CS 423
Operating System Design: Midterm Review

Professor Adam Bates
Spring 2017
Midterm Format

• **Part 1**: In-class exam
  - 50 minute on Monday March 6th
  - Multiple choice questions only (similar to HW1)
  - Open book, open note, but no electronic devices

• **Part 2**: Take home exam.
  - Pick-up in-class Monday, due Wed March 8th.
  - Work independently. No communication with classmates in-person or on piazza.
  - E-mail me if you have a question.
  - Intensity will be modest; shouldn’t take forever.
Everything we’ve covered in class is fair game.
  - Slides
  - Lectures
  - Piazza posts

Not testing out of the textbook, but a helpful source of knowledge to clarify points from class.
System Calls

Function Calls

Caller and callee are in the same Process
- Same user
- Same “domain of trust”

System Calls

- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse
Example System Calls?

Example:
- getuid()  //get the user ID
- fork()    //create a child process
- exec()    //executing a program

Don’t confuse system calls with stdlib calls

Differences?
- Is printf() a system call?
- Is rand() a system call?
Each system call has analogous procedure calls from the standard I/O library:

<table>
<thead>
<tr>
<th>System Call</th>
<th>Standard I/O call</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>fopen</td>
</tr>
<tr>
<td>close</td>
<td>fclose</td>
</tr>
<tr>
<td>read/write</td>
<td>getchar/putchar</td>
</tr>
<tr>
<td></td>
<td>getc/putc</td>
</tr>
<tr>
<td></td>
<td>fgetc/fputc</td>
</tr>
<tr>
<td></td>
<td>fread/fwrite</td>
</tr>
<tr>
<td></td>
<td>gets/puts</td>
</tr>
<tr>
<td></td>
<td>fgets/fputs</td>
</tr>
<tr>
<td></td>
<td>scanf/printf</td>
</tr>
<tr>
<td></td>
<td>fscanf/fprintf</td>
</tr>
<tr>
<td>lseek</td>
<td>fseek</td>
</tr>
</tbody>
</table>
Processes

- Possible process states
  - Running (occupy CPU)
  - Blocked
  - Ready (does not occupy CPU)
  - Other states: suspended, terminated

Question: in a single processor machine, how many process can be in running state?

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
• What UNIX call creates a process?
  - `fork()` duplicates a process so that instead of one process you get two.
    - The new process and the old process both continue in parallel from the statement that follows the `fork()`
  - How can you tell the two processes apart?
    - `fork()` returns
      - 0 if child
      - -1 if fork fails
      - Child’s PID if parent process
  - If the parent code changes a global variable, will the child see the change?
    - Nope! On fork, child gets new program counter, stack, file descriptors, heap, globals, pid!
• What if we need the child process to execute different code than the parent process?
  ▪ Exec function allows child process to execute code that is different from that of parent
  ▪ Exec family of functions provides a facility for overlaying the process image of the calling process with a new image.
  ▪ Exec functions return -1 and sets errno if unsuccessful
# Threads and Processes

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>
### Threads vs. Processes

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>
What is the difference between kernel and user threads? Pros and cons?

- **Kernel thread packages**
  - Each thread can make blocking I/O calls
  - Can run concurrently on multiple processors

- **Threads in User-level**
  - Fast context switch
  - Customized scheduling
Things suitable for threading

- Block for potentially long waits
- Use many CPU cycles
- Respond to asynchronous events
- Execute functions of different importance
- Execute parallel code
• What is a signal in UNIX/Linux?
  • A way for one process to send a notification to another
  • A signal can be “caught”, “ignored”, or “blocked”

  ▪ Signal is **generated** when the event that causes it occurs.
  ▪ Signal is **delivered** when a process receives it.
  ▪ The **lifetime** of a signal is the interval between its generation and delivery.
  ▪ Signal that is generated but not delivered is **pending**.
  ▪ Process **catches** signal if it executes a **signal handler** when the signal is delivered.
  ▪ Alternatively, a process can **ignore** a signal when it is delivered, that is to take no action.
  ▪ Process can temporarily prevent signal from being delivered by **blocking** it.
  ▪ **Signal Mask** contains the set of signals currently blocked.
User-generated Signals

- How can you send a signal to a process from the command line?
  - `kill` 😓
  - `kill -l` will list the signals the system understands
  - `kill [-signal] pid` will send a signal to a process.
    - The optional argument may be a name or a number (default is SIGTERM).
  - To unconditionally kill a process, use:
    - `kill -9 pid` which is `kill -SIGKILL pid`. 
When do deadlocks occur (hint: 4 preconditions)?

- Mutual exclusion
- Hold and wait condition
- No preemption condition
- Circular wait condition
OS Goals

1. Provide an interface for applications to do common functions.
2. Manage resources
3. Structure concurrent application execution
The POSIX Standard Specifies UNIX Interface

The OS Runs on Multiple Platforms while presenting the same Interface:

- Web Server
- Browser
- Slack
- Pop Mail

The POSIX Standard (machine independent part)

- Read/Write
- Standard Output
- Device Control
- File System
- Communication

Hardware

Network
Steps in Making a Syscall

**read (fd, buffer, nbytes)**
Some system calls for file management:

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd = open(file, how, ...)</code></td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td><code>s = close(fd)</code></td>
<td>Close an open file</td>
</tr>
<tr>
<td><code>n = read(fd, buffer, nbytes)</code></td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td><code>n = write(fd, buffer, nbytes)</code></td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td><code>position = lseek(fd, offset, whence)</code></td>
<td>Move the file pointer</td>
</tr>
<tr>
<td><code>s = stat(name, &amp;buf)</code></td>
<td>Get a file’s status information</td>
</tr>
</tbody>
</table>
### Directory and file system management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = mkdir(name, mode)</code></td>
<td>Create a new directory</td>
</tr>
<tr>
<td><code>s = rmdir(name)</code></td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td><code>s = link(name1, name2)</code></td>
<td>Create a new entry, name2, pointing to name1</td>
</tr>
<tr>
<td><code>s = unlink(name)</code></td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td><code>s = mount(special, name, flag)</code></td>
<td>Mount a file system</td>
</tr>
<tr>
<td><code>s = umount(special)</code></td>
<td>Unmount a file system</td>
</tr>
</tbody>
</table>
### Some system calls for…

... miscellaneous tasks:

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = chdir(dirname)</td>
<td>Change the working directory</td>
</tr>
<tr>
<td>s = chmod(name, mode)</td>
<td>Change a file’s protection bits</td>
</tr>
<tr>
<td>s = kill(pid, signal)</td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td>seconds = time(&amp;seconds)</td>
<td>Get the elapsed time since Jan. 1, 1970</td>
</tr>
</tbody>
</table>
What is a process’s state?

- Program Counter
- Code Segment
- Data Segment
- Heap
- Stack Segment
- Stack
- Registers

- OpCode
- Operand
- Current Instruction
- Data Operand
- Offset
- Stack Pointer
- Offset
The state for processes that are not running on the CPU are maintained in the Process Control Block (PCB) data structure.

Process Control Block

An alternate PCB diagram

Updated during context switch
The Context Switch

Load State (Context)

Save State (Context)
Interrupts Recap

How does interrupt handling change the instruction cycle?

Fetch Stage: Start

Execute Stage: Fetch next instruction → Execute Instruction

Interrupt Stage: Check for INT, init INT handler → HALT

Interrupts disabled
Hardware Interrupts

- Hardware generated:
  - Different I/O devices are connected to different physical lines (pins) of an “Interrupt controller”
  - Device hardware signals the corresponding line
  - Interrupt controller signals the CPU (by signaling the Interrupt pin and passing an interrupt number)
  - CPU saves return address after next instruction and jumps to corresponding interrupt handler
Why Hardware INTs?

- Hardware devices may need asynchronous and immediate service. For example:
  - Timer interrupt: Timers and time-dependent activities need to be updated with the passage of time at precise intervals
  - Network interrupt: The network card interrupts the CPU when data arrives from the network
  - I/O device interrupt: I/O devices (such as mouse and keyboard) issue hardware interrupts when they have input (e.g., a new character or mouse click)
Processing HW INT’s

**Hardware**

1. Device controller or other hardware issues an interrupt.
2. Processor finishes execution of current instruction.
3. Processor signals acknowledgment of interrupt.
4. Processor pushes PSW and PC onto control stack.
5. Processor loads new PC value based on interrupt.

**Software**

1. Save remainder of state information???
3. Restore process state information.
4. Restore old PSW and PC.
Other Interrupts

- **Software Interrupts:**
  - Interrupts caused by the execution of a software instruction:
    - INT <interrupt_number>
  - Used by the system call `interrupt()`

- Initiated by the running (user level) process

- Cause current processing to be interrupted and transfers control to the corresponding interrupt handler in the kernel
Other Interrupts

- Exceptions
  - Initiated by processor hardware itself
  - Example: divide by zero
- Like a software interrupt, they cause a transfer of control to the kernel to handle the exception
Interrupts (as the name suggests) have the highest priority (compared to user and kernel threads) and therefore run first

- What are the implications on regular program execution?
  - Must keep interrupt code short in order not to keep other processing stopped for a long time
  - Cannot block (regular processing does not resume until interrupt returns, so if the interrupt blocks in the middle the system “hangs”)
Designing an Interrupt handler (top half):

- request_irq (irq, handler, flags, name, dev)
- free_irq (irq, dev)
- Notes:
  - handler is a pointer to the interrupt handler function
  - Interrupt handlers need not be re-entrant (same irq is masked until handler exits)
  - IRQ lines can be shared by multiple devices. The parameter dev is a unique “cookie” to be supplied by the given device and checked by the handler
  - The kernel sequentially invokes all handlers registered for a given irq
Designing an Interrupt Handler (Bottom Half):

- Since the interrupt handler must be minimal, all other processing related to the event that caused the interrupt must be deferred
  
  - Example:
    - Network interrupt causes packet to be copied from network card
    - Other processing on the packet should be deferred until its time comes

- The deferred portion of interrupt processing is called the “Bottom Half”
How are interrupts handled on multicore machines?

- On x86 systems each CPU gets its own local Advanced Programmable Interrupt Controller (APIC). They are wired in a way that allows routing device interrupts to any selected local APIC.
- The OS can program the APICs to determine which interrupts get routed to which CPUs.
- The default (unless OS states otherwise) is to route all interrupts to processor 0.
soft_irq’s

- 32 handlers that must be statically defined in the Linux kernel.
- A hardware interrupt (before returning) uses `raise_softirq()` to mark that a given soft_irq must execute the bottom half.
- At a later time, when scheduling permits, the marked soft_irq handler is executed:
  - When a hardware interrupt is finished
  - When a process makes a system call
  - When a new process is scheduled
soft_irq types

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTIRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- ...

Tasklet

- Bottom halves multiplexed on top of soft_irq’s
- Scheduled using
  - tasklet_schedule()
  - tasklet_hi_schedule()
- Same tasklet invocations are serialized
- Tasklets can be created or removed dynamically
- Cannot sleep (cannot save their context)
Work Queues

- Work deferred to its own thread
- Can be scheduled together with other threads according to priorities set by a scheduling policy
- Associated with its thread control block and hence can block (and save context)
  - DECLARE_WORK(name, void (*func)(void *), void *data);
  - INIT_WORK(struct work_struct *work, void (*func)(void *), void *data);
  - schedule_work(&work);
Denial of Service

Client requests get queued up in the listen queue First-come first-served

Server

Connected socket

accept()

OS

80

Listen queue

Hardware Interrupts copy packets from network card

soft_irq

put packets in the right application queue

Has a lower priority than the OS kernel (hence, does not get to run)
What Are Scheduling Goals?

• What are the goals of a scheduler?
• Linux Scheduler’s Goals:
  ▪ Generate illusion of concurrency
  ▪ Maximize resource utilization (e.g., mix CPU and I/O bound processes appropriately)
  ▪ Meet needs of both I/O-bound and CPU-bound processes
    ▪ Give I/O-bound processes better interactive response
    ▪ Do not starve CPU-bound processes
  ▪ Support Real-Time (RT) applications
# Scheduling API

<table>
<thead>
<tr>
<th>System call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nice()</td>
<td>change the priority</td>
</tr>
<tr>
<td>getpriority()</td>
<td>get the maximum group priority</td>
</tr>
<tr>
<td>setpriority()</td>
<td>set the group priority</td>
</tr>
<tr>
<td>sched_getscheduler()</td>
<td>get the scheduling policy</td>
</tr>
<tr>
<td>sched_setscheduler()</td>
<td>set the scheduling policy and priority</td>
</tr>
<tr>
<td>sched_getparam()</td>
<td>get the priority</td>
</tr>
<tr>
<td>sched_setparam()</td>
<td>set the priority</td>
</tr>
<tr>
<td>sched_yield()</td>
<td>relinquish the processor voluntarily</td>
</tr>
<tr>
<td>sched_get_priority_min()</td>
<td>get the minimum priority value</td>
</tr>
<tr>
<td>sched_get_priority_max()</td>
<td>get the maximum priority value</td>
</tr>
<tr>
<td>sched_rr_get_interval()</td>
<td>get the time quantum for Round-Robin</td>
</tr>
</tbody>
</table>
Used for non real-time processes

- Complex heuristic to balance the needs of I/O and CPU centric applications
- Processes start at 120 by default
  - Static priority
    - A “nice” value: 19 to -20.
    - Inherited from the parent process
    - Altered by user (negative values require special permission)
  - Dynamic priority
    - Based on static priority and applications characteristics (interactive or CPU-bound)
    - Favor interactive applications over CPU-bound ones
- Timeslice is mapped from priority
How does a static priority translate to real CPU access?

if (static priority < 120)
   Quantum = 20 \( (140 - \text{static priority}) \)
else
   Quantum = 5 \( (140 - \text{static priority}) \)
(in ms)

Higher priority $\rightarrow$ Larger quantum
Completely Fair Scheduler

- Merged into the 2.6.23 release of the Linux kernel and is the default scheduler.
- Scheduler maintains a red-black tree where nodes are ordered according to received virtual execution time.
- Node with smallest virtual received execution time is picked next.
- Priorities determine accumulation rate of virtual execution time:
  - Higher priority \(\rightarrow\) slower accumulation rate.
CFS dispenses with a run queue and instead maintains a time-ordered red-black tree. Why?

An RB tree is a BST w/ the constraints:
1. Each node is red or black
2. Root node is black
3. All leaves (NIL) are black
4. If node is red, both children are black
5. Every path from a given node to its descendent NIL leaves contains the same number of black nodes

Takeaway: In an RB Tree, the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf.
Other scheduling policies

- What if you want to maximize throughput?
  - Shortest job first!

- What if you want to meet all deadlines?
  - Earliest deadline first!
  - Problem?
  - Works only if you are not “overloaded”. If the total amount of work is more than capacity, a domino effect occurs as you always choose the task with the nearest deadline (that you have the least chance of finishing by the deadline), so you may miss a lot of deadlines!
**Problem:**
- It is Monday. You have a homework due tomorrow (Tuesday), a homework due Wednesday, and a homework due Thursday
- It takes on average 1.5 days to finish a homework.

**Question: What is your best (scheduling) policy?**
- You could instead skip tomorrow’s homework and work on the next two, finishing them by their deadlines
- Note that EDF is bad: It always forces you to work on the next deadline, but you have only one day between deadlines which is not enough to finish a 1.5 day homework – you might not complete any of the three homeworks!
Task Scheduling

- How to assign task priorities?
  - Rate Monotonic (large rate = higher priority)

<table>
<thead>
<tr>
<th>Breaks task (2 ms every 4 ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Steering wheel task (4.5 ms every 10 ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Velocity control task (0.45 ms every 15 ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Intuition: Urgent tasks should be higher in priority

*Is there a problem here??*
• Deadlines are missed!
• Average Utilization < 100%

Breaks task (2 ms every 4 ms)

Steering wheel task (4.5 ms every 10 ms)

Velocity control task (0.45 ms every 15 ms)
• Deadlines are missed!
• Average Utilization < 100%

Fix:
Give this task invocation a lower priority (EDF)
Re: Real Time Scheduling of Periodic Tasks...

- Result #1: Earliest Deadline First (EDF) is the optimal dynamic priority scheduling policy for independent periodic tasks (meets the most deadlines of all dynamic priority scheduling policies)

- Result #2: Rate Monotonic Scheduling (RM) is the optimal static priority scheduling policy for independent periodic tasks (meets the most deadlines of all static priority scheduling policies)
How should we account for priority inversion?

Priority Inversion

High-priority task

Preempt.

Unlock S

Priority Inversion

Low-priority task

Attempt to lock S results in blocking

Lock S

Unlock S

Lock S
Consider the case below: a series of intermediate priority tasks is delaying a higher-priority one.

Unbounded Priority Inversion

High-priority task

Intermediate-priority tasks

Low-priority task

Attempt to lock S results in blocking

Preempt.

Unbounded Priority Inversion

Preempt.

Lock S
Solution: Let a task inherit the priority of any higher-priority task it is blocking
Maximum Blocking Time

Priority Inheritance Protocol:
Priority Ceiling Protocol

- Definition: The priority ceiling of a semaphore is the highest priority of any task that can lock it.
- A task that requests a lock $R_k$ is denied if its priority is not higher than the highest priority ceiling of all semaphores currently locked by other tasks (say it belongs to semaphore $R_h$).
  - The task is said to be blocked by the task holding lock $R_h$.
- A task inherits the priority of the top higher-priority task it is blocking.
Priority Ceiling Protocol:

Need Blue but Priority is lower than Red ceiling

Need Yellow but Priority is lower than Red ceiling

Needs Red, waits for 1 critical section to complete.

Done
Part of UEFI since 2013:

- different power saving states in a platform-independent manner
- The standard was originally developed by Intel, Microsoft, and Toshiba (in 1996), then later joined by HP, and Phoenix.
- The latest version is "Revision 6" published in April 2015.
ACPI Global States

- **G0**: working
- **G1**: Sleeping and hibernation (several degrees available)
- **G2**: Soft Off: almost the same as G3 Mechanical Off, except that the power supply still supplies power, at a minimum, to the power button to allow wakeup. A full reboot is required.
- **G3**: Mechanical Off: The computer's power has been totally removed via a mechanical switch (as on the rear of a PSU).
ACPI “Sleep” States

C-States:

- **C0**: is the operating state.

- **C1** (often known as Halt): is a state where the processor is not executing instructions, but can return to an executing state instantaneously. All ACPI-conformant processors must support this power state.

- **C2** (often known as Stop-Clock): is a state where the processor maintains all software-visible state, but may take longer to wake up. This processor state is optional.

- **C3** (often known as Sleep) is a state where the processor does not need to keep its cache, but maintains other state. This processor state is optional.
P-States:

- **P0** max power and frequency
- **P1** less than P0, voltage/frequency scaled
- **P2** less than P1, voltage/frequency scaled
- ...
- **Pn** less than P(n-1), voltage/frequency scaled
Power of Computation

- Terminology
  - $R$ : Power spent on computation
  - $V$ : Processor voltage
  - $f$ : Processor clock frequency
  - $R_0$ : Leakage power

- Power spent on computation is:
  - $R = k_v V^2 f + R_0$
    where $k_v$ is a constant
Energy of Computation

- Power spent on computation is:
  - \( R = k_v \ V^2 f + R_0 \)

- Consider a piece of computation of length \( C \) clock cycles and a processor operating at frequency \( f \)

- The execution time is \( t = C/f \)

- Energy spent is:
  - \( E = R \ t = (k_v \ V^2 f + R_0)(C/f) \)
Reducing voltage and frequency:

- Processor voltage can be decreased if clock frequency is decreased
  - Voltage and frequency can be decreased roughly proportionally.
  - In this case (where $V \sim f$):

\[
R = k_f f^3 + R_0
\]
\[
E = (k_f f^3 + R_0)(C/f) = k_f f^2 C + R_0 C/f
\]
There exists a minimum frequency below which no energy savings are achieved

\[ E = k_f f^2 C + R_0 C/f \]

\[ \frac{dE}{df} = 2k_f f C - R_0 C/f^2 = 0 \]

\[ f = \sqrt[3]{\frac{R_0}{2k_f}} \]
When should we perform dynamic voltage scaling?

Dynamic Voltage Scaling (DVS)

- Reduce Frequency Only
  - Processor Sleeps when Idle
    - Good idea! (i.e., Go Slower)
  - Processor Always On
    - Find Critical Frequency that minimizes energy
- Reduce Frequency and Voltage
  - Processor Sleeps when Idle
    - Bad idea! (i.e. Go Fast)
The Cost of Wakeup

- Turning Processor off...
- Energy expended on wakeup, $E_{wake}$
- To sleep or not to sleep?
  - Not to sleep (for time $t$):
    \[ E_{no-sleep} = (k_v V^2 f + R_0) t \]
  - To sleep (for time $t$) then wake up:
    \[ E_{sleep} = P_{sleep} t + E_{wake} \]
  - To save energy by sleeping: $E_{sleep} < E_{no-sleep}$

\[
t > \frac{E_{wake}}{k_v V^2 f + R_0 - P_{sleep}}
\]

Minimum sleep interval
• DPM refers to turning devices off (or putting them in deep sleep modes)
• Device wakeup has a cost that imposes a minimum sleep interval (a breakeven time)
• DPM must maximize power savings due to sleep while maintaining schedulability
The Problem with work-conserving scheduling:

Task 1 (C=2, P=12)

Task 2 (C=1, P=16)

No opportunity to sleep! :-(

Minimum sleep period
The Problem with work-conserving scheduling:

Task 1 (C=2, P=12)

Task 2 (C=1, P=16)

Solution: Postpone and batch!

Minimum sleep period
How many proc to use?

- Consider using one processor at frequency $f$ versus two at frequency $f/2$
- **Case 1:** Total power for one processor
  - $k_f f^3 + R_0$
- **Case 2:** Total power for two processors
  - $2 \{ k_f (f/2)^3 + R_0 \} = k_f f^3/4 + 2 R_0$
- **The general case:** $n$ processors
  - $n \{ k_f (f/n)^3 + R_0 \} = k_f f^3/n^2 + n R_0$
How many proc to use?

- The general case: $n$ processors
  - $Power = n \{k_f (f/n)^3 + R_0\} = k_f f^3 / n^2 + n R_0$
  - $dPower/dn = -2 k_f f^3 / n^3 + R_0 = 0$

$$n = 3 \sqrt[3]{\frac{2 k_f f^3}{R_0}}$$

- What if $n$ is not an integer?
Storage Hierarchy

Performance

CPU Registers
- Size: 32-64 bits

Cache
- Size: 4-128 words

Memory
- Size: 512-16k words

Secondary Storage
History: Mem Overlays

Used when process memory requirement exceeded the physical memory space
• Approach: Multiprogramming with fixed memory partitions
• Divides memory into $n$ fixed partitions (possible unequal)
• Problem?
  • Fragmentation!

History: Fixed Partition Allocation

Program 1

Program 2

Program 3

Free Space
History: Relocation Register

CPU Instruction Address

Logical Address MA

Base Register

Logical Address MA

Physical Address MA+BA

Memory
History: Variable Partition Allocation

Memory wasted by External Fragmentation
History: Storage Placement Strategy

- **Best Fit**
  - Use the hole whose size is equal to the need, or if none is equal, the hole that is larger but closest in size.
  - Problem: Creates small holes that can't be used.

- **First Fit**
  - Use the first available hole whose size is sufficient to meet the need.
  - Problem: Creates average size holes.

- **Next Fit**
  - Minor variation of first fit: search from the last hole used.
  - Problem: Slightly worse performance than first fit.

- **Worst Fit**
  - Use the largest available hole.
  - Problem: Gets rid of large holes making it difficult to run large programs.
Virtual Memory

- Provide user with virtual memory that is as big as user needs
- Store virtual memory on disk
- Cache parts of virtual memory being used in real memory
- Load and store cached virtual memory without user program intervention
Request Page 3…

Virtual Memory Stored on Disk

Memory

Page Table

VM
Frame

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Paging

Request Page 1...

Memory

Virtual Memory Stored on Disk

Page Table

VM  Frame

1  1
1  2
1  3
1  4
Paging

Request Page 6…

Memory

Page Table

VM  Frame

Virtual Memory Stored on Disk
Paging

Request Page 2...

Memory

Virtual Memory Stored on Disk

Page Table

VM  Frame

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Request Page 8. Swap Page 1 to Disk First…
Request Page 8. … now load Page 8 into Memory.
Page Mapping Hardware

Virtual Address (P,D)

Page Table

P D

F D

Physical Address (F,D)

Virtual Memory

Contents(P,D)

Physical Memory

Contents(F,D)
Page Mapping Hardware

Page size 1000
Number of Possible Virtual Pages 1000
Number of Page Frames 8
Page Faults

- Access a virtual page that is not mapped into any physical page
  - A fault is triggered by hardware
- Page fault handler (in OS’s VM subsystem)
  - Find if there is any free physical page available
    - If no, evict some resident page to disk (swapping space)
  - Allocate a free physical page
  - Load the faulted virtual page to the prepared physical page
  - Modify the page table
Paging Issues

- Page size is $2^n$
  - usually 512 bytes, 1 KB, 2 KB, 4 KB, or 8 KB
  - E.g. 32 bit VM address may have $2^{20}$ (1 MB) pages with 4k ($2^{12}$) bytes per page

- Page table:
  - $2^{20}$ page entries take $2^{22}$ bytes (4 MB)
  - Must map into real memory
  - Page Table base register must be changed for context switch

- No external fragmentation; internal fragmentation on last page only
Translation Lookaside Buffers

Optimization:

Virtual address

<table>
<thead>
<tr>
<th>VPage#</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TLB

<table>
<thead>
<tr>
<th>VPage#</th>
<th>PPage#</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hit

<table>
<thead>
<tr>
<th>PPage#</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Miss

Real page table

Physical address
If a virtual address is presented to MMU, the hardware checks TLB by comparing all entries simultaneously (in parallel).

If match is valid, the page is taken from TLB without going through page table.

If match is not valid
- MMU detects miss and does a page table lookup.
- It then evicts one page out of TLB and replaces it with the new entry, so that next time that page is found in TLB.
Issues:

- What TLB entry to be replaced?
  - Random
  - Least Recently Used (LRU)

- What happens on a context switch?
  - Invalidate the entire TLB contents

- What happens when changing a page table entry?
  - Change the entry in memory
  - Invalidate the TLB entry
Translation Lookaside Buffers

Effective Access Time:

- TLB lookup time = $\sigma$ time unit
- Memory cycle = $m \mu s$
- TLB Hit ratio = $\eta$
- Effective access time
  - $Eat = (m + \sigma) \eta + (2m + \sigma)(1 - \eta)$
  - $Eat = 2m + \sigma - m \eta$
Page Mapping Hardware

Virtual Address (P,D)

Page Table

0
1
0
1
1
0
1

P→F

Physical Address (F,D)

Physical Memory

Virtual Memory

Contents(P,D)

Contents(F,D)
Page Mapping Hardware

Virtual Address (004006)

Page Table

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>4→5</td>
</tr>
</tbody>
</table>

Physical Address (F,D)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>005</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>006</td>
</tr>
</tbody>
</table>

Virtual Memory

Contents(4006)

Contents(5006)

Physical Memory

Page size 1000
Number of Possible Virtual Pages 1000
Number of Page Frames 8
Page Faults

- Access a virtual page that is not mapped into any physical page
  - A fault is triggered by hardware
- Page fault handler (in OS’s VM subsystem)
  - Find if there is any free physical page available
    - If no, evict some resident page to disk (swapping space)
  - Allocate a free physical page
  - Load the faulted virtual page to the prepared physical page
  - Modify the page table
Paging Issues

- Page size is $2^n$
  - usually 512 bytes, 1 KB, 2 KB, 4 KB, or 8 KB
  - E.g. 32 bit VM address may have $2^{20}$ (1 MB) pages with 4k ($2^{12}$) bytes per page

- Page table:
  - $2^{20}$ page entries take $2^{22}$ bytes (4 MB)
  - Must map into real memory
  - Page Table base register must be changed for context switch

- No external fragmentation; internal fragmentation on last page only
Translation Lookaside Buffers

**Optimization:**

Virtual address

<table>
<thead>
<tr>
<th>VPage#</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPage#</td>
<td>PPage#</td>
</tr>
<tr>
<td>VPage#</td>
<td>PPage#</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>VPage#</td>
<td>PPage#</td>
</tr>
</tbody>
</table>

**TLB**

Hit

<table>
<thead>
<tr>
<th>PPage#</th>
<th>offset</th>
</tr>
</thead>
</table>

Physical address

Miss

Real page table
Translation Lookaside Buffers

- If a virtual address is presented to MMU, the hardware checks TLB by comparing all entries simultaneously (in parallel).
- If match is valid, the page is taken from TLB without going through page table.
- If match is not valid
  - MMU detects miss and does a page table lookup.
  - It then evicts one page out of TLB and replaces it with the new entry, so that next time that page is found in TLB.
Translation Lookaside Buffers

Issues:

- What TLB entry to be replaced?
  - Random
  - Least Recently Used (LRU)
- What happens on a context switch?
  - Invalidate the entire TLB contents
- What happens when changing a page table entry?
  - Change the entry in memory
  - Invalidate the TLB entry
Effective Access Time:

- TLB lookup time = $\sigma$ time unit
- Memory cycle = $m$ $\mu$s
- TLB Hit ratio = $\eta$
- Effective access time
  - $Eat = (m + \sigma) \eta + (2m + \sigma)(1 - \eta)$
  - $Eat = 2m + \sigma - m \eta$

Note: Doesn’t consider page faults. How would we extend?
What does this buy us?

<table>
<thead>
<tr>
<th>Virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
</tr>
</tbody>
</table>

Directory

<table>
<thead>
<tr>
<th>pte</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
</tbody>
</table>

pte

| .   |
| .   |

| .   |
| .   |

| .   |
| .   |
What does this buy us?

Answer: Sparse address spaces, and easier paging

Virtual address

Directory

pte
A logical address (on 32-bit x86 with 4k page size) is divided into:
- A page number consisting of 20 bits
- A page offset consisting of 12 bits

Divide the page number into:
- A 10-bit page directory
- A 10-bit page number

Example: Addressing in a Multi-level Page Table system.
Since each level is stored as a separate table in memory, converting a logical address to a physical one with a n-level page table may take $n+1$ memory accesses. Why?