



OPERATING SYSTEMS

Module2_Part1

Textbook : Operating Systems Concepts by Silberschatz



Process Management

Process:

Most central concept of any operating system is the process.

process is a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process.

The process includes the current activity as represented by the value of the program counter and the content of the processor registers.

The program is only part of a process.

Also it includes the process stack which contain temporary data (such as method parameters return address and local variables) & a data section which contain global variables.

Process Management

Difference between process & program:

A program by itself is not a process. A program in execution is known as a process. A program becomes a process when an executable file is loaded into memory.

A program is

a passive entity, such as the contents of a file stored on disk .

process is an active entity with

- The program code, also called **text section**
- Current activity including **program counter**, processor registers
- **Stack** containing temporary data
 - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time



A system therefore consists of a collection of processes:

operating system processes -executing system code

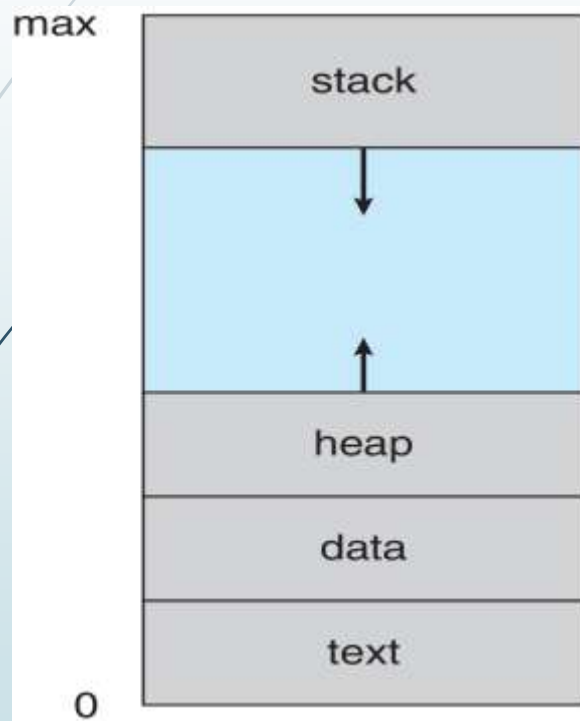
user processes-executing user code.

Potentially all these processes can execute concurrently with the CPU

By switching the CPU between processes

the operating system can make the computer more productive.

Process in memory



Attributes of a process

Process ID
Program Counter
Process State
Priority
General Purpose Registers
List of Open Files
List of Open Devices

Process Attributes



Process States

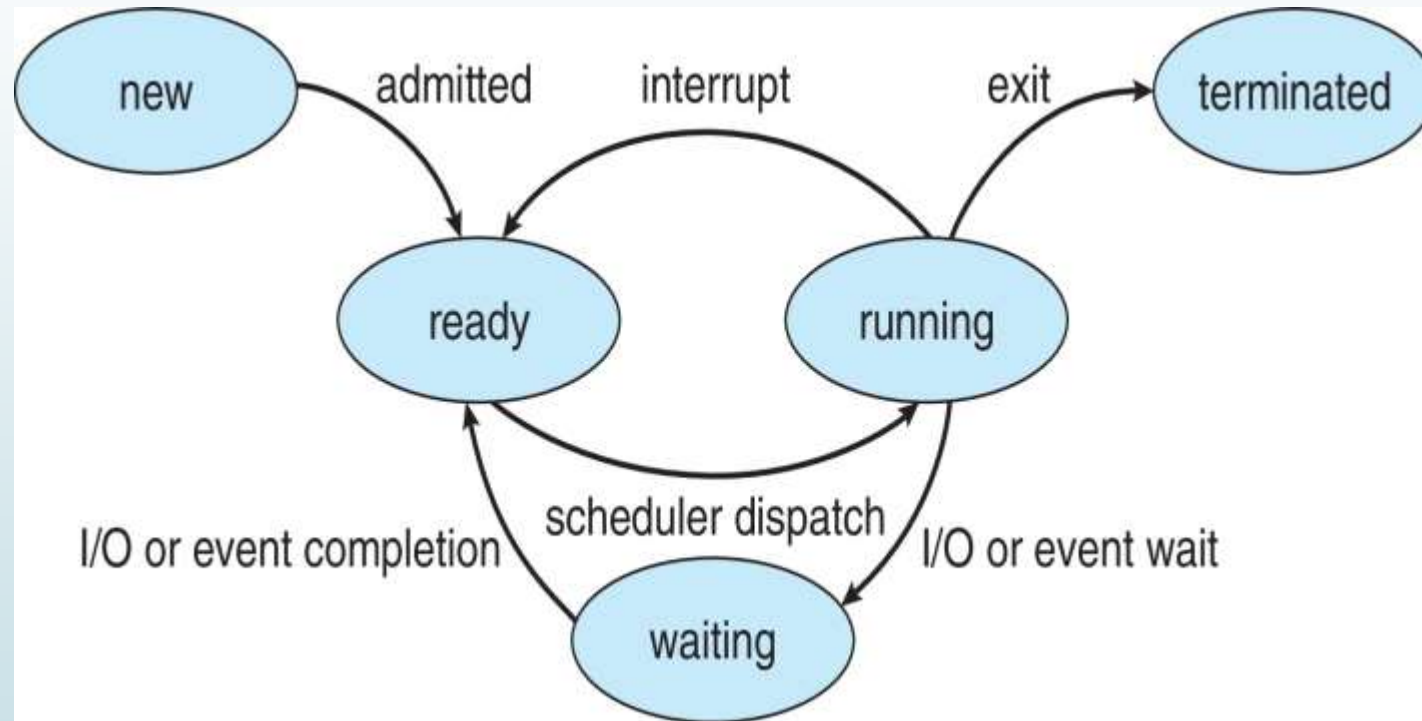
Process State

As a process executes, it changes state. The state of a process is defined in part by the

current activity of that process. Each process may be in one of the following states:

- **New:** The process is being created
- **Running:** Instructions are being executed
- **Waiting:** The process is waiting for some event to occur
- **Ready:** The process is waiting to be assigned to a processor
- **Terminated:** The process has finished execution

Diagram of process states





Process control block

- Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**.
- It contains many pieces of information associated with a specific process, including these:
 - Process state – running, waiting, etc.
 - Program counter – indicates the address of the next instruction to be executed
- CPU registers – include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.

Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterwards

Process control block

CPU scheduling information

- priorities, scheduling queue pointers

Memory-management information

- memory allocated to the process This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system

Accounting information

- CPU used, clock time elapsed since start, time limits

I/O status information

- I/O devices allocated to process, list of open files





Threads

A **thread** is a single sequential flow of control within a program. It is possible that multiple **threads** can be run at the same time and performing different tasks in a single program.

Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.

A process was an executing program with a single thread of control. Most modern operating systems now provide features enabling a process to contain multiple threads of control.

On a system that supports threads, the PCB is expanded to include information for each thread

In a multithreaded process on a single processor, the processor can switch execution between threads, resulting in concurrent execution.



Threads

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.

It shares with other threads belonging to the same process, its code section, data section, and other operating-system resources, such as open files and signals for example.

A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.



OPERATING SYSTEMS

Module2_Part2

Textbook : Operating Systems Concepts by Silberschatz



Scheduling

Scheduling is a fundamental function of OS.

When a computer is multiprogrammed, it has multiple processes competing for the CPU at the same time.

If only one CPU is available, then a choice has to be made regarding which process to execute next.

This decision making process is known as scheduling and the part of the OS that makes this choice is called a scheduler. The algorithm it uses in making this choice is called scheduling algorithm.

The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs.

It determines which process is moved from ready state to running state

Scheduling Queues

- **Scheduling queues:** As processes enter the system, they are put into a job queue. This queue consists of all process in the system.

The process that are residing in main memory and are ready & waiting to execute is kept on a list called ready queue.

This queue is generally stored as a linked list.

A ready queue header contains pointers to the first & final PCB in the list. The PCB includes a pointer field that points to the next PCB in the ready queue.

The lists of processes waiting for a particular I/O device are kept on a list called device queue. Each device has its own device queue.

A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution & is given the CPU.

Queuing diagram

A common representation of process scheduling is a **queuing diagram**.

Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues.

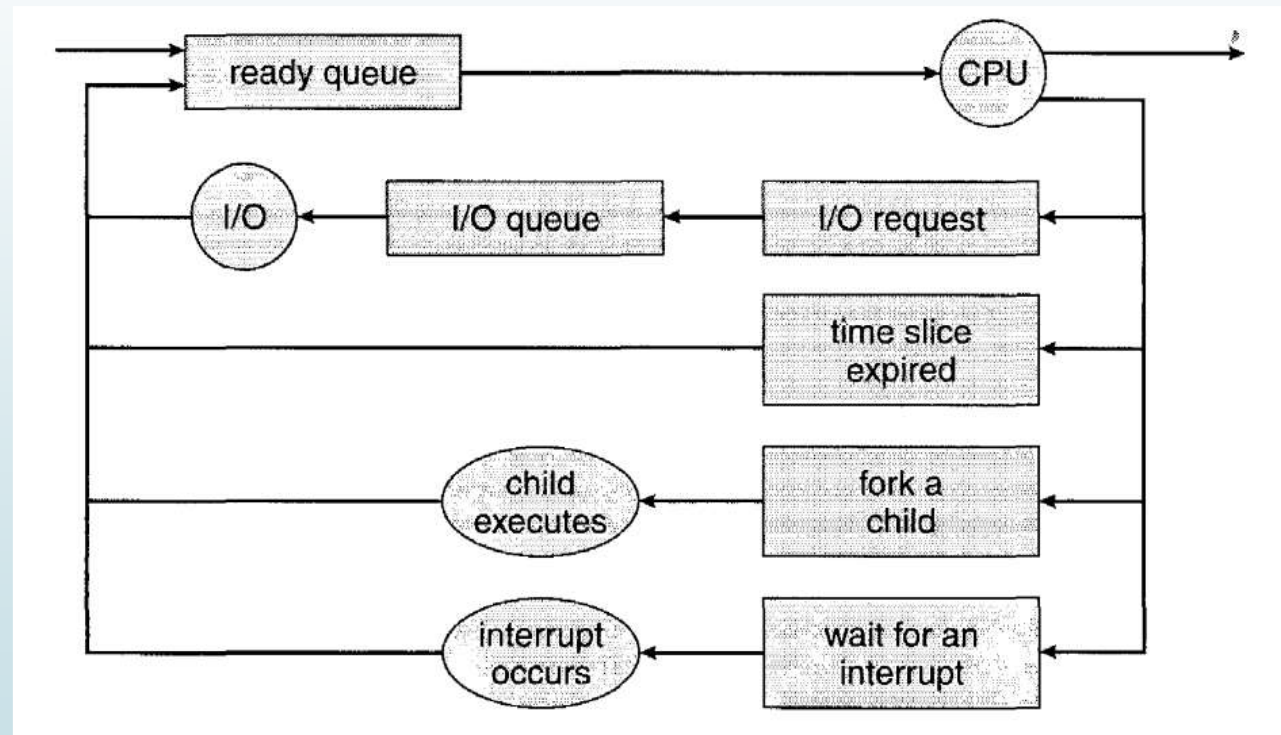
The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched.

Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Queuing diagram



Types of schedulers

Types of schedulers:

There are 3 types of schedulers mainly used:

► 1. Long term scheduler:

Long term scheduler selects process from the disk & loads them into memory for execution.

It controls the degree of multi-programming i.e. no. of processes in memory.

It executes less frequently than other schedulers. So, the long term scheduler is needed to be invoked only when a process leaves the system.

Most processes in the CPU are either I/O bound or CPU bound.

An I/O bound process is one that spends most of its time in I/O operation than it spends in doing computing operation.

A CPU bound process is one that spends more of its time in doing computations than I/O operations.

It is important that the long term scheduler should select a good mix of I/O bound & CPU bound processes.



Types of schedulers

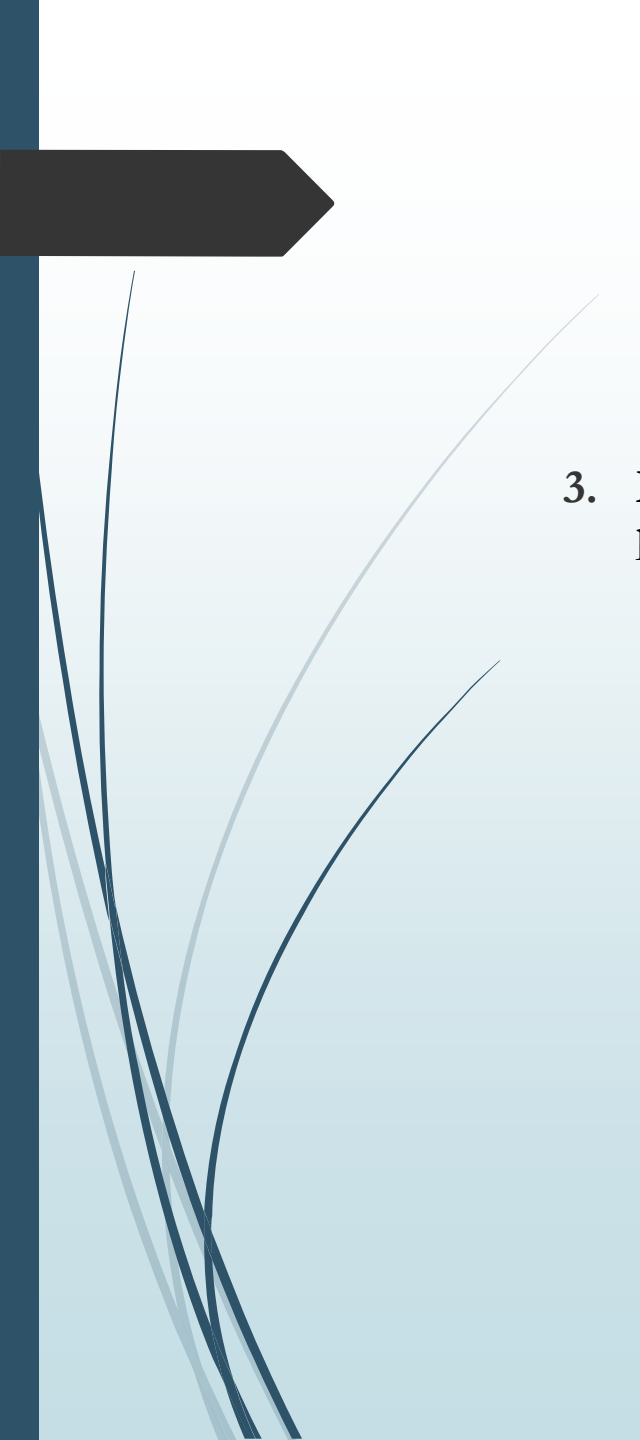
2. Short - term scheduler:

The short term scheduler selects among the process that are ready to execute & allocates the CPU to one of them.

The primary distinction between these two schedulers is the frequency of their execution.

The short-term scheduler must select a new process for the CPU quite frequently. It must execute at least one in 100ms.

Due to the short duration of time between executions, it must be very fast.

- 
3. **Medium - term scheduler:** some operating systems introduce an additional intermediate level of scheduling known as medium - term scheduler.

The main idea behind this scheduler is that sometimes it is advantageous to remove processes from memory & thus reduce the degree of multiprogramming.

At some later time, the process can be reintroduced into memory & its execution can be continued from where it had left off. This is called as swapping.

The process is swapped out & swapped in later by medium term scheduler.

Swapping is necessary when the available memory limit is exceeded which requires some memory to be freed up.



Context switch

Switching the CPU to another process requires
saving the state of the old process
loading the saved state for the new process.

This task is known as a **Context Switch**.

The **context** of a process is represented in the **Process Control Block(PCB)** of a process;

it includes the value of the CPU registers, the process state and memory-management information.

When a context switch occurs,

- the Kernel saves the context of the old process in its PCB
- loads the saved context of the new process scheduled to run.



Operations on processes

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically.
- Thus, these systems must provide a mechanism for process creation and termination.

Process creation

► Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a parent process, and the new processes are called the children of that process, forming a **tree** of processes

Most operating systems identify processes according to a unique process identifier (**pid**), which is typically an integer number.

► Resource sharing options

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

► Execution options

- Parent and children execute concurrently
- Parent waits until children terminate



Process termination

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **exit()** system call. At that point, the process may return a status value (typically an integer) to its parent process (via the `wait()` system call).

All the resources of the process—including physical and virtual memory, open files, and

I/O buffers—are deallocated by the operating system.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated.

- The task assigned to the child is no longer required.

- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.



Cascading termination

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated.

This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.



OPERATING SYSTEMS

Module2_Part3

Textbook : Operating Systems Concepts by Silberschatz



Interprocess communication

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

A process is ***independent*** if

- it cannot affect or be affected by the other processes executing in the system.
- it does not share data with any other process .

A process is ***cooperating*** if

- it can affect or be affected by the other processes executing in the system.
- it shares data with other processes



Interprocess communication

- Reasons for providing an environment that allows process cooperation:

Information sharing. Since 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. Notice that 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.



Interprocess communication

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

Two fundamental models of interprocess communication:

- shared memory**

- message passing.**

In the shared memory model

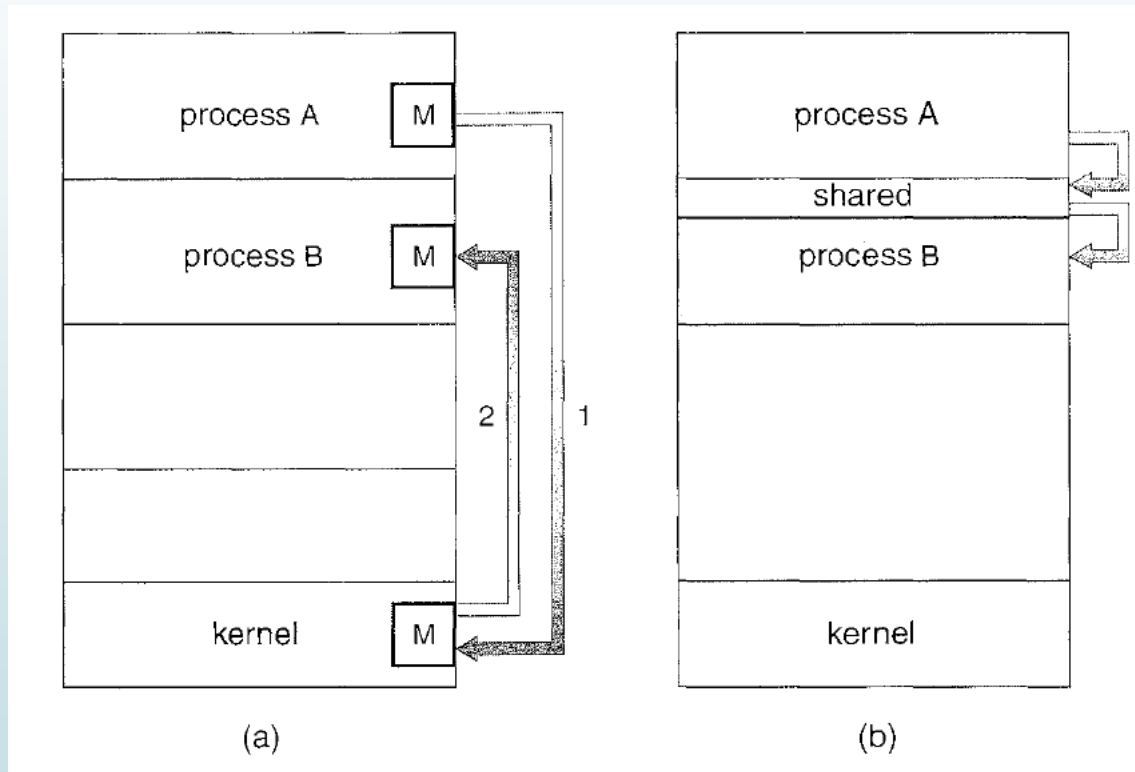
- a region of memory that is shared by cooperating processes is established.

- Processes can then exchange information by reading and writing data to the shared region.

In the message-passing model,

- communication takes place by means of messages exchanged between the cooperating processes.

Interprocess communication



a) Message passing b) shared memory

Shared memory and Message passing

► Message passing

- is useful for exchanging smaller amounts of data.
- is easier to implement than is shared memory for intercomputer communication.
- are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.

Shared memory allows

- maximum speed and convenience of communication.
- is faster than message passing, In contrast,
- system calls , only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.



Shared memory systems

- Interprocess communication using shared memory requires ,communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process, creating the shared memory segment.
- Other processes that wish to communicate using this shared memory segment must attach it to their address space.
- In shared memory system , cooperating processes can exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.



Shared memory systems

let's consider the **producer-consumer problem**, which is a common paradigm for cooperating processes. A **producer process produces information that is consumed by a consumer process.**

One solution to the producer–consumer problem uses shared memory. To allow producer and

consumer processes to run concurrently, we must have available a buffer of items that can be

filled by the producer and emptied by the consumer.

This buffer will reside in a region of memory that is shared by the producer and consumer processes.

A producer can produce one item while the consumer is consuming another item.

The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

The producer should produce data only when the buffer is not full.

A accessing memory buffer should not be allowed to producer and consumer at the same time



Shared memory systems

Two types of buffers can be used.

unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Shared memory systems

Bounded buffer , inter process communication using shared memory.

The following variables reside in a region of memory shared by the producer and consumer processes:

Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



Bounded buffer , inter process communication using shared memory.

The shared buffer is implemented as a circular array with two logical pointers: **in** and **out**.

in points to the next free position in the buffer;

out points to the first full position in the buffer.

The buffer is empty when **in == out**;

the buffer is full when **((in + 1)% BUFFER SIZE) == out**.

Producer consumer problem using shared memory

The producer process has a local variable **next produced** in which the new item to be produced is stored.

```
item next_produced;

while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

The producer process using shared memory.

Producer consumer problem using shared memory

The consumer process has a local variable **next consumed** in which the item to be consumed is stored.

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

The consumer process using shared memory.



OPERATING SYSTEMS

Module2_Part4

Textbook : Operating Systems Concepts by Silberschatz



Message passing

Here the operating system provides the means for cooperating processes to communicate with

each other via a message-passing facility.

Message passing provides a mechanism

- to allow processes to communicate
- to synchronize their actions without sharing the same address space

It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

A message-passing facility provides at least two operations:

`send(message)`

`receive(message).`

Messages sent by a process can be of either fixed or variable size.



Message passing

- If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them.

This link can be implemented in a variety of ways

Direct or indirect communication

Synchronous or asynchronous communication

Automatic or explicit buffering



Message passing

direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and

receive() primitives are defined as:

send(P, message)—Send a message to process P.

receive(Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.



Message passing

indirect communication, the messages are sent to and received from ***mailboxes***, or

ports.

A mail box can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

Each mailbox has a unique identification number.

A process can communicate with another process via a number of different mailboxes, but two

processes can communicate only if they have a shared mailbox.

The send() and receive() primitives are defined as follows:

- send(A, message)—Send a message to mailbox A.
- receive(A, message)—Receive a message from mailbox A.

Synchronous or asynchronous communication

Blocking(synchronous) or nonblocking(asynchronous)

- **Blocking** is considered **synchronous**
 - **Blocking send** -- the sender is blocked until the message is received
 - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** -- the sender sends the message and continue
 - **Non-blocking receive** -- the receiver receives:
 - A valid message, or
 - Null message

Message passing

- When both send() and receive() are blocking, we have a **rendezvous** between the sender and the receiver.

The producer-consumer problem

- Producer

```
message next_produced;
while (true) {
/* produce an item in next_produced */

    send(next_produced) ;
}
```

- Consumer

```
message next_consumed;
while (true) {
    receive(next_consumed)

/* consume the item in next_consumed */
}
```



buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
 1. Zero capacity – no messages are queued on a link.
Sender must wait for receiver
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits



OPERATING SYSTEMS

Module2_Part4

Textbook : Operating Systems Concepts by Silberschatz



Scheduling algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue

is to be allocated the CPU. There are many different CPU-scheduling algorithms. Some of

them are

First-Come, First-Served Scheduling

Shortest-Job-First Scheduling

shortest-remaining-time-first

Priority Scheduling

Round-Robin Scheduling



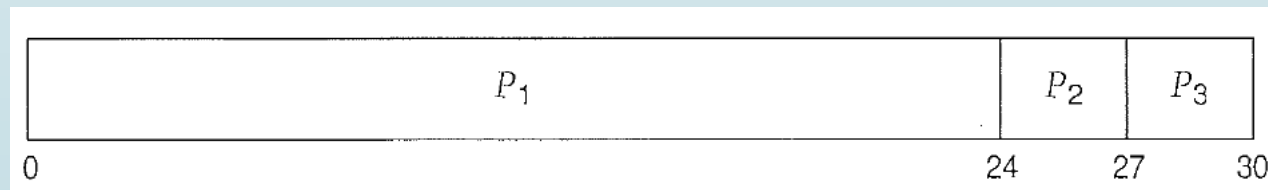
First-Come, First-Served Scheduling

- the simplest CPU-scheduling algorithm
- the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- The code for FCFS scheduling is simple to write and understand.

- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:
- Example with 3 processes

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

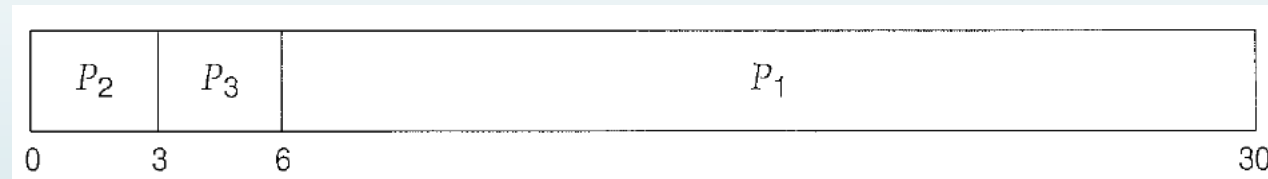
- Suppose that the processes arrive in the order: P_1, P_2, P_3
- The Gantt Chart for the above schedule is:(Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes)



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

First-Come, First-Served Scheduling

If the processes arrive in the order P_2 , P_3 , P_1 , however, the results will be as shown in the following Gantt chart:



The average waiting time is now $(6 + 0 + 3)/3 = 3$ milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.



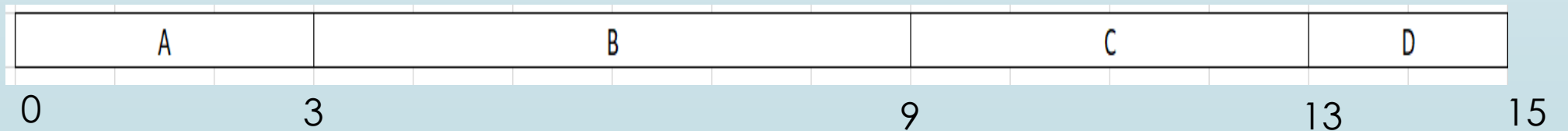
First-Come, First-Served Scheduling

- the FCFS scheduling algorithm is nonpreemptive.
- Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.
- The processes with higher burst time arrived before processes with smaller burst time, then smaller processes have to wait for a long time for longer processes to release cpu (Convoy effect)

First-Come, First-Served Scheduling Example

For the processes listed draw gantt chart illustrating their execution

process	Arrival time	Processing time
A	0	3
B	1	6
C	4	4
D	6	2

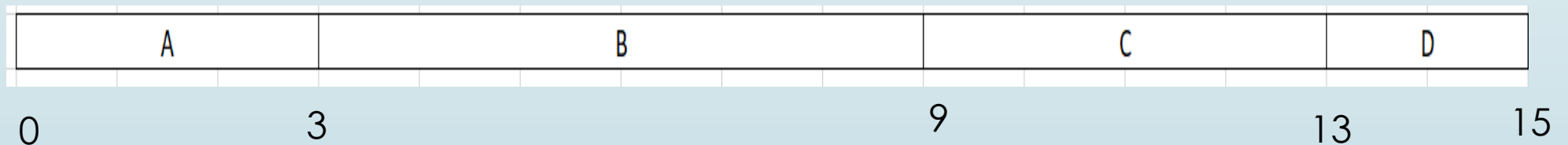


First-Come, First-Served Scheduling

Example

- For the process listed what is the average turn around time?

Process	Arrival time	Processing time	Completion time	Turn around time
A	0	3	3	3
B	1	6	9	8
C	4	4	13	9
D	6	2	15	9



Turn around time = completion time - arrival time

$$\text{Average turn around time} = ((3-0) + (9-1) + (13-4) + (15-6)) / 4 = 7.25$$

First-Come, First-Served Scheduling

Example

For the processes listed what is the waiting time for each process?

Process	Arrival time	Processing time	Completion time	Turn around time	Waiting time
A	0	3	3	3	0
B	1	6	9	8	2
C	4	4	13	9	5
D	6	2	15	9	7

Waiting time = turn around time – execution time

A: $(3-3)=0$ B: $(8-6)=2$ C: $(9-4)=5$ D: $(9-2)=7$



OPERATING SYSTEMS

Module2_Part6

Textbook : Operating Systems Concepts by Silberschatz



Scheduling algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue

is to be allocated the CPU. There are many different CPU-scheduling algorithms. Some of

them are

First-Come, First-Served Scheduling

Shortest-Job-First Scheduling

shortest-remaining-time-first

Priority Scheduling

Round-Robin Scheduling

Shortest job first scheduling algorithm

The shortest-job-first (SJF) scheduling algorithm.

- associates with each process the length of the process's next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

Assumption: run time for processes are known in advance

Shortest Job First yields smallest average turnaround time, if all jobs are available simultaneously.



SJF

- SJF is an optimal algorithm because it decreases the wait times for short processes much more than it increases the wait times for long processes.
It gives minimum turn around time

Consider the case of 4 jobs, with run times of a,b,c,and d respectively.

The first job finishes with time a,the second job finishes with time a+b and so on.

The average turn around time $= (4a+3b+2c+d)/4$. It is clear that 'a' contributes to the average

than the other times, so it should be the shortest job ,with b next, then c and so on. So we

can say that SJF is optimal

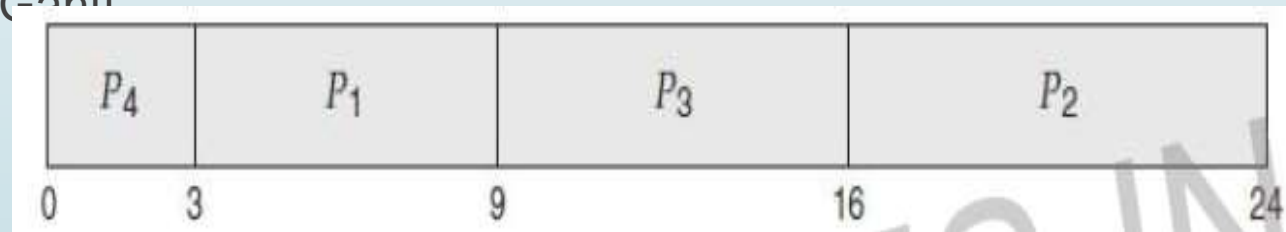
SJF

- As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- Using SJF scheduling, we would schedule these processes according to the following Gantt

- chart:



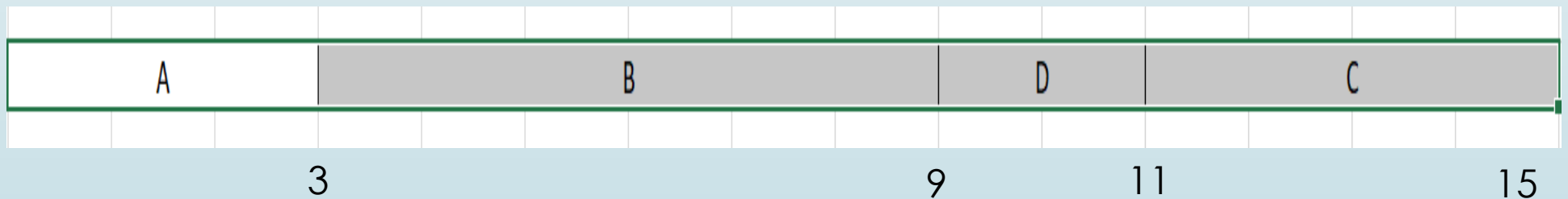
The waiting time is 3 milliseconds for process P_1 , 16 milliseconds for process P_2 , 9

milliseconds for process P_3 , and 0 milliseconds for process P_4 . Thus, the average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time

SJF

For the processes listed draw gantt chart illustrating their execution

process	Arrival time	Processing time
A	0	3
B	1	6
C	4	4
D	6	2

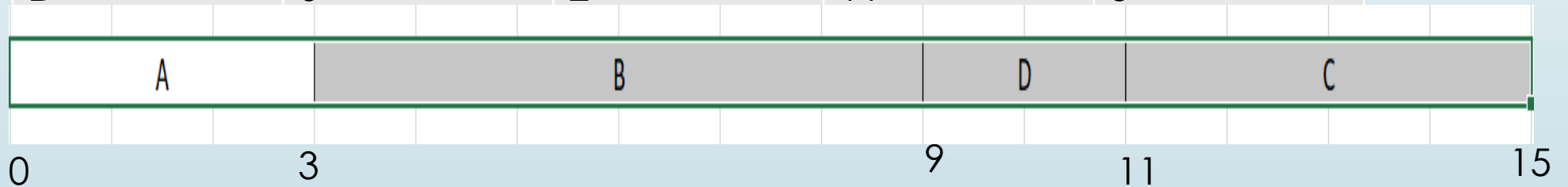


Process A start executing: It is the only choice at time 0 . At time 3, B is the only choice .At time 9, B completes, process D runs because D is shorter than process C

SJF

- For the process listed what is the average turn around time?

Process	Arrival time	Processing time	Completion time	Turn around time
A	0	3	3	3
B	1	6	9	8
C	4	4	15	11
D	6	2	11	5



Turn around time=completion time –arrival time

Average turn around time= $((3-0)+(9-1)+(15-4)+(11-6))/4 = 6.75$

SJF

- For the processes listed what is the waiting time for each process?

Process	Arrival time	Processing time	Completion time	Turn around time	Waiting time
A	0	3	3	3	3
B	1	6	9	8	8
C	4	4	15	11	11
D	6	2	11	5	5

Waiting time=turn around time – execution time

A:(3-3)=0 B:(8-6)=2 C:(11-4)=7 D:(5-2)=3



fork() system call

fork() system call is used to create child processes in a C program.

It takes no arguments and returns a process ID.

After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call.

Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**.

If **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.

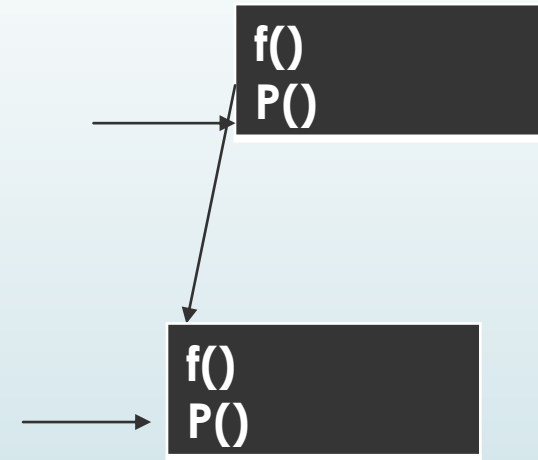
fork() system call

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

output

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

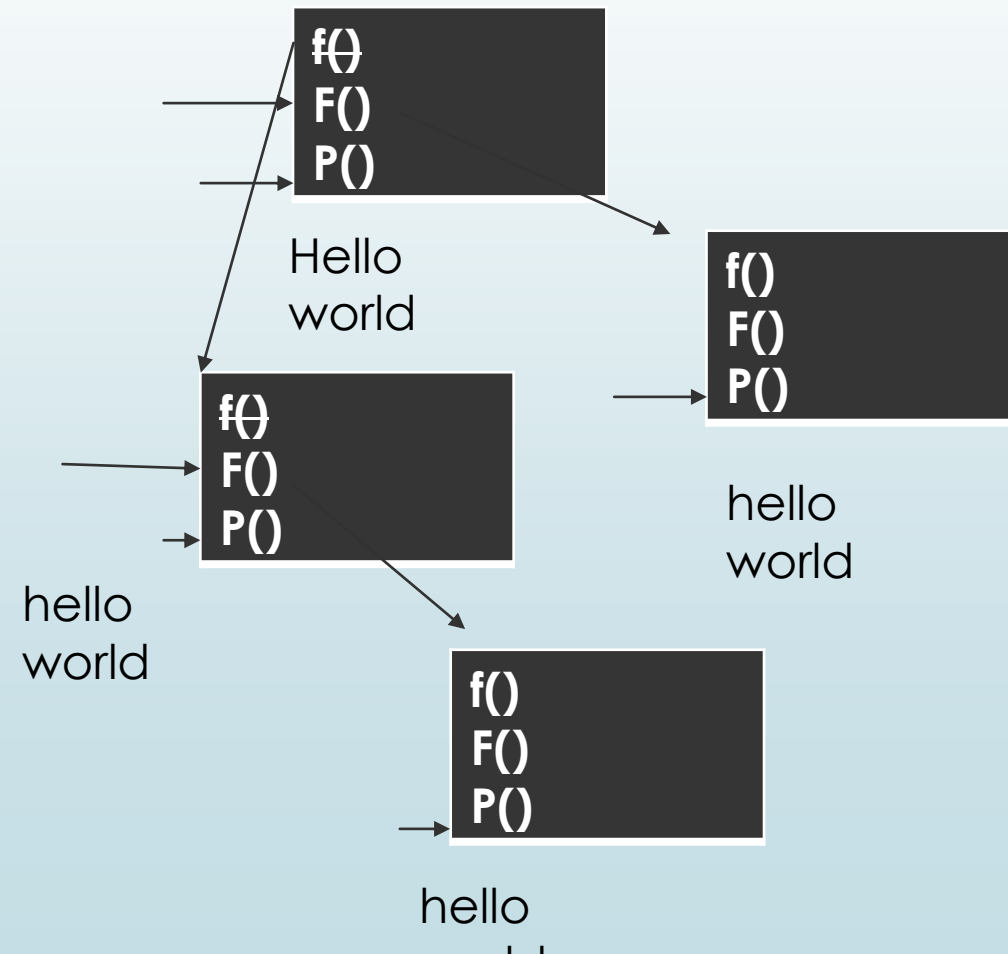


fork() system call

► Consider this code

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
main()
{
    fork();
    fork();
    printf("hello world");
}
```

Four times hello world will be printed



getpid() system call

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    //variable to store calling function's process id
    pid_t process_id; // pid_t unsigned integer type
    //variable to store parent function's process id
    pid_t p_process_id;

    //getpid() - will return process id of calling function
    process_id = getpid();
    //getppid() - will return process id of parent function
    p_process_id = getppid();

    //printing the process ids
    printf("The process id: %d\n", process_id);
    printf("The process id of parent function: %d\n", p_process_id);
    return 0;
}
```

Output

The process id: 31120

The process id of parent function: 31119



Exec system call

Exec system call

The `exec()` system call is used to execute a file which is residing in an active process. When `exec()` is called the previous executable file is replaced and new file is executed.

process id will be the same.

Exec() system call

- There are two programs ex1.c ex2.c

Ex1.c

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
#include <stdlib.h>
```

```
Int main(int argc, char *argv[])
```

```
{
```

```
printf("Pid of ex1.c=%d\n",getpid());
```

```
Char *args[] ={hello",NULL};
```

```
execv("./ex2",args);
```

```
printf("Back to Ex1.c");
```


```
Return 0;
```

```
}
```

Exec sytem call

► Ex2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
Int main(int argc,char *argv[])
{
printf("We are in ex2.c\n");
printf("Pid of ex2.c=%d\n",getpid());
Return 0;
}
```

- 
- Compile these two programs
 - `gcc ex1.c -o ex1`
 - `gcc ex2.c -o ex2`
 - Run the first program `./ex1`

 - Pid of ex1.c=5962
 - We are in ex2.c
 - Pid of ex2.c=5962

Wait system call

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main()
{
    pid_t q;
    q=fork();
    if(q==0)//child
    {
        printf("I am a child having Id %d/n",getpid());
        printf("My parent's id is %d\n",getppid());
    }
    else{//parent
        printf(" My child's id is %d/n",q);
        printf("I am parent having id %d\n",getpid());
    }
    printf("Common");
}
```

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.



Wait system call

- Output may be
 - My child's id is 188
 - I am a child having Id 188
 - I am parent having id 157
 - My parent's id is 157
 - Common
 - Common

Wait system call

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

int main()
{
    pid_t q;
    q=fork();
    if(q==0)//child
    {
        printf("I am a child having Id %d\n",getpid());
        printf("My parent's id is %d\n",getppid());
    }
    else{//parent
        wait(NULL);
        printf(" My child's id is %d\n",q);
        printf("I am parent having id %d\n",getpid());
    }
    printf("Common");
}
```



Wait system call

when compiles and run

I am a child having Id 256

My parent's id is 255

Common

My child's id is 256

I am parent having id 255

Common

exit()

- It deletes all buffers and closes all open files before ending the program.

```
// C program to illustrate exit() function.
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("START");

    exit(0); // The program is terminated here

    // This line is not printed
    printf("End of program");
}
```

Output
START



OPERATING SYSTEMS

Module2_Part7

Textbook : Operating Systems Concepts by Silberschatz



Scheduling algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue

is to be allocated the CPU. There are many different CPU-scheduling algorithms. Some of

them are

First-Come, First-Served Scheduling

Shortest-Job-First Scheduling

shortest-remaining-time-first

Priority Scheduling

Round-Robin Scheduling



Shortest remaining time first scheduling algorithm

- The SJF algorithm can be either preemptive or nonpreemptive.
- The choice arises when a new process arrives at the ready queue while a previous process is still executing.
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
- A preemptive SJF algorithm will preempt the currently executing process, and allow the shorter process to run whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.

SRTF

- As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

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

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:

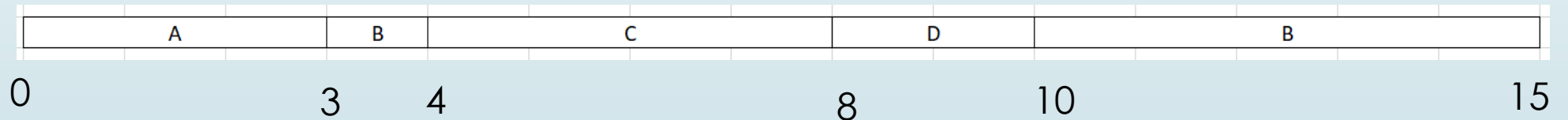


Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)] / 4 = 26 / 4 = 6.5$ milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

SRTF

For the processes listed draw gantt chart illustrating their execution

process	Arrival time	Processing time
A	0.000	3
B	1.001	6
C	4.001	4
D	6.001	2

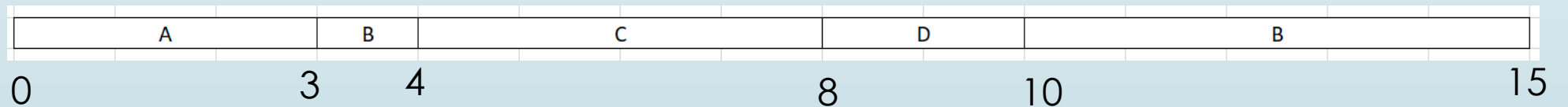


Process A start executing at time 0. It remains running when process B arrives because remaining time is less. At time 3 process B is the only process in the queue. At time 4.001, process C arrives and start running because its remaining time(4) is less than B's remaining time(4.999), At 6.001 process B remains running because its remaining time (1.999) is less than D's remaining time. When process C terminates process D runs its remaining time is less than that of process B. Then process B runs

SRTF

- For the process listed what is the average turn around time?

Process	Arrival time	Processing time	Completion time	Turn around time
A	0.000	3	3	3
B	1.001	6	15	14
C	4.001	4	8	4
D	6.001	2	10	4



Turn around time=completion time –arrival time

Average turn around time= $((3-0)+(15-1)+(8-4)+(10-6))/4 = 6.25$

SRTF

- For the processes listed what is the waiting time for each process?

Process	Arrival time	Processing time	Completion time	Turn around time	Waiting time
A	0.001	3	3	3	0
B	1.001	6	15	14	8
C	4.001	4	8	4	0
D	6.001	2	10	4	2

Waiting time=turn around time – execution time

A:(3-3)=0 B:(14-6)=8 C:(4-4)=0 D:(4-2)=2



OPERATING SYSTEMS

Module2_Part8

Textbook : Operating Systems Concepts by Silberschatz



Scheduling algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue

is to be allocated the CPU. There are many different CPU-scheduling algorithms. Some of

them are

First-Come, First-Served Scheduling

Shortest-Job-First Scheduling

shortest-remaining-time-first


Priority Scheduling

Round-Robin Scheduling



Priority scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (usually, smallest integer \equiv highest priority)
- Two schemes:
 - Preemptive
 - Nonpreemptive
- Problem \equiv **Starvation** – low priority processes may never execute
- Solution \equiv **Aging** – as time progresses increase the priority of the process
- Note: SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

- 
- As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P_1, P_2, \dots, P_5 , with the length of the CPU burst given in milliseconds: assume that low numbers represent high priority.

Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

► Priority scheduling Gantt Chart



► Average waiting time = 8.2



Priority scheduling

- A major problem with priority scheduling algorithms is starvation.
- A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU

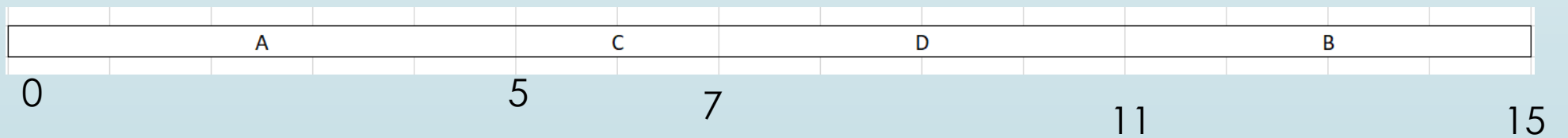
A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

Example

simple case -non preemptive

Assume large number high priority in this case

process	Arrival time	Burst time	priority
A	0.0000	5	4
B	2.0001	4	2
C	2.0001	2	6
D	4.0001	4	3



Waiting time=turn around time – execution time

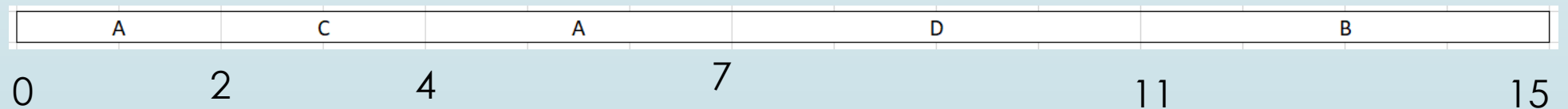
A:(5-5)=0 B:(13-2)=11 C:(5-2)=3 D:(7-4)=3 average waiting time = 3.75

Example

simple case - preemptive

Assume large number high priority in this case

process	Arrival time	Burst time	priority
A	0.0000	5	4
B	2.0001	4	2
C	2.0001	2	6
D	4.0001	4	3



Waiting time=turn around time – execution time

5

A:(7-5)=2 B:(13-4)=9 C:(2-2)=0 D:(7-4)=3 average waiting time =3.5



OPERATING SYSTEMS

Module2_Part9

Textbook : Operating Systems Concepts by Silberschatz



Scheduling algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue

is to be allocated the CPU. There are many different CPU-scheduling algorithms. Some of

them are

First-Come, First-Served Scheduling

Shortest-Job-First Scheduling

shortest-remaining-time-first

Priority Scheduling

Round-Robin Scheduling



Round robin scheduling

- Round Robin(RR) scheduling algorithm is mainly designed for time-sharing systems. This algorithm is similar to FCFS scheduling, but in Round Robin(RR) scheduling, preemption is added which enables the system to switch between processes.
- A small unit of time, called a **time quantum** or time slice, is defined. Each process is assigned a time slice or time quantum for execution.
- Once a process is executed for the given time period, that process is preempted and another process executes for the given time period.



Round robin

To implement RR scheduling,

we keep the ready queue as a FIFO queue of processes.

New processes are added to the tail of the ready queue.

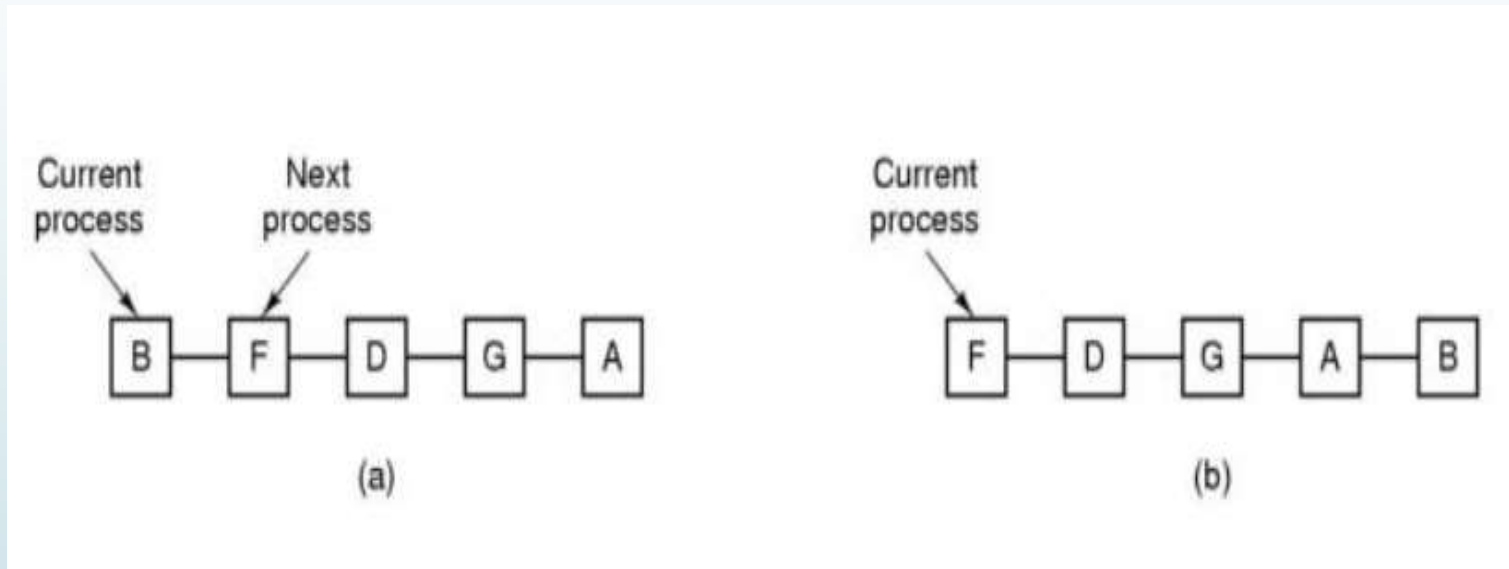
The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen.

The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.

if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue

Round robin scheduling



←
Front of the
queue

When B completes its time slice, put back to the tail of the queue. F will run. When F completes F put back to the tail of the queue. Then D runs and so on

Round robin scheduling

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

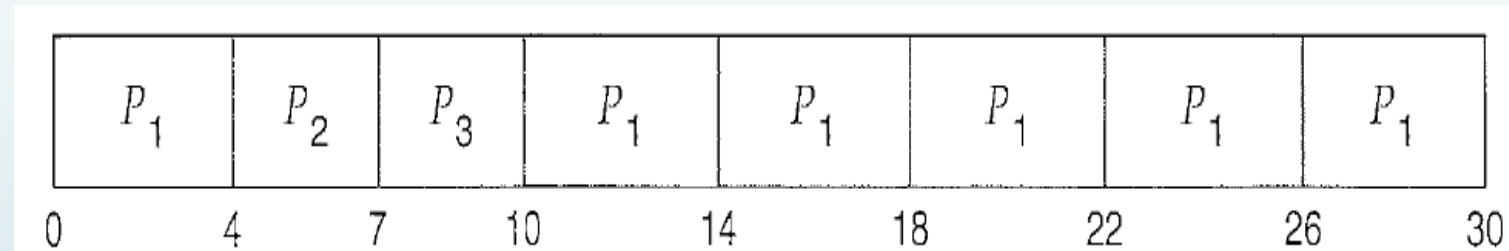
- Example with 3 processes

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. it requires another 20 milliseconds, it is preempted after the first time quantum
- CPU is given to the next process in the queue, process P_2 . Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires.
- The CPU is then given to the next process, process P_3 . Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum.

Round robin scheduling

➡ The resulting RR schedule is as follows:



Let's calculate the average waiting time for the above schedule. P_1 waits for 6 milliseconds ($10 - 4$), P_2 waits for 4 milliseconds, and P_3 waits for 7 milliseconds. Thus, the average waiting time is $17/3 = 5.66$ milliseconds.

Round robin scheduling

- If there are n processes in the ready queue and the time quantum is q , then each process gets 1 *timeslice* of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum.

At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy.

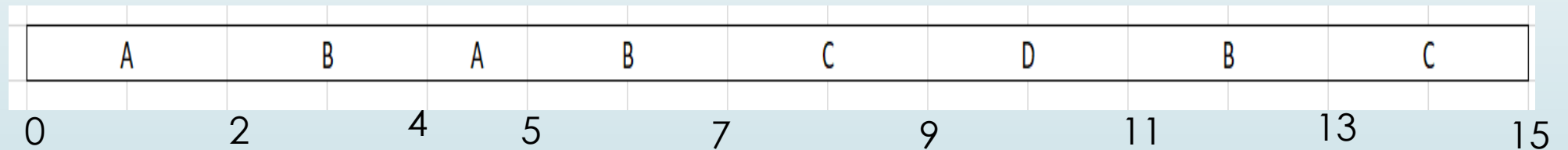
Setting too short causes too many process switched and lowers cpu efficiency

We have to take time quantum in between

Round robin scheduling

For the processes listed draw gantt chart illustrating their execution(quantum=2)

process	Arrival time	Processing time
A	0.000	3
B	1.001	6
C	4.001	4
D	6.001	2

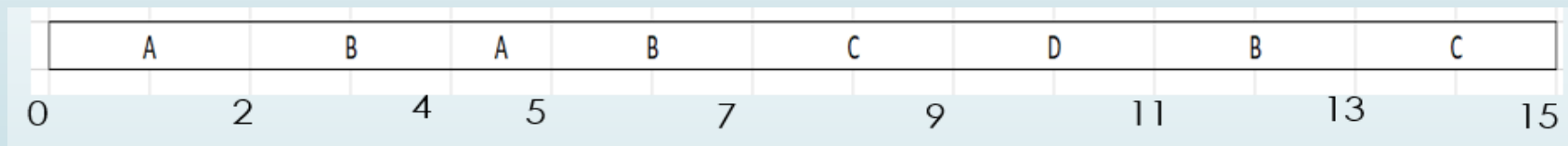


When Process A's first time quantum expires process B runs. At time 4 process A restarts process B returns to the ready queue. At time 4.001 process C enters the ready queue after B. At time 6 process D enters the ready queue after C. Starting at &process C D B and C runs in sequence.

Round robin scheduling

- For the process listed what is the average turn around time?

Process	Arrival time	Processing time	Completion time	Turn around time
A	0.000	3	5	5
B	1.001	6	13	12
C	4.001	4	15	11
D	6.001	2	11	5



Turn around time = completion time - arrival time

Average turn around time = $((5-0) + (13-1) + (15-4) + (11-6)) / 4 = 8.25$

SRTF

- For the processes listed what is the waiting time for each process?

Process	Arrival time	Processing time	Completion time	Turn around time	Waiting time
A	0.001	3	5	5	0
B	1.001	6	13	12	8
C	4.001	4	15	11	0
D	6.001	2	11	5	2

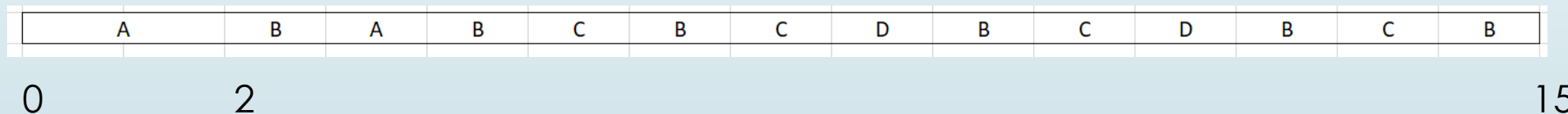
Waiting time=turn around time – execution time

A:(5-3)=2 B:(12-6)=6 C:(11-4)=7 D:(5-2)=3

Round robin scheduling

For the processes listed draw gantt chart illustrating their execution(quantum=1)

process	Arrival time	Processing time
A	0.000	3
B	1.001	6
C	4.001	4
D	6.001	2



Remember here A runs two times, At 1 A runs again. B does not arrive until 1.001

Do as an exercise average turn around time and waiting time

Time Quantum and Context Switch Time

The effect of context switching on the performance of RR scheduling.

Assume, that we have only one process of 10 time units.

If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.

If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch.

If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly

Although the time quantum should be large compared with the context switch time, it should not be too large. Time quantum q should be large compared to context switch time. q usually 10 milliseconds to 100 milliseconds, Context switch < 10 microseconds

