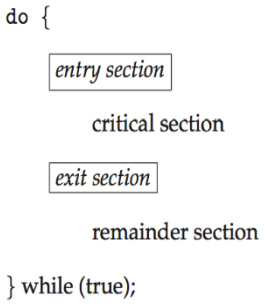
Chapter 5

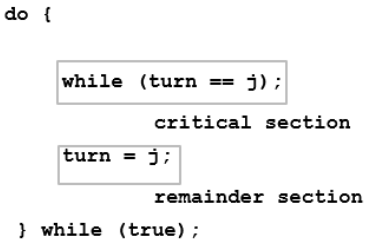
Critical Section Problem

Definition: The Critical Section Problem is when multiple processes need to use the same resource, and we need to make sure only one process uses it at a time to avoid errors.

- When one process in critical section, no other may be in its critical section.
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**



General Structure of Process P_i



Algorithm

Solution to Critical-Section Problem

- **Mutual Exclusion** – If one process is using the critical section, no other process can enter it. This prevents conflicts when multiple processes try to access shared resources.
- **Progress** – If no process is in the critical section, but some want to enter, the system must decide fairly who goes next. It cannot keep delaying the decision forever.
- **Bounded Waiting** – Once a process requests to enter the critical section, there must be a limit on how many other processes can go ahead of it. This ensures no process is stuck waiting forever.

  Assumptions
  - Each process runs at some speed greater than zero.
  - The system does not assume which process is faster or slower.

Preemptive vs. Non-Preemptive Scheduling

Feature	Preemptive Scheduling	Non-Preemptive Scheduling
Definition	Allows the operating system to **interrupt** a running process and assign the CPU to another process.	Once a process starts, it runs until it finishes or waits.
Interruptions	A process can be paused and resumed later.	No process is interrupted once it gets the CPU.
Example	Round Robin, Shortest Remaining Time First (SRTF)	First Come First Serve (FCFS), Shortest Job First (SJF)
Response Time	Faster, as high-priority tasks can interrupt.	Slower, since a process must finish before the next one runs.
Use Case	Used in multitasking OS (Windows, Linux).	Used in simple systems like batch processing.

Peterson's Solution

How It Works:

Peterson's Solution uses:

1. **A turn variable** – Indicates whose turn it is to enter the critical section.
2. **A flag array** – Keeps track of whether a process wants to enter its critical section.

Algorithm:

For two processes **P0** and **P1**:

1. A process sets its flag to **true** (indicating it wants to enter).
2. It sets turn to the other process (allowing it to enter first if needed).
3. It waits until the other process's flag is **false** or if it is its turn.
4. It enters the **critical section** safely.
5. After finishing, it sets its flag to **false**, allowing the other process to enter.

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);
                critical section
        flag[i] = false;
                remainder section
    } while (true);
```

Semaphore

Semaphore ***S*** (integer variable) can only be accessed via two indivisible (atomic) operations. They are –

1. **wait()**
2. **signal()**

Originally called **P( )** and **V( )**

wait() and signal():

```
wait(S) {
    while (S <= 0); // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

Algorithm:

```
semaphore S = 1;

Process() {
     wait(S);
     // Critical Section
     signal(S);
}
```

Deadlock and Starvation

Deadlock: Deadlock happens when two or more processes (tasks) **get stuck** because each one is waiting for the other to release something it needs, but no one can move forward. It's like a traffic jam where everyone is waiting for the other car to move, but no one can.

Example:

Here, we have two processes (P0 and P1) and two semaphores (S and Q) initialized to 1 (indicating the resources are available).

P_0

```
wait(S);
wait(Q);
 ...
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
  ...
signal(Q);
signal(S);
```

Scenario:

- P0 waits for S and then for Q.
- P1 waits for Q and then for S.

If P0 acquires S first and P1 acquires Q first, both processes will be stuck in the second wait since P0 is waiting for Q and P1 is waiting for S. Neither can proceed, leading to **deadlock**.

Solution:

Deadlock can be avoided or handled by using techniques such as:

- **Lock ordering:** Always acquire resources in a fixed order to avoid circular dependencies.
- **Deadlock detection and recovery:** Periodically check for deadlocks and act (like aborting a process).
- **Timeouts:** Processes only wait for a certain time, and if they can't get a resource, they release their resources and try again.

Starvation: Starvation happens when a process is waiting forever to get what it needs to run, while other processes keep going ahead of it. It's like waiting in a long line, and someone always cuts in front of you, so you never get your turn.

Example:

If one process has **low priority** and others have **higher priority**, the lower-priority process might always get pushed back, and never gets a chance to run.

Solution:

You can fix starvation by giving everyone a fair turn, or gradually increasing the priority of the waiting process so it eventually gets a chance to run.

Dining Philosopher Problem

Problem Statement:

- There are **5 philosophers** sitting around a circular table.
- Each philosopher has a **plate of food** but needs **two chopsticks** (one on the left and one on the right) to eat.
- There are only **5 chopsticks**, so not all philosophers can eat at the same time.
- A philosopher can either **think** or **eat,** but cannot eat without holding both chopsticks.

Issues to Solve:

1. **Mutual Exclusion** – No two neighbors should use the same chopstick at the same time.
2. **Deadlock Prevention** – If all philosophers pick up their left chopstick first, they will wait forever for the right chopstick.
3. **Starvation Prevention** – Every philosopher should eventually get a turn to eat.

Solution Using Semaphores:

1. Semaphore chopstick [5] initialized to 1.
2. chopstick[i] = 1 means chopstick is available.
3. Each philosopher follows this process:
   - **Pick up the left chopstick** (wait on chopstick[i]).
   - **Pick up right chopstick** (wait on chopstick[(i+1) % 5]).
   - **Eat** (critical section).
   - **Put down right chopstick** (signal chopstick[(i+1) % 5]).
   - **Put down left chopstick** (signal chopstick[i])

The structure of Philosopher *i*:

```
do {
     wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );
               //  eat
     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );
               //  think
} while (TRUE);
```

Bounded Buffer Problem

Problem Statement:

- **Producer:** Adds data (items) into the buffer. If the buffer is full, the producer must wait.
- **Consumer:** Removes data from the buffer. If the buffer is empty, the consumer must wait.
- The buffer has a **maximum size** (bounded), so it can't hold an infinite number of items.

Key Concepts:

1. **Producer Process:**
   - The producer generates data and stores it in the buffer.
   - It waits if the buffer is full.
2. **Consumer Process:**
   - The consumer consumes data from the buffer.
   - It waits if the buffer is empty.

Issues to Solve:

- **Mutual Exclusion:** Only one process (producer or consumer) should access the buffer at a time to prevent conflicts.
- **Buffer Overflows:** If the producer keeps adding items when the buffer is full, it can overflow.
- **Buffer Underflows:** If the consumer tries to consume when the buffer is empty, it can underflow.

Semaphore Solution (using Counting Semaphores):

- **Empty** semaphore: Tracks the number of empty slots in the buffer.
- **Full** semaphore: Tracks the number of filled slots in the buffer.
- **Mutex** semaphore: Ensures mutual exclusion, allowing only one process to access the buffer at a time.

Solution Algorithm:

Initialization:

- Set empty = N (buffer size).
- Set full = 0 (no items in the buffer initially).
- Set mutex = 1 (mutex for mutual exclusion).

Producer Algorithm:

```
do {
        // Produce item
        wait(empty);
        wait(mutex);

        // Add item to buffer
        signal(mutex);
        signal(full);
    } while (true);
```

Consumer Algorithm:

```
do {
        wait(full);
        wait(mutex);
        // Remove item from buffer

        signal(mutex);
        signal(empty);
        // Consume item
    } while (true);
```

How It Works:

- The **producer** waits if the buffer is full (waiting on the empty semaphore), and the **consumer** waits if the buffer is empty (waiting on the full semaphore).
- The **mutex** ensures that only one process (either producer or consumer) accesses the buffer at a time to prevent race conditions.

Process Synchronization Benefits

1. **Prevents Data Corruption**: Synchronization ensures that multiple processes do not access shared data at the same time, preventing data corruption.
2. **Ensures Consistency**: It helps maintain data consistency by ensuring that shared resources are accessed in an orderly manner.
3. **Avoids Race Conditions**: Synchronization prevents race conditions where processes compete for resources, leading to unpredictable behavior.
4. **Improves Efficiency**: By properly managing access to shared resources, synchronization allows processes to run smoothly without unnecessary delays.
5. **Better Resource Management**: It ensures that resources (like memory or files) are used efficiently without conflicts or wastage.
6. **Prevents Deadlock**: Proper synchronization techniques can help avoid deadlocks, where processes are stuck waiting for each other.
7. **Improves System Stability**: Synchronization ensures that processes can run in parallel without causing system instability or crashes.
8. **Supports Concurrent Processing**: It allows multiple processes to work together efficiently, making use of the system's full processing power.
9. **Enhances Multitasking**: Synchronization helps manage multiple tasks that run simultaneously, making multitasking safer and more effective.

Chapter 6

Scheduling Criteria

1. **CPU Utilization:** Keep the CPU busy as much as possible, avoiding idle time.
2. **Throughput:** The number of processes that finish their work in a given amount of time.
3. **Turnaround Time:** The total time it takes to complete a process, from the moment it starts until it finishes.
4. **Waiting Time:** The amount of time a process has been waiting in the ready queue, waiting for the CPU to start working on it.
5. **Response Time:** The time from when a request is made until the first response is given (not the final output). This is especially important in systems where users are interacting with the computer in real-time (like time-sharing systems).

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First Come First Serve (FCFS) Scheduling

Definition: FCFS is a **non-preemptive** scheduling algorithm where processes are executed in the order they arrive in the ready queue. The first process that arrives gets executed first, followed by the next one, and so on.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Advantages of FCFS:

- Simple and easy to implement.
- Works well when processes have similar lengths or are expected to complete quickly.

Disadvantages of FCFS:

- **Convoy Effect**: If a process with a long execution time arrives first, it can delay all subsequent processes, leading to **long waiting times** for others (especially for short processes).
- **No Preemption**: Processes can't be interrupted, meaning if a process is stuck or slow, it can affect the overall system performance.

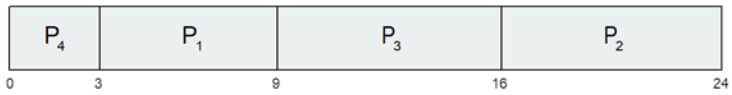
Shortest-Job-First (SJF) Scheduling

Definition: SJF is a **non-preemptive** scheduling algorithm that selects the process with the **shortest burst time** (the shortest amount of time it needs to complete) next.

SJF is optimal – gives minimum average waiting time for a given set of processes.

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Advantages of SJF:

1. Minimizes waiting time and turnaround time.
2. Efficient for CPU-bound processes with predictable burst times.

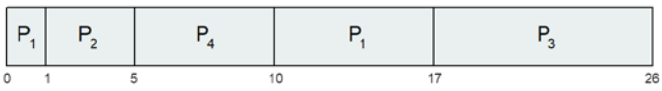
Disadvantages of SJF:

1. Difficult to predict burst time in advance.
2. Can cause starvation for long processes.
3. Non-preemptive, leading to potential blocking of shorter processes by longer ones.

Shortest Remaining Time First (SRTF) Scheduling

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Round Robin (RR) Scheduling

Definition:

Round Robin (RR) is a preemptive CPU scheduling algorithm where each process is given a small, fixed amount of CPU time called a **time quantum** (usually 10-100 milliseconds). When a process's time quantum expires, it is preempted and moved to the end of the ready queue, and the next process in line gets its turn.

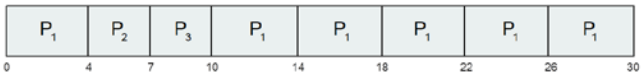
How It Works:

- Each process in the ready queue gets the CPU for **at most** the time quantum (q).
- If the process finishes before the time quantum expires, it releases the CPU early.
- If the process does not finish within its time quantum, it is **preempted** (interrupted) and placed back at the end of the ready queue.
- The CPU scheduler moves to the next process in the queue.
- The timer interrupt occurs after each time quantum to switch to the next process.

Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

CPU Scheduling Benefits

1. **Maximizes CPU Usage**: It keeps the CPU busy as much as possible, reducing downtime.
2. **Fairness**: It ensures that every process gets a fair amount of time on the CPU.
3. **Increases Process Completion**: Good scheduling helps complete more processes in a given time.
4. **Reduces Waiting Time**: It makes processes wait less before they get their turn to run.
5. **Faster Process Completion**: By managing process order, it reduces the total time for a process to finish.
6. **Improves Response Time**: In systems with users, it speeds up the time it takes to respond to their requests.
7. **Balances CPU Time**: It makes sure no process takes all the CPU time, giving others a chance too.
8. **Supports Multiple Users**: In multi-user systems, it helps share the CPU effectively, so everyone can interact with the system.
9. **Better Resource Management**: It helps manage CPU resources, avoiding problems like deadlock and making the system run more smoothly.