MIT-WPU

॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

Operating Systems

School of Computer Engineering and technology

Unit II

Process Management:

- Processes: Definition, Process Relationship, Different states of a Process, Process State transitions,
- Process Control Block (PCB), Context switching.
- Thread: Definition, Various states, Benefits of threads, Types of threads, Concept of multithreading.

Process Scheduling:

- Foundation and Scheduling objectives, Types of Schedulers, Scheduling criteria:
- CPU utilization, Throughput, Turnaround Time, Waiting Time, Response Time. Scheduling
- Algorithms: Pre-emptive and non-pre-emptive, FCFS, SIF, RR.

References

1. William Stallings, Operating System: Internals and Design Principles, Prentice Hall, ISBN-10: 0-13-380591-3, ISBN-13: 978-0-13-380591-8, 8th Edition
2. Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, WILEY,ISBN 978-1-118-06333-0, 9th Edition

Overview

1. What is a Process?
2. Process States
3. Process Description
4. Process Control

1. What is a Process

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources
- Process is an entity that consists of two essential elements.
    - program code (which may be shared with other processes that are executing the same program)
    - set of data associated with that code.

Continued..

A process is comprised of:

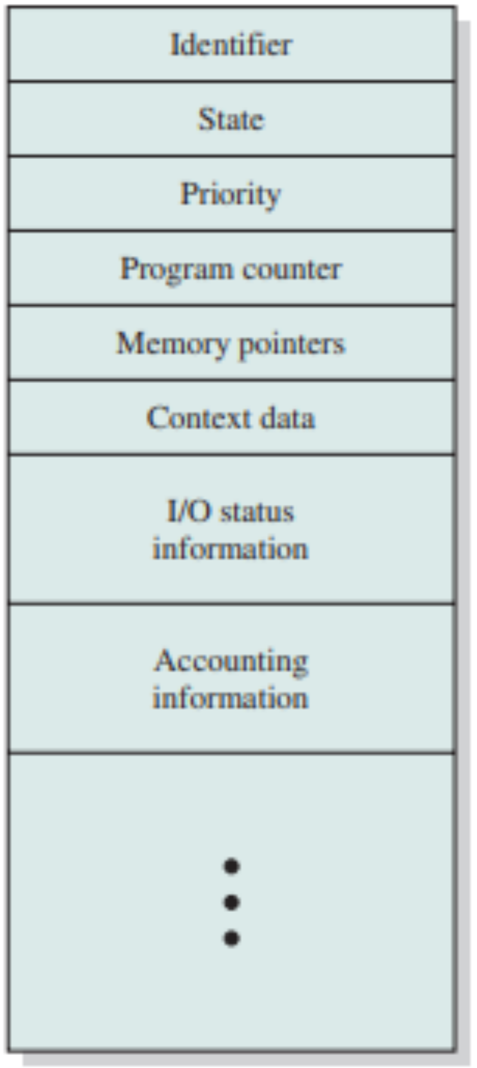
- Program code/instructions
- Data
- Stack
- A number of attributes describing the state of the process.

? When a process is mapped on to the memory it has an address space. This address space includes the code , data and stack for the process.

Process Control Block

A process can be uniquely categorized by the following elements, stored in a data structure called as **Process Control Block (PCB)**

- Identifier
- State
- Priority
- Program counter
- Memory pointers
- Context data
- I/O status information
- Accounting information



MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Operations Carried out through PCB

- **Process Scheduling:** The different information like Process priority, process state, and resources used can be used by the OS to schedule the process on the execution stack. The scheduler checks the priority and other information to set when the process will be executed.

- **Multitasking:** Resource allocation, process scheduling, and process synchronization altogether helps the OS to multitask and run different processes simultaneously.

- **Context Switching:** When context switching happens in the OS the process state is saved in the CPU register and a copy of it is stored in the PCB. When the CPU switches to another process and then switches back to that process the CPU fetches that value from the PCB and restores the previous state of the process.

- **Resources Sharing:** The PCB stores information like the resources that a process is using, such as files open and memory allocated. This information helps the OS to let a new process use the resources which are being used by any other process to execute sharing of the resources.

2. Process States (Process Execution)

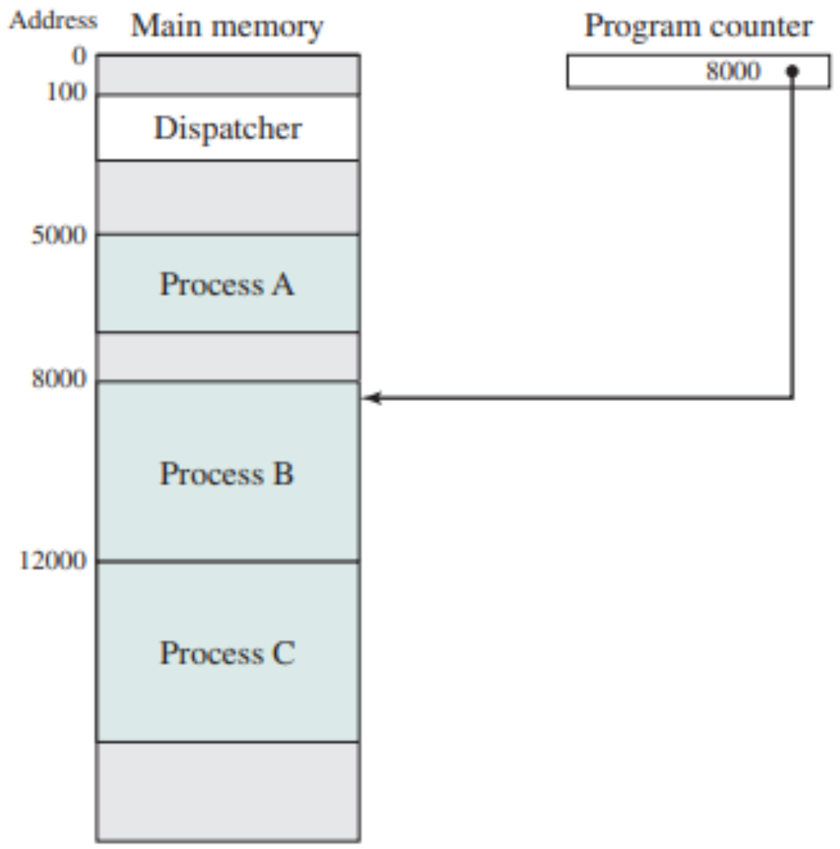
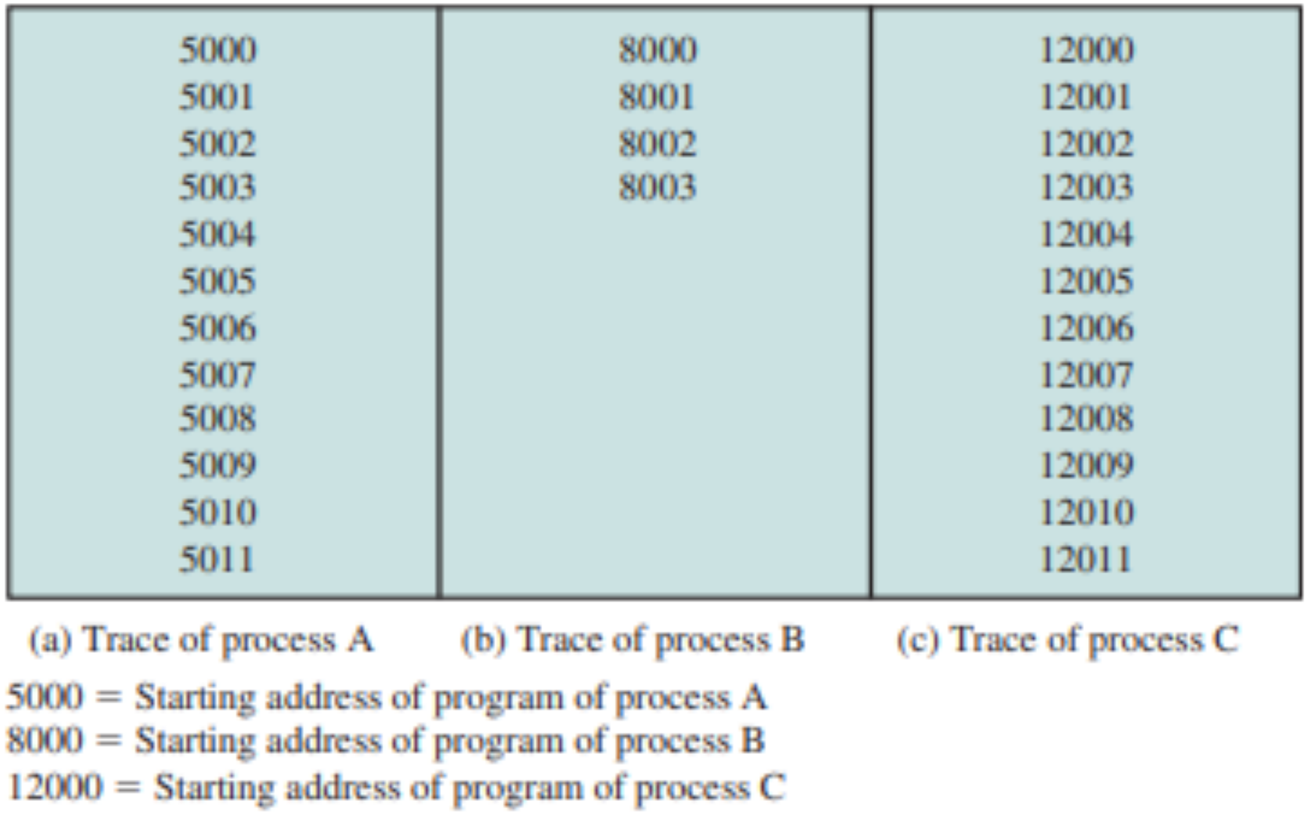


Figure 3.2 **Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13**

- Figure shows a memory layout of three processes
- all three processes are represented by programs that are fully loaded in main memory.
- there is a small dispatcher program that switches the processor from one process to another

2. Process States(Process Execution)…

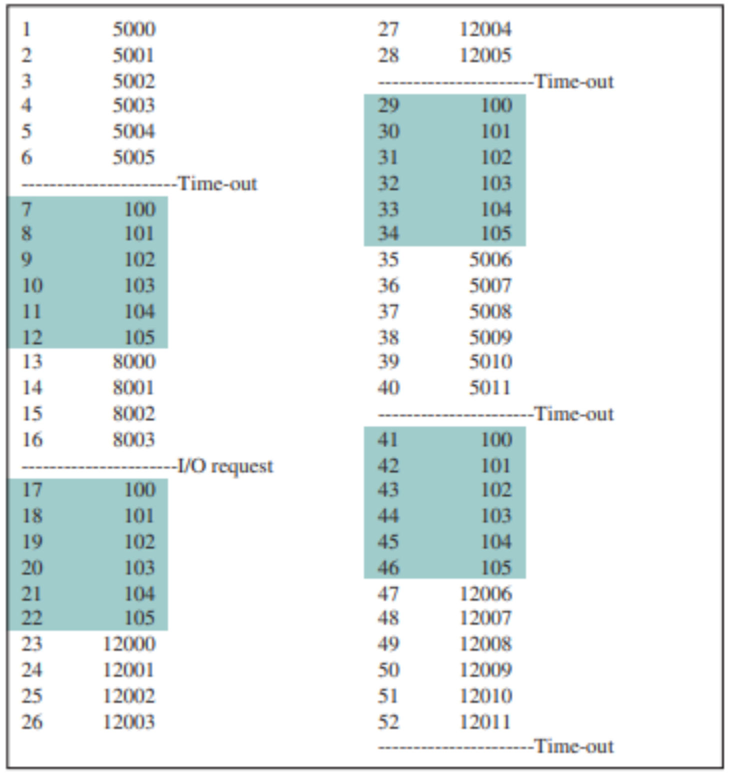


5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011
(a) Trace of process A	(b) Trace of process B	(c) Trace of process C

5000 = Starting address of program of process A
8000 = Starting address of program of process B
12000 = Starting address of program of process C

- Figure shows the traces of each of the processes during the early part of their execution.
- The first 12 instructions executed in processes A and C are shown.
- Process B executes four instructions, and we assume that the fourth instruction invokes an I/O operation for which the process must wait.

2. Process States(Process Execution)…

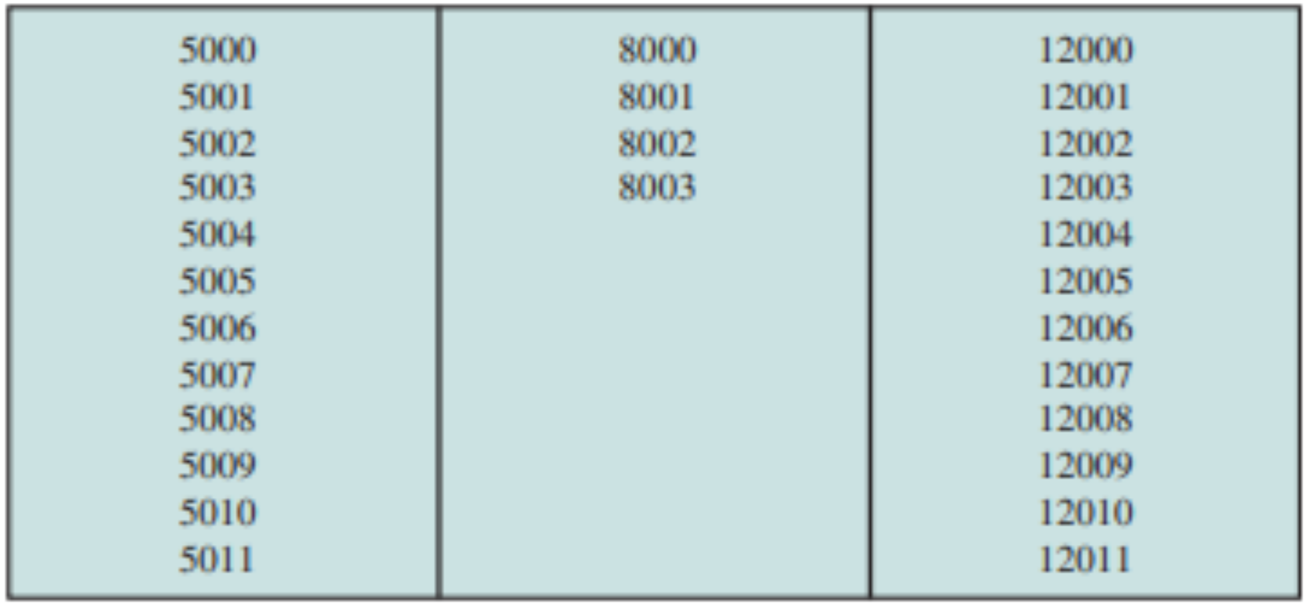


1	5000	27	12004
2	5001	28	12005
3	5002	---------------------Time-out	
4	5003	29	100
5	5004	30	101
6	5005	31	102
---------------------Time-out		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	---------------------Time-out	
16	8003	41	100
---------------------I/O request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
		---------------------Time-out	

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

- Figure shows the interleaved traces resulting from the first 52 instruction cycles

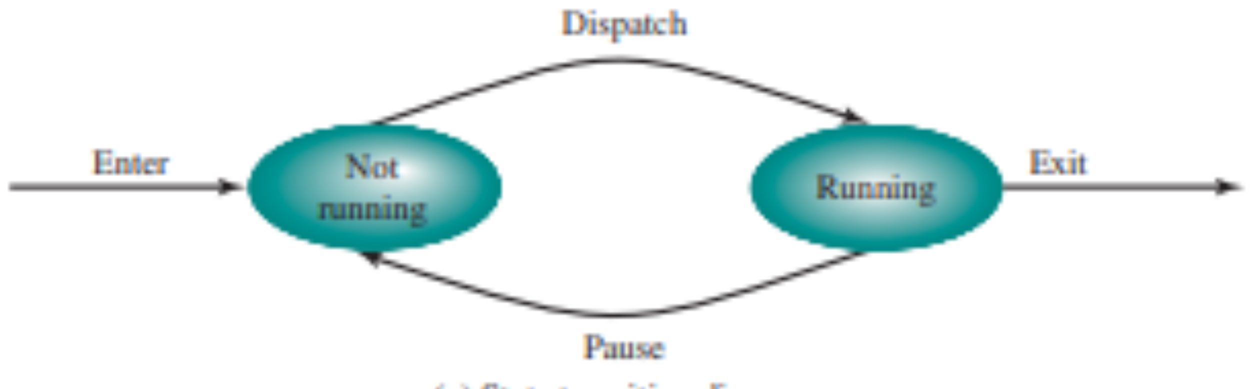


5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of process A (b) Trace of process B (c) Trace of process C

5000 = Starting address of program of process A
8000 = Starting address of program of process B
12000 = Starting address of program of process C

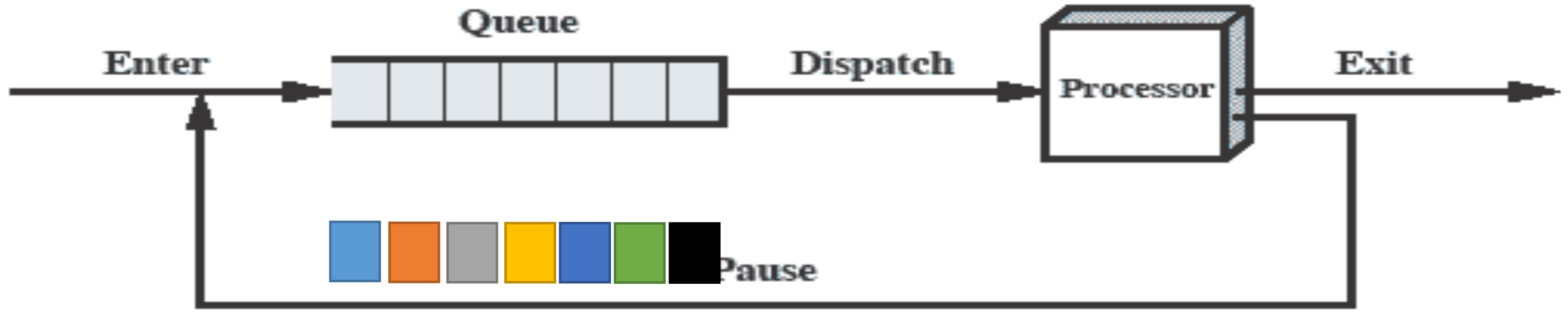
2. Process States(A Two-State Process Model)…



(a) State transition diagram

- The operating system’s principal responsibility is controlling the execution of processes
- determining the interleaving pattern for execution and allocating resources to processes.
- a process may be in one of two states: Running or Not Running, as shown in Figure
- When the OS creates a new process, it creates a process control block for the process and enters that process into the system in the Not Running state.

2. Process States(A Two-State Process Model)…



(b) Queuing diagram

- Processes moved by the dispatcher of the OS to the CPU then back to the queue until the task is completed
- Processes that are not running must be kept in some sort of queue, waiting their turn to execute.
- There is a single queue in which each entry is a pointer to the process control block of a particular process.
- the queue may consist of a linked list of data blocks, in which each block represents one process

Process Creation(A Five-State Model)

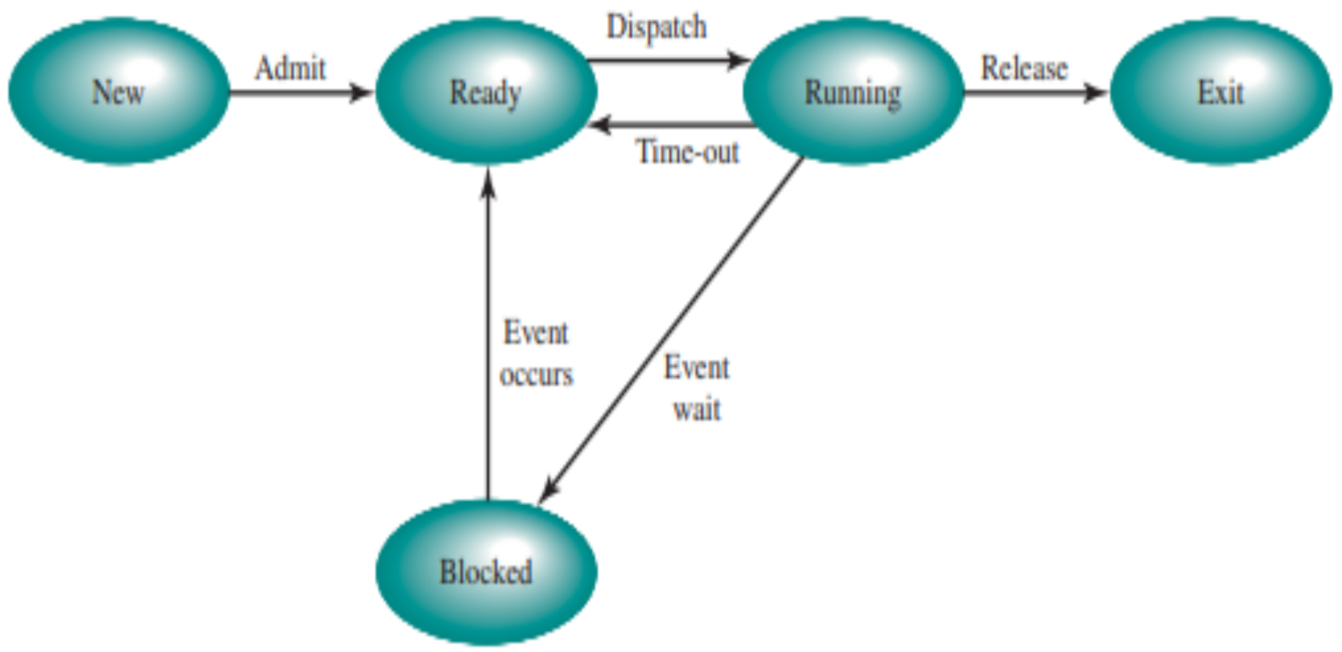
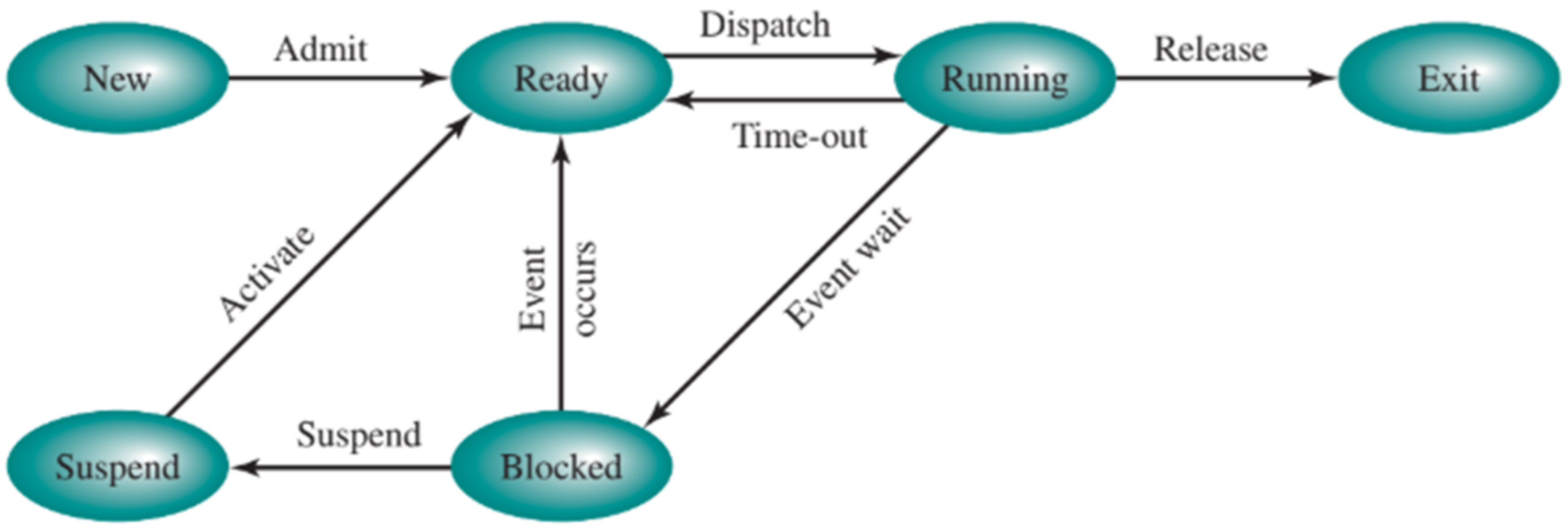


Figure 3.6 **Five-State Process Model**

The possible transitions that leads to types of events are as below:

New → Ready
After creation, the process enters the ready queue.
Ready → Running
Scheduler dispatches the process to the CPU.
Running → Blocked
Process requests I/O and must wait.
Blocked → Ready
I/O completes; process is now ready to resume.
Running → Ready
Time slice expires or preemption occurs.
Running → Exit
Process finishes or is forcefully terminated.

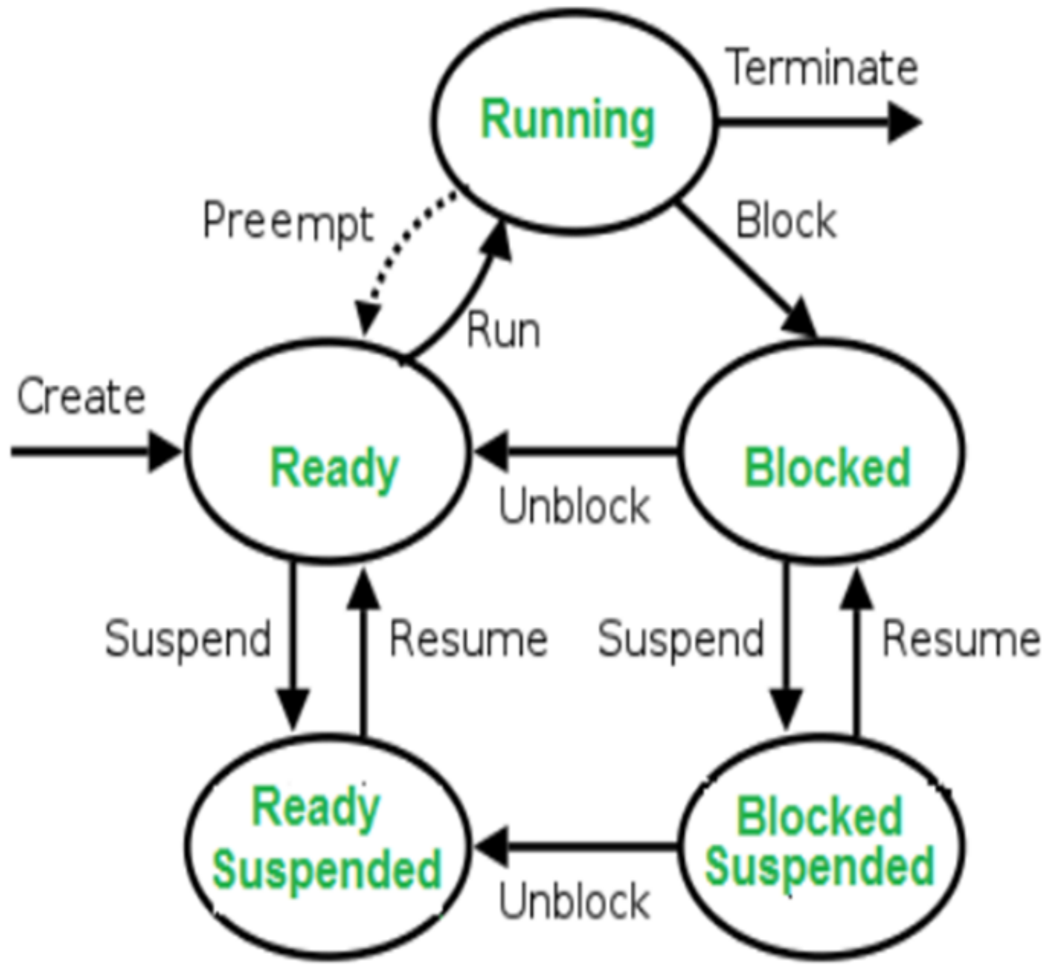
Process Creation(A Seven State Model)

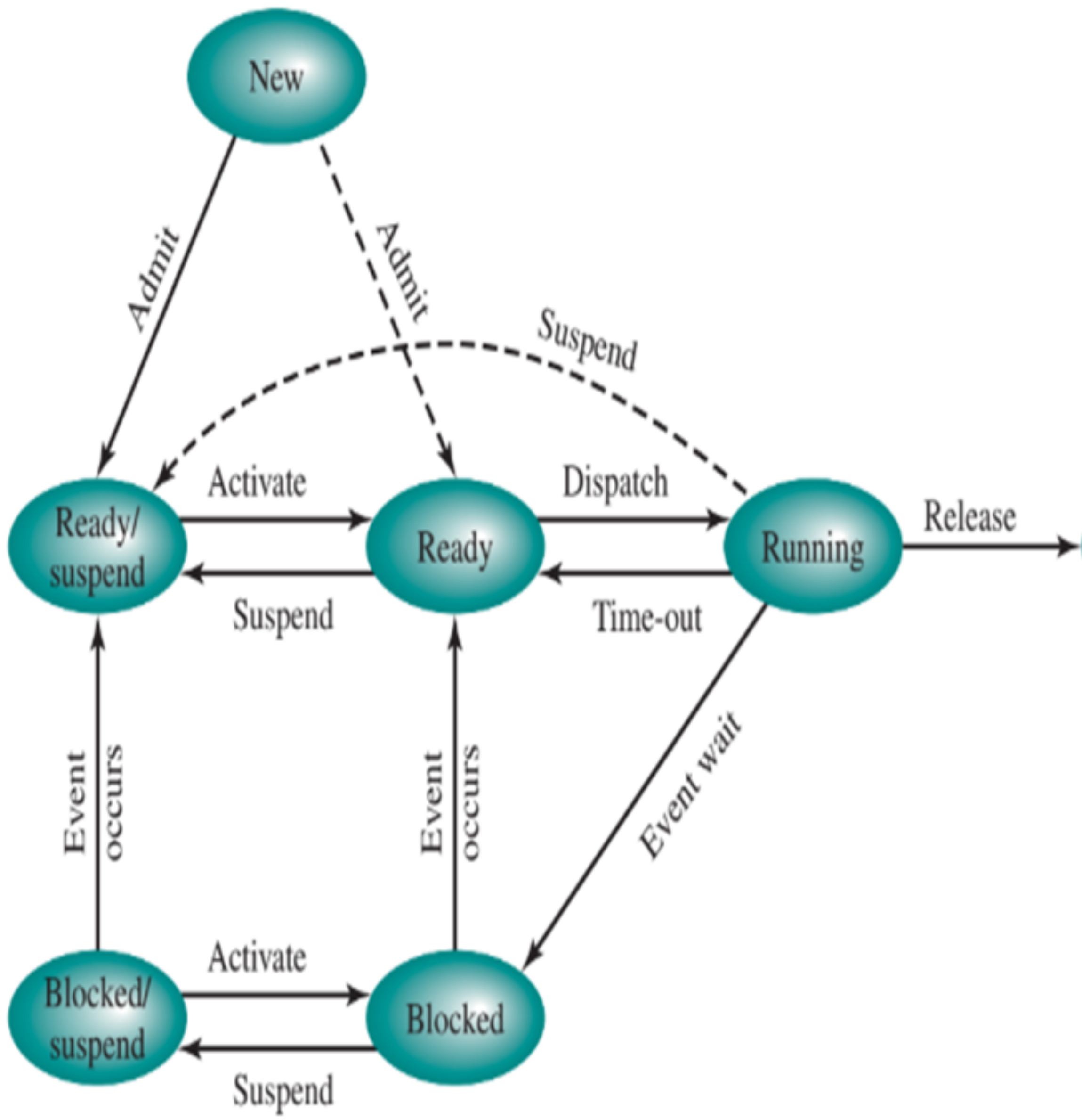


- **Ready:** The process is in main memory and available for execution
- **Blocked:** The process is in main memory and awaiting an event.

- **Blocked/Suspend:** The process is in secondary memory and awaiting an event.
- **Ready/Suspend:** The process is in secondary memory but is available for execution as soon as it is loaded into main memory.

Seven State Model

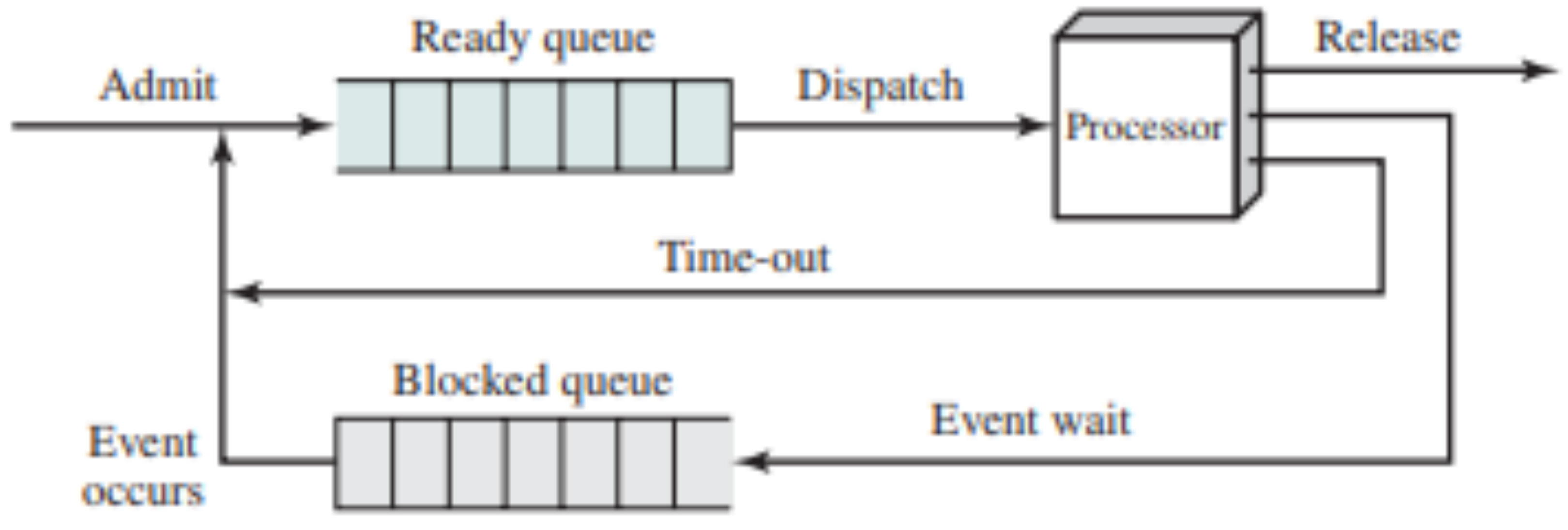




(b) With two suspend states

1. New
•The process is being created.
•Memory, I/O, and PCB are being allocated.
2. Ready
•The process is loaded in memory and waiting for CPU allocation.
•It is **ready to run** but waiting its turn on the CPU.
3. Running
•The process is **actively executing** on the CPU.
•Only one process can be in the running state per core at a time.
4. Waiting / Blocked
•The process **cannot proceed until some I/O or event completes**.
•It waits for external resources like disk, printer, or input.
5. Terminated /Exit
•The process has **finished execution** or was aborted.
•OS cleans up resources and removes it from the process table.

Queueing Model for (A Five-State Model)



(a) Single blocked queue

- When a running process is removed from execution, it is either terminated or placed in the Ready or Blocked queue, depending on the circumstances.
- Finally, when an event occurs, any process in the Blocked queue that has been waiting on that event only is moved to the Ready queue.

Process Creation

When a new process is created , the following happens :-

1.Allocates space to the process in memory.

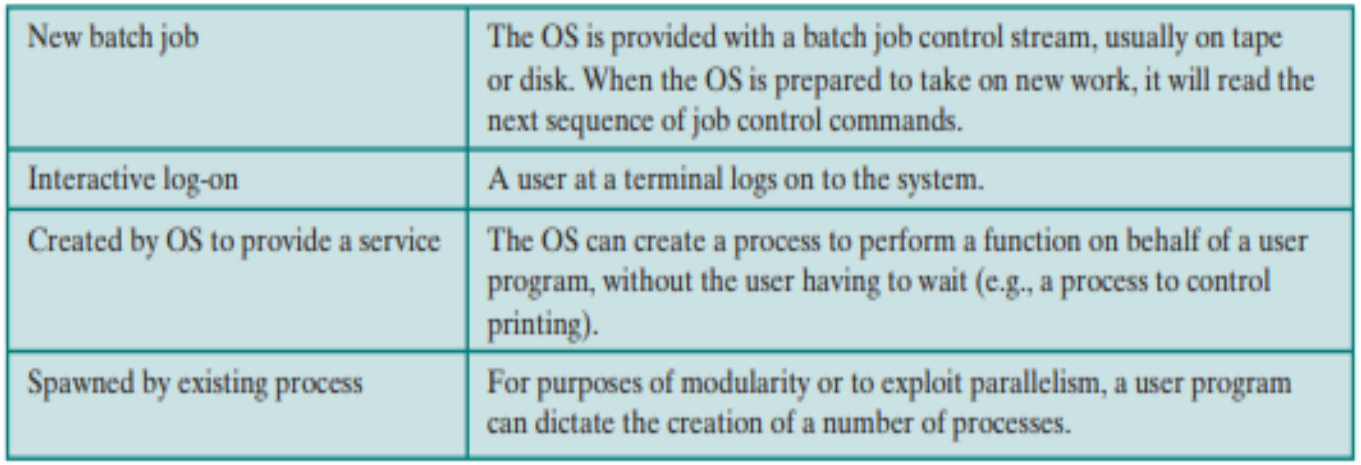
2.Assign a unique process ID to the process

3. A Process control block (PCB) gets associated with the process.

4. OS maintains pointers to each process's PCB in a process table so that it can access the PCB quickly.

5.OS maintains accounting file on each process

Table 3.1 Reasons for Process Creation



New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive log-on	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

- When one process spawns another, the former is referred to as the parent process, and the spawned process is referred to as the child process.

Process Termination

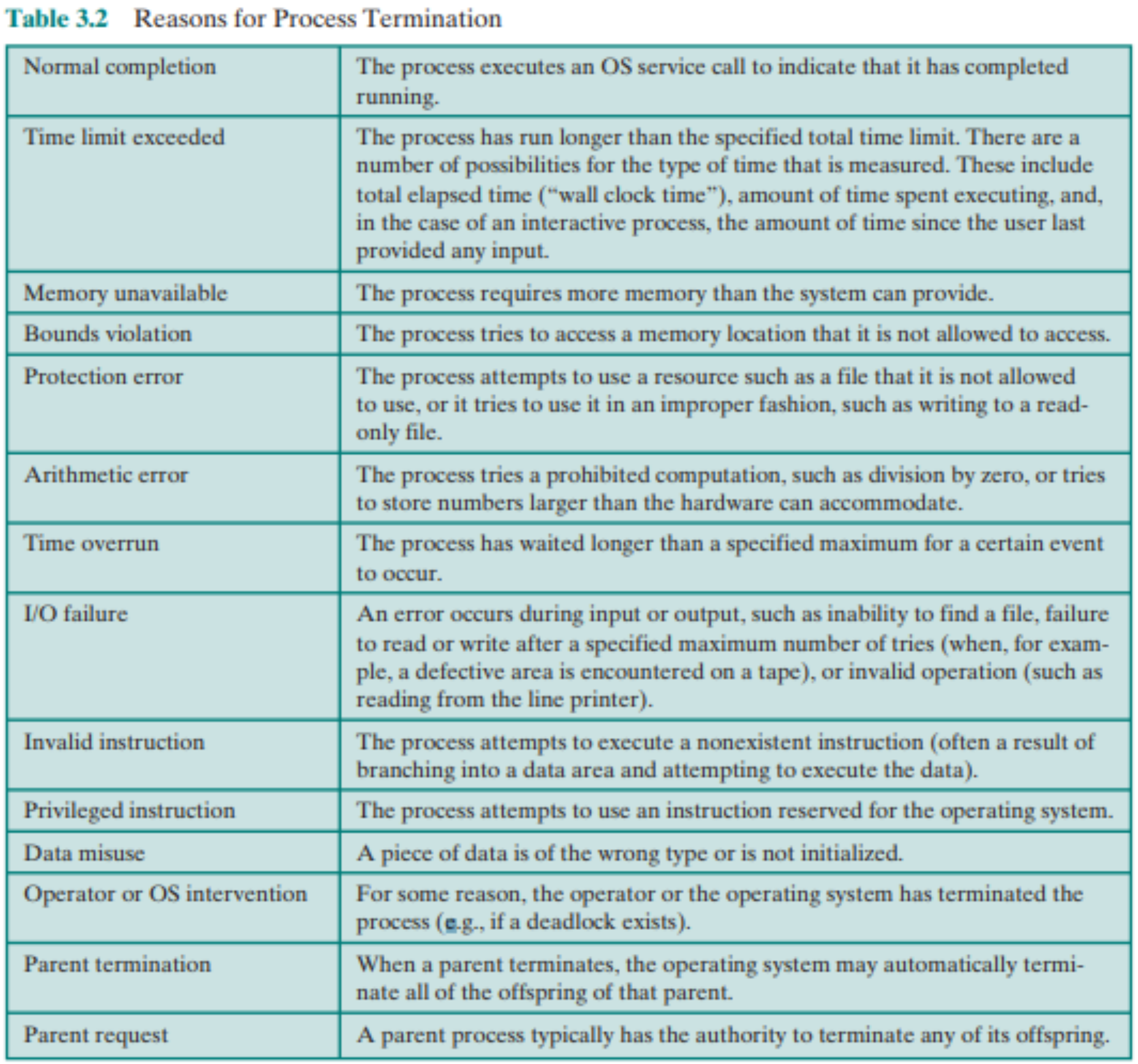


Table 3.2 Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Process Control

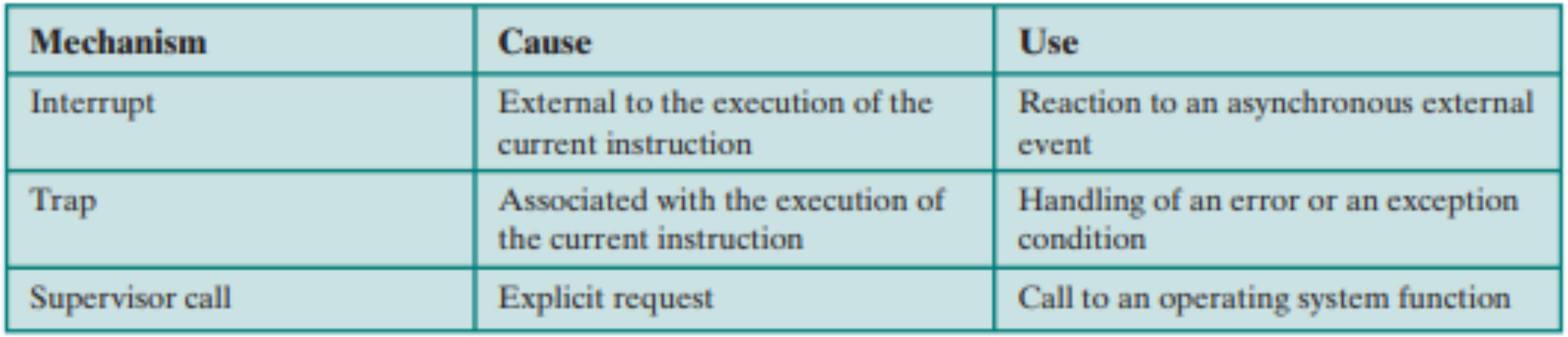
Modes of Execution

- Most processors support at least two modes of execution.
- Certain instructions can only be executed in the more-privileged mode.
- These would include reading or altering a control register, such as the program status word; primitive I/O instructions; and instructions that relate to memory management.
- In addition, certain regions of memory can only be accessed in the more-privileged mode.
- The less-privileged mode is often referred to as the user mode, because user programs typically would execute in this mode.
- The more-privileged mode is referred to as the system mode, control mode, or kernel mode.

Process Control

Process Switching

- A running process is interrupted and the OS assigns another process to the Running state and turns control over to that process
- A process switch may occur any time that the OS has gained control from the currently running process



Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

Process Control

- **Mode switching**

There are two modes of operations :-

1. **Kernel mode** (Privileged mode): It can access its own data-structures as well as the user mode data structures.

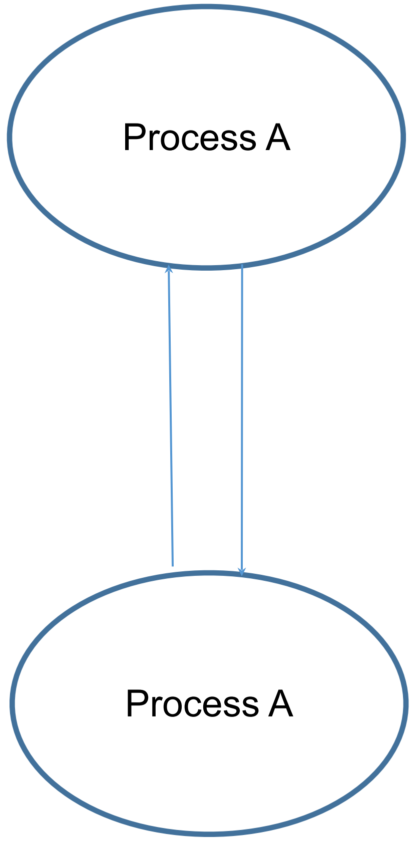
2. User mode : It can access only the user mode data structures.

- User programs initially work in the User mode.
- Whenever a system call is encountered the control switches to the kernel mode.
- All interrupts are serviced in the Kernel mode.
- When the system call is serviced the control returns back to the user mode

Process Control(Mode switching)

- Kernel mode

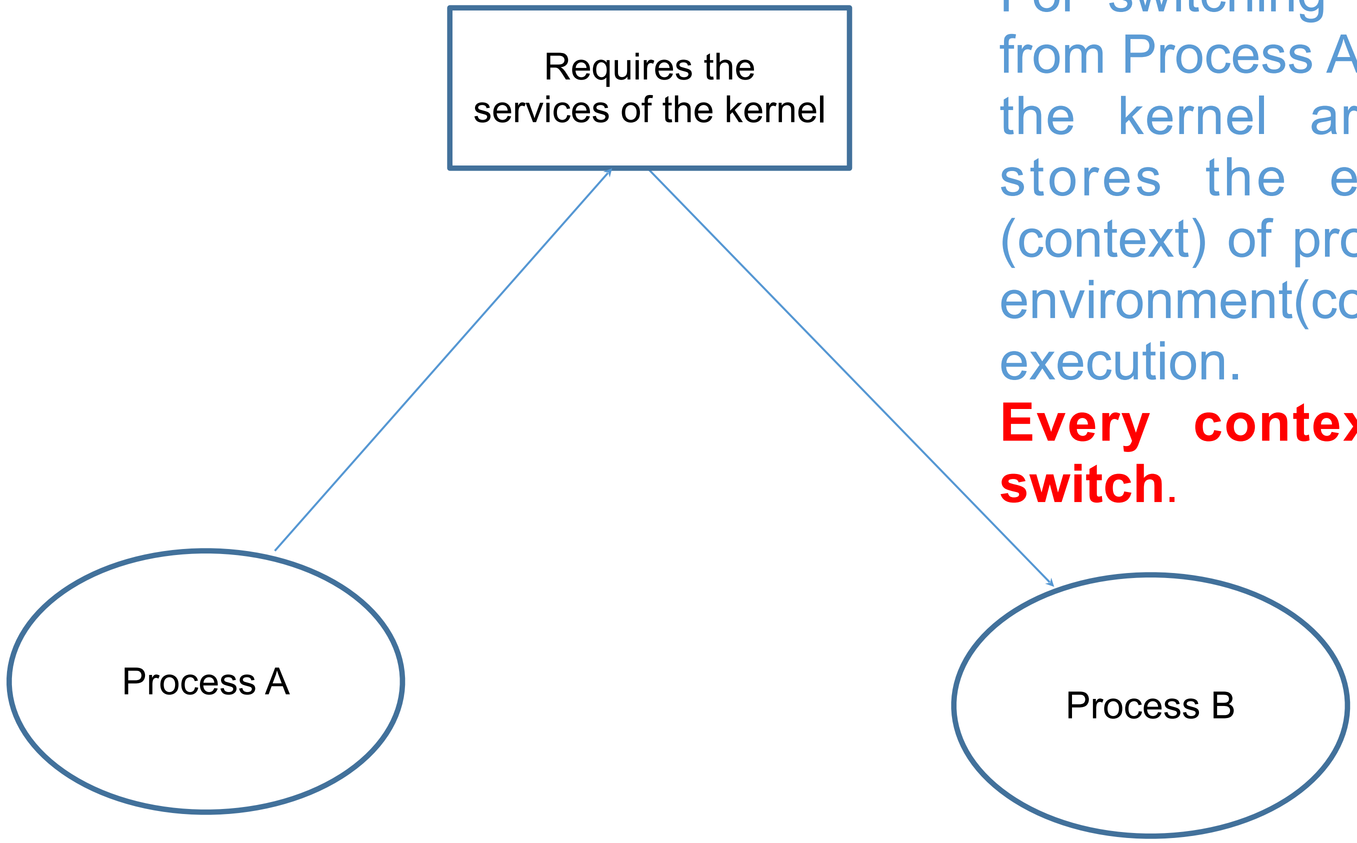
- User mode



Process A was executing in User mode and has now switched to the Kernel mode due to some system call. This is mode switch.

As, every context switch is a mode switch, every mode switch may/may not be a context switch.

Process Control - context switch



- **Kernel mode**
- **User mode**

For switching of context (environment) from Process A to process B, services of the kernel are needed. The kernel stores the execution environment (context) of process A and retrieves the environment(context) for process B for execution.

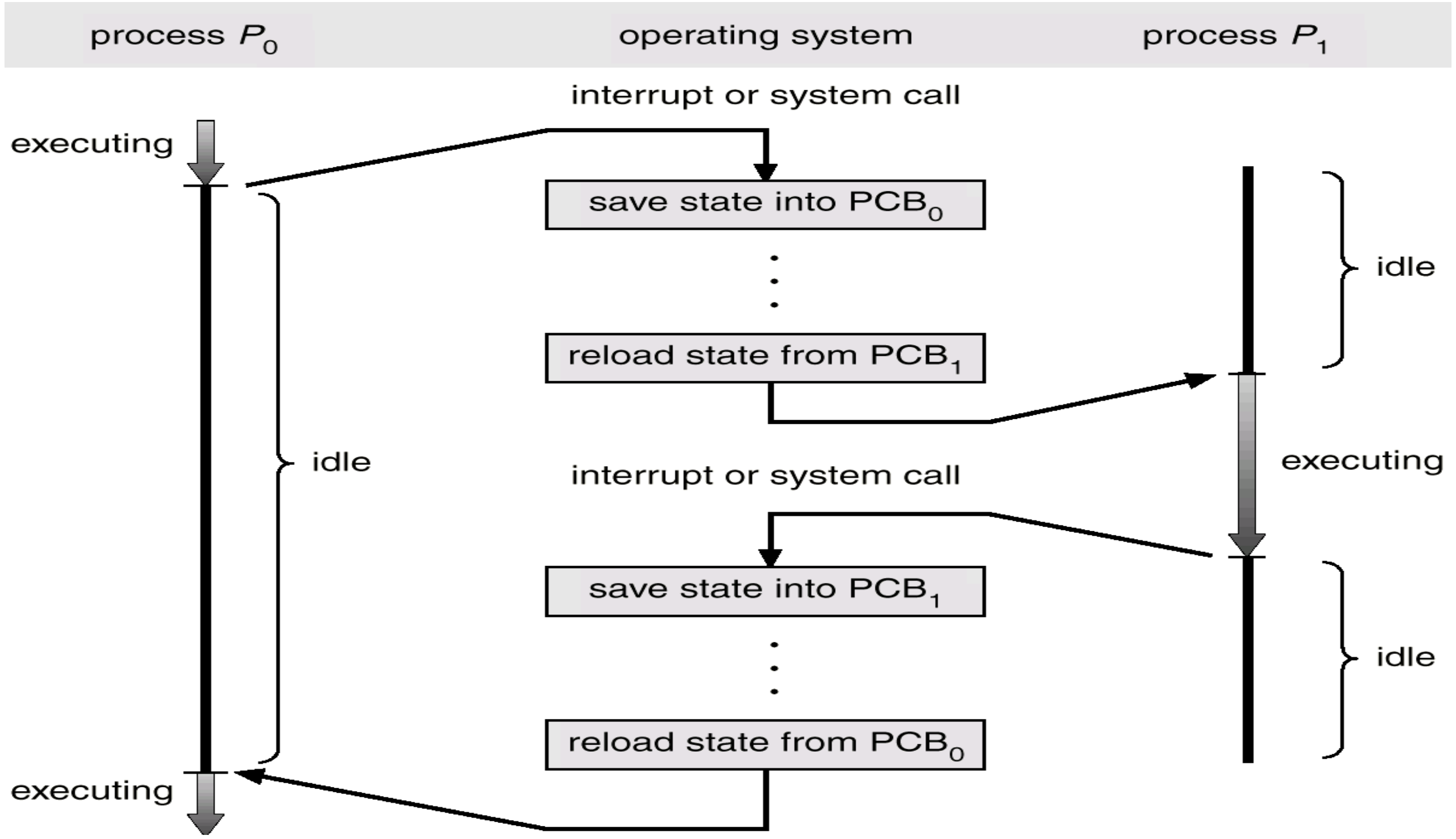
Every context switch is a mode switch.

Process Control - context switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.

- Context-switch time is overhead; the system does not do the useful work while switching.

Process Control - context switch



CPU Switch From Process to Process

Process Control

Change of process state

- 1. Save the context of the processor, including program counter and other registers.
- 2. Update the process control block of the process that is currently in the Running state. This includes changing the state of the process to one of the other states (Ready; Blocked; Ready/Suspend; or Exit).
- 3. Move the process control block of this process to the appropriate queue (Ready; Blocked on Event i ; Ready/Suspend).
- 4. Select another process for execution
- 5. Update the process control block of the process
- 6. Update memory management data structures.
- 7. Restore the context of the processor

Process Relationships

•Parent and Child Processes
•Process Tree Structure
•Examples:
•Init process in Linux
•Forking in Unix

Types of Process Relationships

- Independent Process
- Cooperating Process (shared data, IPC)

Independent Process

A process that does **not** share data or resources with any other process.
Characteristics:
- Cannot affect or be affected by other processes.
- No shared memory or communication.

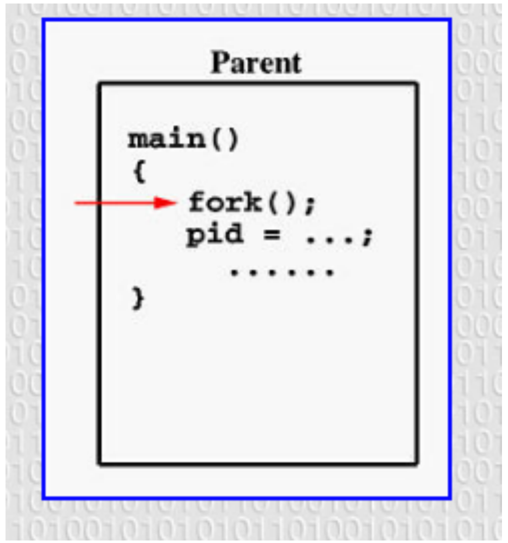
Example:
- A calculator app running separately from a text editor.

Cooperating Process

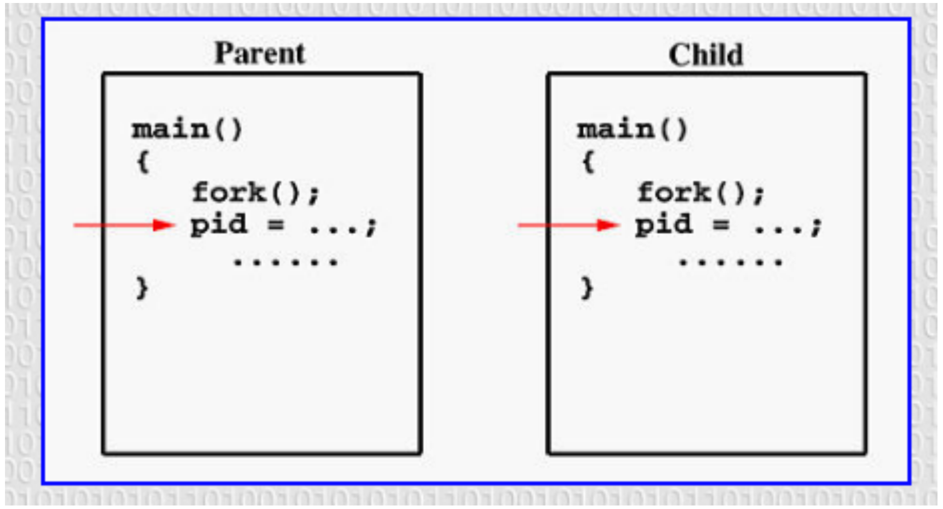
A process that shares data or communicates with other processes.
Characteristics:
Can share data and resources
Requires inter-process communication (IPC)
Common in multitasking systems
Example:
A group of people collaborating on a Google Doc. Each person is a process that reads/writes, and changes are reflected live to others.

Process Creation - Fork

- Purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller.
- After a process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call.
- To distinguish the parent from the child, the returned value of **fork()** can be used:
  - **fork()** returns a **negative value**, the creation of a child process was unsuccessful.
  - **fork()** returns **a zero to the** newly created child process.
  - **fork()** returns a **positive value**, the ***process ID*** of the child process to the parent
- Returned process ID is of type **pid_t** defined in **sys/types.h**
- Process can use function **getpid()** to retrieve the process ID assigned to this process
- **Unix/Linux will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.**

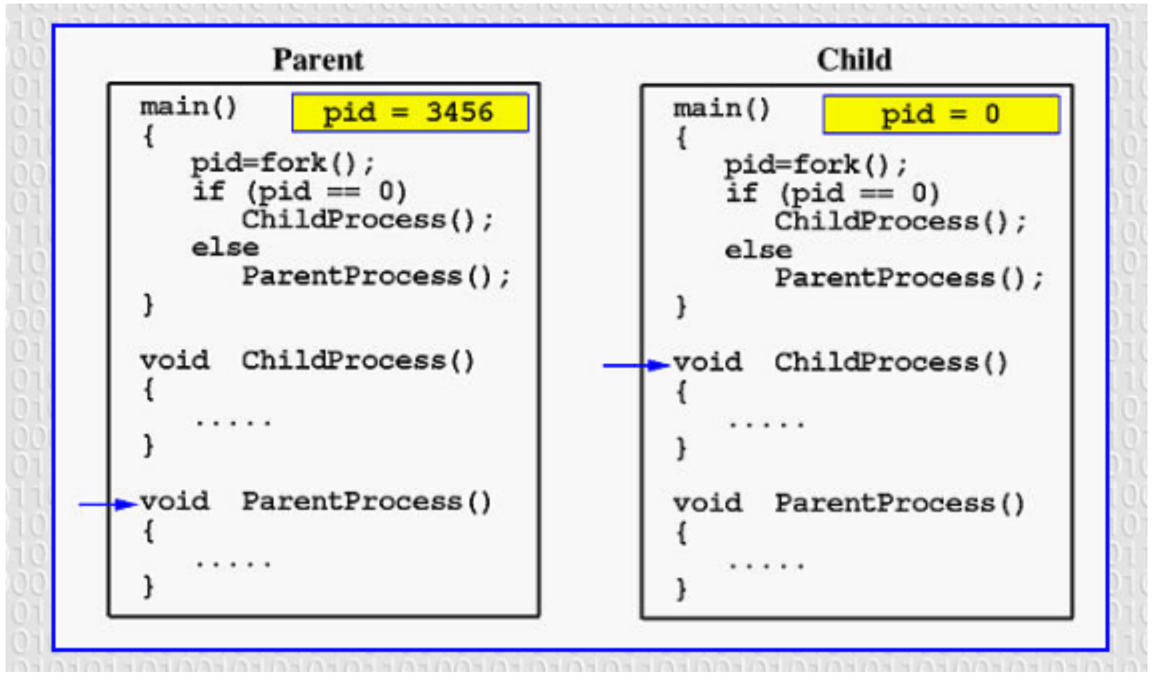


```
main()
{
    fork();
    pid = ...;
      ......
}
```



```
main()
{
    fork();
    pid = ...;
      ......
}
```

```
main()
{
    fork();
    pid = ...;
      ......
}
```



Parent:

```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess()
{
    .....
}

void  ParentProcess()
{
    .....
}
```

Child:

```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess()
{
    .....
}

void  ParentProcess()
{
    .....
}
```

Process Management—creating a Process in Unix(example)

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
/* pid_t : is a long integer type data type...prototype in types.h */
pid_t num_pid;
main(){
num_pid=fork(); /* return value of fork */
if(num_pid==0) {/* this is child process */
 printf("this is the child process id %d\n",getpid());
}
if(num_pid>0){ /* this is parent process */
		printf("this is the parent process id %d",getpid());
}
exit();
}
```

```
$cc program.c
$ ./a.out
   this is the child process id 1001
   this is the parent process id 1000
```

Process Creation – Parent/Child

- **Parent Process**
- The parent process has its unique ID
- The parent process creates a child process by giving a call to fork() system call

- **Child Process**
- The child process has its unique ID
- The child process gets created due to the fork() system call

?The child is initially a duplication of the parent process.
?The child and parent do exist in separate address spaces.
?The child inherits all data structures

A client-server application can be built using the parent-child concept.
Any IPC mechanism can be implemented using the parent child relationship.

fork()

- #include <stdio.h>
- #include <sys/types.h>
- #include <unistd.h>
- int main()
- {   fork();
-             printf("Hello world!\n");
-             return 0;
- }

fork()

```
• #include <stdio.h>
• #include <sys/types.h>
• #include <unistd.h>
• int main()
• {          fork();
•            printf("Hello world!\n");
•            return 0;
• }
```

Output:

```
Hello world!
Hello world!
```

fork()

- #include <stdio.h>
- #include <sys/types.h>
- int main()
- {
-     fork();
-     fork();
-     fork();
-     printf("hello\n");
-     return 0;
- }

fork()

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
        fork();
        fork();
        fork();
        printf("hello\n");
        return 0;
}
```

Output:

```
hello
hello
hello
hello
hello
hello
hello
hello
```

here n = 3,
$2^3 = 8$

If we want to represent the relationship between the processes as a tree hierarchy it would be the following:

The main process: P0
Processes created by the 1st fork: P1
Processes created by the 2nd fork: P2, P3
Processes created by the 3rd fork: P4, P5, P6, P7

```
        P0
     /  |  \
    P1  P4  P2
   / \        \
  P3  P6       P5
 /
P7
```

fork()

```
void forkexample()
{           if (fork() == 0)
                        printf("Hello from Child!\n");
            else
                        printf("Hello from Parent!\n");
}
int main()
{
            forkexample();
            return 0;
}
```

fork()

```
• void forkexample()
• {           if (fork() == 0)
•                       printf("Hello from Child!\n");
•             else
•                       printf("Hello from Parent!\n");
• }
• int main()
• {
•             forkexample();
•             return 0;
• }
```

```
Hello from Child!
Hello from Parent!
(or)
Hello from Parent!
Hello from Child!
```

.

Here, two outputs are possible because the parent process and child process are running concurrently. So we don't know whether the OS will first give control to the parent process or the child process.

fork()

```
• void forkexample()
• {           int x = 1;
•             if (fork() == 0)
•                        printf("Child has x = %d\n", ++x);
•             else
•                        printf("Parent has x = %d\n", --x);
• }
• int main()
• {           forkexample();
•             return 0;
• }
```

fork()

```
• void forkexample()
• {         int x = 1;
•           if (fork() == 0)
•                   printf("Child has x = %d\n", ++x);
•           else
•                   printf("Parent has x = %d\n", --x);
• }
• int main()
• {         forkexample();
•           return 0;
• }
```

```
Parent has x = 0
Child has x = 2
(or)
Child has x = 2
Parent has x = 0
```

Here, variable change in one process does not affect other process because data/state of two processes are different. And also parent and child run simultaneously, so two outputs are possible.

Inter-process communication (IPC)

Processes executing concurrently in the operating system may be either **independent processes or cooperating processes.**

A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.

A process is cooperating if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is a cooperating process.

Reasons for providing an environment that allows process cooperation

Information sharing

Several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

•**Computation speedup**

If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing cores.

•Modularity

We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads

•**Convenience**

Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

IPC for Cooperating processes

Cooperating processes require an interprocess communication(IPC) mechanism that will allow them to exchange data and information.

Two fundamental models of interprocess communication are

1. shared memory
2. message passing

Shared Memory

- In the shared memory model a region of memory that is shared by cooperating processes is established.
- Processes can then exchange information by reading and writing data to the shared region
- Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls

Message Passing

In the message passing model, communication takes place by means of messages exchanged between the cooperating processes

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.

Message passing is also easier to implement in a distributed system than shared memory.(Although there are systems that provide distributed shared memory, we do not consider them in this text.)

Inter-process communication (IPC)

There are several mechanisms for *Inter-Process Communication* (IPC)

- Signals
- FIFOS (named pipes)
- Pipes
- Sockets
- Message passing
- Shared memory
- Semaphores

Programming with pipes

- ? Within programs a pipe is created using a system call named **pipe.**
- ? This system call would create a pipe for one-way communication.
- ? This call would return zero on success and -1 in case of failure.
- ? If successful, this call returns two files descriptors:

Usage

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Programming with pipes

- Usage

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- filesdes is a two-integer array that will hold the file descriptors that will identify the pipe If successful,
- filedes[0] will be open for reading from the pipe and
- filedes[1] will be open for writing down it.
- pipe can fail (returns -1) if it cannot obtain the file descriptors (exceeds user-limit or kernel-limit).

Programming with pipes

Here's an extremely important point: a read from a pipe only gives end-of-file if *all* file descriptors for the write end of the pipe have been closed.

Thus, after a fork, whichever process is intending to do the reading (and thus not the writing) had best close the write end of the pipe.

```
#include<unistd.h>
close(filedes)
```

The above system call closing already opened file descriptor. This implies the file is no longer in use and resources associated can be reused by any other process.

Programming with pipes

write(filedes[1], string, MAX);

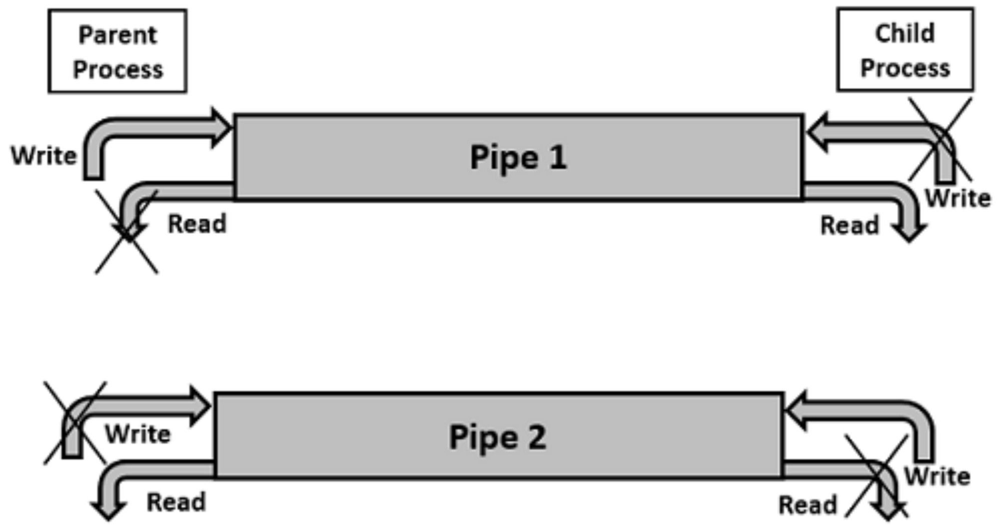
- The above system call is to write to the specified file with arguments of the file descriptor fd, string and the size of buffer.
- The file descriptor id is to identify the respective file, which is returned after calling pipe() system call.
- The file needs to be opened before writing to the file. It automatically opens in case of calling pipe() system call.
- This call would return the number of bytes written (or zero in case nothing is written) on success and -1 in case of failure. Proper error number is set in case of failure.

Programming with pipes

read(filedes[0], line, MAX);

- The above system call is to read from the specified file with arguments of file descriptor fd, string and the size of buffer.

Two-way Communication Using Pipes



- Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both.
- However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes. Two pipes are required to establish two-way communication.

Lab Assignment
Title: Process Management (Inter-process Communication)

- ## Problem statement:
  - Write a program to Implement Pipe and / Shared Memory Concept.

- ## Algorithm
  - Create pipe using pipe system call in parent process.
  - Create a child process using fork, which returns zero to child process and some nonzero positive integer to the parent process.
  - Writes the message and transfer them through pipe in Parent process.
  - Receive the message and read the message in Child process.
  - Print the same message on the console.
  - Use wait system call to collect the exit status of child process in parent process.
  - Terminate the parent process.
-

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<unistd.h>
#define MAX 20

int main()
{
int filedes[2],n;
char string[MAX];
char line[MAX];
pid_t pid;

if((pipe(filedes))<0)
{
printf("\n pipe creation error");
exit(0);
}
```

```
if(pid>0)
{
close(filedes[0]);
write(filedes[1],string,MAX);
}

if(pid==0)
{
close(filedes[1]);
n=read(filedes[0],line,MAX);
line[n]='\0';
printf("\n\n Data read by child is : %s",line);
}

exit(0);
}
```

```c
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<unistd.h>
#define MAX 20

int main()
{
int filedes[2],n;
char string[MAX];
char line[MAX];
pid_t pid;

printf("enter the string to be given to the parent");
fflush(stdin);

scanf("%s",string);
```

-
-
-
-
-

```c
if((pipe(filedes))<0)
{
printf("\n pipe creation error");
exit(0);
}

if((pid=fork())<0)
{
printf("\n fork error");
exit(0);
}

if(pid>0)
{
close(filedes[0]);
write(filedes[1],string,MAX);
}

if(pid==0)
{
close(filedes[1]);
n=read(filedes[0],line,MAX);
line[n]='\0';
printf("\n\n Data read by child is : %s",line);
}

exit(0);
}
```

Process Management-- Threads

- A thread is a part of a program.
- It is an execution unit within a process
- All threads of the same process share the same address space.
- All Threads have separate stacks and individual Thread IDs
- Thread is a lightweight process because :The context switching between threads is inexpensive in terms of memory and resources.

Process Management-- Threads

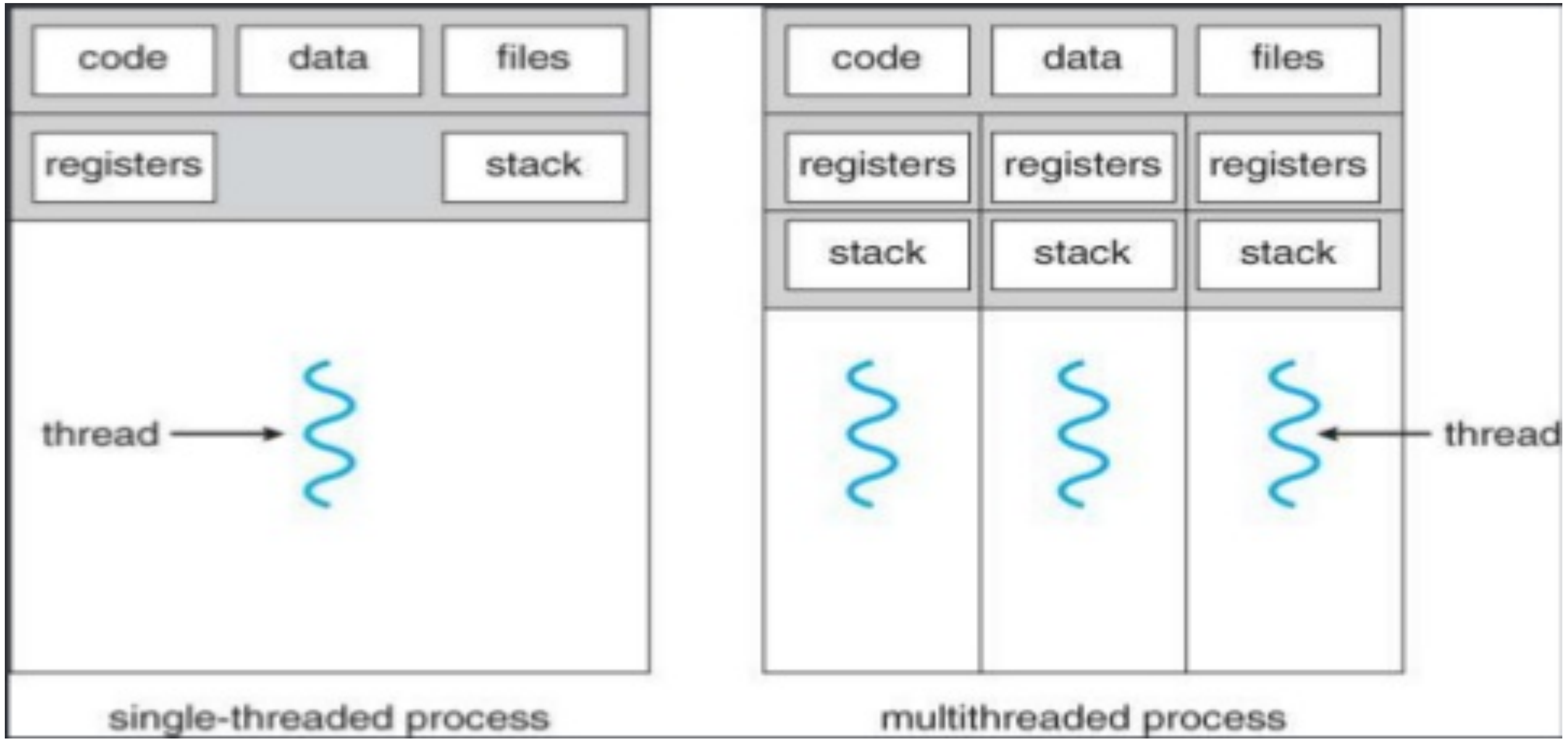
- Multithreading: Ability of an OS to support multiple, concurrent paths of execution within a single process.
- It is also described as the interleaved execution of threads.

Process Management– Differences between threads and processes

Process	Threads
A process is a program in execution	A thread is a part of the process
A process has its own Process ID	A thread has its own thread ID
Every process has its own memory space	Threads use the memory of the process they belong to
Inter process communication is slow as processes have different memory address	Inter thread communication for threads within the same process is fast
The context switching is more expensive in terms of memory and resources.	The context switching is less expensive in terms of memory and resources . Majorly because threads of the same process share the same memory space.

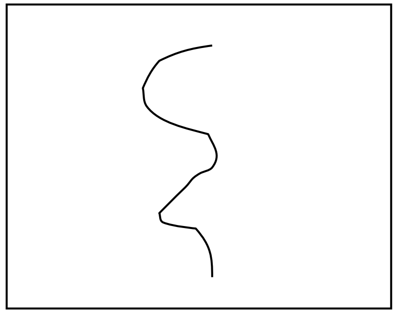
Process Management- ---

Threads, a diagrammatic representation

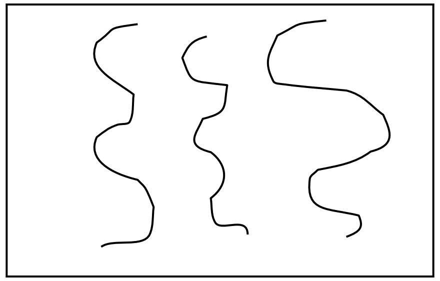


Process Management

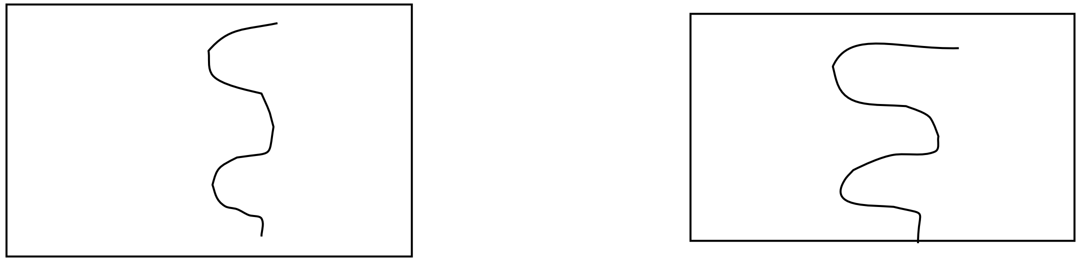
Threads and processes diagrammatic representation



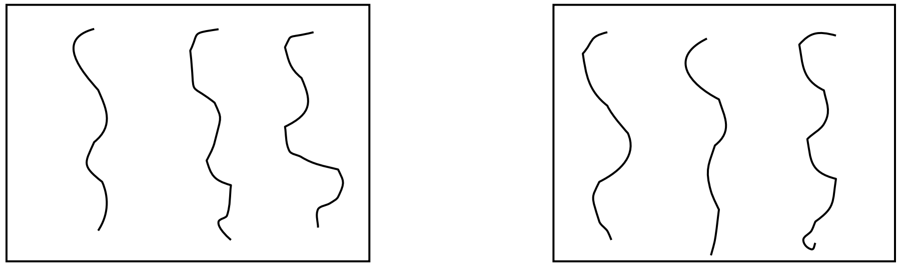
One process
one thread



One Process
Multiple threads



Multiple Processes
one thread per process



Multiple Processes
Multiple thread per process

Multithreading

- ?Operating system supports multiple threads of execution within a single process
- ?Examples:
  - ?MS-DOS supports a single thread
  - ?UNIX supports multiple user processes but only supports one thread per process
  - ?Java run time environment supports one process with multiple threads
  - ?Windows, Solaris, Linux, Mach, and OS/2 support multiple threads

Thread basics

- Thread operations include thread creation, termination, synchronization (join, blocking), scheduling, etc.
- All threads within a process share the same address space.
- **Threads in the same process share:**
    - Process instructions
    - Data
    - open files (descriptors)
    - signals
    - current working directory
    - User and group id
- **Each thread has a unique**:
    - Thread ID
    - set of registers
    - stack for local variables, return addresses
    - priority

Various States of a Thread

Like processes, threads go through several states:

1. **New** – Thread is being created.
2. **Ready** – Thread is ready to run and waiting for CPU.
3. **Running** – Thread is being executed.
4. **Waiting/Blocked** – Thread is waiting for an event or resource.
5. **Terminated** – Thread has completed execution.

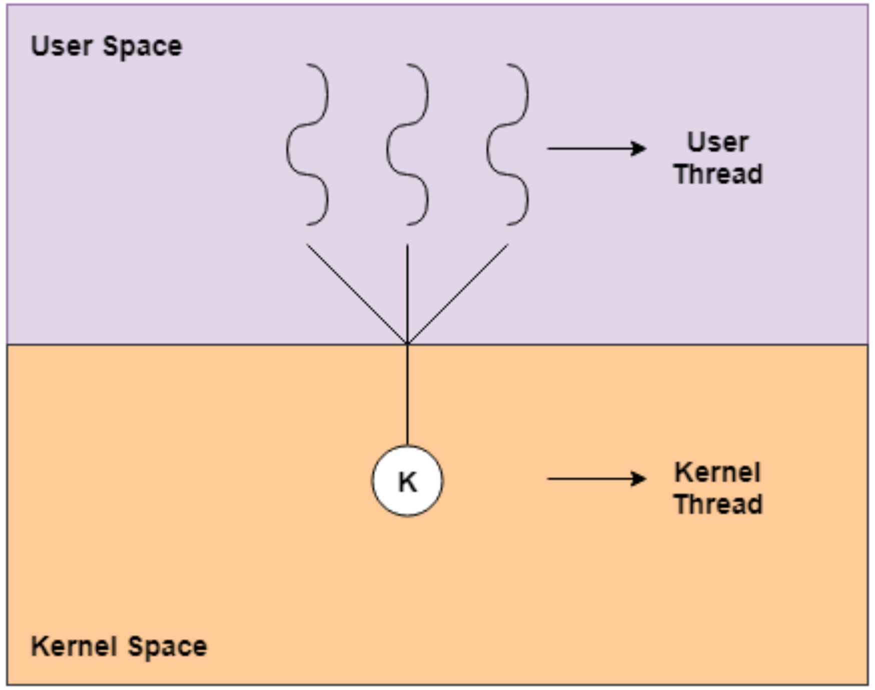
Benefits of Threads

•**Improved performance** on multi-core processors.
•**Faster context switching** than full processes.
•**Resource sharing**: threads of a process share memory and data.
•**Responsiveness**: a multithreaded application remains responsive (e.g., UI thread + background thread).
•**Better resource utilization**.

Types of Threads

There are majorly two types of threac

1. kernel level threads
2. User level threads



User - Level Threads

- User-level threads are implemented by users and the kernel is not aware of the existence of these threads
- Kernel handles them as if they were single-threaded processes.
- User-level threads are much faster than kernel level threads.
- They are represented by a program counter(PC), stack, registers and a small process control block.
- Also, there is no kernel involvement in synchronization for user-level threads.
- User-Level threads are managed entirely by the user-level library

User-Level Threads

- **Advantages of User-Level Threads**
- User-level threads are easier and faster to create than kernel-level threads. They can also be more easily managed.
- User-level threads can be run on any operating system.
- There are no kernel mode privileges required for thread switching in user-level threads.
- **Disadvantages of User-Level Threads**
- Multithreaded applications in user-level threads cannot use multiprocessing to their advantage.
- Entire process is blocked if one user-level thread performs blocking operation.

Kernel-Level Threads

- Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel
- Context information for the process as well as the process threads is all managed by the kernel.
- Because of this, kernel-level threads are slower than user-level threads.

Kernel-Level Threads

- **Advantages of Kernel-Level Threads**
- Multiple threads of the same process can be scheduled on different processors in kernel-level threads.
- The kernel routines can also be multithreaded.
- If a kernel-level thread is blocked, another thread of the same process can be scheduled by the kernel.
- **Disadvantages of Kernel-Level Threads**
- A mode switch to kernel mode is required to transfer control from one thread to another in a process.
- Kernel-level threads are slower to create as well as manage as compared to user-level threads.

POSIX (Portable Operating System Interface Standard) thread functions

pthread_create() ? Creates a new thread

pthread_exit() ? terminates the calling thread

pthread_join ? suspend execution of the calling thread until the target thread terminates

pthread_self() function returns the ID of the calling thread

pthread_detach() function marks the thread identified by *thread* as detached

Process Scheduling:

Process Scheduling:

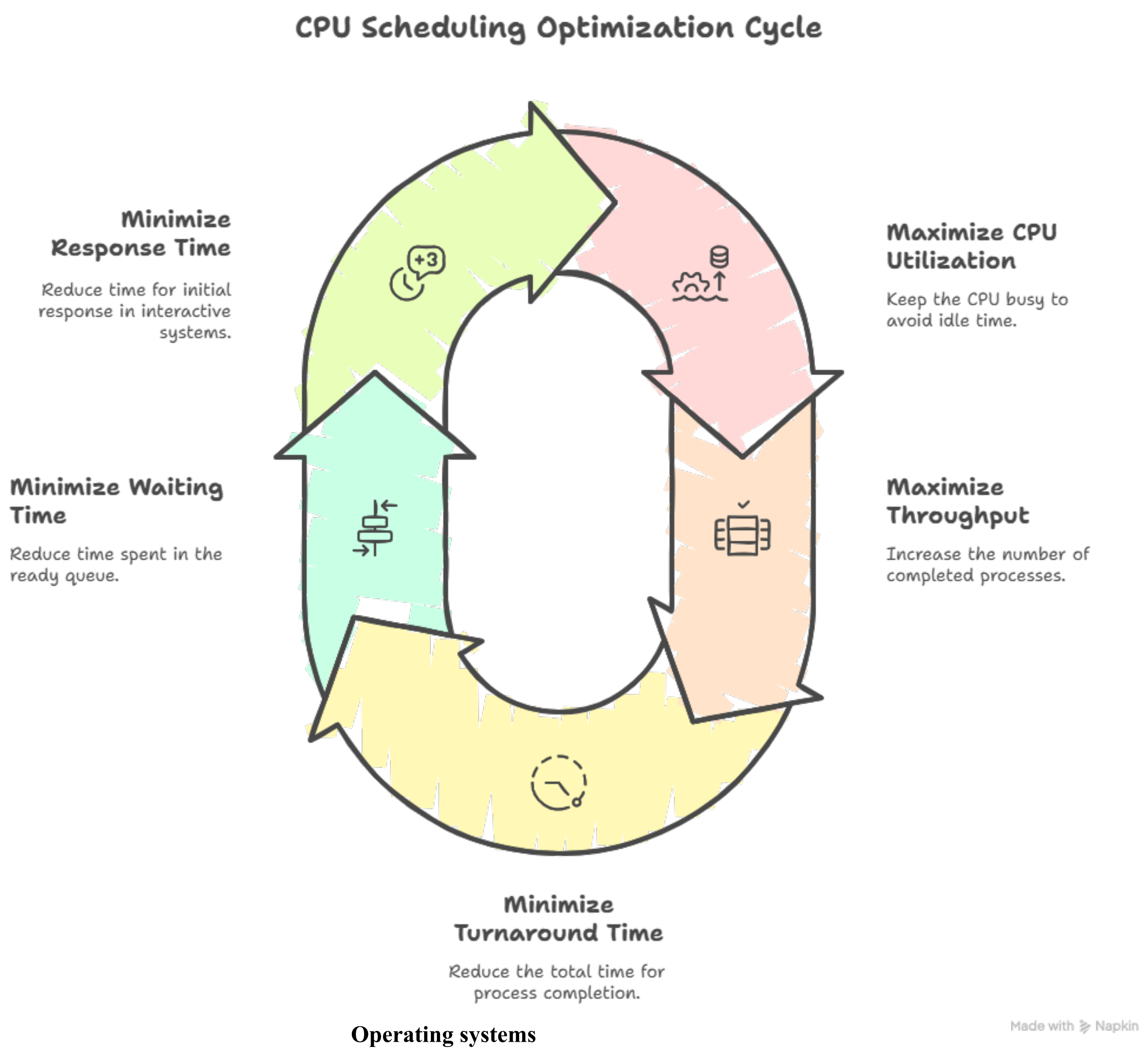
- Foundation and Scheduling objectives, Types of Schedulers, Scheduling criteria:
- CPU utilization, Throughput, Turnaround Time, Waiting Time, Response Time. Scheduling
- Algorithms: Pre-emptive and non-pre-emptive, FCFS, SIF, RR.

Process Scheduling:

Scheduling

Scheduling is the process of determining the order in which tasks, processes, or jobs are executed and how resources are allocated among them. This is crucial in various fields, from manufacturing and project management to computer operating systems.

Process Scheduling:

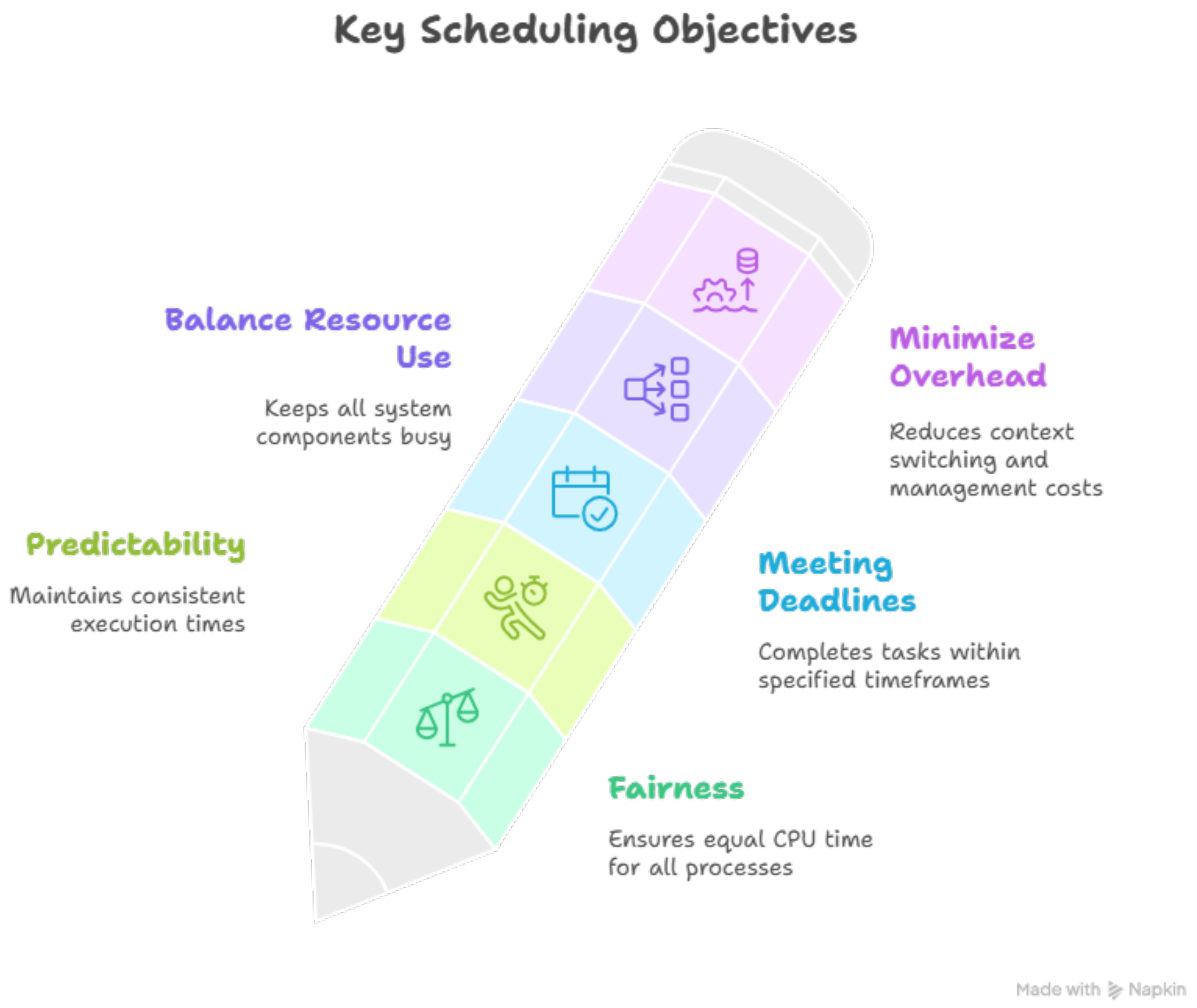

CPU Scheduling Optimization Cycle
Minimize Response Time
Reduce time for initial response in interactive systems.
Maximize CPU Utilization
Keep the CPU busy to avoid idle time.
Minimize Waiting Time
Reduce time spent in the ready queue.
Maximize Throughput
Increase the number of completed processes.
Minimize Turnaround Time
Reduce the total time for process completion.
+3
Made with Napkin

Process Scheduling:

scheduling is essential because:

- **Resource Management:** It allows multiple programs or users to share limited system resources (like the CPU, memory, and I/O devices) effectively.
- **Multiprogramming/Multitasking:** It creates the illusion that multiple tasks are running simultaneously, even on a single CPU, by rapidly switching between them (context switching).
- **Efficiency:** It aims to keep system resources busy and productive.
- **Responsiveness:** For interactive systems, it's vital to provide quick feedback to users.

Process Scheduling:


Key Scheduling Objectives
Balance Resource Use
Keeps all system components busy
Minimize Overhead
Reduces context switching and management costs
Predictability
Maintains consistent execution times
Meeting Deadlines
Completes tasks within specified timeframes
Fairness
Ensures equal CPU time for all processes
Made with Napkin

Process Scheduling:

scheduling is essential because:

- **Minimize Overhead:** Reduce the overhead associated with context switching and managing the scheduling process itself.
- **Balance Resource Use:** Keep all parts of the system busy. For example, processes that are I/O-bound should not hold the CPU unnecessarily while I/O devices are idle.
- **Meeting Deadlines (especially in Real-time Systems):** In real-time operating systems (RTOS), it's critical to ensure that processes complete their execution within specified deadlines to maintain system stability and correctness.
- **Predictability:** For a given process, ensure it runs in approximately the same amount of time under similar system loads.
- **Fairness:** Ensure that each process gets a fair share of the CPU time and resources, preventing any process from being indefinitely postponed (starvation).

Process Scheduling:

Schedulers are specialized components that manage how processes are selected and executed by the CPU.

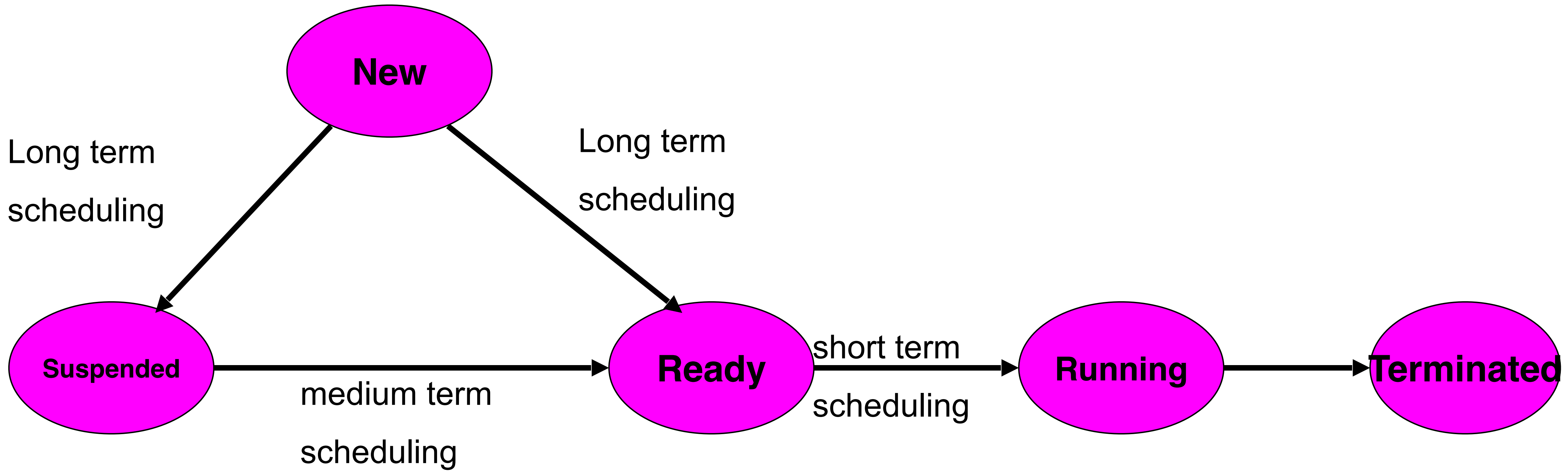
three main types of schedulers:

1. **Long-Term Scheduler (Job Scheduler or Admission Scheduler)**
2. **Short-Term Scheduler (CPU Scheduler or Dispatcher)**
3. **Medium-Term Scheduler (Swapper)**

Scheduler

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready state
  4. Terminates
- Scheduling under 1 and 4 is *non-preemptive*
- All other scheduling is *preemptive*

Types of process schedulers / Scheduling categories



Process Scheduler:

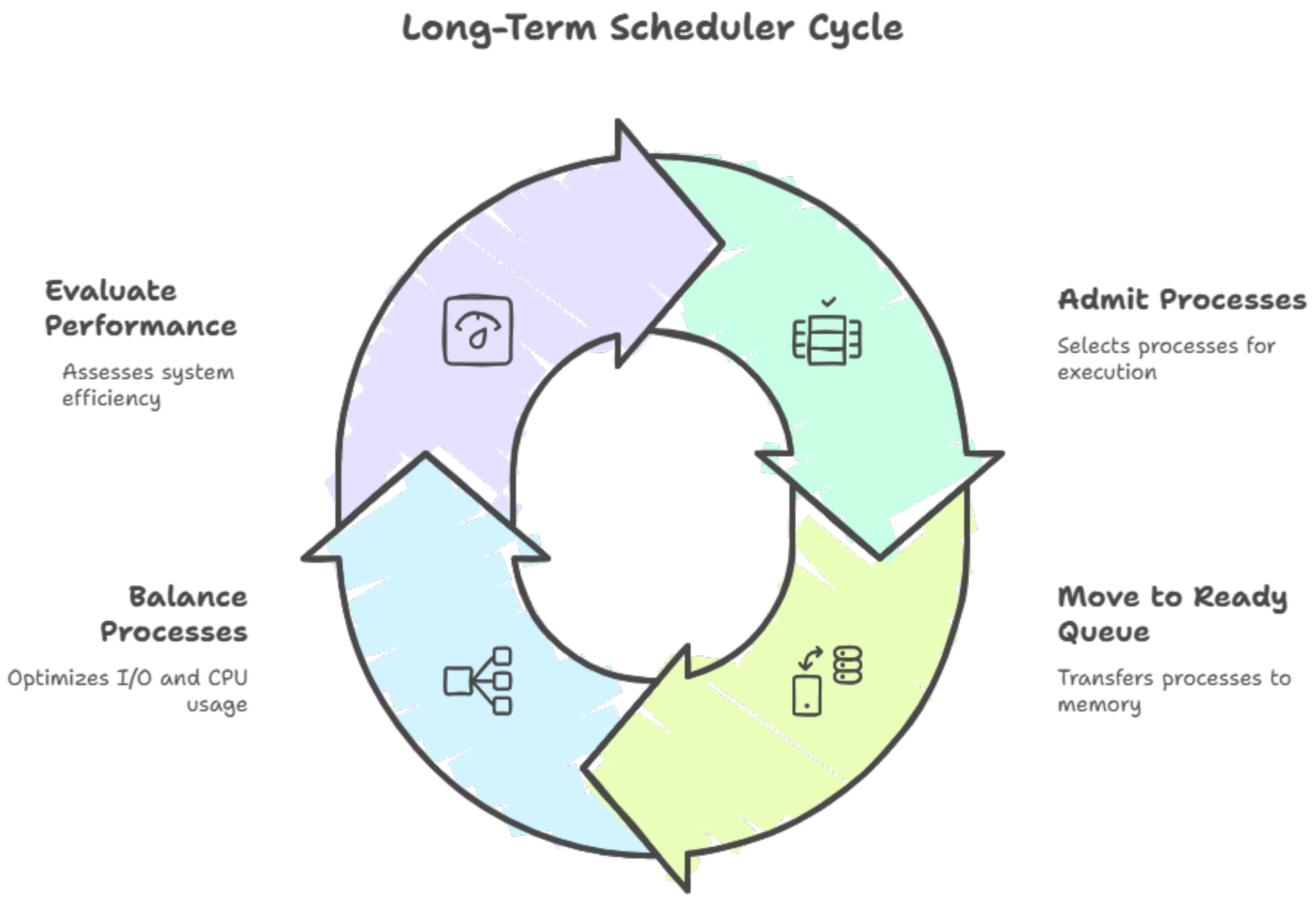
1.Long-Term Scheduler (Job Scheduler or Admission Scheduler)

- **Function**: Controls which processes are admitted into the system for execution.
- **Role**: Moves processes from the job queue (on disk) to the ready queue (in memory).
- **Goal**: Maintains a balance between I/O-bound and CPU-bound processes to optimize system performance.
- **Speed**: Slowest among all schedulers.
- **Presence**: Often absent in time-sharing systems like Windows.

MIT-WPU

॥ विश्वशान्तिर्धुवं ध्रुवा ॥

Long-Term Scheduler (Job Scheduler or Admission Scheduler)



Long-Term Scheduler Cycle
Admit Processes
Selects processes for execution
Move to Ready Queue
Transfers processes to memory
Balance Processes
Optimizes I/O and CPU usage
Evaluate Performance
Assesses system efficiency

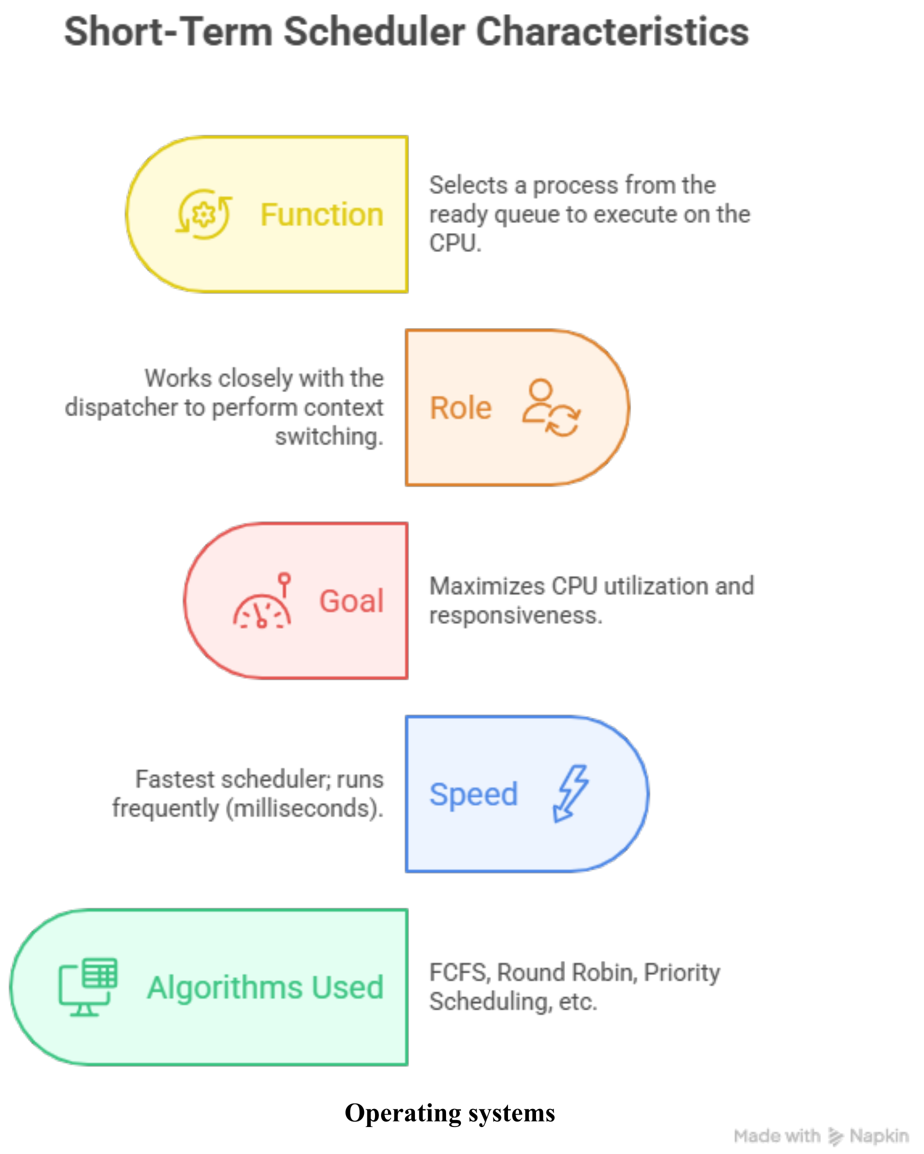
Made with Napkin

Process Scheduler:

2. Short-Term Scheduler (CPU Scheduler)

- Function: Selects a process from the ready queue to execute on the CPU.
- Role: Works closely with the dispatcher to perform context switching.
- Goal: Maximizes CPU utilization and responsiveness.
- Speed: Fastest scheduler; runs frequently (milliseconds).
- Algorithms Used: FCFS, Round Robin, Priority Scheduling, etc.

Short-Term Scheduler (CPU Scheduler)

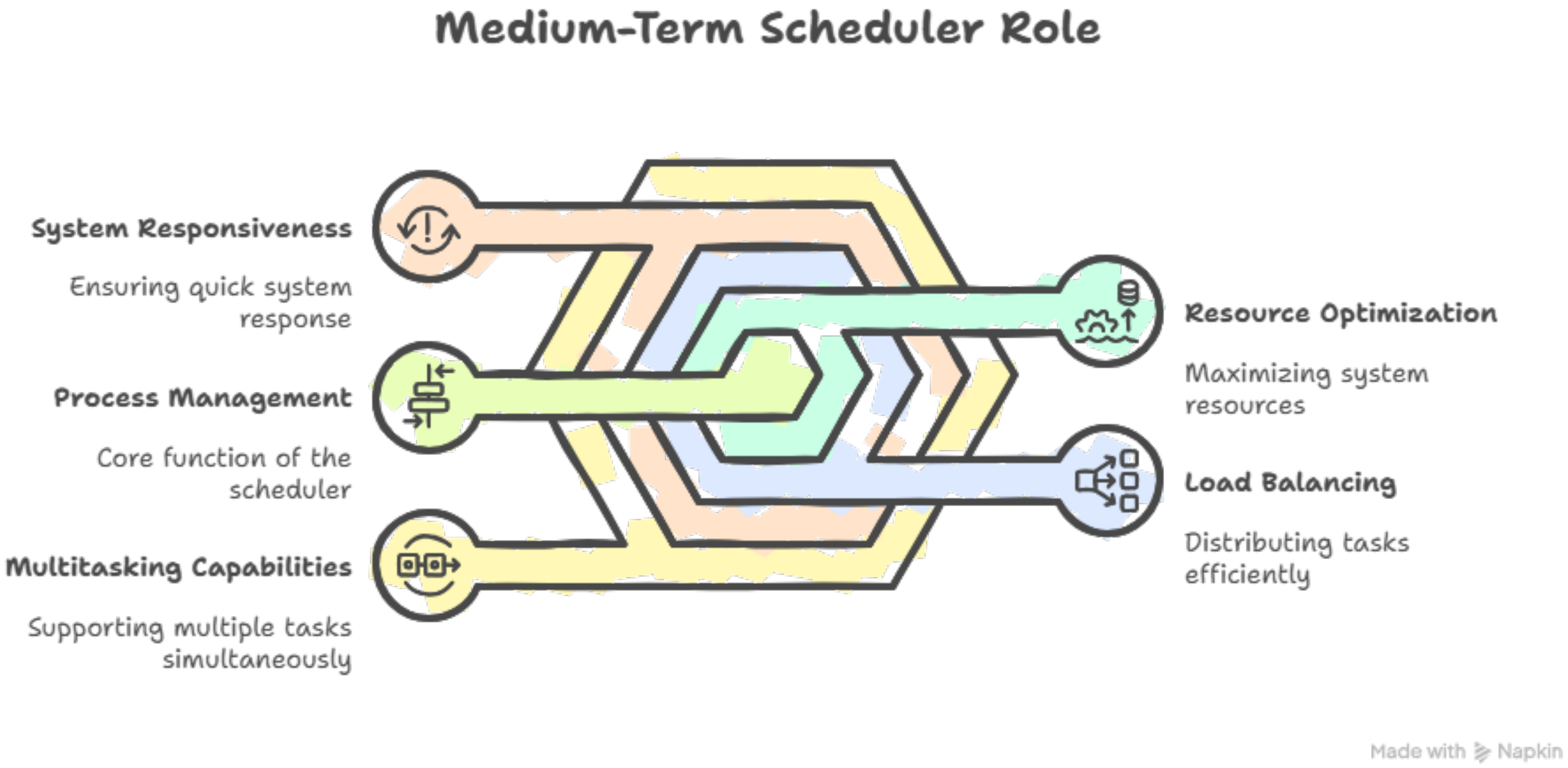

Short-Term Scheduler Characteristics
Function
Selects a process from the ready queue to execute on the CPU.
Role
Works closely with the dispatcher to perform context switching.
Goal
Maximizes CPU utilization and responsiveness.
Speed
Fastest scheduler; runs frequently (milliseconds).
Algorithms Used
FCFS, Round Robin, Priority Scheduling, etc.
Made with Napkin

Process Scheduler:

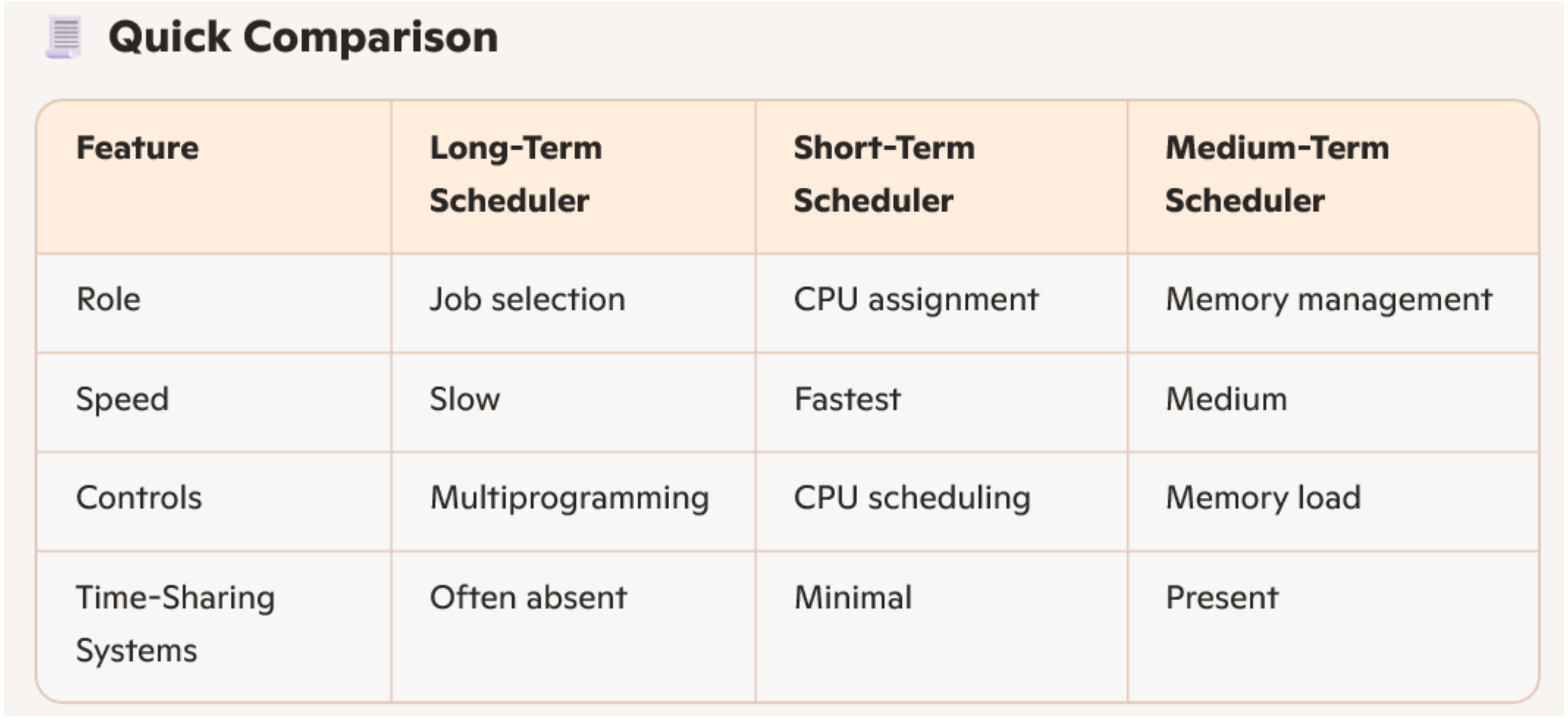
3. Medium-Term Scheduler (Swapper)

- **Function**: Temporarily removes (swaps out) processes from memory to reduce load.
- **Role**: Suspends processes that are waiting (e.g., for I/O) and resumes them later.
- **Goal**: Optimizes memory usage and maintains a good mix of active processes.
- **Speed**: Intermediate—slower than short-term, faster than long-term.
- **Presence**: Common in systems that support **swapping** and **virtual memory**.

Medium-Term Scheduler (Swapper)


Medium-Term Scheduler Role
System Responsiveness
Ensuring quick system response
Process Management
Core function of the scheduler
Multitasking Capabilities
Supporting multiple tasks simultaneously
Resource Optimization
Maximizing system resources
Load Balancing
Distributing tasks efficiently
Made with Napkin

compare Scheduler



Quick Comparison

Feature	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
Role	Job selection	CPU assignment	Memory management
Speed	Slow	Fastest	Medium
Controls	Multiprogramming	CPU scheduling	Memory load
Time-Sharing Systems	Often absent	Minimal	Present

What Are Scheduling Criteria?

Scheduling criteria are the **benchmarks** that help determine how well a scheduling algorithm performs. Since the CPU can execute only one process at a time, the operating system must decide which process to run next.

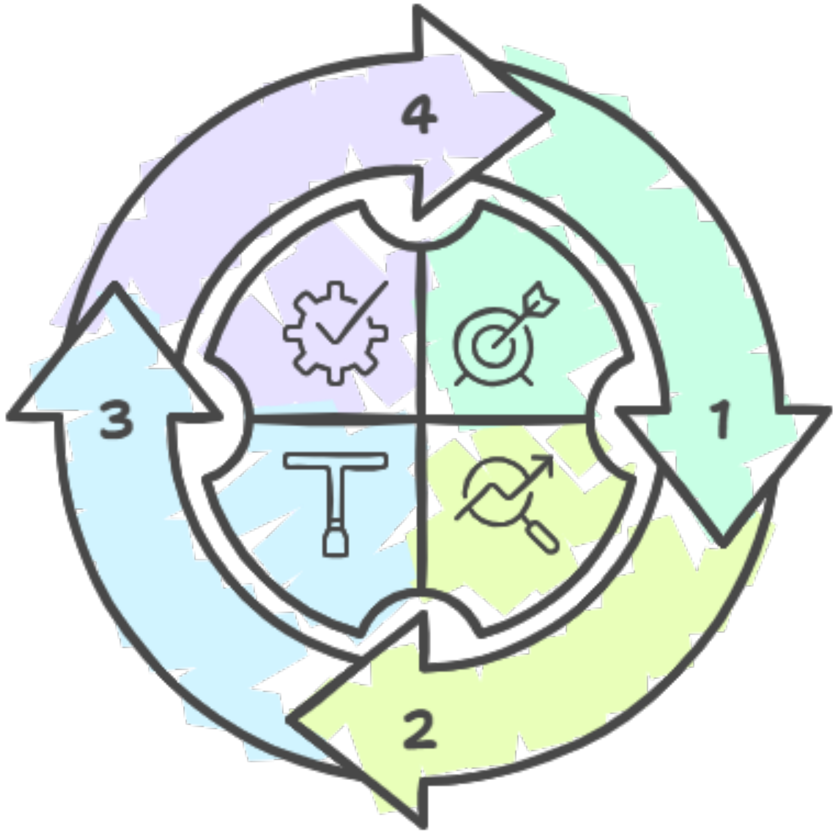
These decisions are guided by specific goals—like speed, fairness, and responsiveness.

What Are Scheduling Criteria?

Scheduling criteria are the **benchmarks** that help determine how well a scheduling algorithm performs. Since the CPU can execute only one process at a time, the operating system must decide which process to run next.

These decisions are guided by specific goals—like speed, fairness, and responsiveness.

Cycle of Scheduling Criteria



Implement Changes

Apply adjustments to scheduling

Define Goals

Establish objectives for scheduling

Adjust Algorithm

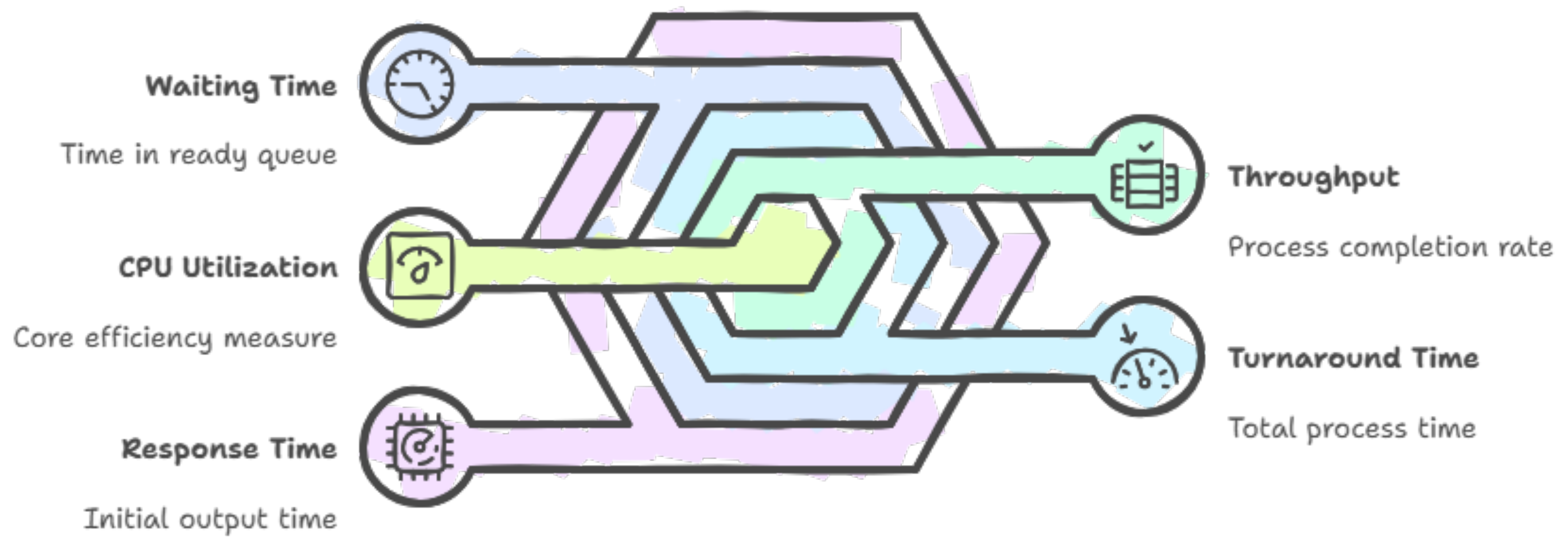
Modify algorithm based on evaluation

Evaluate Performance

Assess algorithm against goals

Made with Napkin

CPU Scheduling Criteria



Scheduling Algorithms

They deal with the problem of deciding which of the processes in ready queue is to be allocated the CPU

Algorithm compared based on following criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number of processes completed or amount of work done per unit time
- **Turnaround time** – time of submission of a process to the time of completion
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced

Optimization Criteria

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

What Are Scheduling Criteria?

CPU Utilization

This refers to how effectively the CPU is being used. Ideally, the CPU should be busy doing useful work as much as possible. If it’s idle too often, system resources are being wasted.

- Goal: Maximize CPU usage.
- Real-world range: Typically between 40% to 90%.
- Why it matters: High utilization means better performance and resource efficiency.

What Are Scheduling Criteria?

Throughput

Throughput is the number of processes completed in a given time frame. It reflects the system's productivity.

- Goal: Maximize throughput.
- Example: If 10 processes finish in 1 second, throughput is 10 processes/sec.
- Why it matters: Higher throughput means the system is handling more work efficiently.

What Are Scheduling Criteria?

Turnaround Time

This is the total time taken from the moment a process is submitted to the moment it is completed.

- Formula: Turnaround Time = Completion Time – Arrival Time
- Goal: Minimize turnaround time.
- Why it matters: Users care about how long their tasks take from start to finish.

What Are Scheduling Criteria?

Waiting Time

Waiting time is the amount of time a process spends in the ready queue, waiting for CPU allocation.

- Formula: Waiting Time = Turnaround Time – Burst Time
- Goal: Minimize waiting time.
- Why it matters: Long waits can lead to poor user experience and inefficiency.

What Are Scheduling Criteria?

Response Time

This is the time from when a process is submitted until it produces its first output.

- Goal: Minimize response time.
- Why it matters: Crucial for interactive systems like GUIs or real-time applications.

What Are Scheduling Criteria?

Fairness

Fairness ensures that all processes get a reasonable share of CPU time. It prevents starvation, where low-priority processes never get executed.

- Goal: Ensure equitable CPU access.
- Why it matters: Promotes system stability and user trust.

What Are Scheduling Criteria?

Predictability

Predictability means consistent performance under similar conditions. It’s especially important in real-time systems where timing guarantees are critical.

- Goal: Maintain consistent behavior.
- Why it matters: Users and developers rely on predictable execution for planning and debugging.

Scheduling Algorithms

- FCFS (First Come First Serve)
- SJF (Shortest Job First)
- Priority scheduling
- Round Robin scheduling

What Are Scheduling Criteria?

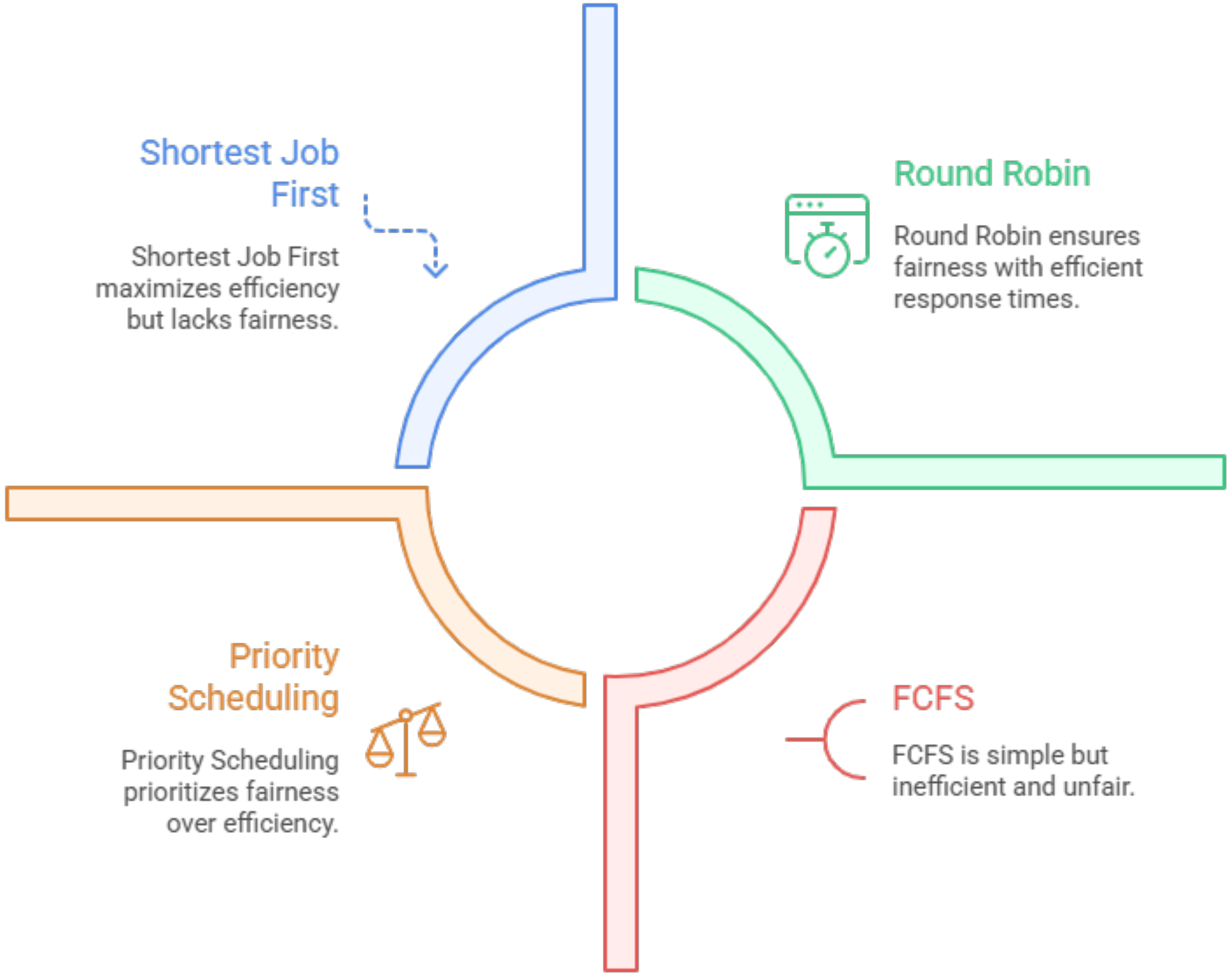
How These Criteria Influence Scheduling Algorithms

Different algorithms prioritize different criteria:

- **FCFS (First Come First Serve)**: Simple, but poor waiting time and turnaround time.
- **SJF (Shortest Job First)**: Excellent turnaround and waiting time, but risks starvation.
- **Round Robin**: Great for fairness and response time, but may increase waiting time.
- **Priority Scheduling**: Optimizes for importance, but can starve low-priority tasks.

Each algorithm is a trade-off. The choice depends on the system’s goals—whether it’s a batch system, real-time system, or interactive environment.

CPU Scheduling Algorithm Trade-offs


Shortest Job First
Shortest Job First maximizes efficiency but lacks fairness.
Round Robin
Round Robin ensures fairness with efficient response times.
Priority Scheduling
Priority Scheduling prioritizes fairness over efficiency.
FCFS
FCFS is simple but inefficient and unfair.

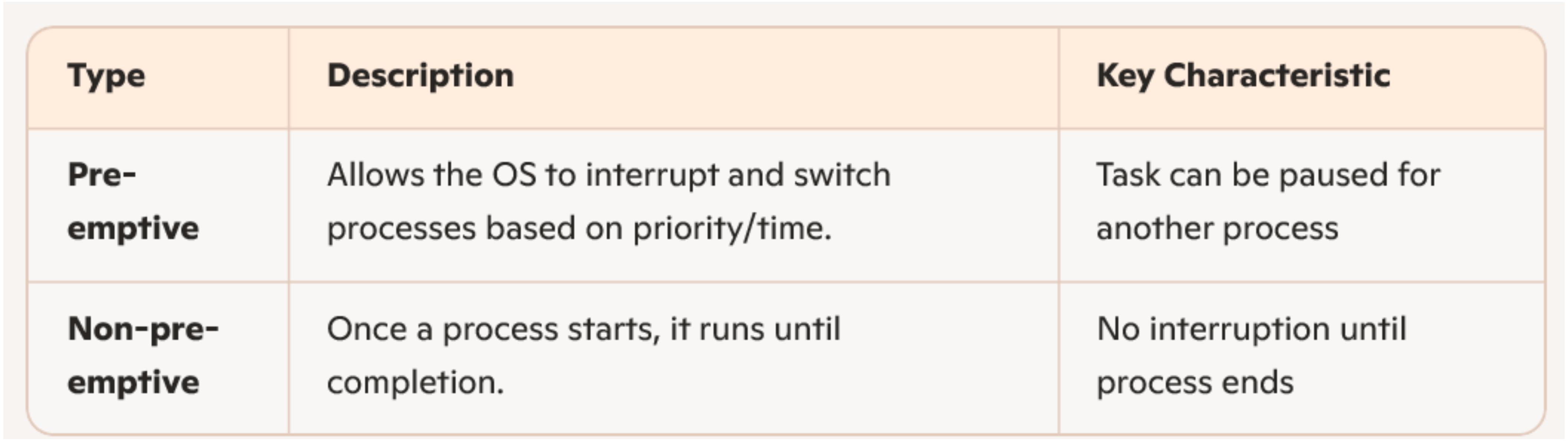
Made with Napkin

CPU / Process Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

- Scheduling may be *preemptive or non-preemptive*
- ***Non-preemptive*** – If CPU allocated to a process, it keeps the CPU until it releases it by terminating or switching to waiting state
- ***Preemptive*** - If CPU allocated to a process, it may be released if high priority process needs the CPU

Scheduling Algorithm- Pre-emptive vs Non-pre-emptive



Type	Description	Key Characteristic
Pre-emptive	Allows the OS to interrupt and switch processes based on priority/time.	Task can be paused for another process
Non-pre-emptive	Once a process starts, it runs until completion.	No interruption until process ends

FCFS Scheduling: Characteristics

Selection Function: max(w), selects the process which is waiting in the ready queue for maximum time.

Decision Mode : Non_preemptive

Throughput: Not emphasized

Response Time: May be high, especially if there is a large variance in the process execution times.

Overhead: Minimum

Effect on Processes: Penalizes short processes

Starvation: No

- **Completion Time**

  Time at which process completes its execution.
- **Turn Around Time**

  Time Difference between completion time and arrival time.

  Turn Around Time = Completion Time – Arrival Time
- **Waiting Time(W.T)**

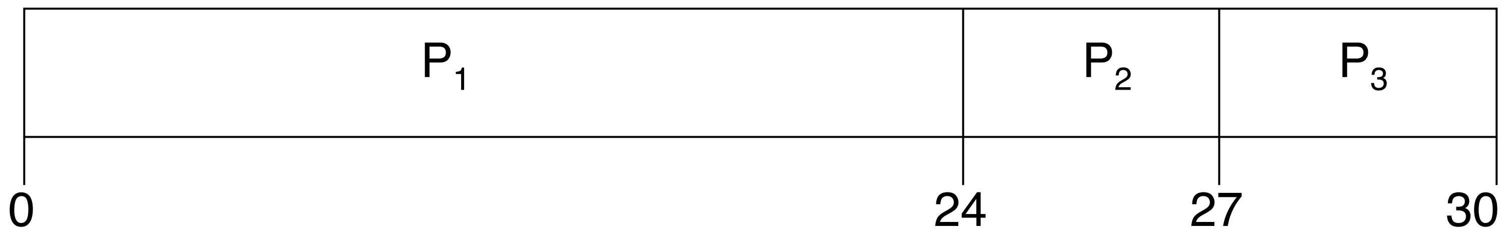
  Time Difference between turn around time and burst time.

  Waiting Time = Turn Around Time – Burst Time

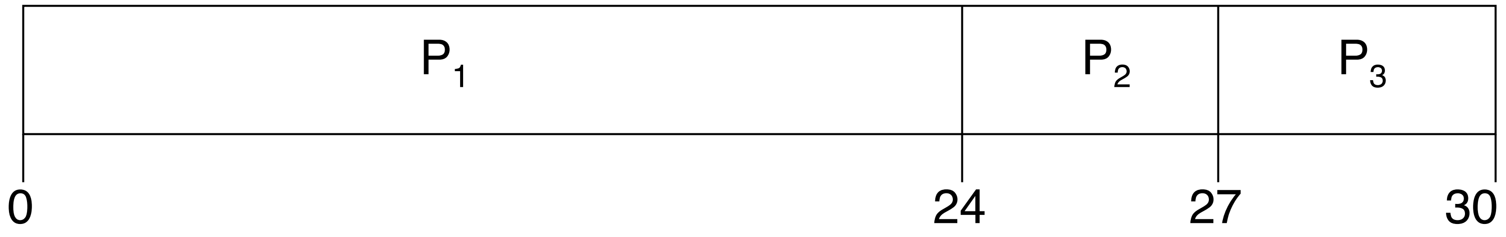
Process	**Burst Time**
P1	24
P2	3
P3	3

- Suppose that the processes arrive in the order: *P1* , *P2* , *P3*

The Gantt Chart for the schedule is:



The Gantt Chart for the schedule is:

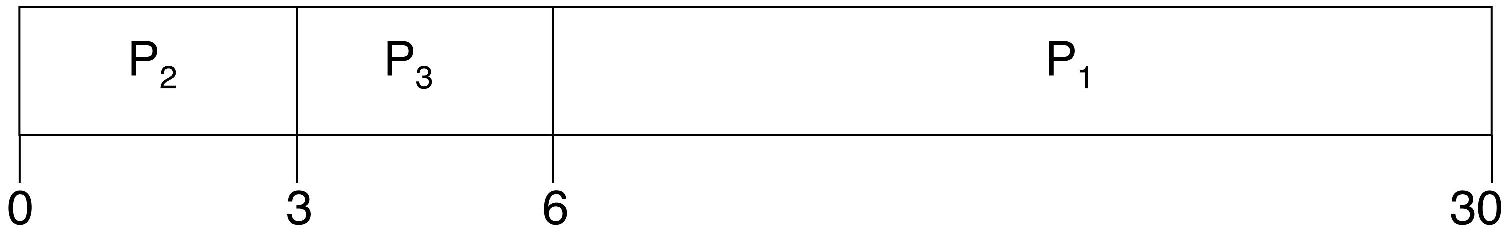


Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
Average waiting time: (0 + 24 + 27)/3 = 17

Turnaround time for *P1* = 24; *P2* = 27; *P3* = 30
Average turnaround time : (24 + 27 + 30)/3 = 27

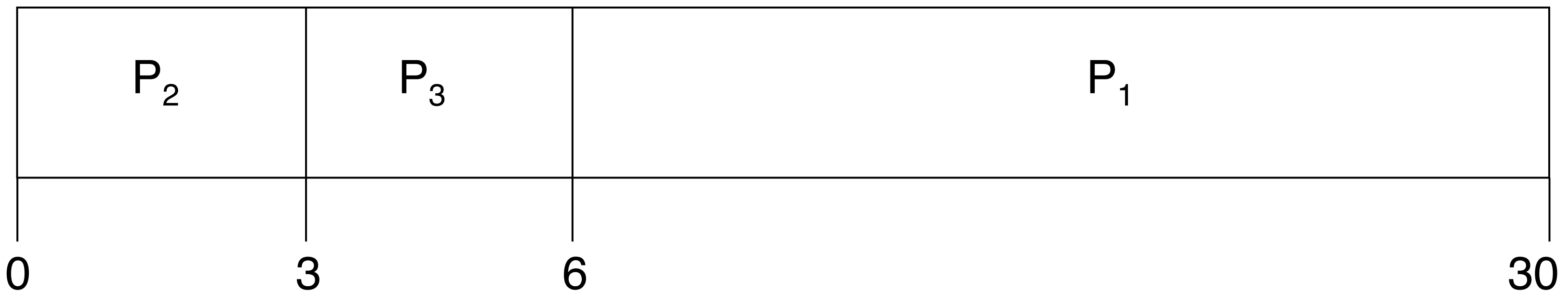
FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order : *P2 , P3 , P1*



FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order : *P2* , *P3* , *P1*



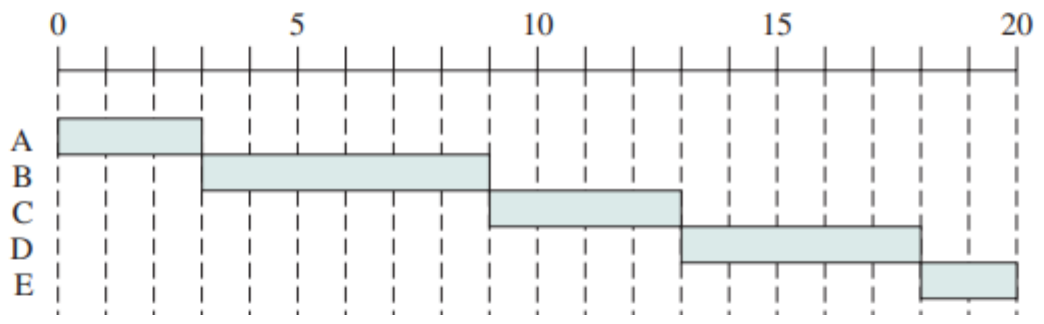
Waiting time for *P1* = 6; *P2* = 0; *P3* = 3
Average waiting time: (6 + 0 + 3)/3 = 3

Turnaround time for *P1* = 30; *P2* = 3; *P3* = 6
Average turnaround time : (30 + 3 + 6)/3 = 13

Example-2 of FCFS

Process	Arrival Time	Burst Time
P1	0.0	3
P2	2.0	6
P3	4.0	4
P4	6.0	5
P5	8.0	2

Example-2 of FCFS



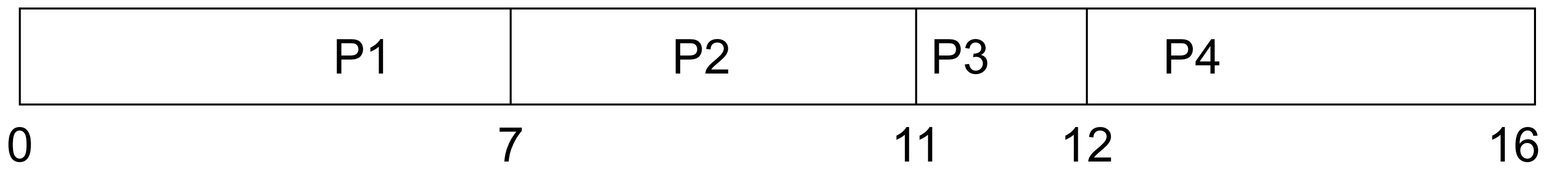
Waiting Time	**Turnaround Time**
0	3
1	7
5	9
7	12
10	12

Avg WT= 23/5=4.6

Avg TAT=8.60

Example-3 of FCFS

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4



Average waiting time = (0 + 5 + 7 + 7)/4 = 4.75
Average Turnaround Time = (7+9+8+11)/4=8.75

First Come First Serve (FCFS) (Non Pre-emptive) Lab implementation

I/P

1.No of processes

2.Arrival Time of all processes

3.Burst time of all processes

O/P

1.Gantt chart

2.Completion time/Finish Time

3.Turnaround time, avg Turnaround time

4.Waiting time, avg Waiting time

Pseudo code for the algorithm

```
ST(0)=AT(0)
CT(i)=ST(i)+BT(i)
ST(i+1)=CT(i)
TAT(i)=CT(i)-AT(i)
WT(i)=TAT(i)-BT(i)
```

Shortest-Job-First (SJF) Scheduling

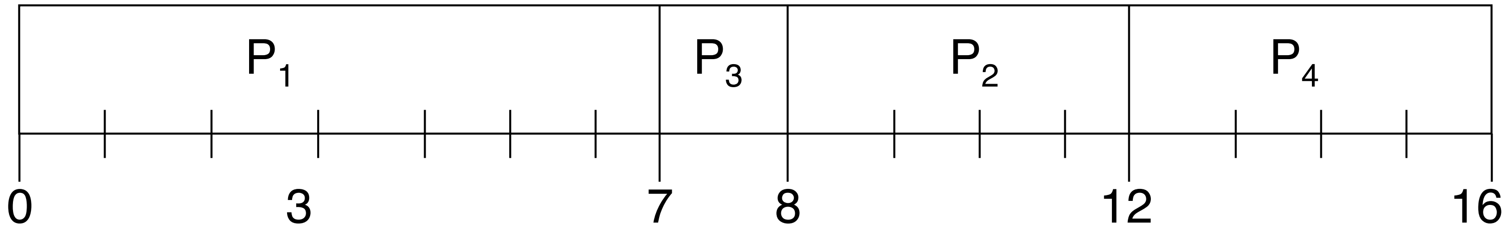
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - **Nonpreemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst
  - **Preemptive** – If a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the **Shortest-Remaining-Time-First (SRTF)**
- **SJF is optimal** – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4



- **Average waiting time = (0 + 6 + 3 + 7)/4 = 4**
- **Average Turnaround Time = (7+10+4+11)/4=8**

Non Preemptive SJF Ex2

Process	AT	BT	CT	TAT	WT
P1	1	7			
P2	2	5			
P3	3	1			
P4	4	2			
P5	5	3			

Non Preemptive SJF Ex2

Process	AT	BT	CT	TAT	WT
P1	1	7	8	7	0
P2	2	5	19	17	12
P3	3	1	9	6	5
P4	4	2	11	7	5
P5	5	3	14	9	6

Shortest Job First Preemptive or Shortest Remaining Time

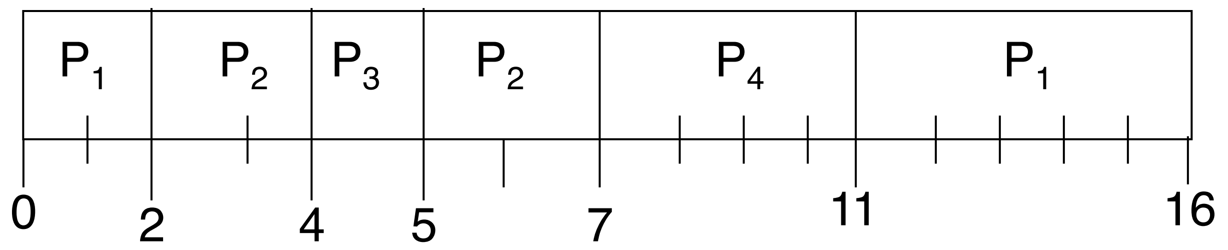
- It is a preemptive version of SJF. In this policy, scheduler always chooses the process that has the **shortest expected remaining processing time.**
- When a new process arrives in the ready queue, it may in fact have a shorter remaining time than the currently running process.
- Accordingly, the scheduler may preempt whenever a new process becomes ready.
- Scheduler must have an estimate of processing time to perform the selection function.

Shortest Job First Preemptive or Shortest Remaining Time: characteristics

- **Selection Function**: minimum total service time required by the process, minus time spent in execution so far.
- **Decision Mode :** Preemptive (At arrival time)
- **Throughput:** High
- **Response Time:** Provides good response time
- **Overhead:** Can be high
- **Effect on Processes:** Penalizes long processes.
- **Starvation**: Possible

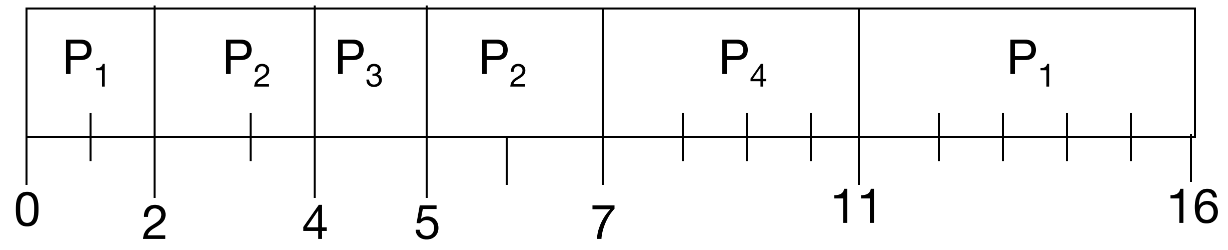
Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Example of Preemptive SJF Ex1

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

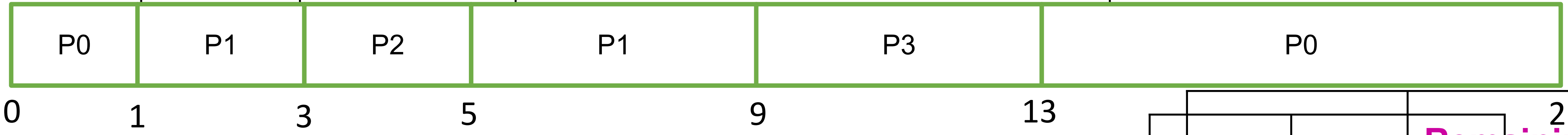


- Average waiting time = (9 + 1 + 0 +2)/4 = 3
- Average turnaround time= (16+5+1+6)/4=7

Shortest Remaining Time Next (SRTN) (Pre-emptive)_Example

Process	Arrival Time (T0)	CPU Burst Time (in milliseconds) (Time required for completion ∆T)
P0	0	10
P1	1	6
P2	3	2
P3	5	4

Gantt Chart



- Initially only process P0 is present and it is allowed to run
- But when process P1 comes, it has shortest remaining run time.
- So, P0 is pre-empted and P1 is allowed to run.
- Whenever new process comes or current process blocks, such type of decision is taken.
- This procedure is repeated till all processes complete their execution.

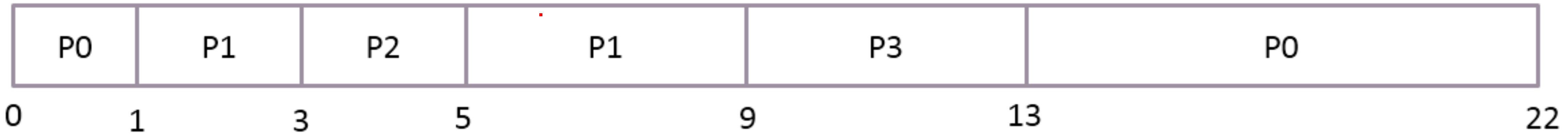
[illegible]

Shortest Remaining Time Next (SRTN) (Pre-emptive)_Example

Output

Process	Arrival Time (T0)	Burst Time (∆T)	Finish Time (T1)
P0	0	10	22
P1	1	6	9
P2	3	2	5
P3	5	4	13

Gantt Chart

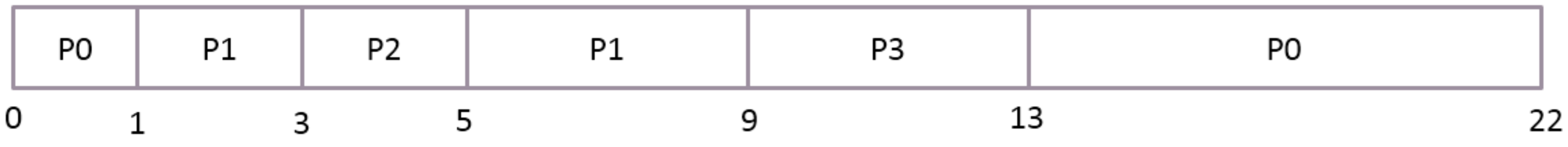


Shortest Remaining Time Next (SRTN) (Pre-emptive)_Example

Output

Process	Arrival Time (T0)	Burst Time (∆T)	Finish Time (T1)	Turnaround Time (TAT = T1 - T0)
P0	0	10	22	22
P1	1	6	9	8
P2	3	2	5	2
P3	5	4	13	8

Gantt Chart



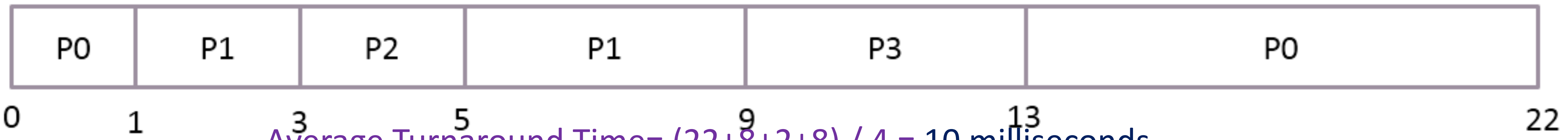
- Average Turnaround Time= (22+8+2+8) / 4 = 10 milliseconds

Shortest Remaining Time Next (SRTN) (Pre-emptive)_Example

Output

Process	Arrival Time (T0)	Burst Time (∆T)	Finish Time (T1)	Turnaround Time (TAT = T1 - T0)	Waiting Time (WT = TAT - ∆T)
P0	0	10	22	22	12
P1	1	6	9	8	2
P2	3	2	5	2	0
P3	5	4	13	8	4

Gantt Chart



- Average Turnaround Time= (22+8+2+8) / 4 = 10 milliseconds
- Average Waiting Time = (12+2+0+4) / 4= 4.5 milliseconds

Priority scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

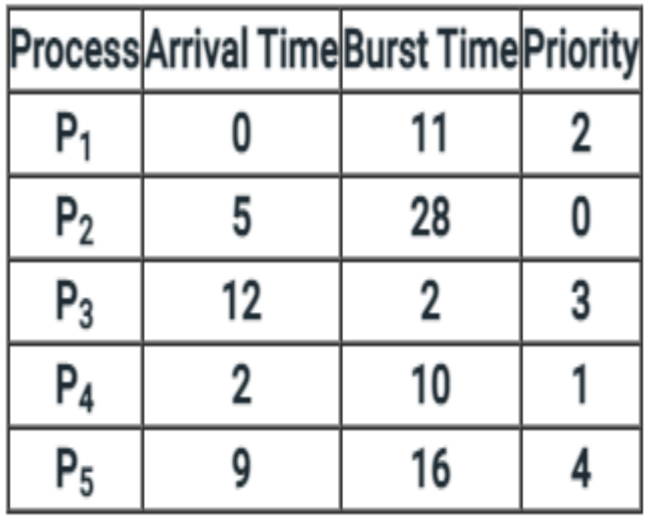
Priority Scheduling-Non Preemptive

Process	AT	BT	Priority	CT	TAT	WT
P1	0	11	2			
P2	5	28	0			
P3	12	2	3			
P4	2	10	1			
P5	9	16	4			

Priority Scheduling-Non Preemptive

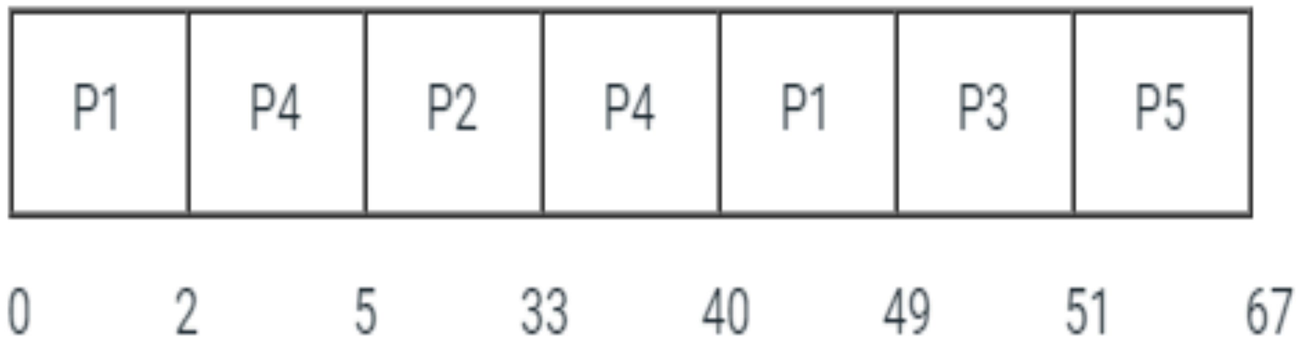
Process	AT	BT	Priority	CT	TAT	WT
P1	0	11	2	11	11	0
P2	5	28	0	39	34	6
P3	12	2	3	51	39	37
P4	2	10	1	49	47	37
P5	9	16	4	67	58	42

Priority Scheduling-Preemptive

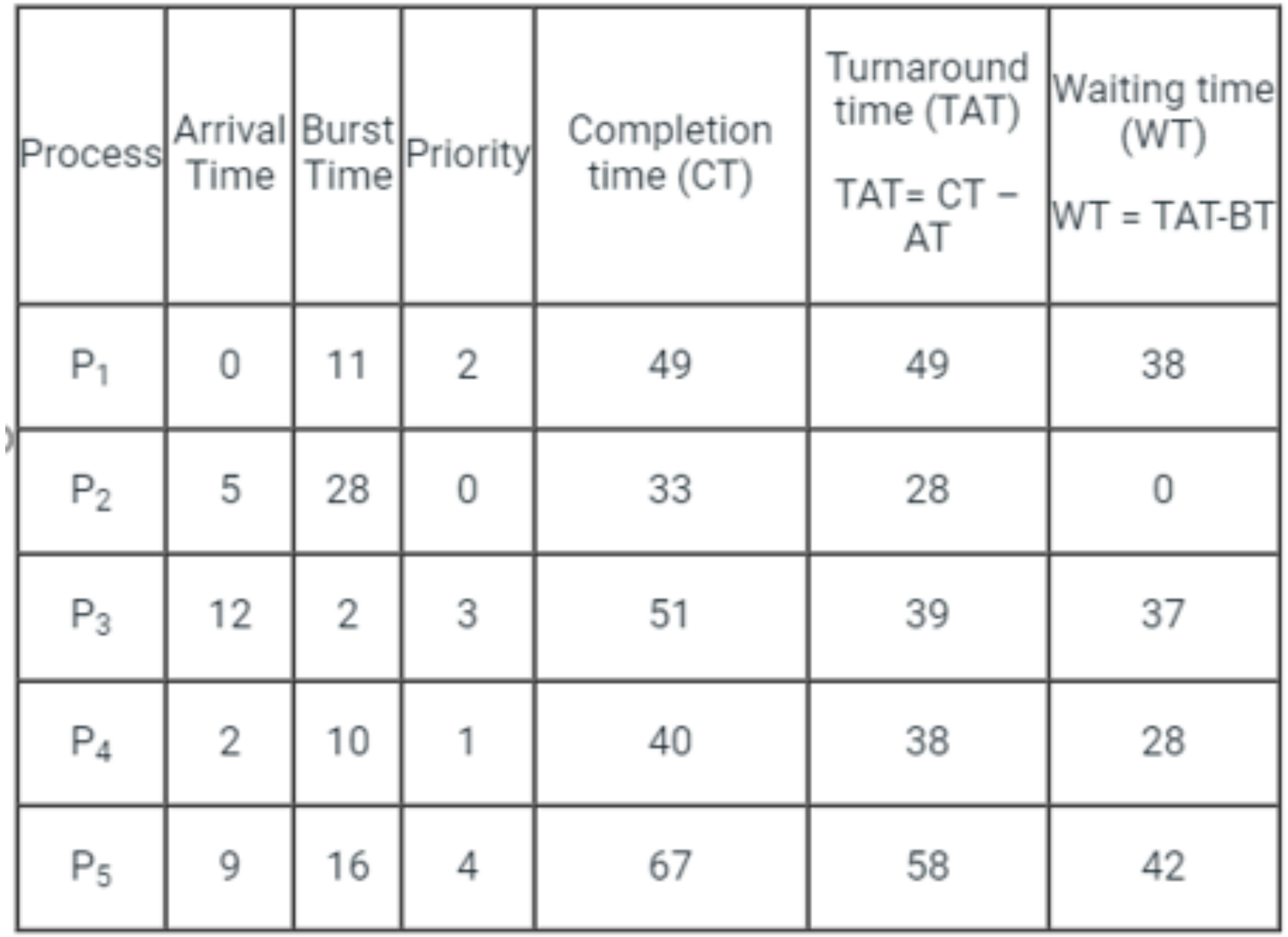


Process	Arrival Time	Burst Time	Priority
P_1	0	11	2
P_2	5	28	0
P_3	12	2	3
P_4	2	10	1
P_5	9	16	4

Gantt chart:



Priority Scheduling-Preemptive



Process	Arrival Time	Burst Time	Priority	Completion time (CT)	Turnaround time (TAT) TAT= CT – AT	Waiting time (WT) WT = TAT-BT
P_1	0	11	2	49	49	38
P_2	5	28	0	33	28	0
P_3	12	2	3	51	39	37
P_4	2	10	1	40	38	28
P_5	9	16	4	67	58	42

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*)
- After time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once.
- No process waits more than $(n-1)q$ time units.

RR: characteristics

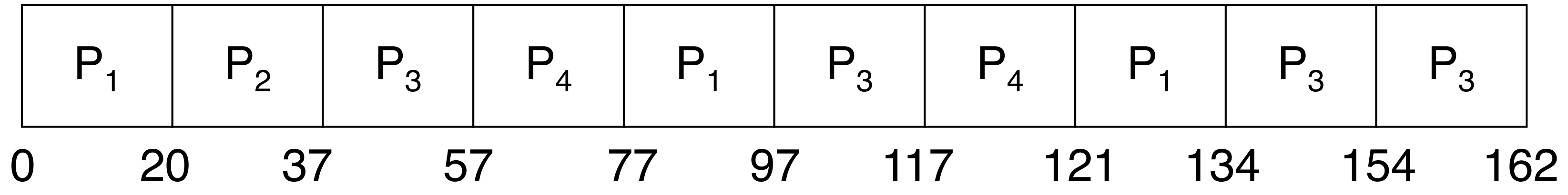
- **Selection Function**: constant
- **Decision Mode :** Preemptive (At time quantum)
- **Throughput:** May be low if time quantum is too small
- **Response Time:** Provides good response time for short processes
- **Overhead:** Minimum
- **Effect on Processes:** Fair treatment
- **Starvation**: No

Example of RR with Time Quantum = 20

Process	Burst Time
P1	53
P2	17
P3	68
P4	24

Example of RR with Time Quantum = 20

Process	Burst Time
P1	53
P2	17
P3	68
P4	24



- Average waiting time = (81+ 20 + 94+ 97)/4 = 73
- Average turnaround time= (134+37+162+121)/4=113.5

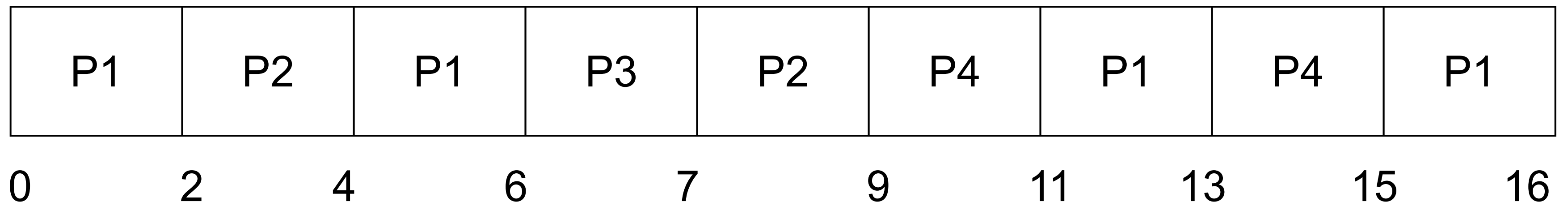
- Typically, higher average turnaround than SJF, but better *response*

Example: Round Robin (By Default preemptive: Time Quantum 2)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

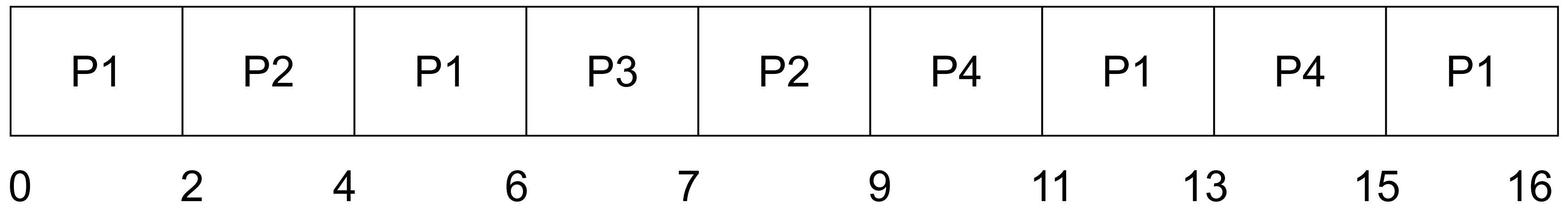
Example: Round Robin (By Default preemptive: Time Quantum 2)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Example: Round Robin (By Default preemptive: Time Quantum 2)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Average waiting time = (9 + 3 + 2 +6)/4 =5
Average turnaround time =(16+7+3+10)/4=9

Round Robin Ex 2 (Q=4)

Process	AT	BT	CT	TAT	WT
P1	0	3			
P2	2	6			
P3	4	4			
P4	6	5			
P5	8	2			

Round Robin Ex 2 (Q=4)

Process	AT	BT	CT	TAT	WT
P1	0	3	3	3	0
P2	2	6	17	15	9
P3	4	4	11	7	3
P4	6	5	20	14	9
P5	8	2	9	11	9

Round Robin Ex 2 (Q=1)

Process	AT	BT	CT	TAT	WT
P1	0	3			
P2	2	6			
P3	4	4			
P4	6	5			
P5	8	2			

Round Robin Ex 2 (Q=1)

Process	AT	BT	CT	TAT	WT
P1	0	3	4	4	1
P2	2	6	18	16	10
P3	4	4	17	13	9
P4	6	5	20	14	9
P5	8	2	15	7	5

References

1. William Stallings, Operating System: Internals and Design Principles, Prentice Hall, ISBN-10: 0-13-380591-3, ISBN-13: 978-0-13-380591-8, 8th Edition
2. Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, WILEY,ISBN 978-1-118-06333-0, 9th Edition

SJF - preemptive

Process	AT	BT	CT	TAT	WT
P1	0	8			
P2	1	4			
P3	2	2			
P4	3	1			

SJF - preemptive

Process	AT	BT	CT	TAT	WT
P1	0	8	15	15	7
P2	1	4	7	6	2
P3	2	2	5	3	1
P4	3	1	5	2	1

FCFS- Non Preemative

Process	AT	BT	CT	TAT	WT
P1	0	5			
P2	1	3			
P3	2	8			
P4	3	6			

FCFS- Non Preemative

Process	AT	BT	CT	TAT	WT
P1	0	5	5	5	0
P2	1	3	8	7	4
P3	2	8	16	14	6
P4	3	6	22	`19	13