Module 3 — Process Management

Introduction to Operating Systems

Table of Contents

Section	Topic	Page
3.1	Process Concept	3
3.2	Process States & Transitions	6
3.3	Process Control Block (PCB)	9
3.4	Process Creation & Termination	11
3.5	Multithreading & Multicore Systems	14
3.6	Process Scheduling Algorithms	19
3.7	Process Synchronization	28
3.8	Deadlock	35
3.9	Numerical Problems with Solutions	42
3.10	Quick Revision Notes	48
3.11	Practice Questions	51

Section 3.1: Process Concept

3.1.1 What is a Process?

A **process** is a program in execution. It is an active entity that includes: - **Text section**: The program code - **Current activity**: Program counter, processor

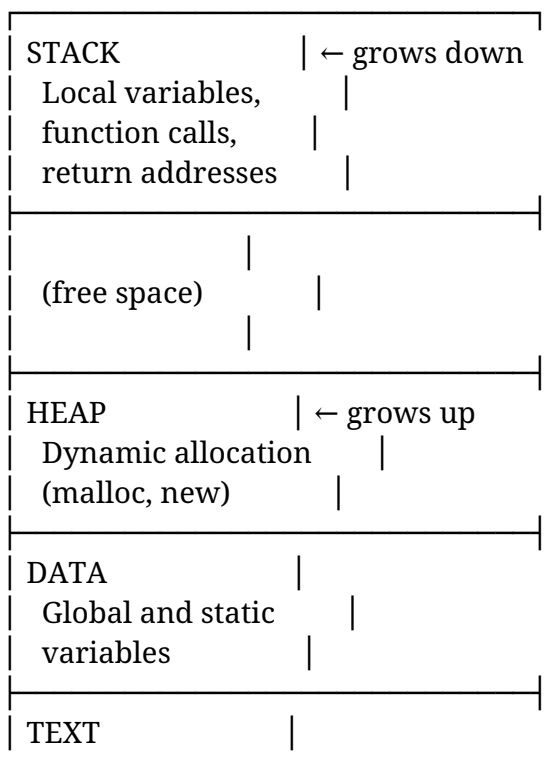
registers - **Stack**: Temporary data (function parameters, return addresses, local variables) - **Data section**: Global variables - **Heap**: Dynamically allocated memory (malloc, new)

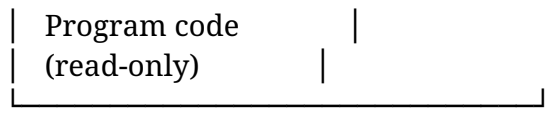
3.1.2 Program vs. Process

Feature	Program	Process
Nature	Passive entity	Active entity
Storage	Stored on disk	Loaded in memory
Lifetime	Static	Dynamic (created, runs, terminates)
Resources	No resources allocated	CPU, memory, I/O allocated
Analogy	Recipe	Cooking (following the recipe)

Note: Multiple processes can run the same program.

3.1.3 Process Memory Layout



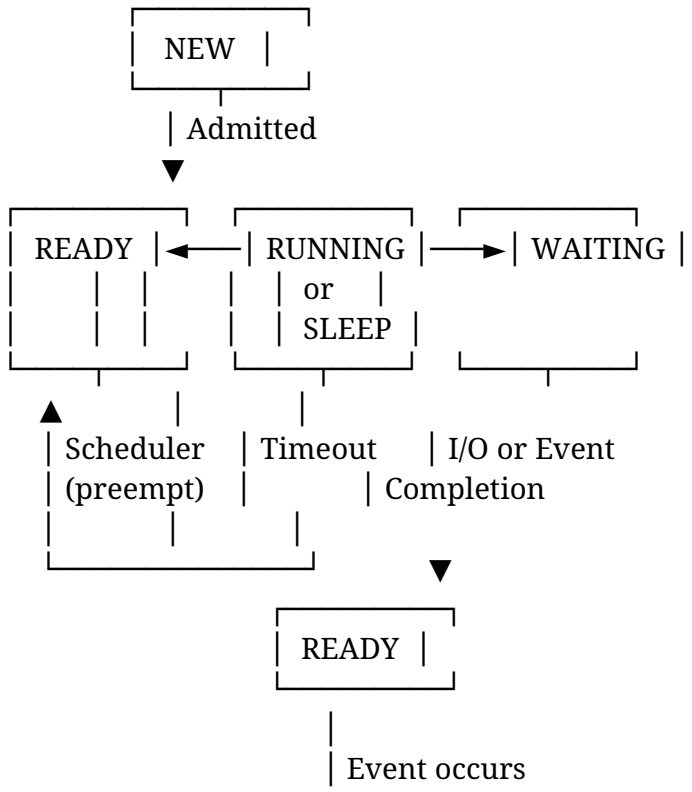


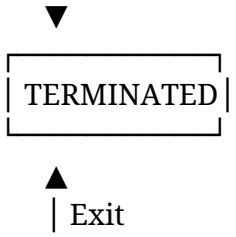
LOW MEMORY ADDRESS

Section	Description	Example
Text	Program instructions, read-only	printf("Hello");
Data	Global and static variables	int global = 10;
Heap	Dynamically allocated memory	malloc(), new
Stack	Function calls, local variables	int local = 5;

Section 3.2: Process States & Transitions

3.2.1 Process State Diagram





3.2.2 Process States

State	Description	Example
NEW	Process is being created	Program loaded into memory, PCB created
READY	Waiting to be assigned to CPU	In ready queue
RUNNING	Instructions being executed	Currently on CPU
WAITING	Waiting for I/O or event	`wait()`, `sleep()`, file read
TERMINATED	Process removed, PCB deallocated	`exit()` called

3.2.3 State Transitions

Transition	From → To	Cause	Example
Admitted	NEW → READY	OS accepts process	New program loaded
Dispatch	READY → RUNNING	Scheduler selects process	CPU assigned
Timeout	RUNNING → READY	Time quantum expired	Interrupt
Event Wait	RUNNING → WAITING	I/O request	`read()`, `wait()`
Event Completion	WAITING → READY	I/O completion	Data arrives from disk
Exit	RUNNING →	Process finished	`exit()` called

Generated by Kimi.ai

Transition	From → To	Cause	Example
	TERMINATED		

Key Points:

- Only one process can be in **RUNNING** state per CPU core
- Multiple processes can be in **READY** or **WAITING** states
- Process **cannot** go directly from WAITING to RUNNING (must go through READY)
- **Dispatcher** is the module that gives CPU control to the selected process

Common Mistakes:

- ❌ Thinking a process can go from WAITING directly to RUNNING
  ✅ Correct: WAITING → READY → RUNNING
- ❌ Thinking multiple processes can run simultaneously on single CPU
  ✅ Correct: Only one process runs; others appear to run due to fast switching

---

Section 3.3: Process Control Block (PCB)

3.3.1 Definition

The **Process Control Block (PCB)** is a data structure in the OS kernel that contains all information about a specific process. It is also called a **Task Control Block**.

3.3.2 PCB Contents

```
┌──────────────────────────────────────────┐
│        PROCESS CONTROL BLOCK             │
├──────────────────────────────────────────┤
│ Process ID (PID)       : 1234            │
│ Process State          : RUNNING         │
│ Program Counter        : 0x0045A3F2      │
│ CPU Registers          : R1=10, R2=20    │
│ Memory Management Info : Base=0x1000     │
│                          Limit=0x2000    │
│ CPU Scheduling Info    : Priority=5      │
│                          Quantum=10ms    │
│ Accounting Info        : CPU_time=245ms  │
```

```
│ I/O Status          : open_files=[f1.txt]│
│ Parent Process ID (PPID): 5678           │
└──────────────────────────────────────────┘
```

Component	Description	Stored Information
Process ID (PID)	Unique identifier	Integer value (1 to max)
Process State	Current state	new, ready, running, waiting, terminated
Program Counter	Address of next instruction	Memory address
CPU Registers	Saved when interrupted	General purpose, index, stack pointers
Memory Management	Memory allocation info	Base and limit registers, page tables
Scheduling Info	Used by CPU scheduler	Priority, queue pointers, quantum
Accounting Info	Resource usage tracking	CPU time used, I/O operations
I/O Status	Open files and devices	File descriptors, device handles
Parent PID (PPID)	Relationship to creator	PPID

3.3.3 Context Switch

A **context switch** is the process of saving the state of one process and loading the state of another.

Steps in Context Switch: 1. Save CPU registers of current process into its PCB 2. Save program counter of current process into its PCB 3. Update process state (RUNNING → READY/WAITING) 4. Load next process's PCB 5. Restore its CPU registers and program counter 6. Start executing new process

Context Switch Overhead: - Pure overhead (no useful work done) - Time depends on hardware support - Typical time: few microseconds to milliseconds

---

Generated by Kimi.ai

Section 3.4: Process Creation & Termination

3.4.1 Process Creation Events

Event	Example
System initialization	Boot process creates init/systemd
User request	Typing command in terminal
Process spawns another	Application launching helper process
Batch job submission	Printing a document

3.4.2 UNIX/Linux Process Creation: fork() and exec()

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();  // Create child process

    if (pid < 0) {
        // Fork failed
        printf("Fork failed!\n");
        return 1;
    }
    else if (pid == 0) {
        // CHILD PROCESS
        printf("Child: My PID = %d\n", getpid());
        printf("Child: Parent PID = %d\n", getppid());

        // Replace child with new program
        execl("/bin/ls", "ls", "-l", NULL);

        // If execl returns, error occurred
        printf("Child: execl failed!\n");
    }
    else {
        // PARENT PROCESS
        printf("Parent: My PID = %d\n", getpid());
        printf("Parent: Child PID = %d\n", pid);
```

Generated by Kimi.ai

```
    // Wait for child to finish
    wait(NULL);
    printf("Parent: Child finished\n");
  }
  return 0;
}
```

Sample Output:

```
Parent: My PID = 1234
Parent: Child PID = 1235
Child: My PID = 1235
Child: Parent PID = 1234
[Output of 'ls -l' command]
Parent: Child finished
```

3.4.3 fork() Behavior

- Parent and child are **identical** after fork (same code, data, stack)
- They execute **independently** (order not guaranteed)
- **Return value distinguishes** parent (child's PID) vs child (0)
- **Child inherits**: open files, environment, signal handlers
- **Child does NOT inherit**: pending signals, timers, memory locks

3.4.4 exec() Family

```
// Different forms of exec()
execl("/bin/ls", "ls", "-l", NULL);    // List arguments
execlp("ls", "ls", "-l", NULL);        // Uses PATH
execv("/bin/ls", args);                // Array of arguments
execvp("ls", args);                    // PATH + array
```

What happens during exec(): - Current program's code is replaced - New program loads into same memory space - **PID remains the SAME** - Open files remain open (usually) - Process state resets (new PC, stack, heap)

3.4.5 Process Hierarchy (UNIX)

```
init (PID = 1) / systemd (PID = 2)
  │
  ├── login (PID=100)
  │     └── bash (PID=200)
  │           ├── gcc (PID=300)
  │           ├── ls  (PID=301)
```

Generated by Kimi.ai

```
│           └── firefox (PID=302)
│
├── sshd (PID=101)
└── cron (PID=102)
```

3.4.6 Process Termination

Normal Termination:

```
exit(status);   // Normal termination with status
return;         // From main() - equivalent to exit()
```

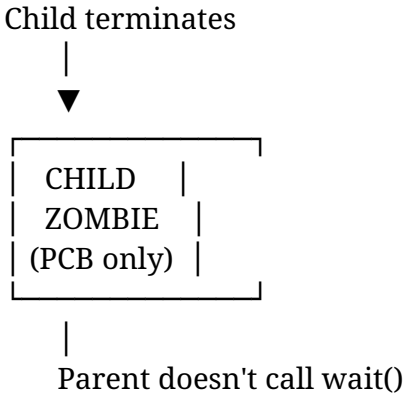
Abnormal Termination:

```
kill -9 PID
```

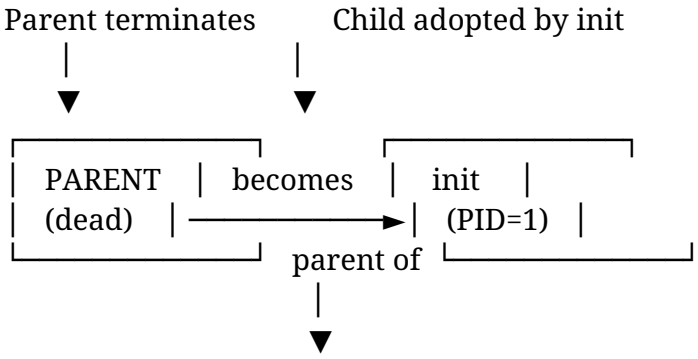
Termination Steps: 1. Close all open files 2. Release allocated memory 3. Deallocate PCB 4. Notify parent (send SIGCHLD signal)

3.4.7 Zombie and Orphan Processes

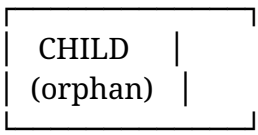
Zombie Process: - Process that has terminated but parent hasn't called `wait()` - PCB remains in memory - Shown as `<defunct>` in `ps`



Orphan Process: - Parent terminates before child - Child is adopted by `init` (PID = 1)



Generated by Kimi.ai

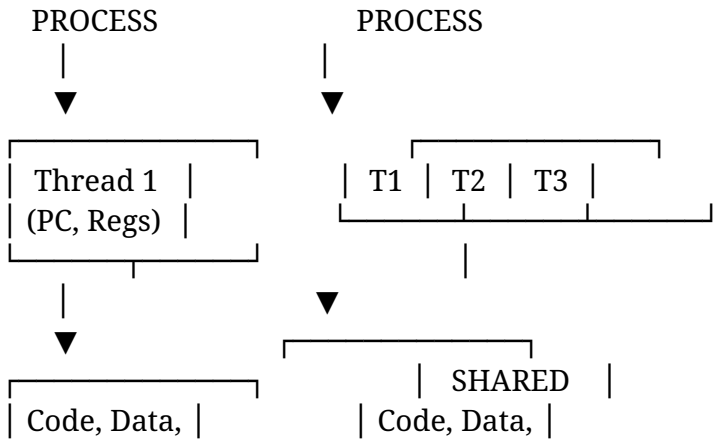


Section 3.5: Multithreading & Multicore Systems

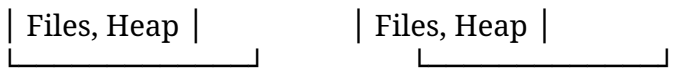
3.5.1 Process vs. Thread

Feature	Process	Thread
Definition	Program in execution	Lightweight unit of execution within process
Memory	Separate memory space	Shares memory with other threads
Creation	Slow (heavyweight)	Fast (lightweight)
Context Switch	Slow (save entire PCB)	Fast (save only registers)
Communication	Indirect (IPC needed)	Direct (shared memory)
Isolation	High	Low
Crash effect	Only that process dies	Can crash entire process

3.5.2 Single-threaded vs Multi-threaded Process



Generated by Kimi.ai



3.5.3 Thread Components

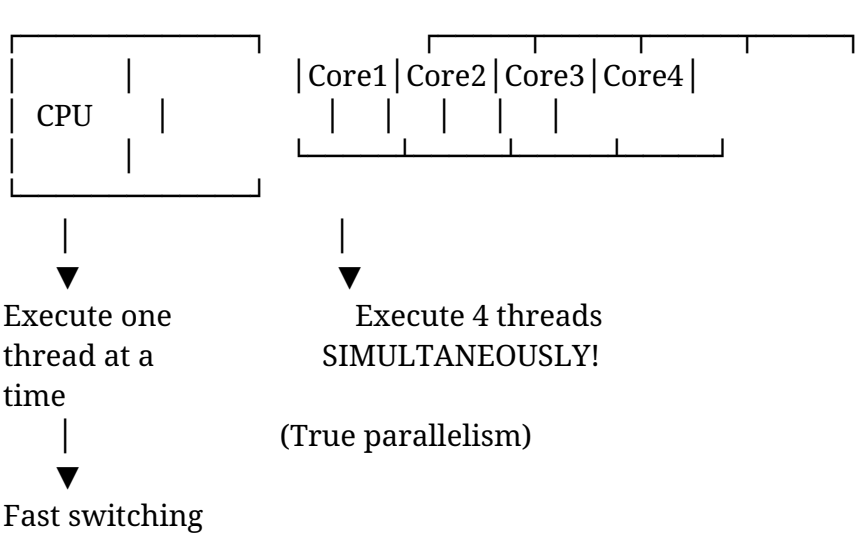
Each thread has: - Thread ID - Program counter (PC) - Register set - Stack

Shared by all threads in a process: - Code section - Data section - Heap - Open files - Signal handlers

3.5.4 Benefits of Multithreading

Benefit	Explanation	Example
Responsiveness	One thread blocks, others continue	Web browser: one thread loads image, other keeps UI responsive
Resource Sharing	Threads share memory efficiently	Multiple tasks accessing same data
Economy	Cheaper than processes	Thread creation: 10-100x faster than process
Scalability	Utilize multiple CPU cores	Different threads on different cores

3.5.5 Multicore Systems



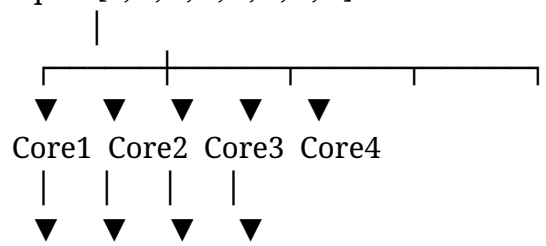
Generated by Kimi.ai

3.5.6 Multicore vs Multiprocessor

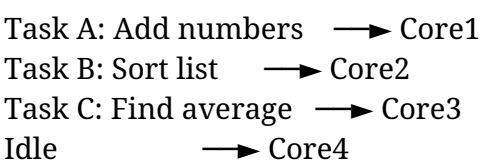
Aspect	Multicore	Multiprocessor
Definition	Multiple cores on single chip	Multiple CPU chips
Communication	Shared cache (faster)	Through motherboard (slower)
Cost	Lower	Higher
Power consumption	Lower	Higher
Example	Intel i7 (8 cores)	Server with 2 CPU sockets

3.5.7 Types of Parallelism

Data Parallelism: Distribute data across cores



Task Parallelism: Distribute tasks across cores



3.5.8 Amdahl's Law

$$\text{Speedup} = 1 / [(1 - P) + (P / N)]$$

Where: - **P** = Parallel portion of program (0 to 1) - **N** = Number of cores - **(1 - P)** = Serial portion (cannot be parallelized)

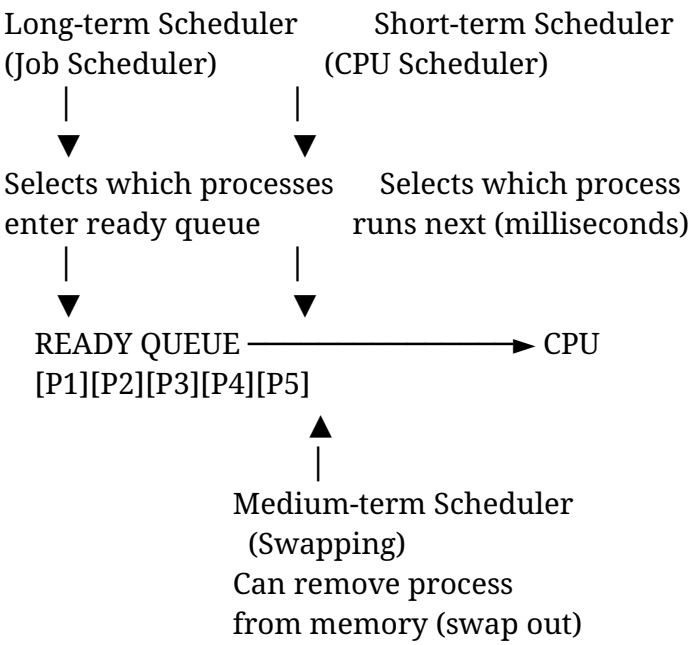
Generated by Kimi.ai

Section 3.6: Process Scheduling Algorithms

3.6.1 Scheduling Metrics

Metric	Definition	Formula	Goal
CPU Utilization	Percentage of time CPU is busy	(Busy time / Total time) × 100	Maximize
Throughput	Number of processes completed per unit time	Processes / Time	Maximize
Turnaround Time	Total time from arrival to completion	Completion - Arrival	Minimize
Waiting Time	Total time spent in ready queue	Turnaround - Burst time	Minimize
Response Time	Time until first response	First CPU time - Arrival	Minimize

3.6.2 Types of Schedulers



Generated by Kimi.ai

Scheduler	Frequency	Decision	Purpose
Long-term	Low (seconds/minutes)	Which process to admit	Control degree of multiprogramming
Short-term	High (milliseconds)	Which process to run next	Maximize CPU utilization
Medium-term	Medium	Swap in/out processes	Reduce memory pressure

Algorithm 1: First-Come, First-Served (FCFS)

Non-preemptive — Once CPU allocated, process runs until completion.

Example:

Process	Burst Time	Arrival Time
P1	24	0
P2	3	0
P3	3	0

Gantt Chart:

```
|P1      |P2  |P3  |
0        24   27   30
```

- Completion times: P1 = 24, P2 = 27, P3 = 30
- Turnaround times: P1 = 24, P2 = 27, P3 = 30
- Average Turnaround = (24 + 27 + 30) / 3 = **27**
- Average Waiting = (0 + 24 + 27) / 3 = **17**

Problem: Convoy effect — short process waits behind long process.

Algorithm 2: Shortest Job First (SJF)

Non-preemptive — Process with smallest burst time runs first.

Example:

Process	Burst Time	Arrival Time
P1	6	0

Generated by Kimi.ai

Process	Burst Time	Arrival Time
P2	8	0
P3	7	0
P4	3	0

Order by burst time: P4(3), P1(6), P3(7), P2(8)

Gantt Chart:

```
|P4 |P1    |P3    |P2    |
0   3    9     16     24
```

- Average Turnaround = (3 + 9 + 16 + 24) / 4 = **13**
- Average Waiting = (0 + 3 + 9 + 16) / 4 = **7**

Problem: Cannot predict future burst times.

Solution: Use exponential averaging (predict based on past behavior).

Algorithm 3: Shortest Remaining Time First (SRTF)

Preemptive version of SJF.

Example:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Execution: - At time 0: P1 runs (remaining 8) - At time 1: P2 arrives (burst 4) < P1 remaining (7) → **Preempt P1** - At time 1: P2 runs (remaining 4) - At time 2: P3 arrives (burst 9) > P2 remaining (3) → Continue P2 - At time 3: P4 arrives (burst 5) > P2 remaining (2) → Continue P2 - At time 5: P2 completes - At time 5: Compare P1(7), P3(9), P4(5) → Run P4 (smallest)

Gantt Chart:

```
|P1|P2|P2|P2|P4|P4|P4|P4|P4|P1|P1|...|P3|
0  1  2  3  4  5  6  7  8  9  10 11   21
```

Generated by Kimi.ai

- Wait times: P1 = 9, P2 = 0, P4 = 2, P3 = 9
- Average Waiting = (9 + 0 + 2 + 9) / 4 = **5**

Algorithm 4: Priority Scheduling

Each process has a priority (integer). **Lower number = higher priority**.

Example:

Process	Burst Time	Priority
P1	10	3
P2	1	1 (highest)
P3	2	4
P4	1	5
P5	5	2

Order: P2(1), P5(2), P1(3), P3(4), P4(5)

Gantt Chart:

```
|P2|P5    |P1        |P3  |P4  |
0  1     6          16   18   19
```

- Average Waiting = (1 + 0 + 6 + 16 + 18) / 5 = **8.2**

Problem: Starvation — low priority processes may never execute.

Solution: Aging — increase priority of waiting processes over time.

Algorithm 5: Round Robin (RR)

Each process gets a small unit of CPU time (**time quantum**), usually 10-100 ms.

Example (Quantum = 4ms):

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart:

Generated by Kimi.ai

```
|P1   |P2  |P3  |P1   |P1   |P1   |P1   |
0     4    7    10    14    18    22    26
```

- Completion times: P2 = 7, P3 = 10, P1 = 26
- Turnaround times: P2 = 7, P3 = 10, P1 = 26
- Average Turnaround = (7 + 10 + 26) / 3 = **14.33**
- Average Waiting = (4 + 7 + 16) / 3 = **9**

Quantum Size Effect:

Quantum Size	Effect
Very large (∞)	Degenerates to FCFS
Very small (1ms)	Too many context switches (high overhead)
Optimal (10-100ms)	Balanced — 80% of processes should finish within quantum

Algorithm 6: Multilevel Queue Scheduling

```
┌─────────────────────────────────────────┐
│          READY QUEUE                    │
├─────────────────────────────────────────┤
│  FOREGROUND (RR)                        │
│  [P1] [P2] [P3]  ← Highest priority     │
│                                         │
│  BACKGROUND (FCFS)                      │
│  [P4] [P5] [P6]  ← Lower priority       │
└─────────────────────────────────────────┘
```

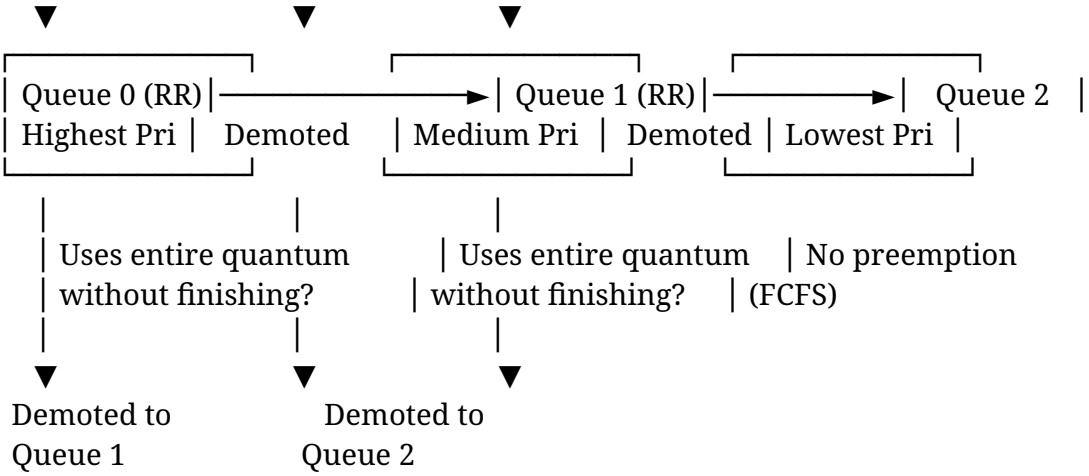
Characteristics: - Processes permanently assigned to one queue - Each queue has its own scheduling algorithm - Priority between queues (e.g., foreground always before background)

Algorithm 7: Multilevel Feedback Queue (MLFQ)

Most sophisticated — Processes can move between queues.

```
Quantum = 8ms          Quantum = 16ms          FCFS
    │                      │                    │
```

Generated by Kimi.ai



Rules: 1. New process enters Queue 0 2. If process uses entire quantum → demoted to next queue 3. If process gives up CPU (I/O) → stays in same queue 4. Schedule all processes in higher queue before lower 5. **Aging**: Move to higher queue after waiting too long (prevents starvation)

Section 3.7: Process Synchronization

3.7.1 The Critical Section Problem

Critical Section: Code segment that accesses shared resources.

```
// Shared variable
int counter = 0;

// Producer code
while (1) {
   /* produce item */
   counter++;  // CRITICAL SECTION!
   /* store item */
}

// Consumer code
while (1) {
   /* remove item */
   counter--;  // CRITICAL SECTION!
   /* consume item */
}
```

Problem: counter++ is **not atomic**! It compiles to:

```
LOAD counter to register
INCREMENT register
STORE register to counter
```

3.7.2 Requirements for Solution

Requirement	Meaning
Mutual Exclusion	Only ONE process can be in critical section at a time
Progress	If no process is in CS, only waiting processes can enter (no outside interference)
Bounded Waiting	No infinite waiting — limit on how many times others can enter first

3.7.3 Peterson's Solution (2 processes)

```
int flag[2];   // flag[i] = true if process i wants to enter
int turn;      // whose turn to enter

// Process 0
do {
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1);  // Wait

   // CRITICAL SECTION

   flag[0] = false;
   // REMAINDER SECTION
} while (1);

// Process 1
do {
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0);  // Wait

   // CRITICAL SECTION
```

Generated by Kimi.ai

```
    flag[1] = false;
    // REMAINDER SECTION
} while (1);
```

3.7.4 Semaphores

Definition:

```
typedef struct {
    int value;
    struct process *queue;
} semaphore;
```

wait() — also called P() or down():

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        // Block process and add to queue
        block();
    }
}
```

signal() — also called V() or up():

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        // Wake up a blocked process
        wakeup();
    }
}
```

Binary Semaphore (Mutex):

```
semaphore mutex = 1;  // Initialize to 1

do {
    wait(&mutex);     // Entry section

    // CRITICAL SECTION

    signal(&mutex);   // Exit section
    // REMAINDER SECTION
} while (1);
```

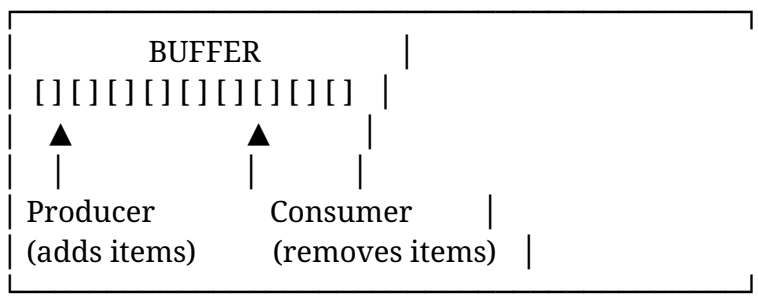
Counting Semaphore (Producer-Consumer):

Generated by Kimi.ai

```
semaphore empty = N;  // N empty slots
semaphore full = 0;    // 0 full slots
semaphore mutex = 1;   // Mutual exclusion

// PRODUCER
do {
   produce item;
   wait(&empty);      // Wait if no empty slots
   wait(&mutex);      // Lock for critical section

   // add item to buffer;

   signal(&mutex);    // Unlock
   signal(&full);     // Increment full count
} while (1);

// CONSUMER
do {
   wait(&full);       // Wait if nothing to consume
   wait(&mutex);      // Lock for critical section

   // remove item from buffer;

   signal(&mutex);    // Unlock
   signal(&empty);    // Increment empty count
   consume item;
} while (1);
```

3.7.5 Classic Synchronization Problems

1. Bounded Buffer (Producer-Consumer)

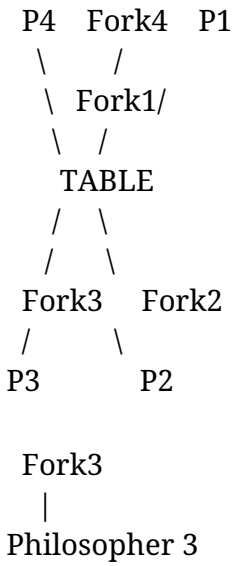


Solution using semaphores:

Generated by Kimi.ai

```
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

2. Dining Philosophers



Problem: 5 philosophers, 5 forks, need 2 forks to eat.

Risk: Deadlock if all pick up left fork simultaneously!

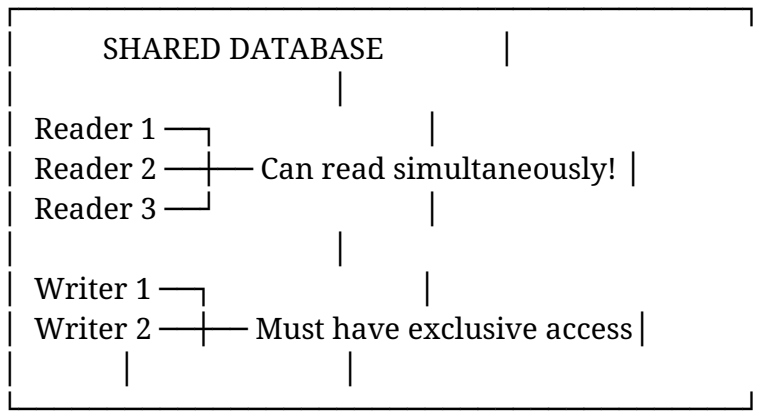
Solution: Pick up forks in different order

```
semaphore fork[5] = {1, 1, 1, 1, 1};

void philosopher(int i) {
  while (1) {
    think();

    // Solution: Last philosopher picks up right first
    if (i == 4) {
      wait(&fork[(i+1) % 5]);  // Pick up right
      wait(&fork[i]);          // Pick up left
    } else {
      wait(&fork[i]);          // Pick up left
      wait(&fork[(i+1) % 5]);  // Pick up right
    }

    eat();

    signal(&fork[i]);          // Put down left
```

Generated by Kimi.ai

```
    signal(&fork[(i+1) % 5]);    // Put down right
  }
}
```

3. Readers-Writers Problem



First Readers-Writers Problem — No reader waits unless writer has lock.

Writers may starve.

```
semaphore mutex = 1;  // For reader count
semaphore wrt = 1;    // For writer
int read_count = 0;

// READER
wait(&mutex);
read_count++;
if (read_count == 1)
  wait(&wrt);      // First reader locks out writers
signal(&mutex);

// Reading...

wait(&mutex);
read_count--;
if (read_count == 0)
  signal(&wrt);    // Last reader releases writers
signal(&mutex);

// WRITER
wait(&wrt);
// Writing...
signal(&wrt);
```

Generated by Kimi.ai

Section 3.8: Deadlock

3.8.1 Definition

Deadlock: Two or more processes are waiting indefinitely for resources held by each other.

3.8.2 Necessary Conditions (ALL must hold)

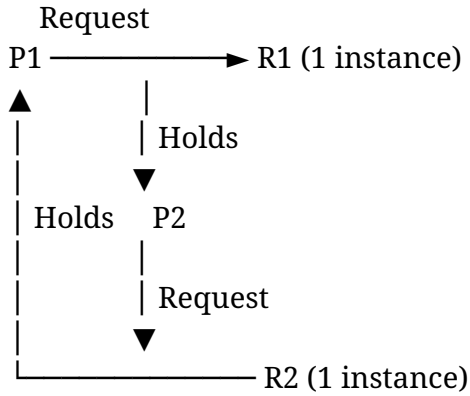
Condition	Description	Example
1. Mutual Exclusion	Resources cannot be shared	Printer can only be used by one process
2. Hold and Wait	Process holds resources while waiting for others	Holds Resource A, waits for Resource B
3. No Preemption	Resources cannot be forcibly taken	Can’t take printer from a process
4. Circular Wait	Cycle of processes waiting	P1 → P2 → P3 → P1

3.8.3 Resource Allocation Graph

Nodes: - ○ = Process - □ = Resource Type - ● = Instance of resource

Edges: - Process → Resource (Request edge) - Resource → Process (Assignment edge)

Deadlock Example:



Rules: - No cycle → No deadlock - Cycle with single-instance resources → Deadlock exists - Cycle with multi-instance resources → Possible deadlock

Generated by Kimi.ai

3.8.4 Deadlock Handling Strategies

Strategy 1: Prevention (Break one condition)

Condition to Break	Method	Disadvantages
Mutual Exclusion	Make resources sharable	Not possible for some resources (printer)
Hold and Wait	Request all resources at once	Low resource utilization
No Preemption	Force release resources	Difficult to implement
Circular Wait	Impose total ordering of resources	Need to define global order

Example of ordering:

```
// Order resources: Tape drive (1), Printer (2), Disk (3)
// Process must request in increasing order

// Good: Request in order
wait(tape);    // 1
wait(printer); // 2

// Bad: Could cause deadlock
wait(printer); // 2
wait(tape);    // 1 (would be blocked)
```

Strategy 2: Avoidance (Banker's Algorithm)

Banker's Algorithm (Dijkstra, 1965)

```
int Available[m];   // Available instances of each resource
int Max[n][m];      // Maximum demand of each process
int Allocation[n][m];// Currently allocated to each process
int Need[n][m];     // Remaining need = Max - Allocation
```

Safety Algorithm:

```
bool isSafe() {
   int Work[m] = Available;
   bool Finish[n] = {false};

   while (true) {
      bool found = false;
```

Generated by Kimi.ai

```
      for (int i = 0; i < n; i++) {
        if (!Finish[i] && Need[i] <= Work) {
          Work = Work + Allocation[i];
          Finish[i] = true;
          found = true;
        }
      }
      if (!found) break;
    }
    return all(Finish == true);
}
```

Example:

System state: Resources: A(10), B(5), C(7)

Process	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Available: 3 3 2

Safety Check: 1. Need[P1] = (1,2,2) ≤ Available(3,3,2) → **YES**

New Available = (3,3,2) + (2,0,0) = (5,3,2) → Finish P1 2. Need[P3] = (0,1,1) ≤ Available(5,3,2) → **YES**

New Available = (5,3,2) + (2,1,1) = (7,4,3) → Finish P3 3. Need[P4] = (4,3,1) ≤ Available(7,4,3) → **YES**

New Available = (7,4,3) + (0,0,2) = (7,4,5) → Finish P4 4. Need[P0] = (7,4,3) ≤ Available(7,4,5) → **YES**

New Available = (7,4,5) + (0,1,0) = (7,5,5) → Finish P0 5. Need[P2] = (6,0,0) ≤ Available(7,5,5) → **YES**

Generated by Kimi.ai

Safe sequence: <P1, P3, P4, P0, P2>

System is SAFE!

Strategy 3: Detection & Recovery

Detection: Periodically check for deadlock using wait-for graph.

Recovery Methods:

Method	Description	Pros/Cons
Process Termination	Abort deadlocked processes	Simple but may lose work
— Abort all	Kill all deadlocked processes	Drastic
— Abort one by one	Kill until deadlock broken	Better but slower
Resource Preemption	Take resources from processes	Complex
— Select victim	Choose process to preempt	Need selection criteria
— Rollback	Return to safe state	Need checkpointing
— Starvation	Same victim always chosen	Need to ensure fairness

Strategy 4: Ostrich Algorithm

“Ignore the problem” — Most modern OS use this! - Assumption: Deadlocks are rare - Cost of prevention/avoidance is high - User can reboot if deadlock occurs

3.8.5 Deadlock Comparison Table

Strategy	Approach	Overhead	Examples
Prevention	Break one condition	High	Embedded systems
Avoidance	Banker’s algorithm	High	Mainframes
Detection	Check	Medium	Some Unix

Generated by Kimi.ai

Strategy	Approach	Overhead	Examples
	periodically		versions
Ostrich	Ignore	None	Windows, Linux

Section 3.9: Numerical Problems with Solutions

Problem 1: FCFS

Process	Burst Time	Arrival Time
P1	5	0
P2	3	1
P3	4	2
P4	2	3

Gantt Chart:

```
|P1   |P2  |P3   |P4 |
0     5    8     12  14
```

- Completion Times: P1=5, P2=8, P3=12, P4=14
- Turnaround: P1=5-0=5, P2=8-1=7, P3=12-2=10, P4=14-3=11
- **Avg TAT = (5+7+10+11)/4 = 33/4 = 8.25**
- Waiting: P1=5-5=0, P2=7-3=4, P3=10-4=6, P4=11-2=9
- **Avg WT = (0+4+6+9)/4 = 19/4 = 4.75**

Problem 2: SJF (Non-preemptive)

Same processes as Problem 1.

At time 0: Available: P1(5) → P1 runs

At time 5: Available: P2(3), P3(4), P4(2) → Order: P4, P2, P3

Gantt Chart:

```
|P1   |P4 |P2  |P3   |
0     5   7    10    14
```

- Completion: P1=5, P4=7, P2=10, P3=14
- TAT: P1=5, P2=9, P3=12, P4=4

- **Avg TAT = (5+9+12+4)/4 = 30/4 = 7.5**
- WT: P1=0, P2=6, P3=8, P4=2
- **Avg WT = (0+6+8+2)/4 = 16/4 = 4**

Problem 3: Round Robin (Quantum = 3)

Process	Burst Time
P1	10
P2	5
P3	8

Execution Trace:

Time	Process	Burst Used	Remaining
0-3	P1	3	7
3-6	P2	3	2
6-9	P3	3	5
9-12	P1	3	4
12-14	P2	2	0 (completes)
14-17	P3	3	2
17-19	P1	2	2
19-21	P3	2	0 (completes)
21-23	P1	2	0 (completes)

Gantt Chart:

```
|P1 |P2 |P3 |P1 |P2 |P3 |P1 |P3 |P1 |
0  3  6  9  12  14  17  19  21  23
```

- Completion: P1=23, P2=14, P3=21
- Turnaround: P1=23, P2=14, P3=21
- Waiting: P1=23-10=13, P2=14-5=9, P3=21-8=13
- **Avg WT = (13+9+13)/3 = 35/3 ≈ 11.67**

Generated by Kimi.ai

Problem 4: Page Replacement (For reference)

Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 3 frames.

FIFO:

```
Ref: 7 0 1 2 0 3 0 4 2 3 0 3 2
F1:  7 7 7 2 2 2 2 4 4 4 0 0 0
F2:  - 0 0 0 0 3 3 3 2 2 2 2 2
F3:  - - 1 1 1 1 0 0 0 3 3 3 3
PF:  Y Y Y Y N Y Y Y Y Y Y N N
```

Page Faults = 10

LRU:

```
Ref: 7 0 1 2 0 3 0 4 2 3 0 3 2
F1:  7 7 7 2 2 2 2 2 2 2 2 2 2
F2:  - 0 0 0 0 0 0 4 4 4 0 0 0
F3:  - - 1 1 1 3 3 3 3 3 3 3 3
PF:  Y Y Y Y N Y N Y N N Y N N
```

Page Faults = 9

Optimal:

```
Ref: 7 0 1 2 0 3 0 4 2 3 0 3 2
F1:  7 7 7 2 2 2 2 2 2 2 2 2 2
F2:  - 0 0 0 0 0 0 0 0 0 0 0 0
F3:  - - 1 1 1 3 3 4 4 3 3 3 3
PF:  Y Y Y Y N Y N Y N N Y N N
```

Page Faults = 7

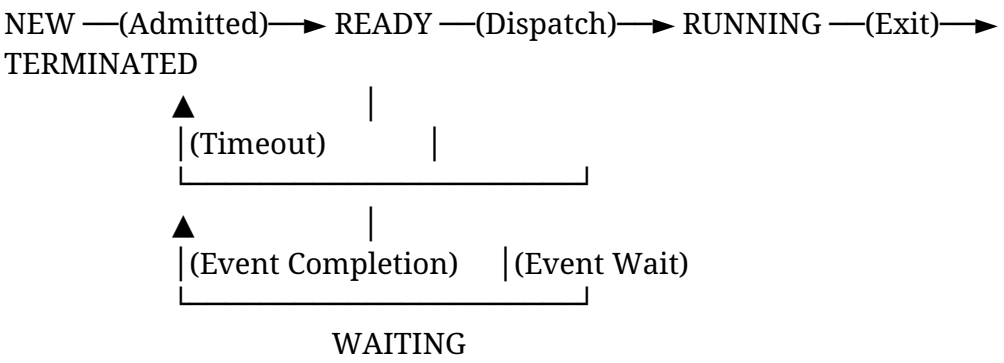
Section 3.10: Quick Revision Notes

Formulas

- **Turnaround Time** = Completion Time - Arrival Time
- **Waiting Time** = Turnaround Time - Burst Time
- **Response Time** = First CPU Time - Arrival Time
- **CPU Utilization** = (Total Time - Idle Time) / Total Time
- **Throughput** = Number of Processes / Total Time

Generated by Kimi.ai

Process State Diagram (Summary)



Deadlock Conditions (4 necessary)

1. **M**utual Exclusion
2. **H**old and Wait
3. **N**o Preemption
4. **C**ircular Wait

Mnemonic: “Most Houses Need Care”

Scheduling Algorithm Summary

Algorithm	Preemptive?	Best For	Problem
FCFS	No	Batch	Convoy effect
SJF	No	Minimum avg wait	Need future knowledge
SRTF	Yes	Minimum avg wait	Starvation
Priority	Both	Priority tasks	Starvation
RR	Yes	Interactive	Quantum selection

Semaphore Operations (Remember)

- wait() = P() = down() = decrement and block if negative
- signal() = V() = up() = increment and wakeup if ≤ 0

Generated by Kimi.ai

Linux Commands

Command	Purpose
ps	List processes
ps -ef	List all processes
top	Dynamic process view
kill -9 PID	Kill process
nice -n 10	Run with priority
&	Run in background
jobs	List background jobs
fg	Bring to foreground

Section 3.11: Practice Questions

Short Answer Questions (2-3 marks)

1. Define process. Differentiate between process and program.
2. Draw and explain process state diagram.
3. What is PCB? List any 5 components of PCB.
4. What is context switch? Why is it considered overhead?
5. Differentiate between preemptive and non-preemptive scheduling.
6. What are the conditions for deadlock?
7. What is starvation? How can it be prevented?
8. Explain the difference between fork() and exec() system calls.
9. What is a zombie process? What is an orphan process?
10. Define critical section. List the three requirements for its solution.

Medium Answer Questions (5-7 marks)

11. Explain the Producer-Consumer problem and provide a semaphore-based solution.
12. Compare and contrast SJF and Round Robin scheduling algorithms with examples.

13. Explain Banker’s algorithm for deadlock avoidance with an example.
14. What is multithreading? Explain its benefits over single-threaded processes.
15. Explain the Dining Philosophers problem and provide a deadlock-free solution.

Long Answer Questions (10-15 marks)

16. Five processes arrive at time 0 with burst times: P1=10, P2=29, P3=3, P4=7, P5=12. Calculate average waiting time and turnaround time for FCFS, SJF, and Round Robin (quantum=5).
17. Consider the following snapshot of a system with 4 resources (A, B, C, D) and 5 processes:

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0
P1	1	0	0	0	1	7	5	0				
P2	1	3	5	4	2	3	5	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

Is the system in a safe state? If yes, find the safe sequence.

18. Explain the Multilevel Feedback Queue scheduling algorithm with a diagram. How does it solve the starvation problem?
19. Write a detailed note on classical synchronization problems (Bounded Buffer, Readers-Writers, Dining Philosophers).
20. Explain deadlock detection and recovery strategies with examples.

Multiple Choice Questions (Quick Check)

1. A process is in _____ state when it is waiting for I/O.

   _

Generated by Kimi.ai

a) Ready

–

b) Running

–

c) **Waiting** ✓

–

d) Terminated

2. Which scheduling algorithm has the minimum average waiting time?

–

a) FCFS

–

b) **SJF** ✓

–

c) Round Robin

–

d) Priority

3. The condition where low priority processes never run is called:

–

a) Deadlock

–

b) **Starvation** ✓

–

c) Aging

–

d) Thrashing

4. Which is NOT a necessary condition for deadlock?

–

a) Mutual Exclusion

–

Generated by Kimi.ai

b) Hold and Wait

–

c) No Preemption

–

d) **Priority** ✓

5. The wait() operation on a semaphore:

–

a) Increments value

–

b) **Decrements value** ✓

–

c) Does nothing

–

d) Creates process

Final Summary Table

Topic	Key Takeaway
Process	Program in execution + resources
PCB	Contains ALL info about a process
Context Switch	Pure overhead — save/load PCB
SJF	Optimal for average waiting time
RR	Fair, good for interactive systems
Semaphore	Integer with atomic wait/signal
Deadlock	4 necessary conditions: M, H, N, C
Multithreading	Shares memory, faster than processes

End of Module 3 — Study Material

Generated by Kimi.ai