 - Prevent access to private memory of other processes
 - » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
 - » Kernel data protected from User programs
 - » Programs protected from themselves

2/23/10

CS162 ©UCB Spring 2010

Lec 11.18

Binding of Instructions and Data to Memory

- **Binding of instructions and data to addresses:**
 - Choose addresses for instructions and data from the standpoint of the processor

```

data1: dw    32                0x300 00000020
...
start: lw   r1,0(data1)       0x900 8C2000C0
      jal  checkit           0x904 0C000340
loop:  addi r1, r1, -1        0x908 2021FFFF
      bnz r1, r0, loop       0x90C 1420FFFF
...
checkit: ...                 0xD00 ...
    
```

A pink arrow points from the assembly code to the corresponding memory addresses.

- Could we place data1, start, and/or checkit at different addresses?
 - » Yes

- » When? Compile time/Load time/Execution time
- Related: which physical memory locations hold particular instructions or data?

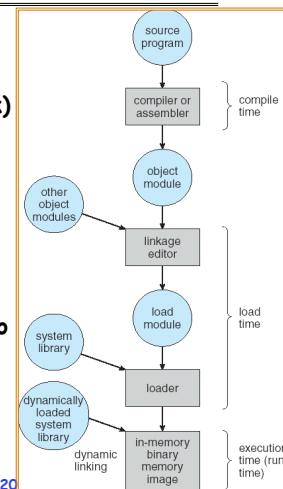
2/23/10

CS162 ©UCB Spring 2010

Lec 11.19

Multi-step Processing of a Program for Execution

- **Preparation of a program for execution involves components at:**
 - Compile time (i.e. "gcc")
 - Link/Load time (unix "ld" does link)
 - Execution time (e.g. dynamic libs)
- **Addresses can be bound to final values anywhere in this path**
 - Depends on hardware support
 - Also depends on operating system
- **Dynamic Libraries**
 - Linking postponed until execution
 - Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
 - Stub replaces itself with the address of the routine, and executes routine

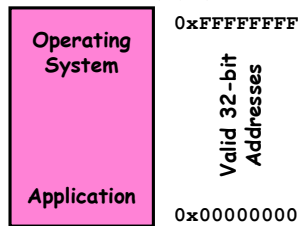


2/23/10

CS162 ©UCB Spring 2010

Recall: Uniprogramming

- Uniprogramming (no Translation or Protection)
 - Application always runs at same place in physical memory since only one application at a time
 - Application can access any physical address



- Application given illusion of dedicated machine by giving it reality of a dedicated machine
- Of course, this doesn't help us with multithreading

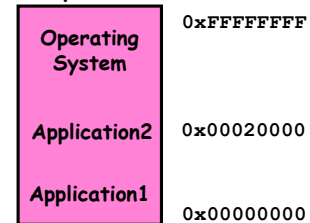
2/23/10

CS162 ©UCB Spring 2010

Lec 11.21

Multiprogramming (First Version)

- Multiprogramming without Translation or Protection
 - Must somehow prevent address overlap between threads



- Trick: Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
 - » Everything adjusted to memory location of program
 - » Translation done by a linker-loader
 - » Was pretty common in early days

- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

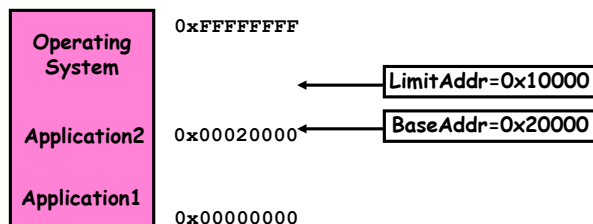
2/23/10

CS162 ©UCB Spring 2010

Lec 11.22

Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?



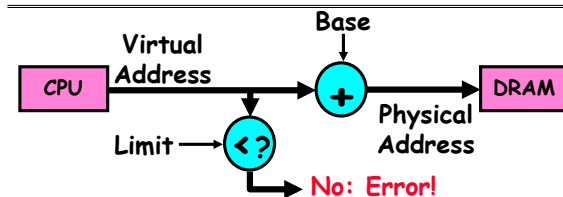
- Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
 - » If user tries to access an illegal address, cause an error
- During switch, kernel loads new base/limit from TCB
 - » User not allowed to change base/limit registers

2/23/10

CS162 ©UCB Spring 2010

Lec 11.23

Segmentation with Base and Limit registers



- Could use base/limit for **dynamic address translation** (often called "segmentation"):

- Alter address of every load/store by adding "base"
- User allowed to read/write within segment
 - » Accesses are relative to segment so don't have to be relocated when program moved to different segment
- User may have multiple segments available (e.g x86)
 - » Loads and stores include segment ID in opcode:

x86 Example: `mov [es:bx], ax`

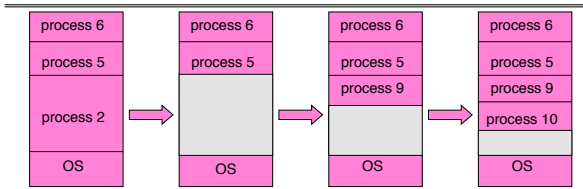
- » Operating system moves around segment base pointers as necessary

2/23/10

CS162 ©UCB Spring 2010

Lec 11.24

Issues with simple segmentation method

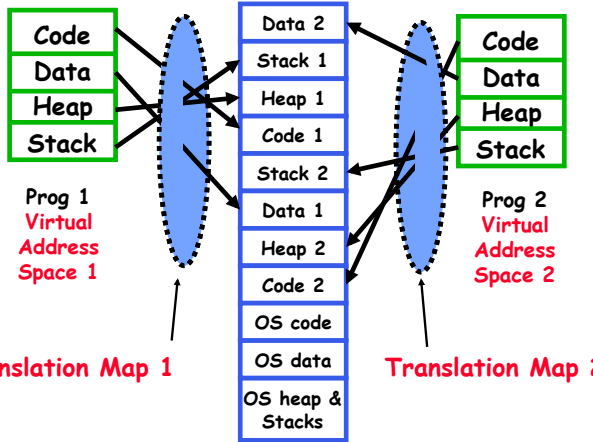


- **Fragmentation problem**
 - Not every process is the same size
 - Over time, memory space becomes fragmented
- **Hard to do inter-process sharing**
 - Want to share code segments when possible
 - Want to share memory between processes
 - Helped by providing multiple segments per process
- **Need enough physical memory for every process**

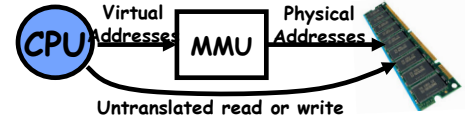
Multiprogramming (Translation and Protection version 2)

- **Problem:** Run multiple applications in such a way that they are protected from one another
- **Goals:**
 - Isolate processes and kernel from one another
 - Allow flexible translation that:
 - » Doesn't lead to fragmentation
 - » Allows easy sharing between processes
 - » Allows only part of process to be resident in physical memory
- **(Some of the required) Hardware Mechanisms:**
 - **General Address Translation**
 - » Flexible: Can fit physical chunks of memory into arbitrary places in users address space
 - » Not limited to small number of segments
 - » Think of this as providing a large number (thousands) of fixed-sized segments (called "pages")
 - **Dual Mode Operation**
 - » Protection base involving kernel/user distinction

Example of General Address Translation



Two Views of Memory



- **Recall: Address Space:**
 - All the addresses and state a process can touch
 - Each process and kernel has different address space
- **Consequently: two views of memory:**
 - View from the CPU (what program sees, virtual memory)
 - View from memory (physical memory)
 - Translation box converts between the two views
- **Translation helps to implement protection**
 - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- **With translation, every program can be linked/loaded into same region of user address space**
 - Overlap avoided through translation, not relocation

Example of Translation Table Format

Two-level Page Tables
32-bit address:

10	10	12
P1 index	P2 index	page offset

- **Page:** a unit of memory translatable by memory management unit (MMU)
 - Typically 1K - 8K
- **Page table structure in memory**
 - Each user has different page table
- **Address Space switch:** change pointer to base of table (hardware register)
 - Hardware traverses page table (for many architectures)
 - MIPS uses software to traverse table

2/23/10 CS162 ©UCB Spring 2010 Lec 11.29

Dual-Mode Operation

- Can Application Modify its own translation tables?
 - If it could, could get access to all of physical memory
 - Has to be restricted somehow
- To Assist with Protection, **Hardware** provides at least two modes (Dual-Mode Operation):
 - "Kernel" mode (or "supervisor" or "protected")
 - "User" mode (Normal program mode)
 - Mode set with bits in special control register only accessible in kernel-mode
- Intel processor actually has four "rings" of protection:
 - PL (Privilege Level) from 0 - 3
 - » PLO has full access, PL3 has least
 - Privilege Level set in code segment descriptor (CS)
 - Mirrored "IOPL" bits in condition register gives permission to programs to use the I/O instructions
 - Typical OS kernels on Intel processors only use PLO ("user") and PL3 ("kernel")

2/23/10 CS162 ©UCB Spring 2010 Lec 11.30

For Protection, Lock User-Programs in Asylum

- **Idea:** Lock user programs in padded cell with no exit or sharp objects
 - Cannot change mode to kernel mode
 - User cannot modify page table mapping
 - Limited access to memory: cannot adversely effect other processes
 - » Side-effect: Limited access to memory-mapped I/O operations (I/O that occurs by reading/writing memory locations)
 - Limited access to interrupt controller
 - What else needs to be protected?
- **A couple of issues**
 - How to share CPU between kernel and user programs?
 - » Kinda like both the inmates and the warden in asylum are the same person. How do you manage this???
 - How do programs interact?
 - How does one switch between kernel and user modes?
 - » OS → user (kernel → user mode): getting into cell
 - » User → OS (user → kernel mode): getting out of cell

2/23/10 CS162 ©UCB Spring 2010 Lec 11.31

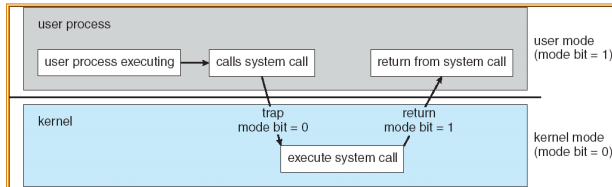
How to get from Kernel→User

- What does the kernel do to create a new user process?
 - Allocate and initialize address-space control block
 - Read program off disk and store in memory
 - Allocate and initialize translation table
 - » Point at code in memory so program can execute
 - » Possibly point at statically initialized data
 - Run Program:
 - » Set machine registers
 - » Set hardware pointer to translation table
 - » Set processor status word for user mode
 - » Jump to start of program
- How does kernel switch between processes?
 - Same saving/restoring of registers as before
 - Save/restore PSL (hardware pointer to translation table)

2/23/10 CS162 ©UCB Spring 2010 Lec 11.32

User→Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
 - Would defeat purpose of protection!
 - So, how does the user program get back into kernel?



- **System call:** Voluntary procedure call into kernel
 - Hardware for controlled User→Kernel transition
 - Can any kernel routine be called?
 - » No! Only specific ones.
 - System call ID encoded into system call instruction
 - » Index forces well-defined interface with kernel

2/23/10

CS162 ©UCB Spring 2010

Lec 11.33

System Call Continued

- What are some system calls?
 - I/O: open, close, read, write, lseek
 - Files: delete, mkdir, rmdir, truncate, chown, chgrp, ..
 - Process: fork, exit, wait (like join)
 - Network: socket create, set options
- Are system calls constant across operating systems?
 - Not entirely, but there are lots of commonalities
 - Also some standardization attempts (POSIX)
- What happens at beginning of system call?
 - » On entry to kernel, sets system to kernel mode
 - » Handler address fetched from table/Handler started
- System Call argument passing:
 - In registers (not very much can be passed)
 - Write into user memory, kernel copies into kernel mem
 - » User addresses must be translated!
 - » Kernel has different view of memory than user
 - Every Argument must be explicitly checked!

2/23/10

CS162 ©UCB Spring 2010

Lec 11.34

User→Kernel (Exceptions: Traps and Interrupts)

- A system call instruction causes a synchronous exception (or "trap")
 - In fact, often called a software "trap" instruction
- Other sources of **Synchronous Exceptions:**
 - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
 - Segmentation Fault (address out of range)
 - Page Fault (for illusion of infinite-sized memory)
- Interrupts are **Asynchronous Exceptions**
 - Examples: timer, disk ready, network, etc....
 - **Interrupts can be disabled, traps cannot!**
- On system call, exception, or interrupt:
 - Hardware enters kernel mode with interrupts disabled
 - Saves PC, then jumps to appropriate handler in kernel
 - For some processors (x86), processor also saves registers, changes stack, etc.
- Actual handler typically saves registers, other CPU state, and switches to kernel stack

2/23/10

CS162 ©UCB Spring 2010

Lec 11.35

Additions to MIPS ISA to support Exceptions?


- Exception state is kept in "Coprocessor 0"
 - Use mfc0 read contents of these registers:
 - » **BadVAddr (register 8):** contains memory address at which memory reference error occurred
 - » **Status (register 12):** interrupt mask and enable bits
 - » **Cause (register 13):** the cause of the exception
 - » **EPC (register 14):** address of the affected instruction
- | | | | | | | | | |
|--------|------|---|-----|---|------|---|-----|---|
| | 15 | 8 | 5 | 4 | 3 | 2 | 1 | 0 |
| Status | Mask | | k | e | k | e | k | e |
| | | | old | | prev | | cur | |
- Status Register fields:
 - Mask: Interrupt enable
 - » 1 bit for each of 5 hardware and 3 software interrupts
 - k = kernel/user: 0⇒kernel mode
 - e = interrupt enable: 0⇒interrupts disabled
 - **Exception⇒6 LSB shifted left 2 bits, setting 2 LSB to 0:**
 - » run in kernel mode with interrupts disabled

2/23/10

CS162 ©UCB Spring 2010

Lec 11.36

Intel x86 Special Registers



intel
pentium
EXTREME EDITION

80386 Special Registers

Segment registers

Code Seg. (CS)	Data Seg. (DS)
Stack Seg. (SS)	Extra Seg. (ES)
Extra Seg. (FS)	Extra Seg. (GS)

Index: 15 3 2 1 0

Index	T I RPL
-------	---------


RPL = Requestor Privilege Level
 TL = Table Indicator
 (0 = GDT, 1 = LDT)
 Index = Index into table

Protected Mode segment selector:

Typical Segment Register
Current Priority is RPL
Of Code Segment (CS)

2/23/10
CS162 ©UCB Spring 2010
Lec 11.37

Communication



- Now that we have isolated processes, how can they communicate?
 - Shared memory: common mapping to physical page
 - » As long as place objects in shared memory address range, threads from each process can communicate
 - » Note that processes A and B can talk to shared memory through different addresses
 - » In some sense, this violates the whole notion of protection that we have been developing
 - If address spaces don't share memory, all inter-address space communication must go through kernel (via system calls)
 - » Byte stream producer/consumer (put/get): Example, communicate through pipes connecting stdin/stdout
 - » Message passing (send/receive): Will explain later how you can use this to build remote procedure call (RPC) abstraction so that you can have one program make procedure calls to another
 - » File System (read/write): File system is shared state!

2/23/10
CS162 ©UCB Spring 2010
Lec 11.38

Closing thought: Protection without Hardware

- Does protection require hardware support for translation and dual-mode behavior?
 - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
- Protection via Strong Typing
 - Restrict programming language so that you can't express program that would trash another program
 - Loader needs to make sure that program produced by valid compiler or all bets are off
 - Example languages: LISP, Ada, Modula-3 and Java
- Protection via software fault isolation:
 - Language independent approach: have compiler generate object code that provably can't step out of bounds
 - » Compiler puts in checks for every "dangerous" operation (loads, stores, etc). Again, need special loader.
 - » Alternative, compiler generates "proof" that code cannot do certain things (Proof Carrying Code)
 - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)

2/23/10
CS162 ©UCB Spring 2010
Lec 11.39

Summary

- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
 - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
 - Pros: Optimal (average response time)
 - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
 - Multiple queues of different priorities
 - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Lottery Scheduling:
 - Give each thread a priority-dependent number of tokens (short tasks ⇒ more tokens)
 - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness
- Evaluation of mechanisms:
 - Analytical, Queuing Theory, Simulation

2/23/10
CS162 ©UCB Spring 2010
Lec 11.40

Summary (2)

- **Memory is a resource that must be shared**
 - **Controlled Overlap:** only shared when appropriate
 - **Translation:** Change Virtual Addresses into Physical Addresses
 - **Protection:** Prevent unauthorized Sharing of resources
- **Simple Protection through Segmentation**
 - Base+limit registers restrict memory accessible to user
 - Can be used to translate as well
- **Full translation of addresses through Memory Management Unit (MMU)**
 - Every Access translated through page table
 - Changing of page tables only available to user
- **Dual-Mode**
 - Kernel/User distinction: User restricted
 - User→Kernel: System calls, Traps, or Interrupts
 - Inter-process communication: shared memory, or through kernel (system calls)

2/23/10

CS162 ©UCB Spring 2010

Lec 11.41