Announcements

- **Midterm #1**
  - Will be in class on Thursday
- **Project #3**
  - Handout is on the Web site
Sharing Memory

- **Pages can be shared**
  - several processes may share the same code or data
  - several pages can be associated with the same page frame
  - given read-only data, sharing is always safe

- **When writes occur, decide if processes share data**
  - operating systems often implement “copy on write” - pages are shared until a process carries out a write
    - when a shared page is written, a new page frame is allocated
    - writing process owns the modified page
    - all other sharing processes own the original page
  - page could be shared
    - processes use semaphores or other means to coordinate access
Page Sharing

Page Directory

Page Table

Page Frames

Page Table

Page Directory

P1

Shared Pages

P2
What Happens when a virtual address has no physical address?

- called a *page fault*
  - a trap into the operating system from the hardware
- caused by: the first use of a page
  - called *demand paging*
  - the operating system allocates a physical page and the process continues
  - read code from disk or init data page to zero
- caused by: a reference to an address that is not valid
  - program is terminated with a “segmentation violation”
- caused by: a page that is currently on disk
  - read page from disk and load it into a physical page, and continue the program
- caused by: a copy on write page
OS Protection attributes (Win32)

- NOACCESS: attempts to read, write or execute will cause an access violation
- READONLY: attempts to write or execute memory in this region cause an access violation
- READWRITE: attempts to execute memory in this region cause an access violation
- EXECUTE: Attempts to read or write memory in this region cause an access violation
- EXECUTE_READ: Attempts to write to memory in this region cause an access violation
- EXECUTE_READ_WRITE: Do anything to this page
- WRITE_COPY: Attempts to write will cause the system to give a process its own copy of the page. Attempts to execute cause access violation
- EXECUTE_WRITE_COPY: Attempts to write will cause the system to give a process its own copy of a page. Can’t cause an access violation
Handling a page fault

1) Check if the reference is valid
   - if not, terminate the process

2) Find a page frame to allocate for the new process
   - for now we assume there is a free page frame.

3) Schedule a read operation to load the page from disk
   - we can run other processes while waiting for this to complete

4) Modify the page table entry to the page

5) Restart the faulting instruction
   - hardware normally will abort the instruction so we just return from the trap to the correct location.
Page Fault – Page is Paged out

1) Fault
2) Read from Disk
3) Make Entry
4) Continue
Page State (hardware view)

- **Page frame number** (location in memory or on disk)
- **Valid Bit**
  - indicates if a page is present in memory or stored on disk
- **A *modify or dirty* bit**
  - set by hardware on write to a page
  - indicates whether the contents of a page have been modified since the page was last loaded into main memory
  - if a page has not been modified, the page does not have to be written to disk before the page frame can be reused
- **Reference bit**
  - set by the hardware on read/write
  - cleared by OS
  - can be used to approximate LRU page replacement
- **Protection attributes**
  - read, write, execute
What happens when we fault and there are no more physical pages?

• **Need to remove a page from main memory**
  - if it is “dirty” we must store it to disk first.
    - dirty pages have been modified since they were last stored on disk.

• **How to we pick a page?**
  - Need to choose an appropriate algorithm
    - should it be global?
    - should it be local (one owned by the faulting process)
Page Replacement Algorithms

- **FIFO**
  - Replace the page that was brought in longest ago
  - However
    - old pages may be great pages (frequently used)
    - number of page faults may increase when one increases number of page frames (discouraging!)
      - called belady’s anomaly
        - 1,2,3,4,1,2,5,1,2,3,4,5 (consider 3 vs. 4 frames)

- **Optimal**
  - Replace the page that will be used furthest in the future
  - Good algorithm(!) but requires knowledge of the future
  - With good compiler assistance, knowledge of the future is sometimes possible
Page Replacement Algorithms

- **LRU**
  - Replace the page that was actually used longest ago
  - Implementation of LRU can be a bit expensive
    - e.g. maintain a stack of nodes representing pages and put page on top of stack when the page is accessed
    - maintain a time stamp associated with each page

- **Approximate LRU algorithms**
  - maintain reference bit(s) which are set whenever a page is used
  - at the end of a given time period, reference bits are cleared
FIFO Example (3 frames)

- Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
  - access 1 - (1) fault
  - access 2 - (1,2) fault
  - access 3- (1,2,3) fault
  - access 4 - (2,3,4) fault, replacement
  - access 1 - (3,4,1) fault, replacement
  - access 2 - (4,1,2) fault, replacement
  - access 5 - (1,2,5) fault, replacement
  - access 1- (1,2,5)
  - access 2 - (1,2,5)
  - access 3 - (2,5,3) fault, replacement
  - access 4 - (5,3,4) fault, replacement
  - access 5 - (5,3,4)

- 9 page faults
Page Replacement Algorithms

• **FIFO**
  – Replace the page that was brought in longest ago
  – However
    • old pages may be great pages (frequently used)
    • number of page faults may increase when one increases number of page frames (discouraging!)
      – called belady’s anomaly
        – 1,2,3,4,1,2,5,1,2,3,4,5 (consider 3 vs. 4 frames)

• **Optimal**
  – Replace the page that will be used furthest in the future
  – Good algorithm(!) but requires knowledge of the future
  – With good compiler assistance, knowledge of the future is sometimes possible
Page Replacement Algorithms

- **LRU**
  - Replace the page that was actually used longest ago
  - Implementation of LRU can be a bit expensive
    - e.g. maintain a stack of nodes representing pages and put page on top of stack when the page is accessed
    - maintain a time stamp associated with each page

- **Approximate LRU algorithms**
  - maintain reference bit(s) which are set whenever a page is used
  - at the end of a given time period, reference bits are cleared
FIFO Example (3 frames)

- Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
  - access 1 - (1) fault
  - access 2 - (1,2) fault
  - access 3- (1,2,3) fault
  - access 4 - (2,3,4) fault, replacement
  - access 1 - (3,4,1) fault, replacement
  - access 2 - (4,1,2) fault, replacement
  - access 5 - (1,2,5) fault, replacement
  - access 1- (1,2,5)
  - access 2 - (1,2,5)
  - access 3 - (2,5,3) fault, replacement
  - access 4 - (5,3,4) fault, replacement
  - access 5 - (5,3,4)

- 9 page faults
LRU Example (3 frames)

- Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
  - access 1 - (1) fault
  - access 2 - (1,2) fault
  - access 3 - (1,2,3) fault
  - access 4 - (2,3,4) fault, replacement
  - access 1 - (3,4,1) fault, replacement
  - access 2 - (4,1,2) fault, replacement
  - access 5 - (1,2,5) fault, replacement
  - access 1 - (2,5,1)
  - access 2 - (5,1,2)
  - access 3 - (1,2,3) fault, replacement
  - access 4 - (2,3,4) fault, replacement
  - access 5 - (3,4,5) fault, replacement

- 10 page faults
LRU Example (4 frames)

- Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
  - access 1 - (1) fault
  - access 2 - (1,2) fault
  - access 3 - (1,2,3) fault
  - access 4 - (1,2,3,4) fault, replacement
  - access 1 - (2,3,4,1)
  - access 2 - (3,4,1,2)
  - access 5 - (4,1,2,5) fault, replacement
  - access 1 - (4,2,5,1)
  - access 2 - (4,5,1,2)
  - access 3 - (5,1,2,3) fault, replacement
  - access 4 - (1,2,3,4) fault, replacement
  - access 5 - (2,3,4,5) fault, replacement

- 8 faults
FIFO Example (4 frames)

- Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
  - access 1 - (1)  fault
  - access 2 - (1,2) fault
  - access 3- (1,2,3) fault
  - access 4 - (1,2,3,4) fault, replacement
  - access 1 - (1,2,3,4)
  - access 2 - (1,2,3,4)
  - access 5 - (2,3,4,5) fault, replacement
  - access 1- (3,4,5,1) fault, replacement
  - access 2 - (4,5,1,2) fault, replacement
  - access 3 - (5,1,2,3) fault, replacement
  - access 4 - (1,2,3,4) fault, replacement
  - access 5 - (2,3,4,5) fault, replacement

- 10 Page faults
Thrashing

• Virtual memory is not “free”
  – can allocate so much virtual memory that the system spends all its time getting pages
  – the situation is called thrashing
  – need to select one or more processes to swap out

• Swapping
  – write all of the memory of a process out to disk
  – don’t run the process for a period of time
  – part of medium term scheduling

• How do we know when we are thrashing?
  – check CPU utilization?
  – check paging rate?
  – Answer: need to look at both
    • low CPU utilization plus high paging rate --> thrashing
Working Sets and Page Replacement

- **Programs usually display reference locality**
  - temporal locality
    - repeated access to the same memory location
  - spatial locality
    - consecutive memory locations access nearby memory locations
  - memory hierarchy design relies heavily on locality reference
    - sequence of nested storage media

- **Working set**
  - set of pages referenced in the last delta references
Improving Heap Locality

- **Malloc (or new) don’t ensure locality among requests**
  - Two calls to malloc could get memory on different cache lines, pages, etc.

- **Option 1:**
  - Malloc a large chunk of memory and parcel it out yourself

- **Option 2:**
  - Add a “near” hint parameter to malloc
  - Indicates that memory should be allocated near the target location
    - It’s only a performance hint, and malloc can ignore it
    - Allows locality improvement without major changes
Preventing Thrashing

- Need to ensure that we can keep the working set in memory
  - if the working sets of the processes in memory exceed total page frames, then we need to swap a process out
- How do we compute the working set?
  - can approximate it using a reference bit
Implementation Issues

• **How big should a page be?**
  - want to trade cost of fault vs. fragmentation
    • cost of fault is: trap + seek + latency + transfer
  - Does the OS page size have to equal the HW page size?
    • no, just needs to be a multiple of it

• **How does I/O relate to paging**
  - if we request I/O for a process, need to lock the page
    • if not, the I/O device can overwrite the page

• **Can the kernel be paged?**
  - most of it can be.
  - what about the code for the page fault handler?