CIS3110 - Lecture 2 - Key Concepts Summary

Assignment 1 Concepts: Memory Management

1. **Memory Allocation IDs**
   - IDs are not sequential indexes but unique identifiers
   - Similar to jersey numbers on a team - they identify specific allocations
   - The same ID always refers to the same allocation
2. **Single Malloc Model**
   - The assignment models a embedded system with fixed memory
   - Only one physical memory block (one malloc call) to represent hardware reality
   - Memory management must be implemented within this constraint
3. **Memory Management Overhead**
   - Need to track allocations with metadata
   - Overhead reduces available memory for user allocations
   - Example: 4096-byte pool with 8-byte overhead per allocation can provide at most 4088 bytes to a user
4. **Memory Merging**
   - When two adjacent free chunks exist, they should be merged
   - Prevents fragmentation
   - Enables larger future allocations
   - Memory is one-dimensional, so adjacency cases are limited
5. **First Fit vs Best Fit vs Worst Fit**
   - **First Fit**: Use the first free block large enough
   - **Best Fit**: Use the block that leaves the least wasted space
   - **Worst Fit**: Use the block that leaves the most leftover space

Process and Thread Fundamentals (3.1, 3.4, 3.5, 4.1, 4.2, 4.3, 4.6)

1. **Thread Concept**
   - A thread is a sequential flow of execution through a program
   - Like drawing a path through source code with ink
   - Represents the active execution path of a program

2. **Single vs Multi-threaded Programs**
    - Traditional programs have one thread
    - Modern programs can have multiple threads of execution
    - Multiple threads can exist within a single process
3. **CPU Time Sharing**
    - One CPU can run multiple processes by time-slicing
    - Creates illusion of simultaneity
    - Critical sections have no control over thread scheduling
4. **Process vs Thread**
    - A process is a running program
    - A process can contain multiple threads
    - Processes are isolated from each other; threads within a process share memory

Process Memory Organization (3.1, 3.4, 3.5)

1. **Process Image Components**
    - Contains all memory and state information for a running process
    - Organized into three main segments:
2. **Text Segment**
    - Contains machine instructions (compiled code)
    - Fixed size once program is loaded
    - Doesn't change during execution
3. **Data Segment**
    - Contains:
        - Constants (strings, initialized values)
        - Static variables
        - Global variables
        - Heap (for dynamic memory allocation)
    - The heap can grow upward as needed
    - Static variables persist for the program's lifetime
4. **Stack Segment**
    - Stores local variables and function call information
    - Grows downward from high memory addresses

- Enables function calls and recursion
- Automatically manages memory for local variables
- Older languages like early Fortran didn't support recursion

5. **Memory Layout**
    - Text segment is anchored
    - Data grows upward from top of text
    - Stack grows downward from high memory
    - In theory, stack and heap could collide
    - Modern 64-bit systems make this collision extremely unlikely

Virtual Memory Concept

1. **Virtual Address Space**
    - Each process has its own virtual address space
    - Processes can't directly access other processes' memory
    - Prevents accidental access to I/O or other programs
    - Maps to physical memory through the operating system
2. **Process Table Entry**
    - Contains all information about a running process
    - Stores:
        - Program counter and registers
        - Stack pointer
        - Virtual address space mapping
        - Scheduling information (status, priority)
        - CPU time usage
3. **Process Status (PS Command)**
    - Shows all running processes
    - Includes owner, PID, CPU/memory usage, status
    - Status indicators (R=running, S=sleeping)
    - Most processes spend time sleeping until work is needed

Synchronization Mechanisms (6.1, 6.2, 6.4, 6.5, 6.6)

1. **Kernel-level Synchronization**

- SPL_TTY/SPL_X (FreeBSD) mark critical sections
- Only available to kernel code in privileged mode
- Prevents interrupts from corrupting shared data

2. **User Space vs Kernel Space**
   - User programs cannot directly call kernel functions
   - System calls trap into kernel mode
   - Kernel manages privileged operations
3. **Producer-Consumer Pattern**
   - Common synchronization challenge
   - One entity produces data, another consumes it
   - Shared data structure must be protected
4. **User-space Synchronization Primitives**
   - **Mutex**: One-at-a-time access control
     - Critical_begin/critical_end protect shared resources
     - Only one thread can be in a critical section
   - **Semaphore**: Counting-based access control
     - Allows configurable number of concurrent accesses
     - Example: Barber shop with limited chairs
5. **Busy Waiting Problem**
   - Simple mutex implementation requires busy waiting
   - Burns CPU cycles while waiting for conditions
   - Inefficient when queue is full or empty
   - Better solutions will be discussed in future lectures

Critical Sections Implementation

1. **Naive Queue Implementation**
   - Uses critical sections to protect shared queue
   - add_queue busy-waits when queue is full
   - del_queue busy-waits when queue is empty
   - Safe but inefficient under load
2. **Implementation Strategy**
   - Copy shared data inside critical section

- Evaluate copied data outside critical section
- Re-enter critical section to modify shared data
- Prevents deadlock but causes busy waiting

This lecture laid groundwork for understanding process management, memory organization, and synchronization primitives. Future lectures will build on these concepts to introduce more efficient synchronization mechanisms.