Betriebssysteme

Jil Zerndt
FS 2025

TODO: add images/graphics!!!!

Midterm 2012

Resource Management and Deadlocks

Resource Graph Analysis

Ein Rechnersystem besitzt zwei Tapestationen (T1, T2) und zwei Disks (D1, D2). Zur Zeit laufen drei Prozesse (P1, P2, P3), wobei folgendes gilt:

- Prozess P1 kopiert Daten von Disk D1 auf die Tapestation T2 und möchte Daten auf den Disk D2 schreiben
- Prozess P2 hat Tapestation T1 alloziert und möchte Daten auf Disk D2 schreiben
- Prozess P3 hat Disk D2 alloziert und möchte Daten nach Tapestation T2 kopieren

Ist das eine Deadlocksituation, wenn die Ressourcen exklusiv alloziert werden und wenn möchte schreiben das Gleiche wie anfordern bedeutet? Begründen Sie Ihre Antwort (Ressourcengraphen zeichnen und analysieren).

Ja, dies ist eine Deadlocksituation.

Analyse:

- P1 hält D1, möchte T2 und D2
- P2 hält T1, möchte D2
- P3 hält D2, möchte T2

Es entsteht ein Zyklus: P1 → D2 → P3 → T2 → P1

DRAW RESOURCE GRAPH! AND ADD HERE

Bedingungen für Deadlock:

- Mutual Exclusion: Ressourcen werden exklusiv alloziert
- No Preemption: Ressourcen können nicht weggenommen werden
- Hold & Wait: Prozesse halten Ressourcen und warten auf weitere
- Circular Wait: Es gibt einen Zyklus im Ressourcengraph

Alle vier Bedingungen sind erfüllt → Deadlock!

Deadlock Detection in Resource Allocation Graphs

Resource Graph erstellen

- Kreise: Prozesse (P1, P2, P3, ...)
- Rechtecke: Ressourcen (R1, R2, R3, ...)
- Pfeile von Prozess zu Ressource: Reqüst (Anforderung)
- Pfeile von Ressource zu Prozess: Allocation (Zuteilung)

Deadlock-Analyse

- Suche nach Zyklen im Graph
- Ein Zyklus bedeutet Deadlock bei Single-Instance Ressourcen
- Bei Multi-Instance Ressourcen: Prüfe ob alle Instanzen blockiert

Deadlock-Bedingungen prüfen

- Mutual Exclusion: Exklusive Ressourcennutzung
- Hold & Wait: Halten und gleichzeitig warten
- No Preemption: Keine Unterbrechung möglich
- Circular Wait: Zyklische Warteabhängigkeiten

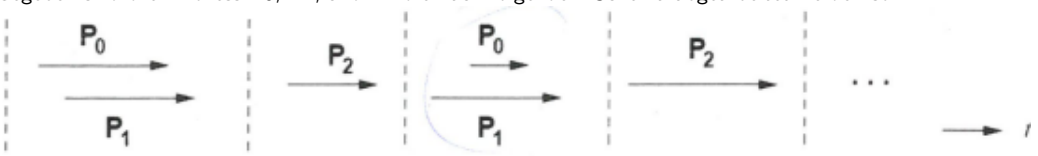
Lösungsansätze

- Prevention: Eine der vier Bedingungen verhindern
- Avoidance: Banker's Algorithm
- Detection & Recovery: Deadlock erkennen und auflösen

Synchronization with Semaphores

Semaphore Implementation

Gegeben sind drei Prozess P0, P1, und P2 die nach folgendem Schema abgearbeitet werden soll:



P0 → P2 → P0 → P2 → ...
P1 (parallel)

Die Verarbeitung startet mit den beiden Prozessen P0 und P1, die parallel verarbeitet werden sollen (es spielt kein Rolle, welcher der beiden Prozesse zürst mit seiner Verarbeitung beginnt oder aufhört). Wenn beide Prozesse eine Iteration ihrer Funktion working(x) beendet haben, folgt Prozess P2, etc.
Schreiben Sie Pseudocode mit maximal 3 Semaphoren S0, S1 und S3, der garantiert, dass die oben skizzierte Reihenfolge eingehalten wird. Verwenden Sie dazu ausschliesslich Befehle der Form up(S0) und down(S0), etc. Geben Sie an, wie die Semaphore initialisiert werden müssen.

ABGLEICHEN MIT LÖSUNG UND BESSER FORMATIEREN

P0	P1	P2
Sem S0: 1 Sem S3: 0	Sem S1: 1 Sem S3: 0	Sem S2: 0

while(1) {
down(S0);
working(0);
up(S3);
}
	while(1) {
	down(S1);
	working(1);
	up(S3);
	}
		while(1) {
		down(S2);
		working(2);
		up(S0);
		up(S1);
		}

Initialisierung:

- S0 = 1 (P0 kann starten)
- S1 = 1 (P1 kann starten)
- S2 = 0 (P2 muss warten)
- S3 = 0 (Synchronisation zwischen P0/P1 und P2)

Ablauf:

- P0 und P1 starten parallel
- Beide signalisieren mit up(S3) wenn fertig
- P2 wartet mit down(S2) bis beide P0 und P1 fertig sind
- P2 gibt mit up(S0) und up(S1) die nächste Runde frei

Semaphore-basierte Synchronisation

Semaphore verstehen

- Semaphore S ist ein Zähler mit atomaren Operationen
- down(S) oder P(S): Dekrementiert S, blockiert bei $S \leq 0$
- up(S) oder V(S): Inkrementiert S, weckt wartende Prozesse
- Initialisierung bestimmt verfügbare Ressourcen

Synchronisationsmuster

- Mutual Exclusion: Binary Semaphore (0/1) um kritische Bereiche
- Signaling: Ein Prozess signalisiert einem anderen (Producer-Consumer)
- Rendezvous: Zwei Prozesse warten aufeinander
- Barrier: Mehrere Prozesse warten aufeinander

Typische Synchronisationsprobleme

- Producer-Consumer: Buffer-Management mit vollen/leeren Plätzen
- Reader-Writer: Mehrere Leser oder ein Schreiber
- Dining Philosophers: Deadlock-Vermeidung bei zyklischen Abhängigkeiten
- Barrier Synchronization: Alle warten aufeinander

Implementierungsschritte

- Identifiziere Synchronisationspunkte
- Bestimme benötigte Semaphore und deren Initialisierung
- Verwende down() vor kritischen Bereichen/Warten
- Verwende up() nach kritischen Bereichen/Signaling
- Teste auf Deadlock-Freiheit und korrekte Reihenfolge

Memory Management

Buddy System

Ein Betriebssystem-Kernel verwaltet seine Datenbuffer mit einem Buddy System, wobei insgesamt 8MByte Speicher zur Verfügung stehen. Zur Zeit sind folgende Buffer mit 62KByte, 34KByte und 9KByte alloziert worden. Wie viel Speicher geht dabei durch interne Fragmentierung insgesamt verloren, Angabe in KByte:

Buddy System alloziert in Potenzen von 2:

- 62 KByte → nächste 2er-Potenz: 64 KByte
- 34 KByte → nächste 2er-Potenz: 64 KByte
- 9 KByte → nächste 2er-Potenz: 16 KByte

Interne Fragmentierung = Alloziert - Angefordert:

- (64 - 62) = 2 KByte
- (64 - 34) = 30 KByte
- (16 - 9) = 7 KByte

Gesamt: 2 + 30 + 7 = **39 KByte** interne Fragmentierung

Page Tables

Ein Prozessor besitzt eine Wortbreite von 32Bit. Pointer (Adressen) werden in 32Bit Worten gespeichert, aber nur die 24 tieferwertigen Bits werden für die Adressbildung verwendet (Bits 25-31sind auf 0 gesetzt).
Die logische Adresse ist wie folgt strukturiert:

6-Bit Page Directory	8-Bit Page Nummer	10-Bit Offset

a) Wie gross ist eine Page, Angabe in KBytes?
b) Wie viele Bytes enthält das Page Directory, wenn pro Eintrag ein Pointer (Adresse) auf eine Page Tabelle eingetragen wird, Angabe in KBytes?
c) Wie viele Page Tabellen kann ein Prozess maximal haben?
d) Wie viele Frames kann das System maximal haben?

Lösung:
a) Page-Grösse = 2^{10} Bit = 1024 Bytes = **1 KByte**
b) Page Directory:

- 6 Bit → 2^6 = 64 Einträge
- 4 Bytes pro Adresse
- 64 × 4 = 256 Bytes = **0.25 KBytes**

c) Page Tabellen: 6 Bit → 2^6 = **64 Page Tabellen**
d) Frames im System:

- 24 Bit physische Adresse
- 10 Bit Offset pro Frame
- Frame-Bits: 24 - 10 = 14 Bit
- Maximale Frames: 2^{14} = **16384 Frames**

Memory Management Analysis

Buddy System

- Alloziert in Potenzen von 2 (1, 2, 4, 8, 16, ... KByte)
- Interne Fragmentierung = Alloziert - Angefordert
- Nächste grössere 2er-Potenz finden: $2^n \geq$ Anforderung
- Beispiel: 33 KByte → 64 KByte (2^6)

Page Table Berechnung

- Page-Grösse = $2^{\text{Offset-Bits}}$ Bytes
- Anzahl Pages = $2^{\text{Page-Number-Bits}}$
- Page Directory Grösse = $2^{\text{Directory-Bits}}$ × Pointer-Grösse
- Maximale Frames = $2^{(\text{Physische-Adress-Bits - Offset-Bits})}$

Adressaufteilung

- Logische Adresse = Page Directory + Page Number + Offset
- Physische Adresse = Frame Number + Offset
- Page Table Entry enthält Frame Number
- Translation: Page Number → Frame Number

Memory Hierarchie

- Page Directory zeigt auf Page Tables
- Page Tables zeigen auf Frames
- Mehrstufige Page Tables reduzieren Speicherbedarf
- TLB cached häufig verwendete Translations

Page Replacement Algorithms

Least Recently Used (LRU)

Ein Prozess referenziert der Reihe nach folgende Pages:

8 7 5 8 7 3 5 1 3 4 2 1 8 3 1 2

Gehen Sie davon aus, dass zu Beginn keine Pages im Speicher stehen und dass auch das erstmalige Laden einer Page als Page Fault gezählt wird (demand paging). Pro Prozess stehen 4 Frames zur Verfügung.
Tragen Sie in unten stehender Tabelle die den Frames zugewiesenen Pages für den Least Recently Used Algorithmus. Das Page mit diesem Zugriff am weitesten zurückliegend wird dann mit ein neüs Page ersetzt.
Markieren Sie die Spalten mit einem Stern, wo ein Page Fault auftritt. Nehmen Sie an, dass ausschliesslich Demand Paging verwendet wird. Wenn mehrere Frames für das placement resp. replacement in Frage kommen, muss der Frame mit der kleinsten Nummer gewählt werden.

Lösung:

Referenzen	8	7	5	8	7	3	5	1	3	4	2	1	8	3	1	2
frame 1	8*	8	8	8	8	8	8	1*	1	1	1	1	1	1	1	1
frame 2		7*	7	7	7	7	7	7	7	4*	4	4	8*	8	8	8
frame 3			5*	5	5	5	5	5	5	5	2*	2	2	2	2	2
frame 4						3*	3	3	3	3	3	3	3	3	3	3
page fault	×	×	×			×		×		×	×		×			

Page Faults: 8 von 16 Zugriffen
LRU-Logik:

- Bei jedem Zugriff wird die "letzte Verwendung"der Page aktualisiert
- Bei einem Page Fault wird die am längsten nicht verwendete Page ersetzt
- Zeitstempel oder Zugriffs-Historie bestimmen LRU-Page

Page Replacement Algorithms

Least Recently Used (LRU)

- Ersetze die Page, die am längsten nicht verwendet wurde
- Gute Performance, aber aufwändig zu implementieren
- Benötigt Zeitstempel oder Zugriffs-Historie
- Approximation durch Clock Algorithm oder Second Chance

First-In-First-Out (FIFO)

- Ersetze die älteste Page (first loaded)
- Einfach zu implementieren mit Qüü
- Kann zu Belady's Anomaly führen
- Nicht optimal, da alte Pages oft noch verwendet werden

Optimal (OPT)

- Ersetze Page, die am spätesten wieder verwendet wird
- Theoretisch optimal, praktisch nicht implementierbar
- Benötigt Zukunftswissen über Page-Zugriffe
- Wird als Benchmark für andere Algorithmen verwendet

Implementation Tips

- Page Fault tritt auf bei erstem Zugriff auf neü Page
- Bei mehreren Kandidaten: Wähle Frame mit kleinster Nummer
- Clock Algorithm: Circular list mit Reference Bit
- Working Set: Berücksichtige lokale vs. globale Ersetzung

Operating System Concepts

Multiple Choice Qüstions

Pro Teilfrage können eine, mehrere oder keine Antworten zutreffen, kreuzen Sie die richtige(n) Antwort(en) an.

a) Welche der folgenden Aussagen treffen zu?
- ✓ Alle Mutexes können mit Semaphoren realisiert werden
- × Semaphore können in allen Fällen durch Mutexes ersetzt werden
- × Mutexes sollten immer anstelle von Semaphoren eingesetzt werden, weil es dann keine Deadlocks geben kann
- ✓ Mit Semaphoren lässt sich die Verarbeitungsreihenfolge von Prozessen und Threads erzwingen

b) Welcher der folgenden Aussagen treffen zu?
- × Pages müssen grösser als Frames dimensioniert werden
- × Es müssen mindestens so viele Pages wie Frames in einem System vorhanden sein
- × Sowohl interne wie auch externe Fragmentierung treten bei Paging auf
- ✓ Pages und Frames müssen gleich gross dimensioniert werden

c) Welcher der folgenden Aussagen treffen zu?
- ✓ Ein MMU übersetzt Logischen Adressen zu Physikalische Adressen
- × Swap in bedeutet dass ein Prozess auf die Hard-Disk verlagert wird
- × Ein Prozess der auf der Harddisk verlagert worden ist kann sich im Zustand Running befinden

d) Welcher der folgenden Aussagen treffen zu?
- × Bei Static Partitioning tritt External Fragmentation auf
- ✓ Bei Dynamic Partitioning tritt External Fragmentation auf
- × Nur bei Best Fit Allocation wird kein Compaction benötigt

Erklärungen:

a) Semaphore vs. Mutexes:
- ✓ Mutexes sind Binary Semaphores (0/1) - können mit Semaphoren realisiert werden
- × Counting Semaphores (>1) können nicht durch Mutexes ersetzt werden
- × Auch mit Mutexes sind Deadlocks möglich
- ✓ Semaphore eignen sich gut für Prozess-Synchronisation

b) Paging:
- × Pages und Frames sind gleich gross
- × Anzahl ist unabhängig - Virtual Memory ermöglicht mehr Pages als Frames
- × Nur interne Fragmentierung (innerhalb Pages)
- ✓ Pages (logisch) = Frames (physisch) in der Grösse

c) Memory Management:
- ✓ MMU (Memory Management Unit) macht Address Translation
- × Swap in = von Disk in Memory (nicht umgekehrt)
- × Ausgelagerte Prozesse sind nicht Running

d) Partitioning:
- × Static: Interne Fragmentierung (feste Grössen)
- ✓ Dynamic: Externe Fragmentierung (variable Grössen)
- × Alle Dynamic Allocation Methoden benötigen eventüll Compaction

Operating System Core Concepts

Synchronisation Mechanisms

- Mutex: Binary lock (0/1) für kritische Bereiche
- Semaphore: Counting mechanism für Resource Management
- Monitor: High-level synchronization construct
- Condition Variables: Wait/Signal mechanism

Memory Management

- Pages: Logische Memory-Einheiten (Virtual Memory)
- Frames: Physische Memory-Einheiten (Physical Memory)
- MMU: Hardware für Address Translation
- TLB: Cache für Address Translation

Process States und Swapping

- Running: Prozess auf CPU
- Ready: Bereit zur Ausführung
- Blocked: Wartet auf I/O oder Resource
- Swapped: Auf Disk ausgelagert (nicht Ready/Running)

Memory Allocation

- Static Partitioning: Feste Grössen → Interne Fragmentierung
- Dynamic Partitioning: Variable Grössen → Externe Fragmentierung
- Paging: Feste Page/Frame Grösse → Nur interne Fragmentierung
- Compaction: Löst externe Fragmentierung durch Memory-Reorganisation

Diese Exam-Beispiele decken die wichtigsten Konzepte des Betriebssystem-Kurses ab:

- Deadlock Detection und Resource Management
- Process Synchronization mit Semaphoren
- Memory Management (Buddy System, Paging)
- Page Replacement Algorithms
- Grundlegende OS-Konzepte

Die KRs bieten systematische Herangehensweisen für ähnliche Aufgaben in Prüfungen.

Midterm 2019

This chapter contains typical exam questions with detailed solutions and systematic approaches (Kochrezepte) for solving similar problems.

Cache Hit Rate Analysis

Cache Hit Rate Calculation

Problem identification

- Given: Array access pattern, processor architecture, cache block size
- Find: Cache hit rate during sequential array access

Solution approach

- Calculate data type size based on processor architecture
- Determine how many array elements fit in one cache line
- First access to cache line = miss, subsequent accesses = hits
- Hit rate = (accesses per line - 1) / accesses per line

Key formulas

- Elements per cache line = Cache line size / Data type size
- Hit rate = (Elements per line - 1) / Elements per line
- For 32-bit processor: int = 4 bytes
- For 64-bit processor: int = 4 bytes, long = 8 bytes

Cache Hit Rate Calculation Given program code accessing an integer array:

```
#define N (10*1000*1000)
int arVL[N];
for (int i = 0; i < N; i++) {
    sum += arVL[i];
}
```

Calculate hit rate on a 32-bit processor with 64-byte cache lines.

Solution:

- 32-bit processor: `int` = 4 bytes
- Cache line size: 64 bytes
- Elements per cache line: 64 / 4 = 16 integers
- First access per line: cache miss
- Next 15 accesses per line: cache hits
- Hit rate: $h_c = \frac{15}{16} = 0.9375 = 93.75\%$

Memory Access Time Calculation

Average Memory Access Time

Problem components

- Multi-level cache hierarchy
- Hit rates for each level
- Access times for each level
- Main memory access time

Calculation method

- Start from L1 cache (always accessed first)
- For each miss, calculate probability of accessing next level
- Multiply access time by probability of reaching that level
- Sum all weighted access times

Formula

- $T_{avg} = T_{L1} + (1 - h_1) \times T_{L2} + (1 - h_1)(1 - h_2) \times T_{L3} + \ldots$
- Convert clock cycles to time: Time = Cycles × Clock period
- Clock period = 1 / Frequency

Memory Access Time Calculation 2 GHz processor (clock cycle = 0.5 ns) with three-level cache:

Cache Level 1	4 clock cycles
Cache Level 2	10 clock cycles
Cache Level 3	40 clock cycles
Main Memory	60 ns

Hit rate on each level: 90%

Solution:

- L1 access: $4 \times 0.5 = 2$ ns (always accessed)
- L2 access: $10 \times 0.5 = 5$ ns (10% probability)
- L3 access: $40 \times 0.5 = 20$ ns (1% probability)
- Main memory: 60 ns (0.1% probability)
- $T_{avg} = 2 + 0.1 \times 5 + 0.01 \times 20 + 0.001 \times 60$
- $T_{avg} = 2 + 0.5 + 0.2 + 0.06 = 2.76$ ns

Makefile Analysis

Makefile Dependency Analysis

Understanding make behavior

- Make compares timestamps of targets and dependencies
- Target is rebuilt if it's older than any dependency
- Missing targets are always built
- Implicit rules (.c.o:) handle automatic compilation

Analysis steps

- Identify target dependencies from makefile
- Check file timestamps in directory listing
- Determine which object files need recompilation
- Follow linking rules to determine final executable name

Common patterns

- .c.o: rule compiles .c files to .o files automatically
- $@ refers to the target name
- If .c file is newer than .o file, recompilation occurs
- Missing .o files are always created

Makefile Analysis Given Makefile:

```
CFL = -g
CMP = gcc $(CFL)
app: main1.o mythread.o scheduler.o queues.o mylist.o
 $(CMP) main1.o mythread.o scheduler.o queues.o mylist.o -o $@.e
.c.o:
 $(CMP) -c $<
```

Directory listing shows:

- mylist.c: Feb 24 09:31 (newer)
- mylist.o: Feb 24 09:21 (older)
- queues.c: Feb 24 09:25 (newer)
- queues.o: Feb 24 09:21 (older)
- All other .o files are newer than corresponding .c files

Solution:

- Files to be compiled: mylist.c, queues.c (because .c files are newer than .o files)
- Executable name: app.e (from $@.e where $@ = app)

Process Creation with fork()

Fork() Process Tree Analysis

Understanding fork() behavior

- fork() creates an exact copy of the calling process
- Returns child PID to parent, 0 to child
- fork() $>$ 0 is true only in parent process
- Each fork() doubles the number of processes

Analysis method

- Draw a process tree starting with the original process
- For each fork(), branch into parent and child
- Track which path each process takes through if/else statements
- Count total processes and specific execution paths

Counting strategy

- Start with 1 process (the original)
- Each successful fork() adds 1 new process
- Count processes that reach specific code sections
- Be careful with nested forks and conditional execution

Fork() Process Creation Analyze this code snippet:

```
if (fork() > 0)
    fork();
else {
    fork();
    if (fork() > 0)
        printf("Hello World\n");
}
```

Assume all fork() calls succeed.

Solution:
Process tree analysis:

- Original process P1 forks $\rightarrow$ creates P2
- P1 (parent): fork() $>$ 0 is true, executes fork() $\rightarrow$ creates P3
- P2 (child): fork() $>$ 0 is false, goes to else block
- P2 executes fork() $\rightarrow$ creates P4
- P2 executes second fork() $\rightarrow$ creates P5
- P4 executes second fork() $\rightarrow$ creates P6
- P2 (parent of P5): fork() $>$ 0 is true, prints "Hello World"
- P4 (parent of P6): fork() $>$ 0 is true, prints "Hello World"

Results:

- Additional processes created: 5 (P2, P3, P4, P5, P6)
- "Hello World"printed: 2 times (by P2 and P4)

Real-Time Scheduling

Earliest Deadline First Scheduling

EDF Algorithm principles

- Always schedule the task with the earliest deadline
- Deadlines = arrival time + period for periodic tasks
- Preemption occurs when a task with earlier deadline arrives
- Re-scheduling happens at specified intervals or task completion

Scheduling steps

- Calculate all task deadlines for the scheduling period
- At each scheduling point, identify available tasks
- Select task with earliest deadline
- Handle ties with specified priority rules (e.g., shorter period)
- Mark execution periods in timeline diagram

Timeline construction

- Create timeline with scheduling intervals marked
- For each interval, determine which task should run
- Show task execution as filled blocks
- Verify all deadlines are met

EDF Real-Time Scheduling Three periodic tasks with EDF scheduling:

Task	Period	Execution Time
1	4ms	1ms
2	8ms	4ms
3	12ms	3ms

Re-scheduling every 4ms or when task suspends. For equal deadlines, shorter period has higher priority.

Solution approach:
Timeline analysis (0-12ms):

- t=0: T1 (deadline 4), T2 (deadline 8), T3 (deadline 12)
- T1 has earliest deadline $\rightarrow$ runs 0-1ms
- t=1: T2 (deadline 8), T3 (deadline 12)
- T2 has earliest deadline $\rightarrow$ runs 1-4ms (suspended after 3ms)
- t=4: T1 arrives again (deadline 8), T2 continues (1ms left), T3 (deadline 12)
- Multiple tasks with deadline 8: T1 has shorter period $\rightarrow$ higher priority
- Continue this analysis through the full timeline

Multi-Level Scheduling

Multi-Level Round Robin Scheduling

Multi-level principles

- Separate queues for different priority levels
- Higher priority queues are served first
- Round robin within each priority level
- Lower priority tasks only run when higher queues are empty

Scheduling algorithm

- Group processes by priority level
- Always check highest priority queue first
- Use round robin with specified time quantum within each queue
- Preemption occurs when higher priority process arrives
- Track remaining execution time for each process

Timeline construction

- Mark process arrivals on timeline
- At each time unit, determine which process should run
- Apply round robin within the highest non-empty priority queue
- Show context switches and queue changes

Multi-Level Scheduling Five processes with priorities and times:

Process	Priority	Arrival	Execution
P1	0	0	4
P2	1	1	3
P3	1	2	2
P4	0	5	3
P5	0	7	2

Multi-level scheduling (not feedback), no blocking, RR with q=1. Higher number = higher priority.

Solution timeline (0-14):

- t=0-1: P1 runs (only process, priority 0)
- t=1-2: P2 runs (arrives with priority 1, preempts P1)
- t=2-3: P3 runs (priority 1, round robin with P2)
- t=3-4: P2 runs (continues in priority 1 queue)
- t=4-5: P3 runs (finishes remaining 1 time unit)
- t=5-6: P2 runs (finishes last time unit)
- t=6-7: P1 runs (priority 1 queue empty, resume P1)
- Continue pattern with P4, P5 arrivals...

Operating System Concepts

OS Concept Questions

System call identification

- System calls required for kernel mode operations
- I/O operations (screen output, file access) need system calls
- Time-related operations (sleep) need system calls
- Pure computation (arithmetic) doesn't need system calls
- Memory operations within process space don't need system calls

Thread and process concepts

- pthread_join() waits for thread completion
- Prevents main process from terminating before threads finish
- Software interrupts provide controlled kernel mode entry
- PCB (Process Control Block) stores OS process information

Process states

- Zombie: process finished but parent hasn't waited
- Daemon: background process without controlling terminal
- Orphan: process whose parent has terminated

Operating System Multiple Choice Typical OS concept questions:

Which activities require system calls?

- ✓ Display variable value on screen (I/O operation)
- ✗ Compare two variables (CPU operation)
- ✓ Sleep for 1 second (timer operation)
- ✗ Increment integer variable (memory operation)
- ✗ Increment float variable (CPU operation)

What is pthread_join() used for?

- ✗ Terminates all user-level threads
- ✗ Can be replaced by sleep(0)
- ✓ Prevents process termination before threads finish
- ✗ Blocks CPU until all threads terminate

Why use software interrupts for system calls?

- ✗ They are faster than procedure calls
- ✓ Procedure calls cannot switch to system mode
- ✓ Applications don't know system call routine addresses

Exam Preparation Strategy

Time management

- Read all questions first to identify easy wins
- Allocate time based on point values
- Don't spend too much time on single difficult problems
- Reserve time for checking answers

Problem-solving approach

- Identify the problem type and appropriate "Kochrezept"
- Write down given values and what needs to be found
- Draw diagrams for scheduling and process problems
- Show calculation steps clearly
- Double-check units and reasonableness of answers

Common mistakes to avoid

- Forgetting to convert time units (ns, ms, clock cycles)
- Not considering all processes in fork() analysis
- Missing preemption in scheduling problems
- Confusing parent/child behavior in process creation
- Not reading multiple choice questions carefully

SEP 2007

Processes and Threads

Process vs. Thread Differences Explain the difference between processes and threads:

Solution: The main difference is that processes have their own memory space, while threads (belonging to the same process) share the process memory space.

Processes:
- Independent memory spaces
- Higher creation/switching overhead
- Better isolation and fault tolerance
- Communication via IPC mechanisms

Threads:
- Shared memory space within process
- Lower creation/switching overhead
- Faster communication via shared memory
- Risk of interference between threads

Analyzing Process vs. Thread Questions

Key comparison points

- Memory organization: separate vs. shared address space
- Creation overhead: high vs. low
- Communication methods: IPC vs. shared memory
- Isolation level: strong vs. weak
- Context switching cost: expensive vs. cheap

Common exam patterns

- Define fundamental differences
- Compare advantages/disadvantages
- Explain when to use which approach
- Analyze code examples with fork() vs. pthread_create()

Process Creation with fork() Unix systems use fork() for process creation. Describe the system call:

fork() characteristics:
- **Parameters**: None
- **Return value**: Process ID (PID)
  - Parent process receives child's PID
  - Child process receives 0
  - Error case returns -1
- **Behavior**: Creates exact copy of calling process

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        // Error occurred
        perror("Fork failed");
    } else if (pid == 0) {
        // Child process
        printf("Child process: PID = %d\n", getpid());
    } else {
        // Parent process
        printf("Parent: Child PID = %d\n", pid);
    }

    return 0;
}
```

Process Creation Analysis

fork() behavior pattern

- One call, two returns (parent and child)
- Check return value to determine process role
- Parent gets child PID, child gets 0
- Error handling: check for negative return value

Common exam scenarios

- Trace execution flow through fork() calls
- Count total processes created
- Identify output patterns
- Analyze exec() calls after fork()

Thread Programming with POSIX Write a program that creates two threads, both printing "Hello World:

```c
#include <stdio.h>
#include <pthread.h>

void *hello_world(void *arg) {
    printf("Hello World\n");
    return NULL;
}

int main() {
    pthread_t thread[2];

    // Create two threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&thread[i], NULL, hello_world, NULL);
    }

    // Wait for both threads to complete
    for (int i = 0; i < 2; i++) {
        pthread_join(thread[i], NULL);
    }

    return 0;
}
```

Note: Threads created with pthread_create() start immediately.

POSIX Thread Programming

Essential POSIX thread functions

- pthread_create(): Create new thread
- pthread_join(): Wait for thread completion
- pthread_exit(): Terminate calling thread
- pthread_detach(): Detach thread (no join needed)

Programming pattern

- Include pthread.h header
- Define thread function with void* signature
- Create threads in loop if multiple needed
- Always join threads to avoid zombies
- Compile with -lpthread flag

Process States and Interrupts

Process State Transitions Name and explain the three main states processes can transition between:

The three fundamental process states:
- **Running**: Process is currently executing on CPU
- **Ready**: Process is ready to execute, waiting for CPU assignment
- **Blocked/Sleeping**: Process cannot execute, waiting for an event (e.g., I/O completion)

State transitions:
- Ready → Running: Scheduler assigns CPU
- Running → Ready: Time quantum expires or preemption
- Running → Blocked: Process waits for I/O or resource
- Blocked → Ready: Awaited event occurs

Process State Analysis

State identification
- Running: Has CPU, actively executing
- Ready: Waiting for CPU only
- Blocked: Waiting for external event/resource
- Additional states: Swapped, Zombie, Created

Transition triggers
- Timer interrupts (quantum expiration)
- I/O operations (blocking)
- System calls
- Resource availability
- Scheduler decisions

Interrupt Handling Explain alternatives to interrupts and why interrupts improve performance:

Alternative: Polling
- OS continuously checks device status
- Wastes CPU time on unnecessary checks
- Creates busy-waiting loops

Interrupt advantages:
- CPU only responds when device needs attention
- Eliminates wasteful polling loops
- Allows CPU to focus on productive work
- Enables asynchronous I/O operations

Interrupt handling strategies:
- Non-nested: Disable interrupts during handling (simple but may miss urgent interrupts)
- Priority-based: Higher priority interrupts can preempt lower priority ones (complex but responsive)

Interrupt System Analysis

Interrupt vs. polling comparison
- Efficiency: Interrupts save CPU cycles
- Responsiveness: Interrupts provide better response time
- Complexity: Polling is simpler to implement
- Resource usage: Interrupts minimize waste

Interrupt handling strategies
- Nested vs. non-nested interrupts
- Priority schemes
- Top-half/bottom-half division (Linux)
- Maskable vs. non-maskable interrupts

Scheduling Algorithms

FIFO, SJF, and SRT Scheduling Given processes with arrival times and burst times, determine execution order:

Process	Arrival Time	Burst Time
P	0	10
Q	4	5
R	5	10
S	6	1

FIFO (First Come First Served):
Time: 0-10(P), 10-15(Q), 15-25(R), 25-26(S)
SJF (Shortest Job First):
Time: 0-10(P), 10-11(S), 11-16(Q), 16-26(R)
SRT (Shortest Remaining Time):
Time: 0-4(P), 4-6(Q), 6-7(S), 7-12(Q), 12-18(P), 18-28(R)

Scheduling Algorithm Analysis

FIFO/FCFS approach
- Execute processes in arrival order
- Non-preemptive
- Simple but may cause long wait times

SJF approach
- At decision point, choose process with shortest total time
- Non-preemptive
- Optimal for average waiting time

SRT approach
- At each time unit, choose process with shortest remaining time
- Preemptive version of SJF
- May cause frequent context switches

Problem-solving steps
- Create timeline showing all events (arrivals, completions)
- For each algorithm, trace through decision points
- Calculate metrics: turnaround time, waiting time, response time

Round Robin Scheduling Why should the time quantum be larger than typical interaction time?

Reason: If the quantum is larger than interaction processing time, the entire user interaction can be completed within one quantum. This ensures the system responds quickly to user input without interrupting the interactive process.
If the quantum is too small, interactive processes get interrupted before completing their response, leading to poor user experience.

Round Robin Analysis

Quantum size considerations
- Too large: Approaches FIFO, poor response time
- Too small: High context switching overhead
- Optimal: Slightly larger than typical interaction time

Performance factors
- Context switch overhead
- Interactive response requirements
- CPU-bound vs. I/O-bound process mix
- System throughput vs. responsiveness trade-off

System Calls and Memory Management

System Call Analysis Analyze the output of a program using fork() and exec():

```
int pid = fork();
printf("%s\n", "[1] Time for case distinction");
if (pid) {
    printf("%s\n", "[2] Starting emacs/fstab");
    execl("/bin/emacs", "/etc/fstab", (char *)NULL);
} else {
    printf("%s\n", "[3] Starting emacs/hosts");
    execl("/bin/emacs", "/etc/hosts", (char *)NULL);
}
printf("%s\n", "[4] Two editors started successfully");
// More code follows...
```

Analysis:
- Two editors are started
- Output: [1], [1], [2], [3]
- The code after execl() calls is never reached
- execl() replaces the process image, so [4] is never printed

Execution flow:
- fork() creates parent and child
- Both print [1]
- Parent (pid $\neq$ 0) prints [2], then execl() replaces it with emacs
- Child (pid == 0) prints [3], then execl() replaces it with emacs

Fork and Exec Analysis

Key concepts
- fork() duplicates current process
- exec() family replaces current process image
- Code after exec() only runs if exec() fails
- Count processes by tracing fork() calls

Analysis steps
- Draw execution tree showing parent/child branches
- Mark where each printf() executes
- Identify exec() calls that replace process image
- Determine unreachable code after successful exec()

Virtual Memory Address Translation Given: 16KB page size, 47-bit virtual addresses, 3-level paging, 8-byte page table entries.
How is a virtual address structured?

Calculation:
- Page size: 16KB = 2^{14} bytes $\rightarrow$ 14 bits for page offset
- Remaining bits: 47 - 14 = 33 bits for page table indexing
- 3 levels: 33 $\div$ 3 = 11 bits per page table level

Address structure:

Bits 46-36	Bits 35-25	Bits 24-14	Bits 13-0
Level 1 (11 bit)	Level 2 (11 bit)	Level 3 (11 bit)	Offset (14 bit)

Page table sizes:
- Level 1: 1 table with 2^{11} = 2048 entries
- Level 2: 2048 tables with 2048 entries each
- Level 3: 2048^2 = 4,194,304 tables with 2048 entries each
- Each table: 2048 $\times$ 8 bytes = 16KB (exactly one page)

Virtual Memory Address Analysis

Address structure calculation
- Page offset bits = log_2(page size in bytes)
- Remaining bits = total address bits - offset bits
- Bits per level = remaining bits $\div$ number of levels

Page table size calculation
- Entries per table = $2^{\text{bits per level}}$
- Table size = entries $\times$ entry size in bytes
- Number of tables per level = cumulative from higher levels

Common mistakes to avoid
- Forgetting to account for page offset bits
- Mixing up table levels in size calculations
- Not considering that tables should fit in pages

Synchronization and File Systems

Producer-Consumer Problem Implement a semaphore-based solution for the producer-consumer problem:

```
const int N = 4;          // Maximum buffer size
int sem_write = N;        // Semaphore for write slots
int sem_read = 0;         // Semaphore for read slots

// Producer process
while(1) {
    wait(sem_write);      // Wait for empty slot
    write(value);         // Write to buffer
    signal(sem_read);     // Signal data available
}

// Consumer process
while(1) {
    wait(sem_read);       // Wait for data
    read(value);          // Read from buffer
    signal(sem_write);    // Signal slot available
}
```

Key points:

- sem_write tracks empty buffer slots
- sem_read tracks filled buffer slots
- Producer waits for empty slots, signals filled slots
- Consumer waits for filled slots, signals empty slots

Synchronization Problem Solving

Producer-Consumer pattern

- Identify shared resource (buffer)
- Determine capacity constraints
- Create semaphores for available/used slots
- Producer: wait(empty), produce, signal(full)
- Consumer: wait(full), consume, signal(empty)

General synchronization approach

- Identify critical sections
- Determine synchronization requirements
- Choose appropriate mechanism (mutex, semaphore, monitor)
- Ensure deadlock avoidance
- Test for race conditions

File System Free Space Management Describe two efficient methods for tracking free blocks:

Method 1: Bitmap

- Each bit represents one block
- 0 = free, 1 = allocated (or vice versa)
- Fast scanning for free blocks
- Compact representation
- Requires main memory for efficiency

Method 2: Free block list with (start, length) tuples

- List of contiguous free block ranges
- Each entry: (starting block, number of blocks)
- Efficient for large contiguous areas
- Dynamic size based on fragmentation

File System Design Analysis

Free space management methods

- Bitmap: Good for uniform access, fixed size
- Linked list: Simple but slow random access
- Grouping: Combines free blocks into groups
- Counting: Stores (address, count) pairs

Evaluation criteria

- Space efficiency
- Access speed
- Implementation complexity
- Fragmentation handling
- Main memory requirements

Page Replacement and Memory Management

Page Replacement Algorithms Compare LRU and FIFO page replacement algorithms:

LRU (Least Recently Used) typically causes fewer page faults because:

- Exploits locality principle
- Recently accessed pages likely to be accessed again soon
- Keeps "hot"pages in memory longer
- Better prediction of future access patterns

FIFO (First In, First Out):

- Simple implementation
- No consideration of access patterns
- May replace frequently used pages
- Can suffer from Belady's anomaly

Optimal algorithm (theoretical):

- Replace page that will be accessed furthest in future
- Not implementable (requires future knowledge)
- Used as performance benchmark

Page Replacement Analysis

Algorithm comparison factors

- Page fault frequency
- Implementation complexity
- Hardware support requirements
- Performance under different workloads
- Memory overhead for bookkeeping

Common algorithms

- FIFO: Simple, poor performance
- LRU: Good performance, moderate complexity
- Clock: Approximates LRU, hardware efficient
- Optimal: Theoretical best, not implementable

Advanced Topics

Priority Inversion Problem Explain priority inversion and how to prevent it:

Priority inversion occurs when:
- High-priority process waits for low-priority process
- Medium-priority processes preempt low-priority process
- High-priority process effectively blocked by medium-priority processes

Prevention methods:
- **Priority inheritance**: Low-priority process temporarily inherits high priority
- **Priority ceiling**: Resources assigned maximum priority of potential users
- **Dynamic priority adjustment**: Increase priority of waiting processes over time

Priority-Based Scheduling Issues

Common priority problems

- Priority inversion
- Starvation of low-priority processes
- Priority assignment difficulties
- Real-time deadline misses

Solution approaches

- Priority inheritance protocols
- Aging mechanisms
- Fair-share scheduling
- Deadline-based scheduling

Linux O(1) Scheduler Explain why the Linux O(1) scheduler is called O(1):

O(1) refers to constant time complexity:
- Time to find next process is independent of total number of processes
- Uses active and expired arrays with 140 priority levels each
- Bitmap indicates which priority levels have waiting processes
- Simply finds highest priority bit set and takes first process from that queue

Algorithm:
- Check bitmap for highest priority with waiting processes
- Take first process from that priority queue
- Constant time regardless of system load

Algorithm Complexity Analysis

Time complexity evaluation

- O(1): Constant time, independent of input size
- O(log n): Logarithmic time, scales well
- O(n): Linear time, proportional to input size
- Identify bottleneck operations

Scheduler efficiency factors

- Process selection time
- Context switch overhead
- Load balancing cost
- Priority calculation complexity

SEP 2019

Cache Performance Analysis

Cache Hit Rate Calculation

Given information analysis

- Identify data type size (e.g., int = 4 bytes on 32-bit system)
- Note cache line size (e.g., 64 bytes)
- Determine access pattern (sequential vs. random)

Calculate elements per cache line

- Elements per line = Cache line size / Element size
- For sequential access: First access = miss, rest = hits
- Hit rate = (Elements per line - 1) / Elements per line

Apply formula

- Hit rate = Number of hits / Total accesses
- Express as percentage or decimal

Cache Access Analysis Given code accessing an integer array sequentially:

```
#define N (10*1000*1000)
int arVL[N];
for (int i = 0; i < N; i++) {
    sum += arVL[i];
}
```

Analysis:

- 32-bit system: int = 4 bytes
- Cache line size: 64 bytes
- Elements per cache line: 64/4 = 16 integers
- Access pattern: Sequential
- For every 16 accesses: 1 miss + 15 hits
- Hit rate: 15/16 = 0.9375 = 93.75%

Average Memory Access Time

Multi-level cache formula

- $T_a = T_1 + (1 - h_1) \times [T_2 + (1 - h_2) \times [T_3 + (1 - h_3) \times T_{mem}]]$
- Where T_i = access time for level i, h_i = hit rate for level i

Step-by-step calculation

- Convert clock cycles to nanoseconds: cycles × cycle time
- Apply formula from innermost level outward
- L3 miss probability = $(1 - h_1) \times (1 - h_2) \times (1 - h_3)$

Memory Access Time Calculation 2 GHz processor (0.5 ns cycle), 90% hit rate on all levels:

L1 Cache	4 cycles = 2.0 ns
L2 Cache	10 cycles = 5.0 ns
L3 Cache	40 cycles = 20.0 ns
Main Memory	60 ns

Calculation:

- $T_a = 2.0 + 0.1 \times [5.0 + 0.1 \times [20.0 + 0.1 \times 60.0]]$
- $T_a = 2.0 + 0.1 \times [5.0 + 0.1 \times [20.0 + 6.0]]$
- $T_a = 2.0 + 0.1 \times [5.0 + 0.1 \times 26.0]$
- $T_a = 2.0 + 0.1 \times [5.0 + 2.6] = 2.0 + 0.1 \times 7.6$
- $T_a = 2.0 + 0.76 = 2.76$ ns

Build System Analysis

Make Dependency Analysis

Examine file timestamps

- List all source files (.c) and object files (.o)
- Compare timestamps: source newer than object = recompile needed
- Check dependency requirements in Makefile

Apply make rules

- Target depends on all listed prerequisites
- Suffix rule (.c.o:) applies to C source compilation
- Variable substitution: $@ = target name, $< = first prerequisite

Determine final executable name

- Follow target name and any modifications (e.g., $@.e)
- Check for explicit output naming in link command

Make Analysis Given Makefile and file listing:

```
# Makefile
CFL = -g
CMP = gcc $(CFL)
app: main1.o mythread.o scheduler.o queues.o mylist.o
    $(CMP) main1.o mythread.o scheduler.o queues.o mylist.o -o $@.e
.c.o:
    $(CMP) -c $<

# File listing (key timestamps)
# mylist.c    Feb 24 09:31  (source)
# mylist.o    Feb 24 09:21  (object)
# queues.c    Feb 24 09:25  (source)
# queues.o    Feb 24 09:21  (object)
```

Analysis:

- Files to recompile: mylist.c and queues.c (sources newer than objects)
- Final executable: app.e (from target äpp"+ ".eëxtension)
- Other .o files are up-to-date, no recompilation needed

Process Creation

Fork Analysis

Draw process tree

- Start with original process
- At each fork(), branch into parent and child
- Label each process with execution path

Trace execution paths

- Parent: fork() returns child PID (> 0)
- Child: fork() returns 0
- Follow if-else logic for each process

Count results

- Total processes = original + all children created
- Count specific outputs by tracing paths to printf statements

Process Creation Analysis Analyze the following code:

```
if (fork() > 0)
    fork();
else {
    fork();
    if (fork() > 0)
        printf("Hello World\n");
}
```

Process tree and analysis:

- Original process P forks → creates child C1
- Parent P (fork() > 0): executes second fork() → creates child C2
- Child C1 (fork() $== 0$): executes fork() → creates child C3
- Child C1: executes second fork() → creates child C4, then prints
- Child C3: executes second fork() → creates child C5, then prints

Results:

- Additional processes created: 5 (C1, C2, C3, C4, C5)
- "Hello World"printed: 2 times (by C1 and C3)

Memory Management

Buddy System Fragmentation

Determine buddy block sizes

- For each allocation request, find smallest power-of-2 block that fits
- Block size = $2^{\lceil \log_2(\text{request size}) \rceil}$

Calculate internal fragmentation

- Fragmentation per block = Block size - Requested size
- Total fragmentation = Sum of all individual fragmentations

Account for merging

- When blocks are freed, buddies may merge
- Track which blocks remain allocated vs. freed

Buddy System Analysis 8MB system allocates buffers: 62KB, 34KB, 9KB

Allocation analysis:

- 62KB → 64KB block (internal fragmentation: 2KB)
- 34KB → 64KB block (internal fragmentation: 30KB)
- 9KB → 16KB block (internal fragmentation: 7KB)
- Total internal fragmentation: $2 + 30 + 7 = 39$KB

The remaining memory is still available for allocation but in specific block sizes according to the buddy system structure.

Page Table Calculations

Extract address components

- Page Directory bits determine max number of page tables
- Page Number bits determine entries per page table
- Offset bits determine page size: $2^{\text{offset bits}}$

Calculate system limits

- Page size = $2^{\text{offset bits}}$ bytes
- Max page tables = $2^{\text{directory bits}}$
- Entries per page table = $2^{\text{page number bits}}$
- Max frames = Max page tables × Entries per table

Page Table Structure Analysis 32-bit addresses with 6-bit page directory, 8-bit page number, 10-bit offset:

Calculations:

- Page size: $2^{10} = 1024$ bytes = 1KB
- Page directory entries: $2^6 = 64$ page tables max
- Entries per page table: $2^8 = 256$ pages per table
- Each page table entry: 4 bytes (32-bit system)
- Page directory size: 64 entries × 4 bytes = 256 bytes
- Max system frames: 64 × 256 = 16,384 frames

Page Replacement

LRU Page Replacement

Track page access order

- Maintain timestamp or access order for each page in memory
- On page fault, identify least recently used page
- Replace LRU page with new page

Handle page faults

- Mark page fault when referenced page not in memory
- Load new page into frame of LRU page
- Update access timestamps for all affected pages

Apply demand paging

- First access to any page is compulsory miss
- Subsequent accesses depend on memory capacity and access pattern

LRU Page Replacement Trace Process references pages: 8,7,5,8,7,3,5,1,3,4,2,1,8,3,1,2
4 frames available, initially empty:

Reference	8	7	5	8	7	3	5	1	3	4	2	1	8	3	1	2
Frame 1	8	8	8	8	8	8	8	1	1	1	1	1	1	1	1	1
Frame 2		7	7	7	7	7	7	7	4	4	4	3	3	3	3	3
Frame 3			5	5	5	5	5	5	5	2	2	2	2	2	2	2
Frame 4						3	3	3	3	3	8	8	8	8	8	8
Page Fault	X	X	X			X		X		X	X					

Total page faults: 8

Key LRU decisions:

- At time 8: Page 1 replaces page 8 (LRU)
- At time 10: Page 4 replaces page 7 (LRU)
- At time 11: Page 2 replaces page 5 (LRU)
- At time 13: Page 8 replaces page 3 (LRU)

Real-Time Scheduling

Earliest Deadline First (EDF)

Calculate absolute deadlines

- For each task instance: Deadline = Arrival time + Period
- Track all active tasks (arrived but not completed)
- Update deadlines when new instances arrive

Schedule by earliest deadline

- At each scheduling point, select task with earliest absolute deadline
- Preempt running task if new arrival has earlier deadline
- Use period as tiebreaker for equal deadlines (shorter period wins)

Handle rescheduling points

- Reschedule when task completes or suspends
- Reschedule at fixed intervals if specified
- Reschedule when new task arrives (if preemptive)

EDF Scheduling Three periodic tasks with rescheduling every 4ms:

Task	Period	Execution Time
T1	4ms	1ms
T2	8ms	4ms
T3	12ms	3ms

Timeline analysis (first 24ms):

- t=0: T1(dl=4), T2(dl=8), T3(dl=12) arrive → Schedule T1
- t=1: T1 completes → Schedule T2
- t=4: T1(dl=8) arrives, T2 continues (dl=8, but T2 started first)
- t=5: T1 executes (same deadline, shorter period wins)
- t=6: T1 completes → Schedule T2
- t=8: T1(dl=12), T2(dl=16) arrive, T3 continues
- And so on...

The schedule ensures all deadlines are met with this task set.

Multi-Level Scheduling

Priority-Based Multi-Level Scheduling

Organize by priority levels

- Group processes by priority (higher number = higher priority)
- Maintain separate ready queue for each priority level
- Always schedule from highest priority non-empty queue

Apply Round Robin within priority

- Use time quantum for processes at same priority level
- Move process to end of same priority queue when quantum expires
- New arrivals join appropriate priority queue

Handle preemption

- Higher priority process always preempts lower priority
- Preemption occurs immediately when higher priority task arrives
- Preempted task returns to head of its priority queue

Multi-Level Scheduling with RR Five processes with time quantum = 1:

Process	Priority	Arrival	Execution
P1	0	0	4
P2	1	1	3
P3	1	2	2
P4	0	5	3
P5	0	7	2

Execution timeline:

- t=0-1: P1 (only process available)
- t=1-2: P2 (higher priority, preempts P1)
- t=2-3: P3 (same priority as P2, P2's quantum expired)
- t=3-4: P2 (continues RR at priority 1)
- t=4-5: P3 (completes)
- t=5-6: P2 (completes), then P4 starts
- t=6-7: P1 (RR at priority 0)
- t=7-8: P4 (continues RR), P5 arrives
- And so on with RR between P1, P4, P5 at priority 0...

System Concepts

System Call Identification

Analyze operation type

- I/O operations: Screen output, file access, network communication
- Resource management: Memory allocation, time delays
- Process control: Process creation, inter-process communication

Distinguish user vs kernel operations

- User mode: Arithmetic, variable manipulation, function calls
- Kernel mode: Hardware access, system resource management
- Mode switch required for system calls

System Call Requirements Which operations require system calls?

- √ Screen output (I/O operation)
- X Variable comparison (CPU operation)
- √ Sleep for 1 second (timer/scheduler)
- X Integer increment (CPU operation)
- X Float increment (CPU operation)

Rule of thumb: Operations requiring hardware access, OS services, or resource management need system calls. Pure computational operations do not.