MANIPAL UNIVERSITY
JAIPUR

Operating System – AI2202 (AY : 2023-24)

Upendra Singh
Assistant Professor
Department of AIML, School of Computer Science and Engineering
Manipal University Jaipur.

Module 3 : Process Management

Creating, scheduling, and terminating processes, as well as managing resources and handling communication between processes.

Outlines

- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-process Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore inter-process communication using shared memory and message passing To describe communication in client-server systems

Part 1 : Process and Threads

Process and thread concepts.

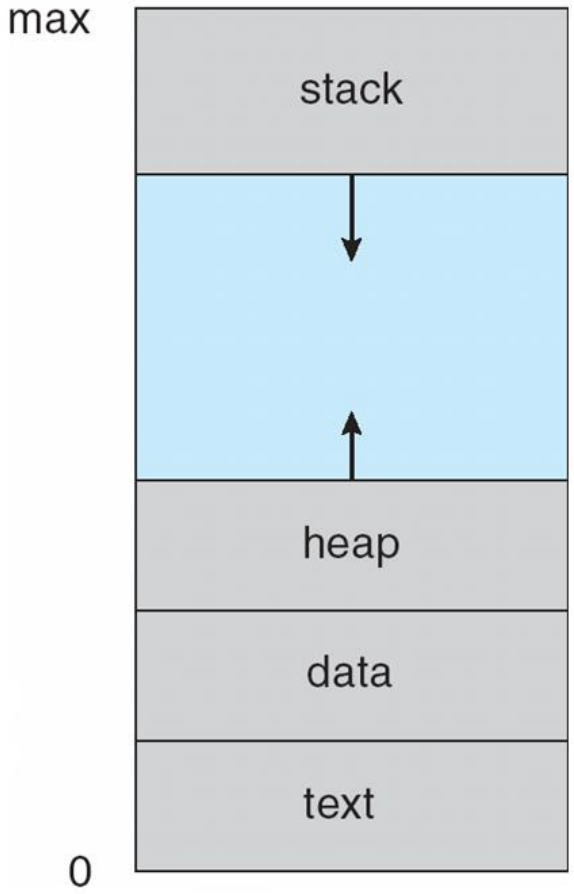
Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- **Process** – a program in execution; process execution must progress in a sequential fashion
- Multiple parts
  - The program code, also called the **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

Process Concept (Cont.)

- Program is a ***passive*** entity stored on disk (**executable file**), the process is an ***active entity***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can have several processes
  - Consider multiple users executing the same program

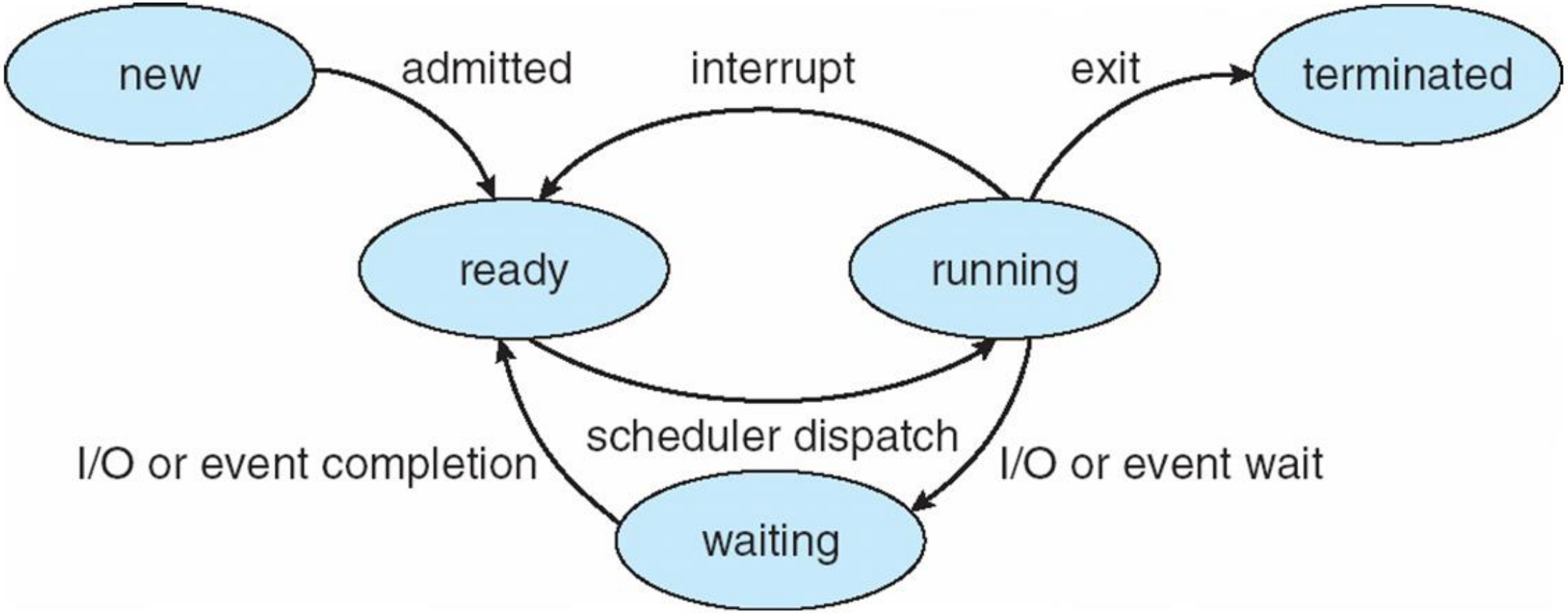
Process in Memory



Process State

- As a process executes, it changes **state**
  - **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

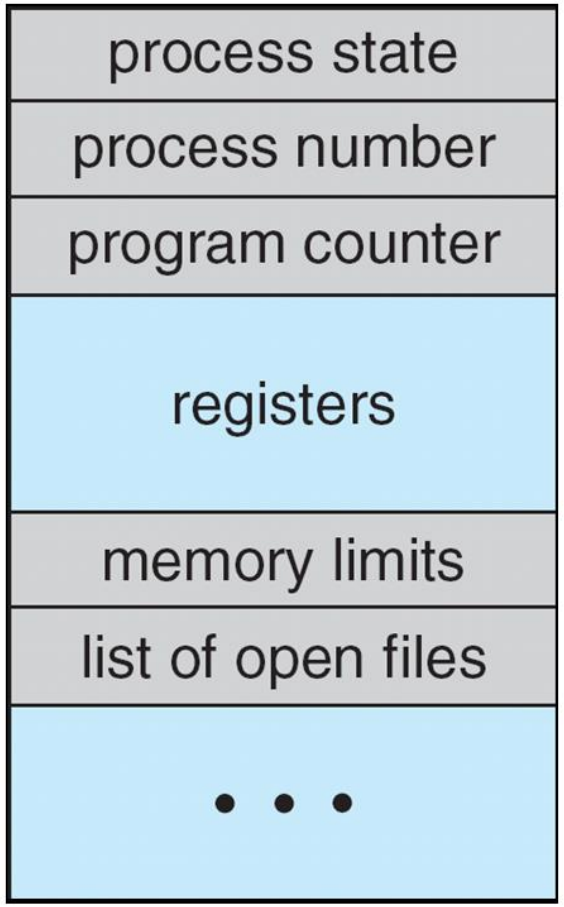
Diagram of Process State



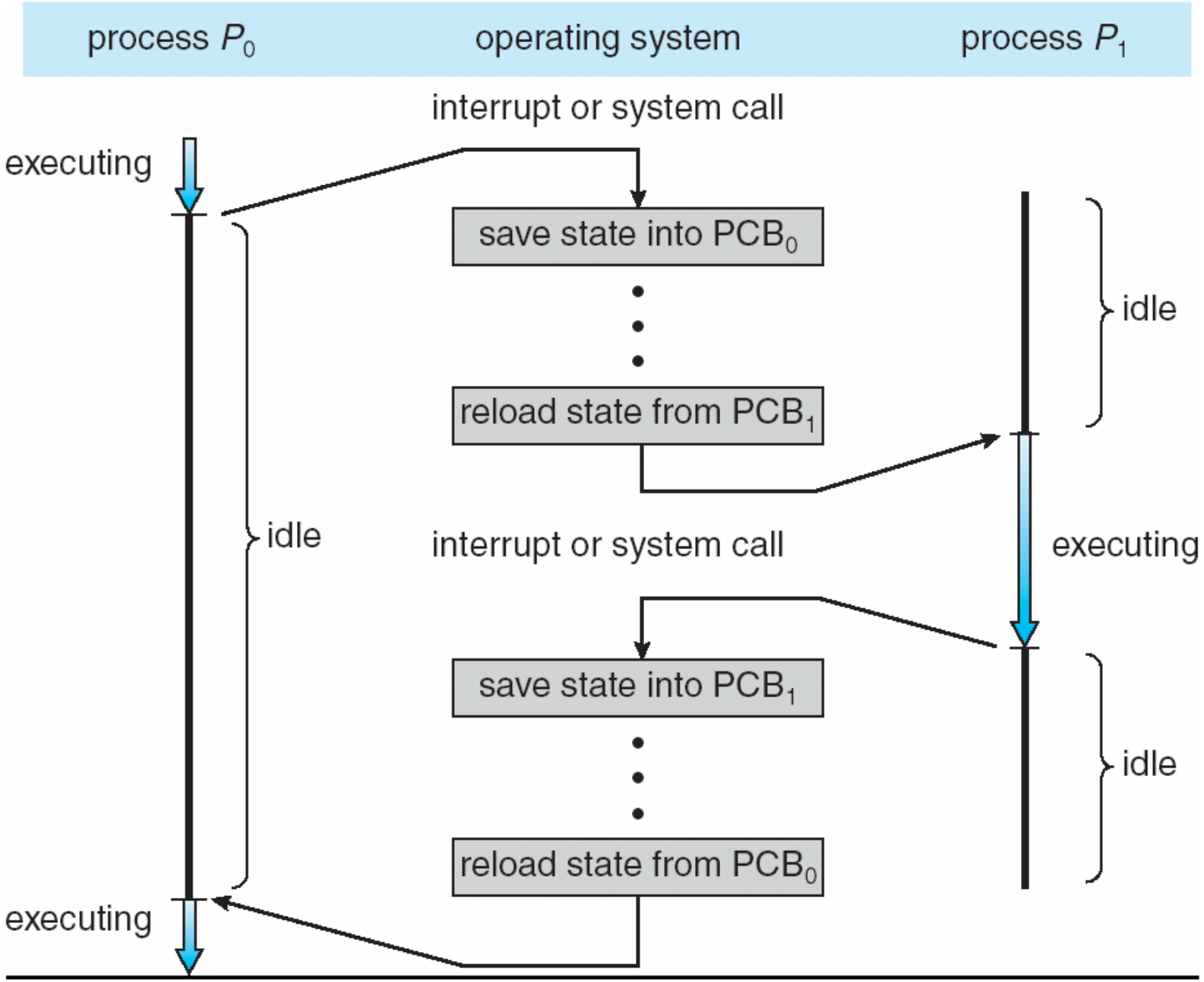
Process Control Block (PCB)

Information associated with each process (also called **task control block**)

- **Process state** – running, waiting, etc
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information-** priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to a process, list of open files



CPU Switch From Process to Process



Context Switch

- When the CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information
*/
struct task_struct *parent; /* this process's
parent */
struct list_head children; /* this process's
children */
struct files_struct *files; /* list of open files
*/
struct mm_struct *mm; /* address space of this
process *7
```

Threads

- So far, the process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB

Threads

- A thread refers to a single sequential flow of activities being executed in a process; it is also known as the thread of execution or the thread of control. A process can have several threads.

- A distinct program counter, a stack of activation records as well as control blocks are used by each thread of the same process.

- Thread is frequently described as a light technique.

Benefits of Threads

- **Enhanced throughput of the system**: When the process is split into many threads, and each thread is treated as a job, the number of jobs done in the unit time increases. That is why the throughput of the system also increases.
- **Effective Utilization of Multiprocessor system**: When you have more than one thread in one process, you can schedule more than one thread in more than one processor.
- **Faster context switch**: The context switching period between threads is less than the process of context switching. The process context switch means more overhead for the CPU.

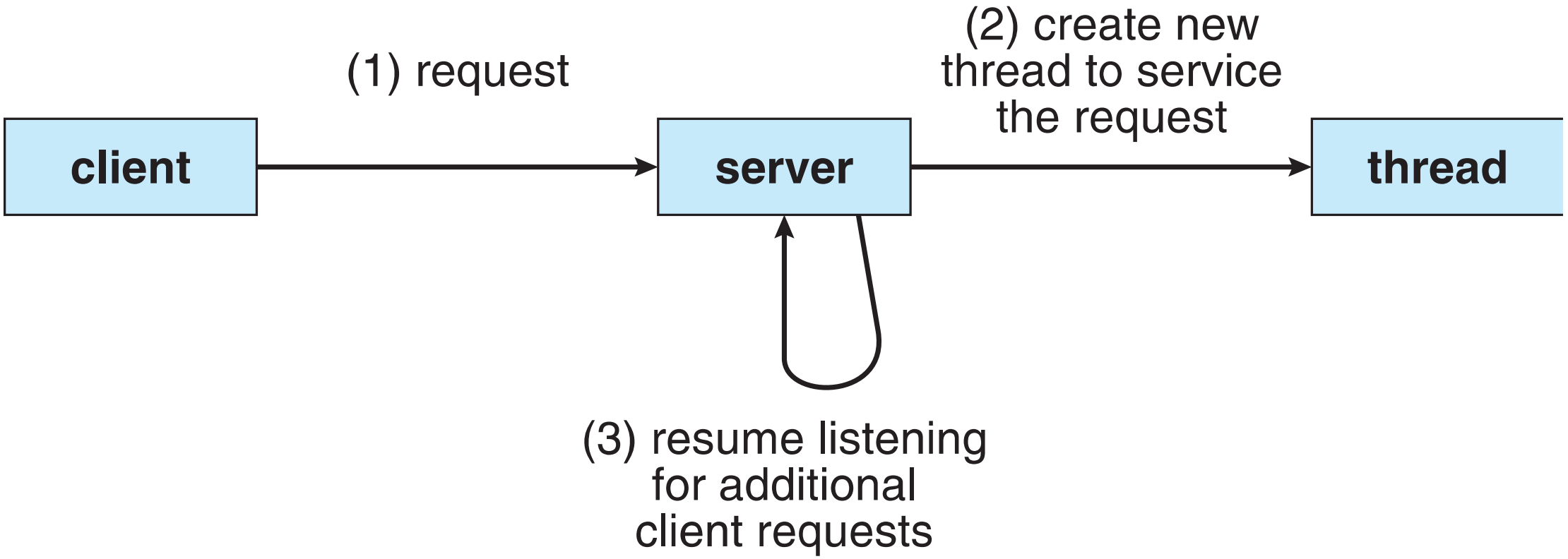
Benefits of Threads

- **Responsiveness**: When the process is split into several threads, and when a thread completes its execution, that process can be responded to as soon as possible.
- **Communication**: Multiple-thread communication is simple because the threads share the same address space, while in process, we adopt just a few exclusive communication strategies for communication between two processes.
- **Resource sharing**: Resources can be shared between all threads within a process, such as code, data, and files. Note: The stack and register cannot be shared between threads. There is a stack and register for each thread.

Benefits of Threads vs Processes

- **Less time** to **create** a new thread than a process, because the newly created thread uses the current process address space.
- Less time to **terminate** a thread than a process.
- Less time to **switch** between two threads within the same process, partly because the newly created thread uses the current process address space.
- Less **communication overheads** -- communicating between the threads of one process is simple because the threads share everything: address space, in particular. Data produced by one thread is immediately available to all the other threads.

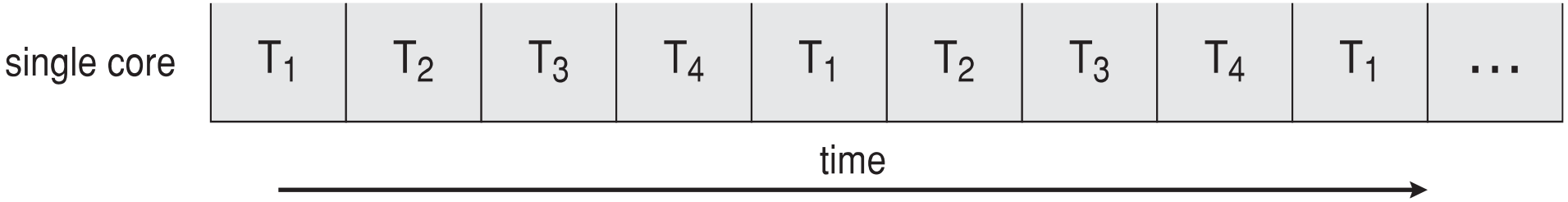
Multithreaded Server Architecture



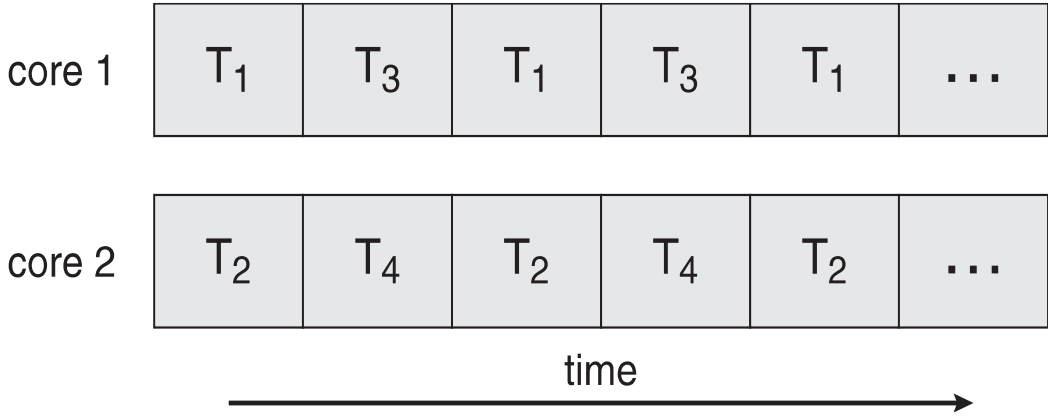
Modern Intel CPUs frequently support two threads per core, while the Oracle T4 CPU supports eight.

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



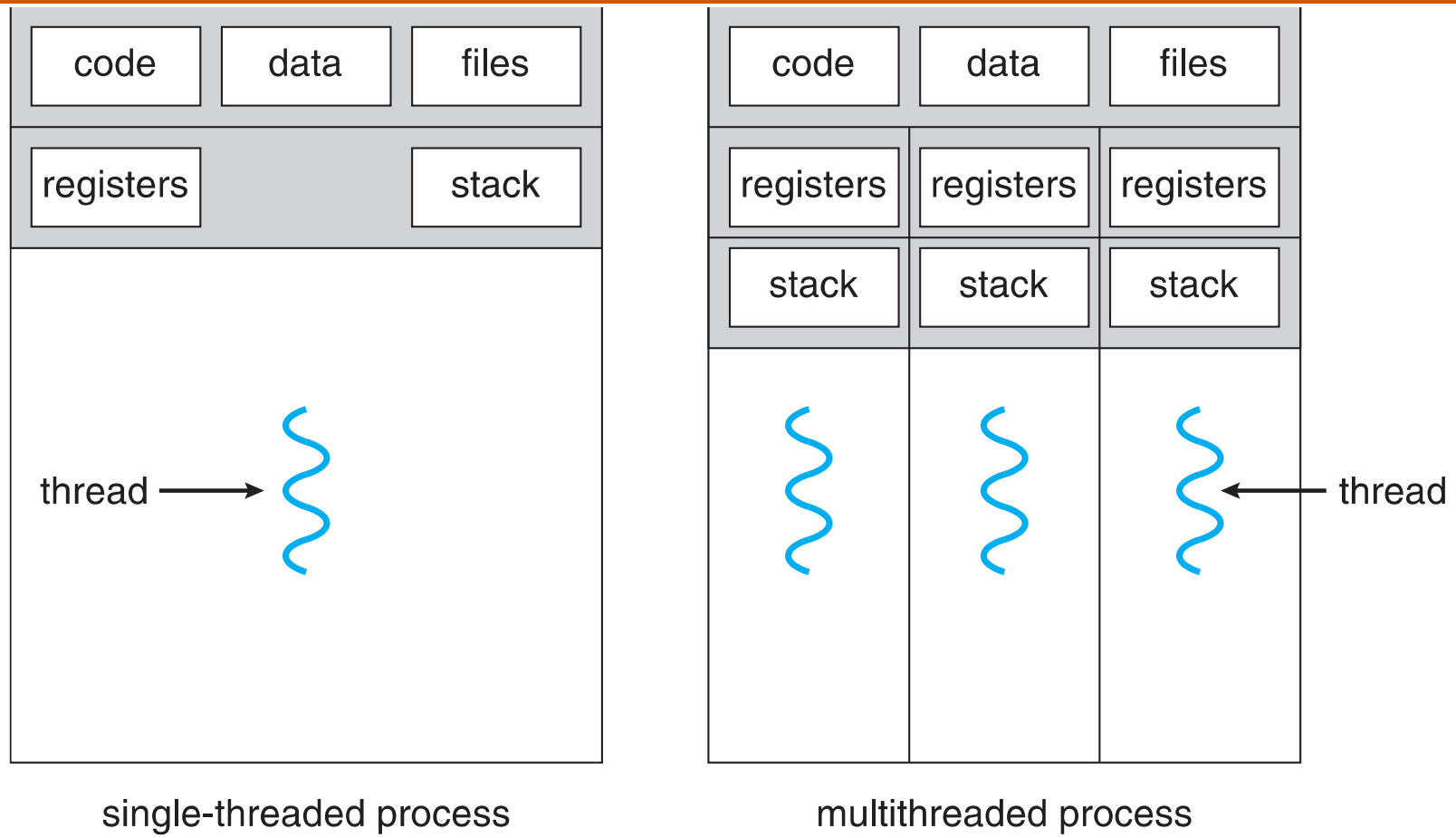
- **Parallelism on a multi-core system:**



Multicore Programming Challenges

- **Identifying tasks**. This involves examining applications to find areas that can be divided into separate, concurrent tasks.
- **Balance**. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.
- **Data splitting**. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
- **Data dependency**. The data accessed by the tasks must be examined for dependencies between two or more tasks.
- **Testing and debugging**. When a program is running in parallel on multiple cores, many different execution paths are possible.

Single and Multithreaded Processes



Types of Threads

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

User-level thread

- The operating system does not recognize the user-level thread. User threads can be easily implemented, and the user implements it. If a user performs a user-level thread-blocking operation, the whole process is blocked.
- It is faster and more efficient.
- Context switch time is shorter than the kernel-level threads.
- It does not require modifications of the operating system.
- User-level thread representation is very simple. The register, PC, stack, and mini thread control blocks are stored in the address space of the user-level process.
- It is simple to create, switch, and synchronize threads without the intervention of the process.

Kernel Level Threads

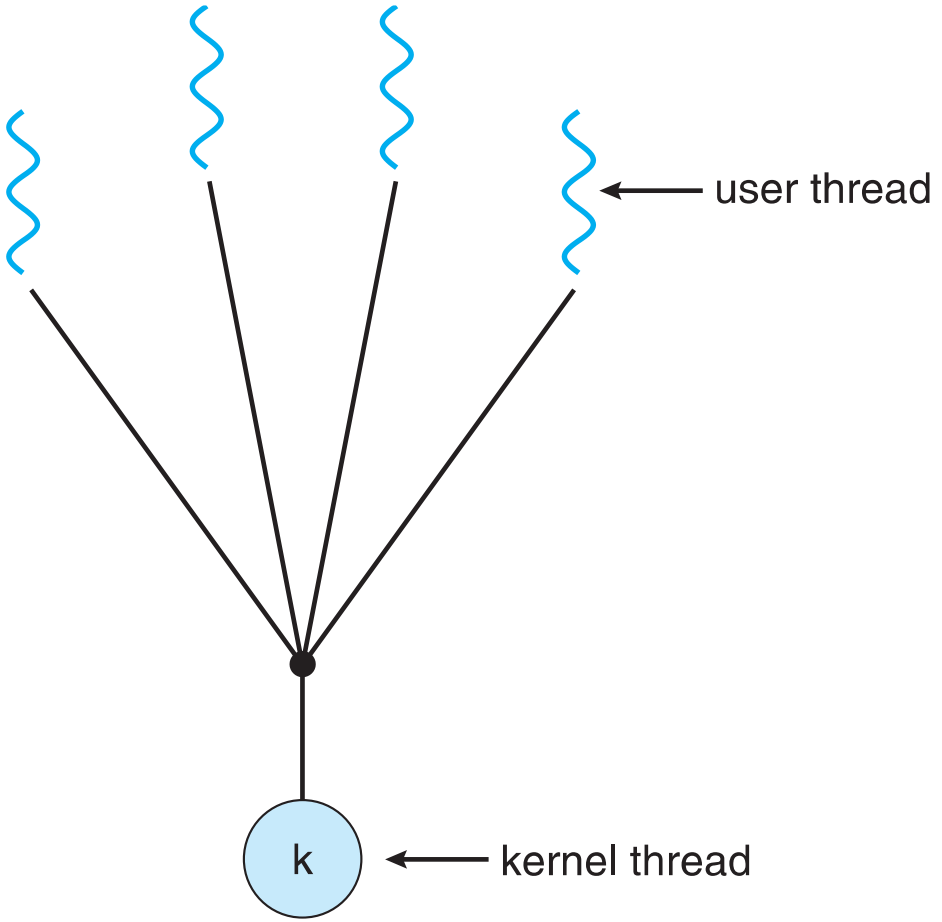
- The kernel thread recognizes the operating system. There is a thread control block and process control block in the system for each thread and process in the kernel-level thread.
- The kernel-level thread is implemented by the operating system. The kernel knows about all the threads and manages them.
- If a kernel thread performs a blocking operation, the other threads' execution can continue.

Multithreading Models

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

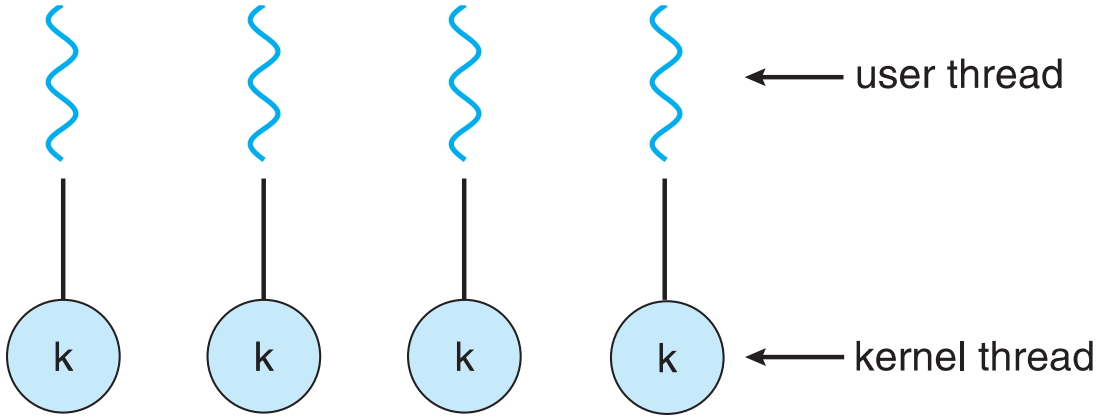
Many-to-One

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



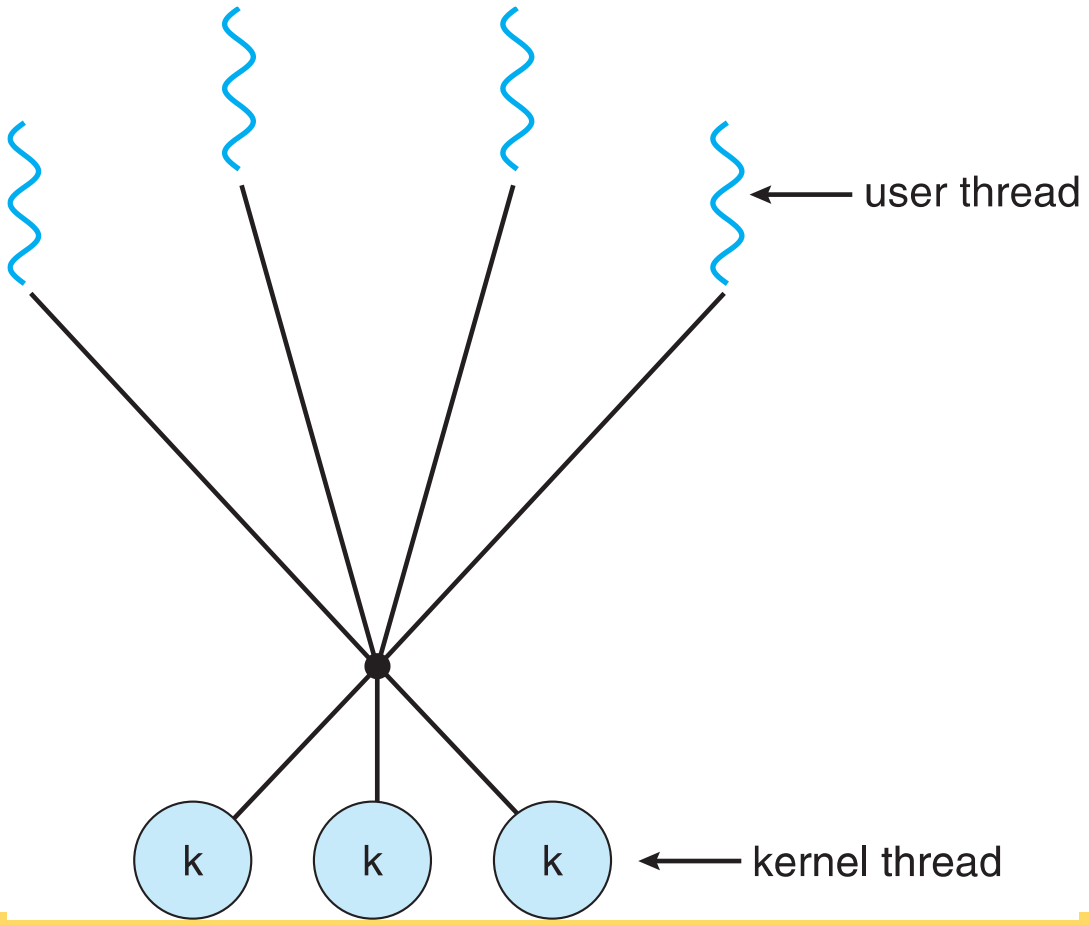
One-to-One

- 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



Many-to-Many Model

- 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



Asynchronous Vs Synchronous

- Two general strategies for creating multiple threads: asynchronous threading and synchronous threading.
- Each thread runs independently of every other thread, and the parent thread need not know when its child terminates. Because the threads are independent, there is typically little data sharing between threads. **Asynchronous** threading is the strategy used in the multithreaded server
- **Synchronous** threading occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes —the so-called ***fork-join*** strategy.

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Multitasking in Mobile Systems

- Some mobile systems (e.g., the early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include short tasks, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if the background process is suspended
  - Service has no user interface, small memory use

Process Creation

- **Parent** process creates **children** processes, which, in turn, create other processes, forming a **tree** of processes
- Generally, a process identified and managed via a **process identifier** (**pid**)

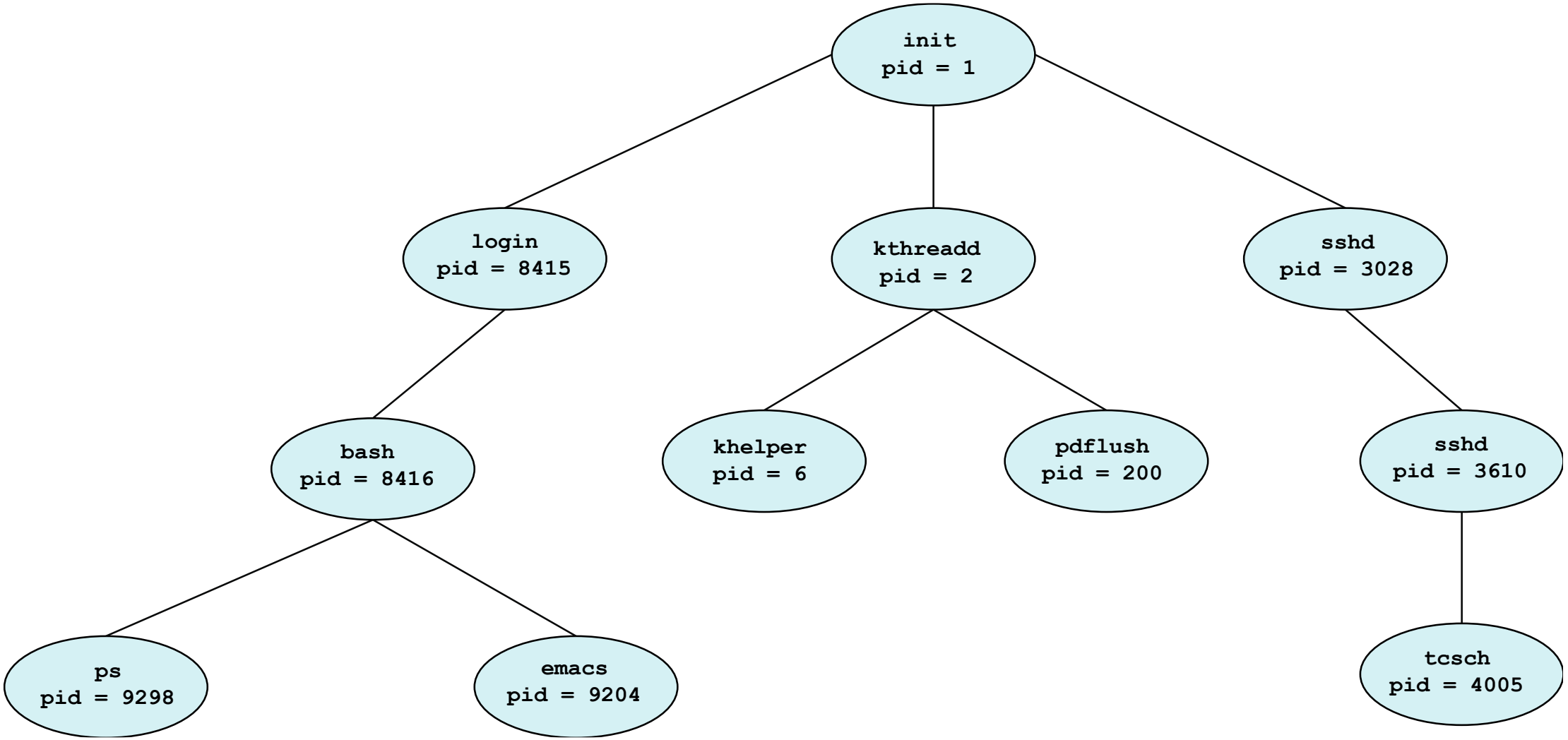
Process Creation (Cont.)

When a process creates a new process, two possibilities for execution exist:

- The parent continues to execute concurrently with its children.
- The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

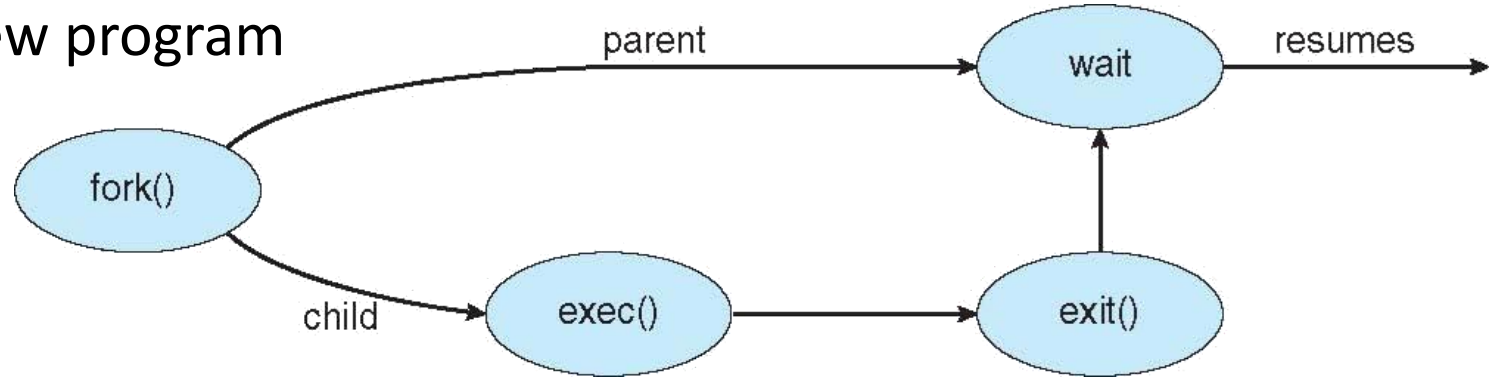
- The child process is a duplicate of the parent process (it has the same program and data as the parent).
- The child process has a new program loaded into it.



A Tree of Processes in Linux

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates a new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



Process Termination

- Process executes the last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process resources are deallocated by the operating system
- The Parent may terminate the execution of the children's processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to a child is no longer required
  - The parent is exiting and the operating system does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for the termination of a child process by using the **`wait()`** system call. The call returns status information and the PID of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**), the process is a **zombie**
- If the parent terminated without invoking **`wait`**, the process is an **orphan**

```
int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

```
int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE A */
      return 0;
   }
}
```

What will be output of this code?

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

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

How many processes are created?

How many processes are created?

```
int main()
{
    if (fork() || fork())
        fork();
    printf("1");
    return 0;
}
```

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as a single process (some still do)
  - If one website causes trouble, the entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk, and network I/O
  - **Renderer** process renders web pages and deals with HTML, and Javascript. A new renderer is created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Part 2 : IPC & Process Synchronization

Cooperation among processes.

Inter-process Communication

“Interprocess Communication or IPC provides a mechanism to exchange data and information across multiple processes, which might be on single or multiple computers connected by a network”.

"Inter-process communication is used for exchanging useful information between numerous threads in one or more processes (or programs)."

Inter-process Communication

- Three problems
  - How to actually do it
  - How to deal with process conflicts (2 airline reservations for the same seat)
  - How to do correct sequencing when dependencies are present, aim the gun before firing it
- Same issues for threads as for processes solutions as well

Cooperating Processes

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

Cooperating Processes

- **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.

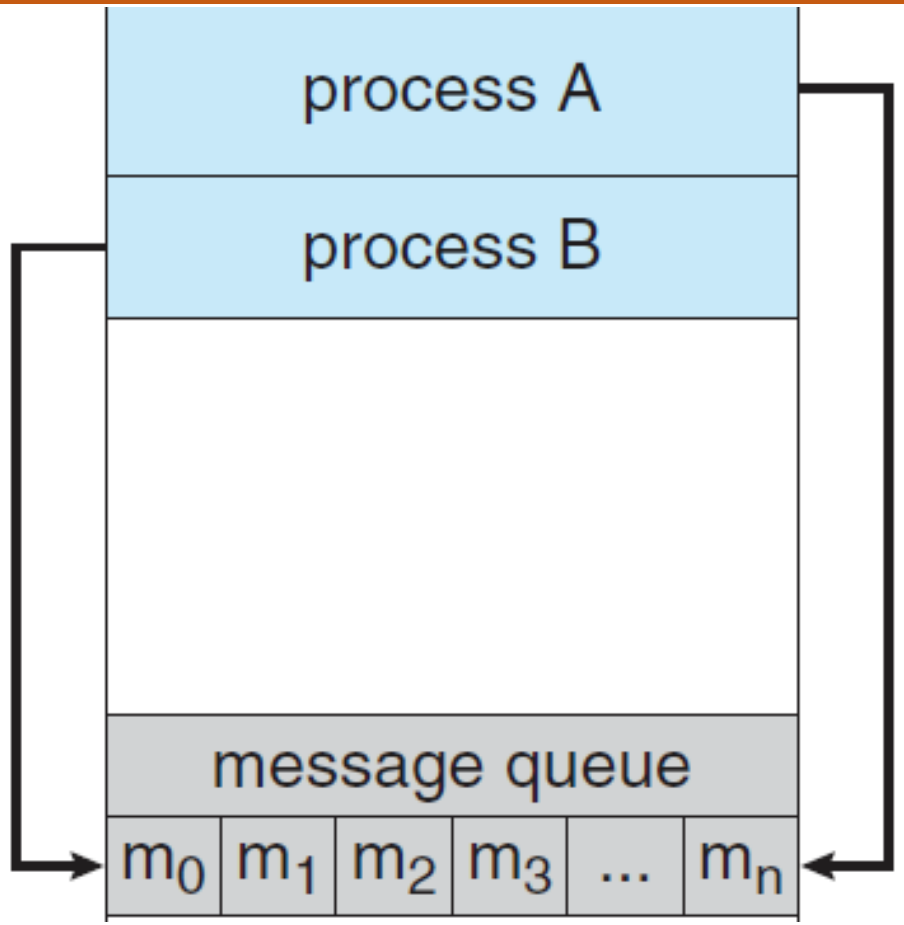
Inter-process Communication

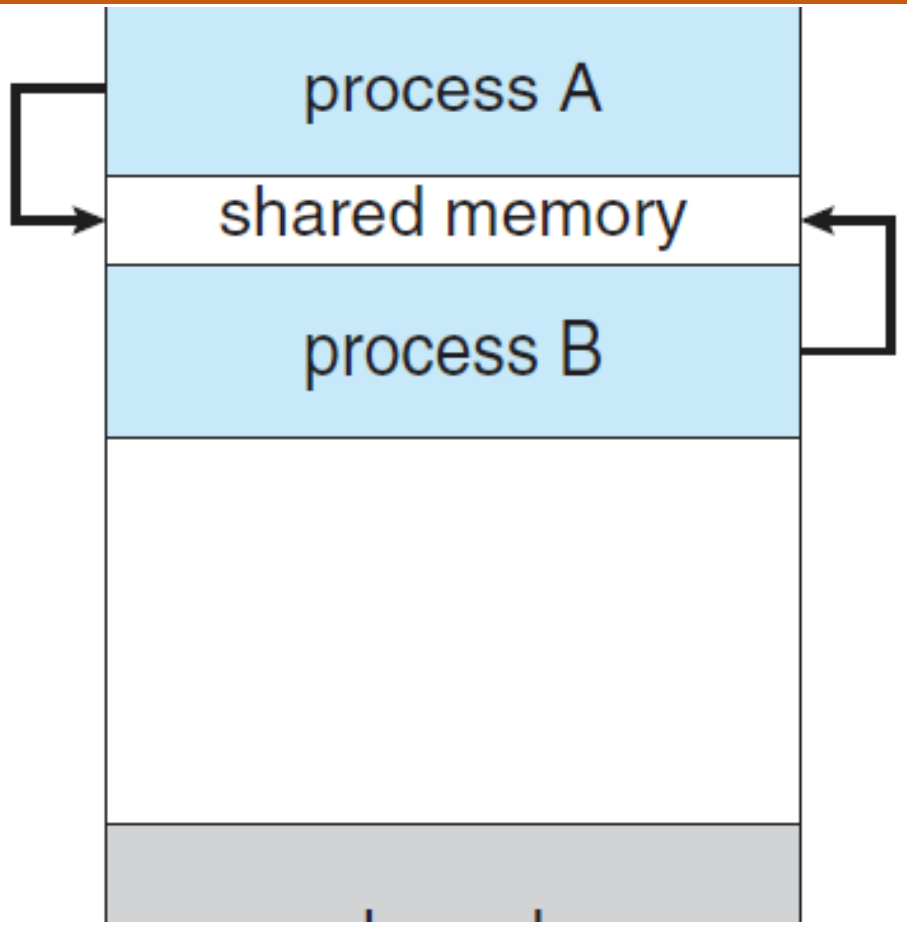
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Inter-process Communication

- Message passing is useful for **exchanging smaller amounts of data**, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory.
- Shared memory can be faster than message passing,
- Since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
- In shared-memory systems, system calls are required 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.

Inter-process Communication





Message Passing

- A message-passing facility provides at least two operations:

  send(message) **receive(message)**

- Messages sent by a process can be either fixed or variable in size.
  - fixed size makes the task of programming more difficult.
  - variable-sized messages require a more complex system level implementation

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.
- Methods for logically implementing a link and the send()/receive() operations:
  - Direct or indirect communication
  - Synchronous or asynchronous communication

Message Passing: Direct Communication

- Under **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.

Message Passing: Direct Communication

- A communication link in this scheme has the following properties:
  1. 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.
  2. A link is associated with exactly two processes.
  3. Between each pair of processes, there exists exactly one link.

Message Passing: Indirect Communication

- With ***indirect communication***, the messages are sent to and received from ***mailboxes***, or ***ports***. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- For Example, POSIX message queues use an integer value to identify a 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.

Message Passing: Indirect Communication

- A communication link has the following properties:
  1. A link is established between a pair of processes only if both members of the pair have a shared mailbox.
  2. A link may be associated with more than two processes.
  3. Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Message Passing: Synchronous or Asynchronous Communication

- Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**.
  - **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - **Nonblocking send**. The sending process sends the message and resumes operation.
  - **Blocking receive**. The receiver blocks until a message is available.
  - **Nonblocking receive**. The receiver retrieves either a valid message or a null.

Message Passing: Producer-Consumer Problem

```
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 */
}
```

Shared Memory: Producer-Consumer Problem

Paradigm for cooperating processes, The producer process produces information that is consumed by a consumer process.

- **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.

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

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

- Solution is correct, but can only use BUFFER_SIZE-1 elements

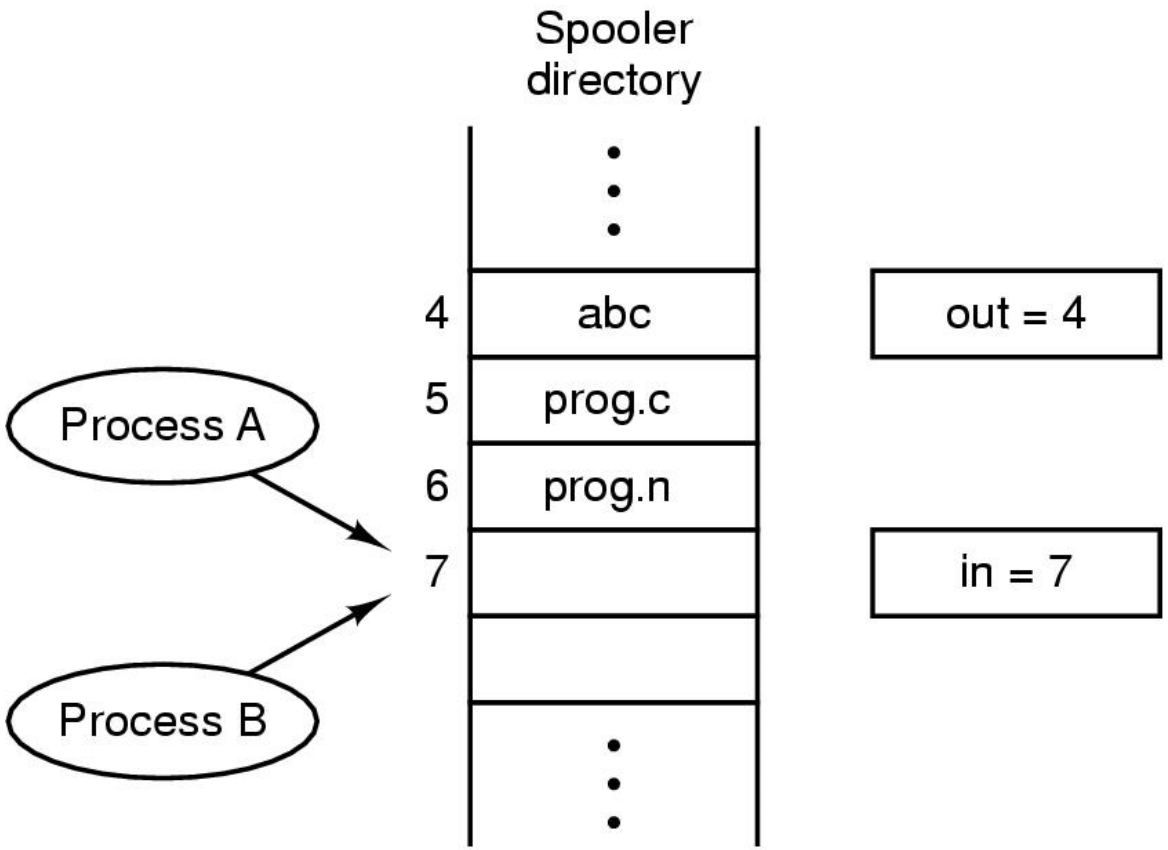
Bounded-Buffer – Producer

```
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;
}
```

Bounded Buffer – Consumer

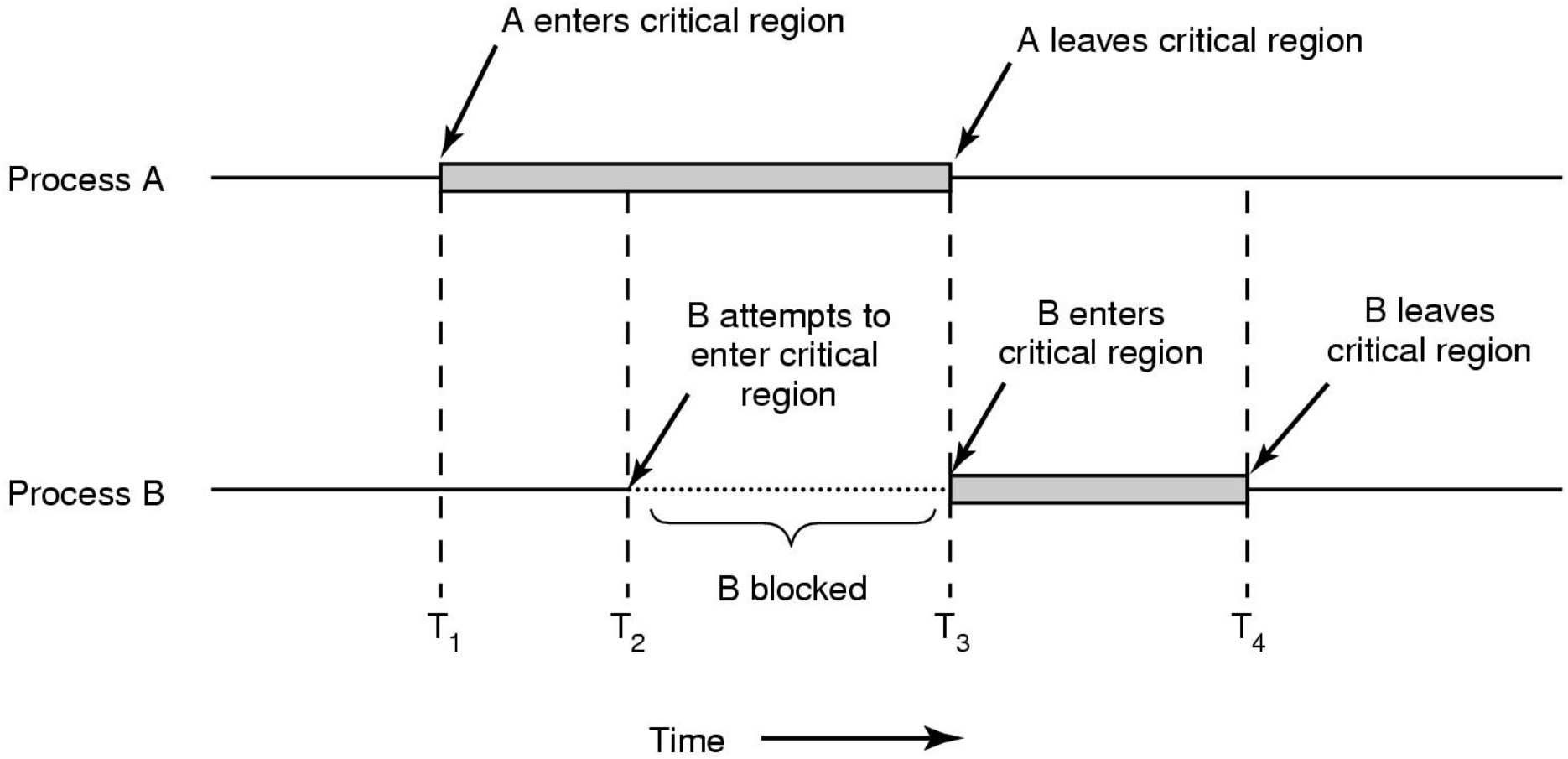
```
item next_consumed;
while (true) {
     while (in == out)
          ; /* do nothing */
     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

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

Race Conditions



- **IN** is a local variable containing a pointer to the next free slot
- **Out** is a local variable pointing to the next file to be printed

What we are trying to do



Critical Section Problem

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

Critical Section

- General structure of process P_i

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

Algorithm for Process Pi

```
do {

    while (turn == j);

        critical section
    turn = j;

        remainder section
  } while (true);
```

How to avoid races

- Mutual exclusion–only one process at a time can use a shared variable/file
- Critical regions-shared memory which leads to races
- Solution- Ensure that two processes can’t be in the critical region at the same time

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **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
3. **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
   - Assume that each process executes at a nonzero speed
   - No assumption concerning the **relative speed** of the ***n*** processes

Properties of a good solution

- Mutual exclusion
- No assumptions about speeds or number of CPUs
- No process outside the critical region can block other processes
- No starvation-no process waits forever to enter the critical region

First attempts-Busy Waiting

- A laundry list of proposals to achieve mutual exclusion

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

Disabling Interrupts

- Idea: process disables interrupts, enters cr, enables interrupts when it leaves cr
- Problems
  - Process might never enable interrupts, crashing system
  - Won't work on multi-core chips as disabling interrupts only affects one CPU at a time

Lock variables

- A software solution-everyone shares a lock
  - When the lock is 0, the process turns it to 1 and enters cr
  - When exiting cr, turn the lock to 0
- Problem-Race condition

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

First me, then you

Problems with strict alternation

- Employs busy waiting-while waiting for the cr, a process spins
- If one process is outside the cr and it is its turn, then the other process has to wait until outside guy finishes both outside AND inside (cr) work

Peterson's Solution

-

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
        int other;                      /* number of the other process */

        other = 1 – process;            /* the opposite of process */
        interested[process] = TRUE;     /* show that you are interested */
        turn = process;                 /* set flag */
        while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
        interested[process] = FALSE;    /* indicate departure from critical region */
}
```

Lock

-

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

Test and Set lock

-

```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

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

The Producer-Consumer Problem (Bounded Buffer Problem)

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item( );         /* generate next item */
                if (count == N) sleep( );       /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer); /* was buffer empty? */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
                item = remove_item( );          /* take item out of buffer */
                count = count − 1;              /* decrement count of items in buffer */
                if (count == N − 1) wakeup(producer); /* was buffer full? */
                consume_item(item);             /* print item */
        }
}
```

The problem with sleep and wake-up calls

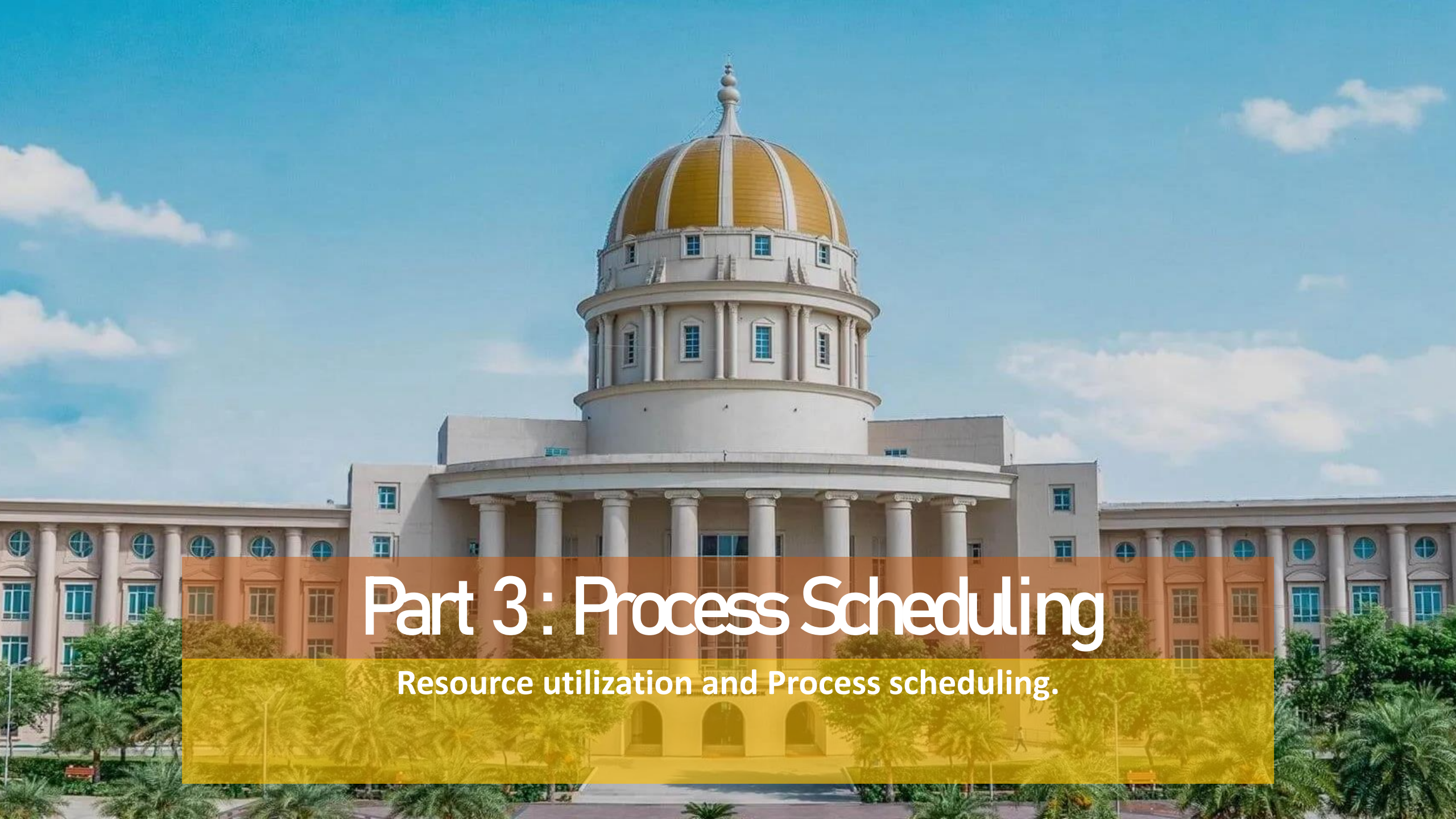
- Empty buffer,count==0
- Consumer gets replaced by producer before it goes to sleep
- Produces something, count++, sends wakeup to consumer
- Consumer not asleep, ignores wakeup, thinks count= = 0, goes to sleep
- Producer fills the buffer, goes to sleep
- P and C sleep forever
- So, the problem is lost wake-up calls

Semaphores

- Semaphore is an integer variable
- Used to sleeping processes/wakeups
- Two operations, down and up
- Down checks semaphore. If not zero, decrements semaphore. If zero, the process goes to sleep
- Up increments semaphore. If more than one process is asleep, one is chosen randomly and enters a critical region (first does a down)
- ATOMIC IMPLEMENTATION-interrupts disabled

Producer Consumer with semaphores

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                 /* TRUE is the constant 1 */
                item = produce_item( );/* generate something to put in buffer */
                down(&empty);          /* decrement empty count */
                down(&mutex);          /* enter critical region */
                insert_item(item);     /* put new item in buffer */
                up(&mutex);            /* leave critical region */
                up(&full);             /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                 /* infinite loop */
                down(&full);           /* decrement full count */
                down(&mutex);          /* enter critical region */
                item = remove_item( );/* take item from buffer */
                up(&mutex);            /* leave critical region */
                up(&empty);            /* increment count of empty slots */
                consume_item(item);    /* do something with the item */
        }
}
```



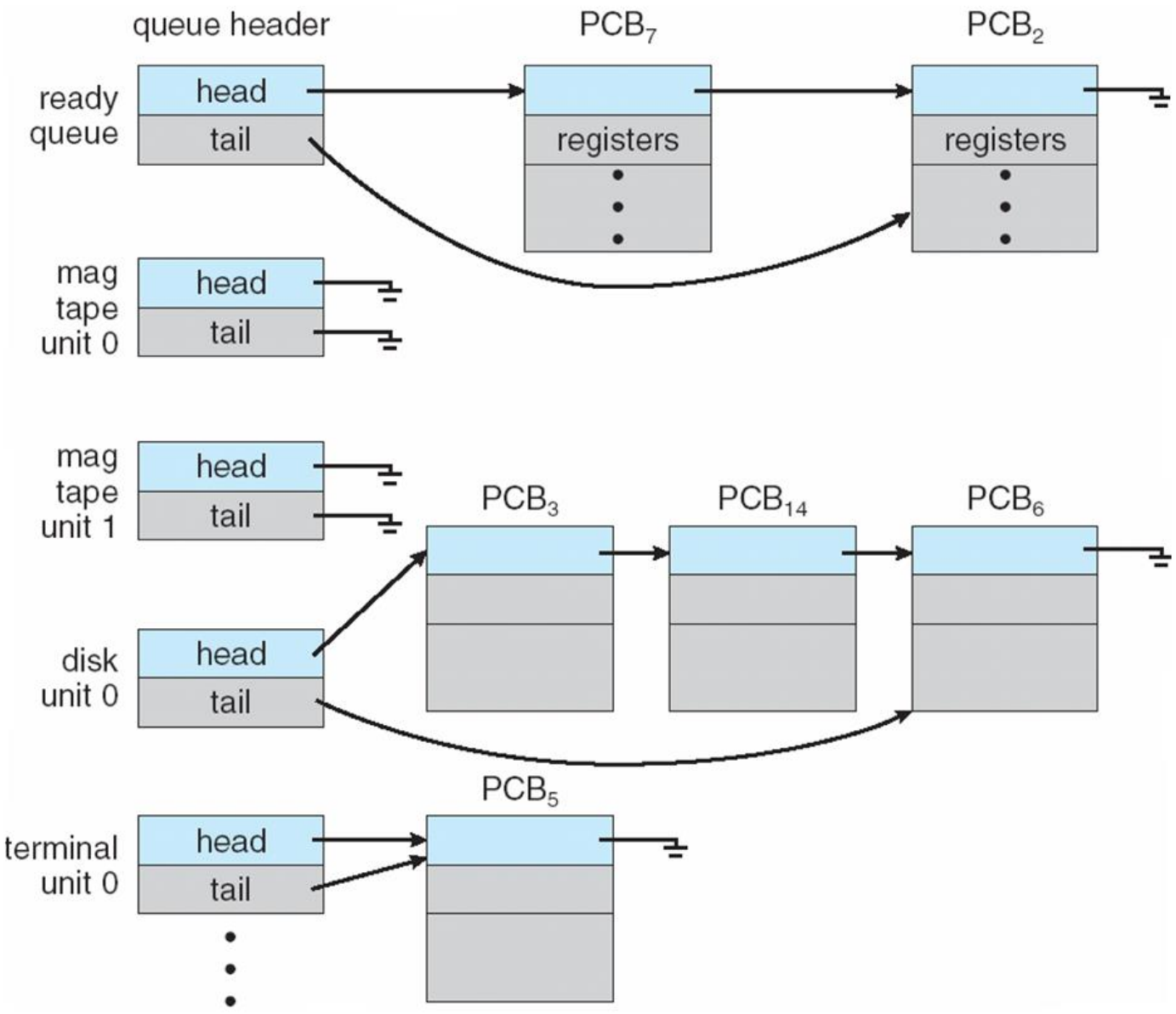
Part 3 : Process Scheduling

Resource utilization and Process scheduling.

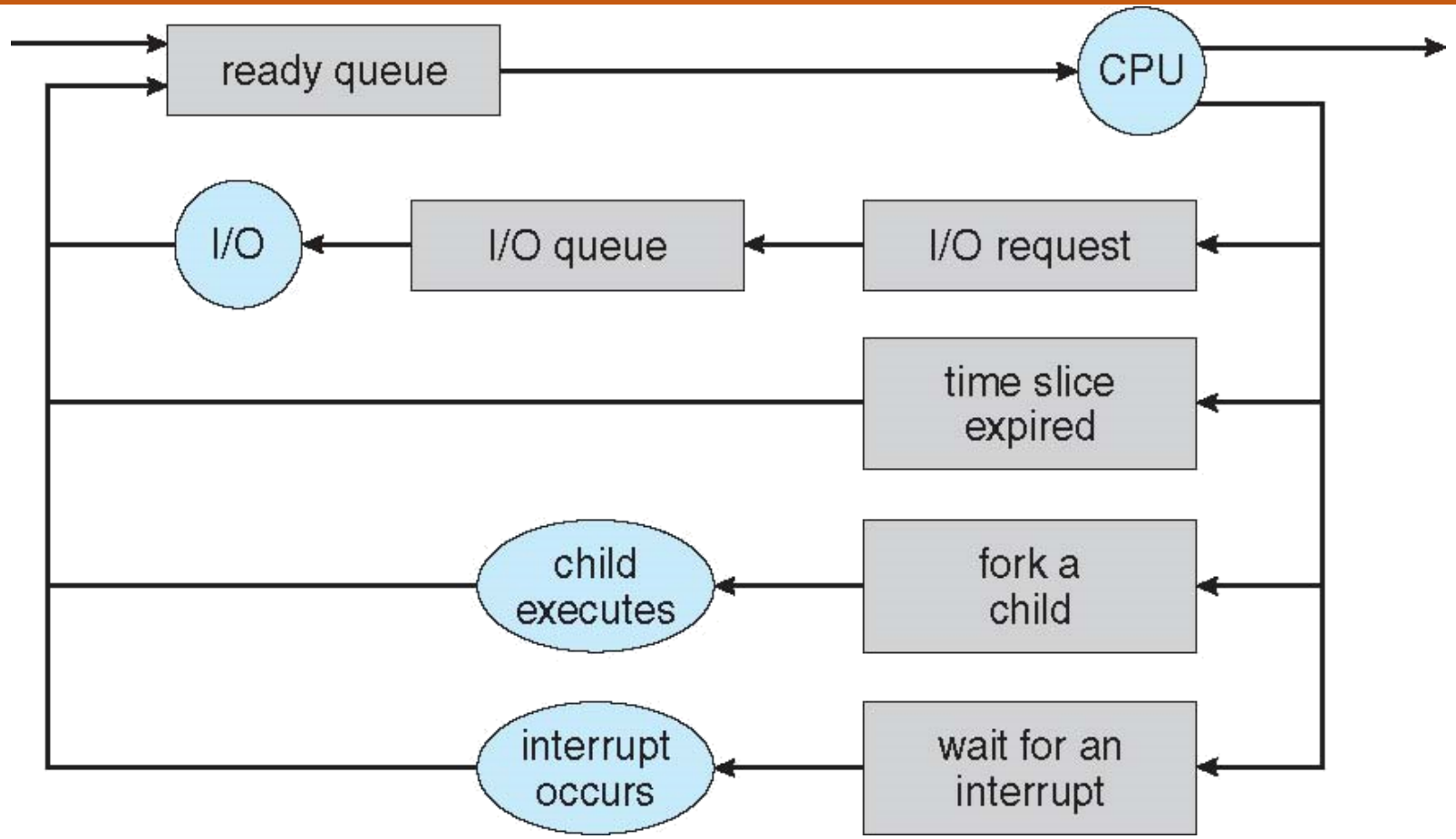
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time-sharing
- **Process scheduler** selects among available processes for the next execution on 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** – a set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Schedulers

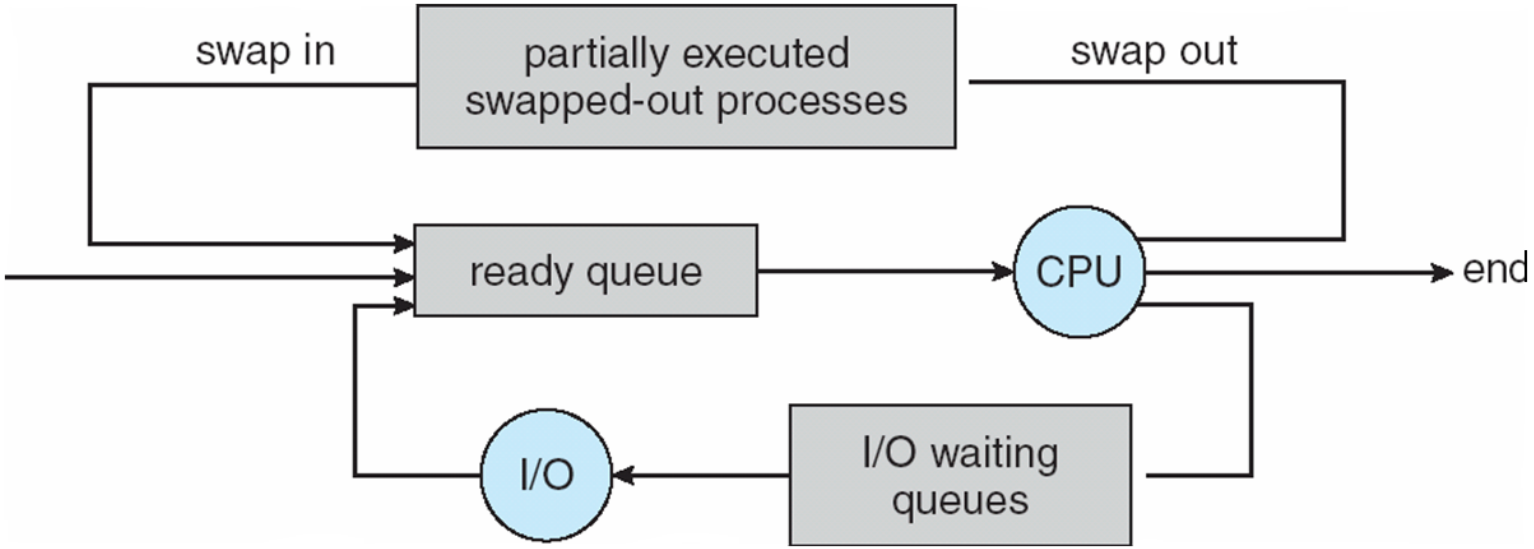
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**

Schedulers

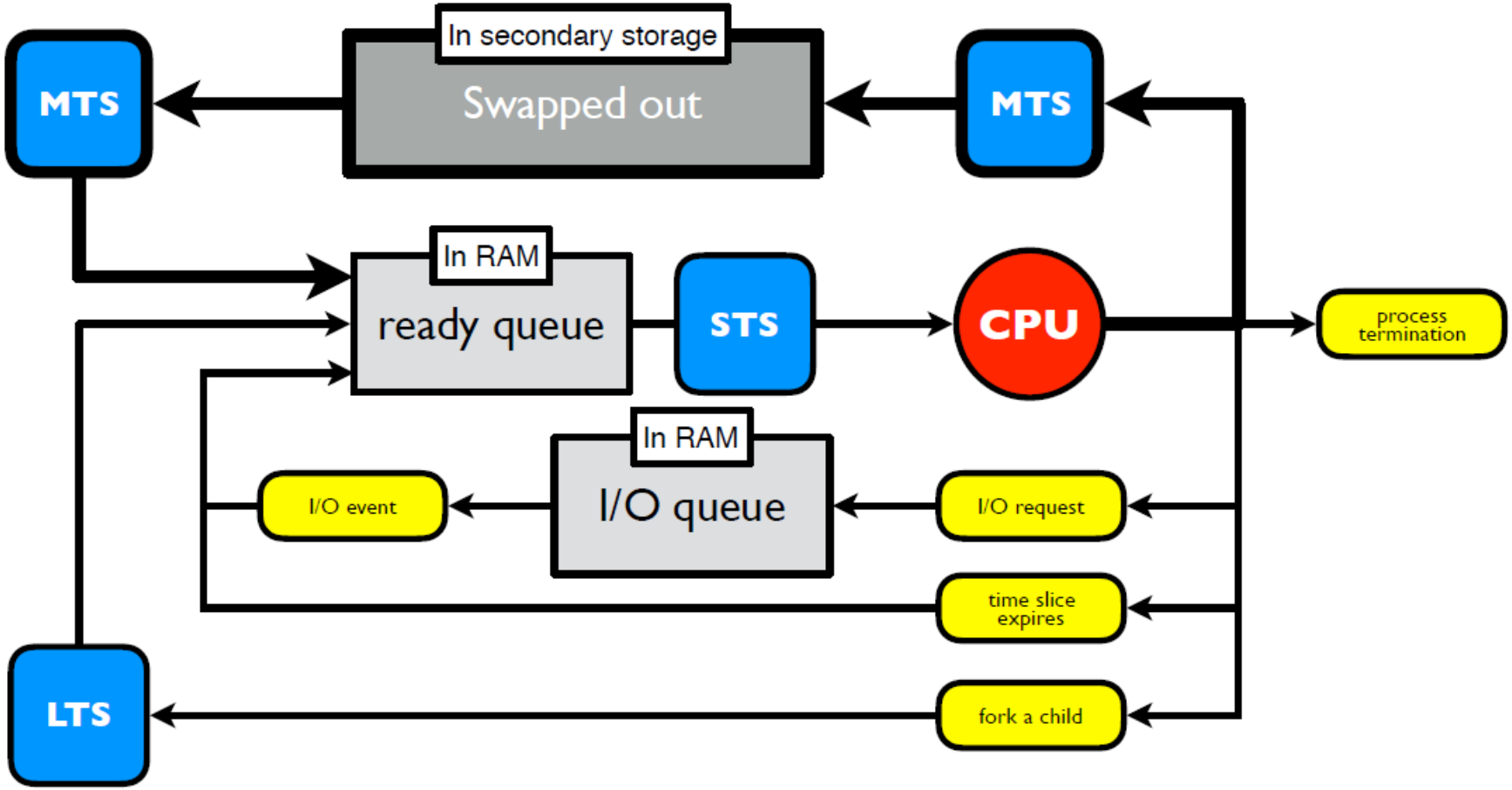
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- **Long-term scheduler** strives for good ***process mix***

Addition of Medium-Term Scheduling

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



Scheduler's


In secondary storage
Swapped out
MTS
MTS
In RAM
ready queue
STS
CPU
process termination
In RAM
I/O queue
I/O event
I/O request
time slice expires
LTS
fork a child

S. No.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Scheduling Algorithm's

Which is the best CPU scheduling algorithm?

This is a common question that comes to our mind. An Operating System uses different algorithms for CPU scheduling. Every algorithm can perform better depending on the situation.

There are five points to decide which algorithm performs better than others.

1. **CPU utilization** – It means how much the CPU is busy. That is, for a given duration of time how long was the CPU doing the processing? The CPU utilization should be high (40-90%)
2. **Throughput** – Throughput is the number of processes that complete execution in a given time.
3. **Turnaround time** – is the interval from the time of submission of the process to the time of its completion

Scheduling Algorithm's

4. Waiting time – is the time spent by the process in the ready queue (waiting for the CPU)

5. Response time – is the time from the submission of the process until it gives the first response.

*Note: An algorithm is effective if it has **high** CPU utilization and throughput, whereas **low** turnaround, waiting and response time.*

CPU Scheduling Algorithm's

- Arrival Time: The time at which the process enters the ready queue or state.
- Burst Time: Time required by a process to get executed on the CPU.
- Completion Time: The time at which the process completes its execution.
- Turnaround Time: Completion Time – Arrival Time
- Waiting Time: Turn-around Time – Burst Time
- Response Time: Time at which process get CPU for the first time – Arrival Time

CPU Scheduling Algorithm's

Pre-emptive Scheduling
Shortest Remaining Time First (SRTF)
Largest Remaining Time First (LRTF)
Round Robin
Priority Based

Non Pre-emptive Scheduling
First Come First Serve (FCFS)
Shortest Job First (SJF)
Largest Job First (LJF)
Multi-level Queue
Highest Response Ratio Next (HRRN)

First Come First Serve

- It is a simple scheduling algorithm.
- The idea is that the process that comes first must use the resource first.
- It schedules according to the arrival time of the process.
- It states that process which request the CPU first is allocated the CPU first.
- It is implemented by using *FIFO (First Come First Serve) queue* and is a *Non Pre-emptive* scheduling algorithm.

First Come First Serve

PROCESS	BURST TIME	WAITING TIME
P1	8	0
P2	3	8
P3	5	11
P4	3	16

First Come First Serve

Process	Arrival Time	Burst Time	Completion Time	Turn-around Time	Waiting Time	Response Time
P1	0	2	2	2	0	0
P2	1	2	4	3	1	1
P3	5	3	8	3	0	0
P4	6	4	12	6	2	2

P1	P2		P3	P4

Gantt Chart

First Come First Serve

Advantages of FCFS :

- Simple
- Easy, useful and understandable
- First come, first served.

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.

Disadvantages of FCFS :

- Because of non pre-emptive scheduling, the process will continue to run until it is finished.
- As the scheduling is non pre-emptive so short processes which are at end of ready queue have to wait for a larger time making them starve leading to problem of starvation.
- Throughput is not efficient.

First Come First Serve

Example :

Consider the following set of four processes. Their arrival time and time required to complete the execution are given in following table. Consider all time values in milliseconds.

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

First Come First Serve

Statistics:

Process	Arrival Time(T0)	CPU Burst Time(ΔT)	Finish Time(T1)	Turnaround Time(TAT=T1-T0)	Waiting Time (WT=TAT-ΔT)
P0	0	10	10	10	0
P1	1	6	16	15	9
P2	3	2	18	15	13
P3	5	4	22	17	13

Average Turnaround Time: (10+15+15+17)/4 = 57/4 = 14.25 ms.

Average Waiting Time: (0+9+13+13)/4 = 35/4 = 8.75 ms.

Shortest Job First (SJF)

- Process with the shortest burst time is scheduled first.
- If two processes have the same burst time, then FCFS is used to break the tie.
- SJF Scheduling can be used in both pre-emptive and non-preemptive modes.
- The Pre-emptive mode of Shortest Job First is called **Shortest Remaining Time First (SRTF)**.

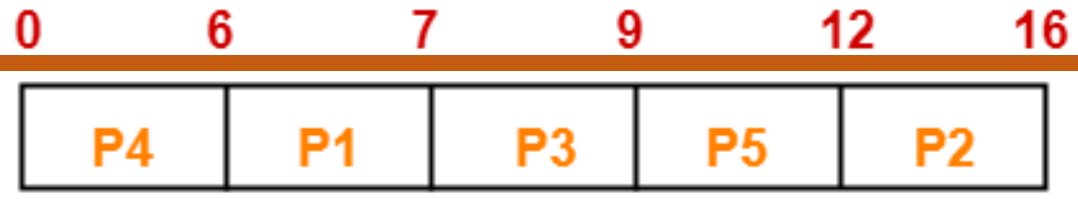
Shortest Job First (SJF)

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

Process Id	Arrival time	Burst time
P_1	3	1
P_2	1	4
P_3	4	2
P_4	0	6
P_5	2	3

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

Shortest Job First (SJF)



Gantt Chart

- Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit
- Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit

Process Id	Exit time	Turn Around time	Waiting time
P_1	7	7 – 3 = 4	4 – 1 = 3
P_2	16	16 – 1 = 15	15 – 4 = 11
P_3	9	9 – 4 = 5	5 – 2 = 3
P_4	6	6 – 0 = 6	6 – 6 = 0
P_5	12	12 – 2 = 10	10 – 3 = 7

Shortest Job First (SJF)

Example: 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 turnaround time.

Shortest Job First (SJF)

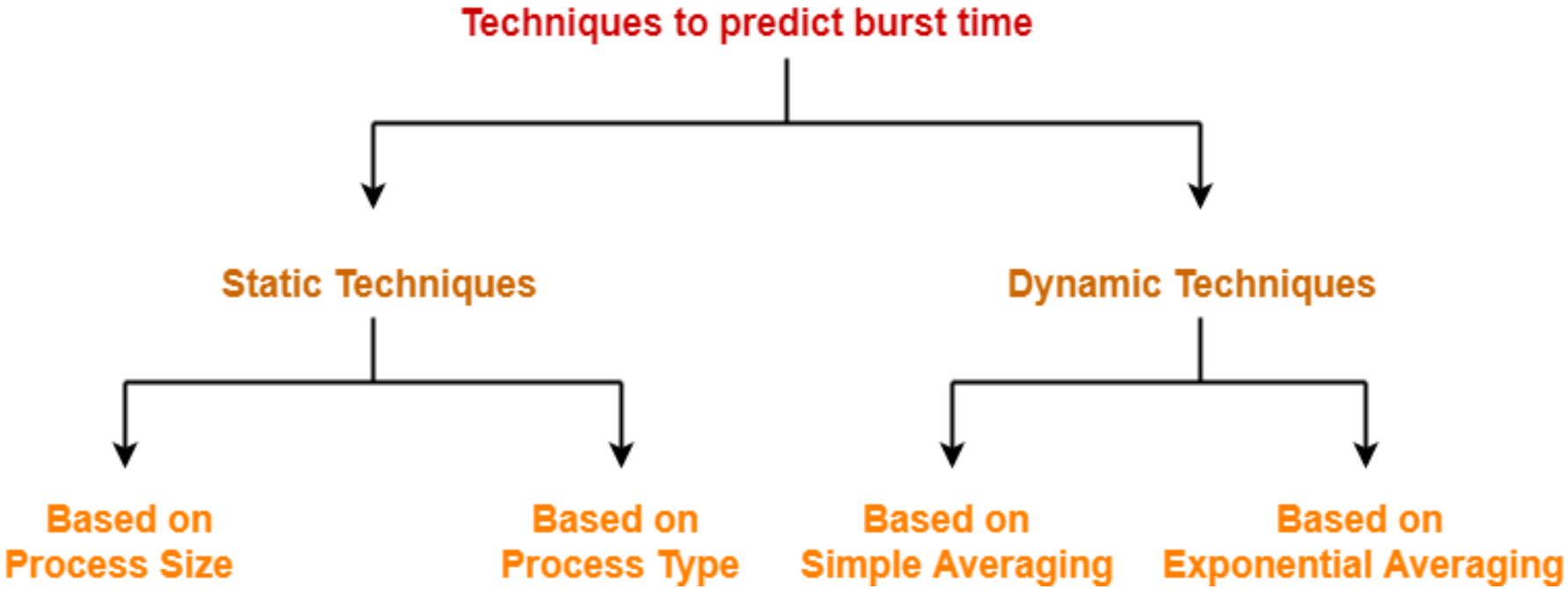
Example: Consider the set of 5 processes whose arrival time and burst time are given below

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 the shortest remaining time first, calculate the average waiting time and average turnaround time.

Technique to predict the Burst Time

- Several techniques try to predict the burst time for the processes so that the algorithm can be implemented.



Static Techniques to predict the Burst Time

1. Based on Process Size

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

Example-

- Consider a process of size 200 KB that took 20 units of time to complete its execution.
- Then, burst time for any future process having a size around 200 KB can be taken as 20 units.

NOTE

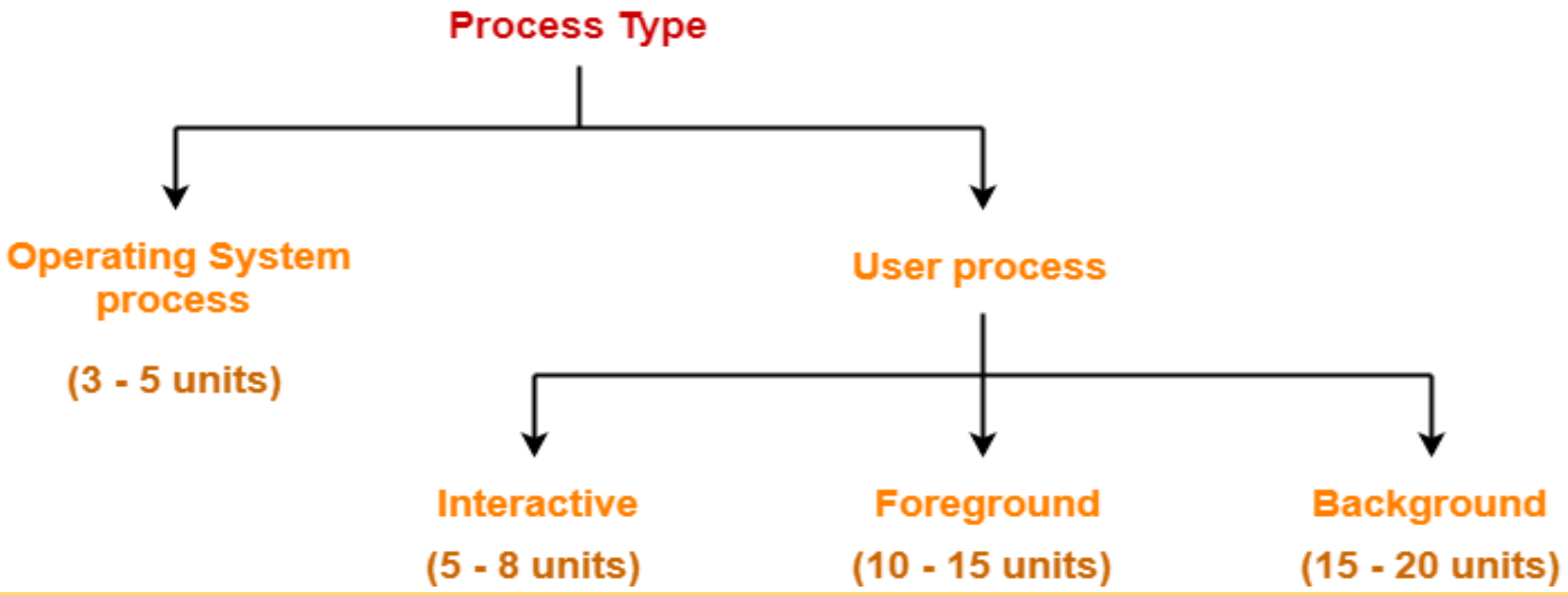
•The predicted burst time may not always be right.

•This is because the burst time of a process also depends on what kind of a process it is.

Static Techniques to predict the Burst Time

2. <u>Based on Process Type</u>

- 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.



Dynamic Techniques to predict the Burst Time

1. 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 $P_1, P_2, \ldots, P_n$ and burst time of each process P_i as t_i, then predicted burst time for process P_{n+1} is given as

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

Dynamic Techniques to predict the Burst Time

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

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

where-

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

Example

- 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 the actual burst time of the first four processes is 4, 8, 6, and 7 units respectively. Given $\alpha = 0.5$.

Solution-

Given:

- Predicted burst time for 1st process = 10 units
- Actual burst time of the first four processes = 4, 8, 6, 7
- $\alpha = 0.5$

Example

Predicted Burst Time for 2nd Process-

Predicted burst time for 2nd process
= α x Actual burst time of 1st process + (1-α) x Predicted burst time for 1st process
= 0.5 x 4 + 0.5 x 10
= 2 + 5
= 7 units

Predicted Burst Time for 3rd Process-

Predicted burst time for 3rd process
= α x Actual burst time of 2nd process + (1-α) x Predicted burst time for 2nd process
= 0.5 x 8 + 0.5 x 7
= 4 + 3.5
= 7.5 units

Example

Predicted Burst Time for 4th Process-

Predicted burst time for 4th process
= α x Actual burst time of 3rd process + (1-α) x Predicted burst time for 3rd process
= 0.5 x 6 + 0.5 x 7.5
= 3 + 3.75
= 6.75 units

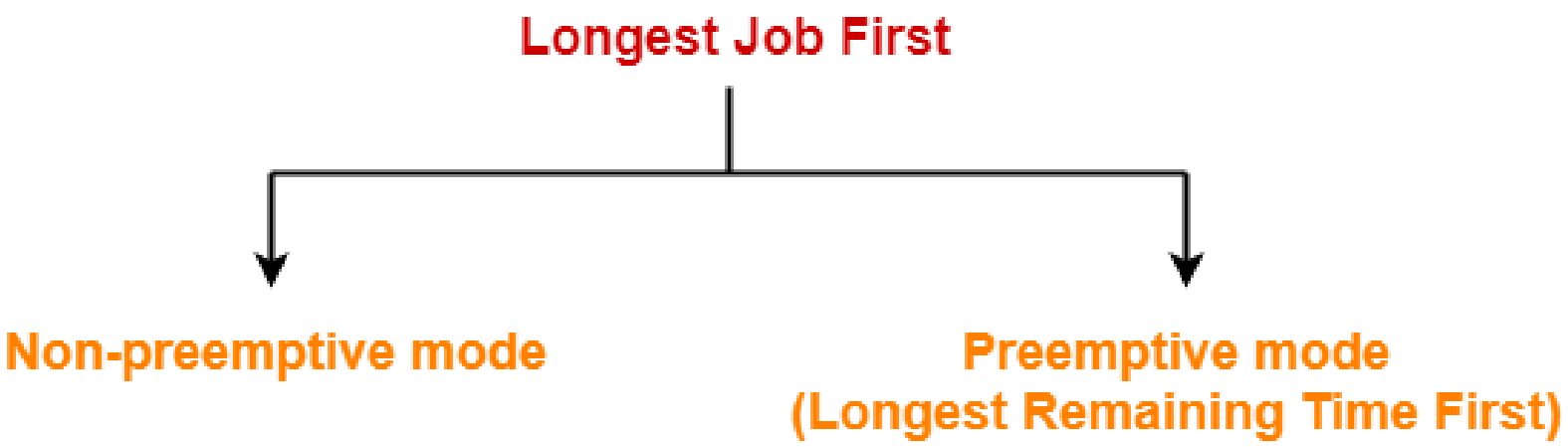
Predicted Burst Time for 5th Process-

Predicted burst time for 5th process
= α x Actual burst time of 4th process + (1-α) x Predicted burst time for 4th process
= 0.5 x 7 + 0.5 x 6.75
= 3.5 + 3.375
= 6.875 units

Longest Job First Algorithm

In LJF Scheduling,

- Out of all the available processes, the CPU is assigned to the process having the largest burst time.
- In case of a tie, it is broken by **FCFS Scheduling**.



Longest Job First Algorithm

- LJF Scheduling can be used in both preemptive and non-preemptive mode.
- Preemptive mode of Longest Job First is called as **Longest Remaining Time First (LRTF)**.

Longest Job First Algorithm

- **Advantages-**

➢ No process can be completed until the longest job also reaches its completion.

➢ All the processes approximately finish at the same time.

- **Disadvantages-**

➢ The waiting time is high.

➢ Processes with smaller burst times may starve for CPU.

Longest Job First Algorithm

- **Example:** Consider the set of 5 processes whose arrival time and burst time are given below. If the CPU scheduling policy is LJF non-preemptive, calculate the average waiting time and average turnaround time.

Process Id	Arrival time	Burst time
P1	0	3
P2	1	2
P3	2	4
P4	3	5
P5	4	6

Longest Job First Algorithm

Process Id	Arrival time	Burst time
P1	0	3
P2	1	2
P3	2	4
P4	3	5
P5	4	6

Solution:

Gantt Chart

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	3	3 – 0 = 3	3 – 3 = 0
P2	20	20 – 1 = 19	19 – 2 = 17
P3	18	18 – 2 = 16	16 – 4 = 12
P4	8	8 – 3 = 5	5 – 5 = 0
P5	14	14 – 4 = 10	10 – 6 = 4

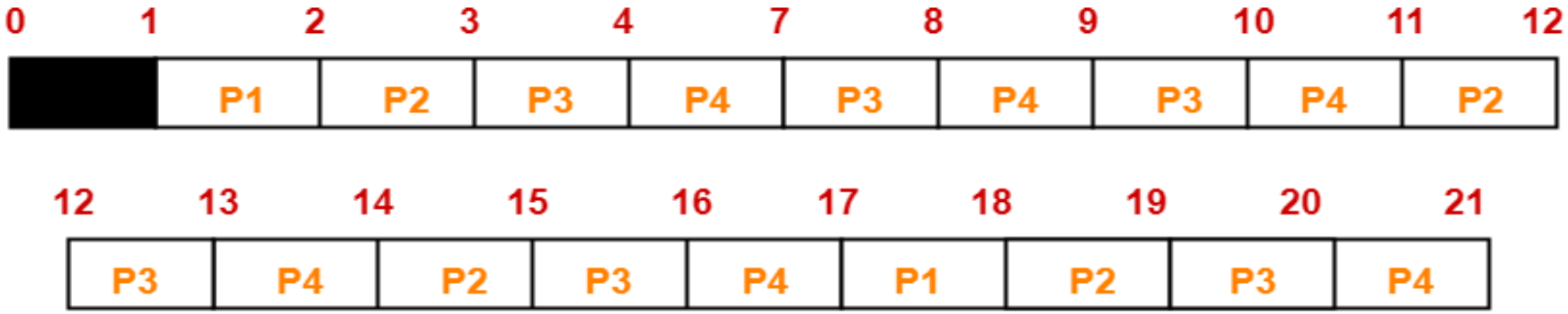
- Average Turn Around time = (3 + 19 + 16 + 5 + 10) / 5 = 53 / 5 = 10.6 unit
- Average waiting time = (0 + 17 + 12 + 0 + 4) / 5 = 33 / 5 = 6.6 unit

Longest Job First Algorithm

- **Example:** Consider the set of 4 processes whose arrival time and burst time are given below. If the CPU scheduling policy is LJF preemptive, calculate the average waiting time and average turnaround time.

Process Id	Arrival time	Burst time
P1	1	2
P2	2	4
P3	3	6
P4	4	8

Solution:



Gantt Chart

Process Id	Exit time	Turn Around time	Waiting time
P1	18	18 – 1 = 17	17 – 2 = 15
P2	19	19 – 2 = 17	17 – 4 = 13
P3	20	20 – 3 = 17	17 – 6 = 11
P4	21	21 – 4 = 17	17 – 8 = 9

- Average Turn Around time = (17 + 17 + 17 + 17) / 4 = 68 / 4 = 17 unit
- Average waiting time = (15 + 13 + 11 + 9) / 4 = 48 / 4 = 12 unit

Highest Response Ration Next Algorithm

In HRRN Scheduling,

- Out of all the available processes, the CPU is assigned to the process having the highest response ratio.
- In case of a tie, it is broken by **FCFS Scheduling**.
- It operates only in non-preemptive mode.

Highest Response Ration Next Algorithm

Response Ratio (RR) for any process is calculated by using the formula-

where-

- W = Waiting time of the process so far
- B = Burst time or Service time of the process

$$\text{Response Ratio} = \frac{W + B}{B}$$

Highest Response Ration Next Algorithm

- **Advantages-**

➢ It performs better than **SJF Scheduling**.

➢ It not only favors the shorter jobs but also limits the waiting time of longer jobs.

- **Disadvantages-**

➢ It can not be implemented practically.

➢ This is because burst time of the processes cannot be known in advance.

Highest Response Ration Next Algorithm

Example:

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

Process Id	Arrival time	Burst time
P0	0	3
P1	2	6
P2	4	4
P3	6	5
P4	8	2

If the CPU scheduling policy is Highest Response Ratio Next, calculate the average waiting time and average turnaround time.

Example:

Gantt Chart

Process Id	Exit time	Turn Around time	Waiting time
P0	3	3 – 0 = 3	3 – 3 = 0
P1	9	9 – 2 = 7	7 – 6 = 1
P2	13	13 – 4 = 9	9 – 4 = 5
P3	20	20 – 6 = 14	14 – 5 = 9
P4	15	15 – 8 = 7	7 – 2 = 5

Average Turn Around time = (3 + 7 + 9 + 14 + 7) / 5

= 40 / 5 = 8 units

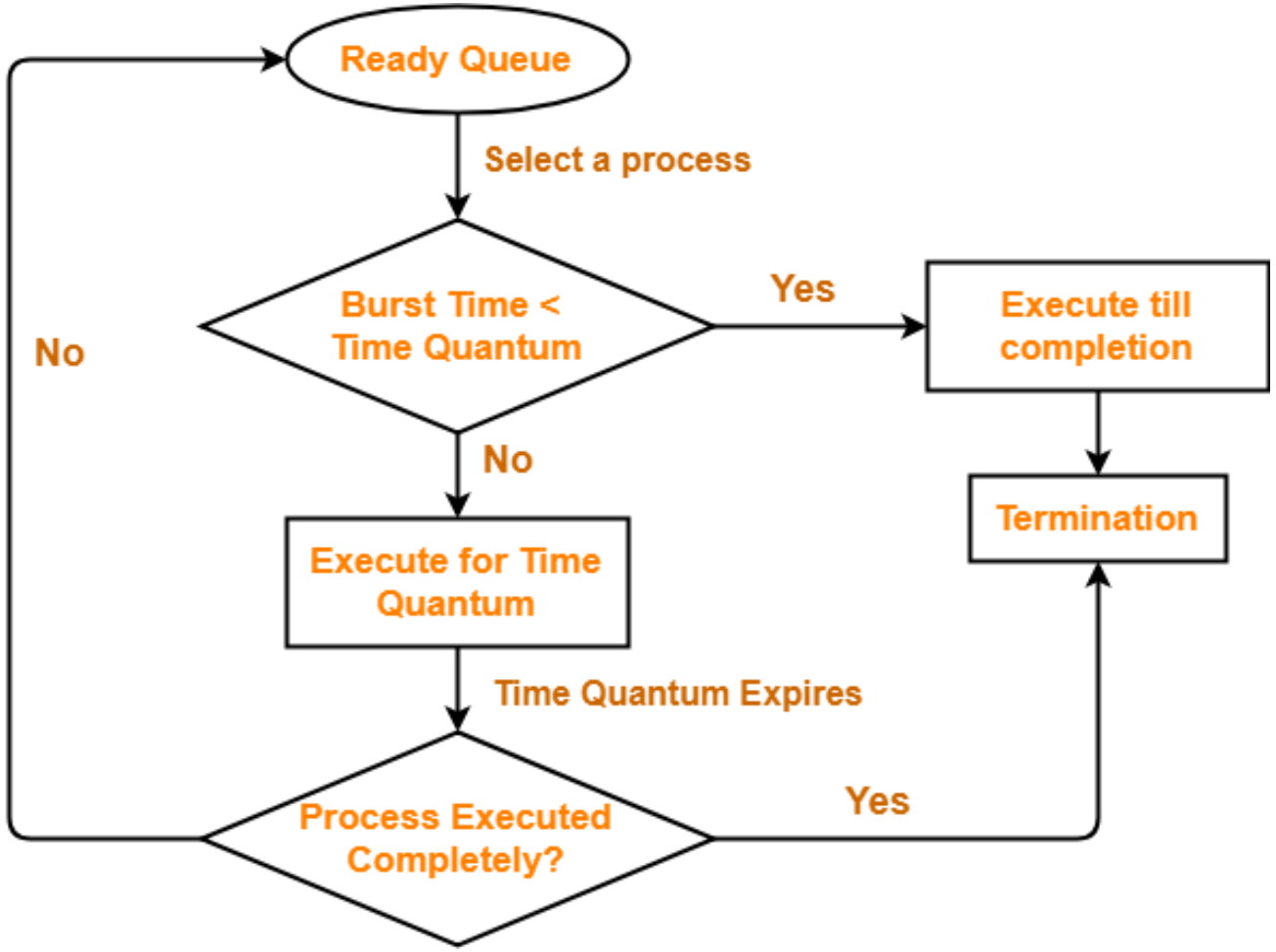
Average waiting time = (0 + 1 + 5 + 9 + 5) / 5 = 20 / 5 = 4 units

Round Robin Scheduling Algorithm

In Round Robin Scheduling,

- CPU is assigned to the process based on FCFS for a fixed amount of time.
- This fixed amount of time is called a **time quantum** or **time slice**.
- After the time quantum expires, the running process is preempted and sent to the ready queue.
- Then, the processor is assigned to the next arrived process.
- It is always preemptive in nature.
- Round Robin Scheduling is FCFS Scheduling with pre-emptive mode.

Round Robin Scheduling Algorithm



Round Robin Scheduling

Round Robin Scheduling Algorithm

- **Advantages-**

➢ It gives the best performance in terms of average response time.

➢ It is best suited for the time-sharing system, client-server architecture, and interactive system.

- **Disadvantages-**

➢ It leads to starvation for processes with larger burst times as they have to repeat the cycle many times.

➢ Its performance heavily depends on time quantum.

➢ Priorities cannot be set for the processes.

Round Robin Scheduling Algorithm

- **Note-01:**
-
- *With the decreasing value of time quantum,*

➢ *The number of context switches increases*

➢ *Response time decreases*

➢ *Chances of starvation decreases*

- *Thus, a smaller value of time quantum is better in terms of response time.*

Round Robin Scheduling Algorithm

Note-02:

With the increasing value of time quantum,

- *The number of context switches decreases*
- *Response time increases*
- *Chances of starvation increases*

Thus, a higher value of time quantum is better in terms of several context switches.

Round Robin Scheduling Algorithm

- **Note-03:**

➢ With the increasing value of time quantum, Round Robin Scheduling tends to become FCFS Scheduling.

➢ When the time quantum tends to infinity, Round Robin Scheduling becomes FCFS Scheduling.

- **Note-04:**

➢ The performance of round-robin scheduling heavily depends on the value of time quantum.

➢ The value of time quantum should be such that it is neither too big nor too small.

Round Robin Scheduling Algorithm

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

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

If the CPU scheduling policy is Round Robin with time quantum = 2 units, calculate the average waiting time and average turnaround time.

Round Robin Scheduling Algorithm

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

Process Id	Arrival time	Burst time
P1	0	4
P2	1	5
P3	2	2
P4	3	1
P5	4	6
P6	6	3

If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turnaround time.

Round Robin Scheduling Algorithm

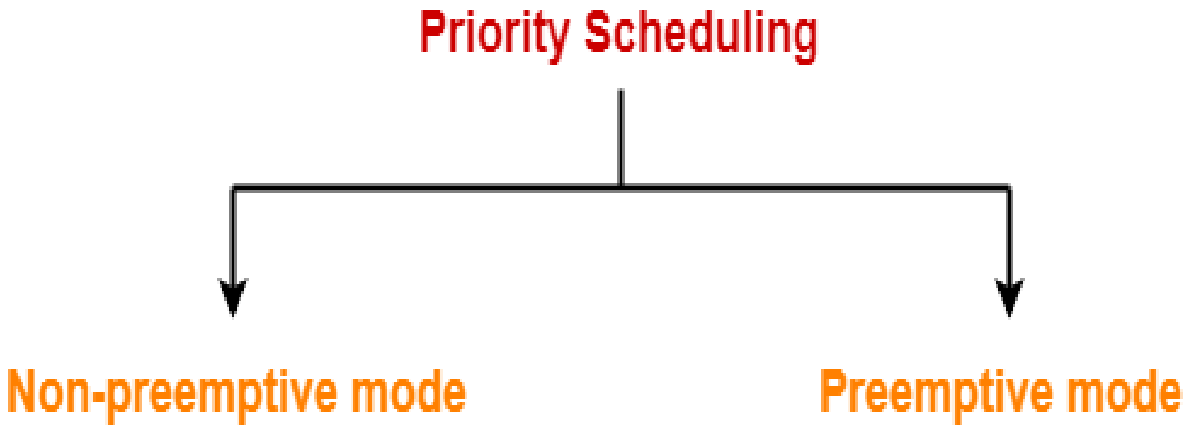
Example: Consider the set of 5 processes whose arrival time and burst time are given below

Process Id	Arrival time	Burst time
P1	5	5
P2	4	6
P3	3	7
P4	1	9
P5	2	2
P6	6	3

If the CPU scheduling policy is Round Robin with time quantum = 3 unit, calculate the average waiting time and average turnaround time.

Priority Scheduling

- In Priority Scheduling,
  - Out of all the available processes, the CPU is assigned to the process having the highest priority.
  - In case of a tie, it is broken by **FCFS Scheduling**.
- Priority Scheduling can be used in both preemptive and non-preemptive modes.



Priority Scheduling

Advantages-

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

Disadvantages-

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

Priority Scheduling

- **Note-01:**

➢ The waiting time for the process having the highest priority will always be zero in preemptive mode.

➢ The waiting time for the process having the highest priority may not be zero in non-preemptive mode.

- **Note-02:**

- Priority scheduling in preemptive and non-preemptive mode behaves exactly the same under the following conditions-

➢ The arrival time of all the processes is the same

➢ All the processes become available

Priority Scheduling

Example: 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 non-preemptive, calculate the average waiting time and average turnaround time. (Higher number represents higher priority)

Solution:

Gantt Chart

Process Id	Exit time	Turn Around time	Waiting time
P1	4	4 – 0 = 4	4 – 4 = 0
P2	15	15 – 1 = 14	14 – 3 = 11
P3	12	12 – 2 = 10	10 – 1 = 9
P4	9	9 – 3 = 6	6 – 5 = 1
P5	11	11 – 4 = 7	7 – 2 = 5

- Average Turn Around time = (4 + 14 + 10 + 6 + 7) / 5 = 41 / 5 = 8.2 unit
- Average waiting time = (0 + 11 + 9 + 1 + 5) / 5 = 26 / 5 = 5.2 unit

Multilevel Queue Scheduling

- Each algorithm supports a different process, but in a general system, some processes require scheduling using a priority algorithm. While some processes want to stay in the system ***(interactive processes),*** others are ***background processes*** whose execution can be delayed.

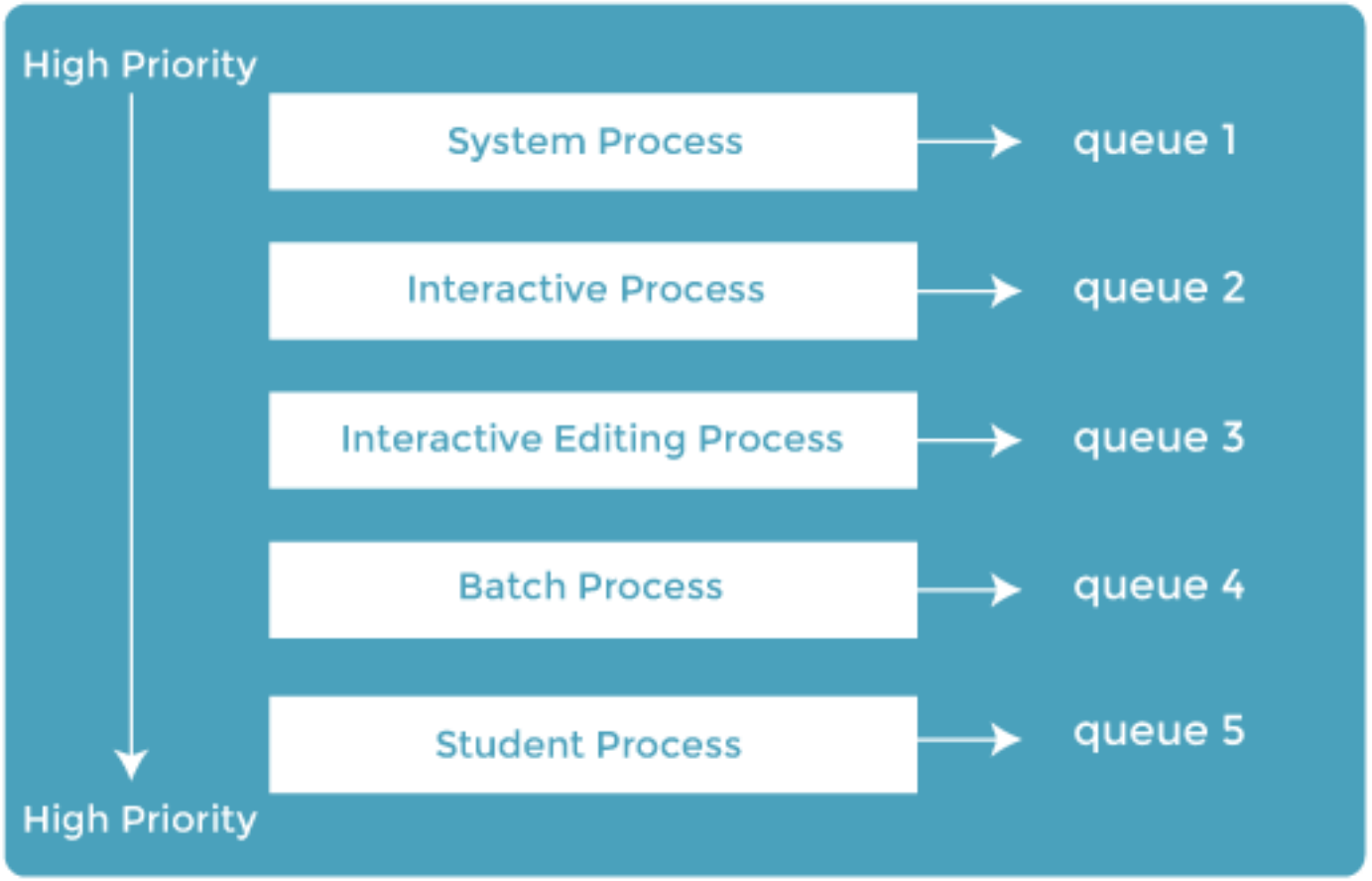
Multilevel Queue Scheduling

- Each queue has its own scheduling algorithm
  - foreground – RRs
  - background – 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

- The ready queue has been partitioned into seven different queues using the multilevel queue scheduling technique.
- These processes are assigned to one queue based on their priority, such as memory size, process priority, or type. The method for scheduling each queue is different.
- Some queues are utilized for the foreground process, while others are used for the background process.
- The ***foreground queue may*** be scheduled using a ***round-robin method,*** and the ***background queue can*** be scheduled using an ***FCFS*** strategy.

Multilevel Queue Scheduling



Multilevel Queue Scheduling

- **Advantages**

1. You can use multilevel queue scheduling to apply different scheduling methods to distinct processes.
2. It will have low overhead in terms of scheduling.

- **Disadvantages**

1. There is a risk of starvation for lower-priority processes.
2. It is rigid in nature.

Multilevel Queue Scheduling

Example: Consider the four processes listed in the table below under multilevel queue scheduling. The queue number denotes the process's queue.

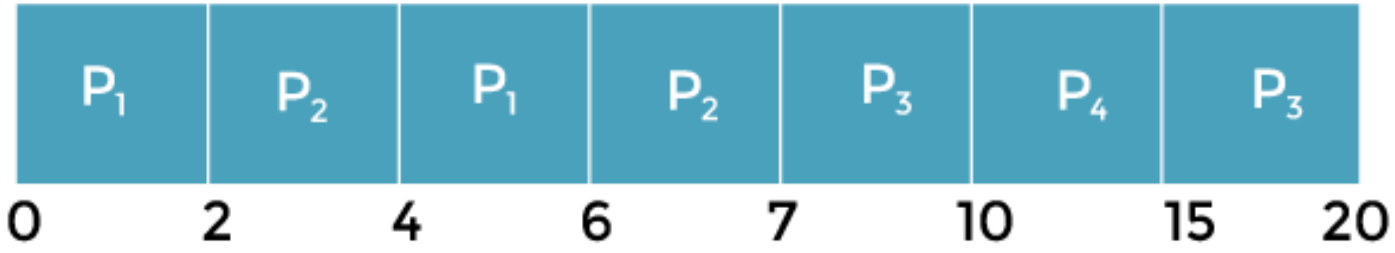
Process	Arrival Time	CPU Burst Time	Queue Number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

Queue 1 has a higher priority than queue **2.** Round Robin is used in queue **1 (Time Quantum = 2),** while FCFS is used in queue **2.**

Multilevel Queue Scheduling

Solution: Both queues have been processed at the start. Therefore, **queue 1 (P1, P2)** runs first (due to greater priority) in a round-robin way and finishes after 7 units.

1.The process in **queue 2 (Process P3)** starts running (since there is no process in queue 1), but while it is executing, P4 enters queue 1 and interrupts P3, and then P3 takes the CPU and finishes its execution.



Practice Set

Consider three processes, all arriving at time zero, with total execution time of 10, 20, and 30 units respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses the shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of the CPU remains idle?

Practice Set

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

Process No.	Arrival Time	Priority	Burst Time		
			CPU Burst	I/O Burst	CPU Burst
P1	0	2	1	5	3
P2	2	3	3	3	1
P3	3	1	2	3	1

- If the CPU scheduling policy is Priority Scheduling, calculate the average waiting time and average turnaround time. (Lower number means higher priority)

Thank
you