OPERATING SYSTEMS

UNIT-3

3.1 Inter Process Communication Mechanisms:

In an operating system, we often have **multiple processes** running at the same time. Sometimes, these processes need to **communicate** with each other — either to share data or to inform one another that **something has happened** (like an event or a status change). This communication between processes is called **Interprocess Communication (IPC)**. Operating systems provide special mechanisms or methods to make this communication possible.

Why do processes need to communicate?

Imagine different processes doing different jobs, like:

- One process is reading data from a file.
- Another process is processing that data.
- A third one is showing the result to the user.

If these processes don't communicate or share information, they can't work together properly.

Types of Processes (Based on Communication)

1. Independent Process

- These processes **don't share any data** with other processes.
- They work completely on their **own**.
- They **don't get affected** by what other processes are doing.
- Example: A calculator app running in the background while you're typing a document — they don't interact with each other.

2. Cooperating Process

- These processes **share data** with other processes.
- They **depend on** or **get affected by** other processes.
- For these processes to work correctly, they must **communicate and share data**, which is done using **IPC**.
- Example: A video editing software (one process loading video frames, another adding effects) — they need to cooperate.

Why Do We Need Cooperating Processes?

Here are some main reasons why cooperation among processes is important:

1. Information Sharing

- Many users or programs may need access to the **same file or data**.
- For example, multiple users may read a file stored in a shared drive.
- The system should allow such **safe and coordinated access**.

2. Computation Speedup

- If a large task is broken into smaller subtasks, and those subtasks run at the **same time (in parallel)**, the job finishes faster.
- Example: When a video is rendered using multiple cores — each core does a part of the work.

3. Modularity

- A complex system is easier to manage when it's divided into **smaller, separate parts** (modules).
- Each module can be created as an **independent process**.
- This makes the system easier to **develop, test, and maintain**.

4. Convenience

- Users often perform **multiple tasks at the same time**.
- For example: You may be writing a document, downloading a file, and listening to music — all at once.
- All these tasks run in **parallel**, and some of them may need to communicate for smooth functioning.

Two Main Models of IPC

There are two major ways in which processes can communicate:

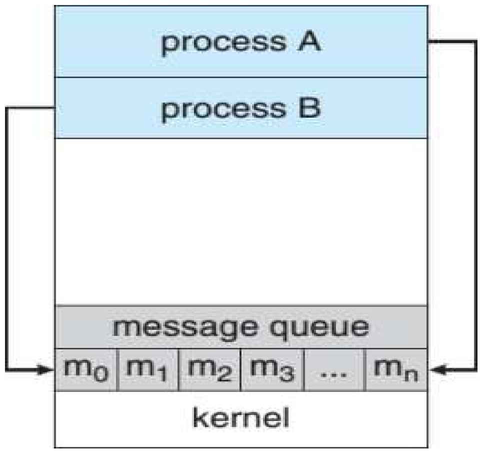
1. Message Passing

- Processes **send and receive messages** from each other.
- No shared memory is used.
- Suitable for systems where **processes are running on different computers or don't have common memory**.
- It is simple and secure because messages go through the operating system.

2. Shared Memory

- A specific area of memory is made **common (shared)** between processes.
- Processes can **read from and write to** this shared area.
- Communication is **faster** compared to message passing.
- But it is **more complex** because processes must make sure they **don't overwrite** each other's data.

1. Message-Passing Systems



In a **message-passing system**, two or more processes **communicate with each other by sending and receiving messages**. This model is useful when processes **do not share the same memory** and especially important in **distributed systems** (like different computers connected over a network).

Key Points:

- In this model, processes **don't share memory** – they just **exchange messages**.
- It helps **synchronize** the processes (i.e., keep them working in proper order).
- It's mostly used when processes are on **different systems** or machines.
- Two main actions involved:
  - **Send** (sending a message)
  - **Receive** (receiving a message)
- Messages can be of **fixed size** or **variable size**.
- **Example**: In an internet chat app, users communicate by **sending and receiving messages**.

Communication Link Example:

Let's say **Process P** and **Process Q** want to talk to each other. They can send and receive messages through a **communication link** like:

- A **hardware bus** (if on the same machine), or
- A **network** (if on different machines).

How Logical Communication Links Are Implemented:

There are **3 main methods** used to set up communication between processes:

1. NAMING

Processes must know **where to send** or **from whom to receive** messages. There are two ways of doing this:

a) Direct Communication

- Processes talk to **each other directly** by using names.
- Example:
  - `send (P, message)` → Send message to process **P**
  - `receive (Q, message)` → Receive message from process **Q**

Properties of direct communication:

- A link is created automatically if two processes want to communicate.
- Each communication link connects **exactly two processes**.
- Both sender and receiver must know each other's name (ID).

b) Indirect Communication

- Messages are sent through **mailboxes (also called ports)**.
- A **mailbox** is like a message box where a process can drop messages, and another can pick them up.

How it works:

- `send (A, message)` → Send message to **mailbox A**
- `receive (A, message)` → Get message from **mailbox A**

Properties of indirect communication:

- A mailbox has a **unique ID**.
- Multiple processes can share a mailbox.
- One process can use **multiple mailboxes** for communication.
- More than two processes can communicate using **one mailbox**.

2. SYNCHRONIZATION

This refers to **how sending and receiving happens — whether a process waits or continues immediately.**

There are **two types**:

i) Synchronous (Blocking) Communication

- **Blocking Send**: The sender **waits** until the message is received.
- **Blocking Receive**: The receiver **waits** until a message is available.

ii) Asynchronous (Non-Blocking) Communication

- **Non-blocking Send**: The sender **sends the message and continues** working without waiting.
- **Non-blocking Receive**: The receiver checks — if a message is there, it receives it. If not, it moves on (may get null or empty response).

3. BUFFERING

When messages are sent, they are usually kept in a **temporary queue** until received. There are **three types** of buffering based on how this queue behaves:

a) Zero Capacity

- No queue at all.
- Sender **must wait** until receiver is ready.
- No message is stored in between.

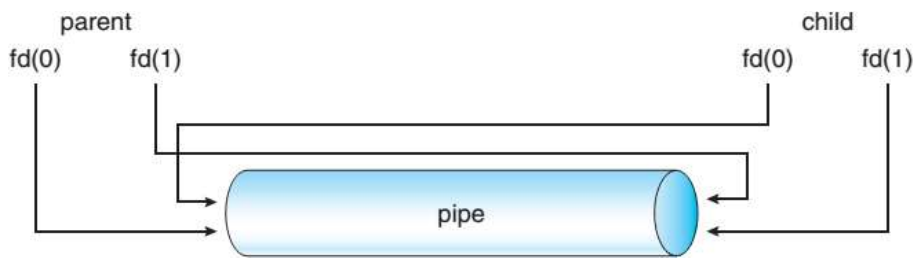
b) Bounded Capacity

- The queue has a **fixed size** (say, n messages max).
- If the queue is **not full**, message is stored, and sender can continue.
- If the queue is **full**, sender **must wait** until space is available.

c) Unbounded Capacity

- The queue can grow as needed (no fixed size).
- Sender **never needs to wait**, because there’s always room for more messages.

2. Pipes



In early **UNIX systems**, **pipes** were one of the **first IPC (Interprocess Communication) mechanisms** used to allow processes to communicate with each other.

There are **two types of pipes** in UNIX:

1. **Ordinary Pipes**
2. **Named Pipes**

1. Ordinary Pipes

Ordinary pipes are used for **one-way communication** between two related processes — usually a **parent** and its **child process**. They follow a **Producer-Consumer model**.

How They Work:

- One process **writes data** into one end of the pipe (called the **write-end**).
- The other process **reads data** from the other end (called the **read-end**).

Key Points:

- **Unidirectional**: Only one-way communication is allowed. If we want two-way communication, we need **two separate pipes**.
- **Only works between related processes**, like a parent and its child.

Creation in UNIX:

- Pipes are created using the function: pipe(int fd[ ])
- This function creates a pipe with two file descriptors:
  - fd[0] → Read-end
  - fd[1] → Write-end

Example Flow:

- A **parent process** creates a pipe.
- Then it **forks** a child process.
- The child process **inherits** the pipe from the parent.
- One process writes to the pipe, and the other reads from it.

System Calls Used:

- Ordinary read() and write() system calls are used to communicate through the pipe.

Important:

- **Ordinary pipes can't be used by unrelated processes**.
- The pipe **doesn't show up as a file** in the file system — it's just in memory.

2. Named Pipes (FIFOs)

Named Pipes, also known as **FIFOs (First In, First Out)**, are used for **communication between unrelated processes**. They allow **two-way communication**, but in a controlled manner.

Key Features:

- **No parent-child relation** is needed.
- Once created, a **named pipe appears as a file** in the file system.
- It can be used by **multiple processes** at once.
- A named pipe continues to **exist even after communication is done**, until it is manually deleted.

Creating a Named Pipe:

- Use the function: mkfifo() to create a FIFO.
- Then use normal file operations: open(), read(), write(), and close().

Communication Nature:

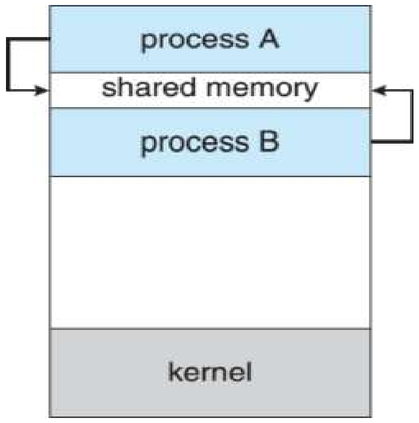
- Although **FIFOs support two-way communication**, in practice, **each FIFO handles only one direction at a time**.
- So if you want **data to travel in both directions**, you need to create **two FIFOs** (one for each direction).

System Support: Both UNIX and Windows support named pipes.

Limitation:

- Communicating processes using FIFOs must be on the **same machine**.
- If communication is needed **between different machines**, use **Sockets** instead.

3. Shared Memory:



In the **shared-memory model**, a specific region of memory is shared between multiple processes that need to cooperate and exchange data.

How It Works:

- A **shared memory region (or segment)** is created inside a process's **address space**.
- Other processes that want to communicate **attach this segment to their own address space**.
- Once attached, processes can **read from or write to this shared area** to exchange data.

Important Notes:

- Normally, operating systems **don't allow** one process to access another process's memory for safety.
- But in shared memory, **this restriction is lifted** by mutual agreement of the processes.
- The **format and location** of the data in the shared region is decided by the **processes themselves**, not by the OS.
- It's the **responsibility of the processes** to make sure they don't **access or update the same memory location at the same time**, which could cause data issues.

Inter-Process Communication (IPC) in UNIX/Linux

IPC (Inter-Process Communication) allows processes to communicate with each other and coordinate their actions. This is essential for processes that need to share data or synchronize execution.

There are several types of IPC mechanisms provided by UNIX/Linux:

1. Pipes
2. FIFOs (Named Pipes)
3. Semaphores (System V API)
4. Message Queues (System V API)
5. Shared Memory (System V API)
6. IPC Status Commands

1. Pipes

Definition:

A **pipe** is a unidirectional communication channel used between **related processes** (such as a parent and its child process).

System Call:

```
int pipe(int fd[2]);
```

Parameters:

- `fd[0]`: File descriptor for reading.
- `fd[1]`: File descriptor for writing.

Explanation:

The `pipe()` system call creates a pair of file descriptors. One end is for reading and the other for writing. Data written into `fd[1]` can be read from `fd[0]`.

Example Program:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main() {
    int fd[2];
    char buffer[20];
    pipe(fd); // create pipe
    if (fork() == 0) { // child process
        close(fd[0]); // close read end
        write(fd[1], "Hello", strlen("Hello") + 1); // write message
    } else { // parent process
        close(fd[1]); // close write end
        read(fd[0], buffer, sizeof(buffer)); // read message
        printf("Received: %s\n", buffer);
    }
    return 0;
}
```

Explanation:

- A pipe is created.
- The child writes "Hello" to the pipe.
- The parent reads from the pipe and prints the message.

1.1 Bidirectional Pipes (Using Two Pipes)

Definition:

Normal pipes are unidirectional. To achieve **two-way communication** (like a conversation), we use **two pipes**.

Concept:

- One pipe is for communication from **Parent to Child**.
- The other pipe is for communication from **Child to Parent**.

Example Program:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
int main() {
    int fd1[2], fd2[2];
    char parent_msg[] = "Hello from Parent";
    char child_msg[] = "Hello from Child";
    char buffer[100];
    pipe(fd1); // Parent to Child
    pipe(fd2); // Child to Parent
    if (fork() == 0) { // Child process
```

```
        close(fd1[1]); // Close write end of fd1
        close(fd2[0]); // Close read end of fd2
        read(fd1[0], buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);
        write(fd2[1], child_msg, strlen(child_msg) + 1);
    } else { // Parent process
        close(fd1[0]); // Close read end of fd1
        close(fd2[1]); // Close write end of fd2
        write(fd1[1], parent_msg, strlen(parent_msg) + 1);
        read(fd2[0], buffer, sizeof(buffer));
        printf("Parent received: %s\n", buffer);
    }
    return 0;
}
```

2. FIFOs (Named Pipes)

Definition:

FIFOs (named pipes) are special file types that allow communication between **unrelated processes**. Unlike regular pipes, FIFOs exist in the filesystem with a name.

System Call:

```
int mkfifo(const char *pathname, mode_t mode);
```

What does `mkfifo()` do?

`mkfifo()` creates a named pipe (FIFO) special file in the filesystem. This FIFO can then be opened using `open()` and used just like a regular file descriptor for reading or writing.

Parameters:

`pathname`: The name of the FIFO file to be created. `mode`: The file permission bits (e.g., `0666` allows read/write for all users).

Example Programs:

Writer.c

```
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
int main() {
    int fd;
    mkfifo("myfifo", 0666); // create FIFO file
    fd = open("myfifo", O_WRONLY); // open FIFO in write-only mode
    write(fd, "Hello FIFO", 10); // write to FIFO
    close(fd);
    return 0;
}
```

Reader.c

```
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    int fd;
    char buffer[20];
    fd = open("myfifo", O_RDONLY); // open FIFO in read-only mode
    read(fd, buffer, sizeof(buffer)); // read from FIFO
    printf("Received: %s\n", buffer);
    close(fd);
    return 0;
}
```

Explanation:

- `mkfifo("myfifo", 0666)` creates a named pipe in the filesystem.
- One program opens it in **write** mode and sends data.
- Another opens it in **read** mode and receives the data.
- Used for communication between **independent** programs.

3. Semaphores (System V IPC)

Definition:

A **semaphore** is a synchronization tool used to control access to a shared resource by multiple processes in a concurrent system.

System V semaphores use a semaphore **set**, which may contain one or more semaphores.

Used For:

- Ensuring mutual exclusion
- Synchronizing processes

Important System Calls:

1. `semget()` – create a semaphore set or get the ID of an existing one.
2. `semop()` – perform operations on semaphore(s).
3. `semctl()` – control and manage semaphore values and properties.

3.1 semget()

```
int semget(key_t key, int nsems, int semflg);
```

Parameters:

- `key`: A unique key to identify the semaphore set (can be generated using `ftok()`).
- `nsems`: Number of semaphores in the set.
- `semflg`: Permissions (e.g., `0666`) and flags (e.g., `IPC_CREAT`).

3.2 semop()

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

Parameters:

- `semid`: Semaphore set identifier returned by `semget()`.
- `sops`: Array of `struct sembuf`, which defines the operations to perform.
- `nsops`: Number of operations to perform.

struct sembuf structure:

```
struct sembuf {
    unsigned short sem_num; // semaphore index
    short sem_op;           // operation (-1, 0, +1)
    short sem_flg;          // operation flags
};
```

3.3 semctl()

```
int semctl(int semid, int semnum, int cmd, ...);
```

Parameters:

- `semid`: ID of semaphore set.
- `semnum`: Semaphore number within the set.
- `cmd`: Command to perform (e.g., `SETVAL`, `GETVAL`).
- Additional argument: union semun (when required).

Simple Semaphore Example (Single Semaphore):

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>

int main() {
    int semid = semget(1234, 1, 0666 | IPC_CREAT);

    semctl(semid, 0, SETVAL, 1); // Set initial value to 1

    struct sembuf p = {0, -1, 0}; // P operation (wait)
    struct sembuf v = {0, 1, 0};  // V operation (signal)

    printf("Trying to enter critical section...\n");
    semop(semid, &p, 1); // wait

    printf("Inside critical section\n");
    sleep(2);
    printf("Exiting critical section\n");
    semop(semid, &v, 1); // signal

    return 0;
}
```

Explanation:

- A semaphore set with 1 semaphore is created.
- Value is initialized to 1 (available).
- Process performs a wait (P) operation before entering critical section.
- After critical section, signal (V) operation is done.

Message Queues

Definition:

A **message queue** is an IPC mechanism that allows processes to communicate by sending and receiving messages. Messages are stored in a queue, and processes can retrieve them in a **first-in, first-out (FIFO)** order.

System Calls:

1. `msgget()` – Creates or accesses a message queue.
2. `msgsnd()` – Sends a message to the queue.
3. `msgrcv()` – Receives a message from the queue.
4. `msgctl()` – Controls or manages message queue properties.

4.1 msgget()

```
int msgget(key_t key, int msgflg);
```

Parameters:

- **`key`**: A unique key to identify the message queue. This can be generated using `ftok()`.
- **`msgflg`**: Flags and permissions for the queue. Flags include:
    - `IPC_CREAT`: Create a new queue if it doesn't exist.
    - `IPC_EXCL`: Fail if the queue already exists.
    - `0666`: Permissions (read/write for all users).

4.2 msgsnd()

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Parameters:

- **`msqid`**: Message queue ID (returned by `msgget()`).
- **`msgp`**: Pointer to the message structure that you want to send.
- **`msgsz`**: Size of the message (the length of data in the message, excluding the `mtype` field).
- **`msgflg`**: Flags (e.g., `MSG_NOERROR`).

4.3 msgrcv()

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Parameters:

- **`msqid`**: Message queue ID (returned by `msgget()`).
- **`msgp`**: Pointer to a buffer where the message will be stored.
- **`msgsz`**: Maximum size of the message to receive.
- **`msgtyp`**: Type of message to receive. Messages can be prioritized.
- **`msgflg`**: Flags (e.g., `MSG_NOERROR`).

4.4 msgctl()

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Parameters:

- **`msqid`**: Message queue ID (returned by `msgget()`).
- **`cmd`**: Command to control the queue, such as:
    - `IPC_STAT`: Get the status of the queue.
    - `IPC_RMID`: Remove the queue.
- **`buf`**: A pointer to the `msqid_ds` structure that holds the information about the message queue.

Example Program for Message Queue:

Sender.c:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>
// Message structure
struct msgbuf {
    long mtype;                 // Message type
    char mtext[100];            // Message content
};
int main() {
int msqid = msgget(1234,0666 | IPC_CREAT); // create/access message queue
    struct msgbuf message;
    message.mtype = 1;  // message type
    strcpy(message.mtext, "Hello, Message Queue!"); // message content
    msgsnd(msqid, &message, sizeof(message.mtext), 0); // send message
    printf("Message sent: %s\n", message.mtext);
    return 0;
}
```

Receiver.c:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>
// Message structure
struct msgbuf {
    long mtype;                 // Message type
    char mtext[100];            // Message content
};
int main() {
int msqid = msgget(1234,0666 | IPC_CREAT); // create/access message queue
    struct msgbuf message;
    msgrcv(msqid, &message, sizeof(message.mtext), 1, 0); // receive
message of type 1
    printf("Received message: %s\n", message.mtext);
    msgctl(msqid, IPC_RMID, NULL); // remove the message queue
    return 0;
}
```

Explanation:

- **Sender Program:**
  - The message queue is created/accessed using `msgget()`.
  - The message structure is populated with a message type (`mtype`) and the message content (`mtext`).
  - `msgsnd()` is used to send the message to the queue.
- **Receiver Program:**
  - The receiver generates the same key and accesses the message queue.
  - It receives the message of type `1` using `msgrcv()`.
  - After receiving, the message queue is removed using `msgctl()` with the `IPC_RMID` command.

Shared Memory

Definition:

Shared memory is an IPC mechanism that allows multiple processes to access the same memory space. This enables efficient communication between processes as they can directly read and write to the shared memory region.

System Calls:

1. `shmget()` – Creates or accesses a shared memory segment.
2. `shmat()` – Attaches the shared memory segment to the process's address space.
3. `shmdt()` – Detaches the shared memory segment from the process's address space.
4. `shmctl()` – Controls or manages shared memory properties.

5.1 shmget()

```
int shmget(key_t key, size_t size, int shmflg);
```

Parameters:

- **`key`**: A unique key to identify the shared memory segment. This can be generated using `ftok()`.
- **`size`**: The size of the shared memory segment in bytes.
- **`shmflg`**: Flags and permissions for the shared memory segment. Common flags include:
  - `IPC_CREAT`: Create a new shared memory segment if it doesn't exist.
  - `IPC_EXCL`: Fail if the segment already exists.
  - `0666`: Permissions (read/write for all users).

5.2 shmat()

```
void* shmat(int shmid, const void *shmaddr, int shmflg);
```

Parameters:

- **`shmid`**: Shared memory segment ID (returned by `shmget()`).
- **`shmaddr`**: Address to which the segment should be attached. If set to `NULL`, the system will choose an appropriate address.
- **`shmflg`**: Flags for attaching the shared memory segment. Common flags include:
  - `SHM_RDONLY`: Attach the segment in read-only mode.
  - `SHM_RND`: Round the address to a multiple of the page size.

5.3 shmdt()

```
int shmdt(const void *shmaddr);
```

Parameters:

- **`shmaddr`**: The address of the shared memory segment that should be detached.

5.4 shmctl()

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Parameters:

- **`shmid`**: Shared memory segment ID (returned by `shmget()`).

- **cmd**: Command to control the shared memory segment, such as:
    - `IPC_STAT`: Get the status of the shared memory segment.
    - `IPC_RMID`: Remove the shared memory segment.
- **buf**: A pointer to the `shmid_ds` structure that holds the information about the shared memory segment.

Structure shmid_ds

```
struct shmid_ds {
    struct ipc_perm shm_perm; // Permissions
    size_t shm_segsz;         // Segment size
    time_t shm_atime;         // Last attach time
    time_t shm_dtime;         // Last detach time
    time_t shm_ctime;         // Last change time
    pid_t shm_cpid;           // PID of creator
    pid_t shm_lpid;           // PID of last operation
    shmatt_t shm_nattch;      // Number of attached processes
};
```

Example Program for Shared Memory:

Writer.c:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {
int shmid = shmget(1234, 1024, 0666 | IPC_CREAT); // create shared memory
char *str = (char*) shmat(shmid, (void*) 0, 0); // attach shared memory
    printf("Enter some text: ");
    fgets(str, 1024, stdin); // write to shared memory

    printf("Written to shared memory: %s\n", str);
    return 0;
}
```

Reader.c:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int main() {
int shmid = shmget(1234, 1024, 0666 | IPC_CREAT); // access shared memory
char *str = (char*) shmat(shmid, (void*) 0, 0); // attach shared memory
printf("Data read from shared memory: %s\n", str); // read from shared
memory
    shmdt(str); // detach shared memory
    shmctl(shmid, IPC_RMID, NULL); // remove shared memory
    return 0;
}
```

Explanation:

- **Writer Program:**
  - The shared memory segment is created using `shmget()`.
  - The writer attaches the segment to its address space using `shmat()`.
  - It then writes user input into the shared memory.
- **Reader Program:**
  - The reader generates the same key and accesses the shared memory segment using `shmget()`.
  - It attaches the segment to its address space using `shmat()`.
  - The reader reads the data from the shared memory and prints it.
  - Finally, the reader detaches the shared memory using `shmdt()` and removes the shared memory segment using `shmctl()`.

IPC Status Commands

IPC status commands are used to manage and get information about IPC resources such as message queues, shared memory, and semaphores. They provide crucial system information such as the state of the resources, attached processes, etc.

System Calls:

- `ipcs` – Displays the current status of IPC resources.
- `ipcrm` – Removes an IPC resource.

1. **ipcs:** The **`ipcs`** command is used to report the status of IPC resources such as shared memory, message queues, and semaphores. It provides detailed information about the resources, such as their IDs, permissions, and other details.

Syntax:

```
ipcs [options]
```

Common Options:

- `-m`: Show shared memory status.
- `-q`: Show message queue status.
- `-s`: Show semaphore status.
- `-a`: Show all IPC resources.
- `-t`: Display the time information.
- `-c`: Show creator's ID for the IPC resources.

Example:

```
ipcs -m
```

This command will display the status of shared memory segments.

2. **ipcrm:** The **`ipcrm`** command is used to remove IPC resources, including message queues, shared memory segments, and semaphores. This is useful for cleaning up unused or orphaned IPC resources.

Syntax:

```
ipcrm [options] resource_id
```

Common Options:

- `-m`: Removes shared memory.
- `-q`: Removes message queue.
- `-s`: Removes semaphore.

Example:

```
ipcrm -m <shmid>
```

This command removes the shared memory segment with the specified `shmid`.

DEADLOCKS

A **deadlock** occurs when a group of processes are stuck waiting for each other to release resources, and **none of them can proceed**. In other words, every process in the group is waiting for something that **only another process in the same group can do**, like releasing a resource.

System Model

A computer system has a **limited number of resources** that must be **shared among multiple processes**.

Types of Resources

1. **Physical Resources** – These are hardware-related:
   - Printers
   - Tape drives
   - DVD drives
   - Memory space
   - CPU cycles
2. **Logical Resources** – These are software-related:
   - Semaphores
   - Mutex locks
   - Files

Each **resource type** can have **multiple identical instances**.

Example: If a system has **2 CPUs**, then the resource type "CPU" has **2 instances**.

Resource Usage Sequence

A process uses a resource by following **three steps**:

1. **Request** –
   The process asks for a resource.
   - If the resource is free → it gets the resource.
   - If not → it has to wait until the resource becomes free.
2. **Use** –
   Once the resource is assigned, the process performs operations using it.
   - Example: If it's a printer, the process starts printing.
3. **Release** –
   After using the resource, the process releases it so that **other waiting processes** can use it.

System Calls for Resource Handling

Type	**Request**	**Release**
Device	`request()`	`release()`
Semaphore	`wait()`	`signal()`
Mutex Lock	`acquire()`	`release()`
Memory	`allocate()`	`free()`
File System	`open()`	`close()`

System Table

The OS maintains a **System Table** to keep track of:

- Whether a **resource is free or allocated**
- Which **process is holding** a particular resource
- Which processes are **waiting** for a resource (stored in a **queue**)

So, if a process requests a resource that is already in use, it will be added to the **waiting queue** for that resource.

DEADLOCK CHARACTERIZATION

Definition

Deadlock Characterization refers to the **set of conditions and structures** that help us **understand how and why a deadlock can occur** in a system. It identifies the **specific features or prerequisites** that must be present for deadlock to happen.

A **deadlock** is a situation in a **multiprogramming system** where **two or more processes get stuck**, each waiting for resources that are being held by the others. Because of this, **none of the processes can proceed**, and they remain blocked forever.

Causes of Deadlock

Deadlock occurs when certain **specific conditions** are all true at the same time. These are known as the **Deadlock Prerequisites**.

i. Deadlock Prerequisites – Four Necessary Conditions

For a deadlock to happen, **all four of these must occur together**:

1. **Mutual Exclusion**
   - Only **one process can use a resource** at any given time.
   - If another process requests it, it must **wait** until it is released.
2. **Hold and Wait**
   - A process is **holding at least one resource** and is **waiting for more** that are being used by other processes.
3. **No Preemption**
   - A resource **can't be forcibly taken away** from a process.
   - The process must **release it willingly**.
4. **Circular Wait**
   - There is a **circular chain of processes** waiting for each other's resources. For example:

     P0 waits for a resource held by P1,

     P1 waits for a resource held by P2,

     …

     Pn waits for a resource held by P0.

If even one of these conditions is missing, deadlock won't happen.

SYSTEMS RESOURCE ALLOCATION GRAPH

The **Resource Allocation Graph (RAG)** is a **visual tool** used to identify **deadlocks** in a system. It shows how **resources are requested and allocated** among processes.

Definition

A **Resource Allocation Graph**, denoted as **G = {V, E}**, is a **directed graph** where:

- **V** is the set of **vertices (nodes)**
- **E** is the set of **edges (arrows)**

The vertices are divided into **two types**:

1. **Process Nodes (P)**: Represent all the active processes in the system
   Example: P = {P1, P2, ..., Pn}
2. **Resource Nodes (R)**: Represent all the resource types in the system
   Example: R = {R1, R2, ..., Rm}

Types of Edges

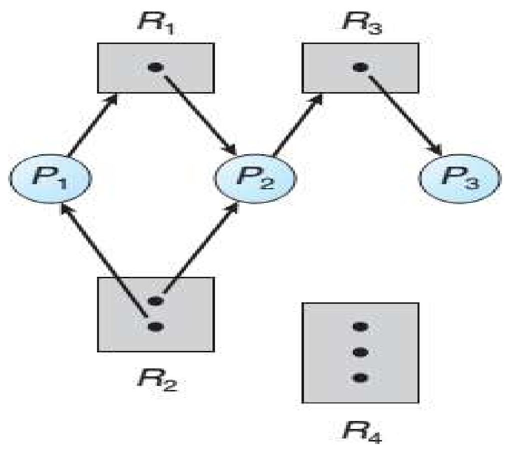
There are **two types of edges** in the graph:

1. **Request Edge (Pi → Rj)**
   - Shows that process **Pi is requesting** resource **Rj**
   - Pi is **waiting** for Rj to be available
2. **Assignment Edge (Rj → Pi)**
   - Shows that an instance of resource **Rj is allocated** to process **Pi**

Symbols Used

- **Process** = Circle
- **Resource** = Rectangle
- **Instance of resource** = Small dot inside rectangle

Resource Allocation Graph without Deadlock



Let's take an example:

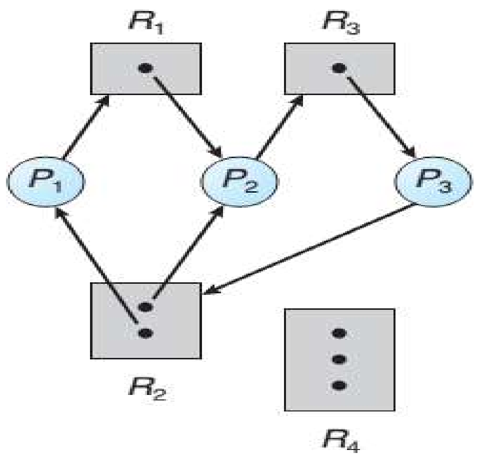
- Process set P = {P1, P2, P3}
- Resource set R = {R1, R2, R3, R4}
- Edges = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}
- R1 and R3 have **1 instance each**, R2 has **2 instances**, and R4 has **3 instances**

Observations:

- P1 is holding **R2** and **waiting** for **R1**
- P2 is holding **R1 and R2** and waiting for **R3**
- P3 is **holding** R3

Since there's **no cycle** in this graph, there is **no deadlock**.

Resource Allocation Graph with a Cycle and Deadlock



Consider a graph where:

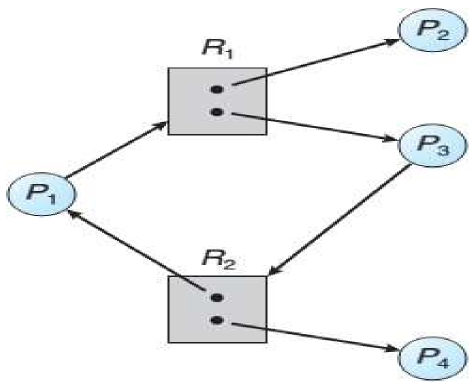
- P1 → R1 → P2 → R3 → P3 → R2 → P1
- P2 → R3 → P3 → R2 → P2

Observations:

- P2 is waiting for R3 (held by P3)
- P3 is waiting for R2 (held by P1 or P2)
- P1 is waiting for R1 (held by P2)

Cycle exists, and **every process is waiting**—this is a **deadlock situation**.

Resource Allocation Graph with a Cycle and No Deadlock



Let’s say we have a cycle:
P1 → R1 → P3 → R2 → P1

Observations:

- A **cycle exists**, but **not all processes are waiting**
- P4 and P2 are **not waiting for any resources**
- P4 might **release R2**, which could be given to P3, **breaking the cycle**

So, in this case, a **cycle exists**, but **no deadlock occurs**.

Methods For Handling Deadlocks

Deadlocks in an operating system can be handled in **four ways**:

1. **Deadlock Prevention**
2. **Deadlock Avoidance**
3. **Deadlock Detection and Recovery**
4. **Deadlock Ignorance (Ostrich Method)**

Deadlock Prevention

Deadlock is prevented by ensuring **at least one of the four necessary conditions** (Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait) **never holds**.

Deadlock Avoidance

The system is carefully scheduled so it **never enters an unsafe state** where deadlocks could occur.

- Requires **prior knowledge** of how resources will be requested.
- Example: **Banker's Algorithm** checks whether resource allocation keeps the system in a safe state.

Deadlock Detection and Recovery

Let deadlocks happen but **detect them** using algorithms and **recover** afterward.

- Use detection algorithms to check if a cycle (deadlock) exists in the resource allocation graph.
- Once detected, take recovery actions such as:
  - **Terminate one or more processes**
  - **Preempt resources** from some processes

Deadlock Ignorance (Ostrich Method)

This method is called the **Ostrich Method**, based on the idea that **"if you ignore the problem, maybe it will go away."**

- **The system assumes deadlocks are rare**, so it simply ignores them.
- Used in systems where:
  - **Deadlocks are very rare**
  - **Cost of prevention or detection is higher than occasional restart**
  - Example: Most personal desktop operating systems like Windows and Linux follow this approach.

Advantage:

- Simple and low-cost to implement

Disadvantage:

- If a deadlock occurs, the system may become unresponsive and **require a manual restart**.

Deadlock Prevention:

Deadlock prevention is a technique that **ensures at least one of the four necessary conditions for deadlock never holds true**. These four conditions are:

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

By breaking even **one** of these conditions, deadlocks can be **completely avoided**.

1. Mutual Exclusion

- This condition applies when **resources cannot be shared** and are **non-sharable**.
- **We cannot prevent deadlock by removing mutual exclusion**, because some resources are naturally non-sharable.

Examples of Non-sharable Resources:

- A mutex lock – only one process can hold it at a time.
- A printer – can't be used by multiple processes simultaneously.

Examples of Sharable Resources:

- Read-only files – multiple processes can access them at the same time.
- These resources **do not cause deadlocks**.

2. Hold and Wait

To prevent this condition, a process must **not hold one resource while waiting for others**.

Two common protocols are used:

Protocol 1 – All-at-once Request:

- A process must request **all resources** it needs **before it starts execution**.
- If any resource is not available, the process must wait **without holding anything**.

Example:

A process needs a DVD drive, disk file, and printer. It must request all three before starting, even if the printer is needed only at the end. This causes poor resource usage, as the printer stays idle for most of the time.

Protocol 2 – Request Only When Free:

- A process can request resources **only when it holds none**.
- After using some resources, it must **release them** before requesting new ones.

Example:

A process copies data from DVD to hard disk. Then, it releases those resources and later requests the disk and printer to complete the job.

Disadvantages:

- **Starvation**: A process might never get all resources together.
- **Low Resource Utilization**: Resources may remain unused for long periods.

3. No Preemption

To prevent this condition:

- If a process requests a resource that is **not immediately available**, it must **release all currently held resources**.
- These resources are added to its waiting list.
- The process is restarted only when **all required resources are available together**.

This ensures no process can "hold" resources while waiting for others.

4. Circular Wait

To break this condition, we can **impose a strict order on resource requests**.

- Assign a **unique number** to every resource type using a function:

`F: R → N` (e.g., F(Tape Drive) = 1, F(Disk Drive) = 5, F(Printer) = 12)

- Processes must request resources **only in increasing order** of these numbers.

Example:

A process that needs a tape drive and a printer must request the **tape drive first**, then the **printer**, following the order.

Alternative Rule:

- A process can only request a resource if it has **released any resource** with a **higher or equal number**.

Disadvantage of Deadlock Prevention

- These techniques often lead to **low system performance**.
- **Resource utilization is poor** because resources are kept idle.
- **System throughput** decreases, and **starvation** may occur in some cases.

Deadlock Avoidance

Deadlock occurs when a group of processes are each waiting for resources that are held by the others in the group. This situation leads to a state where none of the processes can continue their execution.

To prevent this situation, **deadlock avoidance** is used. It works by carefully examining resource requests before they are granted, to ensure the system does not enter an unsafe or deadlocked state.

Safe State: A system is in a **safe state** if there exists at least one sequence of all processes (called a **safe sequence**) such that each process can finish with the available resources at some point.

Unsafe State: A system is in an **unsafe state** if **no such sequence exists**. Unsafe state **does not mean** deadlock has occurred, but it means there is a possibility of entering deadlock if processes proceed.

Safe Sequence: A **safe sequence** is an ordering of processes (e.g., P1, P3, P0, etc.) where each process can finish using currently available resources plus the resources held by previously completed processes.

Ensuring that the system always remains in a **safe state** is the main goal of deadlock avoidance.

1. Resource Allocation Graph Algorithm (Used for Single Instance Resources)

This algorithm is used **only when each resource type has a single instance**.

Graph Components:

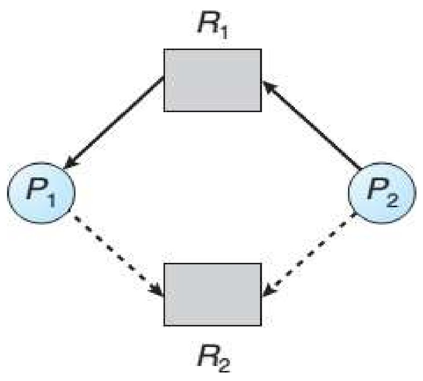
- **Request Edge (Pi → Rj)**: Process Pi is requesting resource Rj.
- **Assignment Edge (Rj → Pi)**: Resource Rj is currently allocated to Pi.
- **Claim Edge (Pi → Rj)**: Process Pi may request Rj in the future. This is shown as a **dashed line**.

Working of the Algorithm:

1. All claim edges must be declared **before** the process starts.
2. When a process Pi requests Rj:
    - Convert **claim edge** to a **request edge**.
3. Check for a **cycle**:
    - If granting request causes a cycle → **Unsafe**, request is denied.
    - If no cycle → **Safe**, request is granted.

Example:

Imagine we have a Resource Allocation Graph. Suppose:



- **P2 requests R2**, which is free.
- But granting R2 to P2 will create a **cycle** in the graph.

So, **even though R2 is available**, it cannot be granted to P2 to **avoid unsafe state**.

- Safe if no cycles.
- Unsafe if a cycle is created.

Limitation: This algorithm **does not work** when multiple instances of a resource are allowed.

2. Safe State Algorithm (To Check System's Safety)

This algorithm checks if the system is currently in a **safe state**. It does not handle resource requests directly but helps us verify if the system is safe.

Example: Safe State Check

Let's consider a system with **12 magnetic tape drives** and **3 processes: P0, P1, and P2**.

Table: Process Needs and Allocation

Process	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

- Total Allocated = 5 (P0) + 2 (P1) + 2 (P2) = **9**
- Available = 12 - 9 = **3**

We check whether a safe sequence exists:

Step 1: Try P1

- P1 needs 4 - 2 = **2** more.
- 3 are available. So allocate 2 to P1.
- P1 finishes and returns 4 tape drives. Total available = 3 + 2 = **5**

Step 2: Try P0

- P0 needs 10 - 5 = **5** more.
- 5 are available. So allocate 5 to P0.
- P0 finishes and returns 10 tape drives. Total available = 5 + 5 = **10**

Step 3: Try P2

- P2 needs 9 - 2 = **7** more.
- 10 are available. So allocate 7 to P2.
- P2 finishes and returns 9 tape drives. Total available = 10 + 2 = **12**

Safe Sequence: **< P1, P0, P2 >**

Conclusion: System is in a **safe state**.

Problem with this algorithm: Even if a resource is free, a process may not get it immediately to avoid unsafe state. This leads to **low resource utilization**.

3. Banker's Algorithm (Used for Multiple Instance Resources)

This is a general solution used when **multiple instances** of resources are available. It is called "Banker's Algorithm" because it works similarly to how a bank lends money while ensuring it can meet the needs of all customers.

Data Structures:

Assume there are **n processes** and **m resource types**.

Name	Description
Available	A vector of length m. Available[j] = number of free instances of resource j
Max	n × m matrix. Max[i][j] = maximum demand of Pi for Rj
Allocation	n × m matrix. Allocation[i][j] = number of instances currently allocated
Need	n × m matrix. Need[i][j] = Max[i][j] - Allocation[i][j]

Safety Algorithm (Step-by-Step):

1. Initialize:
    - Work = Available
    - Finish[i] = false for all i
2. Find a process i such that:
    - Finish[i] == false
    - Need[i] ≤ Work
3. If found:
    - Work = Work + Allocation[i]
    - Finish[i] = true
    - Repeat step 2
4. If all Finish[i] = true → Safe state

Example: Banker's Algorithm

System has **5 processes (P0 to P4)** and **3 resources (A=10, B=5, C=7)**.

Current Snapshot:

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

Step 1: Calculate Need Matrix

Process	**Need (Max - Allocation)**
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

Step 2: Find Safe Sequence

- P1 can be satisfied (Need ≤ Available). Finish P1 → Available becomes 5 3 2
- P3 can be satisfied. Finish P3 → Available becomes 7 4 3
- P4 can be satisfied. Finish P4 → Available becomes 7 4 5
- P0 can be satisfied. Finish P0 → Available becomes 7 5 5
- P2 can be satisfied. Finish P2 → Available becomes 10 5 7

Safe Sequence: **< P1, P3, P4, P0, P2 >**

Resource-Request Algorithm (For Multiple Instances)

When a process Pi requests resources:

1. If Request[i] ≤ Need[i], go to step 2. Else → Error.
2. If Request[i] ≤ Available → Pretend to allocate.
3. Temporarily update:
   - Available = Available - Request[i]
   - Allocation[i] = Allocation[i] + Request[i]
   - Need[i] = Need[i] - Request[i]
4. Use the **Safety Algorithm** to check if the system is still safe.
   - If yes → Grant the request.
   - If not → Rollback and make Pi wait.

Deadlock Detection:

If a system does not use deadlock prevention or avoidance, it must rely on **deadlock detection** and **recovery**.

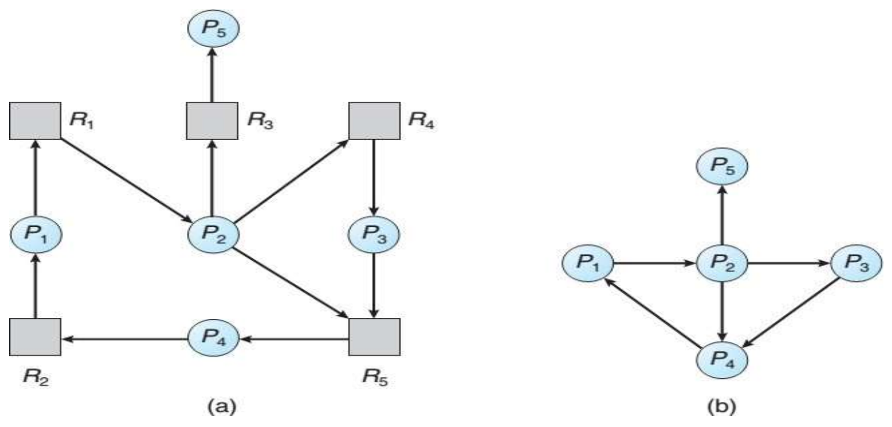
1. Deadlock Detection for Single Instance of Each Resource Type

We use a special type of graph called the **Wait-For Graph (WFG)**.

Wait-For Graph (WFG):

- Created by simplifying the Resource Allocation Graph.
- Resource nodes are removed.
- If a process Pi is waiting for a resource held by process Pj, we draw an edge Pi → Pj.
- If there is a **cycle** in the WFG, **deadlock exists**.

How to Create WFG (Based on Diagram):



From the Resource Allocation Graph (a):

- **P1** → **R1**, **R1** → **P2** → This becomes **P1** → **P2**
- **P2** → **R3**, **R3** → **P5** → This becomes **P2** → **P5**
- **P2** → **R4**, **R4** → **P3** → This becomes **P2** → **P3**
- **P4** → **R2**, **R2** → **P1** → This becomes **P4** → **P1**
- **P5** → **R5**, **R5** → **P4** → This becomes **P5** → **P4**

Thus, the resulting Wait-For Graph (b) contains:

- P1 → P2
- P2 → P5
- P2 → P3
- P4 → P1
- P5 → P4

Here, a **cycle** exists: P1 → P2 → P5 → P4 → P1 → indicating a **deadlock**.

Time Complexity:

Cycle detection in a graph has **$O(n^2)$** complexity, where n is the number of processes.

2. Deadlock Detection for Multiple Instances of Resources

Wait-For Graphs do not work if there are **multiple instances** of a resource type. Instead, we use a matrix-based **Deadlock Detection Algorithm**, similar to the Banker's Algorithm.

Data Structures:

- **Available**: Vector showing currently available instances of each resource.
- **Allocation**: Matrix showing how many instances of each resource are allocated to each process.
- **Request**: Matrix showing how many more resources each process needs.

Algorithm Steps:

1. Let Work = Available, and Finish[i] = false if Allocation[i] ≠ 0; else Finish[i] = true.
2. Find a process i such that:
    - Finish[i] == false
    - Request[i] ≤ Work

3. If such a process exists:
    - Work = Work + Allocation[i]
    - Finish[i] = true
    - Repeat step 2
4. If no such i is found and some Finish[i] == false → **System is in deadlock**.

Example:

System with 5 processes: P0 to P4 and 3 resource types A = 10, B = 5, C = 7.

Process	**Allocation**	**Request**	**Available**
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Step-by-step Detection:

- **Work = [0, 0, 0]**, Finish = [false, false, false, false, false]

Step 1: Check P0

- Request[0] = [0, 0, 0] ≤ Work → Yes
- Work = Work + Allocation[0] = [0, 1, 0], Finish[0] = true

Step 2: Check P2

- Request[2] = [0, 0, 0] ≤ Work → Yes
- Work = [3, 1, 3], Finish[2] = true

Step 3: Check P3

- Request[3] = [1, 0, 0] ≤ Work → Yes
- Work = [5, 2, 4], Finish[3] = true

Step 4: Check P1

- Request[1] = [2, 0, 2] ≤ Work → Yes
- Work = [7, 2, 4], Finish[1] = true

Step 5: Check P4

- Request[4] = [0, 0, 2] ≤ Work → Yes
- Work = [7, 2, 6], Finish[4] = true

Final Check:

All Finish[i] = true → Yes **No deadlock**

System is in a safe state.

Deadlock Recovery

When a deadlock is detected, the system must recover. There are two main approaches:

1. Process Termination

To eliminate the deadlock, processes are aborted and their resources are reclaimed.

Options:

- **Abort all deadlocked processes**: Quick and guarantees recovery, but may waste significant work.
- **Abort one process at a time**: More conservative, but requires running detection after each termination.

Factors in choosing a process to terminate:

1. Process priority
2. How long the process has executed and how much more is needed
3. Resources the process is using
4. Resources the process still needs
5. Number of processes that must be terminated
6. Whether the process is interactive or batch

2. Resource Preemption

Instead of killing a process, we take away its resources and give them to others.

Issues to handle:

1. **Selecting a victim**: Choose the process whose preemption costs the least.
2. **Rollback**: The victim process must be rolled back to a safe state (possibly restarted).
3. **Starvation**: Prevent the same process from always being chosen; use fairness policies.

Deadlock Ignorance

Some systems choose to **ignore deadlocks altogether**, assuming they are rare. This strategy is called the **Ostrich Method**, inspired by the idea of an ostrich burying its head in the sand to avoid danger.

Key Features:

- **No prevention, avoidance, or detection** is performed.
- If a deadlock occurs, the system may **crash or require manual intervention**.

Why Use It?

- Deadlocks are **extremely rare** in some environments (e.g., desktop OS).
- Overhead of handling deadlocks is not justified.
- Suitable for systems where **rebooting is fast and inexpensive**.

Drawbacks:

- May lead to **process starvation**, **system crashes**, or **loss of data**.
- Not suitable for **critical systems** (e.g., banking, aviation).

When It's Acceptable:

- Used in systems like UNIX and Windows where user can manually restart applications.
- Works well if the **cost of preventing deadlock** is higher than the **cost of failure recovery**.