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

Operating Systems

School of Computer Engineering and technology

Module 2

Process Management

- **Process:** Concept of a Process, Process States, Process Control - creation, new program execution, termination. Interposes communication(IPC). Examples of IPC.

- **Threads:** Differences between Threads and Processes. Concept of Threads, Concurrency. Multi- threading, Types of Threads. POSIX Threads functions.

- **Scheduling:** Concept of Scheduler, Scheduling Algorithms: FCFS, SJF, SRTN, Priority, Round Robin.

References

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

Process Management

Concept of a Process :

- Process is an instance of a program in execution.
- It is an entity that can be assigned to and executed on a processor

A process is comprised of:

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

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

Process Management

- Terms ***job, task*** **and** ***process*** are used almost interchangeably.
- Many copies of editor program(passive entity) invoked, each is a separate process(active entity)

Process Management

What is process management ?

- Processes are represented and controlled by the OS, this is known as process management.
- **The Process states** which characterize the behaviour of processes.
- **The Data structures** that are used to manage processes.
- It describes the ways in which the OS uses these data structures to control process execution.

Process Management tasks of an OS

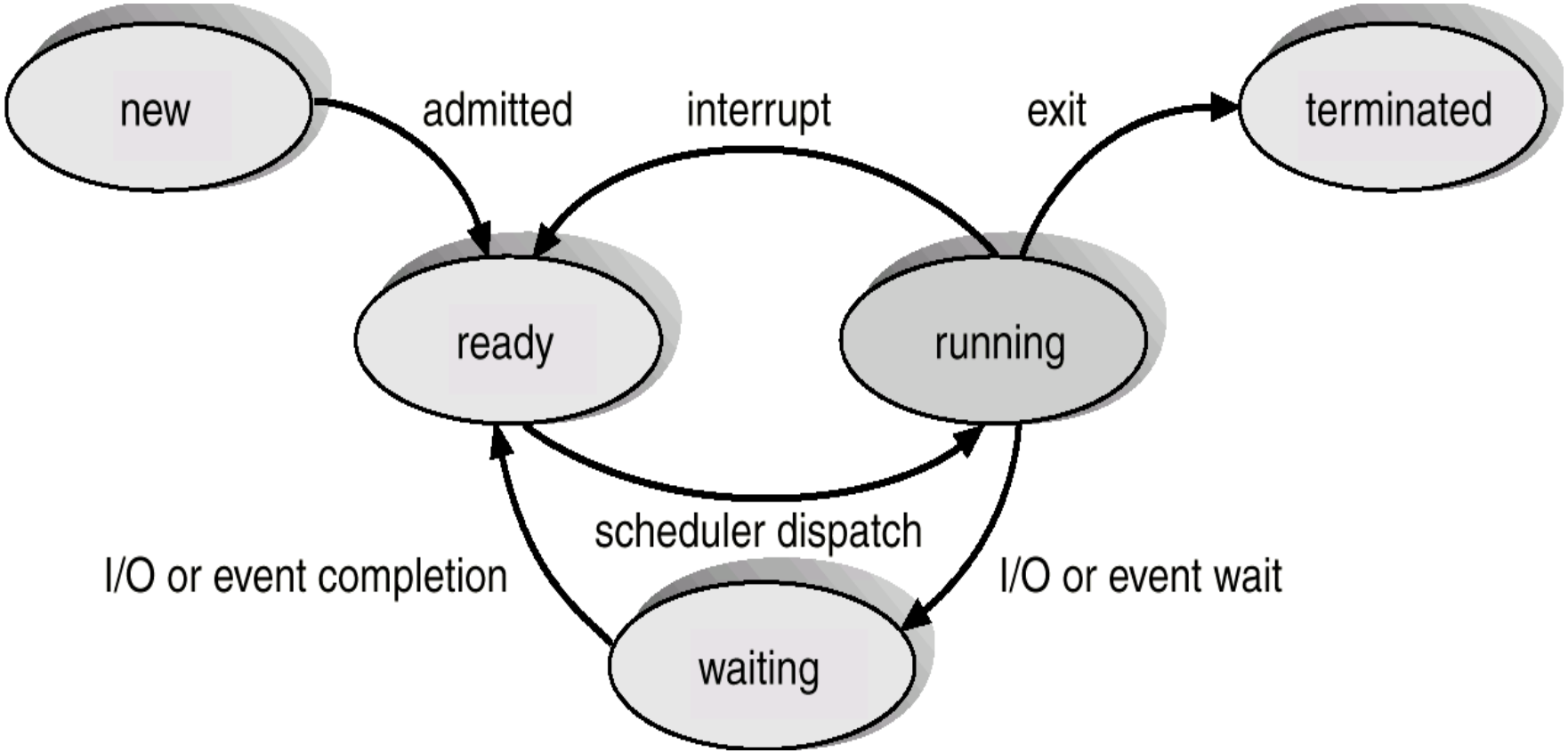
- Interleave the execution of multiple processes
- Allocate resources to processes, and protect the resources of each process from other processes,
- Enable processes to share and exchange information,
- Enable synchronization among processes.

Process States

When a program executes as a process, it goes through multiple states before it completes execution

- **New:** process is created
- **Ready:** process is waiting to be assigned processor. The process has every other resource except the processor
- **Running:** Instructions are being executed. State has all resources including the processor
- **Waiting:** process is waiting for some event to occur (eg. I/O completion). When an executing process needs an I/O device/services, it gets into wait state and when the i/o requirement is fulfilled it goes back into ready state
- **Terminated:** process has finished execution

Diagram for Process States



Suspended State

- Processor is faster than I/O so many processes could be waiting for I/O
- Swap these processes to disk to free up more memory
- Ready/Waiting state becomes ***suspend*** state when swapped to disk

Process Control Block (PCB)

Process Control Block [PCB] : It is a Data-structure maintained by the Operating System. It holds all necessary information related to a Process.

Information associated with each process is a follows:-

Process state

Program counter

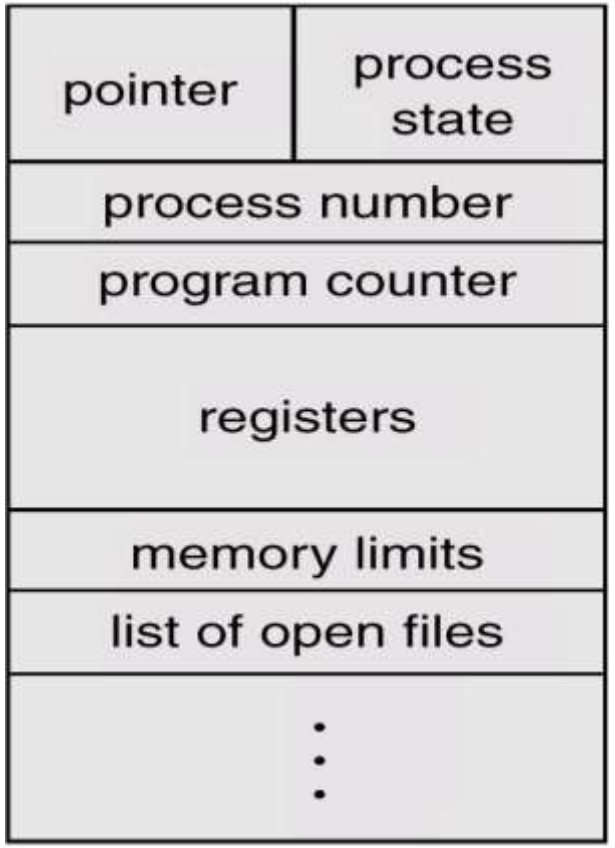
CPU registers

CPU scheduling information

Memory-management information

Accounting information

I/O status information



Process Modes

There are two modes of operations :-

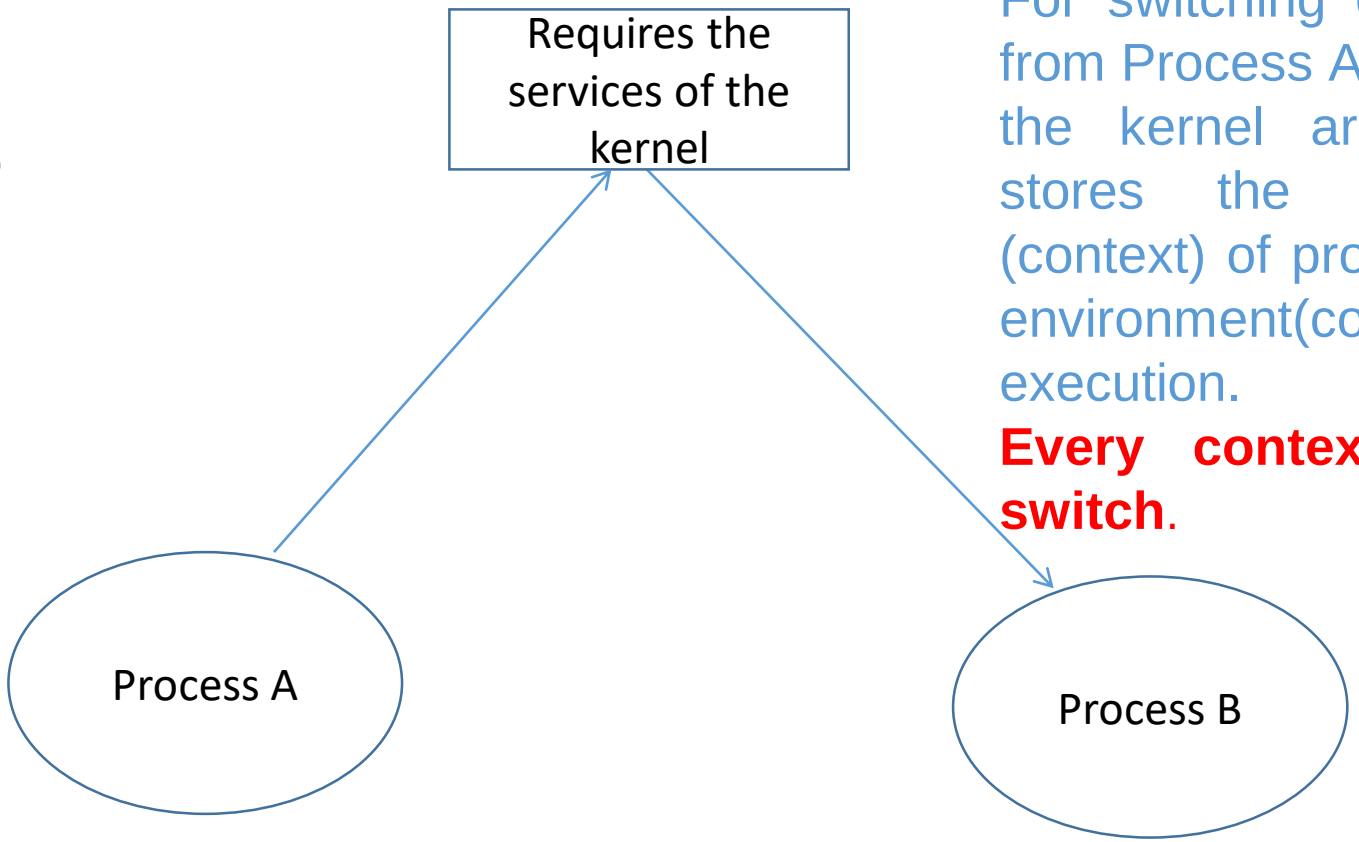
1. **Kernel mode** (Privileged mode): It can access its own data-structures as well as the user mode data structures.

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

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

Process Management --- context switch

- Kernel mode
- User mode



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

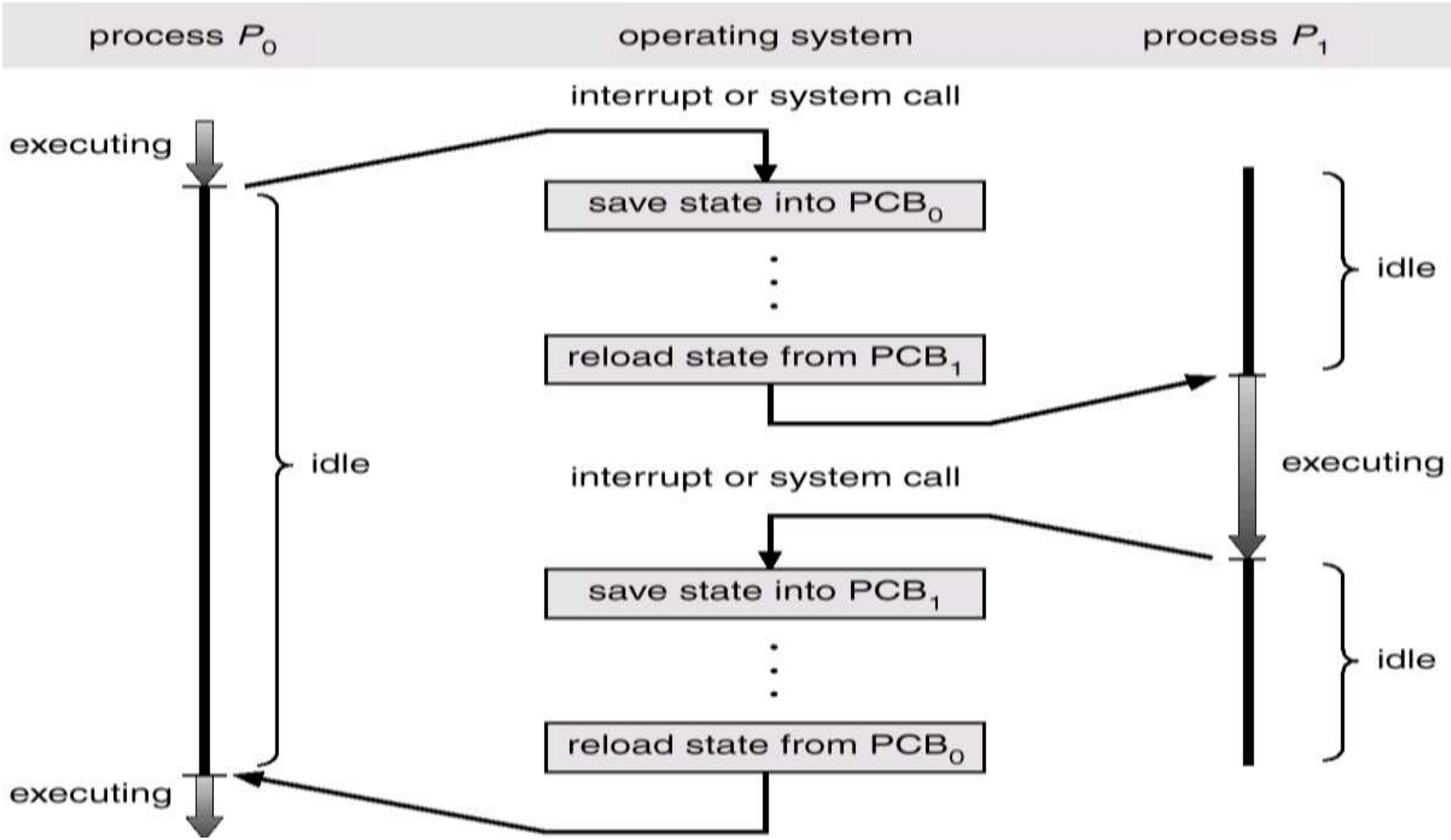
Every context switch is a mode switch.

Context Switch

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

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

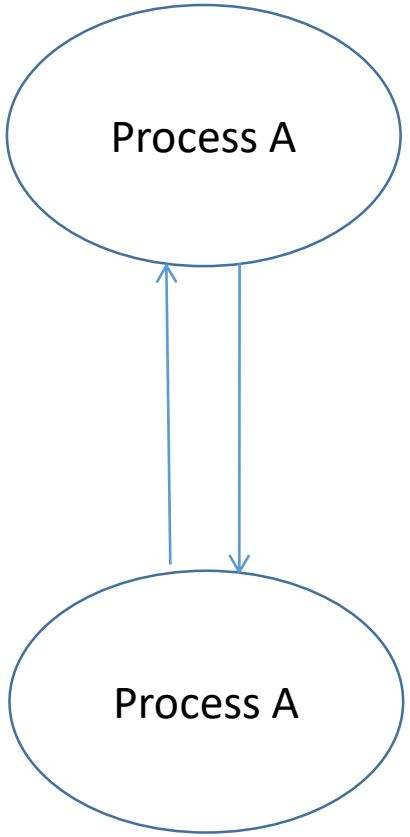
Context Switch



CPU Switch From Process to Process

Process Management – Mode switch

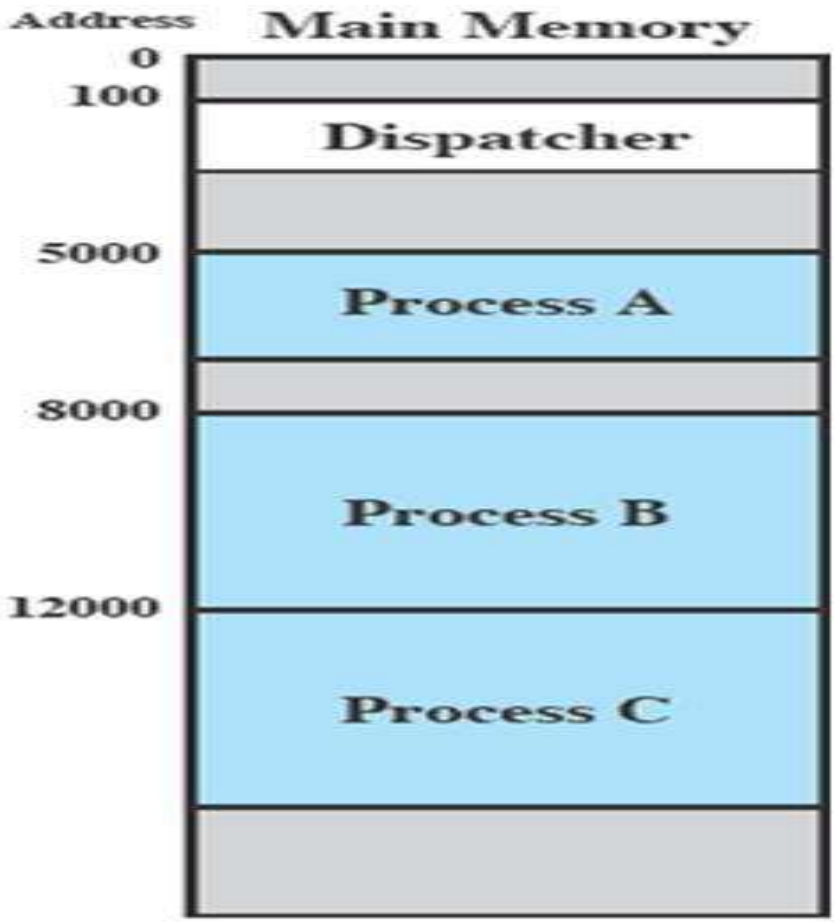
- Kernel mode
- User mode



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

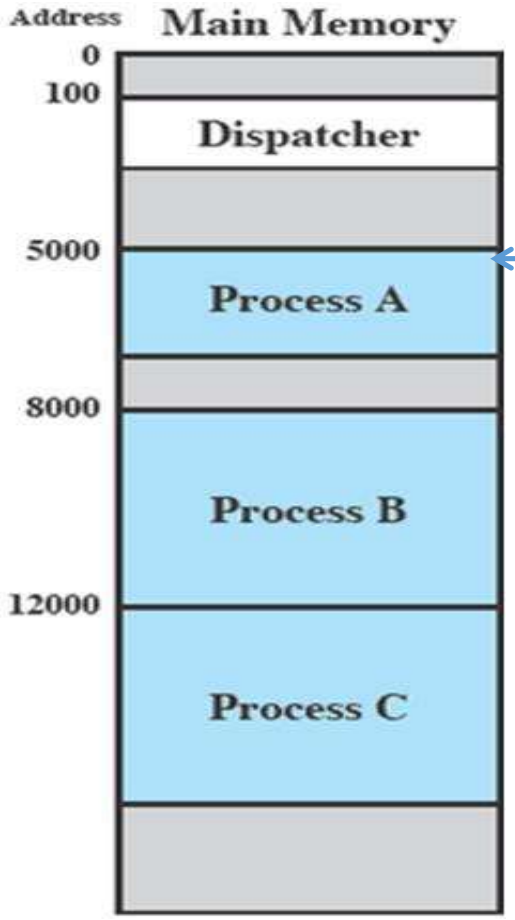
As, every context switch is a mode switch, every mode switch may/may not be a context switch.

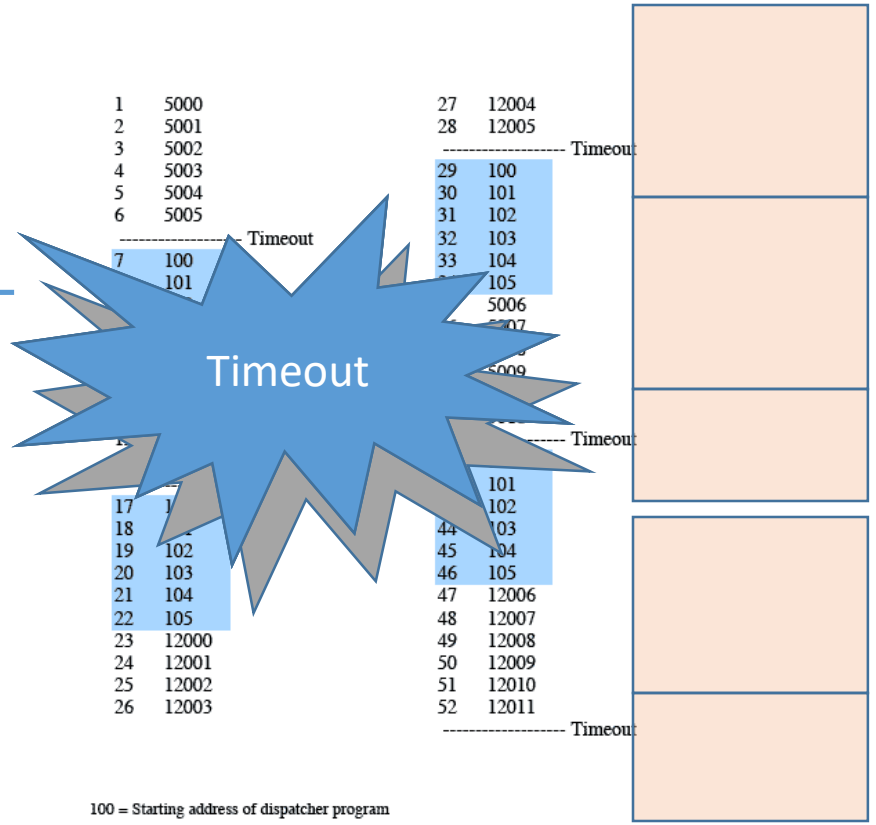
Process Execution



- Consider three processes being executed
- All are in memory (plus the dispatcher)
- ***Dispatcher*** is a small program which switches the processor from one process to another
- Selecting a process among various processes is done by **scheduler**.
- Here the task of scheduler completed.
- Now **dispatcher** comes into picture as scheduler have decide a process for execution, it is dispatcher who takes that process from ready queue to the running status, or providing CPU to that process is the task of dispatcher.

Process Execution



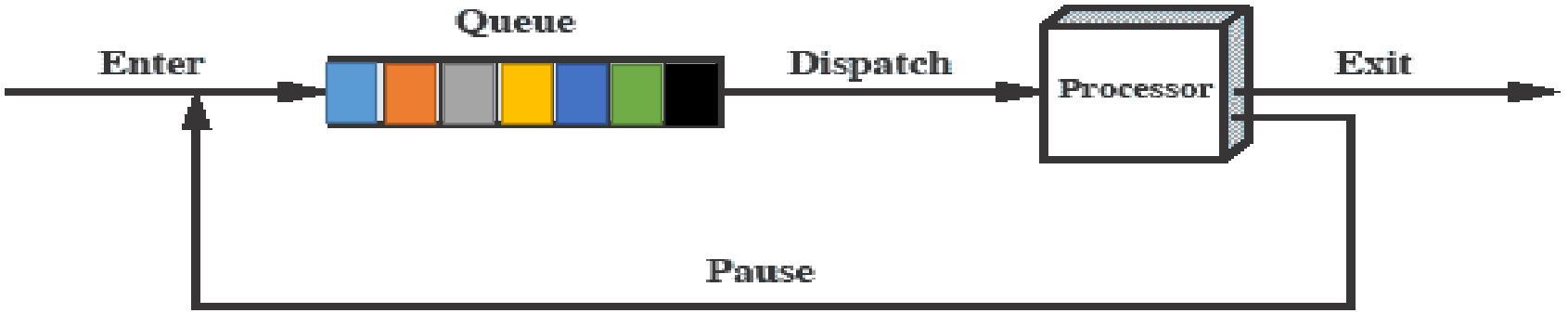


100 = Starting address of dispatcher program

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

Figure 3.4 Combined Trace of Processes of Figure 3.2

Queuing Diagram



(b) Queuing diagram

Processes moved by the dispatcher of the OS to the CPU then back to the queue until the task is competed

Process Creation

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

- Allocates space to the process in memory.
- Assign a unique process ID to the process
- A Process control block (PCB) gets associated with the process.
- OS maintains pointers to each process's PCB in a process table so that it can access the PCB quickly.

After Creation

After creating the process, the Kernel can do one of the following:

- Stay in the parent process.
- Transfer control to the child process
- Transfer control to another process.

Process Creation

- **Reasons to create a new process**
  - New user Job
  - Created by O/S to provide a service
  - Spawned by existing process: The action of creating a new process (Child Process) at the explicit request of another process (Parent Process) is called as process spawning. E.g. A print server or file server may generate a new process for each request that it handles

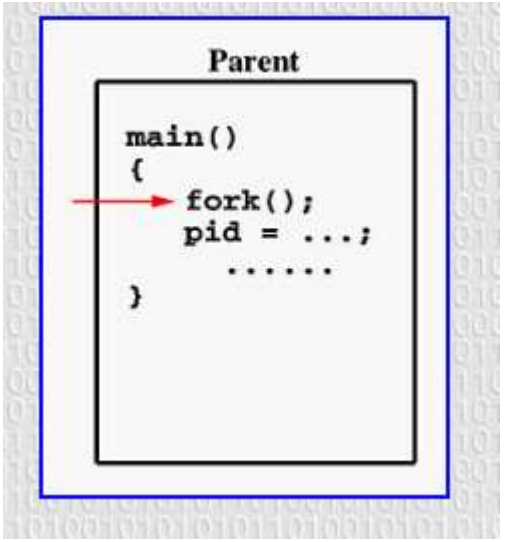
Process Creation

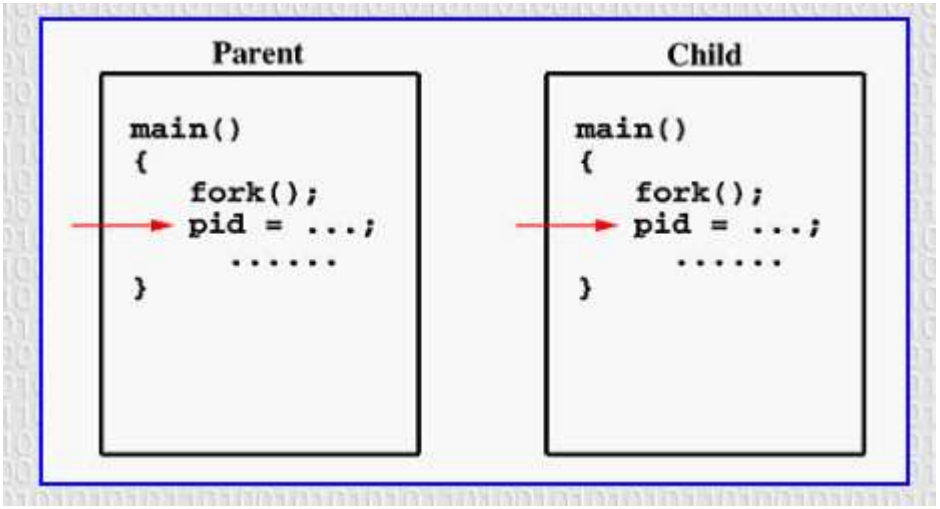
Fork

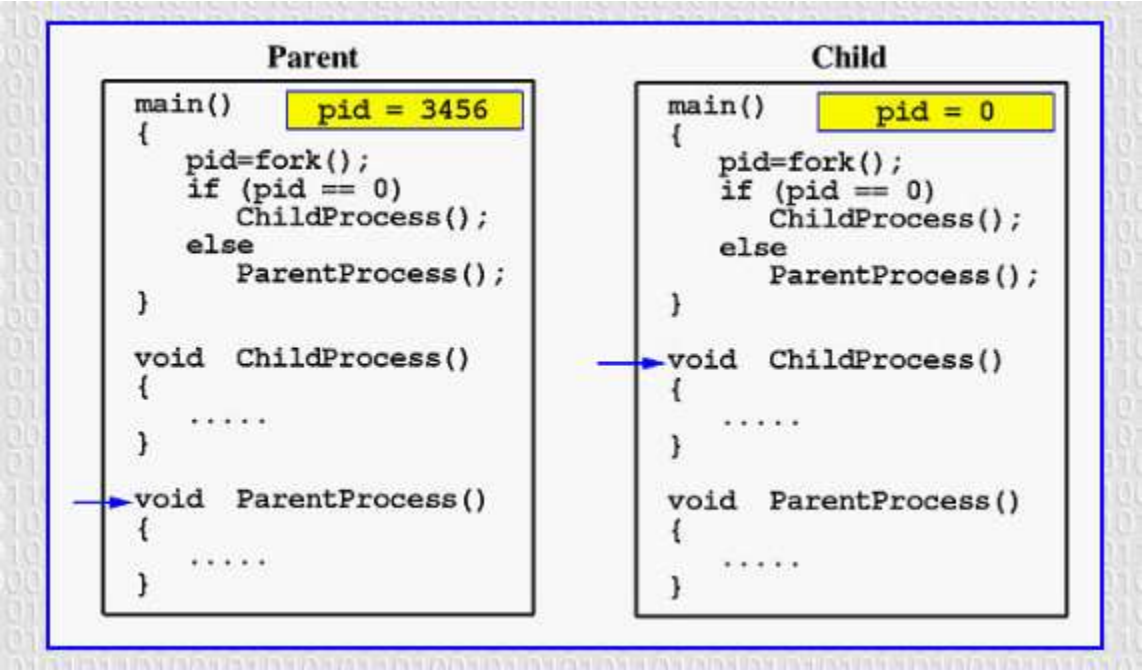
- **System call fork()** is used to create processes. It takes no arguments and returns a process ID.
- The syntax for the fork system call
  - pid = fork();
    - In the parent process, pid is the child process ID
    - In the child process, pid is 0
- Sequence of operations for fork.
  1. It allocates a slot in the process table for the new process
  2. It assigns a unique ID number to the child process
  3. It makes a copy of the context of the parent process.
  4. It returns the ID number of the child to the parent process, and a 0 value to the child process.

Fork

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


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


Parent
main()
{
fork();
pid = ...;
.....
}
Child
main()
{
fork();
pid = ...;
.....
}


Parent
Child
main()
pid = 3456
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}
main()
pid = 0
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}

Process Management—creating a Process in Unix(example)

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

/* pid_t : is a long integer type data type...prototype in types.h */
pid_t num_pid;

main()
{
num_pid=fork(); /* return value of fork */

if(num_pid==0) /* this is child process */
{
 printf("this is the child process id %d\n",getpid());

}

if(num_pid>0) /* this is parent process */
{
printf("this is the parent process id %d",getpid());

}
exit();
}
```

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

Process Creation – Parent/Child

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

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

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

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

fork()

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

fork()

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

Output:

Hello world!
Hello world!

fork()

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

fork()

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

Output:

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

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

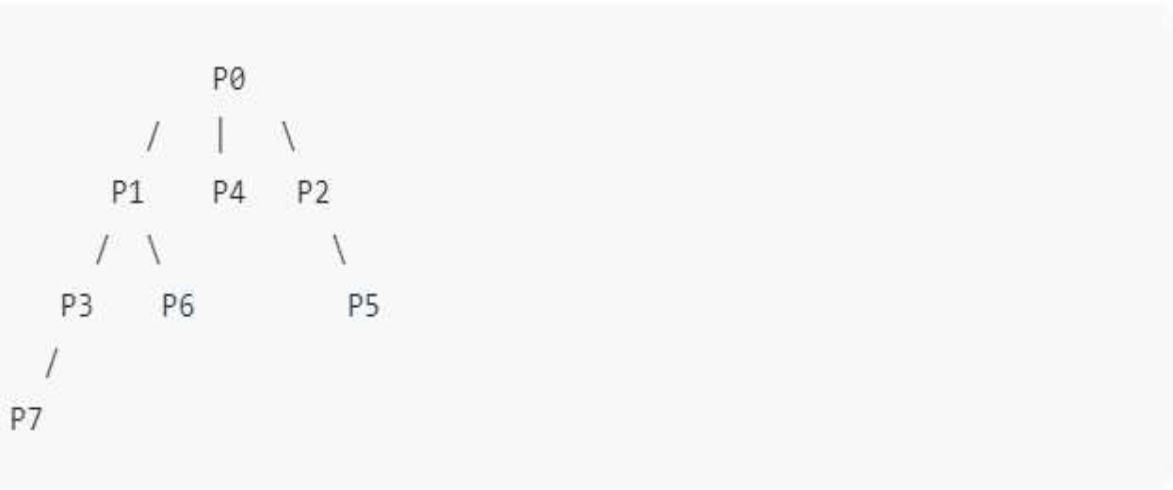
If we want to represent the relationship between the processes as a tree hierarchy it would be the following:

The main process: P0

Processes created by the 1st fork: P1

Processes created by the 2nd fork: P2, P3

Processes created by the 3rd fork: P4, P5, P6, P7



fork()

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

fork()

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

Hello from Child!
Hello from Parent!
(or)
Hello from Parent!
Hello from Child!
.
Here, two outputs are possible because the parent process and child process are running concurrently. So we don’t know whether the OS will first give control to the parent process or the child process.

fork()

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

fork()

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

Parent has x = 0
Child has x = 2
(or)
Child has x = 2
Parent has x = 0
Here, variable change in one process does not affect other process because data/state of two processes are different. And also parent and child run simultaneously, so two outputs are possible.

Process Termination

When a process terminates :

- All the resources held by process are released
- All the information held in all data structures is removed
- A process goes back to becoming a program and is stored on the secondary memory.

Interprocess Communication

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

- **Independent process**: A process is independent if it does not share data with any other processes executing in the system.

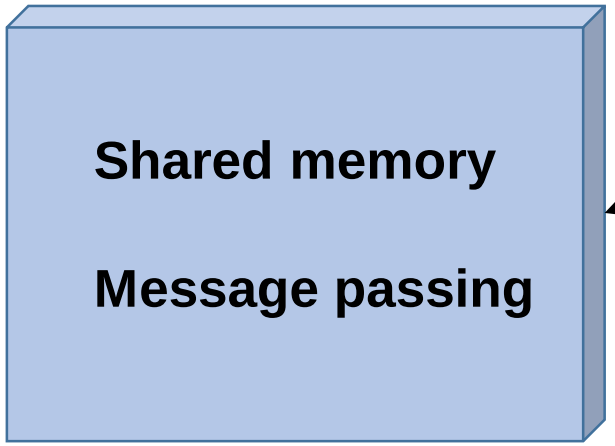
- **cooperating process**: A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process

Reasons for providing an environment that allows process cooperation:

- **Information sharing**: Since several applications may be interested in the same piece of information (for instance, copying and pasting), 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.
- **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads

IPC Models

- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data— that is, *send data to* and *receive data from* each other

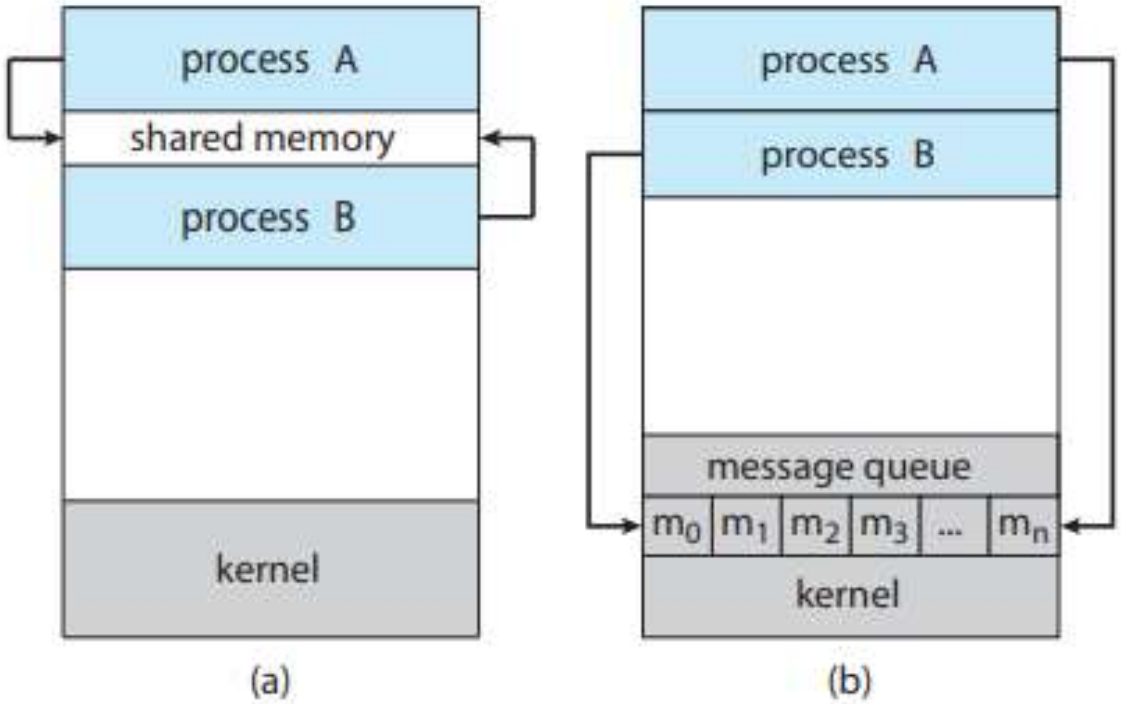


two fundamental models of interprocess communication

- In the **shared-memory model**, a region of memory that is shared by the cooperating processes is established.
- Processes can then exchange information by reading and writing data to the shared region.

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

- The two communications models are contrasted in Figure below:



Shared memory

Message passing

Messaging Passing Method

- **Important points:**

- 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.

Process Management-- Threads

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

Process Management-- Threads

- **Multithreading:** Ability of an OS to support multiple, concurrent paths of execution within a single process.

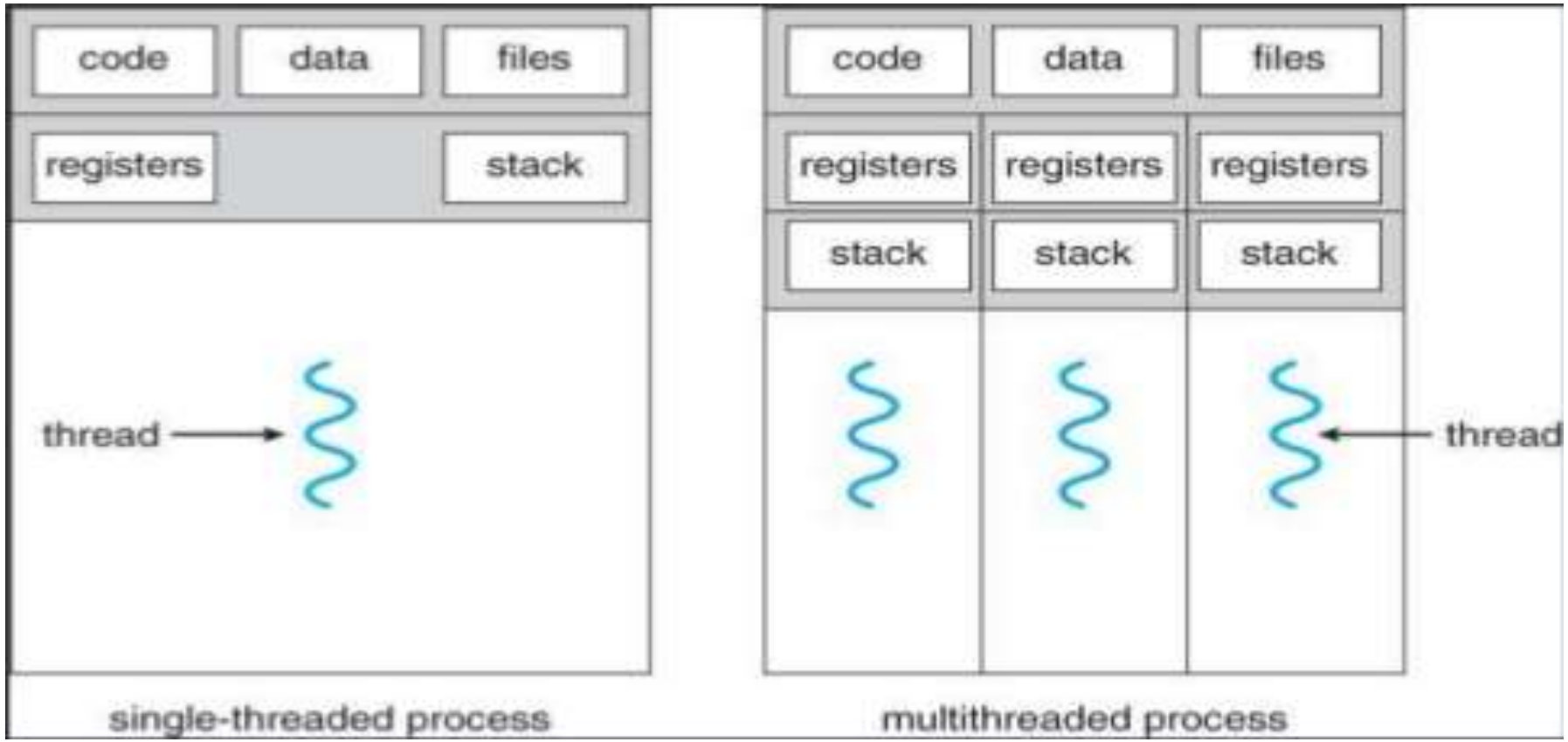
- It is also described as the interleaved execution of threads.

Process Management–Differences between threads and processes

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

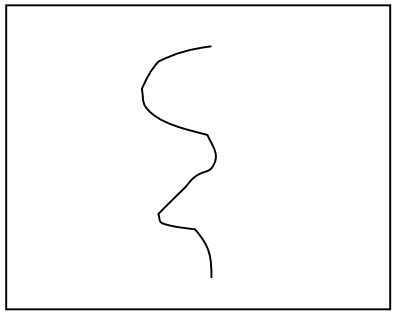
Process Management- ---

Threads, a diagrammatic representation

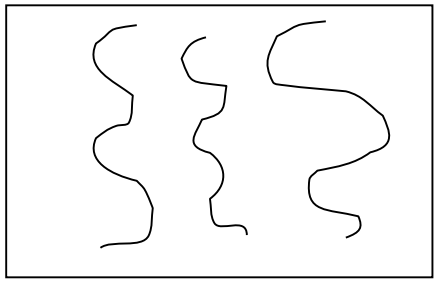


Process Management

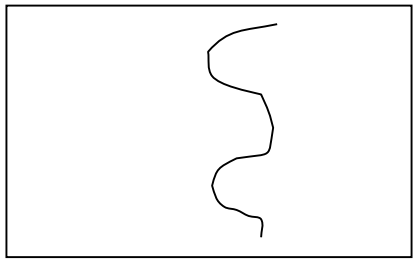
Threads and processes diagrammatic representation

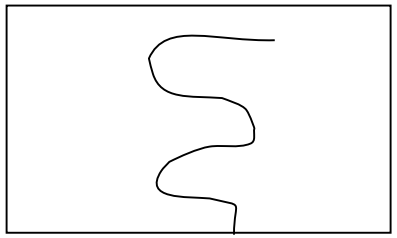


One process
one thread

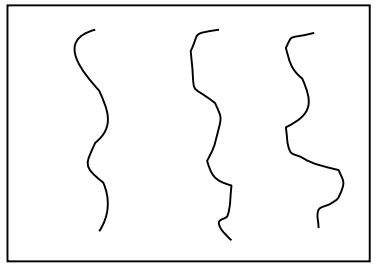


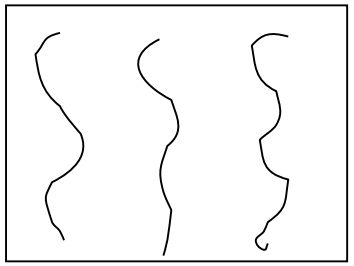
One Process
Multiple threads





Multiple Processes
one thread per process





Multiple Processes
Multiple thread per process

Multithreading

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

Process Management—Thread basics

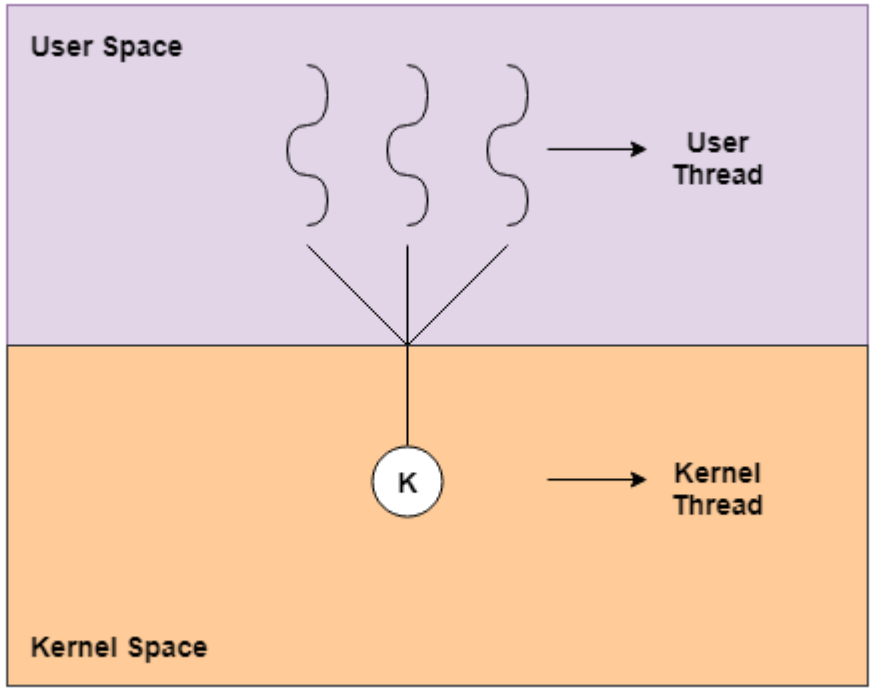
- Thread operations include thread creation, termination, synchronization (join, blocking), scheduling, etc.
- All threads within a process share the same address space.
- Threads in the same process share: Process instructions
  - Data
  - open files (descriptors)
  - signals
  - current working directory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

Process Management—Thread basics

- **Types of Threads** :-

There are majorly two types of threads

1. kernel level threads
2. User level threads



Process Management—Thread basics

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

Process Management—Thread basics

- **Advantages of User-Level Threads**

➢User-level threads are easier and faster to create than kernel-level threads. They can also be more easily managed.

➢User-level threads can be run on any operating system.

➢There are no kernel mode privileges required for thread switching in user-level threads.

- **Disadvantages of User-Level Threads**

➢Multithreaded applications in user-level threads cannot use multiprocessing to their advantage.

➢Entire process is blocked if one user-level thread performs blocking operation.

Process Management—Thread basics

- **Kernel-Level Threads**

➢Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel

➢Context information for the process as well as the process threads is all managed by the kernel.

➢Because of this, kernel-level threads are slower than user-level threads.

Process Management—Thread basics

- **Advantages of Kernel-Level Threads**

➢Multiple threads of the same process can be scheduled on different processors in kernel-level threads.

➢The kernel routines can also be multithreaded.

➢If a kernel-level thread is blocked, another thread of the same process can be scheduled by the kernel.

- **Disadvantages of Kernel-Level Threads**

➢A mode switch to kernel mode is required to transfer control from one thread to another in a process.

➢Kernel-level threads are slower to create as well as manage as compared to user-level threads.

Process Management—POSIX pthread

Portable Operating System Interface Standard (POSIX)

#include<pthread.h>

- int pthread_create (pthread_t *tid, const pthread_attr_t *attr, void *(*func)(void*),void *arg);

- 0 : OK +ve : error

- pthread_t *tid : Returns the thread ID which is of type pthread_t, i.e. long int.
- const pthread_attr_t *attr : Thread attribute list
- void *(*func)(void*) : A function that works as a thread.
- void *arg : A list of arguments sent to the function

Process Management—POSIX pthread

- #include <pthread.h>
- void pthread_exit(void *retval);

- The pthread_exit(); function terminates the calling thread and returns a value via retval (stores the return status of the thread terminated) that (if the thread is joinable) is available to another thread in the same process that calls pthread_join().

Process Management—POSIX pthread

- #include<pthread.h>
- int pthread_join(pthread_t tid, void **status);

- 0:OK +ve:error

- pthread_join() function shall suspend execution of the calling thread until the target thread terminates
- On return from a successful pthread_join() call with a non-NULL status argument, the value passed to pthread_exit() by the terminating thread shall be made available in the location referenced by status.

Process Management—POSIX pthread

- #include<pthread.h>
- pthread_t pthread_self(void);
- Returns: thread ID of calling thread

- The pthread_self() function returns the ID of the calling thread. This is the same value that is returned in *tid in the pthread_create() call that created this thread.

Process Management--POSIX pthread

- #include<pthread.h>
- int pthread_detach(pthread_t tid);
- Returns : 0:OK +ve: error
- The **pthread_detach**() function marks the thread identified by *thread* as detached.
- When a detached thread terminates, its resources are automatically released back to the
- system without the need for another thread to join with the terminated thread.

- **Need for linking –lpthread flag at the time of compilation :-**
- **-lpthread** in essence tells the GCC compiler that it must link the pthread library to the compiled executable. pthread or POSIX Threads is a standardized library for implementing threads in C

pthread example

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


int add1(int a[3])
{
        a[2] = a[1] + a[0];
    printf("result from thread 1 is %d\n",a[2]);
}
int add2(int b[3])
{
        b[2] = b[1] + b[0];
    printf("result from thread 2 is %d\n",b[2]);
}
```

```
main()
{       int arr[4],a[3],b[3],i,ans=0;
        pthread_t thread1,thread2;
        printf("enter 4  numbers\n");
        for(i=0;i<4;i++)
                scanf("%d",&arr[i]);
        a[0]=arr[0];        a[1]=arr[1];

        b[0]=arr[2];        b[1]=arr[3];

        pthread_create(&thread1,NULL,(void*)add1,a);
        pthread_create(&thread2,NULL,(void*)add2,b);
        pthread_join(thread1,NULL);
        pthread_join(thread2,NULL);

        ans=a[2]+b[2];
        printf("the result = %d\n",ans);
}
```

pthread example

```
#include <stdio.h>
#include <pthread.h>
int g = 0;
void myThreadFun(void *vargp)
{       int *myid = (int *)vargp;
        static int s = 0;
        ++s; ++g;
printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
}
```

```
int main()
{       int i;     pthread_t tid[3];
        for (i = 0; i < 3; i++)
                pthread_create(&tid[i], NULL, (void *) myThreadFun,  &tid[i]);
        pthread_exit(NULL);
        return 0;  }
```

pthread example

```
#include <stdio.h>
#include <pthread.h>
int g = 0;
void myThreadFun(void *vargp)
{       int *myid = (int *)vargp;
        static int s = 0;
        ++s; ++g;
printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
}
```

```
int main()
{       int i;      pthread_t tid[3];
        for (i = 0; i < 3; i++)
                pthread_create(&tid[i], NULL, (void *) myThreadFun,  &tid[i]);
        pthread_exit(NULL);
        return 0;  }
```

Global and static variables are shared by all threads.

```
Thread ID: 9945424, Static: 2, Global: 2
Thread ID: 18338128, Static: 4, Global: 4
Thread ID: 26730832, Static: 6, Global: 6
```