1 Introduction Operating System

1.1 What's An Operating System ?

An operating system (OS) is a software program designed to manage and optimize the use of a machine's resources. It provides an easy to use **VM** virtual Machine to interact with the hardware and efficiently execute tasks. A typical machine consists of three main components:

- **Memory:** Includes random access memory (RAM), registers, and storage devices such as hard drives and solid-state drives (SSD).
- **CPU:** The central processing unit (CPU) consists of the Arithmetic Logic Unit (ALU) and the Control Unit (CU), which process and execute instructions.
- **Peripherals:** External devices such as keyboards, mice, displays, printers, and storage media.

1.2 Types Of Operating Systems

1.2.1 Batch Operating System

In a batch operating system, the user does not directly interact with the machine. Instead, the user prepares a stack of punch cards, referred to as **jobs**, which represent the user's programs. The operating system collects these jobs and **sorts them into batches** based on certain criteria, such as similar I/O requirements or resource needs. These batches are then processed sequentially, one after the other.
The main limitation of this system is that **it can only execute one job at a time** , since the CPU only loads one programe in the memory then execute it free the space and loads the next one and so on. If a job is waiting for an I/O operation (e.g., reading from a disk), the CPU enters an **idle state**, leaving its full potential and resources underutilized.

1.2.2 Multi-Programmed Operating System

Similar to a batch operating system, the user still submits jobs to the machine for execution. However, the key difference is that in a multi-programmed operating system, the CPU **loads multiple programs into memory at the same time**.
When the CPU encounters an I/O operation in Job 1, instead of going idle, it can **switch to Job 2 and continue executing instructions**. This allows the CPU to be **fully utilized**, as it can work on other jobs while waiting for I/O operations to complete. By handling multiple jobs concurrently, the system improves overall efficiency and reduces CPU idle time.

1.2.3 Multi-Processing Operating System

Similar to batch and multi-programmed operating systems, users still provide jobs for the machine to execute. However, unlike the previous systems, a multi-processing operating system has **multiple CPUs connected to the same system**, allowing for the **parallel execution** of programs. In some cases, the execution of a **single program can be split across multiple CPUs** . Like the multi-programmed operating system, this system also benefits from minimizing idle time since multiple CPUs can work on **different tasks simultaneously**. Additionally, having multiple CPUs provides **fault tolerance** if one CPU fails, the system can continue functioning with the remaining CPUs, ensuring greater reliability.

1.2.4 Distributed Operating System

In this setup, a set of machines is connected to each other through a local area network (**LAN**). The **parts of the operating system and tasks are spread among the machines** rather than being centralized. If one of the machines fails, the others will detect it and finish any incomplete tasks. The role of the failed machine will then be assigned to another machine, ensuring that the OS is always running. The crucial data is **available on all machines to prevent loss** and provide even more **robust fault tolerance**, ensuring high availability and reliability for users and applications.

1.2.5 Multi-tasking Operating System

Unlike Batch, Multi-programmed, and Multi-processing operating systems that rely on jobs and don't allow direct interaction between the user and the machine, Multi-tasking OS allows users to interact with the machine in real-time. It can be seen as an extension of the Multi-programmed OS. In a multi-tasking system, the CPU loads all programs into memory at once and switches between them rapidly, creating the illusion of parallel execution. However, in reality, the CPU is handling only one program at a time, switching between tasks so fast that it appears as if they are running simultaneously. The operating system ensures that the CPU is fully utilized and doesn't remain idle.

1.2.6 Time Sharing Operating System

In a Time Sharing Operating System, much like Multi-tasking OS the user directly interact with the machine . the CPU allocates a specific time slice to each program or task. Once the allocated time expires, the CPU switches to the next program in a cyclic manner, allowing multiple programs to share the CPU effectively. This process continues until all programs have completed their execution. The main goals of a time-sharing system are to provide interactive user experiences, ensure that all programs receive fair access to CPU resources, and minimize the time users spend waiting for responses from the system.

1.2.7 Real Time Operating System

A Real-Time Operating System (RTOS) is designed to prioritize and execute tasks within strict time constraints, known as deadlines. It loads multiple tasks into memory and manages their execution on the CPU(s). RTOS allows direct interaction with external inputs, including users or devices, ensuring that critical tasks are completed on time.
RTOS can either:

- Multi-task : by using a single CPU that rapidly switches between tasks (similar to a Multi-Tasking OS) .
- Multi-process : by using multiple CPUs to run tasks in parallel (similar to a Multi-Processing OS).

The key feature of an RTOS is that it guarantees timely execution, making it suitable for systems where **meeting deadlines is crucial** , such as in embedded systems, medical devices, or automotive control systems.

1.3 Core Functions Of An Operating System

1.4 History And Evolution Of Operating System

Chapter 2: File Management

1 Introduction

File

Central memory is volatile, so we need a way to preserve data beyond program termination and reboots. This is why we rely on files, which are resources used to store data on storage peripherals.

2 View Of File

View

- **Logical File:** How the user views the file.
- **Physical File:** How the operating system views the file.

2.1 Logical File

Logical File

This is how the user sees the file: described with a unique name , that holds set of data of a given type. Where users can perform operations such as reading, creating, inserting, and deleting data that are accessible via **access functions**(squential , direct).

2.1.1 Sequential Access

Sequential

To access the k-th element, we need to read all the elements before it until we find it.

2.1.2 Direct Access

Direct

Elements are accessed by their relative position. We simply specify the position index i and place the cursor directly at the i-th element.

2.2 Physical File

Physical File

This refers to how the operating system views the file — the implementation of how files are stored on storage peripherals. It involves **the method of allocation**, which is the arrangement of a set of physical blocks.

2.2.1 Disk Allocation

Allocation

The unit of allocation on a hard drive is a physical block, which is composed of n sectors, of total size k bytes. This means that each time the hard drive reads data, it transfers k bytes at a time.

2.2.2 Types of Allocation

Types

- Sequential Allocation
- Zone Allocation
- Chained Block Allocation
- Indexed Allocation

3 Unix Solution

Inodes

Unix utilizes inodes, which are structures containing an array of 13 pointers. These pointers either reference data blocks (physical blocks) directly or point to index blocks, which themselves are arrays of pointers to other blocks.

3.1 Levels of Indirection

Structure

The 13 pointers in an inode are organized into multiple levels of indirection to optimize for both small and large files:

- **Level 0 (Direct Pointers):** The first 10 pointers in the inode array directly reference data blocks. These are used for small files, enabling quick and efficient access.
- **Level 1 (Single Indirect Pointer):** The 11th pointer references an index block. This index block contains pointers that each reference a data block.
- **Level 2 (Double Indirect Pointer):** The 12th pointer references an index block, which in turn points to multiple Level 1 index blocks. Each Level 1 block then points to data blocks.
- **Level 3 (Triple Indirect Pointer):** The 13th pointer references an index block, which points to Level 2 index blocks. Each Level 2 block references Level 1 blocks, which ultimately point to data blocks.

Note

- **Why Use 10 Direct Pointers?** Direct pointers are included to optimize performance for small files, as they provide immediate access to data without additional lookup overhead.
- **How Does Unix Handle Performance Issues with Indirection Levels?** Unix employs a cache buffer to store frequently accessed blocks, using a Least Recently Used (LRU) strategy to maintain efficiency.

3.2 Key Formulas

Important Rules

- S_{File} : size of file
- S_{Block} : size of data block
- $\mathrm{S}_{Pointer}$: size of pointers
- $\mathrm{Nb}_{Pointer}$: number of pointer per index block

$$\mathrm{Nb}_{Pointer} = \frac{\mathrm{S}_{Block}}{\mathrm{S}_{Pointer}}$$

- Nb_{Block} : number of data block of the file

$$\mathrm{Nb}_{Block} = \frac{\mathrm{S}_{File}}{\mathrm{S}_{Block}}$$

3.3 Max File Size Calculation

Max Size

Nb_{Block_0} : number of data block of level 0

$$\text{Nb}_{Block_0} = 10$$

Nb_{Block_1} : number of data block of level 1

$$\text{Nb}_{Block_1} = \text{Nb}_{Pointer}$$

Nb_{Block_2} : number of data block of level 2

$$\text{Nb}_{Block_2} = (\text{Nb}_{Pointer})^2$$

Nb_{Block_3} : number of data block of level 3

$$\text{Nb}_{Block_3} = (\text{Nb}_{Pointer})^3$$

Max : maximumum size the inodes can hold

$$\text{Max} = (\text{Nb}_{Block_0} + \text{Nb}_{Block_1} + \text{Nb}_{Block_2} + \text{Nb}_{Block_3}) \cdot S_{Block}$$

3.4 Number Of Index Block Calculation

Number Of Index Block

- if $\text{Nb}_{Block} \leq \text{Nb}_{Block_0}$

$$\text{Nb}_{Index} = 0$$

- else $\text{Nb}_{Block} \leftarrow \text{Nb}_{Block}$ - Nb_{Block_0}
  - If $\text{Nb}_{Block} \leq \text{Nb}_{Block_1}$

$$\text{Nb}_{Index} = 1$$

  - else $\text{Nb}_{Block} \leftarrow \text{Nb}_{Block}$ - Nb_{Block_1}
    - If $\text{Nb}_{Block} \leq \text{Nb}_{Block_2}$

$$\text{Nb}_{Index} = 1 + (1 + \lceil \frac{\text{Nb}_{Block}}{\text{Nb}_{Block_1}} \rceil)$$

      - Else $\text{Nb}_{Block} \leftarrow \text{Nb}_{Block}$ - Nb_{Block_2}

$$\text{Nb}_{Index} = 1 + (1 + \text{Nb}_{Pointer}) + (1 + \lceil \frac{\text{Nb}_{Block}}{\text{Nb}_{Block_2}} \rceil + \lceil \frac{\text{Nb}_{Block}}{\text{Nb}_{Block_1}} \rceil)$$

Example

Data block size = 128 bytes , pointer size = 2 bytes , File = 1 mega bytes

1. Max file size
2. Number of index block

$$\begin{aligned}\mathrm{Nb}_{Pointer} &= \frac{\mathrm{S}_{Block}}{\mathrm{S}_{Pointer}} \\ &= \frac{128 \text{ bytes}}{2 \text{ bytes}} \\ &= \boxed{64 \text{ pointers}}\end{aligned}$$

$$\boxed{\mathrm{Nb}_{Block_0} = 10}$$

$$\mathrm{Nb}_{Block_1} = \mathrm{Nb}_{Pointer} = \boxed{64}$$

$$\mathrm{Nb}_{Block_2} = (\mathrm{Nb}_{Pointer})^2 = \boxed{64^2}$$

$$\mathrm{Nb}_{Block_3} = (\mathrm{Nb}_{Pointer})^3 = \boxed{64^3}$$

$$\begin{aligned}\mathrm{Max} &= (\mathrm{Nb}_{Block_0} + \mathrm{Nb}_{Block_1} + \mathrm{Nb}_{Block_2} + \mathrm{Nb}_{Block_3}) \cdot S_{Block} \\ &= (10 + 64 + 64^2 + 64^3) \cdot 128 \text{ bytes} \\ &= 266442 \text{ bytes} \\ &= \boxed{260.20 \text{ Kilo bytes}}\end{aligned}$$

$$\begin{aligned}\mathrm{Nb}_{Block} &= \frac{\mathrm{S}_{File}}{\mathrm{S}_{Block}} \\ &= \frac{2^{20}}{128} \\ &= \boxed{8192}\end{aligned}$$

$$\begin{gathered}
\text{Nb}_{Block} > \text{Nb}_{Block_0} \\
\text{Nb}_{Block} \leftarrow \text{Nb}_{Block} - 10 = 8182 \\
\text{Nb}_{Block} > \text{Nb}_{Block_1} \\
\text{Nb}_{Block} \leftarrow \text{Nb}_{Block} - 64 = 8118 \\
\text{Nb}_{Block} > \text{Nb}_{Block_2} \\
\text{Nb}_{Block} \leftarrow \text{Nb}_{Block} - 64^2 = 4022 \\
\text{Nb}_{Block} \leq \text{Nb}_{Block_3}
\end{gathered}$$

$$\begin{aligned}
\text{Nb}_{Index} &= 1 + (1 + \text{Nb}_{Pointer}) + (1 + \lceil \frac{\text{Nb}_{Block}}{\text{Nb}_{Block_1}} \rceil + \lceil \frac{\text{Nb}_{Block}}{\text{Nb}_{Block_2}} \rceil) \\
&= 1 + (1 + 64) + (1 + \lceil \frac{4022}{64^2} \rceil + \lceil \frac{4022}{64} \rceil) \\
&= \boxed{131}
\end{aligned}$$

Chapter 3: Communication Between Processes

1 Introduction

Introduction

When processes communicate with each other, it is called **Inter-Process Communication (IPC)**. IPC allows processes to share information and work together. There are two main scenarios:

- **Same Process**: A single program is divided into threads or split into child processes that communicate internally.
- **Different Processes**: Two separate programs running at the same time exchange information.

2 Same Process

2.1 Child Processes

Child Processes

The parent process is the first instance of the program, which is split into child processes using the primitive **fork()** that inherit all variables from parent and start executing after the fork. These processes communicate with each other and with their parent through pipes.

2.1.1 Fork

fork()

To use **fork()**, include the `<unistd.h>` header ,the **fork()** function splits the current process into two: a parent process and a child process. It returns a process ID (**pid**) with the following results:

- **fork() = -1** : Error, fork failed.
- **fork() = 0** : PID of the child process.
- **fork() > 0** : PID of the parent process.

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

int main() {

pid_t pid = fork();

if (pid == -1) {
printf("Error Fork Failed");
exit(-1);
}

else if(pid == 0){
printf("\nHey From Child Process");
exit(0);
}

else {
printf("\nHey From Parent Process");
}

return 0;
}
```

exit()

Terminates the current process with the given status. Use:

- `0` for a successful termination.
- `-1` for indicating an error.

When Forking Fails

- Exceeded the maximum number of child processes.
- Not enough memory or resources to fork.

2.1.2 Wait

wait(NULL)

To use **wait(NULL)**, include the `<sys/wait.h>` header used by parent processes , wait for its child processes to terminates

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

int main() {

pid_t pid = fork();

if (pid == -1) {
printf("Error Fork Failed");
exit(-1);
}

else if(pid == 0){
printf("\nChild Process First");
exit(0);
}

else {
wait(NULL);
printf("\nThen Parent Process\n");
}

return 0;
}
```

2.1.3 Pipe

pipe()

To use `pipe()` we need to include the header `<unistd.h>`
The `pipe()` function creates a pipe and takes an array of two integers as input:

- `pipe(descriptor)` creates the pipe and stores the file descriptors in the `descriptor` array , if pipe creation fails, `-1` is returned.

The two elements of the `descriptor` array represent the ends of the pipe:

- `read(descriptor[0], &var, sizeof(var))` reads from the pipe.
- `write(descriptor[1], &var, sizeof(var))` writes to the pipe.

To close a pipe end, use the `close(descriptor[i])` function.

Reminder

Frequent Mistakes With Pipes

- **Closing All Read Descriptors Then Trying To Write On Pipe**: Kernel doesn't find where to write the data, causing a SIGPIPE signal or an EPIPE error.
- **Not Closing All Write Descriptors Then Trying To Read From That Pipe**: Kernel waits for the pipe to be written to, as it doesn't see an EOF (End Of File) signal.

Solution

- **Leave at least one read end of the pipe open before writing to it**, so the kernel knows where to write the data.
- **Always close all write ends of a pipe before reading from it**, so the kernel knows that the pipe has reached its EOF.

Note

You can loop through `read()` using `read() > 0`, as it returns the number of bytes read. As data is read, the cursor moves until **EOF** (End of File) is reached.

Example

- **child_1**: Accepts input of n characters. Write only alphabetic characters to the pipe, converts lowercase letters to uppercase. Stops when the user inputs '0'.
- **child_2**: Prints the characters written to the pipe by **child_1**.
- **parent**: Creates the child processes and waits for them to finish.

Code Overview

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


int descriptor[2];

void child_1();

void child_2();


int main () {

 if (pipe(descriptor) == -1){
     printf("Error Pipe Creation Failed");
     exit(-1);
 }

 pid_t pid1;

 pid1 = fork();

 if(pid1 == -1){
     printf("Error Child 1 Processus Creation Failed");
     exit(-1);
  }

 else if(pid1==0){
 child_1();
 }

 else {

wait(NULL);

close(descriptor[1]);
 pid_t pid2;

 pid2 = fork();

 if(pid2 == -1){
   printf("Error Child 2 Processus Creation Failed");
   exit(-1);
 }

 else if (pid2==0){
  child_2();
 }

 else {
  wait(NULL);
  close(descriptor[0]);
  printf("\nEND EX1\n");
 }

 }

 return 0;
}
```

child_1 function :

```c
void child_1() {

 close(descriptor[0]);

 char car;

 printf("Input Char In CHILD 1\n");

 while ((car = getchar()) != '0') {
 if(car>='a' && car<='z'){
  car = toupper(car);
  write(descriptor[1],&car,1);
 }

 else if (car>='A' && car <='Z'){
  write(descriptor[1],&car,1);
 }

 }

 close(descriptor[1]);
 exit(0);

}
```

child_1 function :

```c
void child_2() {

  printf("\nPrinting Inputed Chars From CHILD1 in CHILD2 : ");

  char car;

  while(read(descriptor[0],&car,1) > 0){
   printf("%c",car);
  }

  printf("\n");

  close(descriptor[0]);
  exit(0);
}
```

- **parent**: Write 5 integers.
- **child**: Read the integers of parent and write their double.
- **parent**: Read Double integers of child.

Code Overview

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

int descriptor1[2];
int descriptor2[2];

int n;

void child_1();


void write_int();

void read_int();


int main(){

if(pipe(descriptor1) == -1){
 printf("Failed To Create Pipe 1");
 exit(-1);
}

if(pipe(descriptor2) == -1){
 printf("Failed To Create Pipe 2");
 exit(-1);
}

write_int();

pid_t pid;
pid = fork();

if(pid == -1){
 printf("Failed To Create Child 1");
 exit(-1);
}

else if (pid == 0){
child_1();
}

else{
wait(NULL);
read_int();
}

return 0;
}
```

Parent's Functions

```c
void write_int(){

 for(int i = 0 ; i<5;i++){
  printf("\nInput Number From Parent ");
  scanf("%d",&n);
  write(descriptor1[1],&n,sizeof(int));
}

 close(descriptor1[1]);
}

void read_int() {
 close(descriptor2[1]);
 printf("\n\n");
while(read(descriptor2[0],&n,sizeof(int)) > 0){
 printf("Print Number Of Child From Parent %d\n",n);
 }

 close(descriptor2[0]);
}
```

Child's Functions

```c
void child_1(){
printf("\n\n");

while(read(descriptor1[0],&n,sizeof(int)) > 0){
 printf("Print Number Of Parent From Child %d\n",n);
 n= 2*n;
 write(descriptor2[1],&n,sizeof(int));
}

close(descriptor1[0]);
close(descriptor2[1]);

exit(0);

}
```

2.2 Threads

Thread

Threads are components of the same program that share the same memory space, making them faster than multi-processing. This is because threads do not need to duplicate memory or copy data between separate processes, unlike in multi-processing, where each process has its own memory space.

Chapter 4: Synchronization

1 Introduction

Introduction

In multi-threading and multi-processing, threads and processes often share resources. If these resources are modified without proper synchronization, problems can arise. When a machine has only one CPU core, true parallelism doesn't occur; instead, the core switches between tasks very quickly. This rapid switching, known as context switching, can lead to race conditions where values are overwritten unexpectedly, causing incorrect results.

2 Critical Section

Critical Section

A critical section is a part of a program or code that accesses shared resources (like variables, memory, or hardware) and needs to be executed by only one thread or process at a time to prevent data corruption or inconsistency. Multiple threads or processes trying to access the same critical section simultaneously can lead to race conditions, where the outcome depends on the unpredictable order in which the threads run.

3 Bernstein Conditions

Bernstein Conditions

The Bernstein Conditions are a set of conditions that verify if threads or processes can truly execute at the same time without needing to switch between them or lock resources.

- **No Read-Write Conflict:** No shared resources should be written to while they are being read by another thread or process.
- **No Write-Write Conflict:** No simultaneous writes to the same resource should occur to avoid data corruption.
- **No Read-Read Conflict:** Multiple threads or processes can read the same resource without issue, as reading does not modify the data.

4 Tools To Synchronize

4.1 Semaphore

Semaphore

A semaphore is a synchronization primitive that maintains a positive integer value. This value can only be modified through its atomic operations, which are:

- **P(semaphore):** If the semaphore value is 0, the process waits until it becomes greater than 0, then decrements the semaphore.
- **V(semaphore):** Increments the value of the semaphore by 1, signaling that a resource is available.

There are two main types of semaphores:

- **Binary Semaphore:** The value is restricted to 0 or 1. It is used for locking critical sections to ensure that only one thread can access the resource at a time.
- **Mutex:** The value of the semaphore is unbounded and shared across multiple threads or processes. It not only enforces mutual exclusion, but also help with ressource management.

When To Use Semaphore

Semaphores are commonly used in the following situations:

- **Mutual Exclusion:** Ensuring that critical sections are accessed by only one thread or process at a time, preventing race conditions.
- **Signaling Between Processes:** Semaphores can be used to signal one process from another. For example, a producer can signal a consumer when new data is available.
- **Resource Management:** Managing access to a limited number of resources, such as database connections, printers, or buffers. A semaphore can keep track of how many resources are available and prevent overuse.
- **Synchronizing Threads:** In multi-threaded programs, semaphores can be used to coordinate the execution of threads. For example, you might use a semaphore to ensure that certain threads wait until others have completed their work.

4.2 Monitor

Monitor

prettyBox

When To Use Semaphore

prettyBox

Chapter 5: Deadlock

1 Processes & Resources

S

Processes in an **(OS)** use various resources such as peripherals, variables, and files, following these steps:

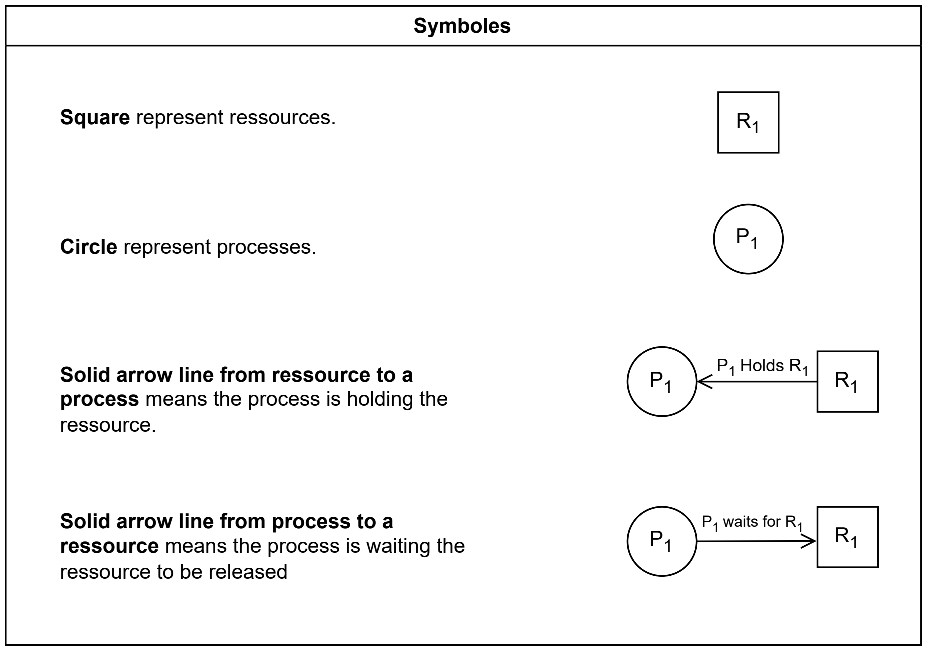
1. Request a resource, if it is unavailable the process is put on wait.
2. Hold and use the resource.
3. Release the resource once the operation is completed.

2 Deadlock

What is Deadlock?

Deadlock is a situation where a set of processes becomes blocked because each process is holding a resource and waiting for another resource held by another process to be released.

3 Resource-Process Graph



Square represent ressources.

Circle represent processes.

Solid arrow line from ressource to a process means the process is holding the ressource.

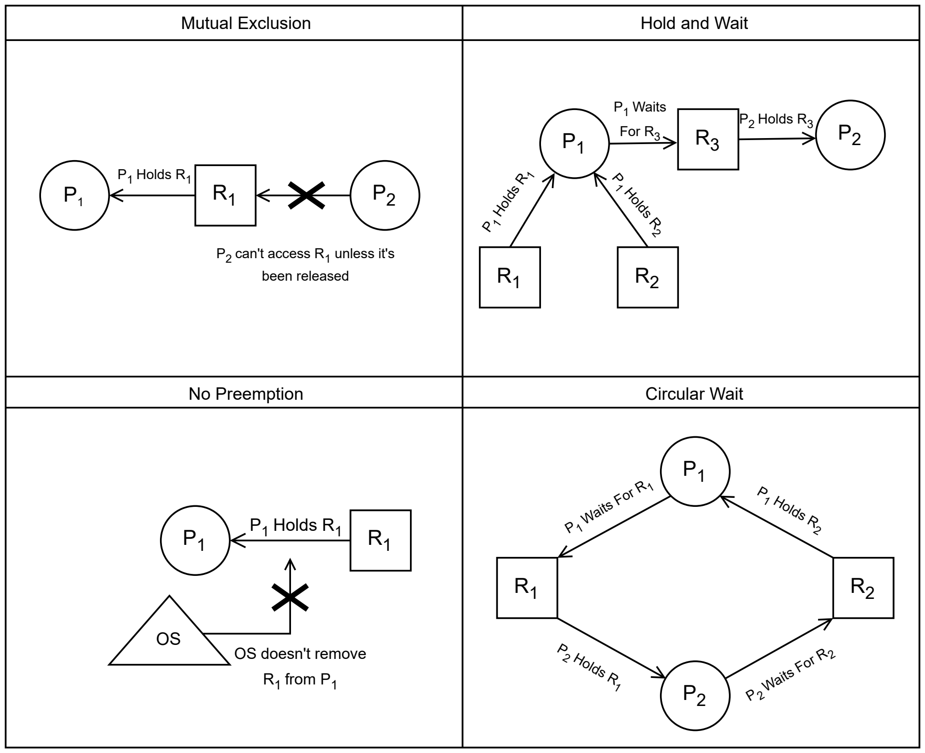
Solid arrow line from process to a ressource means the process is waiting the ressource to be released

4 Conditions for Deadlock

Coffman Conditions

Deadlock can arise if all the following conditions are met:

- **Mutual Exclusion**: Only one process can hold a resource at a time.
- **Hold and Wait**: A process holding at least one resource is waiting for additional resources held by other processes.
- **No Preemption**: The **OS** cannot forcibly remove a resource from a process.
- **Circular Wait**: There is a cycle in the graph of two or more processes waiting for each other to release ressources.



5 Handling Deadlocks

Methods

- **Ignore the Problem**: If resolving deadlocks is too costly, the **OS** may adopt a laissez-faire approach, ignoring deadlocks and requiring a system reboot if necessary.
- **Detection and Resolution**: The system detects deadlocks, often using algorithms like **DFS** on the resource allocation graph, and resolves them through one of the following strategies:
  - **Preemption**: Reassign a resource from one process to another, which may lead to issues like inconsistency or lost progress.
  - **Rollback**: Restore the system to a previously saved state and reallocate resources to avoid deadlock.
  - **Termination**: Kill one or more processes involved in the deadlock to free resources.
- **Avoidance**: Dynamically allocate resources in a way that avoids unsafe states, using the **Banker's Algorithm** to check the safety of each potential allocation before it is made.
- **Prevention**: Prevent deadlocks by ensuring at least one of Coffman's conditions cannot occur:
  - **Mutual Exclusion**: Use serialized access (e.g., queues) to eliminate contention by ensuring resources are accessed in an orderly fashion. This doesn't remove mutual exclusion, but it allows the system to control priority, avoiding chaotic competition between processes.
  - **Hold and Wait**: Require processes to request all the resources they need at the beginning. While this prevents deadlocks, it can lead to inefficiency because resources may remain idle, and a resource may need to be released to be used by another.
  - **No Preemption**: Allow preemption for certain resources by forcibly reallocating them. This approach is rarely practical due to the complexity of saving and restoring resource states.
  - **Circular Wait**: Resources are assigned increasing numerical labels. Processes must request resources in increasing order of their labels. This ensures that no cycles can form in the resource allocation graph preventing deadlock.

6 Depth First Search(DFS)

DFS for Deadlock Detection

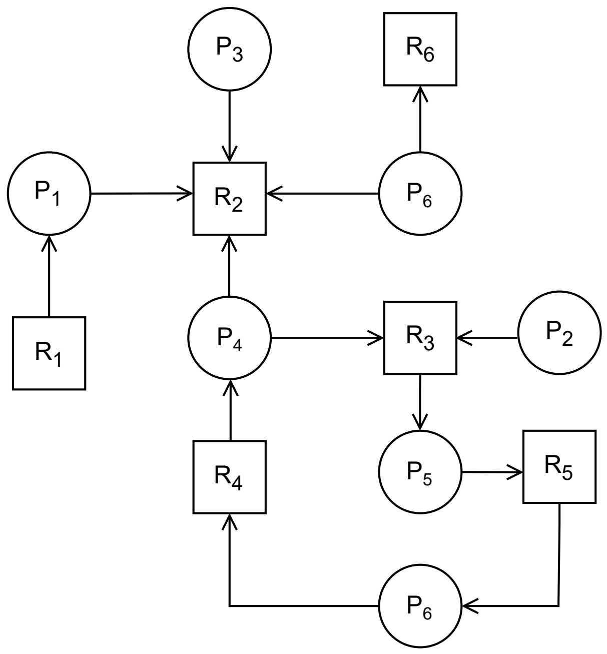
To detect a deadlock in the resource-process graph, we use a **stack** and a **visited list**. The algorithm proceeds as follows:

- Start from any node and push it onto the stack.
- Traverse through the graph by exploring neighbors of the current node.
- For each node, mark it as visited and push it onto the stack.
- If a node is encountered that is already in the stack, a **deadlock** exists, indicating a cycle in the graph.
- If a node has no unvisited neighbors (i.e., no outgoing edges), **backtrack** by popping nodes from the stack and continue exploring other unvisited nodes.
- Repeat the process until all reachable nodes are visited.
- If no cycles are found and all nodes have been explored, **no deadlock** exists.

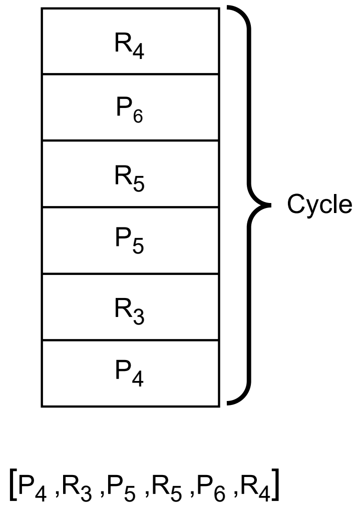
Example

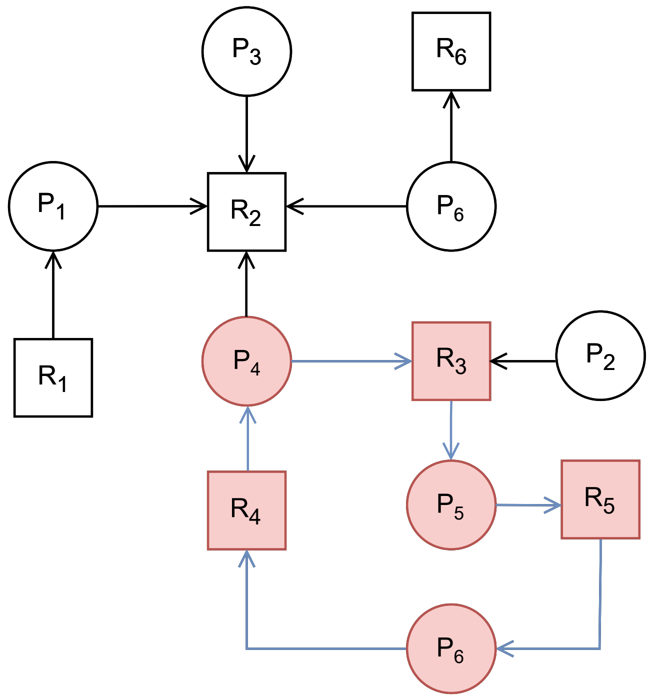
Is there a deadlock and if yes what are the processes responsible for them.

- Process P_1 holds R_1 and requests R_2.
- Process P_2 holds no resource but requests R_3.
- Process P_3 holds no resource but requests R_2.
- Process P_4 holds R_4 and requests R_2, R_3.
- Process P_5 holds R_3 and requests R_5.
- Process P_6 holds R_6 and requests R_2.
- Process P_7 holds R_5 and requests R_4.



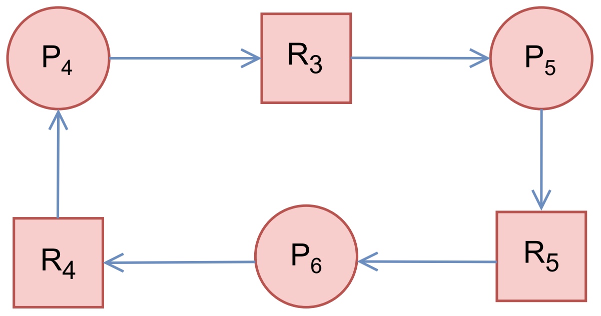
We take P_4 as the starting node





We can notice a cycle in the graph $(P_4, R_3, P_5, R_5, P_6, R_4)$ therefore There is a deadlock and the processes involved are P_4, P_5, P_6

The Cycle



7 Banker's Algorithm

Matrix & Vector Definitions

The following matrices and vectors are used in the Banker's Algorithm:

- **Max**: A matrix indicating the maximum resources each process can request.
- **C**: A matrix representing the currently allocated resources for all processes.
- **Need**: A matrix showing the remaining resources each process requires, calculated as $Need = Max - C$.
- **A**: A vector of available resources at the start of the algorithm.
- **V**: A vector of available resources at the end of the algorithm.

Algorithm Steps

To determine whether the allocation is safe:

1. **Calculate the Need Matrix:**
$$Need = Max - C$$
2. **Check Resource Availability:**
   - For each process i, check if $Need(i,:) \leq A$ (i.e., the process's resource needs are less than or equal to the currently available resources).
   - If $Need(i,:) \leq A$, mark the process as completed:
     - Set all values in $Need(i,:)$ to 0.
     - Update A to reflect the newly available resources:
$$A = A + C(i,:)$$
   - If $Need(i,:) > A$, move to the next process and repeat the check.
3. **Determine Outcome:**
   - If all processes are marked , the allocation is safe and the safe sequence is the order of the marked processes.
   - If some processes remain unmarked and no further $Need(i,:) \leq A$ condition can be satisfied, a deadlock exists.

Note

At the end of the algorithm, if the allocation is safe, the available vector A must equal the final vector V, which can be calculated as:

$$V_i = A_i + \sum C(:,i)$$

Thus, for safe allocation:
$$A = V$$

Notation

- $A(i,:)$: All elements of row i in A.
- $A(:,i)$: All elements of column i in A.

Example

1. Calculate Vector V
2. Calculate Matrix Need
3. Is it safe allocation ? if yes give the safe sequence.

	Max			
	A	B	C	D
P_0	0	0	1	2
P_1	1	7	5	0
P_2	2	3	5	6
P_3	0	6	5	2
P_4	0	6	5	6

	Allocation			
	A	B	C	D
P_0	0	0	1	2
P_1	1	0	0	0
P_2	1	3	5	4
P_3	0	6	3	2
P_4	0	0	1	4

Available			
A	B	C	D
1	5	2	0

$$\text{Max} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 7 & 5 & 0 \\ 2 & 3 & 5 & 6 \\ 0 & 6 & 5 & 2 \\ 0 & 6 & 5 & 6 \end{bmatrix} \quad , \quad C = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 \\ 1 & 3 & 5 & 4 \\ 0 & 6 & 3 & 2 \\ 0 & 0 & 1 & 4 \end{bmatrix} \quad , \quad A = \begin{bmatrix} 1 & 5 & 2 & 0 \end{bmatrix}$$

$$V_i = A_i + \sum C(:,i)$$

$$V = \begin{bmatrix} 0+1+1+0+0+1 & 0+0+3+6+0+5 & 1+0+5+3+1+2 & 2+0+4+2+4+0 \end{bmatrix}$$

$$V = \begin{bmatrix} 3 & 14 & 14 & 12 \end{bmatrix}$$

$$\text{Need} = \text{Max - C} \quad , \quad \text{Need} = \begin{bmatrix} 0-0 & 0-0 & 1-1 & 2-2 \\ 1-1 & 7-0 & 5-0 & 0-0 \\ 2-1 & 3-3 & 5-5 & 6-4 \\ 0-0 & 6-6 & 5-3 & 2-2 \\ 0-0 & 6-0 & 5-1 & 6-4 \end{bmatrix} \quad , \quad \text{Need} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{bmatrix}$$

$$\text{Need} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{bmatrix} \quad , \quad A = \begin{bmatrix} 1 & 5 & 2 & 0 \end{bmatrix}$$

$$\text{Need(1,:)} \leq A \quad \Longrightarrow \quad \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \leq \begin{bmatrix} 1 & 5 & 2 & 0 \end{bmatrix}$$

$$A \leftarrow A + C(1,:) \quad \Longrightarrow \quad A = \begin{bmatrix} 1+0 & 5+0 & 2+1 & 0+2 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 3 & 2 \end{bmatrix}$$

$$\text{Need} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{bmatrix} \quad , \quad A = \begin{bmatrix} 1 & 5 & 3 & 2 \end{bmatrix}$$

$$\text{Need(3,:)} \leq A \quad \Longrightarrow \quad \begin{bmatrix} 1 & 0 & 0 & 2 \end{bmatrix} \leq \begin{bmatrix} 1 & 5 & 3 & 2 \end{bmatrix}$$

$$A \leftarrow A + C(3,:) \quad \Longrightarrow \quad A = \begin{bmatrix} 1+1 & 5+3 & 3+5 & 2+4 \end{bmatrix} = \begin{bmatrix} 2 & 8 & 8 & 6 \end{bmatrix}$$

$$\text{Need} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{bmatrix} \quad , \quad A = \begin{bmatrix} 2 & 8 & 8 & 6 \end{bmatrix}$$

$$\text{Need(2,:)} \leq A \quad \Longrightarrow \quad \begin{bmatrix} 0 & 7 & 5 & 0 \end{bmatrix} \leq \begin{bmatrix} 2 & 8 & 8 & 6 \end{bmatrix}$$

$$A \leftarrow A + C(2,:) \quad \Longrightarrow \quad A = \begin{bmatrix} 2+1 & 8+0 & 8+0 & 6+0 \end{bmatrix} = \begin{bmatrix} 3 & 8 & 8 & 6 \end{bmatrix}$$

$$\text{Need} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{bmatrix} \quad , \quad A = \begin{bmatrix} 3 & 8 & 8 & 6 \end{bmatrix}$$

$$\text{Need(4,:)} \leq A \quad \Longrightarrow \quad \begin{bmatrix} 0 & 0 & 2 & 0 \end{bmatrix} \leq \begin{bmatrix} 3 & 8 & 8 & 6 \end{bmatrix}$$

$$A \leftarrow A + C(4,:) \quad \Longrightarrow \quad A = \begin{bmatrix} 3+0 & 8+6 & 8+3 & 6+2 \end{bmatrix} = \begin{bmatrix} 3 & 14 & 13 & 8 \end{bmatrix}$$

$$\text{Need} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 6 & 4 & 2 \end{bmatrix} \quad , \quad A = \begin{bmatrix} 3 & 14 & 13 & 8 \end{bmatrix}$$

$$\text{Need(5,:)} \leq A \quad \Longrightarrow \quad \begin{bmatrix} 0 & 6 & 4 & 2 \end{bmatrix} \leq \begin{bmatrix} 3 & 14 & 13 & 8 \end{bmatrix}$$

$$A \leftarrow A + C(5,:) \quad \Longrightarrow \quad A = \begin{bmatrix} 3+0 & 14+0 & 13+1 & 8+4 \end{bmatrix} = \begin{bmatrix} 3 & 14 & 14 & 12 \end{bmatrix}$$

$$\text{Need} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad , \quad A = \begin{bmatrix} 3 & 14 & 14 & 12 \end{bmatrix}$$

All processes have been marked therefore the allocation is safe and the safe sequence is P_0, P_2, P_1, P_3, P_4

Note

There can be many correct safe sequence for the same allocation