CIS3110 Lecture 5: I/O and Virtual Memory - Summary Notes

Midterm Information

- Multiple choice format due to class size
- Content covers up to and including the virtual memory examples shown in class
- Exams based on material covered in lectures, not directly from textbook readings
- Suggested practice exercises (past exam questions) are available for review

Virtual Memory Address Translation (10.2, 10.3, 10.4)

- Virtual memory address translation is a bit manipulation process:
  - Virtual addresses split into page number and page offset
  - For example, with hexadecimal 072E and 11-bit offset:
    - Convert to binary: 0000 0111 0010 1110
    - Last 11 bits = page offset
    - Remaining bits = page number
  - Division and modulus operations can be implemented as bit shifts and masks
  - Page number is used to index the page table
  - Page frame number combines with offset to form physical address

Page Faults and Handling (10.6, 10.8, 11.1, 11.2)

- Page fault occurs when the valid bit in page table entry is not set
- The MMU raises an interrupt when a page fault occurs
- Page fault handler in the OS kernel determines next steps:
  - If address is invalid: terminate process (segmentation fault)
  - If page exists on disk: schedule IO to read page into memory
  - If empty page frame available: use it
  - Otherwise: choose a page to replace using page replacement algorithm
- Page fault handling process:
  1. Block the faulting process
  2. Find/create an empty frame (may require page replacement)
  3. If replaced page was modified (dirty bit set), save to disk

4. Read needed page from paging area or executable file
5. Update page table entry (set valid bit, frame number)
6. Mark process as runnable
7. Process will restart the instruction that caused fault

- Special cases in page handling:
  - Text pages (code) and initialized data can be loaded directly from executable file
  - New pages (stack growth, malloc) are zeroed for security reasons
  - Multiple page faults may occur for a single instruction (if operands on different pages)

Protected Virtual Memory (9.3, 9.4, 17.1, 17.2)

- Each process has its own private page table
- Processes cannot access each other's memory unless explicitly shared
- Provides memory protection between processes
- Historical context:
  - Mac OS9 and early Windows (pre-NT) had non-protected memory
  - Mac OSX and Windows NT introduced protected memory
- Shared memory implemented by mapping same physical frame to multiple processes
  - Requires semaphores to manage concurrent access
  - Processes are unaware memory is shared

Working Set and Memory Management (9.5, 10.5)

- Working set: set of pages a process needs for current execution
- Resident set: set of pages actually in memory for a process
- Ideally: resident set = working set (no waste, no excessive page faults)
- Memory access patterns typically show:
  - Locality of reference (nearby memory locations accessed together)
  - Different program phases need different memory regions
  - Working set changes over time as program executes

Page Replacement Algorithms (10.4)

- Goal: minimize page faults by making good replacement decisions
- Algorithms evaluated based on page fault rates with real programs

- Four main strategies:
  1. Optimal (Belady's): replace page not needed for longest time in future
     - Theoretical ideal, requires future knowledge
  2. FIFO (First In, First Out): replace oldest page
     - Simple but not always efficient
  3. LFU (Least Frequently Used): replace page with fewest accesses
  4. LRU (Least Recently Used): replace page unused for longest time
     - Often approximated using reference bits
- Reference/Use Bit:
  - Hardware sets bit when page is accessed
  - Software periodically clears bits
  - Pages with clear bits after clearing are candidates for replacement
- FIFO Example (with 3 frame allocation):
  - Page reference string: A,B,C,D,A,B,C,D,E,F,B,B,B,C,D,E,A
  - Results in 15 page faults

Process Control with fork/exec/wait (3.3, 3.4, 8.4.5)

- Fork: creates an exact copy (clone) of the calling process
  - Returns 0 to child process
  - Returns positive process ID to parent process
  - Returns negative value on failure
- Wait: allows parent to collect child's exit status
  - Parent can determine if child exited successfully or crashed
  - If parent doesn't call wait(), child becomes a "zombie" process
  - Zombie processes keep their entry in the process table with status 'Z'
  - They retain their exit status but have released other resources
- Exec: replaces current process with a new program
  - Two variants: list form (execl) and vector form (execv)
  - With/without path search versions available (execl vs execlp)
  - Never returns on success (process becomes the new program)
  - Returns negative value on failure
  - Professor used Calvin and Hobbes "transmogrifier" analogy to explain exec

- Combined pattern enables Unix-style process creation and control
  - Fork to create new process (creates copy)
  - Exec in child to run different program (transforms process)
  - Wait in parent to collect results (gets exit status)
  - This pattern will be important for Assignment 2