Operating Systems & Systems Programming
Module 3
Operating System Process

Dr. Vikash

Jaypee Institute of Information Technology, Noida

Overview

Process Concept

Definition

- It is an instance or a portion of a program which is getting executed in the processor at the moment
- Executing a process means executing instructions in a sequence.
- Each process has a current state, and a set of system resources acquired by it

Process Concept

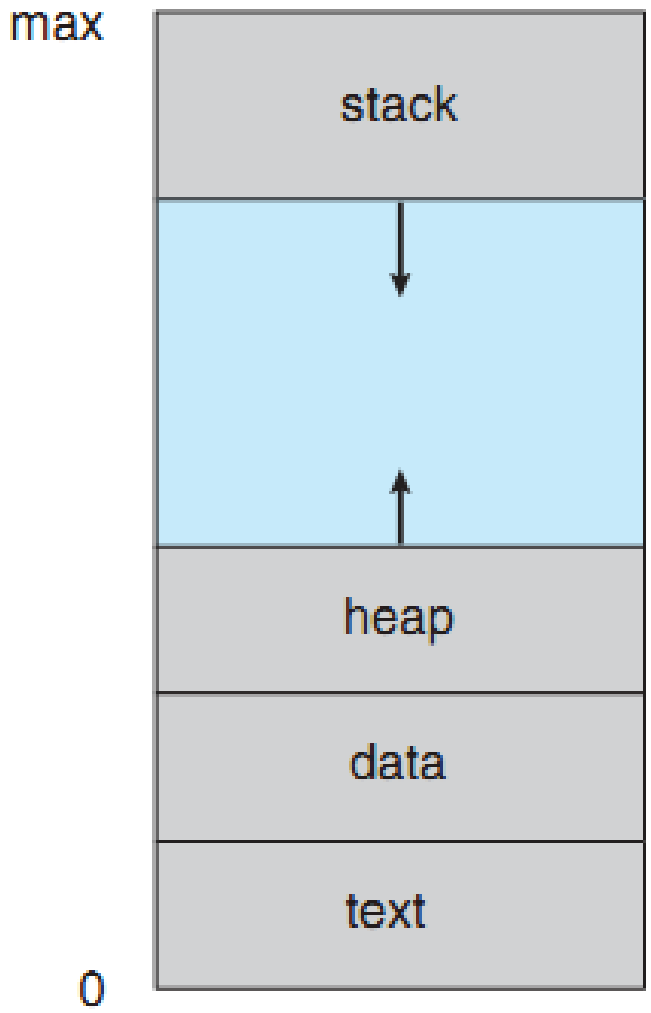
- Types of systems
  - Batch system: Where computational tasks are completed by processors in the form of Jobs.
  - Time-shared systems: Processor is shared among applications.
- process can also be termed as job
- A program is not a process.
- A program is a passive entity, for example a le with a list of instructions stored on disk. Whereas, a process is an active part of a process.
- A program counter specifies the next instruction to execute and a set of associated resources. When an executable le is loaded into RAM, the program becomes a process.

Process Structure

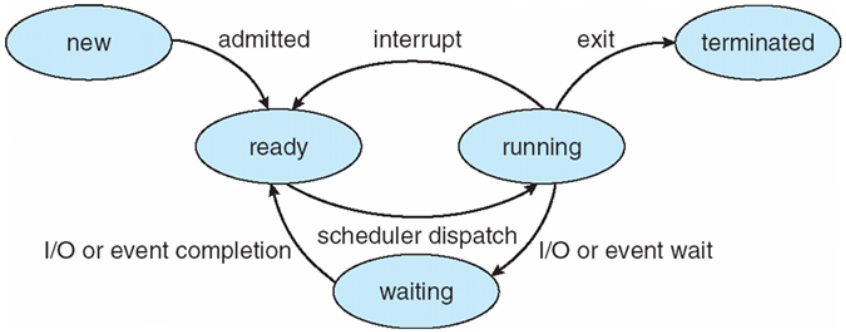
- A process is more then the program code, which is sometimes known as the text section.
- It also includes the current activity:
  - The value of the **program counter**
  - The contents of the **processor's registers**
- It also includes the process **stack**, which contains temporary data.
- It also include the **data section**, which contains global variables.
- It may also include a **heap**, which is memory that is dynamically allocated during process runtime.

A Process in Memory

- A Process consists of following sections in the memory:
  - The code or instructions, also called text section
  - Program counter and registers maintain current activities of a process
  - Stack stores temporary data of a process.
  - For example local variables, function arguments or return addresses
  - Data section stores global variables
  - Memory allocated dynamically during run time is stored in Heap



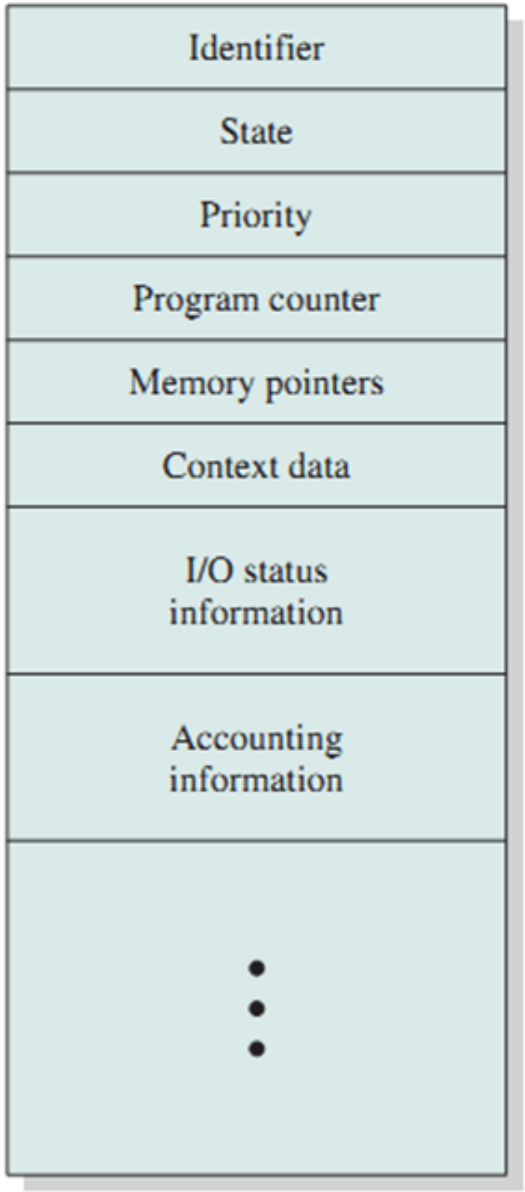
Process Stages



- **New:** When a process is being created.
- **Waiting:** When the process is waiting for some I/O resource or an event to occur like getting input from keyboard, a message from another process, to access disk storage or an another process to finish a task, and so on.
- **Ready:** The process is in queue having all the required resources and waiting to get a processor
- **Running:** When CPU is executing process instructions
- **Terminated:** When process finishes execution of all its instructions

Process Control Block

- Each process has a PCB or a Task Control Block.
- Each process can be uniquely identified by a no. of parameters
- A process execution can be interrupted and later resumed
- It stores process specific information, like
  - State: State of a process can be ready, waiting, running, etc
  - Program counter: It stores location of next instruction to execute
  - CPU scheduling: it provides relative priorities of process and pointers to queue
  - CPU registers: It stores information/ data associated with process



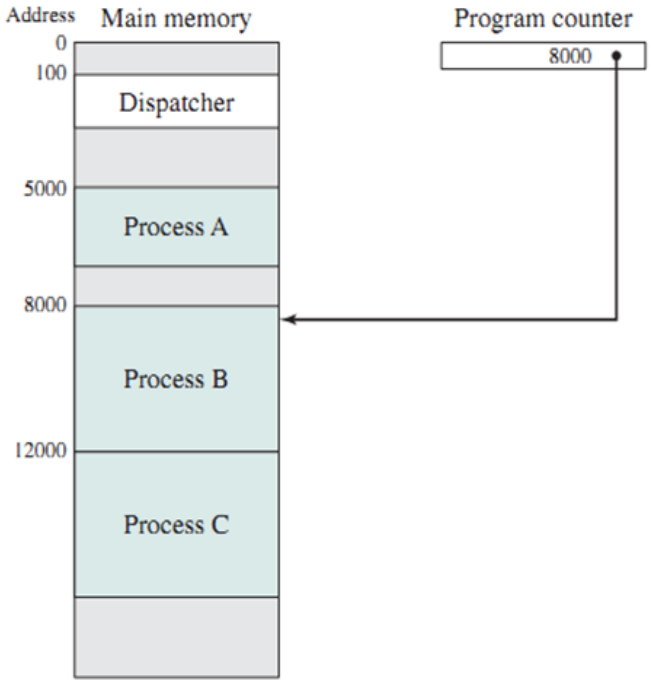
Process Control Block (CONTD.)

- Memory-management information: Size of memory and starting address allocated to the process
- Usage/Accounting information: CPU utilization, turn around time, etc
- I/O status: resources or devices allocated to a process
- Identifier: Each process has a unique id
- Memory pointers: It stores the address of the program code and data related to process
- Priority: Priority of a process compared to other

Trace and Dispatcher

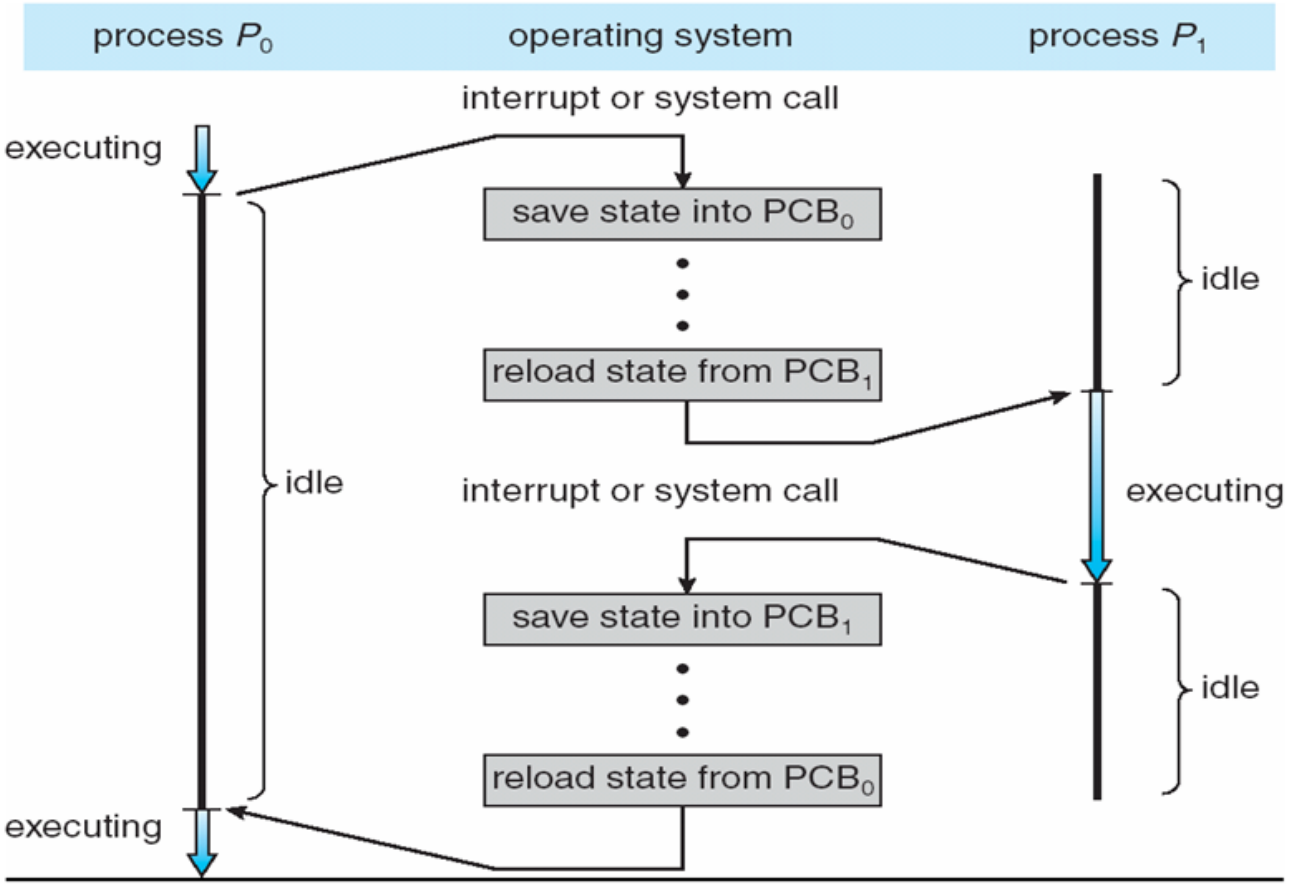
(a) Trace of process A	(b) Trace of process B	(c) Trace of process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

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



- TRACE monitors and notes execution sequence of instructions. It also keeps record of process interleaving while executing.
- DISPATCHER is a small program in operating system that helps in switching processor the from one process to another

Switching Between Processes



Operations in Processes

Majorly user performed two operation in context of processes:

- **Creation:** This the initial step of process execution activity. Process creation means the construction of a new process for the execution. This might be performed by system, user or old process itself.
- **Termination:** Process termination is the activity of ending the process. In other words, process termination is the relaxation of computer resources taken by the process for the execution.

Apart from it there are several operation involved with the processes.

Operations in Processes

- **Scheduling/Dispatching:** The event or activity in which the state of the process is changed from ready to running.
- **Blocking:** When a process invokes an input-output system call that blocks the process and operating system put in block mode.
- **Preemption:** When a timeout occurs that means the process hadn't been terminated in the allotted time interval and next process is ready to execute, then the operating system preempts the process.

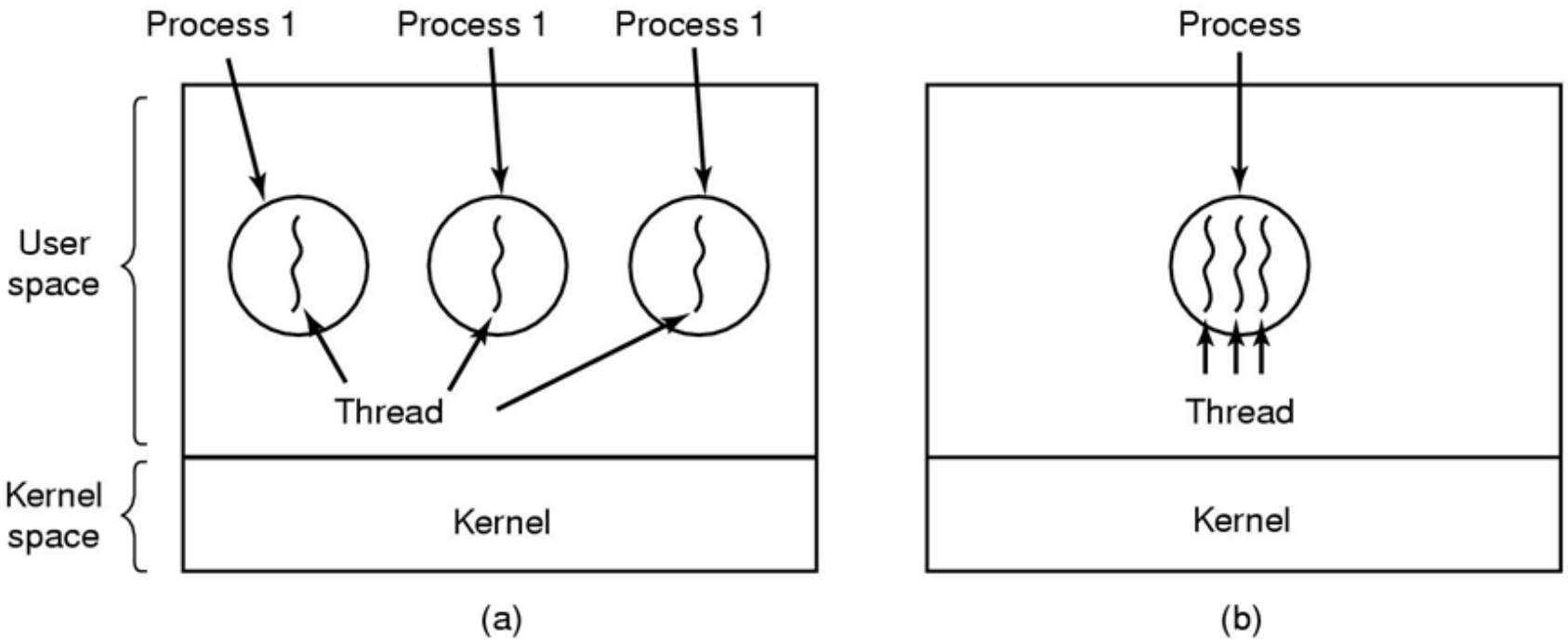
Threads

- A process can be divided into multiple light weight executable entities to parallelize the task.
- These are called Threads.
- Suppose a process consists of multiple program counters (PC)
- Code of a process can be executed from multiple locations or instructions

Process vs Thread

- The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.
- Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.

Threads Model



- Three processes each with one thread
- One process with three threads

Benefits of Threads

- **Responsiveness:** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
- **Faster context switch:** Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
- **Effective utilization of multiprocessor system:** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.
- **Resource sharing:** Resources like code, data, and files can be shared among all threads within a process.
- **Communication:** Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process.
- **Enhanced throughput of the system:** If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

Need of Threads

Threads is able to divide the whole process into multiple tasks:

- **Data Parallelism:** distributes subsets of the same data across multiple cores, same operation on each.
- **Task Parallelism:** distributing threads across cores, each thread performing unique operation.

As number of threads grows, so does architectural support for threading

- CPUs have cores as well as hardware threads
- Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components.
- S is serial portion
- N processing cores

$$S = \frac{1}{S + \frac{(1-S)}{N}} \tag{1}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Fork()

In computing, the fork is an operation whereby a process creates a copy of itself, which is usually called system call created in kernel. Inother words we can say that fork is the primary method of process creation.

- **Purpose of fork():-**
  - It will create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call.
  - Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child process have separate address space.

exec()

In computing, exec is a functionally of an operating system that runs an executable file in the context of an already existing process, replacing the previous executable.

- **Purpose of exec():-**
  - It will create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call.
  - Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child process have separate address space.
  - Forking is enough to create the process but not enough to run a program, to do that the forked child needs to overwrite its own image with the code and data of the new program and mechanism in exec() creation.

Types of Threads

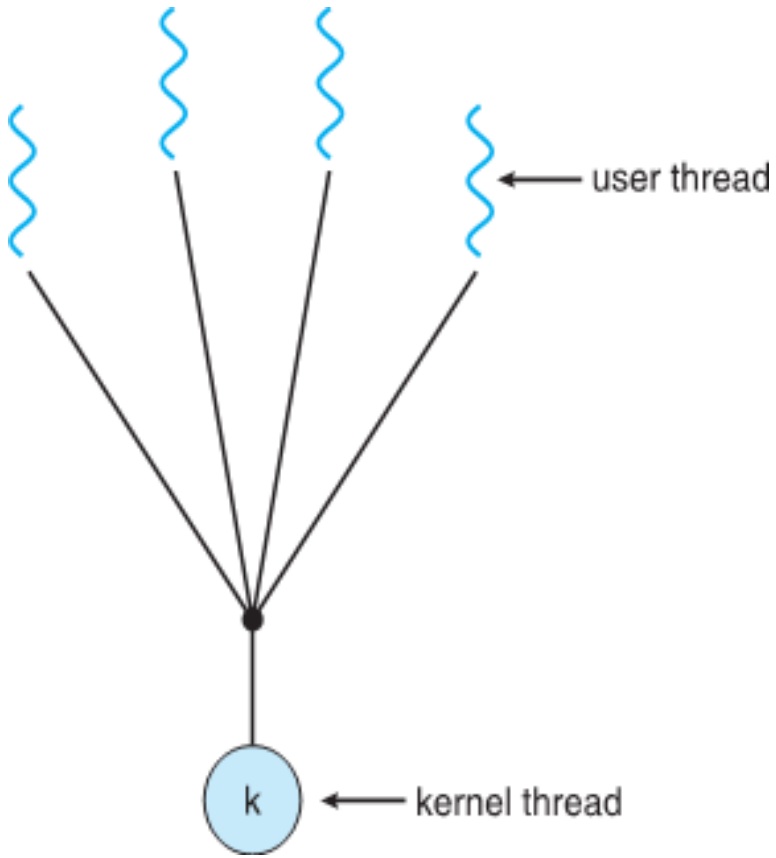
There are two types of threads

- **User Level Thread**– management done by user-level threads library.
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads
- **Kernel Level Thread**– Supported by the Kernel.
- Examples virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multi threading Models

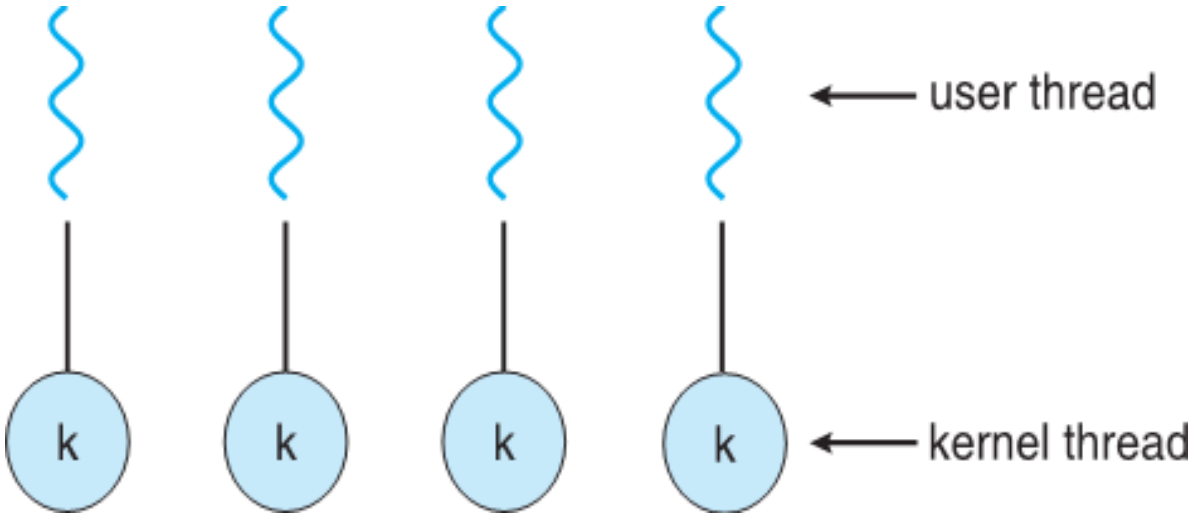
- Many to One
- One to One
- Many to Many

Many-to-One



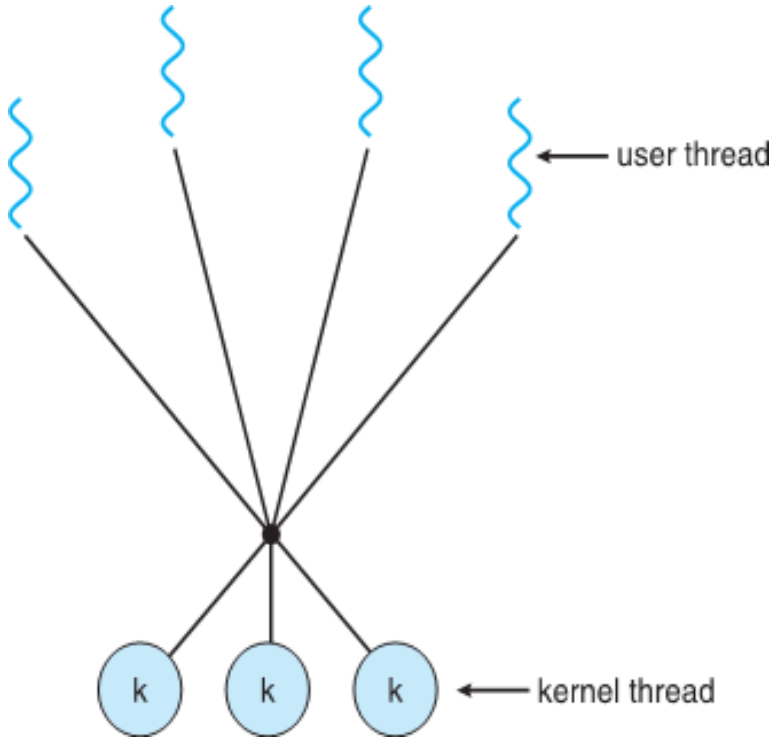
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- **Example:-**
  - Solaris Green Threads
  - GNU Portable Threads

One-to-One



- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

One-to-One



- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

Cooperative Processes: Inter Process Communication

Definition

- Cooperating processes are those that can affect or are affected by other processes running on the system.
- Cooperating processes may share data with each other.

Need of cooperating processes

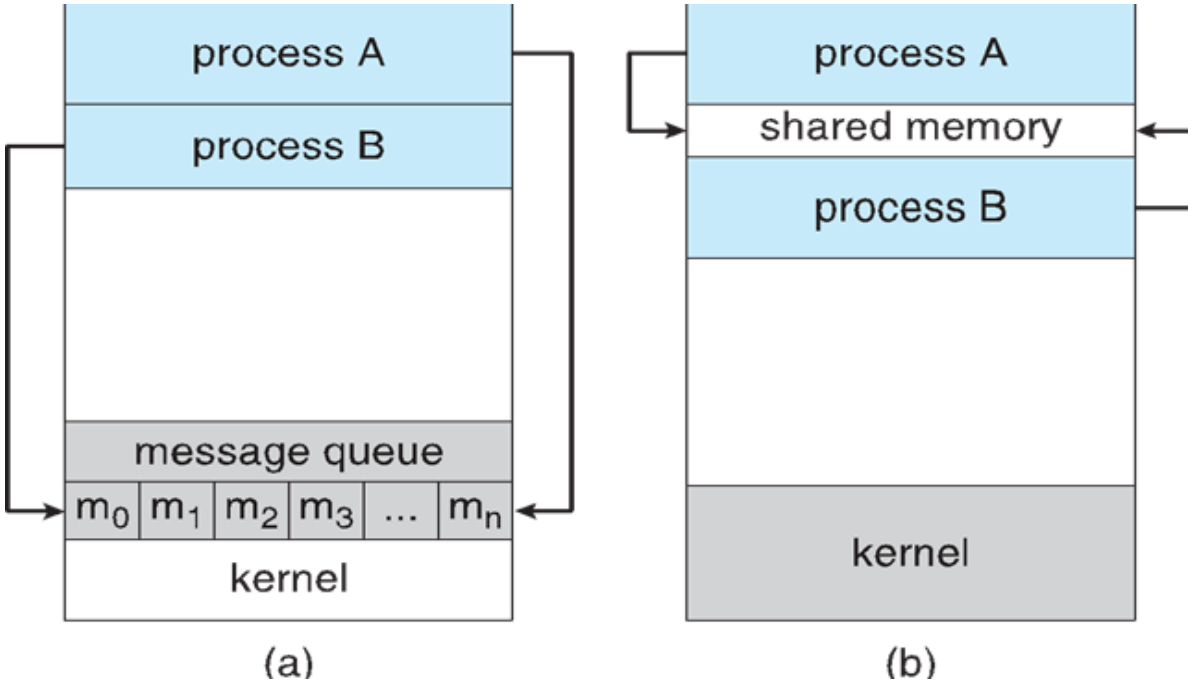
- **Modularity** involves dividing complicated tasks into smaller subtasks. These subtasks can completed by different cooperating processes. This leads to faster and more efficient completion of the required tasks.
- **Information sharing** between multiple processes can be accomplished using cooperating processes. This may include access to the same files. A mechanism is required so that the processes can access the files in parallel to each other.
- **Convenience:** There are many tasks that a user needs to do such as compiling, printing, editing etc. It is convenient if these tasks can be managed by cooperating processes.
- **Computation Speedup:** Subtasks of a single task can be performed parallely using cooperating processes. This increases the computation speedup as the task can be executed faster. However, this is only possible if the system has multiple processing elements.

Cooperative Process: Inter Process Communication

- **Independent process** cannot affect or be affected by the execution of another process
- **Cooperating process** can affect or be affected by the execution of another process
  - Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Cooperative Processes: Method

- **Cooperation by Sharing:** The cooperating processes can cooperate with each other using shared data such as memory, variables, files, databases etc. Critical section is used to provide data integrity and writing is mutually exclusive to prevent inconsistent data.
- **Cooperation by Communication:** The cooperating processes can cooperate with each other using messages. This may lead to deadlock if each process is waiting for a message from the other to perform a operation. Starvation is also possible if a process never receives a message.



Process Synchronization: Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Problem Example

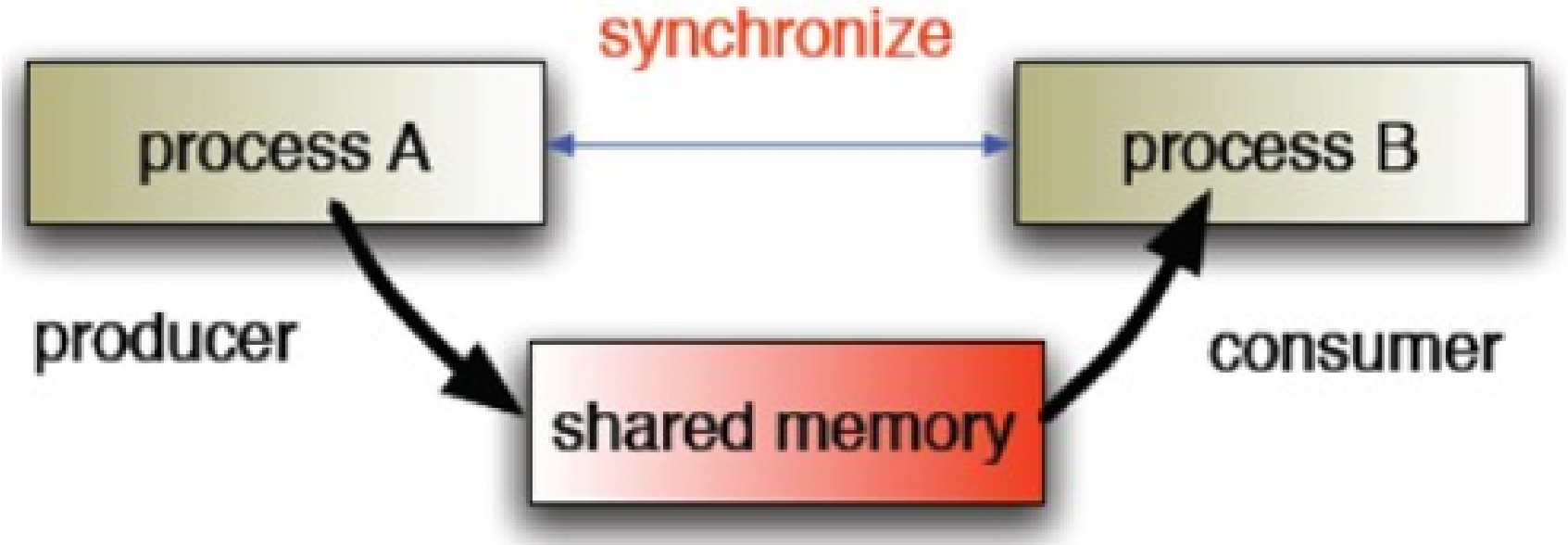
Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Race condition

When more than one processes are executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing race to say that my output is correct this condition known as race condition.

Shared Memory: Producer-Consumer Problem

- A producer process produces information which is consumed by a consumer process.
- Both share a common buffer which can be of two types:
  - **unbounded-buffer:** Unlimited size of buffer
  - **bounded-buffer:** limited buffer size



Producer-Consumer Problem: Solution

Figure: Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Figure: Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Message Passing: Cooperative Communication

- Mechanism for processes to communicate and to synchronize their actions
- Message system processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send(message)**
  - **receive(message)**
- Message can be of either fixed or variable size
- Processes need to establish a communication link between them and use provided operations
- Each call requires marshalling and de-marshalling of information.
- Message passing is useful for exchanging smaller amounts of data.

Message Passing: Cooperative Communication

- If processes P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive
- **Implementation issues:**
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Implementation of communication link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Message Passing: Direct Communication

- Processes must name each other explicitly:
  - **send (P, message)** - send a message to process P
  - **receive(Q, message)** - receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Message Passing: In-direct Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Message Passing: In-direct Communication

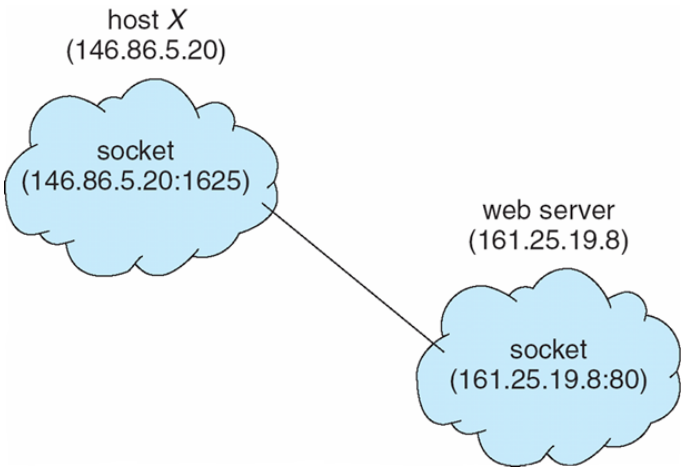
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send(A, message) send a message to mailbox A
  - receive(A, message) receive a message from mailbox A

Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** a number included at start of message packet to differentiate network services on a host
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets
- All ports below 1024 are well known, used for standard services
- Special IP address 127.0.0.1 **(loopback)** to refer to system on which process is running



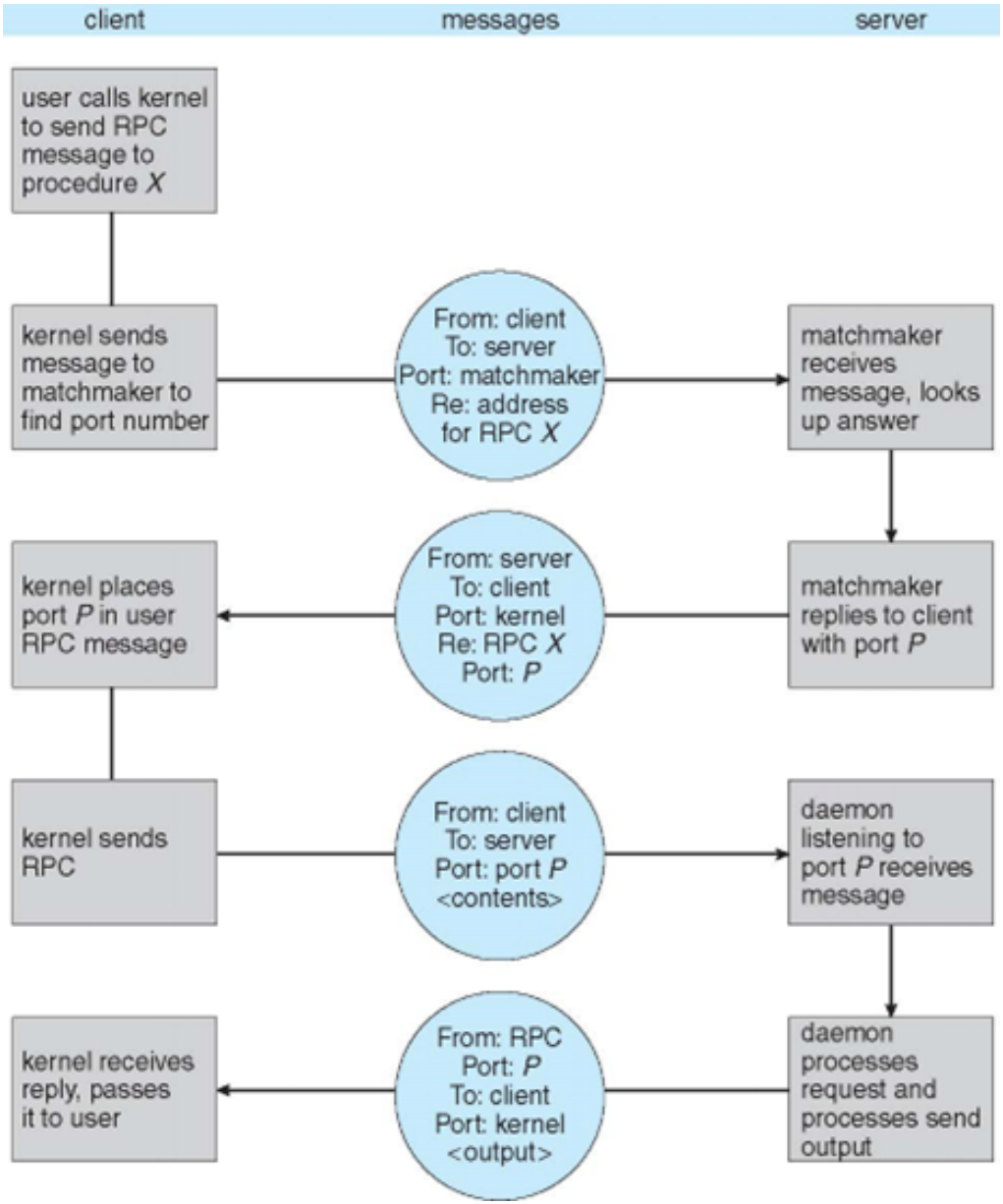
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- Stubs client-side proxy for the actual procedure on the server
- The client-side stub locates the server and marshalls the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in Microsoft Interface Definition Language (MIDL)

Remote Procedure Calls

- Data representation handled via External Data Representation (XDL) format to account for different architectures
  - Big-endian and little-endian
- Remote communication has more failure scenarios than local
  - Messages can be delivered exactly once rather than at most once
- OS typically provides a rendezvous (or matchmaker) service to connect client and server

Remote Procedure Calls

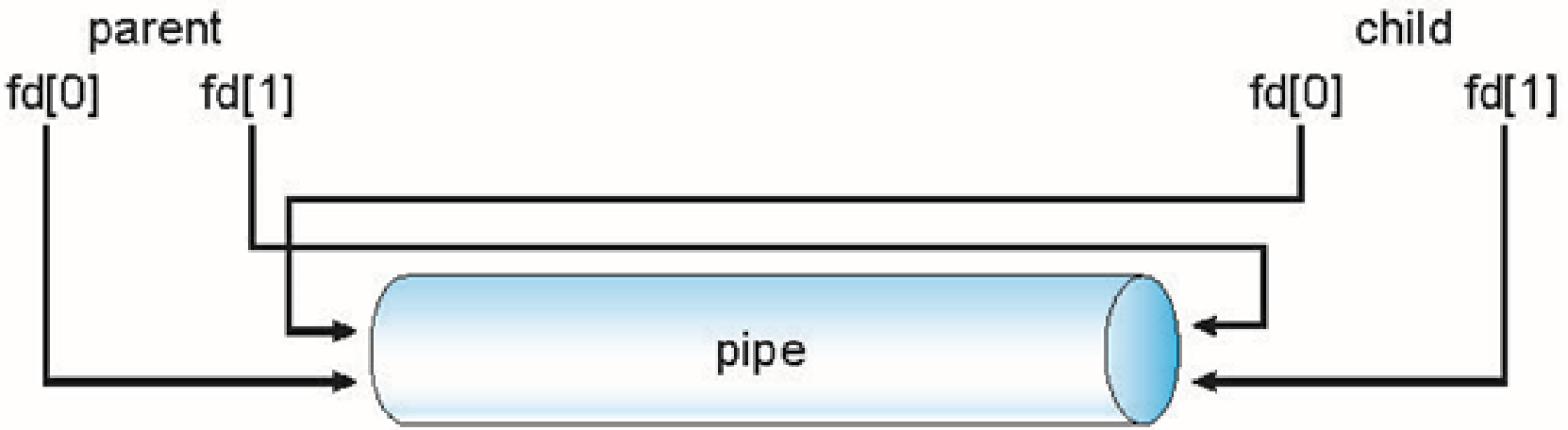


Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., parent-child) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** - cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- **Named pipes** - can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes
- Windows calls these anonymous pipes
- See Unix and Windows code samples in textbook



Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

- Message passing may be either blocking or non-blocking
- **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
- Different combinations possible
- If both send and receive are blocking, we have a **rendezvous**

- Producer-consumer becomes trivial

```
message next_produced;
while (true) {
    /* produce an item in next produced */
send(next_produced);
}
```

```
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 message are queued on a link. Sender must wait for the receiver (rendezvous).
  2. Boundary capacity – finite length of n messages sender must wait if link full.
  3. Unbounded capacity – infinite length sender will never wait.

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    $shm\_fd = shm\_open(name, OCREAT|ORDWR, 0666);$
  - Also used to open an existing segment to share it
  - Set the size of the object
    $ftruncate(shmfd, 4096);$
  - Now the process could write to the shared memory
    $sprintf(sharedmemory, "Writing\ to\ shared\ memory");$

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Critical Section Problem

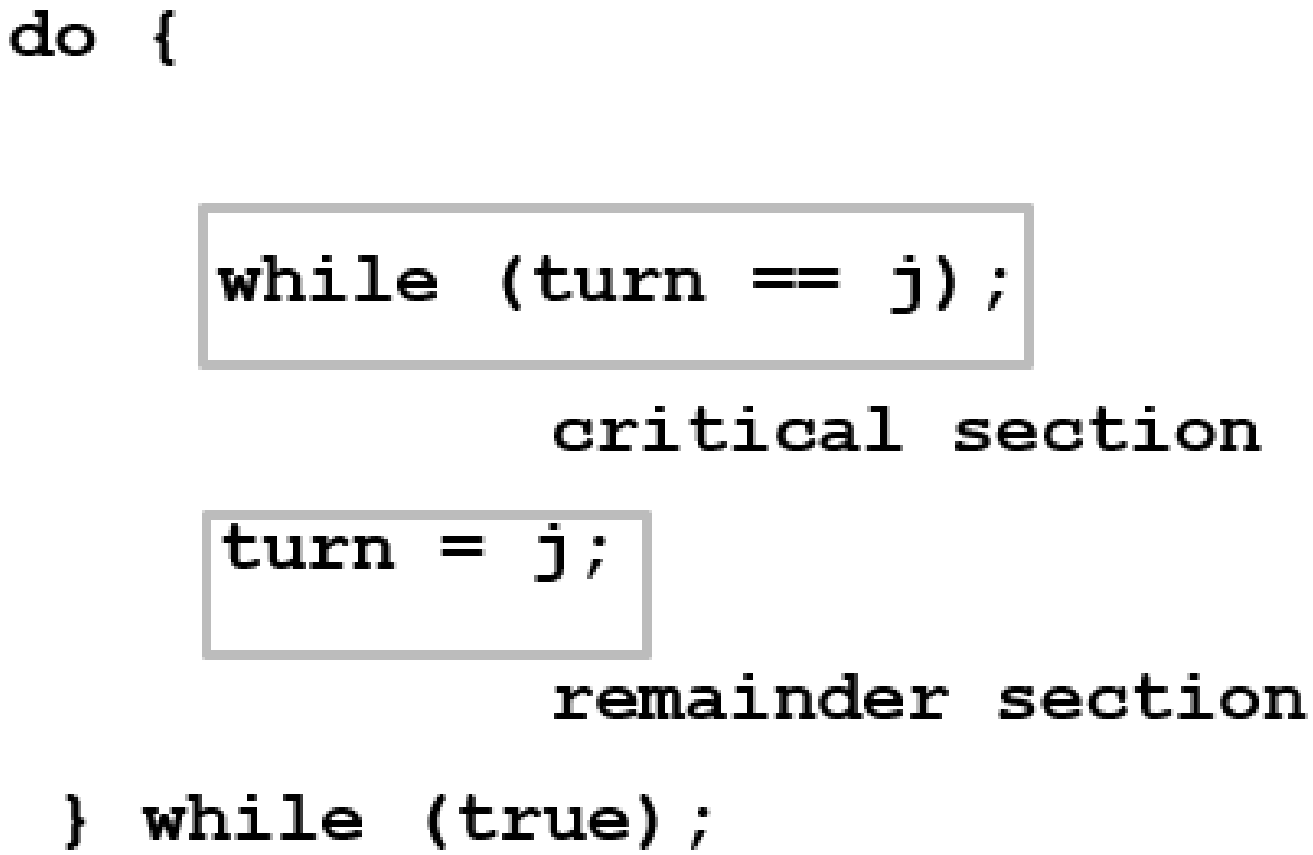
- Consider system of n processes $\{p_0, p_1, ...p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**.

Critical Section Problem

- General structure of process P_i

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Critical Section Problem: Algorithm



```
do {

        while (turn == j);

                critical section

        turn = j;

                remainder section

} while (true);
```

Critical Section Problem: Criteria

- **Mutual Exclusion** - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Note

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the **n** processes

Critical-Section Handling in OS

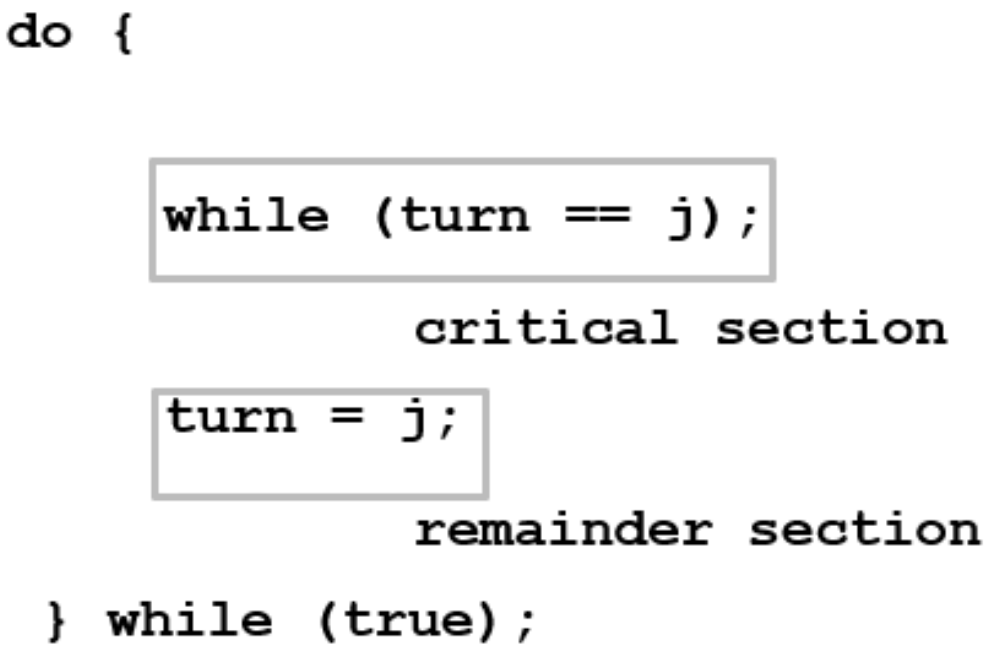
Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** - allows preemption of process when running in kernel mode
- **Non-preemptive** - runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

PETERSON'S SOLUTION

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - *int turn;*
  - *Boolean flag[2]*
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section. *flag[i]* = *true* implies that process Pi is ready!

Algorithm for Process P_i



```
do {

        while (turn == j);

                critical section

        turn = j;

                remainder section

} while (true);
```

- Provable that the three CS requirement are met:
  - Mutual exclusion is preserved
    P_i enters CS only if:
    either flag[j] = false or turn = i
  - Progress requirement is satisfied
  - Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors - could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv:
    }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

test_and_set() Solution

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
            /* critical section */
    lock = false;
            /* remainder section */
 } while (true);
```

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter "value"
3. Set the variable "value" the value of the passed parameter "new_value" but only if "value" == "expected". That is, the swap takes place only under this condition.

compare_and_swap Instruction: Solution

- Shared integer "lock" initialized to 0;
- Solution:

```
do {
      while (compare_and_swap(&lock, 0, 1) != 0)
       ; /* do nothing */
    /* critical section */
 lock = 0;
    /* remainder section */
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
- Calls to **acquire()** and **release()** must be atomic
- Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**. This lock therefore called a **spinlock**

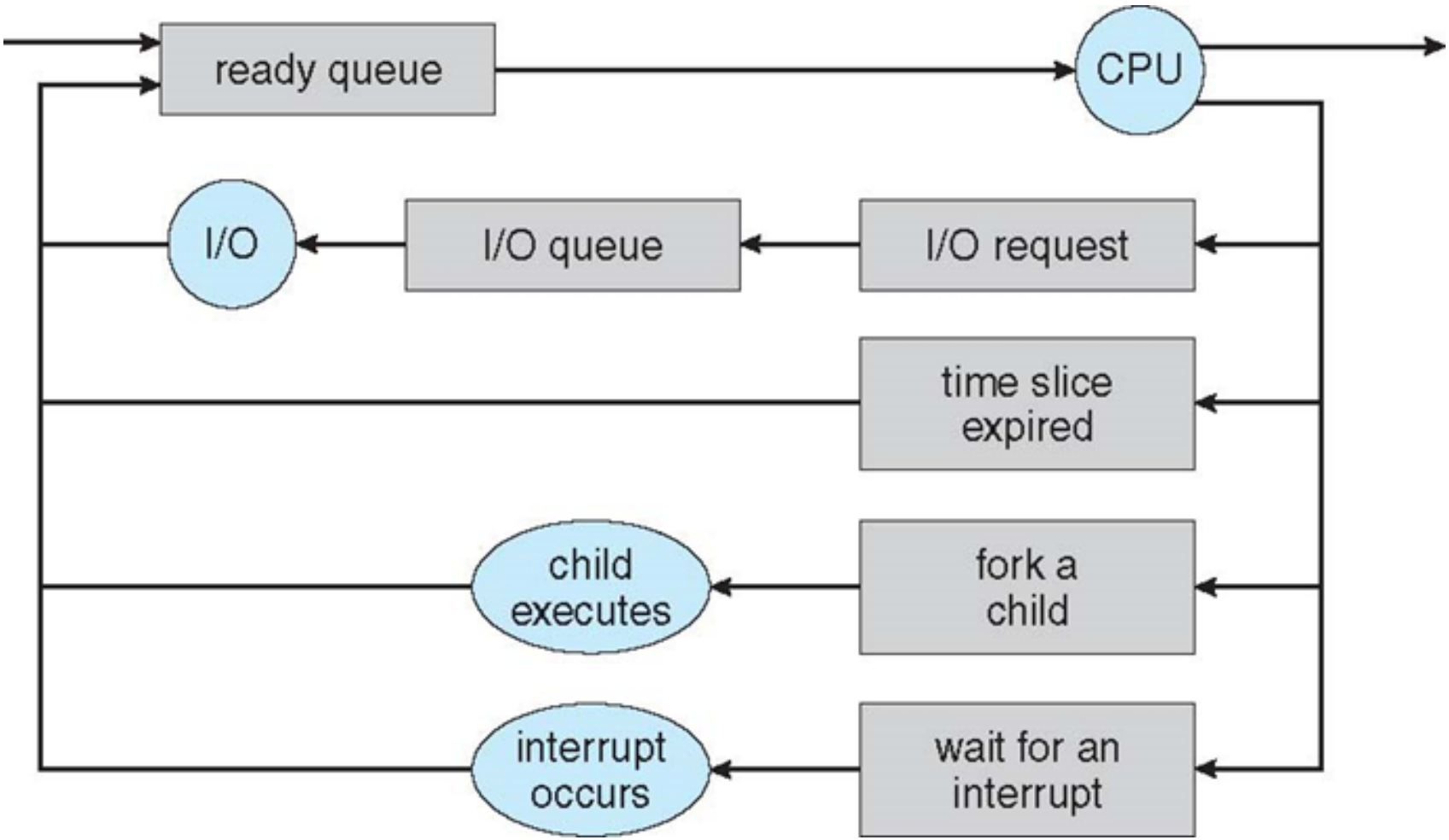
acquire() and release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}
release() {
    available = true;
}
do {
acquire lock
    critical section
release lock
  remainder section
} while (true);
```

Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing and multiprogramming.
- **Process scheduler** selects among available processes for next execution on CPU
- Several processes are kept in memory at one time
- Every time a running process has to wait, another process can take over use of the CPU
- Maintains **scheduling queues** of processes
  - **Job queue** - set of all processes in the system
  - **Ready queue** - set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** - set of processes waiting for an I/O device
  - Processes migrate among the various queues

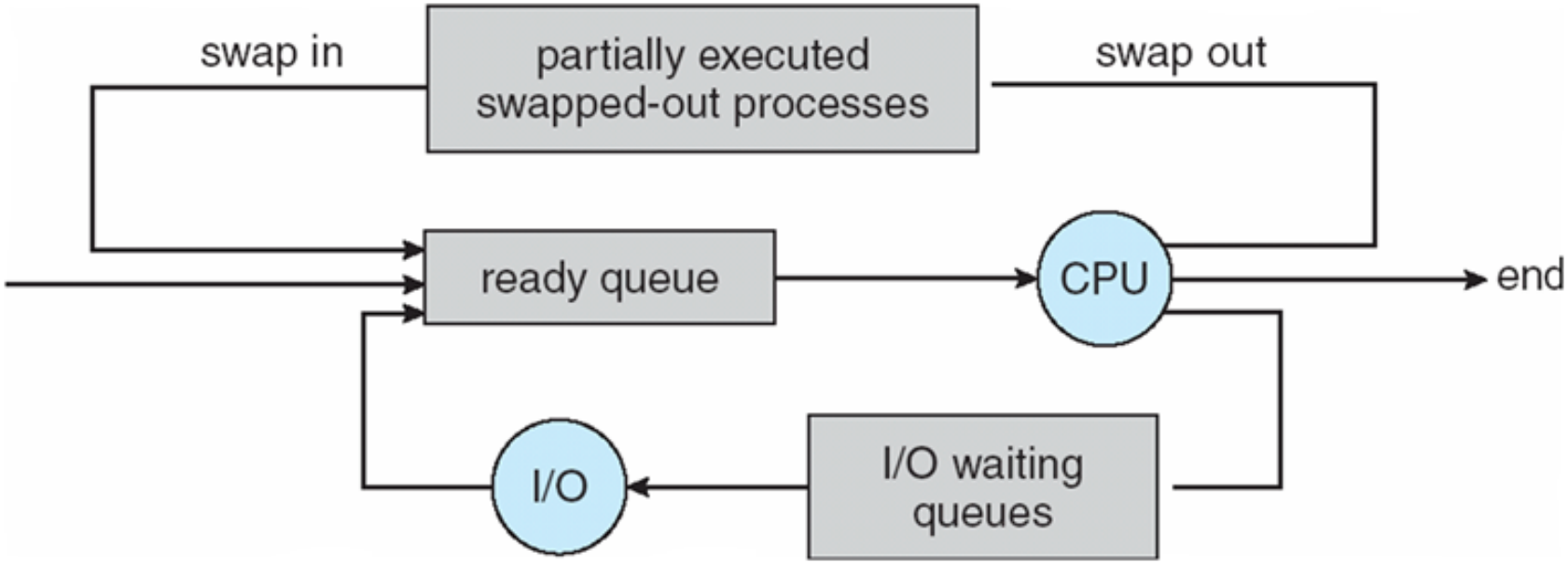
Queueing diagram represents queues, resources, flows



Schedulers

- **Short-term scheduler** (or **CPU scheduler**) - selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**) - selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\rightarrow$ (may be slow)
  - The long-term scheduler controls the degree of multiprogramming
- Processes can be described as either:
  - **I/O-bound process** - spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** - spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good process mix

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Scheduling Criteria

Criteria for comparing CPU scheduling algorithms may include the following

- **CPU utilization** - percent of time that the CPU is busy executing a process
- **Throughput** - number of processes that are completed per time unit
- **Response time** - amount of time it takes from when a request was submitted until the first response occurs (but not the time it takes to output the entire response)
- **Waiting time** - the amount of time before a process starts after first entering the ready queue (or the sum of the amount of time a process has spent waiting in the ready queue)
- **Turnaround time** - amount of time to execute a particular process from the time of submission through the time of completion

- **CPU utilization**

$$CPU\_Utilization = \frac{CPU\_Busy\_Time}{Total\_Time\_Consumed} \times 100\%$$

where,
$Total\_Time\_Consumed = CPU\_Busy\_Time + CPU\_Ideal\_Time$

- **Throughput** Number of process completed in one time quanta.

$$Throughput = \frac{Number\_of\_Process}{Total\_Time\_Consumed}$$

Scheduling Criteria

- **Arrival Time** is the point of time at which a process enters the ready queue.
- **Burst Time** is the amount of time required by a process for executing on CPU. It is also called as execution time or running time.
- **Completion Time** is the point of time at which a process completes its execution on the CPU and takes exit from the system. It is also called as exit time.
- **Waiting Time** is the amount of time spent by a process waiting in the ready queue for getting the CPU.

$$Waiting\_Time = Turn\_Around\_Time - Burst\_Time$$

Scheduling Criteria

- **Response Time** is the amount of time after which a process gets the CPU for the first time after entering the ready queue.

$$Response\_Time = CPU\_Allocation\_Time - Arrival\_Time$$

- **Turnaround time** is the total amount of time spent by a process in the system. When present in the system, a process is either waiting in the ready queue for getting the CPU or it is executing on the CPU.

$$Turn\_Around\_Time = Compilation\_Time - Arrival\_Time$$

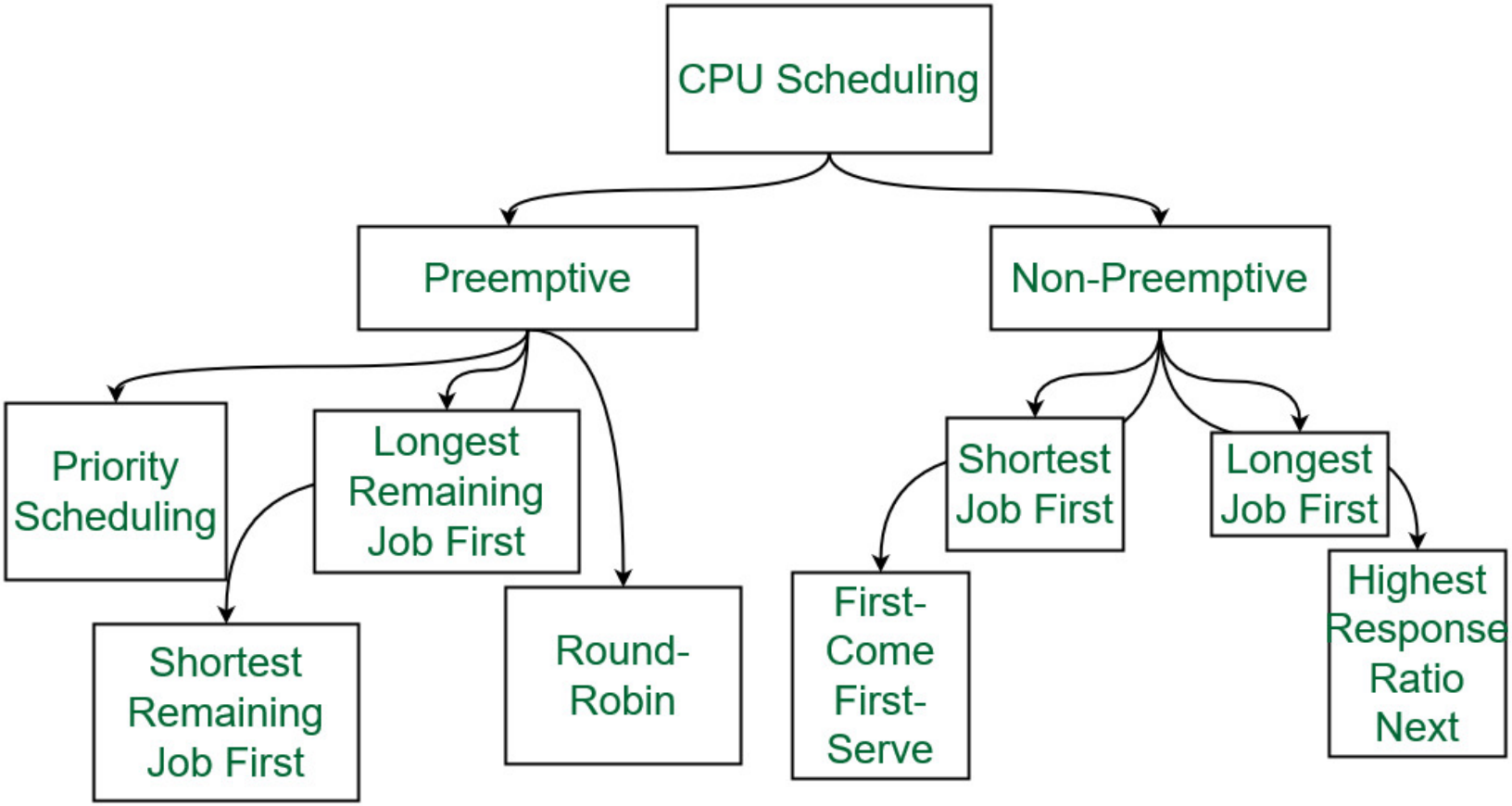
or

$$Turn\_Around\_Time = Burst\_Time + Waiting\_Time$$

Objectives of Process Scheduling Algorithm

- Utilization of CPU at maximum level. Keep CPU as busy as possible.
- Allocation of CPU should be fair.
- Throughput should be Maximum. i.e. Number of processes that complete their execution per time unit should be maximized.
- Minimum turnaround time, i.e. time taken by a process to finish execution should be the least.
- There should be a minimum waiting time and the process should not starve in the ready queue.
- Minimum response time. It means that the time when a process produces the first response should be as less as possible.

Scheduling Algorithms



Single Processor Scheduling Algorithms

1. First Come, First Served (FCFS)
2. Shortest Job First (SJF)
3. Priority
4. Round Robin (RR)

First-Come, First-Served (FCFS)

FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

- **Characteristics of FCFS:**
  - FCFS supports non-preemptive and preemptive CPU scheduling algorithms.
  - Tasks are always executed on a First-come, First-serve concept.
  - FCFS is easy to implement and use.
  - This algorithm is not much efficient in performance, and the wait time is quite high.

First-Come, First-Served (FCFS)

Advantages of FCFS:

- Easy to implement
- First come, first serve method

Disadvantages of FCFS:

- FCFS suffers from Convoy effect.
- The average waiting time is much higher than the other algorithms.
- FCFS is very simple and easy to implement and hence not much efficient.

First-Come, First-Served (FCFS)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order of P_1, P_2, P_3, the GANTT chart for the schedule is:

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

First-Come, First-Served (FCFS)

- Suppose that the processes arrive in the order: P_3, P_2, P_1

- Waiting time for $P_1 = 6$, $P_2 = 0$, $P_3 = 3$
- Average waiting time: $(6+0+3)/3 = 17$

Convoy Effect

In convoy effect,

- Consider processes with higher burst time arrived before the processes with smaller burst time.
- Then, smaller processes have to wait for a long time for longer processes to release the CPU.

First-Come, First-Serve: Question 1

Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3

If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn around time.

First-Come, First-Serve: Question 2

Consider the set of 6 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	3
P2	1	2
P3	2	1
P4	3	4
P5	4	5
P6	5	2

If the CPU scheduling policy is FCFS and there is 1 unit of overhead in scheduling the processes, find the efficiency of the algorithm.

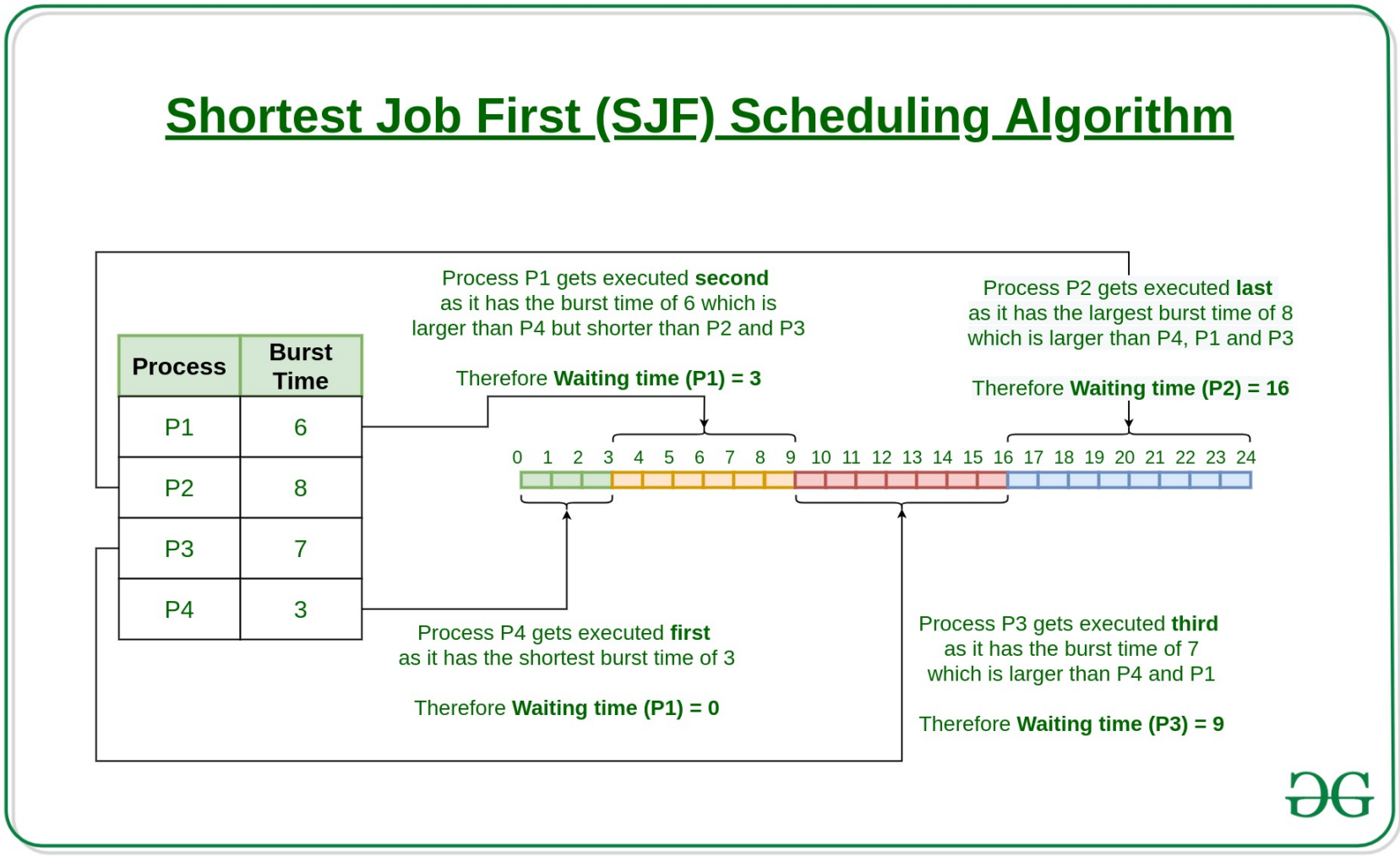
Shortest Job First

Shortest job first (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. In SJF scheduling,

- Out of all the available processes, CPU is assigned to the process having smallest burst time.
- In case of a tie, it is broken by **FCFS Scheduling**.
- Significantly reduces the average waiting time for other processes waiting to be executed.
- SJF scheduling can be used in two categories:
  - **Non-Preemptive:-** once the CPU is given to the process, it cannot be preempted until it completes its CPU burst.
  - **Pre-emptive:-** if a new process arrives with a CPU burst length less than the remaining time of the current executing process, preempt. This scheme is know as the **Shortest-Remaining-Time-First (SRTF).**

- **Advantages:-**
  - SRTF is optimal and guarantees the minimum average waiting time.
  - It provides a standard for other algorithms since no other algorithm performs better than it.

- **Disadvantages:-**
  - It can not be implemented practically since burst time of the processes can not be known in advance.
  - It leads to starvation for processes with larger burst time.
  - Priorities can not be set for the processes.
  - Processes with larger burst time have poor response time

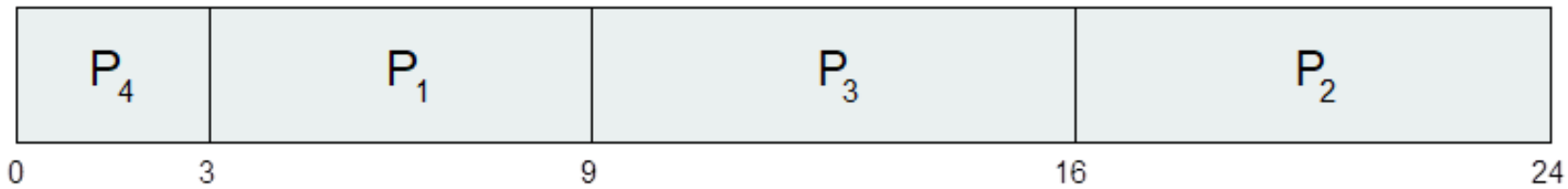


Courtesy: https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/

Shortest Job First: Example

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

SJF scheduling chart



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

Shortest Job First: Question 1

Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

If the CPU scheduling policy is SJF non-preemptive, calculate the average waiting time and average turn around time.

Shortest Job First: Question 2

Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

If the CPU scheduling policy is SJF preemptive, calculate the average waiting time and average turn around time.

Shortest Job First: Question 3

Consider the set of 6 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1

If the CPU scheduling policy is shortest remaining time first, calculate the average waiting time and average turn around time.

Shortest Job First: Question 4

Consider the set of 3 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	9
P2	1	4
P3	2	9

If the CPU scheduling policy is SRTF, calculate the average waiting time and average turn around time.

Shortest Job First: Question 5

Consider the set of 4 processes whose arrival time and burst time are given below-

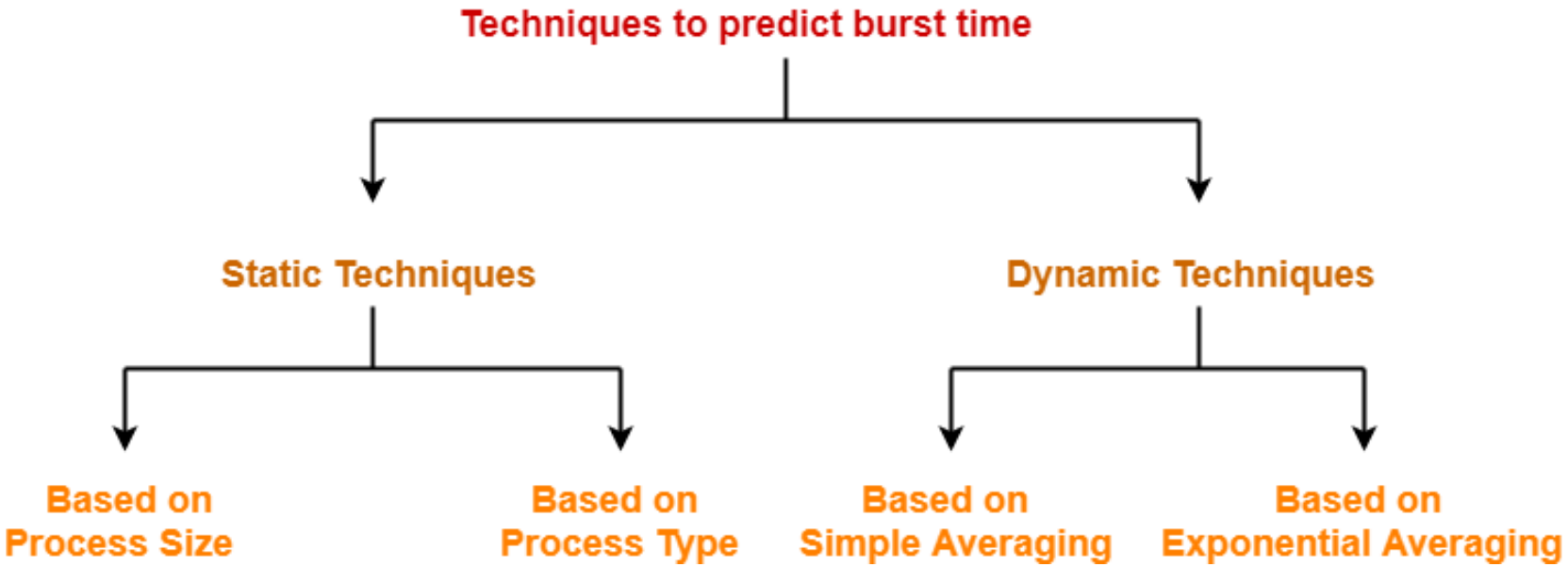
Process Id	Arrival time	Burst time
P1	0	20
P2	15	25
P3	30	10
P4	45	15

If the CPU scheduling policy is SRTF, calculate the waiting time of process P2.

Predicting Burst Time — SJF Scheduling: Need

In SJF Scheduling,

- Out of all the available processes, CPU is assigned to the process having smallest burst time.
- The main drawback of SJF Scheduling is that it can not be implemented practically.
- This is because burst time of the processes can not be known in advance.



Predicting Bust Time: Static Techniques

Based on Process Size

- This technique predicts the burst time for a process based on its size.
- Burst time of the already executed process of similar size is taken as the burst time for the process to be executed.

- The predicted burst time may not always be right.
- This is because the burst time of a process also depends on what kind of a process it is.

Based on Process Type

- This technique predicts the burst time for a process based on its type.
- The following figure shows the burst time assumed for several kinds of processes

Based on Simple Averaging

- Burst time for the process to be executed is taken as the average of all the processes that are executed till now.
- Given n processes $P_1, P_2,...,P_n$ and burst time of each process P_i as t_i, then predicted burst time for process P_{n+1} is given as:

$$T_{n+1} = \frac{1}{n}\sum_{i=1}^{n} t_i$$

Based on Exponential Averaging

- Given n processes $P_1, P_2, ..., P_n$ and burst time of each process P_i as t_i. Then, predicted burst time for process P_{n+1} is given as

$$T_{n+1} = \alpha \times t_n + (1 - \alpha) \times T_n$$

where,

- α is called smoothening factor ($0 \leq \alpha \leq 1$)
- t_n = actual burst time of process P_n
- T_n = Predicted burst time for process P_n

Predicting Burst Time: Example

Problem:- Calculate the predicted burst time using exponential averaging for the fifth process if the predicted burst time for the first process is 10 units and actual burst time of the first four processes is 4, 8, 6 and 7 units respectively. Given $\alpha = 0.5$.

Solution:-

Predicted burst time for 1st process $= 10$ units

Actual burst time of the first four processes $= 4, 8, 6, 7, \alpha = 0.5$

Predicted Burst Time for 2^{nd} Process

Predicted burst time for 2^{nd} process $= \alpha \times$ Actual burst time of 1^{st} process $+ (1 - \alpha) \times$ Predicted burst time for 1^{st} process

Predicted burst time for 2^{nd} process $= 0.5 \times 4 + 0.5 \times 10$

Predicted burst time for 2^{nd} process $= 4 + 3.5$

Predicted burst time for 2^{nd} process $= 7.5$ Units

Priority Scheduling

- A priority number (integer) is associated with each process.
- The CPU is allocated to the process with the highest priority (smallest integer $\rightarrow$ highest priority)
  - **Preemptive:** approach will preempt the CPU if the priority of the newly-arrived process is higher than the priority of the currently running process.
  - **Nonpreemptive:** approach will simply put the new process (with the highest priority) at the head of the ready queue.
- **SJF** is priority scheduling where priority is the inverse of predicted next CPU burst time

Problem with Priority Scheduling

- Problem = **Starvation** - low priority processes may never execute
- Solution = **Aging** - as time progresses increase the priority of the process

Priority Scheduling

Characteristics of Priority Scheduling:

- Schedules tasks based on priority.
- When the higher priority work arrives while a task with less priority is executed, the higher priority work takes the place of the less priority one and
- The latter is suspended until the execution is complete.
- Lower is the number assigned, higher is the priority level of a process.

Advantages:

- It considers the priority of the processes and allows the important processes to run first.
- Priority scheduling in preemptive mode is best suited for real time operating system.

Disadvantages:

- Processes with lesser priority may starve for CPU.
- There is no idea of response time and waiting time.

Priority Scheduling: Example

Process	Burst Time	Priority
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 msec

Priority Scheduling: Question

Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

If the CPU scheduling policy is priority preemptive, calculate the average waiting time and average turn around time. (Higher number represents higher priority)

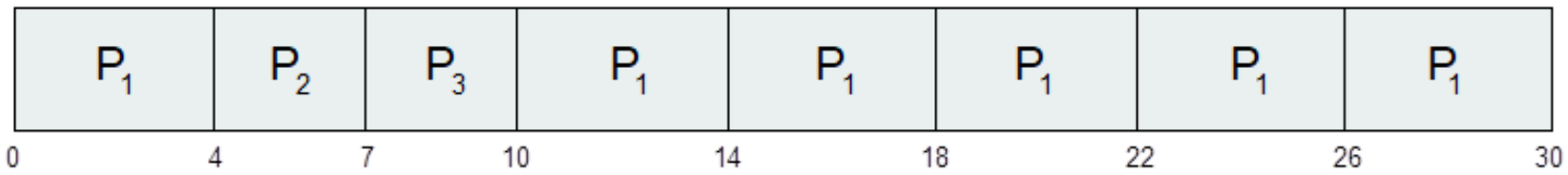
Round Robin Scheduling

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this 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.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\rightarrow$ FIFO
  - q small $\rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Round Robin Scheduling: Example

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:

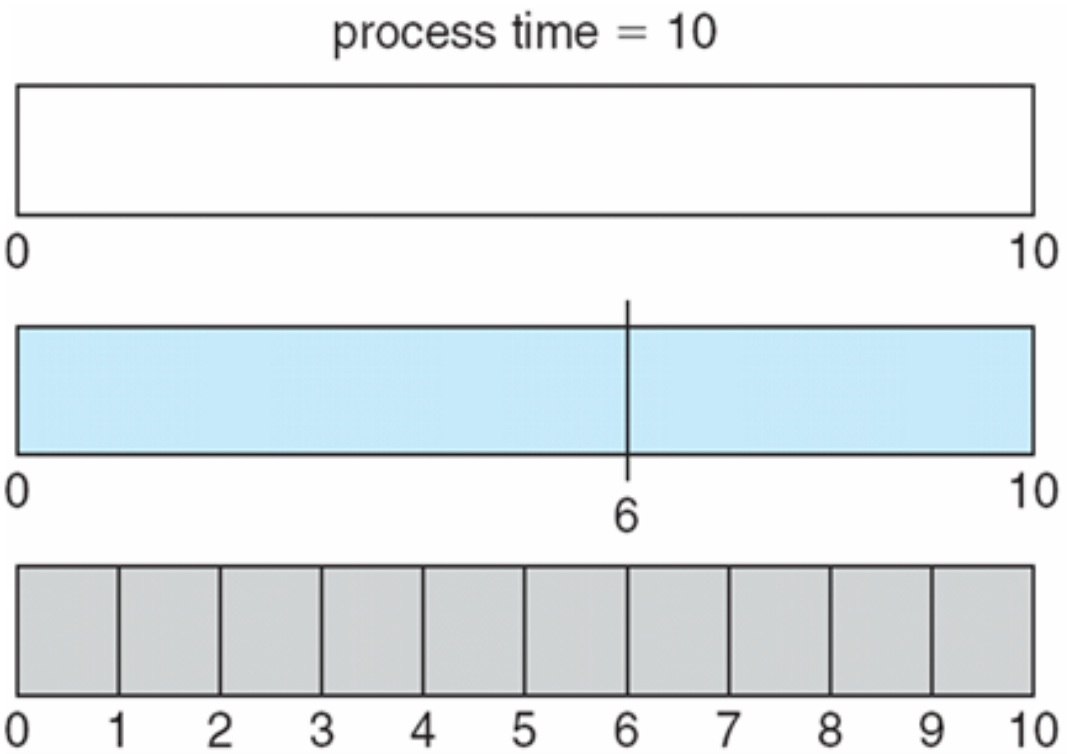


Typically, higher average turnaround than SJF, but better ***response***

q should be large compared to context switch time

q usually 10ms to 100ms, context switch < 10 usec

Context Switching Overhead

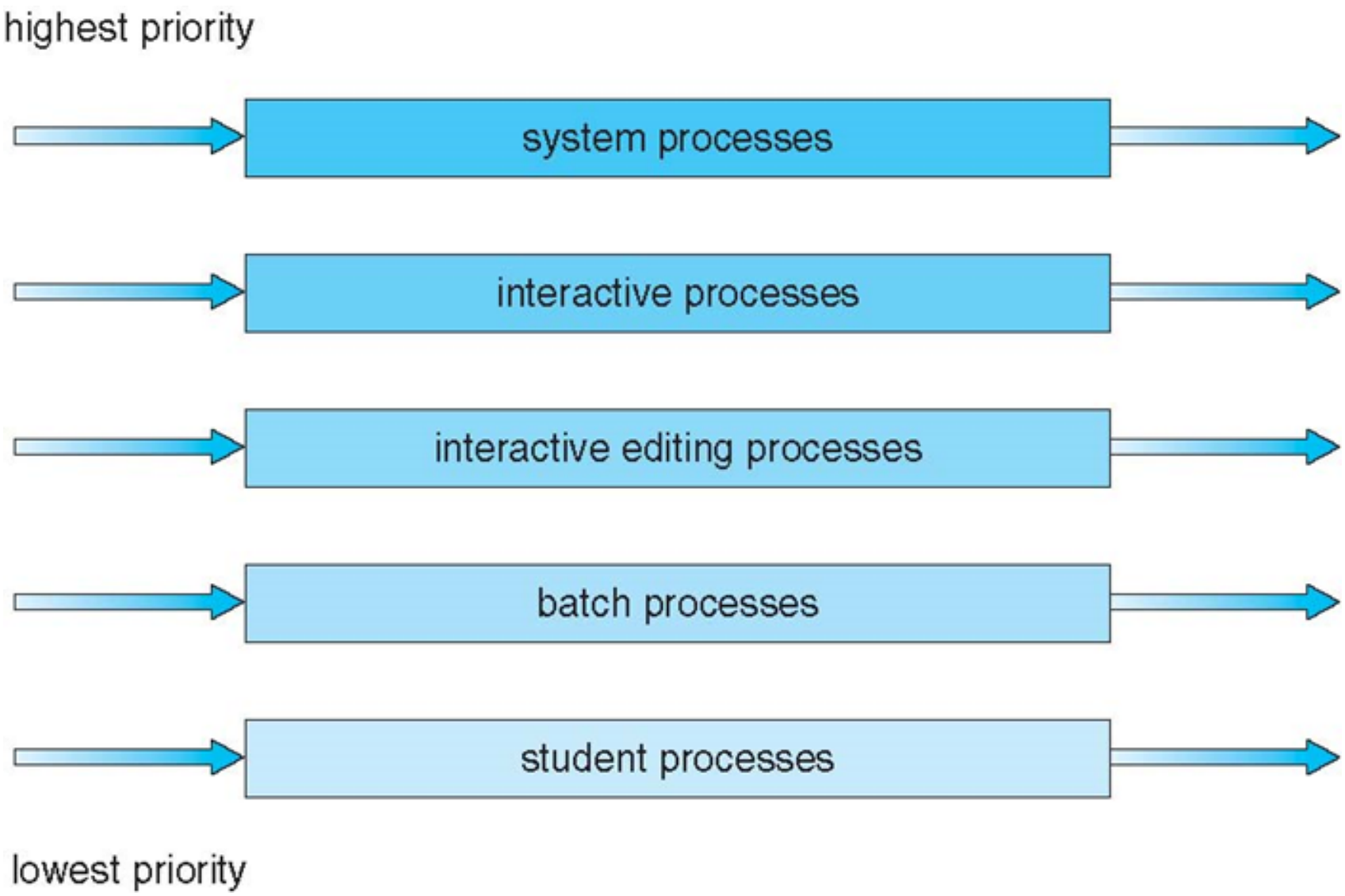


quantum	context switches
12	0
6	1
1	9

Multilevel Queue

- Multi-level queue scheduling is used when processes can be classified into groups
- Ready queue is partitioned into separate queues, eg:
  - **foreground (interactive)**
  - **background (batch)**
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground $\rightarrow$ RR
  - background $\rightarrow$ FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice - each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling

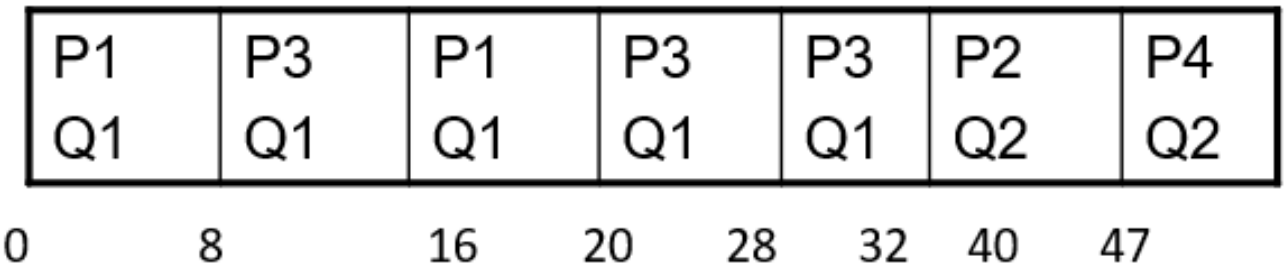


Multilevel Queue Scheduling: Example

Q. Assume there are 2 queues:- Q1(using RR scheduling with quantum =8) for foreground processes and Q2(using FCFS scheduling) for background processes. Consider following processes arriving in the system

Process	Burst time	Type
P1	12	FG
P2	8	BG
P3	20	FG
P4	7	BG

Calculate average waiting time assuming that processes in Q1 will be executed first.(Fixed priority scheduling)



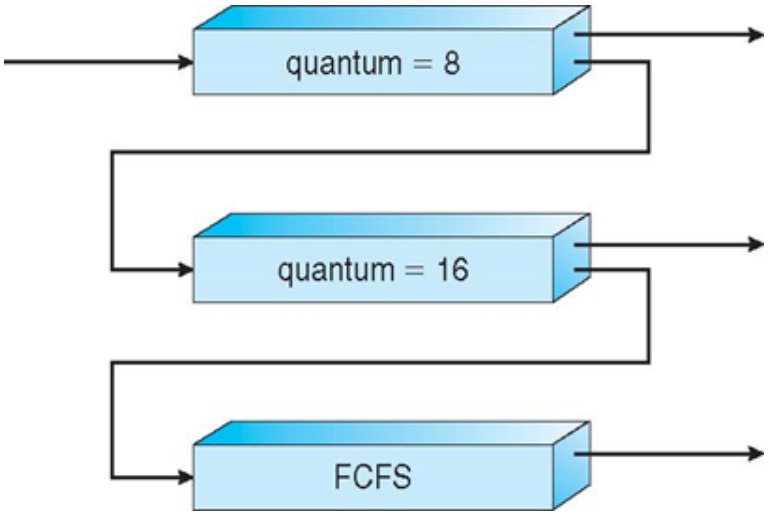
- Waiting time of P1=(20-12) =8
- Waiting time of P2 = (40-8)=32
- Waiting time of P4 =(47-7)=40
- Waiting time of P3 = (32-20)= 12
- Average waiting time = (8 + 32 + 40 + 12)/4=23

Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Multilevel Feedback Queue: Example

- Three queues:
  - Q0 - RR with time quantum 8 milliseconds
  - Q1 - RR time quantum 16 milliseconds
  - Q2 - FCFS
- Scheduling
  - A new job enters queue Q0 which is served FCFS
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue Q1
  - At Q1 job is again served FCFS and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue Q2



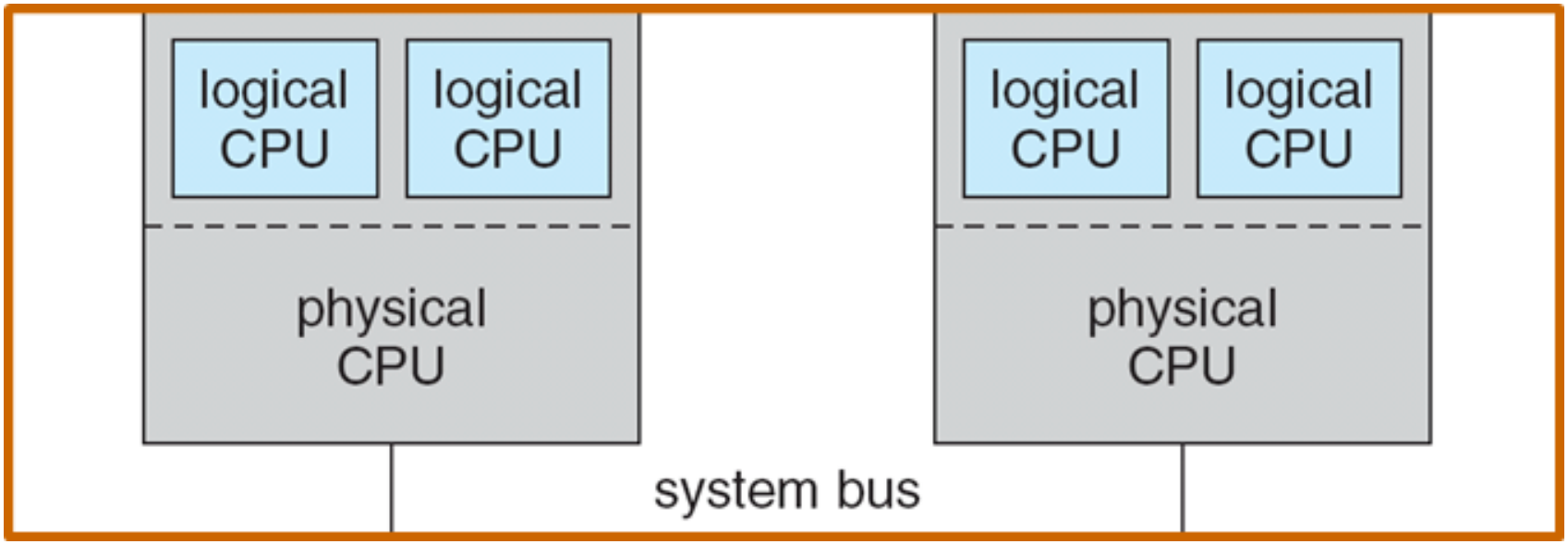
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** - only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** - each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- **Processor affinity** - process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**

Multiple-Processor Scheduling

- Symmetric multiprocessing systems allow several threads to run concurrently by providing multiple physical processors
- An alternative approach is to provide multiple logical rather than physical processors
- Such a strategy is known as symmetric multithreading (SMT)
  - This is also known as hyperthreading technology
- The idea behind SMT is to create multiple logical processors on the same physical processor
  - This presents a view of several logical processors to the operating system, even on a system with a single physical processor
  - Each logical processor has its own architecture state, which includes general-purpose and machine-state registers
  - Each logical processor is responsible for its own interrupt handling
  - However, each logical processor shares the resources of its physical processor, such as cache memory and buses.
- SMT is a feature provided in the hardware, not the software

A typical SMT architecture



Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore is simply an integer variable that is shared between threads.
- This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.
- Semaphores are of two types:
  - **Binary Semaphore:-** This is also known as mutex lock. It can have only two values 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
  - **Counting Semaphore:-** It's value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Counting Semaphore

- The value of counting semaphore may be positive or negative.
  - Positive value indicates the number of processes that can be present in the critical section at the same time.
  - Negative value indicates the number of processes that are blocked in the waiting list.
- **Case 1: Counting Semaphore $\geq$ 0** If the resulting value of counting semaphore is greater than or equal to 0, process is allowed to enter the critical section.
- **Case 2: Counting Semaphore Value $<$ 0** If the resulting value of counting semaphore is less than 0, process is not allowed to enter the critical section.

Counting Semaphore

- The signal operation is executed when a process takes exit from the critical section.
- Signal operation increments the value of counting semaphore by 1. Then, following two cases are possible:
- **Case 1: Counting Semaphore $\leq$ 0** If the resulting value of counting semaphore is less than or equal to 0, a process is chosen from the waiting list and wake up to execute.
- **Case 2: Counting Semaphore Value $>$ 0** If the resulting value of counting semaphore is greater than 0, no action is taken.

Binary Semaphore

- A binary semaphore has two components:
  - An integer value which can be either 0 or 1
  - An associated waiting list (usually a queue)
- The wait operation is executed when a process tries to enter the critical section. Then, there are two cases possible
  - **Case 1: Binary Semaphore Value = 1** If the value of binary semaphore is 1, The value of binary semaphore is set to 0. The process is allowed to enter the critical section.
  - **Case 2: Binary Semaphore Value = 0** If the value of binary semaphore is 0, The process is blocked and not allowed to enter the critical section. The process is put to sleep in the waiting list.

Binary Semaphore

- If the value of binary semaphore is 0,
  - The process is blocked and not allowed to enter the critical section.
  - The process is put to sleep in the waiting list.
- The signal operation is executed when a process takes exit from the critical section. Then, there are two cases possible:
  - **Case 1: Waiting List is Empty** If the waiting list is empty, the value of binary semaphore is set to 1.
  - **Case 2: Waiting List is Not Empty** If the waiting list is not empty, a process is chosen from the waiting list and wake up to execute.

Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time.
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section.
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- **Note that applications may spend lots of time in critical sections and therefore this is not a good solution**

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** - place the process invoking the operation on the appropriate waiting queue
  - **wakeup** - remove one of processes in the waiting queue and place it in the ready queue

```
typedef struct{
        int value;
        struct process *list;
} semaphore;
```

Semaphore Implementation with no Busy waiting

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Deadlock and Starvation

- **Deadlock:-** two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let **S** and **Q** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **Starvation - indefinite blocking** A process may never be removed from the semaphore queue in which it is suspended.
- **Priority Inversion** Scheduling problem when lower-priority process holds a lock needed by higher-priority process.

Classical Problems of Synchronization

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- n buffers, each can hold one item
- Semaphore *mutex* initialized to the value 1
- Semaphore *full* initialized to the value 0
- Semaphore *empty* initialized to the value n

Bounded-Buffer Problem

The structure of the producer process

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
} while (true);
```

Bounded-Buffer Problem

The structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
} while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers - only read the data set; they do not perform any updates
  - Writers - can both read and write
- Problem - allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore *rw_mutex* initialized to 1
  - Semaphore *mutex* initialized to 1
  - Integer *read_count* initialized to 0

Readers-Writers Problem

The structure of a writer process

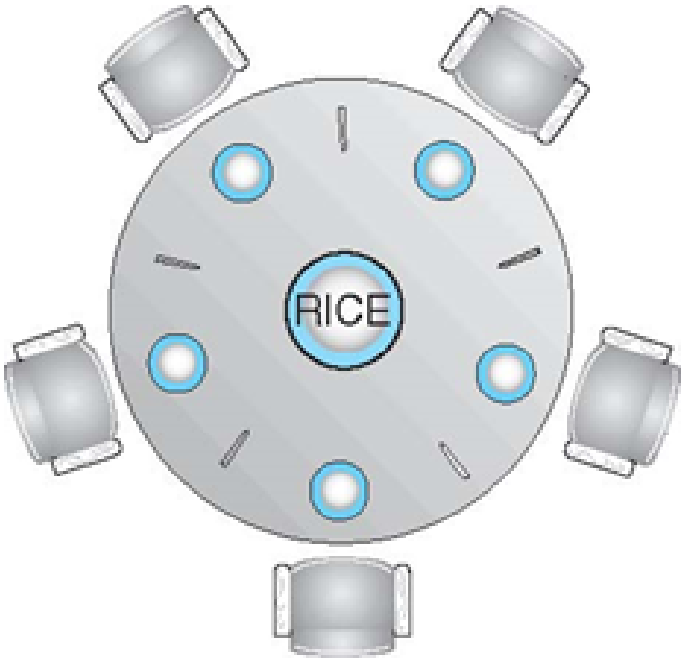
```
do {
        wait(rw_mutex);
           ...
        /* writing is performed */
           ...
     signal(rw_mutex);
} while (true);
```

Readers-Writers Problem

The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
        ...
        /* reading is performed */
        ...
    wait(mutex);
        read count--;
        if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
  - Bowl of rice (data set)
  - Semaphore **chopstick [5]** initialized to 1

Dining-Philosophers Problem Algorithm

The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                // eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

                // think

} while (TRUE);
```

What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution – an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal(mutex) .... wait(mutex)
  - wait(mutex) .... wait(mutex)
  - Omitting of wait (mutex) or signal(mutex) (or both)
- Deadlock and starvation are possible.

Monitors

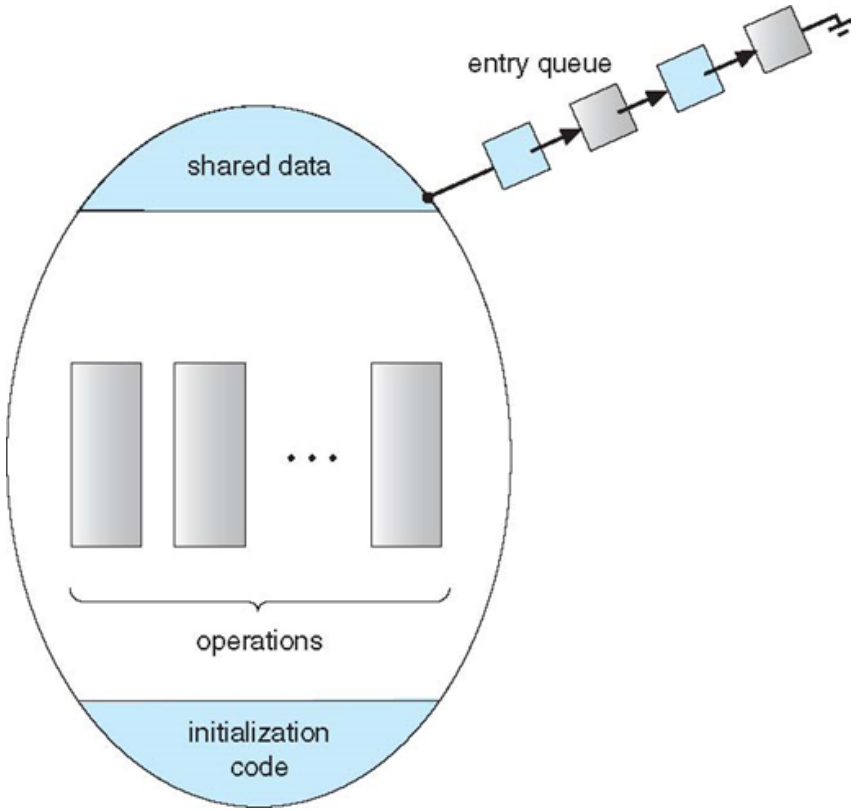
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
}
```

Schematic view of a Monitor



Condition Variables

- If process P invokes **x.signal()**, and process Q is suspended in **x.wait()**, what should happen next ?
- Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait**
  - **Signal and continue**
  - Both have merits and demerits - language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
  - Implemented in other languages including Mesa, C#, Java

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self[i].wait;
}

   void putdown (int i) {
          state[i] = THINKING;
                    // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
}
```

Monitor Solution to Dining Philosophers

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                 state[i] = EATING ;
             self[i].signal () ;
            }
     }

         initialization_code() {
            for (int i = 0; i < 5; i++)
            state[i] = THINKING;
          }
}
```

Monitor Solution to Dining Philosophers

- Each philosopher *i* invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);

EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially = 1)
semaphore next;   // (initially = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    ...
    body of F;
    ...
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation - Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation x.wait can be implemented as:

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

Monitor Implementation - Condition Variables

- The operation `x.signal` can be implemented as:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Resuming Processes within a Monitor

- If several processes queued on condition x, and **x.signal()** executed, which should be resumed ?
- FCFS frequently not adequate
- **conditional-wait** construct of the form **x.wait(c)**
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);
    ...
  access the resource;
    ...

R.release;
```

- Where R is an instance of type **Resource Allocator**

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
            if (busy)
                x.wait(time);
            busy = TRUE;
    }
    void release() {
            busy = FALSE;
            x.signal();
    }
initialization code() {
     busy = FALSE;
    }
```

Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - Events
  - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled** state (thread will block)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Thank You!!!