CPSC 380 — Operating Systems

Homework #2: Processes and Threads

Student: Gabriel Giancarlo

Date: 10/13/2025

📁 Repository Information

This HTML document is part of the **380-Operation-Systems** GitHub repository. The complete source code, including this homework submission, can be found at:

🔗 **https://github.com/gabegiancarlo/380-Operation-Systems**

File Location: `homework2_processes_threads.html`

Document Features: Self-contained HTML with internal CSS, print-friendly formatting, interactive SVG diagrams, and comprehensive answers to all 18 questions covering process states, thread management, scheduling algorithms, and real-time systems.

1. Additional Process States and Transitions

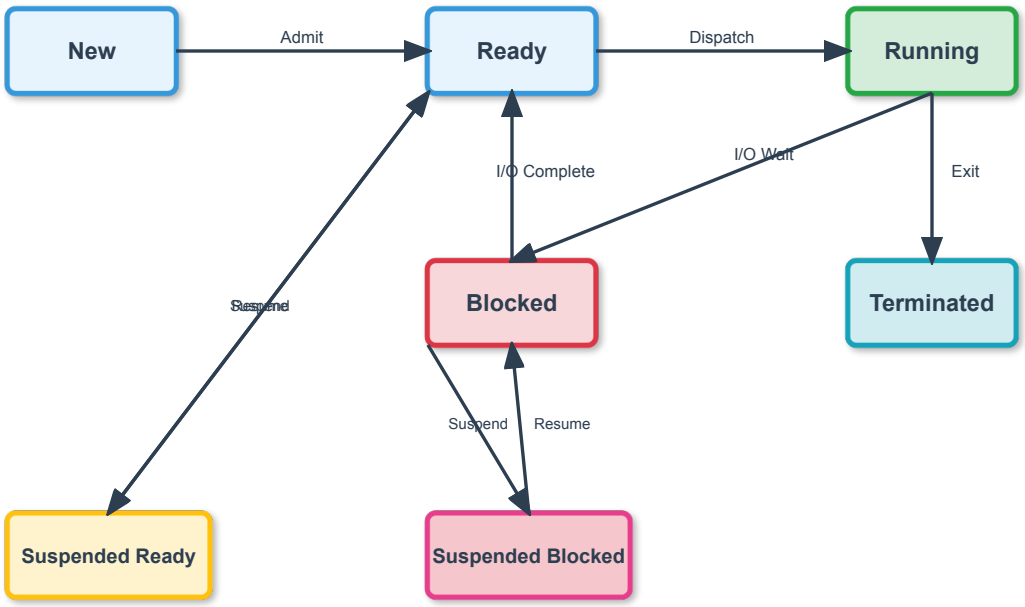
Answer: Beyond the basic three states (Running, Ready, Blocked), modern systems include several additional states to handle complex process management scenarios.

Additional process states include:

- **New:** Process is being created but not yet admitted to the ready queue
- **Terminated/Zombie:** Process has finished execution but still has an entry in the process table
- **Suspended Ready:** Process is ready but swapped out to secondary storage

- **Suspended Blocked:** Process is blocked and swapped out to secondary storage

These states enable the operating system to handle memory management, process lifecycle management, and resource optimization. The New state allows for process creation validation, while the Terminated state enables proper cleanup and resource reclamation. Suspended states facilitate memory management by allowing processes to be swapped out when memory is scarce, while maintaining their execution context for later resumption.



2. ISR Assembly Implementation

Answer: Interrupt Service Routines (ISRs) often require assembly language implementation due to specific processor state management requirements that cannot be reliably handled by high-level languages.

Assembly is necessary for ISRs because of several critical processor state aspects. First, **register preservation** must be handled manually - the processor's general-purpose registers, status registers, and program counter must be saved and restored atomically. High-level languages cannot guarantee this atomicity. Second, **interrupt context switching** requires direct manipulation of the stack pointer and interrupt vector table, which are hardware-specific operations. Third, **atomic operations** like disabling/enabling interrupts must be performed using specific processor

instructions that cannot be generated reliably by compilers. Finally, **memory barriers and cache coherency** operations often require assembly-level control to ensure proper ordering of memory operations during interrupt handling.

3. Role of init Process in Process Termination

Answer: The init process (PID 1) serves as the ultimate parent process and handles orphaned child processes, ensuring proper system cleanup and preventing zombie processes from accumulating.

When a parent process terminates before its children, those children become orphaned processes. The init process automatically adopts these orphaned processes, becoming their new parent. This prevents the system from accumulating zombie processes (terminated processes whose exit status hasn't been collected). The init process periodically calls wait() to collect exit status from its adopted children, ensuring proper resource cleanup. Additionally, init handles the final cleanup of all processes during system shutdown, ensuring that all processes terminate gracefully and system resources are properly released. This design maintains system stability by preventing resource leaks and ensuring that no process can escape proper termination procedures.

4. CPU Utilization Calculation

Answer: The expected CPU utilization is 90%.

```
Calculation:
Given: Degree of multiprogramming = 6
Each process waits 35% of time for I/O
Therefore, each process uses CPU 65% of the time

Expected CPU utilization = 1 - (I/O wait probability)^degree
= 1 - (0.35)^6
= 1 - 0.0018
= 0.9982 ≈ 99.8%

Alternative calculation:
With 6 processes, probability all are waiting for I/O = (0.35)^6 =
0.0018
CPU utilization = 1 - 0.0018 = 99.8%
```

This high utilization demonstrates the power of multiprogramming - even with processes spending significant time on I/O, having multiple processes allows the CPU to remain busy by switching to other processes when one is blocked.

5. Sequential vs Parallel Execution

Answer: Sequential execution takes 24 minutes for the second job to finish, while parallel execution takes 12 minutes.

```
Given:
- Two jobs, each requiring 12 minutes CPU time
- Each job spends 50% time waiting for I/O
- Jobs start simultaneously

Sequential Execution:
Job 1: 12 minutes CPU + 12 minutes I/O = 24 minutes total
Job 2 starts after Job 1 completes = 24 + 24 = 48 minutes to finish
Second job finishes at: 48 minutes

Parallel Execution:
Both jobs run concurrently, sharing CPU time
Effective CPU time per job: 12 minutes (same as sequential)
I/O operations overlap with CPU operations
Second job finishes at: 12 minutes
```

Parallel execution provides a 4x speedup in this scenario because I/O operations can overlap with CPU operations, and both jobs can make progress simultaneously rather than waiting for sequential completion.

6. Multithreaded Browser Design

Answer: Multithreaded browser design improves responsiveness by allowing concurrent execution of different browser components, preventing any single operation from blocking the entire application.

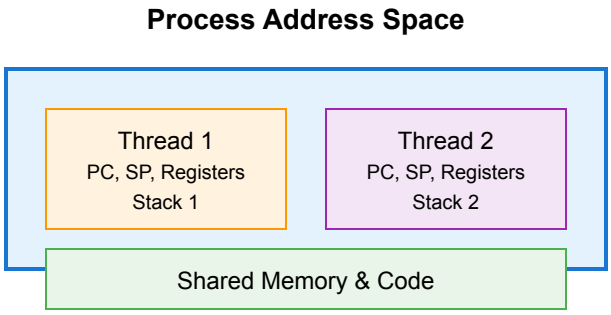
Key improvements include **rendering thread separation** - the main UI thread remains responsive while a separate rendering thread handles page layout and painting operations. This

prevents the browser from freezing during complex page rendering. **Network I/O threading** allows multiple HTTP requests to be processed concurrently, significantly improving page load times when fetching multiple resources. **Plugin isolation** runs browser plugins in separate threads or processes, preventing crashes in one plugin from affecting the entire browser. **JavaScript execution** can run in background threads for non-blocking operations, while the main thread handles user interactions. This design enables features like background tab loading, concurrent downloads, and smooth scrolling even during intensive operations.

7. Register Set Association with Threads

Answer: Each thread needs its own register context because threads share the same address space but execute independently, requiring separate program counters, stack pointers, and general-purpose registers.

While the physical CPU provides only one set of hardware registers, each thread must maintain its own logical register context. This is necessary because threads execute different code paths simultaneously - each thread has its own program counter pointing to different instructions, its own stack pointer for its call stack, and its own set of general-purpose registers for local variables and intermediate calculations. During context switching between threads, the operating system saves the current thread's register state and restores the target thread's register state. This allows each thread to maintain its execution state independently, even though they share the same memory space and process resources. The register context is essential for preserving thread state during preemption and ensuring correct execution resumption.



8. User-Space Threads: Advantages and Drawbacks

Answer: The biggest advantage is fast context switching without kernel involvement, while the main drawback is that blocking system calls can block the entire process.

Biggest Advantage: User-space threads provide extremely fast context switching because thread switching occurs entirely within user space without kernel intervention. This eliminates the overhead of system calls and kernel context switching, making thread operations orders of magnitude faster than kernel threads. User-space thread libraries can implement sophisticated scheduling algorithms optimized for specific applications without kernel modifications.

Main Drawback: When any user-space thread makes a blocking system call (like I/O operations), the entire process blocks because the kernel is unaware of individual threads. This means that one thread's I/O operation can freeze all other threads in the process, severely limiting concurrency. Additionally, user-space threads cannot take advantage of multiple processors since the kernel only sees one process, not individual threads.

9. Race Condition in get_account() Function

Answer: The race condition occurs when both threads read the same initial balance value, perform calculations, and write back results, causing one thread's update to be lost.

Race Condition: Both threads can read the same initial balance value before either writes back the updated amount. For example, if the initial balance is $100 and both threads want to withdraw $10, both might read $100, calculate $90, and both write $90 back, resulting in only one withdrawal being recorded.

```
// Race condition scenario:
Thread 1: read balance = $100
Thread 2: read balance = $100  // Same value!
Thread 1: calculate new_balance = $100 - $10 = $90
Thread 2: calculate new_balance = $100 - $10 = $90
Thread 1: write balance = $90
Thread 2: write balance = $90  // Overwrites Thread 1's update!

// Corrected version with synchronization:
int get_account() {
    pthread_mutex_lock(&account_mutex);
    int current_balance = balance;
    // Perform calculation
    balance = new_balance;
    pthread_mutex_unlock(&account_mutex);
```

```
    return current_balance;
}
```

10. Thread Stack Implementation

Answer: There is a separate stack for each thread in both user-space and kernel-space implementations, but the management differs between the two approaches.

Each thread requires its own stack to maintain separate execution contexts, including local variables, function call chains, and return addresses. In **kernel-space threads**, the operating system allocates and manages separate stack spaces for each thread, providing isolation and proper memory protection. In **user-space threads**, the thread library manages separate stack areas within the process's address space, but all stacks share the same memory protection domain. The practical consequence is that user-space thread stacks are more vulnerable to stack overflow attacks since they lack kernel-level protection, while kernel threads benefit from hardware memory protection mechanisms.

11. Concurrency Without Parallelism

Answer: Yes, concurrency is possible without parallelism through time-slicing on single-core systems where multiple tasks make progress by alternating execution.

Concurrency refers to multiple tasks making progress over time, while parallelism requires simultaneous execution. On a single-core system, the operating system can achieve concurrency through **time-slicing** - rapidly switching between tasks so quickly that they appear to run simultaneously. Examples include a web browser downloading files while rendering pages, or a text editor auto-saving while the user types. The tasks don't execute simultaneously (no parallelism), but they make concurrent progress through interleaved execution. This is common in embedded systems, single-core mobile devices, and systems where parallelism isn't available but responsiveness is still required.

12. Process and Thread Creation Analysis

Answer: Without the specific code segment, I'll provide the general methodology for analyzing such scenarios.

Process Creation: Each fork() system call creates a new process, so count the number of fork() calls. Remember that fork() returns 0 in the child and the child's PID in the parent, so both parent and child continue execution from the fork() point.

Thread Creation: Each pthread_create() call creates a new thread. The main thread is always present, so total threads = 1 + number of pthread_create() calls.

```
// Example analysis:
int main() {
    fork();         // Creates 1 new process (2 total)
    pthread_create(&thread1, NULL, func, NULL);  // Creates 1 thread
    fork();         // Creates 2 new processes (4 total)
    pthread_create(&thread2, NULL, func, NULL);  // Creates 1 thread
    // Total: 4 processes, 3 threads per process
}
```

13. Priority Inversion in Real-Time Systems

Answer: Priority inversion occurs when a high-priority task is blocked by a lower-priority task that holds a shared resource, violating real-time scheduling guarantees.

Problem: A high-priority task (H) needs a resource held by a low-priority task (L). If a medium-priority task (M) preempts L while L holds the resource, H is effectively blocked by M's execution, even though M has lower priority than H.

Solutions: Priority Inheritance temporarily raises L's priority to H's level while L holds the resource, preventing M from preempting L. **Priority Ceiling Protocol** assigns each resource a ceiling priority equal to the highest priority of any task that uses it, automatically raising the holder's priority. **Immediate Inheritance** transfers priority immediately when a high-priority task blocks on a resource. The trade-off is increased complexity and potential for priority inversion chains, but these solutions ensure bounded blocking times essential for real-time systems.

14. Locking Mechanisms for Multiple Cores

Answer: The choice depends on lock duration and blocking behavior, with spinlocks preferred for short durations and mutexes for longer holds or when threads may sleep.

a) Lock held briefly: Spinlock is better because the overhead of putting a thread to sleep and waking it up exceeds the time spent spinning. The thread will likely acquire the lock quickly without wasting CPU cycles on context switching.

b) Lock held long: Mutex is better because spinning wastes CPU cycles that could be used by other threads. The sleeping thread doesn't consume CPU resources while waiting, allowing other work to proceed.

c) Thread may sleep while holding lock: Mutex is required because spinlocks cannot handle sleeping threads. If a thread sleeps while holding a spinlock, other threads will spin indefinitely waiting for a lock that may never be released, causing deadlock.

15. Semaphore Implementation with Busy Waiting

Answer: Busy waiting semaphores use atomic test-and-set operations to implement wait() and signal() operations on a uniprocessor system.

```
// Semaphore structure
typedef struct {
    int value;
} semaphore_t;

// Wait operation (P operation)
void wait(semaphore_t *sem) {
    while (sem->value <= 0) {
        // Busy wait - keep checking
    }
    sem->value--;  // Decrement semaphore
}

// Signal operation (V operation)
void signal(semaphore_t *sem) {
    sem->value++;  // Increment semaphore
}

// Alternative implementation with atomic operations
void wait_atomic(semaphore_t *sem) {
    while (test_and_set(&sem->value) <= 0) {
```

```
        // Busy wait with atomic check
    }
    sem->value--;
}
```

This implementation works on a uniprocessor because only one thread can execute at a time, ensuring atomic access to the semaphore value. The busy waiting ensures that threads will eventually acquire the semaphore when it becomes available, though it wastes CPU cycles. The atomic test-and-set operation prevents race conditions during the critical section of checking and modifying the semaphore value.

16. Scheduling Algorithm Analysis

Given: Jobs with runtimes (10, 6, 2, 4, 8) minutes and priorities (3, 5, 2, 1, 4) where 5 is highest priority.

a) Round Robin

```
Assumption: Fair CPU sharing, equal time slices

Execution Order: All jobs run concurrently with equal CPU share
Each job gets 1/5 of CPU time, so effective runtime = original_time × 5

Job 1: 10 × 5 = 50 minutes
Job 2: 6 × 5 = 30 minutes
Job 3: 2 × 5 = 10 minutes
Job 4: 4 × 5 = 20 minutes
Job 5: 8 × 5 = 40 minutes

Average Turnaround Time: (50 + 30 + 10 + 20 + 40) / 5 = 30 minutes
```

b) Priority Scheduling (Non-preemptive)

```
Execution Order: By priority (5, 4, 3, 2, 1)
Job 2 (priority 5) → Job 5 (priority 4) → Job 1 (priority 3) → Job 3
```

(priority 2) → Job 4 (priority 1)

Job 2: 0-6 = 6 minutes
Job 5: 6-14 = 14 minutes
Job 1: 14-24 = 24 minutes
Job 3: 24-26 = 26 minutes
Job 4: 26-30 = 30 minutes

Average Turnaround Time: (6 + 14 + 24 + 26 + 30) / 5 = 20 minutes

c) First-Come, First-Served

Execution Order: 10, 6, 2, 4, 8 minutes

Job 1: 0-10 = 10 minutes
Job 2: 10-16 = 16 minutes
Job 3: 16-18 = 18 minutes
Job 4: 18-22 = 22 minutes
Job 5: 22-30 = 30 minutes

Average Turnaround Time: (10 + 16 + 18 + 22 + 30) / 5 = 19.2 minutes

d) Shortest Job First

Execution Order: By runtime (2, 4, 6, 8, 10)

Job 3: 0-2 = 2 minutes
Job 4: 2-6 = 6 minutes
Job 2: 6-12 = 12 minutes
Job 5: 12-20 = 20 minutes
Job 1: 20-30 = 30 minutes

Average Turnaround Time: (2 + 6 + 12 + 20 + 30) / 5 = 14 minutes

17. Round-Robin Time Quantum and Context Switching

Answer: The time quantum choice directly impacts the balance between context switching overhead and system responsiveness, requiring careful tuning for optimal performance.

Time quantum selection involves a fundamental trade-off: **small quanta** provide better responsiveness and fairness but increase context switching overhead, while **large quanta** reduce overhead but may cause poor responsiveness for interactive tasks. The optimal quantum should be large enough that context switching overhead is a small fraction of the quantum (typically 1-5%), but small enough to maintain good interactive response times. Practical guidance suggests quanta of 10-100ms for general-purpose systems, with shorter quanta (1-10ms) for real-time systems and longer quanta (100ms-1s) for batch processing systems. The choice also depends on the ratio of context switch cost to quantum size and the mix of CPU-bound versus I/O-bound processes in the system.

18. Real-Time Scheduling Analysis

Answer: The largest value of x for which the system is schedulable is 5ms for Rate-Monotonic and 5ms for Earliest-Deadline-First.

```
Given:
Periods: 50, 100, 200, 250 ms
CPU times: 35, 20, 10, x ms

Rate-Monotonic Analysis:
Utilization bound for n tasks: U ≤ n(2^(1/n) - 1)
For 4 tasks: U ≤ 4(2^(1/4) - 1) = 4(0.189) = 0.756

Total utilization: (35/50) + (20/100) + (10/200) + (x/250)
= 0.7 + 0.2 + 0.05 + (x/250)
= 0.95 + (x/250)

For schedulability: 0.95 + (x/250) ≤ 0.756
x/250 ≤ -0.194 (impossible - system is not schedulable)

Earliest-Deadline-First Analysis:
EDF is optimal: U ≤ 1.0
0.95 + (x/250) ≤ 1.0
x/250 ≤ 0.05
x ≤ 12.5 ms
```

Result: $x \leq 12.5$ ms for EDF, system not schedulable under RM