Unit - II Process Description And Control

* What is a process ? —

*] Process and Process Control Blocks —

Process is defined as —

- A program in execution.
- An instance of a program running on a computer.
- The entity that can be assigned to and executed on a processor.
- A unit of activity characterised by the execution of a sequence of instructions a current state, and an associated set of system resources.

*] Simplified Process Control Block —

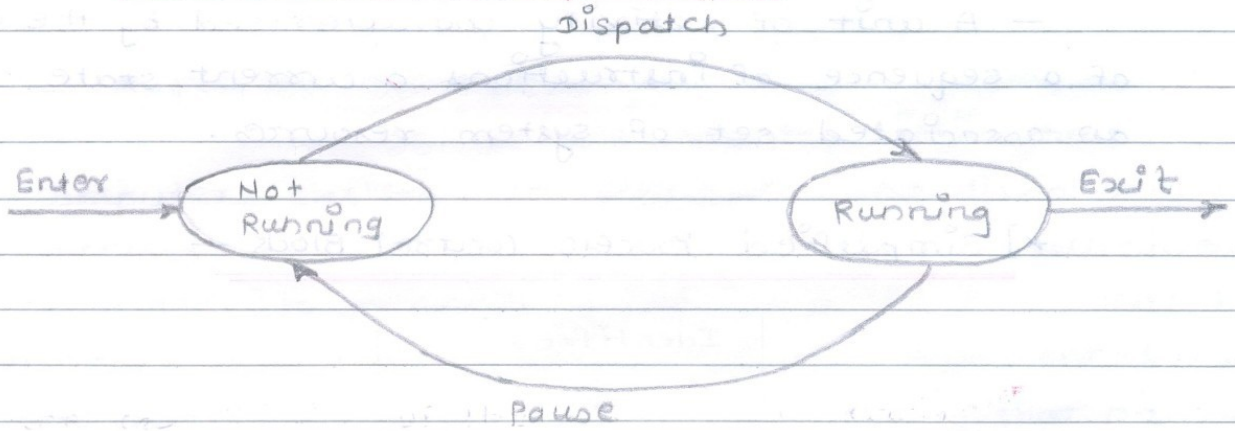
Identifier
State
Priority
Program Counter
Memory Pointers
Context Data
I/O status Information
Accounting information
. . .

Fig - Simplified Process Control Block

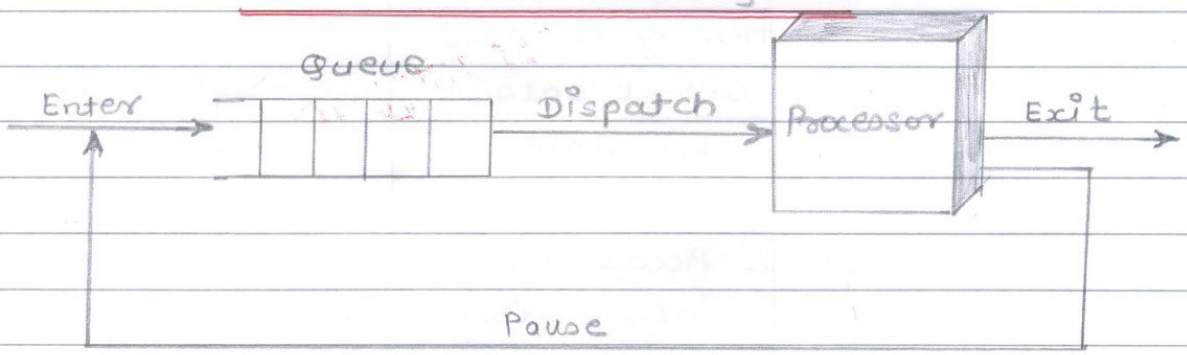
* Process States –

We can characterize the behaviour of an individual process by listing the sequence of instructions that execute for that process. Such a listing is referred to as a trace of the process. We can characterized behaviour of the processor by showing how the traces of the various processes are interleaved.

* A Two - State Process Model –



a) State transition diagram



b) Queuing diagram

Fig – Two - State Process Model

DATE :

* The creation and Termination of Processes -

*] Process creation — When a new process is to be added to those currently being managed, the OS builds the data structures that are used to manage the process and allocates address space in main memory to the process.

When the OS creates a process at the explicit request of another process, the action is referred to as process spawning.

*] Process Termination - Any computer system must provide a means for a process to indicate its completion. A batch job should include a Halt instructions or an explicit OS service call for termination

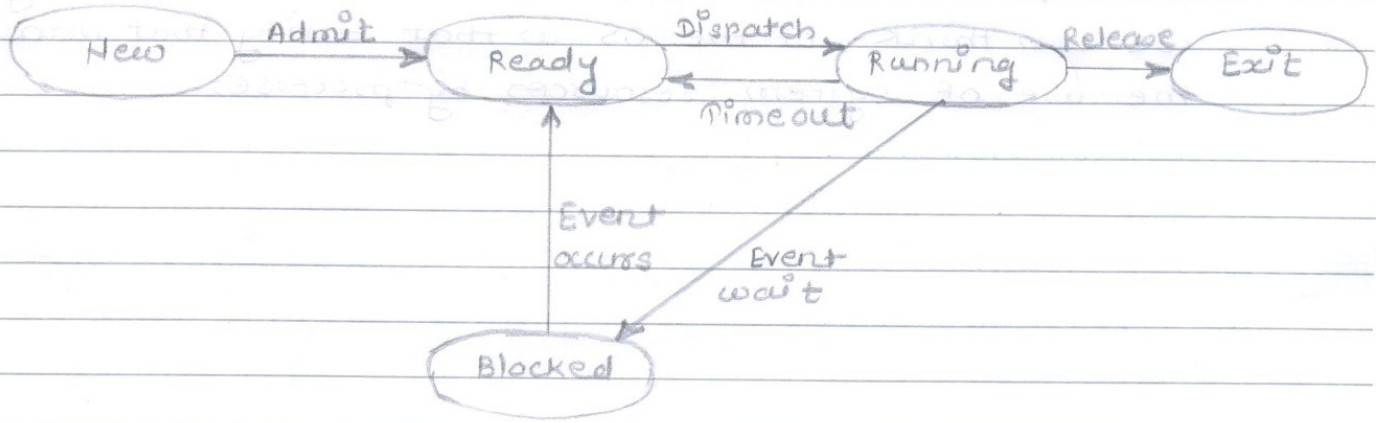


Fig - Five - State Process Model

The five states in this new diagram are as follows -

- Running :- The process that is currently being executed. we will assume a computer with a single processor, so at most one process at a time can be in this state.
- Ready :- A process that is prepared to execute when given the opportunity.
- Blocked / Waiting :- A process that can not execute until some event occurs, such as the completion of an I/O operation.

- New :- A process that has just been created but has not yet been admitted to the pool of executable processes by the OS. Typically, a new process has not yet been loaded into main memory, although its process control block has been created.

- Exit :- A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason.

* Process Description -

The OS controls events within the computer system. It schedules and dispatches processes for execution by the processor, allocates resources to processes, and responds to requests by user processes for basic services. Fundamentally, we can think of the OS as that entity that manages the use of system resources by processes.

* Operating System Control Structures —

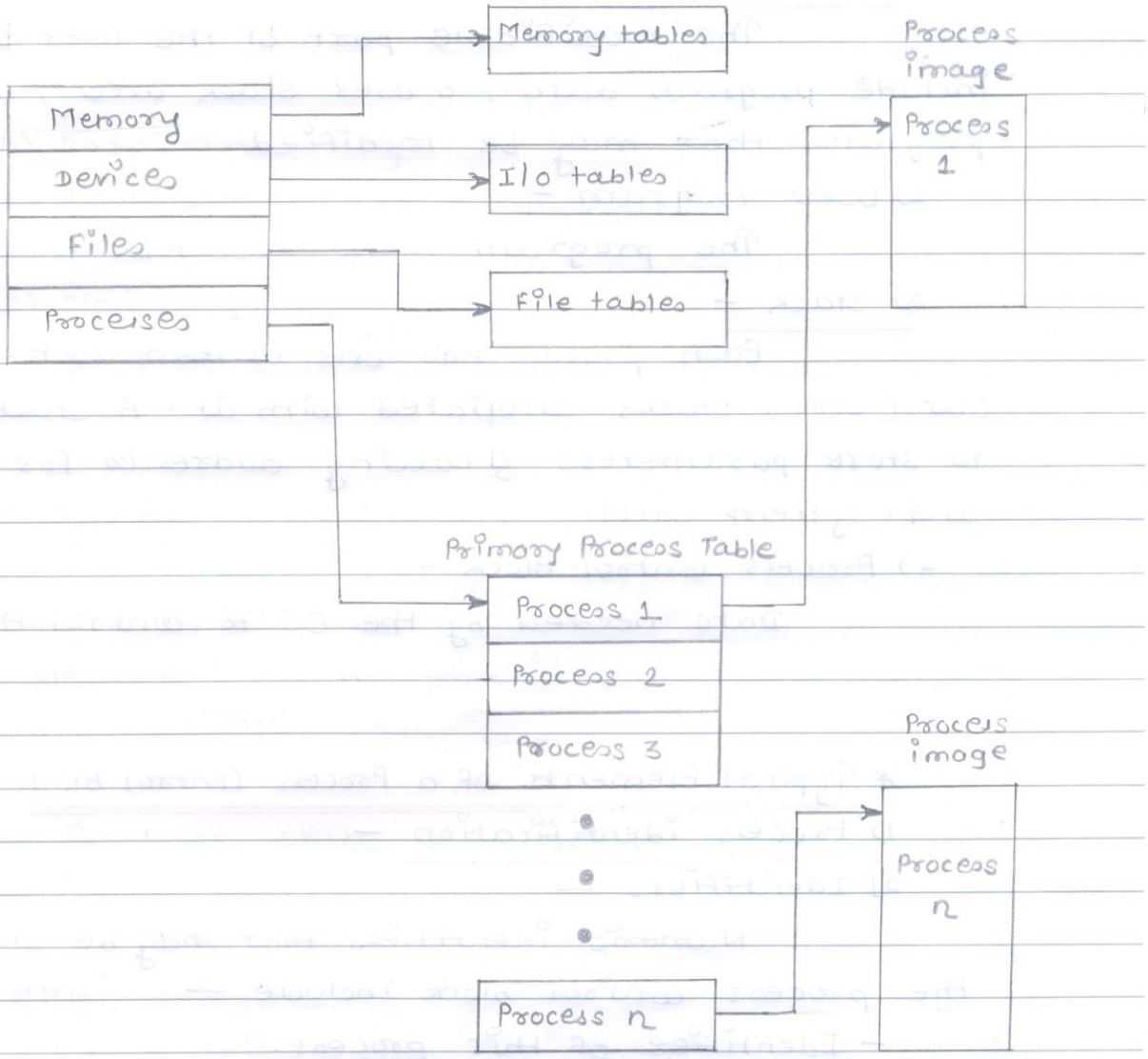


Fig — General structure of Operating System Control Tables

* Process Control structures —

Consider what the OS must know if it is to manage and control a process. First it must know where the process is located, and second, it must know the attributes of the process that are necessary for its management (e.g - process ID and process state)

* Typical Elements of a Process Image –

1) User Data –

The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.

2) User Program –

The program to be executed.

3) Stack –

Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters & calling addresses for procedure and system calls.

4) Process Control Block –

Data needed by the OS to control the process.

* Typical Elements of a Process Control Block –

1) Process identification –

a] Identifiers –

Numeric identifiers that may be stored with the process control block include –

- Identifier of this process.
- Identifier of the process that created this processes (parent process)
- User identifier

2) Processor State Information –

a] User-visible Registers

b] Control and Status Registers

c] Stack Pointers

3) Process Control Information –

a] Scheduling and state information –

i] Process state

ii] Priority
iii] Scheduling - related information
iv] Event

b) Data structuring
c) Interprocess Communication
d) Process Privileges
e) Memory Management
f) Resource Ownership and Utilization

* Process Attributes –

A sophisticated multiprogramming system requires a great deal of information about each process.

We can group the process control block information into three general categories –

- Process Identification
- Processor state Information
- Process control information

With respect to process identification, in virtually all operating systems, each process is assigned a unique numeric identifier, which may simply be an index into the primary process table.

Processor state information consists of the contents of processor registers. While a process is running, of course, the information is in the registers. When a process is interrupted, all of this register information must be saved so that it can be restored when the process resumes execution.

The third major category of information in the process control block can be called, for want of a better name, process control information.

DATE :

* Process List structures –

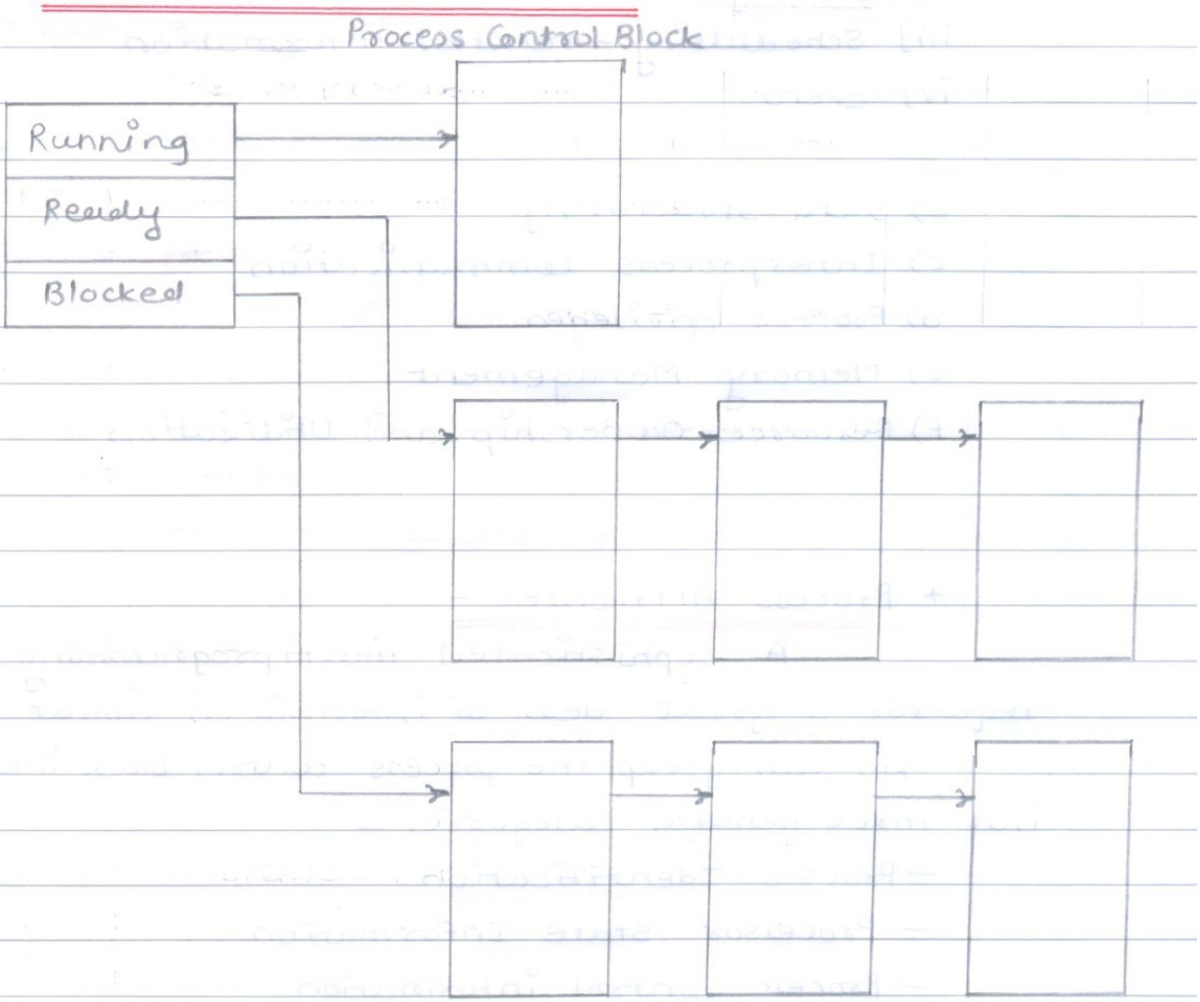


Fig – Process List structures

* The Role of the Process Control Block –

The process control block is the most important data structure in an OS. Each Process Control Block contains all of the information about a process that is needed by the OS. The blocks are read and/or modified by virtually every module in the OS, including those involved ~~in the OS~~ with scheduling, resource allocation, interrupt processing & performance monitoring and analysis. One can say that the set of process control blocks defines the state of the OS.

* Process Control –

*] Modes of Execution –

The less privileged mode is often referred to as the user mode. The more privileged mode is referred to as the system mode, control mode, or kernel mode. This last term refers to the kernel of the OS, which is that portion of the OS that encompasses the important system functions.

The below table lists the functions typically found in the kernel of an OS.

* Typical Functions of an Operating System Kernel –

1) Process Management –

a] Process creation and termination.
b] Process scheduling and dispatching.
c] Process switching.
d] Process synchronization and support for interprocess communication.
e] Management of process control blocks.

2) Memory Management –

a] Allocation of address space to processes.
b] Swapping
c] Page and segment management.

3) I/O Management –

a] Buffer management
b] Allocation of I/O channels & devices to processes.

4) Support Functions –

a] Interrupt handling
b] Accounting
c] Monitoring

* Process Creation –

Once the OS decides, for whatever reason, to create a new process, it can proceed as follows –

1) Assign a unique process identifier to the new process.
2) Allocate space for the process.
3) Initialize the process control block
4) Set the appropriate linkages.
5) Create or expand other data structures.

* Process Switching –

When to switch processes – A process switch may occure any time that the OS has gained control from the currently running process. Table suggests the possible events that may give control to the OS.

* Mechanism for Interrupting the execution of a process –

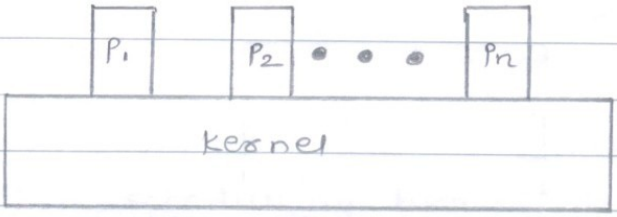
Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction.	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction.	Handling to an error or an exception condition.
Supervisor call	Explicit request	Call to an operating system function.

* Execution of the Operating system -

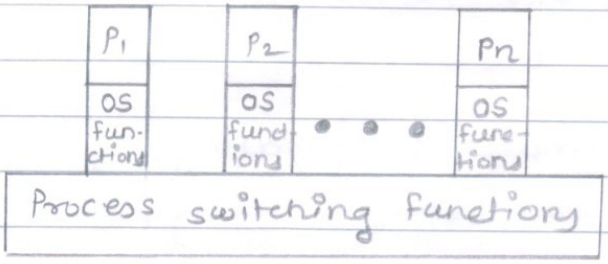
We pointed out two intriguing facts about operating systems.

- The OS functions in the same way as ordinary computer software in the sense that the OS is a set of programs executed by the processor.
- The OS frequently relinquishes control & depends on the processor to restore control to the OS.

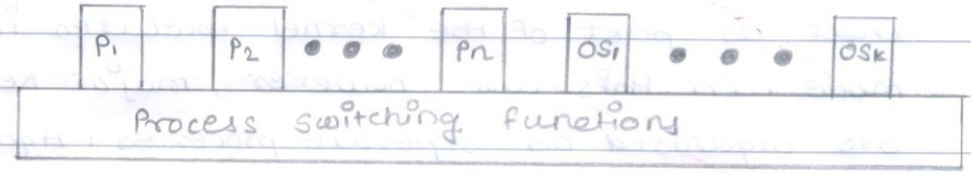
* Execution within User Processes -



a) Seperate Kernel



b) OS functions execute within user processes



c) OS Functions execute as seperate processes

Fig - Relationship between Operating System and User Processes

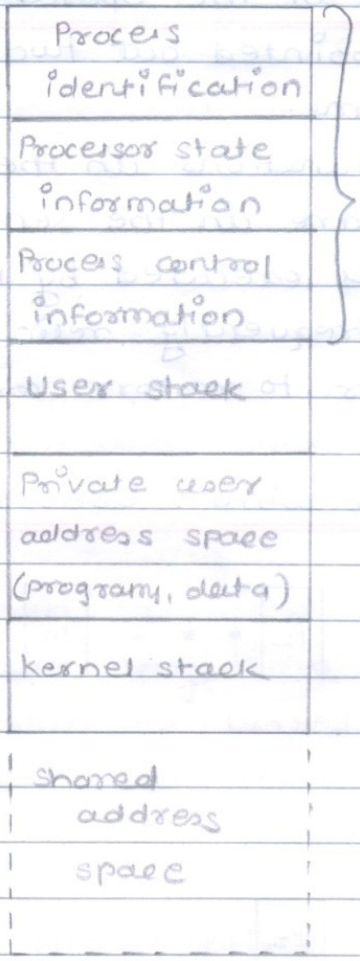


Fig – Process Image : Operating system executes within user space.

* Process Based Operating System –

As in the other options, the software that is part of the kernel executes in a kernel mode. In this case, however, major kernel functions are organised as separate processes. Again, there may be a small amount of process-switching code that is executed outside of any process.

This approach has several advantages. It imposes a program design discipline that encourages the use of a modular OS with minimal, clean interfaces between the modules. In addition, some noncritical operating system functions are conveniently implemented

* Processes And Threads –

The discussion so far has presented the concept of a process as embodying two characteristics.

* Resource ownership –

A process includes a virtual address space to hold the process image.

* Scheduling / execution –

The execution of a process follows an execution path (trace) through one or more programs.

To distinguish the two characteristics, the unit of dispatching is usually referred to as a thread or lightweight process, while the unit of resource ownership is usually still referred to as a process or task.

* Multithreading –

Multithreading refers to the ability of an OS to support multiple, concurrent paths of execution within a single process. The traditional approach of a single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach. The two arrangements shown in the left half of fig. ~~above~~ are single-threaded approaches. MS-DOS is an example of an OS that supports a single user process and a single thread. Other operating systems, such as some variants of UNIX, support multiple user processes but only support one thread per process. The right half of the fig. depicts multithreaded approaches. A Java run-time environment is an example of a system of one process with multiple threads.

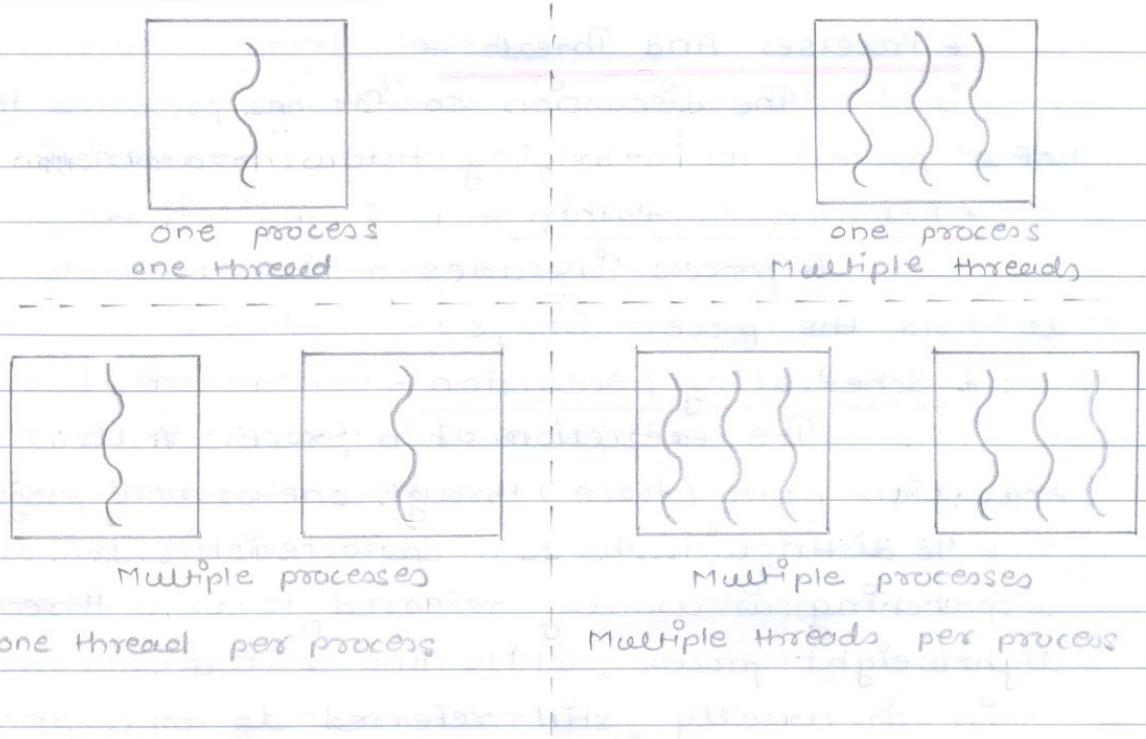


Fig – Threads and Processes [ANDE 97]

* Single threaded process model –

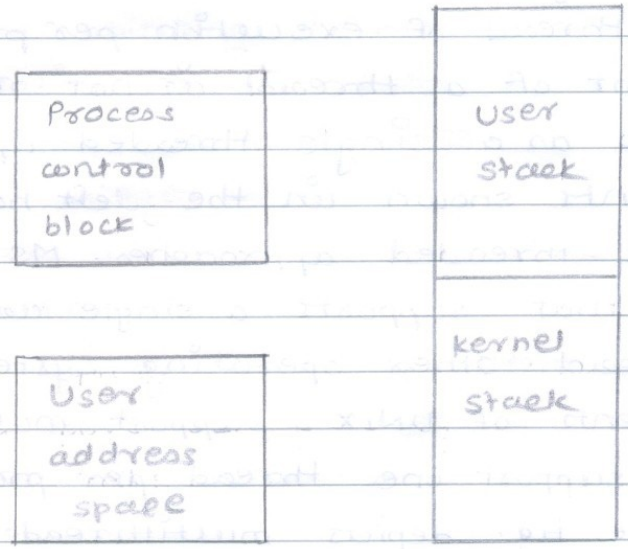


Fig - Single threaded process model

* Multithreaded process model —

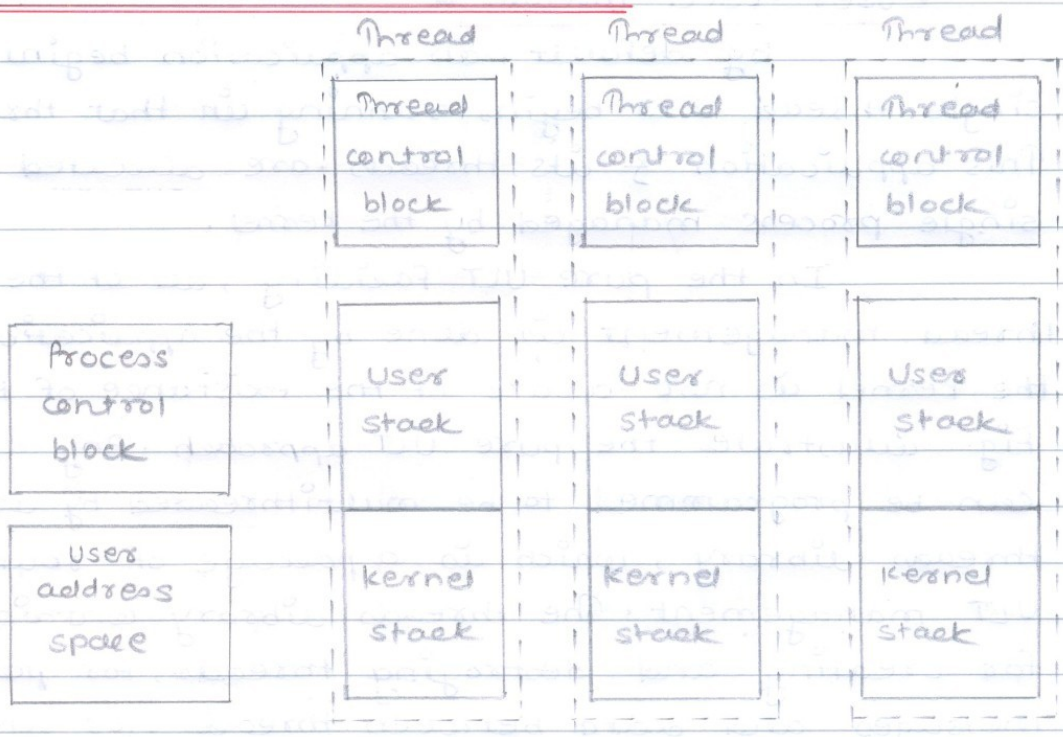
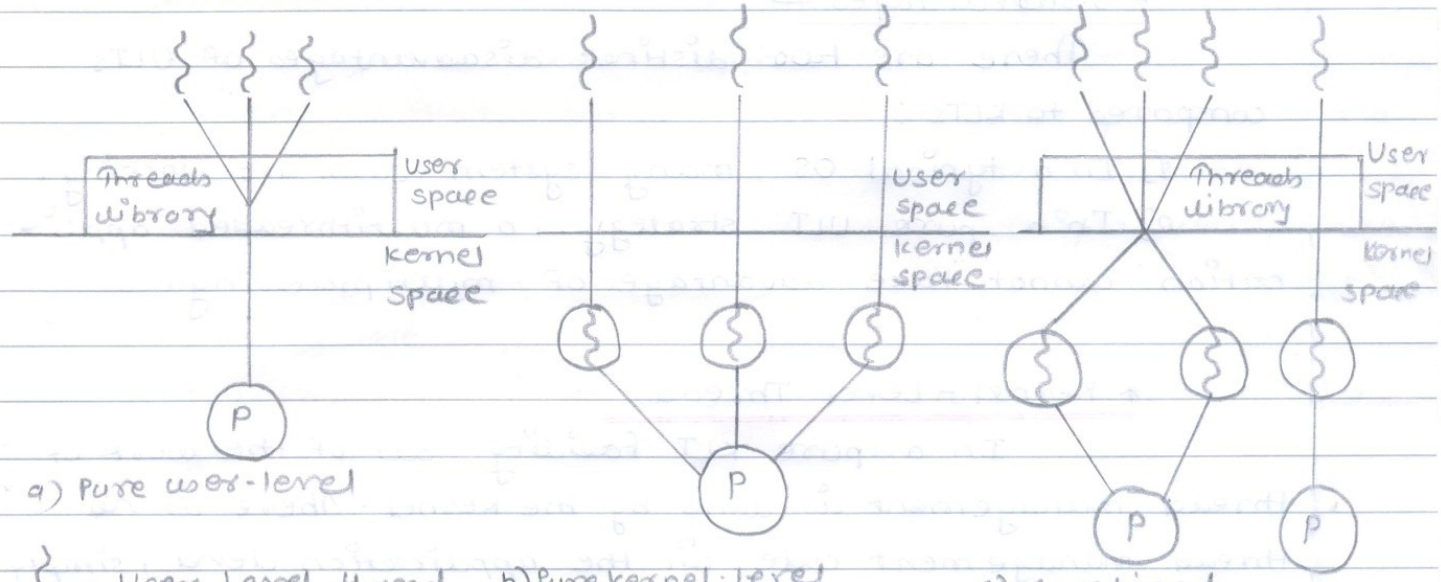


Fig - Multithreaded Process Model

* User-level and Kernel-Level Threads —

There are two broad categories of thread implementation : user-level threads (ULTs) and kernel-level threads (KLTs). The latter are also referred to in the literature as kernel-supported threads or lightweight processes.



- User Level thread
- Kernel-Level thread
- Process

Fig - User-Level & Kernel-Level Threads

* User - Level Threads —

By default, an application begins with a single thread and begins running in that thread. This application & its threads are allocated to a single process managed by the kernel.

In the pure ULT facility, all of the work of thread management is done by the application and the kernel is not aware of the existance of threads. Fig. illustrates the pure ULT approach. Any application can be programmed to be multithreaded by using a threads library, which is a package of routines for ULT management. The threads library contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution, and for saving and restoring thread contexts.

* Advantages —

There are number of advantages to the use of ULTs instead of KLTs, including the following.

1] Thread switching does not require kernel mode privileges.
2] Scheduling can be application specific.
3] ULTs can run on any OS.

* Disadvantages —

There are two distinct disadvantages of ULTs compared to KLTs:

1] In a typical OS, many system calls are blocking.
2] In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing.

* Kernel - Level Threads —

In a pure KLT facility, all of the work of thread management is done by the kernel. There is no thread management code in the application level, simply an application programming interface (API) to the

kernel thread facility. Windows is an example of this approach.

* Combined Approaches –

Some operating systems provide a combined ULT/KLT facility. In a combined system, thread creation is done completely in user space, as in the bulk of the scheduling and synchronization of threads within an application. The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs. The programmer may adjust the number of KLTs for a particular application and processor to achieve the best overall results.

* LINUX Process And Thread Management –

LINUX Tasks –

A process, or task, in Linux is represented by a task_struct data structure. The task_struct data structure contain information in a number of categories.

- State
- Scheduling information
- Identifiers
- Interprocess Communication
- Links
- Times & timers
- File system
- Address space
- Processor-specific context
- Running
- Interruptible
- Uninterruptible
- Stopped
- Zombie

* Linux Threads-

Traditional UNIX systems support a single thread of execution per process, while modern UNIX systems typically provide support for multiple kernel-level threads per process. As with traditional UNIX systems, older versions of the Linux kernel offered no support for multithreading. Instead, applications would need to be written with a set of user-level library functions, the most popular of which is known as pthread (POSIX thread) libraries, with all of the threads mapping into a single kernel-level process.

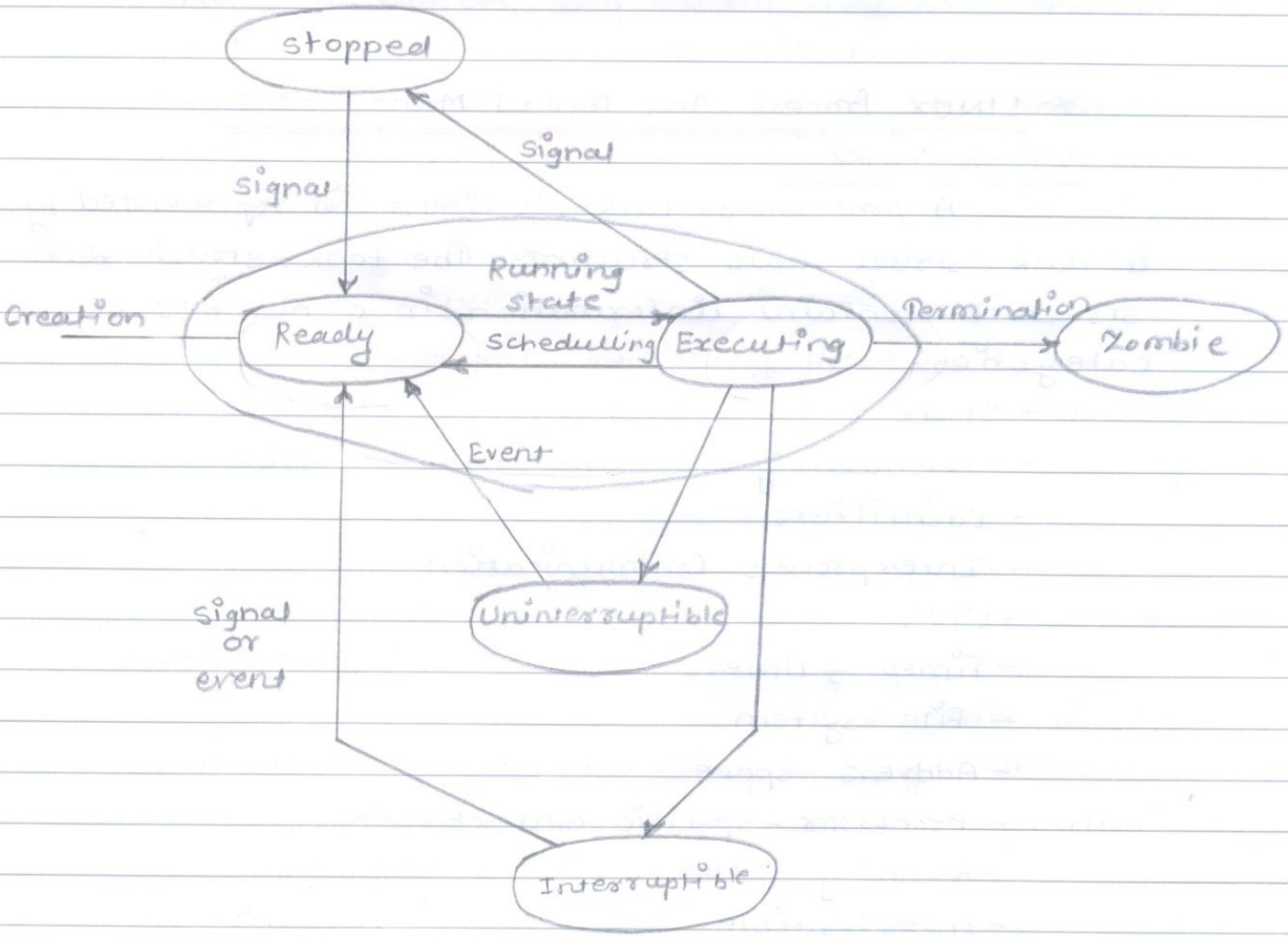


Fig - Linux Process / Thread Model

* Thread Programming Using PThreads –

What is a thread ?

– Multiple stands of execution in a single program are called threads. A more precise defination is that a thread is sequence of control within a process.

When a process executes a fork call, a new copy of the process is created with its own variables and its own PID. This new process is scheduled independantly, and (in general) executes almost independantly of the process that created it. When we create a new thread in a process, in contrast, the new thread of execution gets its own stack (and hence local variables) but shares global variables, file descriptors, signal handlers, and its current directory state with the process that created it.

Linux first aquired thread support around 1996, with a library often referred to as "Linux Threads". This was very close to the POSIX standard (indeed, for many purposes the differences are not noticeable) and it was a significant step forward that enabled Linux programers to use threads for the first time.

The two principle projects were New Generation POSIX Threads (NGPT) and Native POSIX Thread Library (NPTL). Both projects had to make changes to the Linux kernel to support the new libraries, and both offered significant performance improvements over the older Linux threads.

"The Native POSIX Thread Library for Linux" by Ulrich Drepper and Ingo Molnar, which at the time of this writing, is at http://people.redhat.com/drepper/npt-1-design.pdf.

DATE :

* Advantages and Drawbacks of Threads -

Creating a new thread has some distinct advantages over creating a new process in certain circumstances. The overhead cost of creating a new thread is significantly less than that of creating a new process.

Following are some advantages of using threads:

- Sometimes it is very useful to make a program appear to do two things at once. The classic example is to perform a real-time word count on a document while still editing the text. One thread can manage the user's input and perform editing.
- The performance of an application that mixes input, calculation, and output may be improved by running these as three seperate threads.
- Now that multi-cored CPUs are common even in desktop and laptop machines, using multiple threads inside a process can, if the application is suitable enable a single process to better utilize the hardware resources available.
- In general, switching between threads requires the operating system to do much less work than switching between processes.

Threads also have draw backs:

- Writing multithreaded programs requires very careful design.
- Debugging a multithreaded program is much, much harder than debugging a single-threaded one,
- A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine.

* A First Threads Program -

There is a whole set of library calls associated with threads, most of whose names start with pthread. To use these library calls, you must define the macro _REENTRANT, include the file pthread.h, and link with the threads library using -l pthread.

pthread_create creates a new thread, much as fork creates a new process.

```
# include <pthread.h>
int pthread_create (pthread_t *thread, pthread_attr_t
* attr, void
* (* start_routine) (void *), void *arg);
```

When a thread terminates, it calls the pthread_exit function, much as a process calls exit when it terminates. This function terminates the calling thread, returning a pointer to an object. Never use it to return a pointer to a local variable, because the variable will cease to exit when the thread does so, causing a serious bug. pthread_exit is declared as follows:

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

pthread_join is the thread equivalent of wait- that processes use to collect child processes. This function is declared as follows:

```
# include <pthread.h>
int pthread_join (pthread_t th, void ** thread-
return);
```

The first parameter is the thread for which to wait, the identifier that pthread_create filled in for you. The second argument is a pointer to a pointer that itself points to the return value from the thread. Like pthread_create, this function returns zero for success and an error code on failure.

* Uniprocessor Scheduling *

* Types of processor scheduling -

The aim of processor scheduling is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives, such as response time, throughput and processor efficiency. In many systems, this scheduling activity is broken down into three seperate functions. long-, medium-, and short term scheduling. The names suggests the relative time scales with which this functions are performed.

* Types of scheduling -

1) Long term scheduling - The decision to add to the pool of processes to be executed.

2) Medium term scheduling - The decision to add to the number of processes that are partially or fully in main memory.

3) Short-term scheduling - The decision as to which available process will be executed by the processor.

4) I/O scheduling - The decision as to which process's pending I/O request shall be handled by an available I/O device.

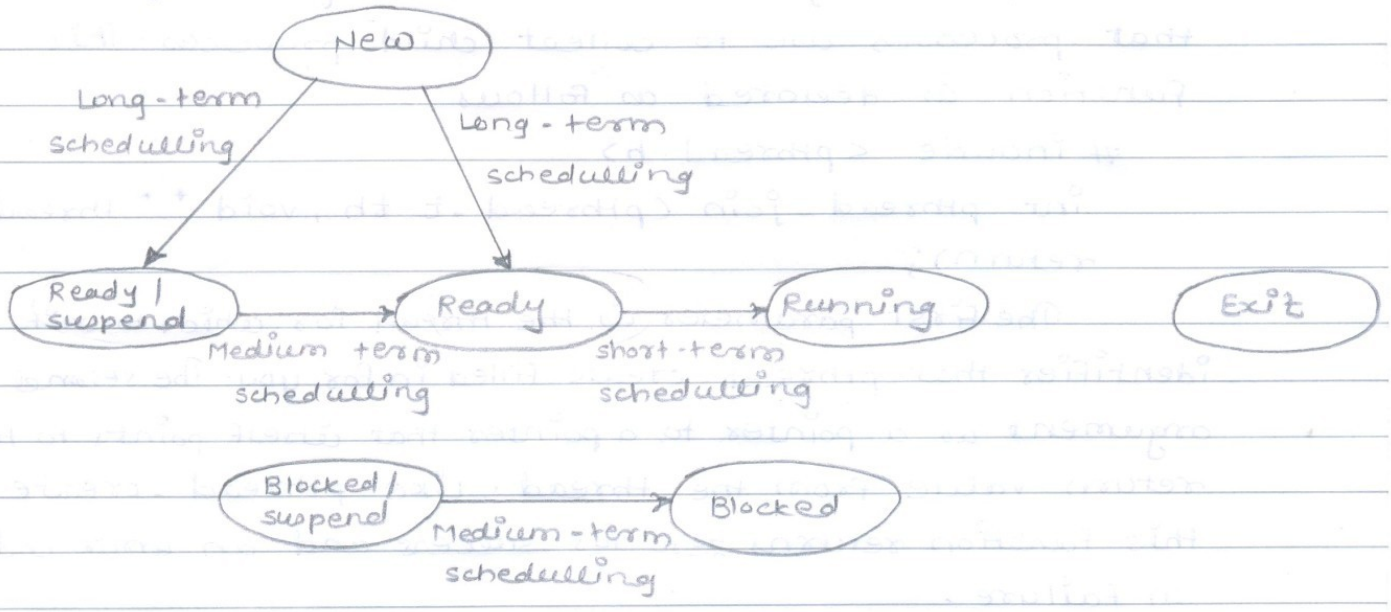


Fig- Scheduling and process state transitions

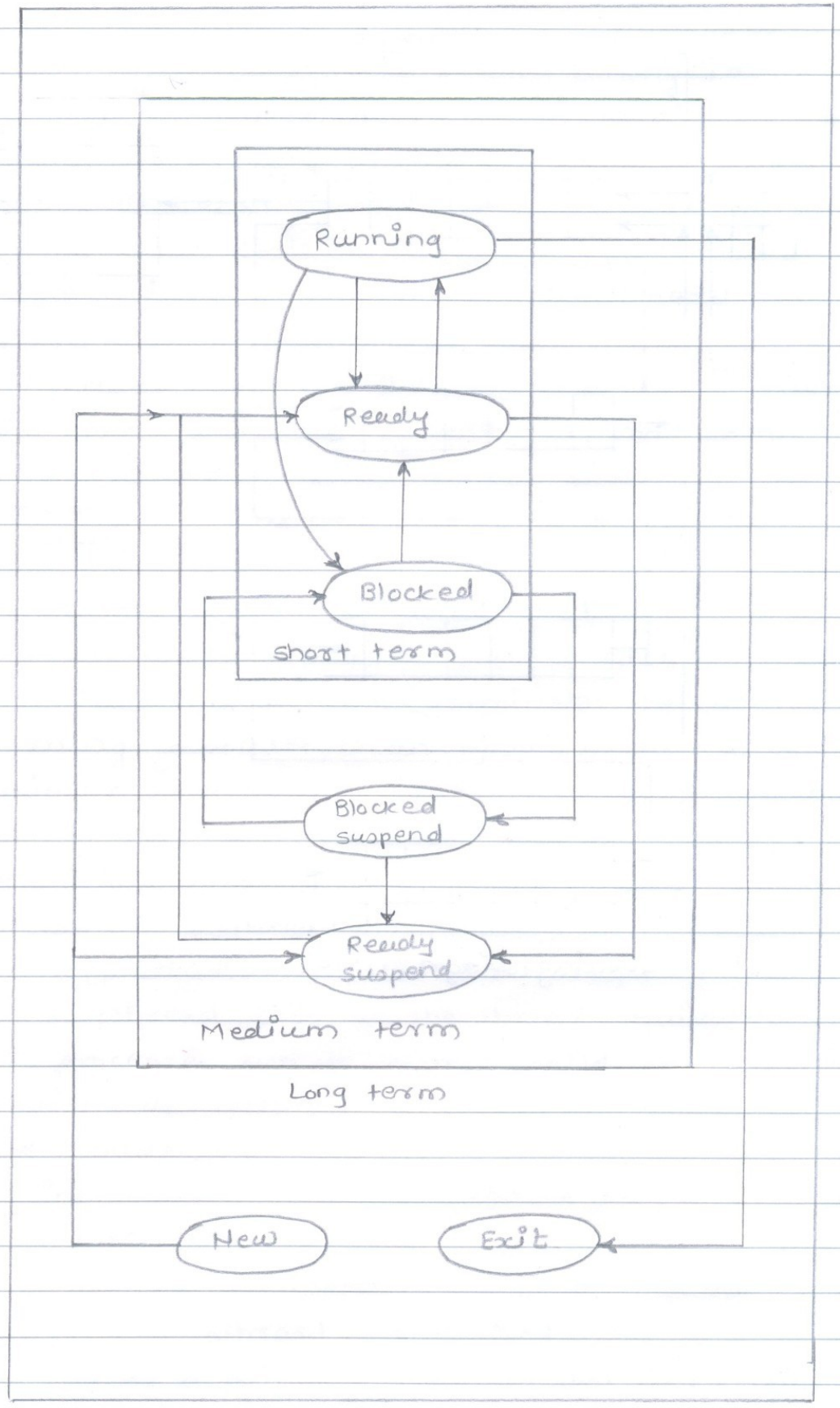


Fig - Levels of scheduling

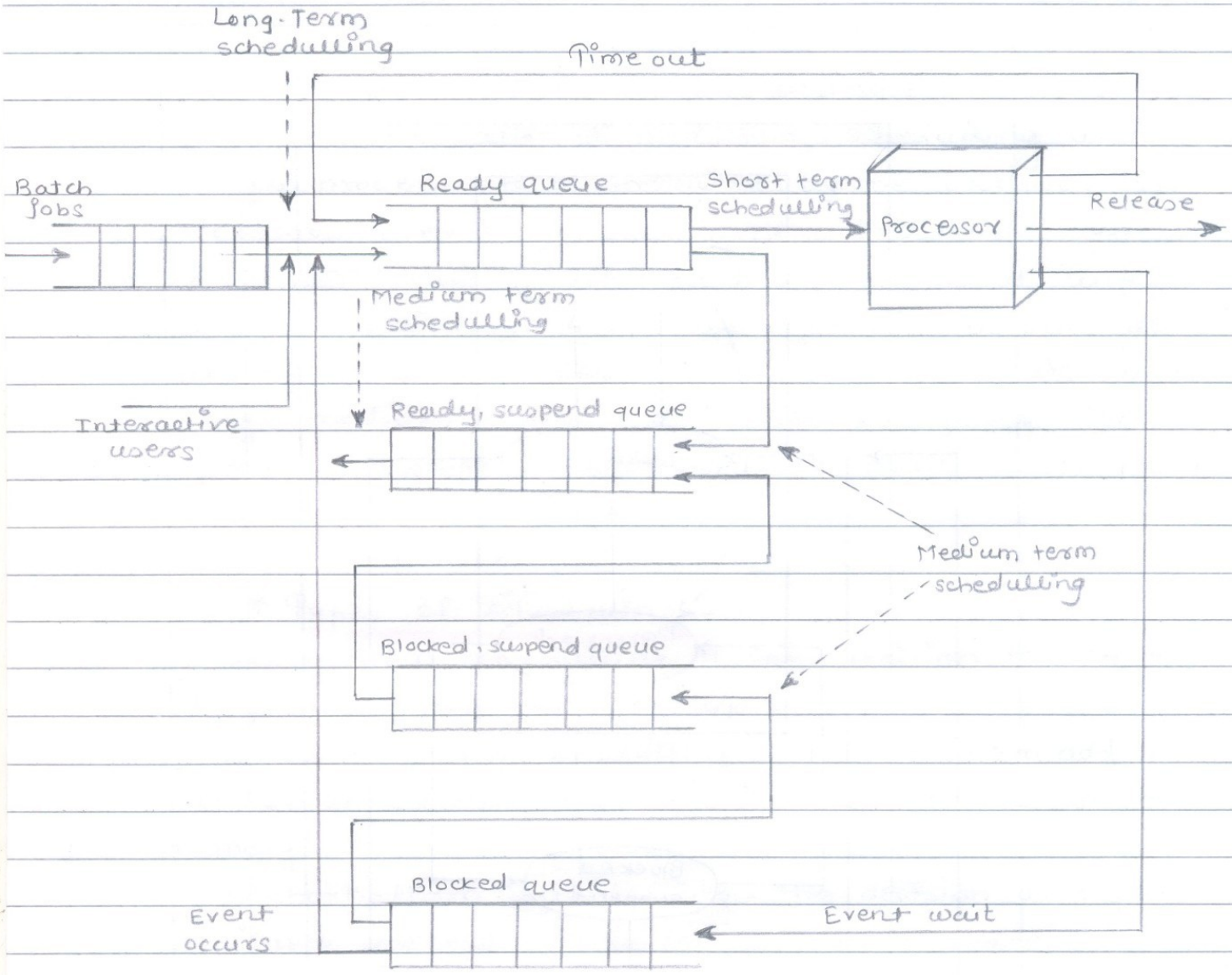


Fig- Queuing Diagram for scheduling

DATE :

* Scheduling Algorithms *

Short - Term Scheduling Criteria -

1) User oriented, Performance Related -

a] Turnaround time - This is the interval of time between the submission of a process and its completion.

b] Response time - For an interactive process, this is the time from the submission of a request until the response begins to be received.

c] Deadlines - When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.

2) User oriented, other -

a] Predictability - A given job should run in about the same amount of time and at about the same cost regardless of the load on the system.

3) System oriented, Performance related -

a] Throughput - The scheduling policy should attempt to maximize the number of processes completed per unit time.

b] Processor utilization - This is the percentage of time that the processor is busy. For an expensive shared system, this is a significant criteria.

4] System oriented, other -

a] Fairness - In the absence of guidance from the user or other system-supplied guidance, processes should be treated the same, and no process should suffer starvation.

b] Enforcing priorities - When processes are

assigned priorities the scheduling policy should favour higher priority processes.

c] Balancing Resources – The scheduling policy should keep the resources of the system busy.

The decision mode specifies the instants in time at which the selection function is exercised. There are two general categories.

1) Nonpreemitive :–

In this case, once a process is in the Running state, it continues to execute until a) it terminates or b) it blocks itself to wait for I/o or to request some operating system service.

2) Preemitive –

The currently running process may be interrupted and moved to the Ready state by the operating system. The decision to preempt may be performed when a new process arrives; when an interrupt occurs that places a blocked process in the Ready state; or periodically, based on a clock interrupt.

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Table – Process Scheduling Example

* A Comparision of Scheduling Policies –

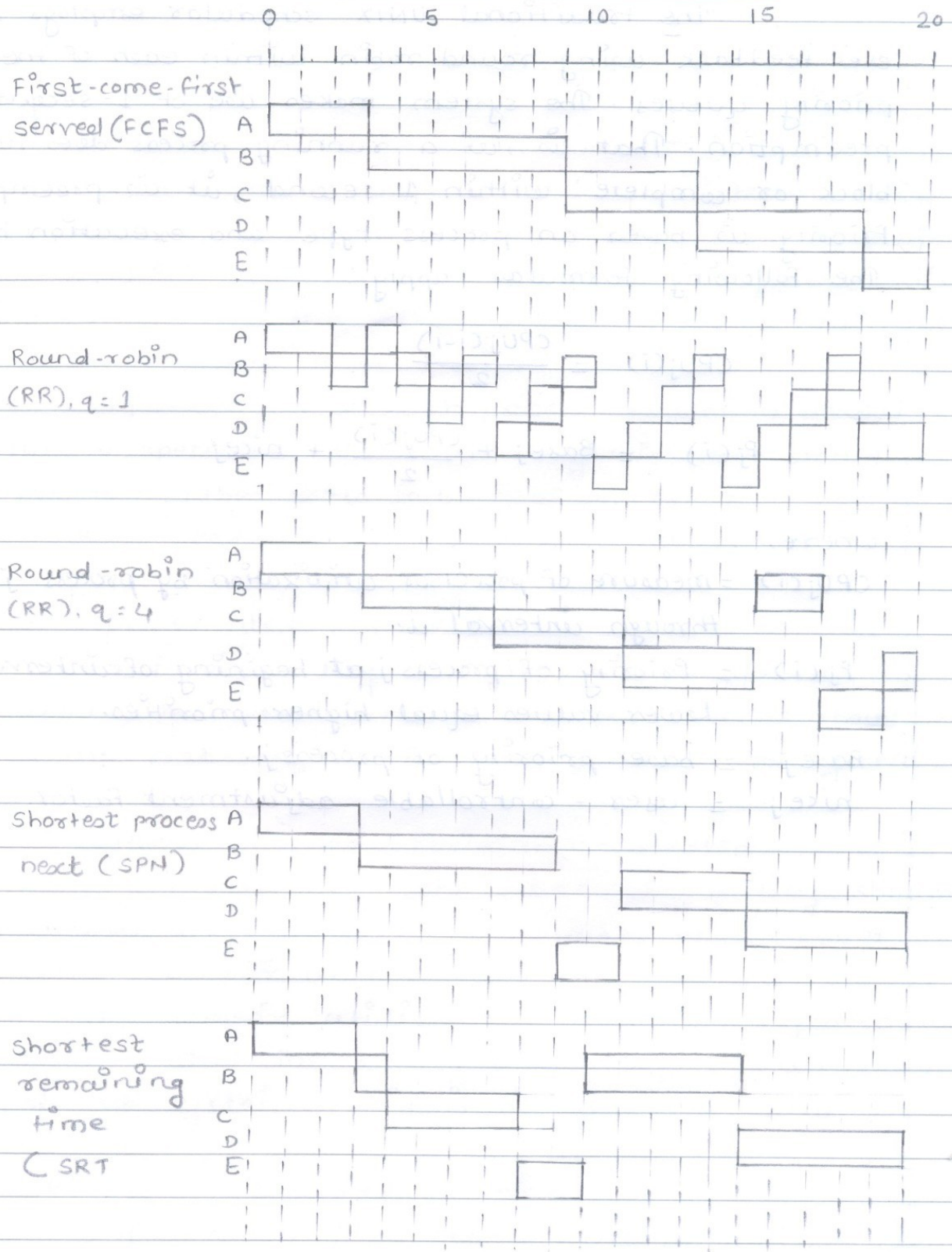


Fig – A Comparision of Scheduling Policies

* Traditional Unix scheduling –

The traditional UNIX schedular employs multi-level feedback using round robin within each of the priority queues. The system makes use of 1-second preemption. That is, if a running process does not block or complete within 1 second, it is preempted. Priority is based on process type and execution history. The following formulas apply:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

where,

$CPU_j(i)$ = measure of processor utilization by process j through interval i.

$P_j(i)$ = Priority of process j at begining of interval i; Lower values equal higher priorities.

$Base_j$ = base priority of process j.

$nice_j$ = user-controllable adjustment factor.

* Example of a Traditional UNIX Process Scheduling

	Process A		Process B		Process C	
	Priority	CPU count	Priority	CPU count	Priority	CPU count
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	68	7 8 9 • • 67	67	15	75	30
4	76	33	68	7 8 9 • • 67	67	15
5	68	16	76	33	68	7

Fig – Example of a Traditional UNIX Process Scheduling

* Multiprocessor Scheduling

We can classify multiprocessor systems as follows:

1) Loosely coupled or distributed multiprocessor/cluster :-

Consists of a collection of relatively autonomous systems, each processor having its own main memory and I/O channels.

2) Functionally specialized processors -

An example is an I/O processor. In this case, there is a master, general-purpose processor; specialized processors are controlled by the master processor and provide services to it.

3) Tightly coupled multiprocessing -

Consists of a set of processors that share a common main memory & are under the integrated control of an operating system.

* Granularity -

A good way of characterizing multiprocessors & placing them in context with other architectures is to consider the synchronisation granularity, or frequency of synchronisation, between processes in a system.

* Synchronisation Granularity and Processes.

Grain size	Description	Synchronisation Interval (Instructions)
Fine	Parallelism inherent in a single instruction stream.	< 20
Medium	Parallel processing or multitasking within a single application.	20 - 200

Grain size	Description	Synchronisation Interval (Instructions)
Coarse	Multiprocessing of concurrent processes in a multiprogramming environment	200 - 2000
Very coarse	Distributed processing across network nodes to form a single computing environment.	2000 - 1M
Independant	Multiple unrelated processes	not applicable

* Thread Scheduling -

On a uniprocessor, threads can be used as a program structuring aid and to overlap I/O with processing. Because of the minimal penalty in doing a thread switch compared to a process switch, these benefits are realized with little cost. However, the full power of threads becomes evident in a multiprocessor system. In this environment, threads can be used to exploit true parallelism in an application. If the various threads of an application are simultaneously run on seperate processors, dramatic gains in performance are possible.

Among the many proposals for multiprocessor thread scheduling & processor assignment, four general approaches stand out.

1) Load sharing -

Processes are not assigned to a particular processor. A global queue of ready threads is maintained, and each processor, when idle,

selects a thread from the queue. The term load sharing is used to distinguish this strategy from load-balancing schemes in which work is allocated on a more permanent basis.

2) Gang scheduling –

A set of related threads is scheduled to run on a set of processors at the same time, on a one-to-one basis.

3) Dedicated processor assignment –

This is the opposite of the load sharing approach and provides implicit scheduling defined by the assignment of threads to processor. Each program is allocated a number of processors equal to a number of threads in the program, for the duration of the program execution. When the program terminates, the processors return to the general pool for possible allocation to another program.

4) Dynamic scheduling –

The number of threads in a process can be altered during the course of execution.

* Real Time scheduling –

Real-time computing may be defined as that type of computing in which the correctness of the system depends not only on the logical results of the computation but also on the time at which the results are produced. We can define a real time system by defining what is meant by a real-time process or task.

A hard real-time task is one that must meet its deadline; otherwise it will cause unacceptable damage or a fatal error to system. A soft real-time task has an associated

deadline that is desirable but not mandatory; it still makes sense to schedule and complete the task even if it has passed its deadline.

* Characteristics of Real Time Operating Systems -

Real-time operating systems can be characterized as having unique requirements in five general areas -

- Determinism
- Responsiveness
- User control
- Reliability
- Fail-soft operation.

* Classes of Algorithms -

1) static table-driven approaches -

These perform a static analysis of feasible schedules of dispatching. The result of the analysis is a schedule that determines, at run time, when task must begin execution.

2) static priority-driven preemptive approaches -

Again, a static analysis is performed, but no schedule is drawn up. Rather, the analysis is used to assign priorities to tasks, so that a traditional priority-driven preemptive scheduler can be used.

3) Dynamic planning-based approaches -

Feasibility is determined at run time (dynamically) rather than offline prior to the start of execution (statically). An arriving task is accepted for execution only if it is feasible to meet its time constraints. One of the results of the feasibility analysis is a schedule or plan that is used to decide when to dispatch this

DATE :

4] Dynamic best effort approaches -

No feasibility analysis is performed. The system tries to meet all deadlines and aborts any started process whose deadline is missed.

* Android Process Management -

The process Lifecycle Hierarchy -

A process on Android can be in one of five different stages at any given time, from most important to least important.

1] Foreground process -

The app you are using is considered the foreground process. other processes can also be considered foreground processes - for e.g - if they are interacting with the process that's currently in the foreground. There are only a few foreground processes at any given time.

2] Visible process -

A visible process isn't in the foreground, but is still affecting what you see on your screen. For e.g - the foreground process may be a dialog that allows you to see an app behind it - the app visiable in the background would be a visiable process.

3] Service process -

A service process is not tied to any app that's visiable on your screen. However, its doing something in the background, such as playing music or downloading data in the background. For e.g - if you start playing music and switch to another app, the music - playing is in the background is being

handled by a service process.

4) Background process –

Background processes are not currently visiable to the user. They have no impact on the experience of using the phone. At any given time, many background processes are currently running. You can think of these background processes as "paused" apps. They're kept in memory so you can quickly resume using them when you go back to them, but they are not using valuable CPU time or other non-memory resources.

5) Empty process –

An empty process doesn't contain any app data anymore. It may be kept around for caching purposes to speed up app launches later, or the system may kill it as necessary.

Android Automatically Manages Processes –

Android does a good job of automatically managing these processes, which is why you don't need a task killer on Android.

When Android needs more system resources, it will start killing the least important processes first. Android will start to kill empty and background processes to free up memory if you are running low. If you need more memory - for e.g - if you're playing a particulary demanding game on a device without much RAM, Android will then start to kill service processes, so your streaming music and file downloads may stop.