Page Allocation and Replacement (con't)
I/O Systems

March 16, 2010
Ion Stoica
http://inst.eecs.berkeley.edu/~cs162
Review: Page Replacement Policies

- **FIFO (First In, First Out)**
  - Throw out oldest page. Be fair - let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages

- **MIN (Minimum):**
  - Replace page that won't be used for the longest time
  - Great, but can't really know future...
  - Makes good comparison case, however

- **RANDOM:**
  - Pick random page for every replacement
  - Typical solution for TLB’s. Simple hardware
  - Pretty unpredictable - makes it hard to make real-time guarantees

- **LRU (Least Recently Used):**
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
Goals for Today

- Finish Page Allocation Policies
- Working Set/Thrashing
- I/O Systems
  - Hardware Access
  - Device Drivers

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from the lecture notes by Kubiatowicz.
Second Chance Algorithm

- **Approximate LRU**
  - Replace *an* old page, not *the* oldest page

- **FIFO with “use” bit**

- **Details**
  - A “use” bit per physical page
  - On page fault check page at head of queue
    - If use bit=1 → clear bit, and move page at tail (give the page second chance!)
    - If use bit=0 → replace page
  - Moving pages to tail still complex
Second-Chance List Algorithm (VAX/VMS)

- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list
Second-Chance List Algorithm (con't)

- How many pages for second chance list?
  - If 0 ⇒ FIFO
  - If all ⇒ LRU, but page fault on every page reference

- Pick intermediate value. Result is:
  - Pro: Few disk accesses (page only goes to disk if unused for a long time)
  - Con: Increased overhead trapping to OS (software / hardware tradeoff)

- With page translation, we can adapt to any kind of access the program makes
  - Later, we will show how to use page translation / protection to share memory between threads on widely separated machines

- Question: why didn’t VAX include “use” bit?
  - Strecker (architect) asked OS people, they said they didn’t need it, so didn’t implement it
  - He later got blamed, but VAX did OK anyway
Free List

Set of all pages in Memory

Single Clock Hand:
Advances as needed to keep freelist full ("background")

Free Pages For Processes

- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique ("Pageout demon")
  - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
  - If page needed before reused, just return to active set
- Advantage: Faster for page fault
  - Can always use page (or pages) immediately on fault
Demand Paging (more details)

• Does software-loaded TLB need use bit?
  Two Options:
  - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
  - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU

• Core Map
  - Page tables map virtual page → physical page
  - Do we need a reverse mapping (i.e. physical page → virtual page)?
    » Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
    » Can’t push page out to disk without invalidating all PTEs
Allocation of Page Frames (Memory Pages)

• How do we allocate memory among different processes?
  - Does every process get the same fraction of memory?
    Different fractions?
  - Should we completely swap some processes out of memory?
• Each process needs minimum number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    » instruction is 6 bytes, might span 2 pages
    » 2 pages to handle from
    » 2 pages to handle to

• Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Local replacement – each process selects from only its own set of allocated frames
Fixed/Priority Allocation

- **Equal allocation** (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes ⇒ process gets 20 frames

- **Proportional allocation** (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:
    \[ s_i = \text{size of process } p_i \text{ and } S = \sum s_i \]
    \[ m = \text{total number of frames} \]
    \[ a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m \]

- **Priority Allocation**:
  - Possible behavior: If process \( p_i \) generates a page fault, select for replacement a frame from a process with lower priority number
  - Perhaps we should use an adaptive scheme instead???
    - What if some application just needs more memory?
Administrivia

- We have an anonymous feedback link on the course homepage
  - Please use to give feedback on course

- Would you like an extra 5% for your course grade?
  - Attend lectures and sections! 5% of grade is participation
  - Midterm was only 20%

- Should be working on Project 3 now
  - You must rely on your tests, not the autograder
Page-Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

- Question: What if we just don’t have enough memory?
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing = a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?
Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the “Working Set”
  - Working Set defines minimum number of pages needed for process to behave “well”

- Not enough memory for Working Set ⇒ Thrashing
  - Better to swap out process?
Working-Set Model

<table>
<thead>
<tr>
<th>Page reference table</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 3 4 4 4 3 4 4 4 1 3 2 3 4 4 4 4 4 4 ...</td>
</tr>
</tbody>
</table>

- $\Delta$ = working-set window = fixed number of page references
  - Example: 10,000 instructions
- $WS_i$ (working set of Process $P_i$) = total set of pages referenced in the most recent $\Delta$ (varies in time)
  - if $\Delta$ too small will not encompass entire locality
  - if $\Delta$ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum |WS_i| = total\ demand\ frames$
- if $D > m \Rightarrow$ Thrashing
  - Policy: if $D > m$, then suspend/swap out processes
  - This can improve overall system behavior by a lot!
What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in

- Clustering:
  - On a page-fault, bring in multiple pages “around” the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages

- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set
Demand Paging Summary

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past

- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not “in use”
  - If page not “in use” for one pass, than can replace

- Nth-chance clock algorithm: Another approx LRU
  - Give pages multiple passes of clock hand before replacing

- Second-Chance List algorithm: Yet another approx LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.

- Working Set:
  - Set of pages touched by a process recently

- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn't fit in memory
  - Need to swap out a process
The Requirements of I/O

- So far in this course:
  - We have learned how to manage CPU, memory
- What about I/O?
  - Without I/O, computers are useless (disembodied brains?)
  - But... thousands of devices, each slightly different
    » How can we standardize the interfaces to these devices?
  - Devices unreliable: media failures and transmission errors
    » How can we make them reliable???
  - Devices unpredictable and/or slow
    » How can we manage them if we don't know what they will do or how they will perform?
- Some operational parameters:
  - Byte/Block
    » Some devices provide single byte at a time (e.g. keyboard)
    » Others provide whole blocks (e.g. disks, networks, etc)
  - Sequential/Random
    » Some devices must be accessed sequentially (e.g. tape)
    » Others can be accessed randomly (e.g. disk, cd, etc.)
  - Polling/Interrupts
    » Some devices require continual monitoring
    » Others generate interrupts when they need service
Example Device-Transfer Rates (Sun Enterprise 6000)

- Device Rates vary over many orders of magnitude
  - System better be able to handle this wide range
  - Better not have high overhead/byte for fast devices!
  - Better not waste time waiting for slow devices
The Goal of the I/O Subsystem

• Provide Uniform Interfaces, despite wide range of different devices
  - This code works on many different devices:
    ```c
    FILE fd = fopen("/dev/something","rw");
    for (int i = 0; i < 10; i++) {
      fprintf(fd,"Count %d\n",i);
    }
    close(fd);
    - Why? Because code that controls devices ("device driver") implements standard interface.

• We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  - Can only scratch surface!
Want Standard Interfaces to Devices

- **Block Devices**: *e.g.* disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- **Character Devices**: *e.g.* keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- **Network Devices**: *e.g.* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include **socket** interface
    » Separates network protocol from network operation
    » Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes
How Does User Deal with Timing?

• Blocking Interface: “Wait”
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data

• Non-blocking Interface: “Don’t Wait”
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing

• Asynchronous Interface: “Tell Me Later”
  - When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user
Main components of Intel Chipset: Pentium 4

• Northbridge:
  - Handles memory
  - Graphics

• Southbridge: I/O
  - PCI bus
  - Disk controllers
  - USB controllers
  - Audio
  - Serial I/O
  - Interrupt controller
  - Timers
How does the processor actually talk to the device?

- CPU interacts with a Controller
  - Contains a set of registers that can be read and written
  - May contain memory for request queues or bit-mapped images

- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - **I/O instructions**: in/out instructions
    - Example from the Intel architecture: `out 0x21,AL`
  - **Memory mapped I/O**: load/store instructions
    - Registers/memory appear in physical address space
    - I/O accomplished with load and store instructions
### Example: Memory-Mapped Display Controller

- **Memory-Mapped:**
  - Hardware maps control registers and display memory into physical address space
    - Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the “frame buffer”) changes image on screen
    - Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - Say enter a set of triangles that describe some scene
      - Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - Say render the above scene
      - Addr: 0x0007F004

- Can protect with page tables

<table>
<thead>
<tr>
<th>Physical Address Space</th>
<th>Display Memory</th>
<th>Graphics Command Queue</th>
<th>Command Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80020000</td>
<td>0x80010000</td>
<td>0x8000F000</td>
<td>0x0007F004</td>
</tr>
<tr>
<td>0x80010000</td>
<td></td>
<td></td>
<td>0x0007F000</td>
</tr>
<tr>
<td>0x80020000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3/16/10</td>
<td>CS162 ©UCB Spring 2010</td>
<td>Lec 16.26</td>
<td></td>
</tr>
</tbody>
</table>
Transfering Data To/From Controller

- **Programmed I/O:**
  - Each byte transferred via processor in/out or load/store
  - **Pro:** Simple hardware, easy to program
  - **Con:** Consumes processor cycles proportional to data size

- **Direct Memory Access:**
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly

- **Sample interaction with DMA controller (from book):**

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
Summary

• Working Set:
  - Set of pages touched by a process recently

• Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn't fit in memory
  - Need to swap out a process

• I/O Devices Types:
  - Many different speeds (0.1 bytes/sec to GBytes/sec)
  - Different Access Patterns:
    » Block Devices, Character Devices, Network Devices
  - Different Access Timing:
    » Blocking, Non-blocking, Asynchronous

• I/O Controllers: Hardware that controls actual device
  - Processor Accesses through I/O instructions, load/store to special physical memory
  - Report their results through either interrupts or a status register that processor looks at occasionally (polling)