Week 6 - Operating Systems Journal

Overview

This journal covers the key topics from Week 6 revision materials, including operating system architecture, CPU scheduling, process synchronization, memory management, and inter-process communication.

---

1. Operating System Fundamentals

1.1 Primary Functions of Operating Systems

The operating system provides four core functions: - **Process Management**: Creating, scheduling, and terminating processes - **Memory Management**: Allocating and deallocating memory to processes - **File System Management**: Organizing and managing files on storage devices - **I/O Device Management**: Controlling and coordinating I/O devices

Note: Compiling source code is NOT a function of the OS - it's performed by compilers.

1.2 Operating System Architectures

Layered Architecture

- Organized in hierarchical layers
- Each layer uses services from the layer below
- **Kernel layer** (lowest) directly interacts with hardware
- Higher layers provide abstractions for applications

Monolithic Kernel

- All OS services run in kernel space
- **Advantages**:
- Better performance due to reduced context switching
- Direct function calls between components
- **Disadvantages**:
- Less modular
- Harder to maintain
- Security vulnerabilities affect entire system
- **Examples**: Traditional Linux, Traditional Unix

Microkernel Architecture

- Minimal kernel with only essential services
- Most services run in user space
- **Advantages**:
- Better security and isolation
- More modular design
- Easier to maintain and extend
- **Disadvantages**:
- More context switching overhead
- Potentially lower performance
- **Examples**: Windows NT (original design), Minix, QNX

---

2. Boot Process

Boot Sequence

1. **BIOS/UEFI**:
2. Initializes hardware
3. Performs Power-On Self Test (POST)

4. Loads the bootloader

5. **Bootloader** (e.g., GRUB, LILO):

6. Loads the operating system kernel into memory

7. Transfers control to the kernel

8. **Kernel**:

9. Initializes system components
10. Mounts root filesystem

11. Starts init system

12. **Init System**:

13. First user-space process (PID 1)
14. Starts system services and daemons

GNU/Linux

- **GNU** = "GNU's Not Unix" (recursive acronym)
- Free software project providing utilities used with Linux
- Linux kernel + GNU utilities = GNU/Linux system

---

3. CPU Scheduling Algorithms

3.1 First-Come-First-Serve (FCFS)

- Processes execute in order of arrival
- **Non-preemptive**
- Simple to implement
- **Disadvantage**: Convoy effect (short processes wait for long ones)

Example Calculation: ``` Process | AT | BT | CT | TAT | WT P1 | 0 | 8 | 8 | 8 | 0 P2 | 1 | 4 | 12 | 11 | 7 P3 | 2 | 9 | 21 | 19 | 10 P4 | 3 | 5 | 26 | 23 | 18

Formulas: - Completion Time (CT): When process finishes - Turnaround Time (TAT) = CT - AT - Waiting Time (WT) = TAT - BT - Average WT = 8.75 units - Average TAT = 15.25 units ```

3.2 Shortest Job First (SJF)

- Executes process with shortest burst time first
- **Non-preemptive version**: Once started, runs to completion
- **Preemptive version** (SRTF): Can preempt if shorter job arrives
- **Advantage**: Minimizes average waiting time
- **Disadvantage**: Can cause **starvation** of longer processes

Example Calculation: ``` Execution Order: P1 → P2 → P4 → P3

Process | AT | BT | CT | TAT | WT P1 | 0 | 8 | 8 | 8 | 0 P2 | 1 | 4 | 12 | 11 | 7 P4 | 3 | 5 | 17 | 14 | 9 P3 | 2 | 9 | 26 | 24 | 15

Average WT = 7.75 units Average TAT = 14.25 units ```

3.3 Round Robin (RR)

- Time quantum/time slice allocated to each process
- Preemptive
- Fair allocation
- Good for time-sharing systems

3.4 Priority Scheduling

- Each process assigned a priority
- Higher priority processes execute first
- Can cause starvation
- **Aging**: Gradually increase priority of waiting processes to prevent starvation

3.5 Multi-Level Feedback Queue (MLFQ)

- Multiple queues with different priorities
- Processes move between queues based on behavior
- Automatically identifies I/O-bound vs CPU-bound processes
- I/O-bound processes get higher priority for quick CPU access

---

4. Process Synchronization

4.1 Race Conditions

Definition: When the outcome depends on the non-deterministic ordering of execution of multiple processes/threads accessing shared data.

Example: Banking application ``` Thread 1: Withdraw $500 Thread 2: Withdraw $300 Balance: $600

Without synchronization: 1. T1 reads balance = $600 2. T2 reads balance = $600 3. T1 calculates new balance = $100 4. T2 calculates new balance = $300 5. T1 writes $100 6. T2 writes $300 (overwrites T1) Result: $800 withdrawn from $600 account! ```

4.2 Synchronization Mechanisms

Semaphore

- Uses a **counter** to control access to shared resources
- Can allow multiple simultaneous accesses
- **Binary semaphore**: Acts like mutex (0 or 1)
- **Counting semaphore**: Allows N concurrent accesses

Mutex (Mutual Exclusion Lock)

- Binary lock (locked/unlocked)
- Only one thread can hold the lock
- Ensures atomic execution of critical section

Monitor

- High-level synchronization construct
- Encapsulates shared data and procedures
- Only one process can be active in monitor at a time

Spinlock

- Busy-waiting lock
- Process continuously checks if lock is available
- Efficient for short wait times

4.3 Critical Section

The section of code that must be executed atomically (read → check → calculate → write).

Solution Requirements: 1. Mutual exclusion 2. Progress 3. Bounded waiting

---

5. Memory Management

5.1 Memory Allocation Algorithms

First-Fit

- Allocates first hole large enough
- Fast
- Can lead to fragmentation

Best-Fit

- Selects **smallest hole** that can accommodate the process
- Minimizes wasted space
- Slower (must search entire list)

Example: `Request: 150 KB Holes: 100 KB, 250 KB, 180 KB, 300 KB Best-Fit selects: 180 KB Remaining: 30 KB`

Worst-Fit

- Selects **largest available hole**
- Leaves larger remaining holes
- Can reduce external fragmentation

Example: `Request: 150 KB Holes: 100 KB, 250 KB, 180 KB, 300 KB Worst-Fit selects: 300 KB Remaining: 150 KB (still usable)`

Next-Fit

- Like First-Fit but continues from last allocation point
- Better distribution of holes

5.2 Fragmentation

External Fragmentation

- Free memory scattered in small holes
- Total free memory sufficient but not contiguous
- **Solution**: Compaction, Paging

Internal Fragmentation

- Allocated memory larger than requested
- Wasted space within allocated partition
- **Occurs in**: Fixed partition schemes, Paging

5.3 Paging

- Divides memory into fixed-size pages
- **Eliminates external fragmentation**
- **Can have internal fragmentation** (unused space in last page)
- Uses page table for address translation

5.4 Segmentation

- Divides memory into variable-size segments
- Each segment has base address and limit

Address Translation: ``` Logical Address: (Segment #, Offset) Physical Address = Base[Segment] + Offset

Validation: Offset must be < Limit[Segment] Otherwise: Segmentation Fault ```

Example: ``` Segment Table: Segment | Base | Limit 0 | 2000 | 800 1 | 4300 | 1200

Logical (0, 400): - Check: 400 < 800 ✓ - Physical = 2000 + 400 = 2400

Logical (1, 1500): - Check: 1500 < 1200 ✗ - Result: SEGMENTATION FAULT ```

---

6. Inter-Process Communication (IPC)

6.1 Pipes (Unnamed/Anonymous Pipes)

Characteristics: - Unidirectional communication - Parent-child process communication - No name in filesystem - Automatic cleanup when closed - FIFO buffer

Use Case: One-time data transfer from parent to child (e.g., 2 KB configuration data)

Advantages: - Simple to use - Automatic inheritance through fork() - No manual resource management

6.2 FIFOs (Named Pipes)

Characteristics: - Have a name in the filesystem - Can be used by **unrelated processes** - Persist until explicitly deleted - Unidirectional

Key Difference from Pipes: - Named pipes accessible by any process with permissions - Regular pipes only for parent-child

6.3 Shared Memory

Characteristics: - Fastest IPC mechanism - Multiple processes map same physical memory - **Zero-copy** data transfer - No system call overhead after setup

Advantages: 1. **High performance**: Direct memory access 2. **Efficient for large data**: Ideal for 100 MB transfers 3. **Scalable**: No size limits like message queues

Disadvantages: - **Requires explicit synchronization** (semaphores, mutexes) - More complex to implement - Risk of race conditions

Use Case: Frequently exchanging large amounts of data (100 MB)

6.4 Message Queues

Characteristics: - Messages sent/received in FIFO order - Automatic synchronization - Kernel manages the queue

Comparison with Shared Memory: - Slower (requires copying data twice: sender → kernel → receiver) - Easier to use (built-in synchronization) - Size limits - Better for small, structured messages

6.5 Sockets

Characteristics: - **Only IPC for network communication** (different machines) - Bidirectional communication - Can be used locally or over network

TCP Sockets (SOCK_STREAM)

Use for: File transfer, reliable data exchange

Advantages: - **Reliable**: Guaranteed delivery in correct order - **Connection-oriented**: Established connection - **Error checking**: Checksums and acknowledgments - **Automatic retransmission** of lost packets

UDP Sockets (SOCK_DGRAM)

Use for: Real-time streaming, video/audio

Characteristics: - Connectionless - Unreliable (no guaranteed delivery) - Faster (less overhead) - Packets can be lost or arrive out of order

---

7. Signals and Process Management

7.1 Common Unix/Linux Signals

SIGINT (Signal Interrupt)

- Signal number: 2
- Sent when user presses **Ctrl+C**
- Can be caught and handled
- Default: Terminate process

SIGTERM (Signal Terminate)

- Signal number: 15
- Request for **graceful termination**
- Can be caught and handled
- Allows cleanup operations
- Default signal sent by `kill` command

SIGKILL (Signal Kill)

- Signal number: 9
- **Cannot be caught or ignored**
- Forces immediate termination
- No cleanup possible
- Last resort for unresponsive processes

SIGCHLD (Signal Child)

- Sent by kernel to parent when child process terminates
- Parent should call `wait()` or `waitpid()` to collect exit status
- Prevents zombie processes

SIGSTOP

- Pauses process execution
- Cannot be caught or ignored
- Resumes with SIGCONT

7.2 Graceful Shutdown Example

Scenario: Web server with worker processes

Correct Signal Flow: `1. Admin sends SIGTERM to main server process 2. Server stops accepting new connections 3. Server sends SIGTERM to all worker children 4. Workers complete current requests (up to timeout) 5. Workers exit normally 6. Kernel sends SIGCHLD to parent for each child 7. Parent calls wait() to reap children 8. After timeout, parent sends SIGKILL to remaining workers 9. Parent performs cleanup and exits`

Common Mistakes: ```c // WRONG: Creates zombie processes void shutdown_handler(int sig) { for (each child) { kill(child_pid, SIGTERM); } exit(0); // Parent exits without wait() }

// CORRECT: Wait for children void shutdown_handler(int sig) { shutdown_flag = 1; // Set flag }

// In main loop: if (shutdown_flag) { for (each child) { kill(child_pid, SIGTERM); }

```
// Wait with timeout
alarm(30);  // 30-second timeout
while (wait() > 0);  // Reap all children

// Force kill remaining
for (each child) {
    kill(child_pid, SIGKILL);
}

cleanup_resources();
exit(0);
```

} ```

---

8. System Performance Analysis

8.1 Identifying Bottlenecks

Case Study 1: Disk I/O Bottleneck ``` Symptoms: - CPU usage: 25% - Memory usage: 40% - Disk I/O wait: 85% - Slow response time

Bottleneck: Disk I/O

Solutions: 1. Upgrade to SSD/NVMe drives 2. Implement caching (Varnish, Redis) 3. Increase RAM for filesystem cache 4. Optimize database queries 5. Implement RAID for better I/O ```

Case Study 2: CPU Scheduling Bottleneck ``` Symptoms: - 5 CPU-bound processes - 5 I/O-bound processes - CPU utilization: 95% - I/O utilization: 30% - Low throughput

Bottleneck: Poor scheduling (CPU-bound monopolizing CPU)

Solutions: 1. Higher priority for I/O-bound processes 2. Multi-level feedback queue scheduling 3. Shorter time quantum for I/O processes

Result: Better CPU/I/O overlap ```

8.2 Workload Optimization

CPU-Bound vs I/O-Bound Balance: - I/O-bound processes need frequent short CPU bursts - CPU-bound processes need longer continuous execution - **MLFQ** automatically balances by: - Moving I/O-bound to higher priority queues - Giving longer time slices to CPU-bound processes - Maximizing both CPU and I/O utilization

---

9. Key Concepts Summary

Process States

1. **New**: Process being created
2. **Ready**: Waiting for CPU
3. **Running**: Executing on CPU
4. **Waiting**: Waiting for I/O or event
5. **Terminated**: Finished execution

Context Switching

- Saving state of current process
- Loading state of next process
- Overhead depends on architecture

- **Monolithic kernels**: Less context switching
- **Microkernels**: More context switching

Zombie Processes

- Process has terminated but parent hasn't collected exit status
- Still occupies entry in process table
- **Prevention**: Parent must call `wait()` or `waitpid()`

Critical Section Problem

Three requirements: 1. **Mutual Exclusion**: Only one process in critical section 2. **Progress**: Selection of next process cannot be postponed indefinitely 3. **Bounded Waiting**: Limit on times other processes can enter before waiting process

---

10. Practical Applications

Choosing IPC Mechanisms

Scenario	Best Choice	Reason
Parent → Child (2 KB, one-time)	Pipe	Simple, automatic inheritance
Unrelated processes (same machine)	FIFO or Shared Memory	Named access
Large data (100 MB), frequent	Shared Memory	Zero-copy, high performance
Small structured messages	Message Queue	Built-in synchronization
Different machines	Sockets	Network communication
File transfer over network	TCP Sockets	Reliability required
Video streaming	UDP Sockets	Speed over reliability

Memory Allocation Strategy

Algorithm	Best For	Trade-off
First-Fit	Speed	Fast but fragmented
Best-Fit	Minimizing waste	Slower, tiny fragments
Worst-Fit	Large remaining holes	Fewer but larger holes
Next-Fit	Balanced distribution	Even spread of allocations

Scheduling Algorithm Selection

Workload	Algorithm	Reason
Batch processing	FCFS or SJF	Maximize throughput
Interactive systems	Round Robin	Fair, responsive
Real-time systems	Priority	Meet deadlines
Mixed CPU/I/O	MLFQ	Automatic optimization
Server with many clients	Priority with aging	Prevent starvation

---

11. Common Pitfalls and Best Practices

Synchronization

- Always protect shared data with proper synchronization
- Avoid deadlocks (circular wait)
- Minimize critical section size
- Use appropriate mechanism (mutex for exclusion, semaphore for counting)

Signal Handling

- Never call `exit()` directly in signal handler without cleanup
- Use flags in handlers, do work in main loop
- Always reap child processes to prevent zombies
- Use SIGTERM for graceful shutdown, SIGKILL as last resort

Memory Management

- Validate segment offsets before access

- Choose allocation algorithm based on workload
- Monitor fragmentation
- Consider paging for large address spaces

IPC Selection

- Match mechanism to use case
- Consider overhead vs performance needs
- Remember synchronization requirements
- Plan for error handling

Review Checklist

- [ ] Understand OS architecture differences (monolithic vs microkernel)
- [ ] Know boot process sequence
- [ ] Calculate scheduling metrics (TAT, WT, CT)
- [ ] Compare FCFS vs SJF performance
- [ ] Explain race conditions and synchronization solutions
- [ ] Differentiate fragmentation types
- [ ] Apply memory allocation algorithms
- [ ] Choose appropriate IPC mechanism for scenarios
- [ ] Understand signal types and proper usage
- [ ] Identify system bottlenecks from metrics
- [ ] Design graceful shutdown with proper signal handling
- [ ] Calculate physical addresses from logical addresses

End of Week 6 Journal