Operating System

Unit – 1
(Part-C)
Introduction

Mayank Mishra
School of Electronics Engineering
KIIT-Deemed to be University

System Call

- We are familiar with how the user work in user mode and kernel mode.
- If the user is using any application or API or writing any program, then we generally write in user mode. But if we want to access any functionality of operating system then we have to go to the Kernel mode.
- For going to Kernel mode from user mode, we need **System Call** (as our access is on the user mode).
- **System Call** is a programmatic way through which we shift from user mode to kernel mode.

- A **System Call** is a way for a program to request services from the operating system (OS) that the program itself cannot perform.

- A number of services are requested by the program, and the OS responds by launching a number of systems calls to fulfill the request.

- A system call can be written in high-level languages like C or Pascal or in assembly language. If a high-level language is used, the operating system may directly invoke system calls, which are predefined functions.

- A system call is initiated by the program executing a specific instruction, which triggers a switch to kernel mode, allowing the program to request a service from the OS. The OS then handles the request, performs the necessary operations, and returns the result back to the program.

Example :

- Let us assume we are writing a ‘C’ program and we want to print the final outcome of “78*7” on the monitor
- Here “78*7” is up to user mode, processor will do the calculation “78*7”. But our final objective is to get the value printed on the monitor or I have to access the printer. So, whenever I have to access devices, I have to take the help and it can be done in Kernel.

- Considering the Operating System such as Linux, if we write ‘C’ program in the text editor then we can directly use system call there itself.

- Number of System Call varies with the different Operating Systems.

- In some Operating Systems, we don’t write System Call directly, we rather use API or local libraries such as ‘printf’ or ‘scanf’ (Here, ‘printf’ or ‘scanf’ is not a System call, they are functions)

- **System Call** is a programmatic way through which we shift from user mode to kernel mode.

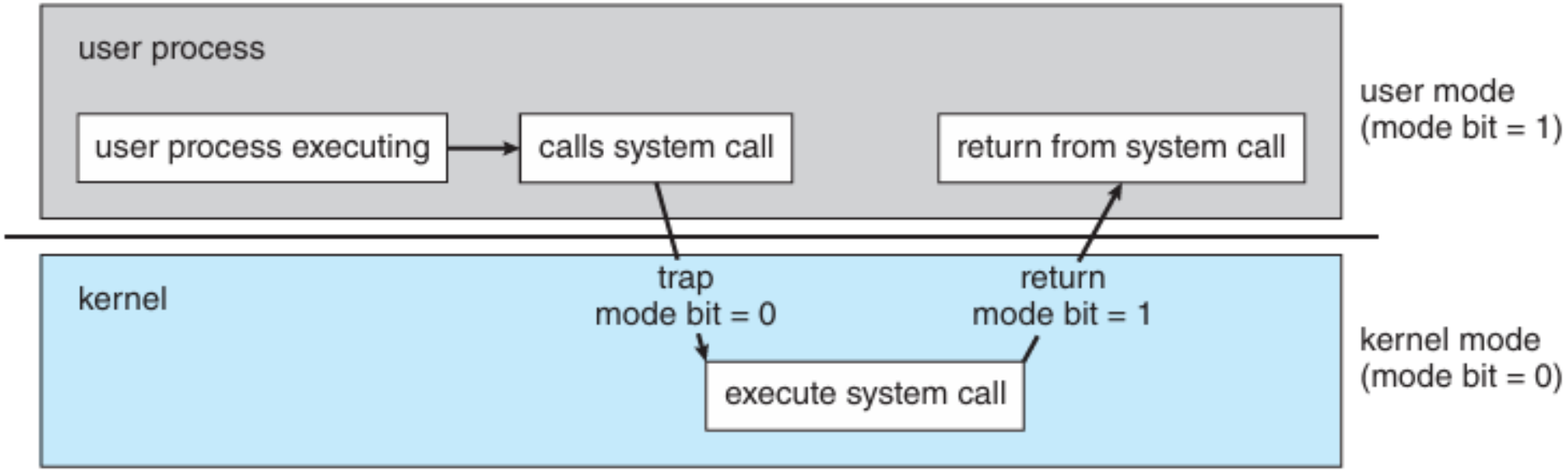


Fig. Transition from user to kernel mode.

- When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.

- At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0).

Types of System Call

- System calls can be grouped roughly into six major categories:

  - **Process Control**
  - **File Management**
  - **Device Management**
  - **Information Maintenance**
  - **Communications**
  - **Protection**

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices
- Protection
  - get file permissions
  - set file permissions

Examples of Windows and Unix System Calls:

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

- **open() :** An open() call allows a process to start accessing the file stored in the file system.
- **read() :** It is used to obtain data from a file on the file system.
- **write() :**It is used to write data from a user buffer to a device like a file. This system call is one way for a program to generate data.
- **close() :** It is used to end file system access. When this system call is invoked, it signifies that the program no longer requires the file, and the buffers are flushed, the file information is altered, and the file resources are de-allocated as a result.
- **fork() :** processes generate clones of themselves using the fork() system call. It is one of the most common ways to create processes in operating systems.

fork ( )

- fork() system call generally returns a value: **0** (child process), **1** (parent process), **-1** (Let's say OS and kernel is busy so child process has been not created.

- **Child process** will run parallelly with the **Parent Process**.

- Creating new processes with the fork system call facilitates the running of several tasks concurrently within an operating system. The system's efficiency and multitasking skills are improved by this concurrency.

- Code reuse: The child process inherits an exact duplicate of the parent process, including every code segment, when the fork system call is used. By using existing code, this feature encourages code reuse and streamlines the creation of complicated programmes.

- Process isolation is achieved by giving each process started by the fork system call its own memory area and set of resources. System stability and security are improved because of this isolation, which prevents processes from interfering with one another.

Elaboration of fork() implementation:

```c
#include <stdio.h>
int main()
{

    printf("Hello");
}
```

?????

```c
#include <stdio.h>
int main()
{
  fork();
    printf("Hello \n");
}
```

?????

```c
#include <stdio.h>
int main()
{
  fork();
  fork();
    printf("Hello \n");
}
```

?????

Elaboration of fork() implementation:

```c
#include <stdio.h>
int main()
{

    printf("Hello");
}
```

Output:

Hello

```c
#include <stdio.h>
int main()
{
  fork();
    printf("Hello \n");
}
```

Output:

Hello
Hello

```c
#include <stdio.h>
int main()
{
  fork();
  fork();
    printf("Hello \n");
}
```

Output:

Hello
Hello
Hello
Hello

Question: How many times “Hello World!” will be printed if we execute the below C program?

```c
#include <stdio.h>

int main()
{
  if (fork() && fork()){
  fork();

  }
    printf("Hello World! \n");

}
```

Question: How many times “Hello World!” will be printed if we execute the below C program?

```c
#include <stdio.h>

int main()
{
  if (fork() && fork()){
  fork();

  }
    printf("Hello World! \n");

}
```

Output:

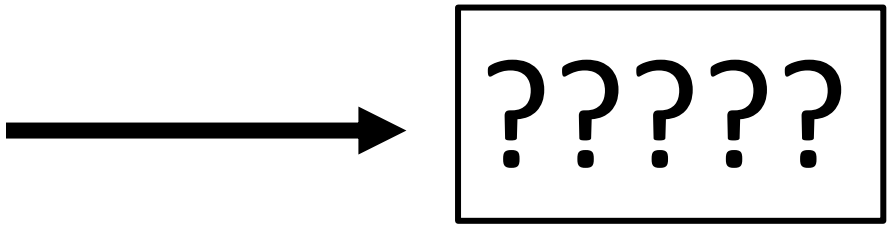
```
Hello World!
Hello World!
Hello World!
Hello World!
```

Answer: 4 Times

Home Work

Question: How many times “1” will be printed if we execute the below C program?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
  int a;
  for (a=1; a<5; a++)
        fork();
  printf("1");
}
```



Process Concept

- Informally, as mentioned earlier, a **process** is a program in execution.

- The status of the current activity of a process is represented by the value of the program counter and the contents of the processor’s registers.

- A **program counter** is a special register in a computer processor that contains the memory address (location) of the next program instruction to be executed. Before the CPU fetches an instruction from memory, it references the program counter for the correct memory address. After the CPU fetches the instruction, it increases the program counter by 1 so it points to the next instruction in the program's sequence.

- A **processor register (CPU register)** is one of a small set of data holding places that are part of the computer processor.

- The memory layout of a process is typically divided into multiple sections.

- These sections include:
  - **Text section:** the executable code *(it stores the instructions of the program)*
  - **Data section:** global variables *(we can use the global variable throughout the program)*
  - **Heap section:** memory that is dynamically allocated during program run
  - **Stack section:** temporary data storage when invoking functions (such as function parameters, return addresses, and local variables).

- Notice that the sizes of the text and data sections are fixed, as their sizes do not change during program run time.

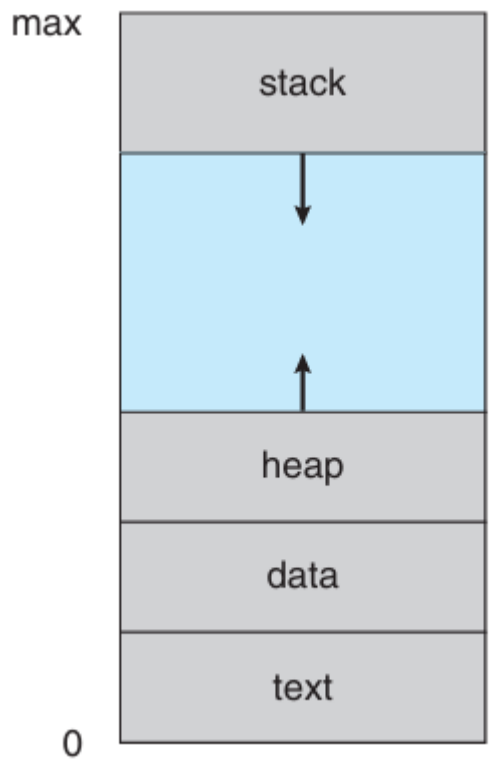


Fig. Layout of a process in memory

- However, the **stack** and **heap** sections can **shrink** and **grow** dynamically during program execution. Each time a function is called, an **activation record** containing function parameters, local variables, and there turn address is pushed onto the stack; when control is returned from the function, the activation record is popped from the stack.

- Similarly, the **heap** will grow as memory is dynamically allocated, and will shrink when memory is returned to the system. Although the **stack** and **heap** sections **grow** toward one another, the operating system must ensure they do not **overlap** one another.

- We emphasize that a program by itself is not a process. A program is a **passive entity**, such as a file containing a list of instructions stored on disk (often called an **executable file**).

An executable file (EXE file) is a computer file that contains an encoded sequence of instructions that the system can execute directly when the user clicks the file icon.

- In contrast, a process is an **active entity**, with a program counter specifying the next instruction to execute and a set of associated resources.

- A program becomes a process when an executable file is loaded into memory.

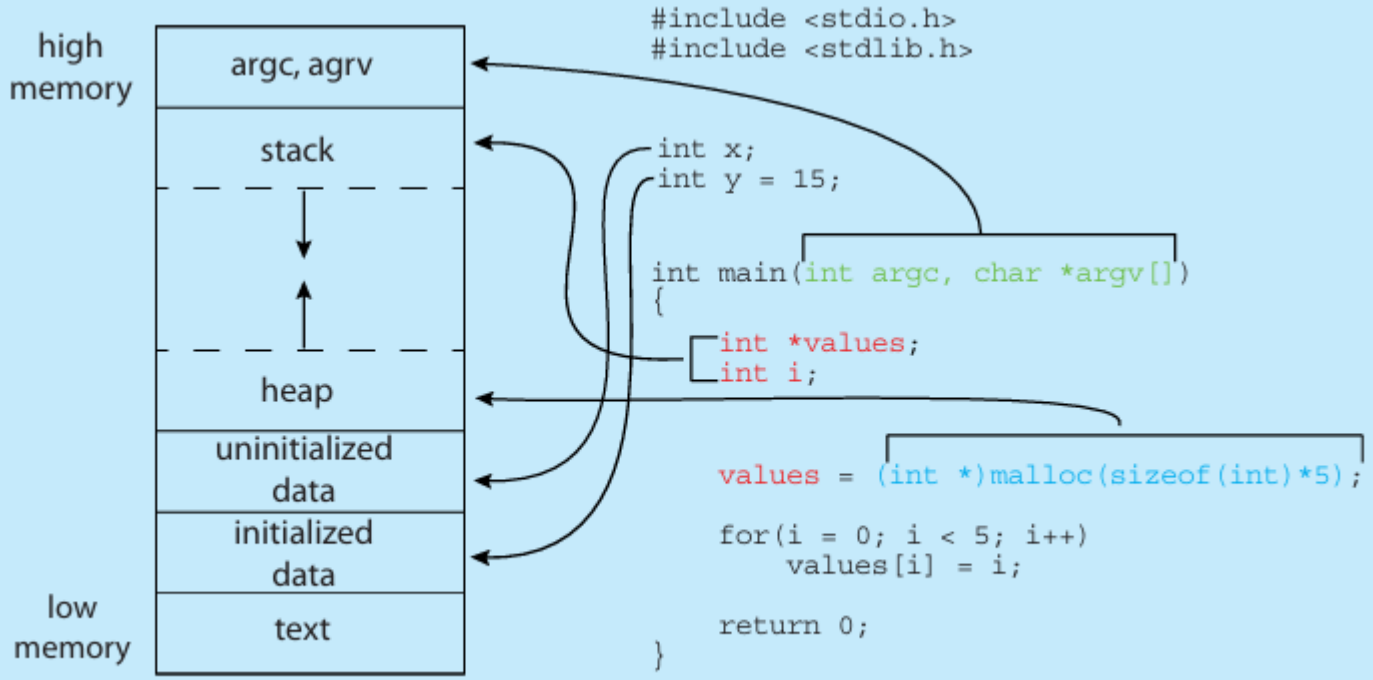
- Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line..

(Some basic information)

MEMORY LAYOUT OF A C PROGRAM

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure [redacted], with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



Process State

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

- The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:
  - **New:** The process is being created.
  - **Running:** Instructions are being executed.
  - **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
  - **Ready :** The process is waiting to be assigned to a processor.
  - **Terminated:** The process has finished execution.

- A **program counter** is a special register in a computer processor that contains the memory address (location) of the next program instruction to be executed. Before the CPU fetches an instruction

The Five-State Model (5T)

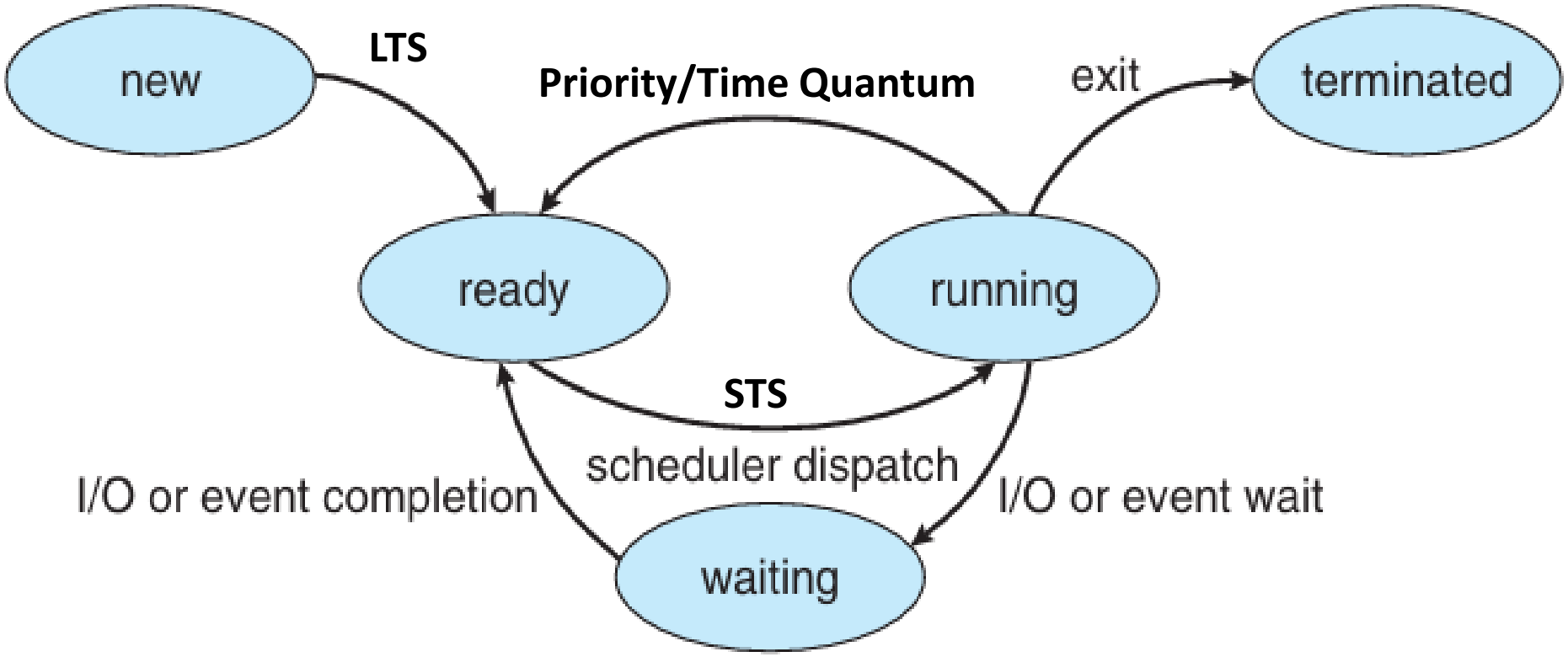
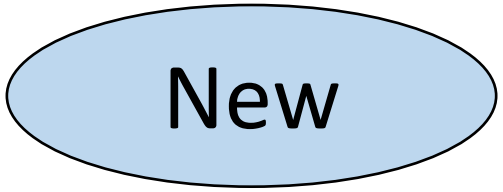


Fig. The Five-State Model



- This state tells that process has been created.

- ***What does "process created" means?***
  - Suppose we have written C program and stored in a secondary memory. (we have not run the program)
  - When we turned on an application, or when we turned on Mozilla Firefox, Chrome in our laptop.
  - Even when we turn on the laptop, by default when Operating system gets mounted in the memory, then lot of process starts running at the background, it is a new state.

- It implies, there was no process before that, now we have created a new process.

Ready

- Once the process has been created, we just poked it (turned ON), now it came into active state.
- Active State means "**ready state**" (also known as **ready queue)**
- **Ready Queue** means the process has come into the RAM (all these processes are coming in queue).
- There could be many processes in the secondary memory, we can't bring all processes in the Ready State. ***Who brings these processes?***
- *It's a **Long-Term Scheduler (LTS)** who brings all the processes. The responsibility of LTS here is to bring more and more processes in the ready state (which we refer it as the concept of multiprogramming)*

Running

- Our task is not filling the ready queue rather we have to go for execution of those processes.
- Every process wants to get CPU for sometime so it can get terminated after the execution.
- Processes will be dispatched from ready queue to Running state. ***How many processes will be brought to the running state? -This depends on how many CPU is being used.***
- If we have uniprocessor system then we have single CPU. For multiprocessor system, parallel processing will be required. (Here we are assuming the case of uniprocessor system)
- Out of all the process in the queue state, one process will be picked. That process will be scheduled and dispatched to the CPU for the execution.
- ***Where is that process staying now?*** *– Process is still in the RAM , but its status got changed (it got new address at a new place where CPU is executing it).*

terminated

- When CPU completed all the instructions means processor got terminated.

- ‘Terminated’ means ‘De-allocation’ (means when process came in **ready state** in the RAM, some address and resources has been provided. As the process has been completed those resources has to be taken back, as we have limited RAM).

- This is the general flow. Let us assume, ‘process, is going on in running state but suddenly a high priority process came in ready state.

- The process that was running before will sent back to ready queue. ***Why we sent back? – “Priority” (The concept of Multitaksing).***

- We have seen previously; multiprogramming means bringing as much process as possible. Out of those, one process will be given to the CPU. If CPU is completing the execution completely, then multitasking have not been done here.

- ***The other reason for sending back can be “time quantum”.*** There is a scheduling method called ‘round robin’ in which we use the concept of ‘time quantum’

- ***Which scheduler is helping here?- Short Term Scheduler (****whose responsibility is to pick a process from ready state and send it to the running state).*

Wait/block

- Suppose we gave a process to CPU and CPU is continuously executing the process.
- In between process said that it wants to give I/O request.
- That process will go to the **wait state.**
- When the I/O request will be completed, it will go to **ready state.**
- **Wait** is also in RAM, but status has changed.

The Seven-State Model (7T)

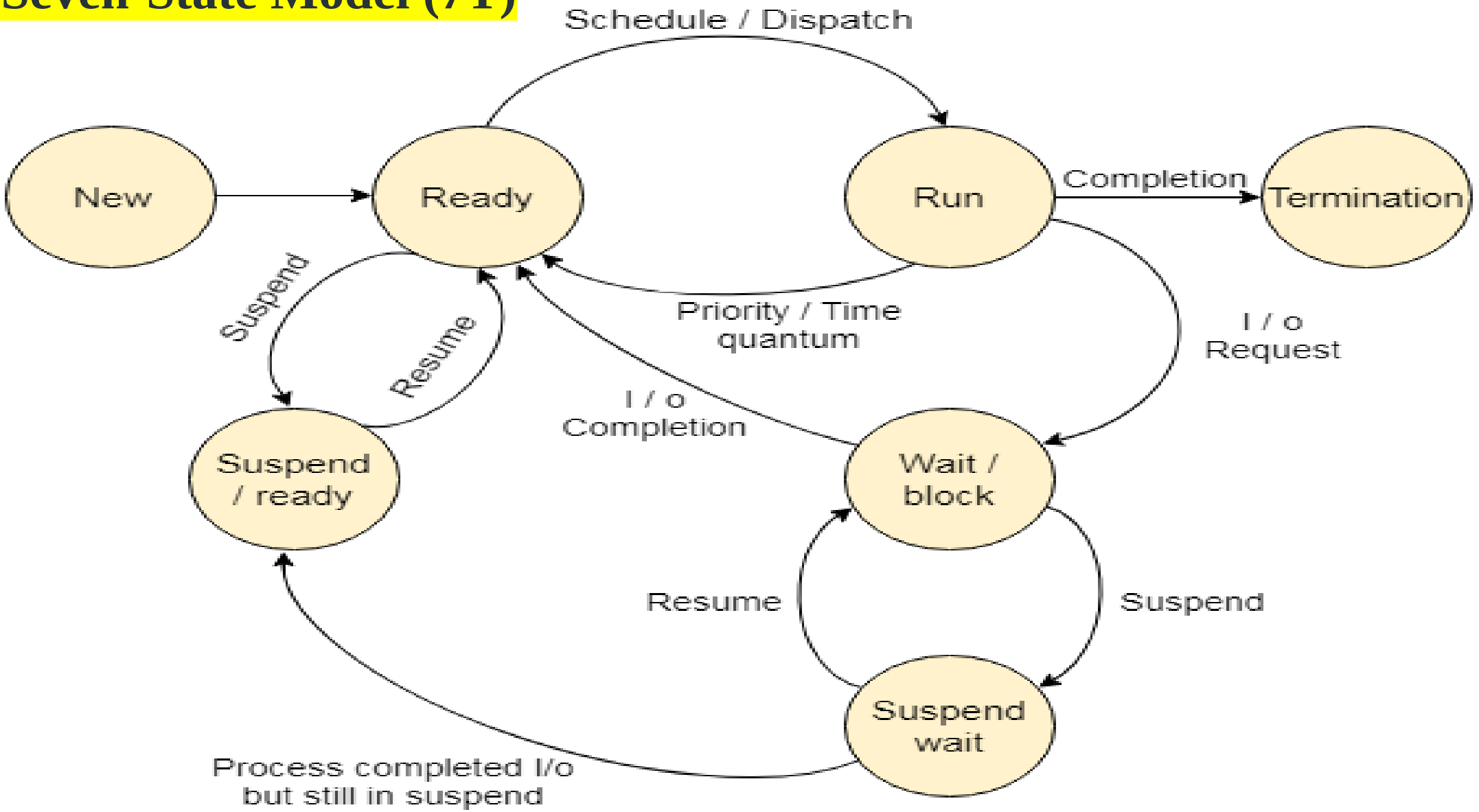


Fig. The Seven-State Model

- Other than the states discussed in the **Five States of the Process,** some additional state also exist.

- Let us assume 20 number of processes have been brought to the **Ready State**. Then the first process will be given to the CPU for the execution.

- Let us consider the scenario, this first process requires I/O request, then it will be transferred to **Wait/Block state**. Then, the second state will be sent to the **running state**. Assuming, second process also require I/O request, so it will be sent to the **Wait/Block state**.

- Let us assume all the process or majority of the process requires I/O request leading to the full occupancy of Wait/Block state (which is also a queue, a wait queue). Then for some time the process will be suspended and will be sent to the **Suspend Wait state.**

- Swapping out means sending back to the secondary memory (the remaining I/O request will be performed there itself). Once the memory will be free in **wait state**, the process from **Suspend Wait state** will be sent back to the **wait state**.

- ***Who handles all that?- Medium Term Scheduler (MTS)***

  If the RAM gets filled due to Ready State or other mentioned state, MTS swap out to Secondary Memory.

- **Considering the other possibility,** let us assume processes are getting filled in the **Ready State.** It could be possible that this queue also be filled. Assuming in n between if some **"high priority"** process arrives (system process or kernel process) which will lead to the transfer of the process from **Ready State** to the **Suspend Ready state.**

- Let us consider the situation when **“wait/block” state** doesn’t get empty, this implies that a greater number of processes are coming in the **“wait/block” state.**

- **What will be the next state?-** It will go for ‘backing store’ means the process from **“Suspend Wait”** will be sent to **“Suspend Ready”** if **“Ready State”** is not free, else it will go to **Ready State.**

Types of Schedulers

Long-Term Scheduler(LTS): Decides how many processes should be made to stay in the ready state. This decides the degree of multiprogramming. Once a decision is taken it lasts for a long time which also indicates that it runs infrequently. Hence it is called a long-term scheduler.

Short-Term Scheduler (STS): Short-term scheduler will decide which process is to be executed next and then it will call the dispatcher. A dispatcher is a software that moves the process from ready to run and vice versa. In other words, it is context switching. It runs frequently. Short-term scheduler is also called CPU scheduler.

Medium-Term Scheduler (MTS): Suspension decision is taken by the medium-term scheduler. The medium-term scheduler is used for swapping which is moving the process from main memory to secondary and vice versa. The swapping is done to reduce degree of multiprogramming.

Multiprogramming

We have many processes ready to run. There are two types of multiprogramming:

- **Preemption:** Process is forcefully removed from CPU. Pre-emption is also called time sharing or multitasking.

- **Non-Preemption:** Processes are not removed until they complete the execution. Once control is given to the CPU for a process execution, till the CPU releases the control by itself, control cannot be taken back forcibly from the CPU.

Process Control Block

- Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**.

- It contains many pieces of information associated with a specific process, including these:
  - **Process State:** The state may be new, ready, running, waiting, halted, and so on.
  - **Process Number:** Unique identification number given to the process.
  - **Program counter:** The counter indicates the address of the next instruction to be executed for this process.

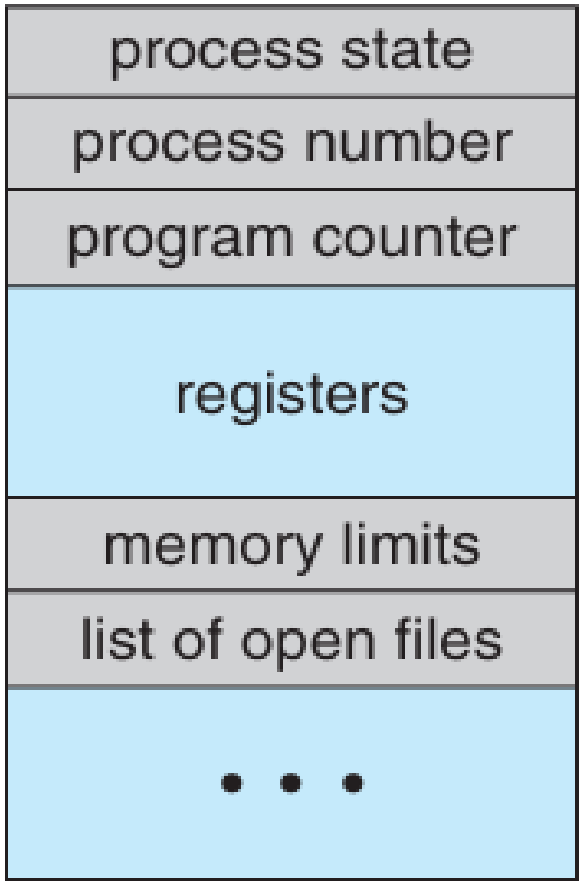


Fig. The Process Control Block

- **CPU Registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. **Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.**

- **CPU-scheduling information**: This information includes a process prior ity, pointers to scheduling queues, and any other scheduling parameters.

- **Memory-management information.** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.

- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

- **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

Thread:

- The process model discussed so far has implied that a process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. Thus, the user cannot simultaneously type in characters and run the spell checker.

- Most modern operating systems have extended the process concept to allow a process to have **multiple threads of execution** and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.

- A **multithreaded** word processor could, for example, assign one thread to manage user input while another thread runs the spell checker.

- On systems that support threads, the **PCB is expanded** to include information for each thread. Other changes throughout the system are also needed to support threads.

Process Scheduling

- The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization.

- The objective of time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it is running.

- To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on a CPU. Each CPU core can run one process at a time.

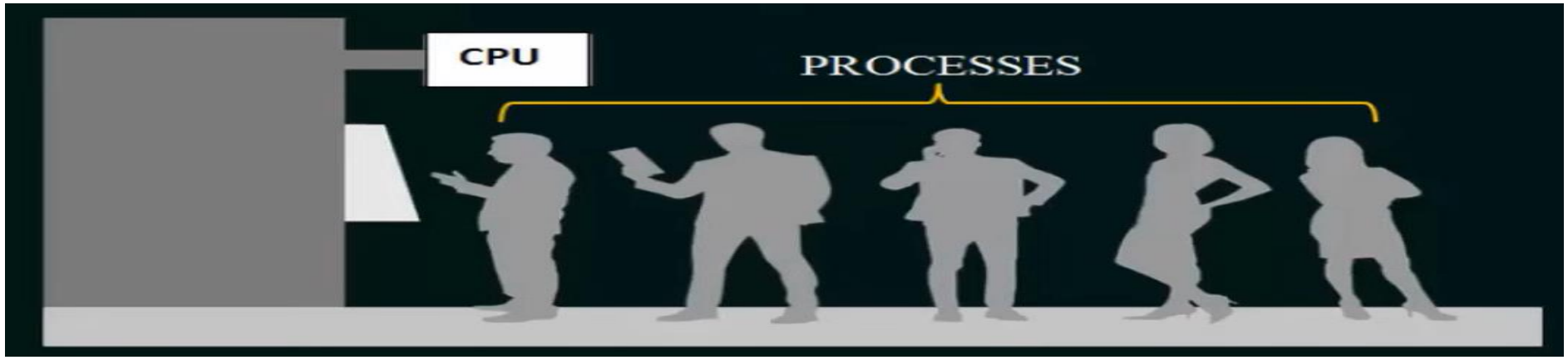
- For a single processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues

- As process enter the system, they are put in **job queue,** which consists of all processes in the system.

- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called **ready queue**.

- The system also includes other queues. When a process is allocated a CPU core, it executes for a while and eventually terminates, is interrupted, or waits for the occurrence of a particular event, such as the completion of an **I/O request**. Suppose the process makes an I/O request to a device such as a disk. Since devices run significantly slower than processors, the process will have to wait for the I/O to become available. Processes that are waiting for a certain event to occur — such as completion of I/O —are placed in a **wait queue**



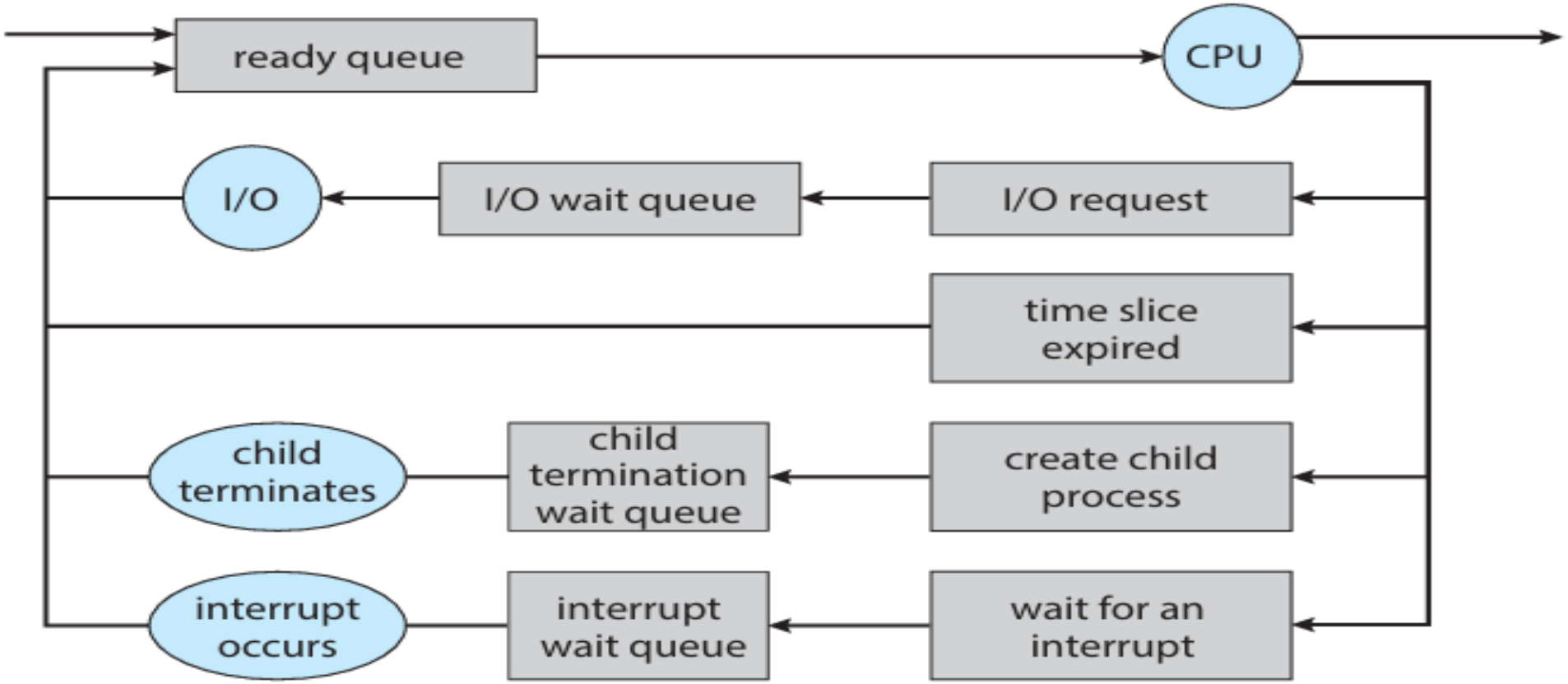


Fig. Queueing-diagram representation of process scheduling

- A common representation of process scheduling is a **queueing diagram** (as shown in above figure). Two types of queues are present: the ready queue and a set of wait queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. Once the process is allocated a CPU core and is executing, one of several events could occur:

1. The process could issue an I/O request and then be placed in an I/O wait queue.

2. The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.

3. The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Scheduling Queues (Contd.)

We have many processes ready to run. There are two types of multiprogramming:

- **Preemption:** Process is forcefully removed from CPU. Pre-emption is also called time sharing or multitasking.

- **Non-Preemption:** Processes are not removed until they complete the execution. Once control is given to the CPU for a process execution, till the CPU releases the control by itself, control cannot be taken back forcibly from the CPU.

Context Switching

- Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems.

- When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU core so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.

- The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state , and memory-management information.

- Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch.**

- Context switching refers to a technique/method used by the OS to switch processes from a given state to another one for the execution of its function using the CPUs present in the system.

- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

- Context switch time is pure overhead, because the system does no useful work while switching.

- Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Atypical speed is a several microseconds.

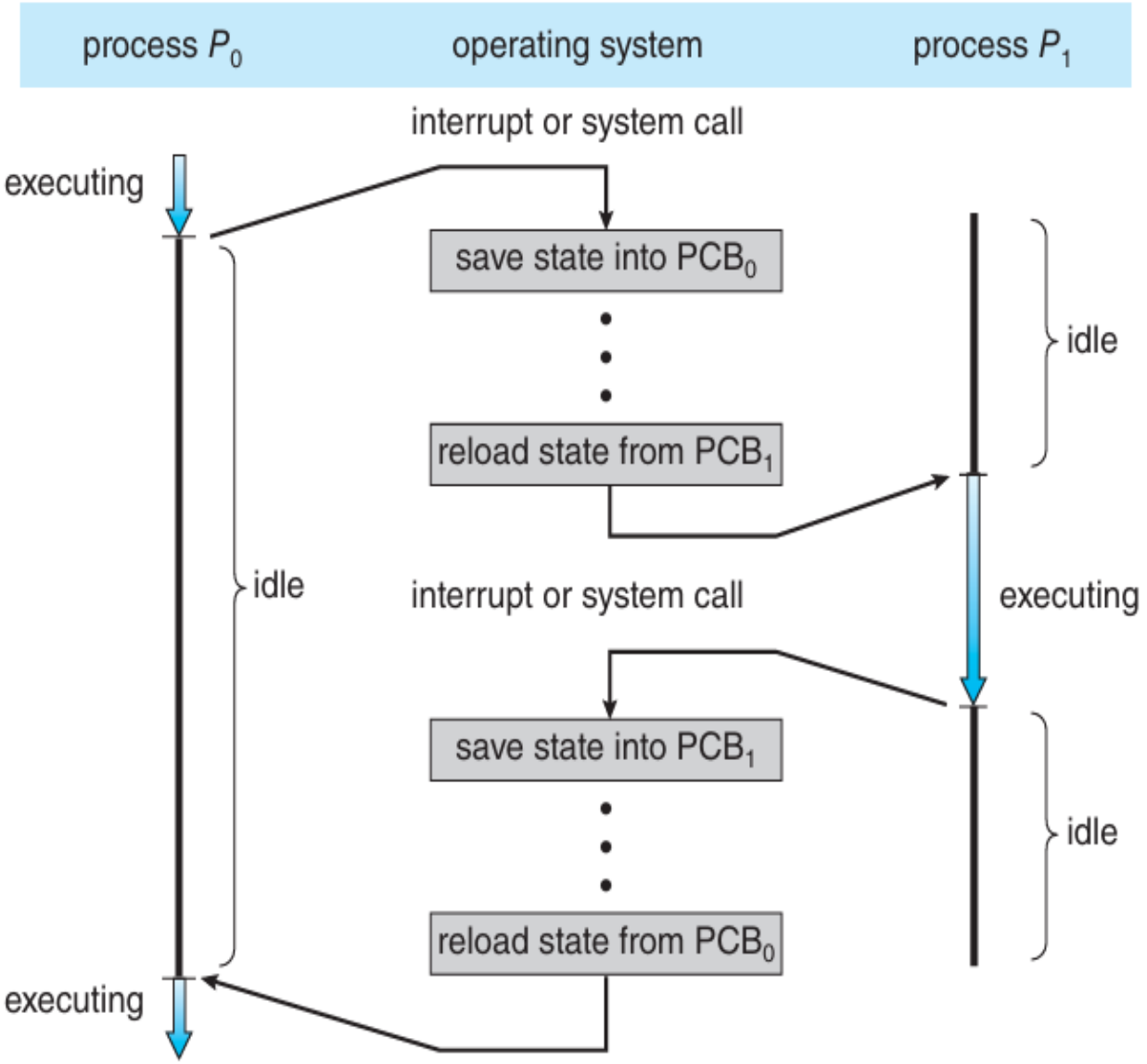


Fig. Diagram showing context switch from process to process

Cause of Context Switching

- **Interrupts:** The CPU requests the data to be read from a disk. In case there are interrupts, the context switching would automatically switch a part of the hardware that needs less time to handle the interrupts.

- **Multitasking:** Context switching is the characteristic of multitasking. They allow a process to switch from the CPU to allow another process to run. When switching a given process, the old state gets saved so as to resume the execution of the process at the very same point in a system.

- **Kernel/User Switch:** It’s used in the OS when it is switching between the kernel mode and the user mode.

Process vs Thread

Figure Source: Internet

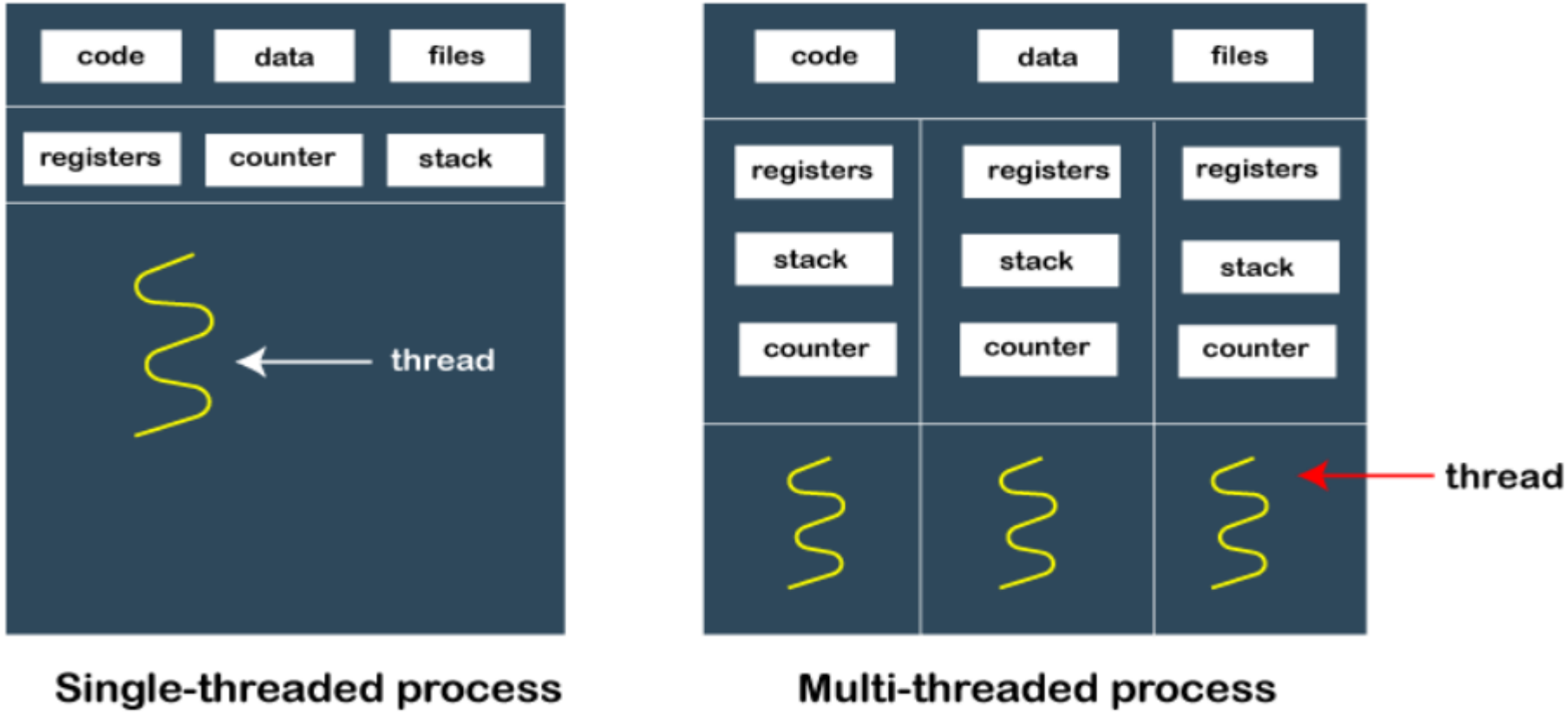
Process vs Thread

- **Process** is the program under execution whereas the thread is part of process. **Threads** of a process can be used when same process is required multiple times. A process can consists of multiple threads.

- Processes are basically the programs that are dispatched from the ready state and are scheduled in the CPU for execution. PCB (Process Control Block) holds the context of process. A process can create other processes which are known as Child Processes. The process takes more time to terminate, and it is isolated means it does not share the memory with any other process. The process can have the following states new, ready, running, waiting, terminated, and suspended.

- **Thread** is the segment of a process which means a process can have multiple threads and these multiple threads are contained within a process. A thread has three states: Running, Ready, and Blocked.

- The thread takes less time to terminate as compared to the process but unlike the process, threads do not isolate.



- All the threads within one process are interrelated to each other. Threads have some common information, such as data segment, code segment, files, etc., that is shared to their peer threads. But contains its own registers, stack, and counter.

- A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

- When a process starts, OS assigns the memory and resources to it. Each thread within a process shares the memory and resources of that process only.

- If a single thread executes in a process, it is known as a single-threaded And if multiple threads execute simultaneously, then it is known as **multithreading.**

- **Types of Thread:** User Level Thread and Kernel-Level Thread.

- **User Level Thread:** As the name suggests, the user-level threads are only managed by users, and the kernel does not have its information. These are faster, easy to create and manage. The kernel takes all these threads as a single process and handles them as one process only. The user-level threads are implemented by user-level libraries, not by the system calls.

- **Kernel Level Thread:** The kernel-level threads are handled by the Operating system and managed by its kernel. These threads are slower than user-level threads because context information is managed by the kernel. To create and implement a kernel-level thread, we need to make a system call.

Process	Thread
❑ A process is an instance of a program that is being executed or processed.	❑ Thread is a segment of a process or a lightweight process that is managed by the scheduler independently.
❑ Processes are independent of each other and hence don't share a memory or other resources.	❑ Threads are interdependent and share memory.
❑ Each process is treated as a new process by the operating system.	❑ The operating system takes all the user-level threads as a single process.
❑ If one process gets blocked by the operating system, then the other process can continue the execution.	❑ If any user-level thread gets blocked, all of its peer threads also get blocked because OS takes all of them as a single process.
❑ Context switching between two processes takes much time as they are heavy compared to thread.	❑ Context switching between the threads is fast because they are very lightweight.
❑ The data segment and code segment of each process are independent of the other.	❑ Threads share data segment and code segment with their peer threads; hence are the same for other threads also.
❑ The operating system takes more time to terminate a process.	❑ Threads can be terminated in very little time.
❑ New process creation is more time taking as each new process takes all the resources.	❑ A thread needs less time for creation.

Multithreading Models

- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, and macOS— support kernel threads.

- The three common ways of establishing such a relationship between user thread and kernel thread: ***the many-to-one model, the one-to one model, and the many-to-many model.***

- The thread takes less time to terminate as compared to the process but unlike the process, threads do not isolate.

Many-to-one model

- The many-to-one model (as shown in Fig.) maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- However, the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

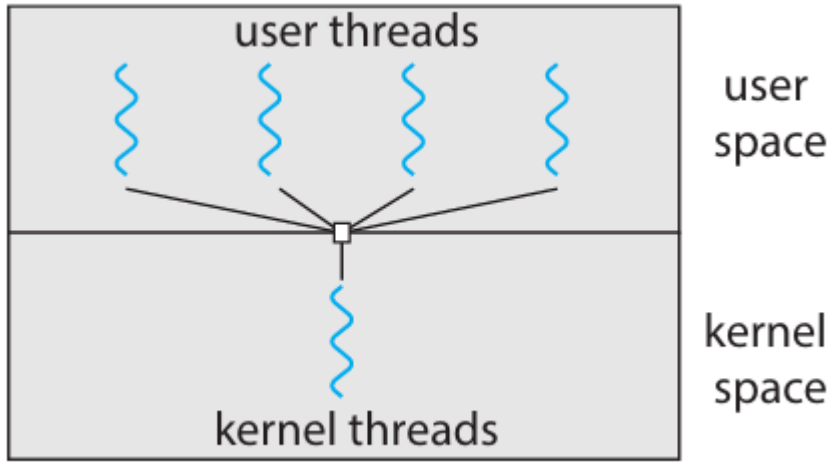


Fig. Many-to-one model

One-to-One Model

- The one-to-one model (as shown in Fig.) maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of a system.
- Linux, along with the family of Windows operating systems, implement the one-to-one model.

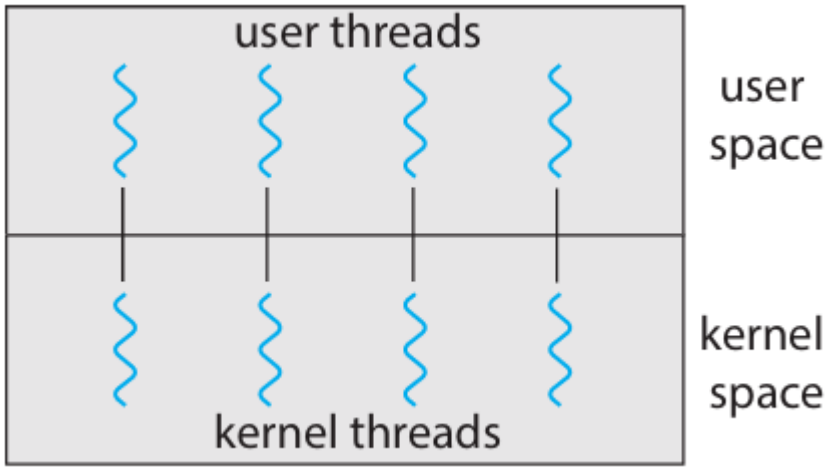


Fig. One-to-One Model

Many-to-Many Model

- The many-to-many model (as shown in Fig.) multiplexes many user-level threads to a smaller or equal number of kernel threads.

- The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a system with eight processing cores than a system with four cores).

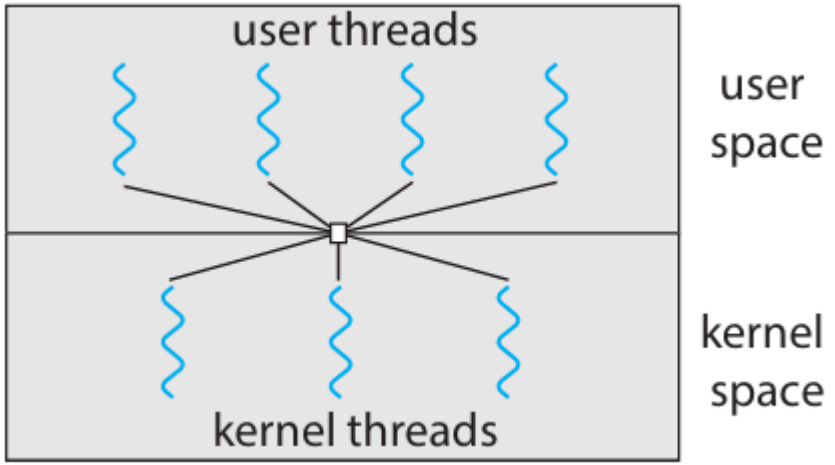


Fig. Many-to-many model.

References

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, “Operating System Concepts,” Eleventh Edition (Willey).

2. Andrew S. Tanenbaum, “Modern Operating Systems”, Fourth Edition (Pearson Publications), 2014.

3. https://www.geeksforgeeks.org/.

4. https://www.javatpoint.com/.

5. https://www.tutorialspoint.com/.

6. https://www.nesoacademy.org/.