CIS3110 Lecture Summary - January 30, 2025

1. Scheduling (Conclusion) (Textbook: 5.1, 5.2, 5.3)

Time Slicing and Preemption

- **Hardware Alarm Clock**: Essential for preemption; allows the kernel to regain control from running processes
- **Time Slice Implementation**:
  - Set alarm clock before putting process on CPU
  - Process runs until: (1) alarm clock rings, (2) process requests I/O (trap), or (3) external interrupt occurs
  - Reset alarm clock for next process
- **Trade-offs**:
  - Long time slice → coarse response time but better throughput (fewer context switches)
  - Short time slice → smoother interactive response but more overhead from context switches
  - If I/O frequency is high, time slice length becomes less relevant

Processor Sharing

- When time slice is very small, creates illusion that all processes run simultaneously
- Works well in practice because most processes spend time waiting for I/O rather than using CPU
- Most modern interactive systems use processor sharing
- If preemption time is infinite (no alarm clock), reverts to FIFO scheduling

Types of Scheduling

Batch Processing (Oldest Style)

- Single program runs to completion before the next program begins
- No need for context switching (only one process at a time)
- Still used in database systems and background processing

Interactive Scheduling

- Allows real-time interaction with the system
- Characterized by frequent I/O operations
- Very sensitive to long gaps between CPU access

- Requires short time slices to maintain responsiveness

Real-Time Scheduling

- Used for systems with predictable processing needs and timing constraints
- Guarantees that every process gets to run within a specific time period
- If a process exceeds its allocated time slice, it's forcibly pulled off the CPU
- Essential for control systems (robotics, navigation, autonomous vehicles)
- **Historical Example**: Margaret Hamilton's Apollo Guidance System used real-time scheduling to prioritize critical landing functions over less essential ones

Priority Scheduling

- Multiple run queues with different priority levels
- Higher priority processes get more frequent CPU time
- Example discipline: Take 7 processes from priority queue 1, then 3 from queue 2, then 1 from queue 3
- Dynamic priority adjustment:
  - Processes that use entire time slice may be moved to lower priority
  - Quick processes may be moved to higher priority
  - System administrators can manually adjust priority levels

2. Memory Management (Textbook: 9.1, 9.2, 9.3, 9.4, 9.5)

Process Memory Model

- **Text Segment**: Executable machine code (read-only)
- **Stack Segment**: Local variables
- **Data Segment**: Everything else
  - Static data and constants
  - Symbols (for debuggers)
  - Shared memory
  - Dynamic data
- Accessing memory outside these segments causes segment fault (sigsegv)
- BSS (Block Started by Symbol): Marks where static data ends and dynamic data begins

Process Creation: Fork

- Creates a duplicate (clone) of the running process

- Parent receives child's process ID as return value; child receives 0
- Copies:
  - Stack memory (local variables)
  - Heap memory (dynamically allocated memory)
  - Static variables (globals, static locals)
- Shares (read-only content):
  - Machine text (instructions)
  - Constants and symbols

Threads

- "Lightweight fork" - shares more memory between processes
- Only duplicates the stack; everything else is shared
- Critical section implications:
  - With fork: Only shared writable memory needs protection
  - With threads: All allocated memory, globals, and static values need protection
- Threads get their own private stack for local variables
- Faster to create than forked processes
- A process always has at least one thread
- When a thread calls fork, entire memory is duplicated but only one thread exists in the new process

Fibers

- POSIX definition: A thread without preemption
- Requires manual yielding of control between threads
- Allows manually passing control between execution contexts

Virtual Memory

- Physical RAM must accommodate memory requirements of all running programs
- Each program runs in a "fake" world with its own address space
- Programs cannot see each other's memory (they're on distinct number lines)
- Kernel is the only entity that can see the real memory world
- Virtual memory allows satisfying large space requirements with limited physical RAM

Paging

- Memory is divided into fixed-size slices called pages
- Only needed pages are kept in physical memory
- Page size is defined by hardware (typical Intel size: 512 bytes)
- Virtual address components:
  - Page number: Which page in the process
  - Offset: Which byte within that page
- Mathematical conversion:
  - Virtual page number = Virtual address ÷ Page size
  - Page offset = Virtual address mod Page size
- Allows data arrangement on pages to be independent of physical location
- The gap between data and stack segments doesn't need physical storage

Memory Management Unit (MMU)

- Hardware that translates virtual addresses to physical addresses
- Translation process:
  1. Divide virtual address into page number and offset
  2. Use page number to index the page table
  3. Get physical frame number from table
  4. Calculate physical address: (Frame number × Page size) + Offset
- Page table entry information:
  - Valid bit (V): Is the page in memory?
  - Read-only bit (RO): Is it read-only?
  - Dirty bit (D): Has it been modified?
  - Used bit (U): Has it been accessed?
  - Page frame number: Where in physical memory is the page stored?
- Page fault occurs when MMU encounters invalid page (not in memory)
  - Results in interrupt to kernel
  - Kernel must resolve the situation

Address Translation Examples (Hexadecimal)

- **Example 1**: Reading virtual address 0x72E with 512-byte (0x200) pages
  - Offset = 0x2E (from 0x72E mod 0x200)

    - Page number = 3 (from 0x72E ÷ 0x200)
    - Page 3 maps to frame D in page table
    - Physical address = (0xD × 0x200) + 0x2E = 0x1A2E
- **Example 2**: Reading virtual address 0x77FF with 2048-byte (0x800) pages
    - Offset = 0x7FF (from 0x77FF mod 0x800)
    - Page number = 0xE (from 0x77FF ÷ 0x800)
    - Page E maps to frame 11 in page table
    - Physical address = (0x11 × 0x800) + 0x7FF = 0x8FFF
- **Example 3**: Reading from page 0 which is not valid
    - Page fault occurs - MMU interrupts the kernel
    - Kernel must handle this case (to be discussed in next lecture)