Module - 2

Process Control And Scheduling

2.1) # Concept of a Process

A process is a program in execution, meaning it is an actively running computer code. A process can be in 5 possible states (discussed later).

A process contains 5 components (that we need to learn about):

a) text section: comprises the compiled program code, read in when the program is starting execution.

b) data section: stores global and static variables that are initialized before execution.

c) Program Counter: Includes address of current and next instruction to be executed.

d) Stack: Contains temporary data like function parameters, return addresses and local variables.

e) Heap: Memory that is dynamically allocated to the process during execution.

Two processes may be associated with the same program, they are still considered 2 separate

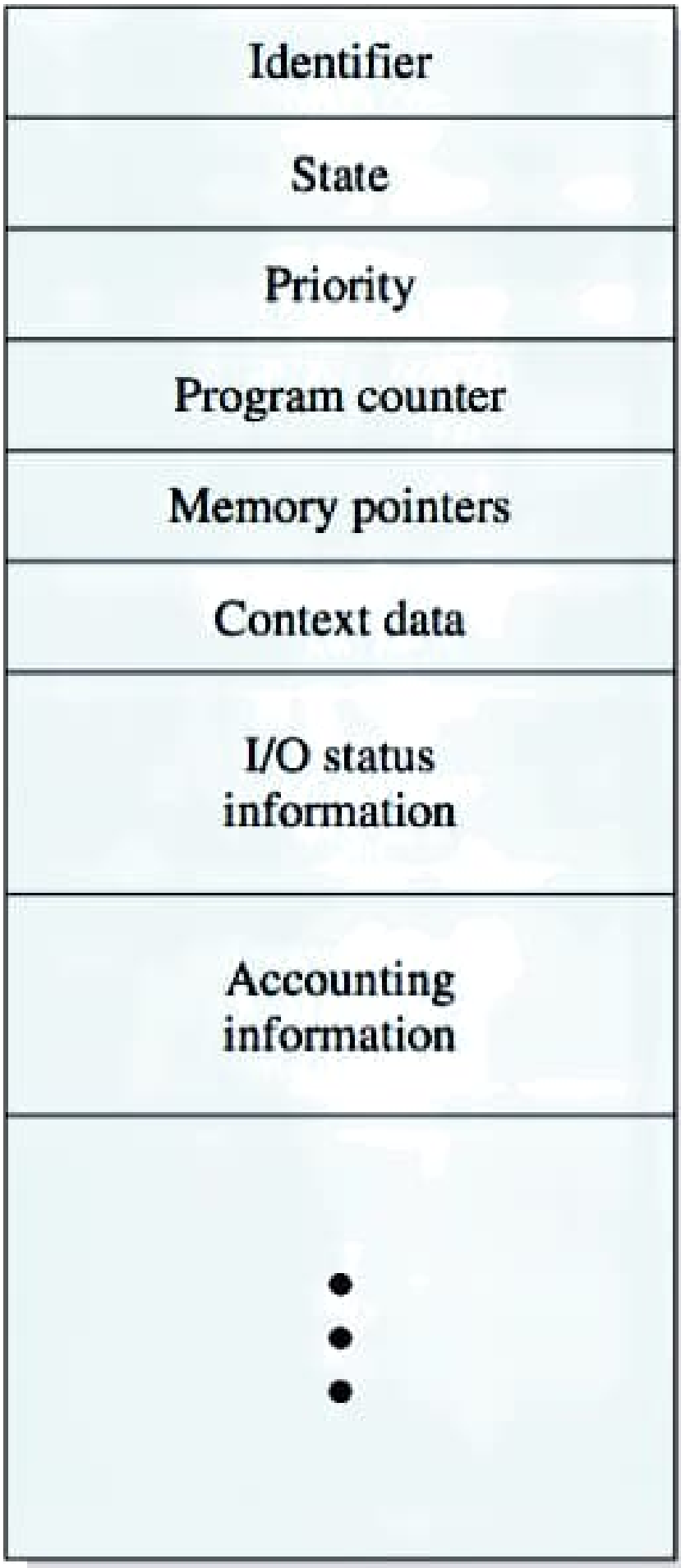


Figure 3.1 Simplified Process Control Block

execution sequences.

Eg: Two browser tabs are opened and worked on. Both will have the same text section because the program code for browser remains same, but they will still have different data section, stack values, etc because the operations on both will be different.

{{ Figure 3.1, Process in Memory }}

Process States:

- Dispatcher: Switches the processor from one process to another.

- Trace of a process: List of the sequence of instructions that execute for that process.

The 5 states of a processor are as follows:

a) Running: The process that is currently being executed.

b) Ready: A process that is prepared to execute when given the opportunity.

c) Blocked/Waiting: A process that cannot execute until some event occurs, eg: completion of an I/O operation.

d) New: A process that has been created in the disk but hasn't been loaded in the Main Memory. It's process control block has been created.

e) Exit: A process that has been released from the pool of executable processes by the OS, either because it completed execution or because it aborted.

Use of New and Exit states:

a) New: The OS keeps some processes in the disk (in new state) because there is a limited amount of processes that can be stored in the ready queue.

b) Exit: Processes that have been terminated may have information needed by other ~~processes~~ Program. Thus the OS maintains the information of that process ~~is~~ for those ~~processes~~ program for some time. After the program has extracted the information needed, the OS deletes the process.

→ All possible transitions between the different states:

a) Null → New: A new process is created to

execute a program.

b) New → Ready: When there is space for a new process to be loaded in the main memory, it is loaded and process is moved to the ready state.

c) Ready → Running: One of the processes in the ready state is chosen by the dispatcher & put in running state.

d) Running → Exit: When a process aborts or completes its execution, it is put in Exit state.

e) Running → Ready: When the running process is paused for some time it is kept in the ready state again. Can happen if process with higher priority enters or if the maximum CPU time is exceeded by the process, etc.

f) Running → Blocked: When the running process initiates an I/O operation and must wait for the operation to complete, it is kept in the blocked state.

g) Blocked → Ready: When the I/O operation it was waiting for completes, it is

sent to the ready state again.

h) Ready → Exit : If the process is terminated by user or if the parent process of this process terminates it when it is in ready state, it's sent to exit state directly.

i) Blocked → Exit : Same as Ready → Exit.

* Since I/O operations are much slower than CPU speed, it could occur that all the ready processes end up in the blocked state. This will take up too much space of the main memory while a new process is being executed.

Thus a new state, called the suspended state can be utilized. One of the blocked processes will be moved to the suspended queue, which is stored outside the main memory, on the disk.

The OS then loads the first suspended process back in the main memory to execute, assuming that the I/O operation it was the suspended process was waiting for has executed.

{{ Figure 3.6 Five Process State Model with one suspended state }}

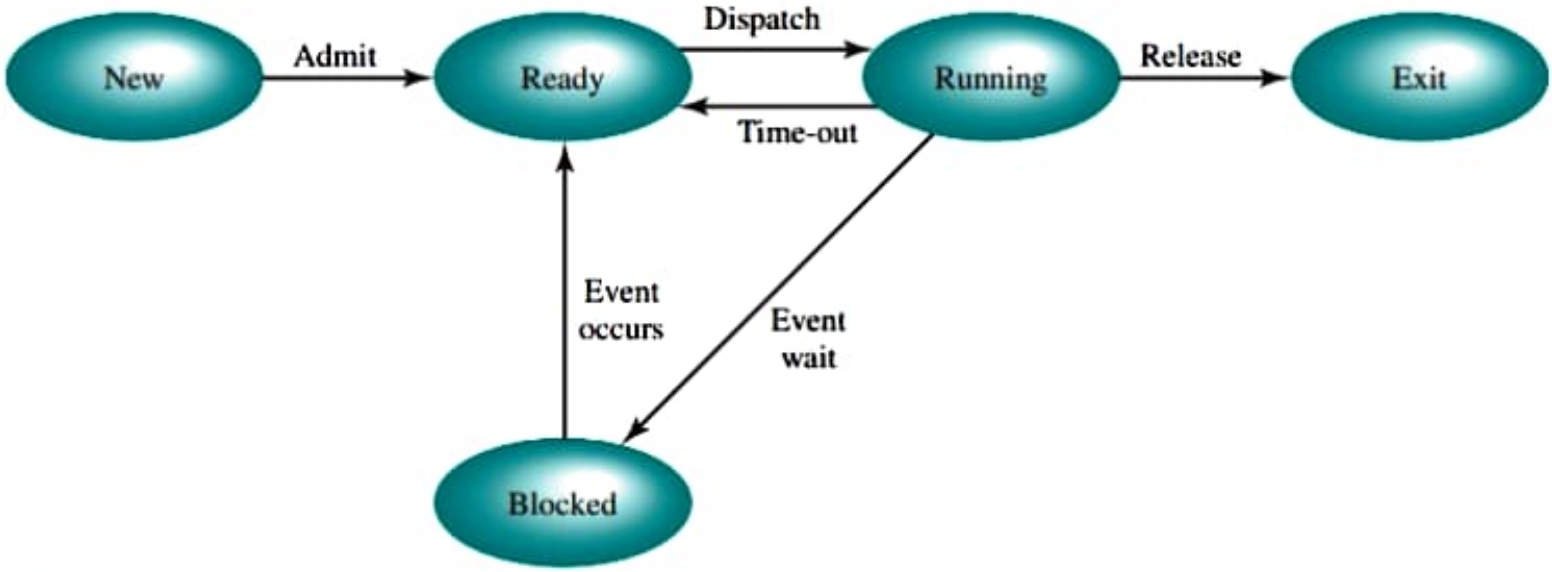
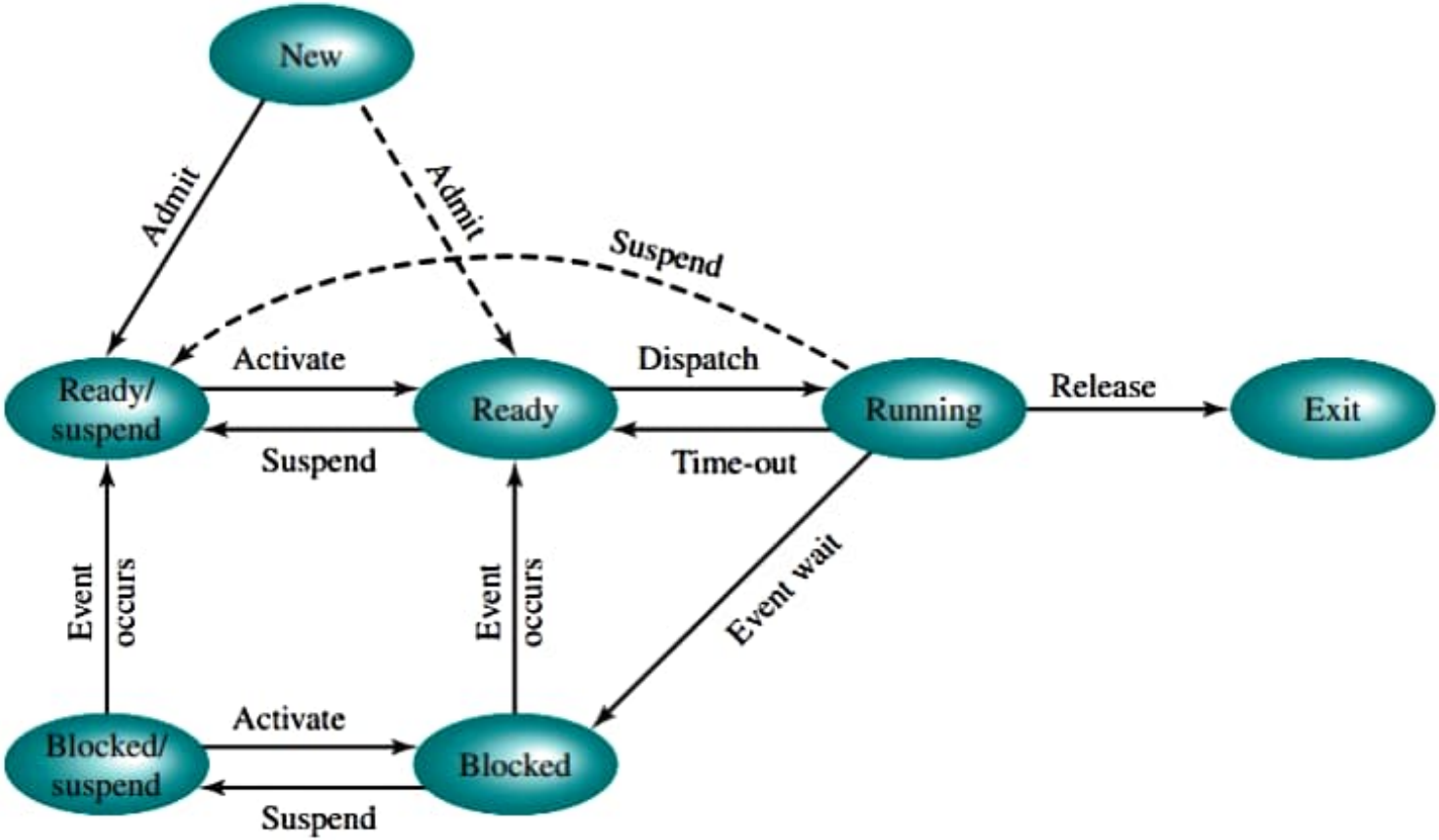


Figure 3.6 **Five-State Process Model**



(b) With two suspend states

Figure 3.9 **Process State Transition Diagram with Suspend States**

* Process Description wasn't in PPT so I ignored.

Process Control Block

Each process is represented in the OS using the process control block (PCB)

It contains the following information about the process:

a) Process State: Process may either be in ready, new, running, blocked, end state. This information is stored.

b) Program Counter: The program stores the address of the next instruction to be executed.

c) CPU - Scheduling information: Information about scheduling parameters like priority of process, pointers to scheduling queues, etc.

d) Memory - Management information: The information about page tables, segment tables, etc.

e) Accounting Information: Information about the CPU time used, time limits, proj process numbers, etc.

f) I/O status information: Information includes the list of I/O devices allocated to

the process, list of open files, etc.

Thus, the PCB serves as a repository for all information that are different from process to process.

Threads: Definition and Types

1) The process can be further divided into two separate & potentially independent components:

a) Resource Ownership: A process may be allocated control or ownership of resources, such as main memory, I/O devices, etc.
The OS needs to prevent other processes intervening by trying to access those resources.

b) Scheduling / Execution: This part of the OS takes care of the scheduling & execution of the processes. Eg: managing the process state of the process, dispatching a process into running state, etc.

- Resource Ownership: Process
- Scheduling / Execution: Thread / Lightweight Process.

2) Types of Threads:

a) User Level Threads (ULT)

b) Kernel Level Thread (KLT)

a) User Level Threads:

i) In pure ULT facility, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads.

ii) Any application can be ~~made~~ programmed to be multithreaded application using the threads library.

iii) The threads library contains codes for creation, deletion, data exchange, message exchange for threads.

→ Following is an example of how user level threads work:

i) Process X is running while thread A is running in that process.

ii) The OS blocks Process X when it requires an I/O input (puts in blocked state). But the thread A is still in running state.

iii) After some time, the OS puts the process X in ready state. Thread A is still running inside the process.

Note that in ii) and iii), the process itself

is not in running state, but the thread keeps running.

iv) The process X is then moved to running state and the execution continues. Now lets say thread A needs thread B to execute and give it some information

Thus thread A goes in Blocked state, while process is still running, & thread B ~~continues~~ continues execution.

{{{ Figure 4.6 Relationships between User-level threads & Process States }}}

Thus, the kernel continues execution as normal, moving process X from one state to the other. The user level threads inside the process execute separately & the kernel doesn't know about ULTs.

{{{ figure 4.5 (a) Pure user-level }}}

→ Advantages of using ~~KLTs~~ ULTs:

i) Thread switching does not require ~~us~~ kernel intervention, thus it ~~avoids~~ ~~cot conte~~ reduces overhead by avoiding context switching between user mode & kernel mode.

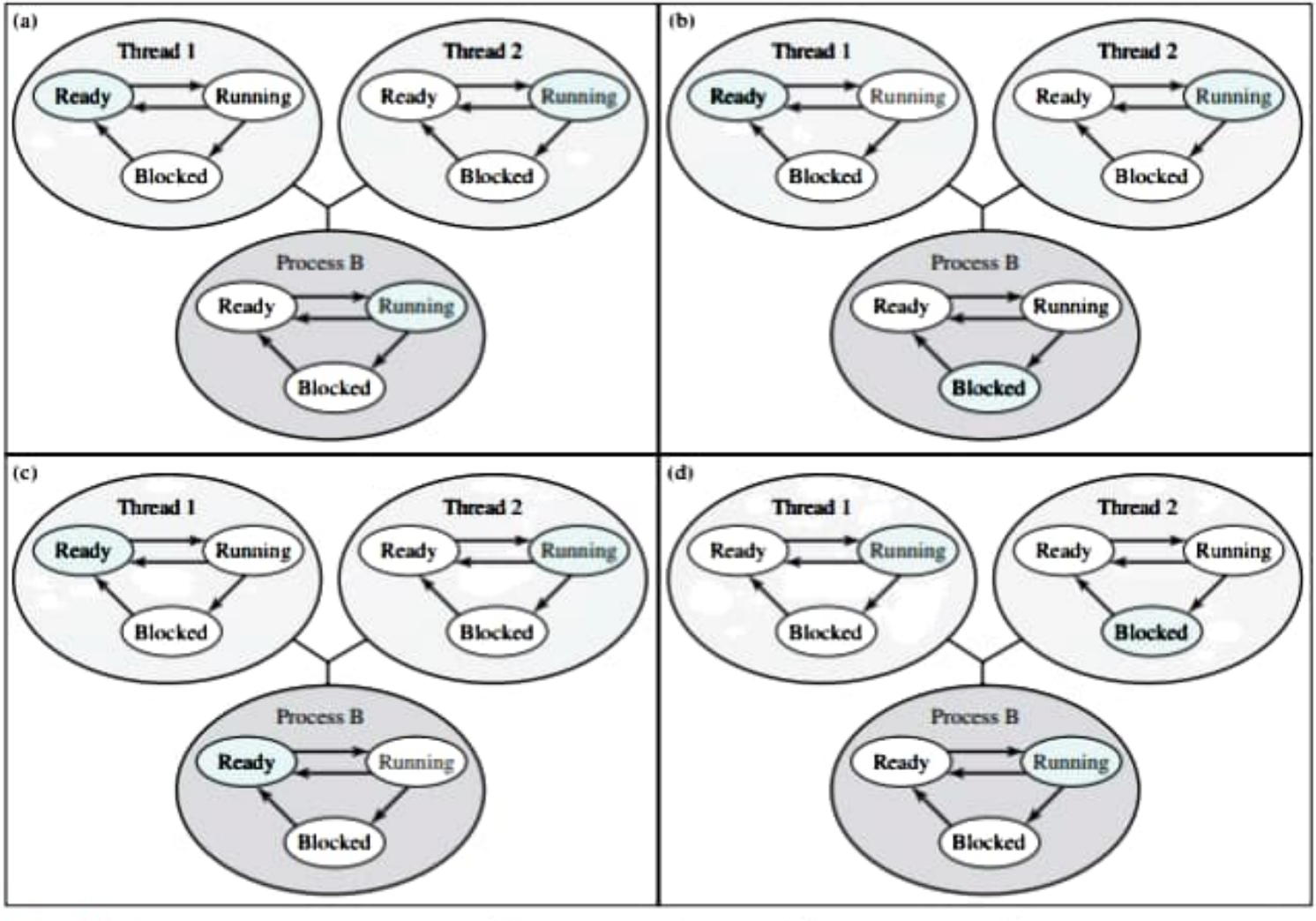
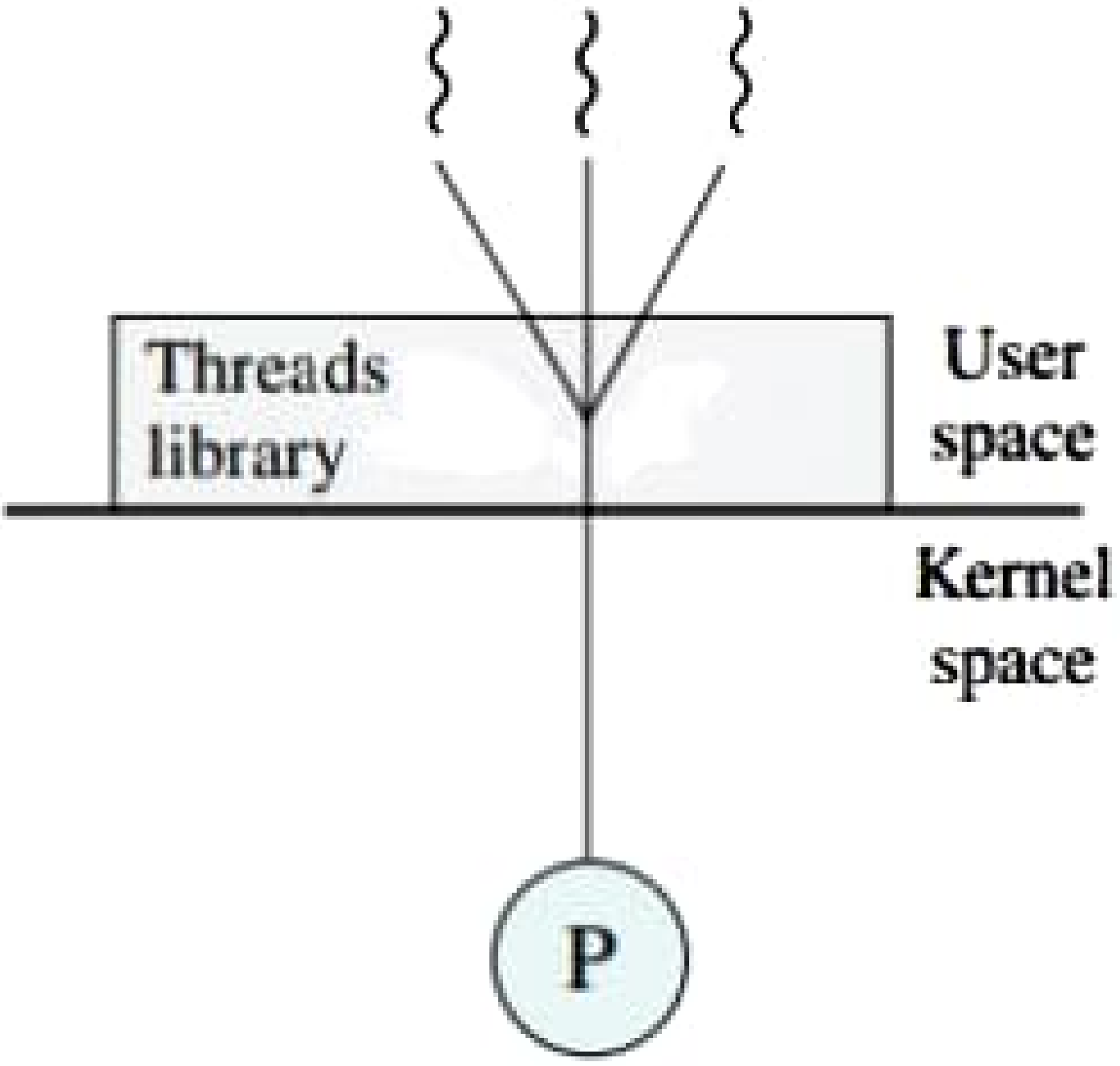


Figure 4.6 **Examples of the Relationships between User-Level Thread States and Process States**



(a) Pure user-level

ii) Different applications may have different scheduling methods that works best for them. Some may prefer first in first out & some priority scheduling.

The ULTs can thus customize the scheduling method according to the application without disturbing the underlying OS scheduler.

iii) ULTs can run on any OS as they do not require OS (kernel) support, since they are made on application level.

→ Disadvantages of ULTs:

i) Many OS system calls are blocking (blocks the process / thread if the I/O device isn't free, for eg.). Thus a thread may get blocked if it requests I/O input by the OS and this blocks all the threads because control was with that thread.

ii) In pure ULT, Multithreading applications cannot use the full extent of multi processing advantages, because each process is still assigned to only 1 processor.

→ Solution to i): Instead of calling the OS system call, the thread can use application level I/O 'jacket' routine. This jacket routine

checks if the I/O device is busy. If it is busy, the thread enters blocked state & gives control to a different thread. After control returns, the ~~kernel~~ jacket checks again.

b) Kernel Level Threads:

i) All of the work of thread management is done by the kernel. There's no thread management code in application level.

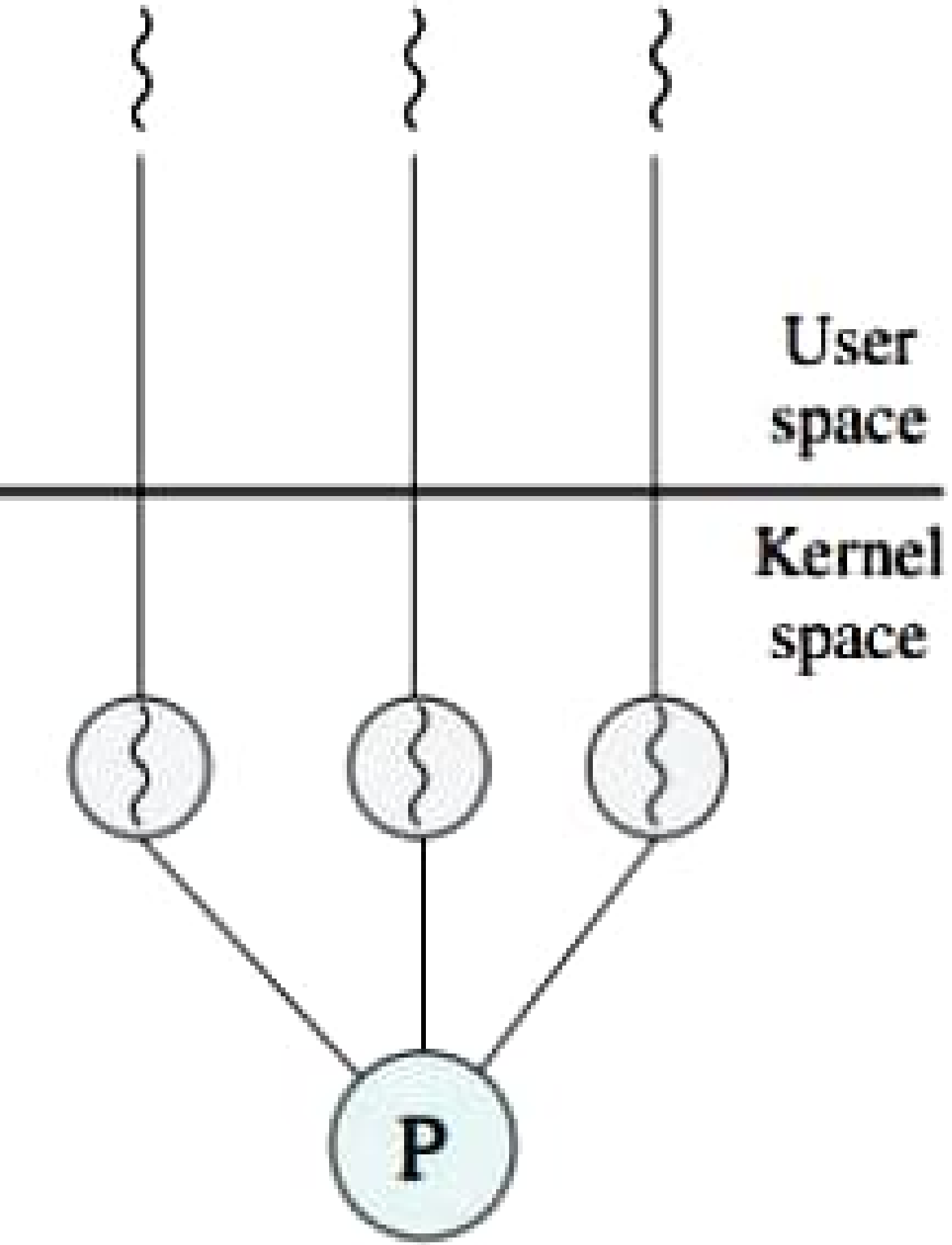
ii) Now, multiple threads can be assigned to a process on multiple processors.

iii) If one thread from the process is blocked, the kernel can itself give control to a different thread.

→ Disadvantage: Context switch has more overhead because the kernel mode & user ~~no~~ mode ~~are~~ ~~switched~~ mode switch is required more, ~~often~~ when threads switch control.

Depending on the requirements, either ULT or KLT can be used.

{{ Figure 4.5 (b) Pure kernel-level }}



(b) Pure kernel-level

Combined Approaches

i) Some Operating systems offer combination of ULT and KLT.

ii) Thread creation is done at the application level.
Bulk of synchronization and scheduling are also done within the application.

iii) Each ULTs is are mapped onto smaller of equal number of KLTs.

iv) The application programmer can adjust the number of KLTs required for their application.

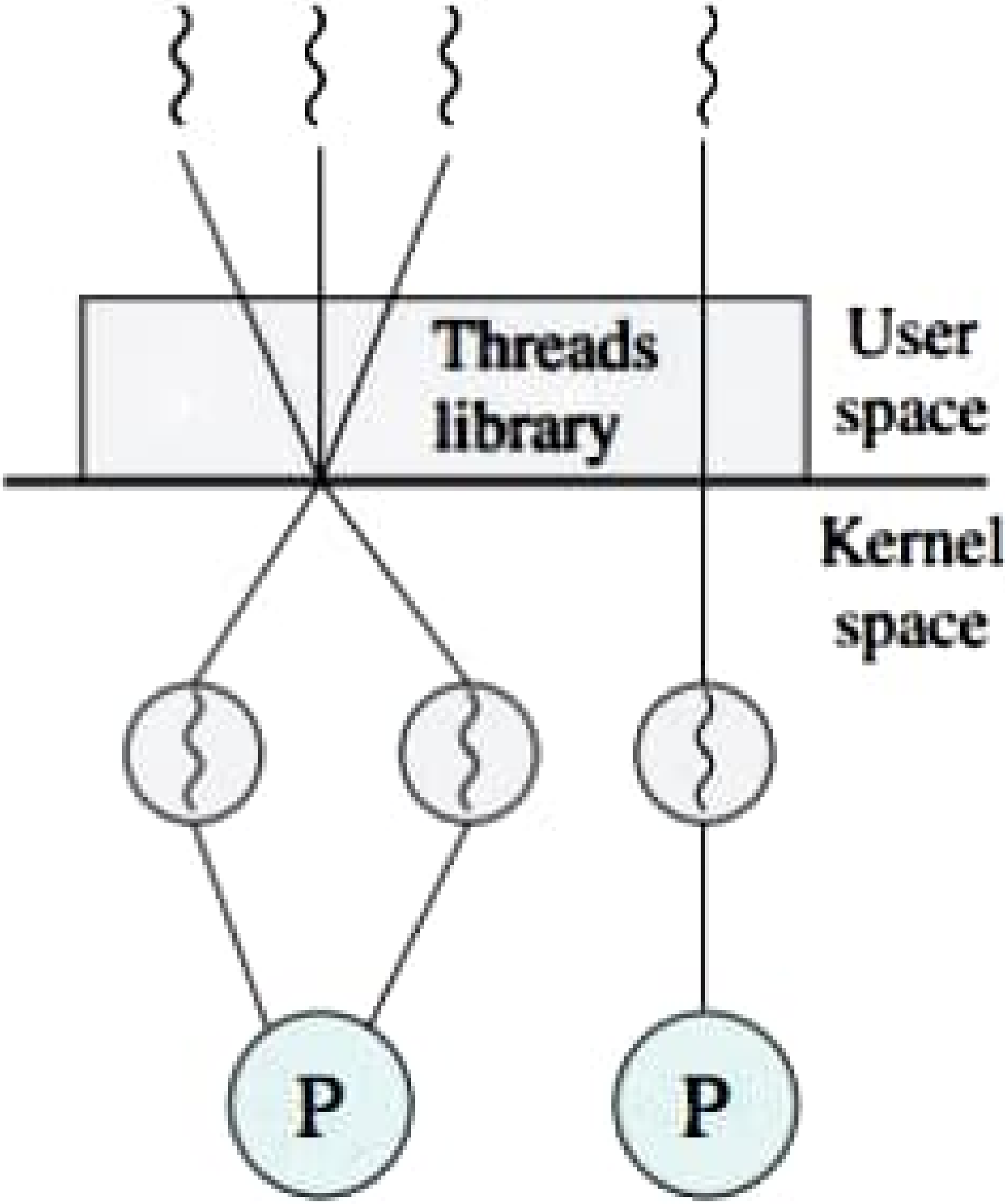
v) Thus process still runs on multiple processors. And a blocking system call does not block the entire process.

{{ figure 4.5 (c) Combined }}

Pure ULT → IDK
Pure KLT → Windows
Pure Combined (LOL) → Solaris.

Concept of Multithreading:

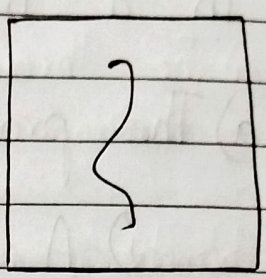
Multithreading refers to the ability of an OS to support multiple, concurrent paths of



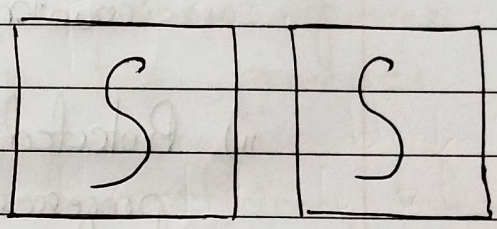
(c) Combined

execution within a single process.

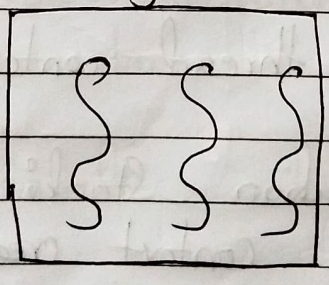
i) The earlier models did not recognize threads concept, thus they are called single-threaded approach.
Eg: MS-DOS.



ii) Some Process OS supported multiple processes, but each process still had only 1 thread. These also fall under single-thread approach.
Eg: UNIX systems

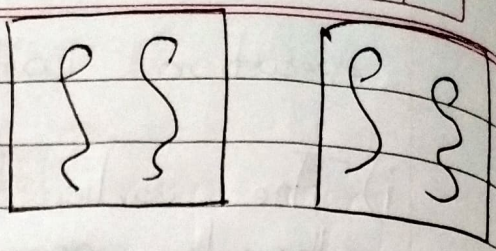


iii) Some operating systems supported multiple threads but only one process. They, thus, fall under the multithread-ed approach. Eg: Java run-time Envn



iv) Some OS support multiple processes along with multiple threads, thus, the fall under multithreaded approach. Eg: Windows, Solaris,

Modern UNIX, etc.



→ In a multithreaded environment.

a) The process is provides:

i) A virtual address space that holds the process image.

(Process image means contains the data section, text section, program counter, stack, heap).

ii) Protected access to processors, other processes (while inter-process communication), files & I/O resources.

b) The thread has:

i) A thread execution state (Ready, Running, etc.)

ii) A saved thread context when not running.

iii) An execution Stack: Used to store all the execution context created during the code execution.

iv) Some per-thread static storage for local variables.

v) Access to the memory and resources of its process, shared with all other threads in that process : (same memory pool among threads).

→ Advantages of threads:

i) Takes far less time to create a thread than to create a process (10x).
ii) Takes less time to terminate a process than a thread.

iii) It takes less time to switch between two threads within the same process than to switch between two processes.

iv) Threads enhance communication efficiency between processes.
Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel, while 2 independent processes require kernel intervention for communication and thus, are slower.

Multithreading Models

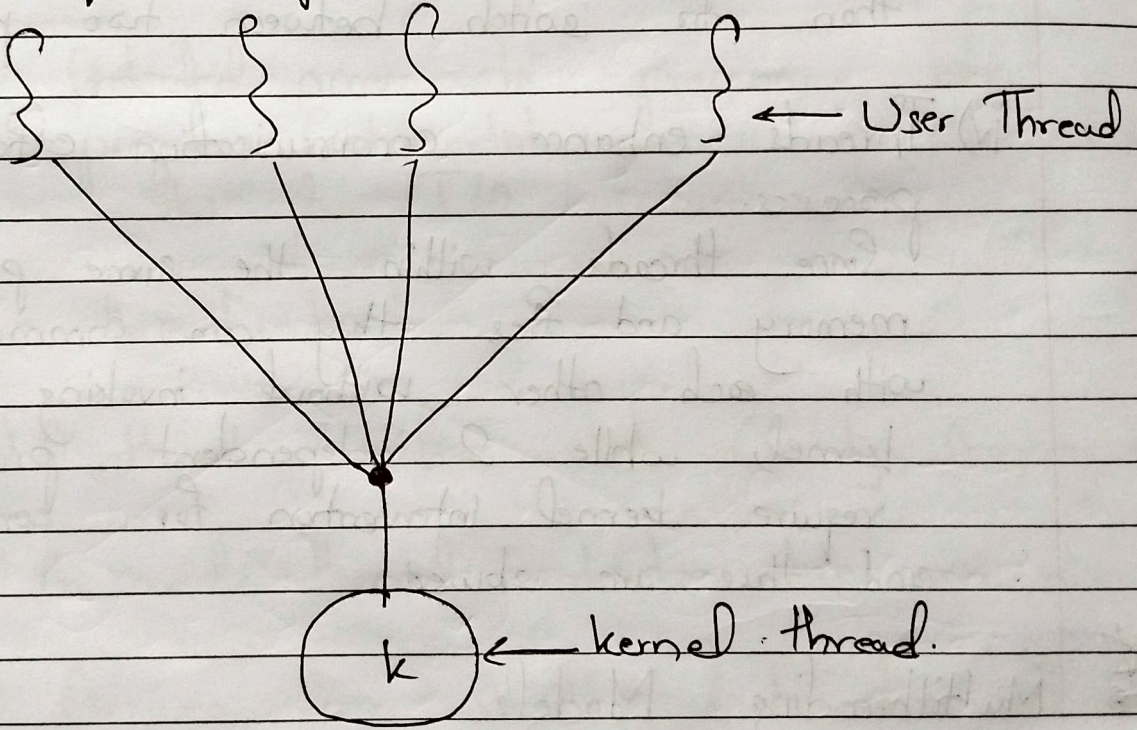
Many systems use the combined approach of using both ULTs and KLTs, thus, a relationship must exist between user threads and kernel threads.
Following are 3 common ways to establish this

relationship:

i) Many - to - One: This model maps many user-level threads to one kernel thread.

a) The thread management is done by the thread library in user space.

b) The entire process will block if 1 of the threads makes a blocking system call.
(Blocking system call: A system call that blocks the process execution until the requested operation is completed).

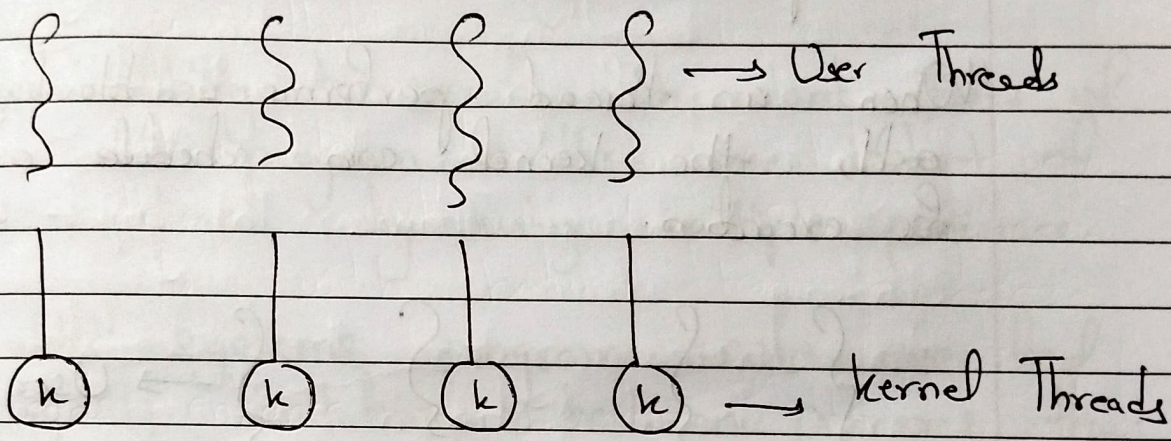


ii) One - to - One Model: Model maps each user thread to a kernel thread.

It provides more concurrency than many-to-one model because even if one

thread makes a blocking system call, the remaining can still run using their kernel threads, thus the entire process is not blocked.

The drawback is that for each user level thread, a corresponding kernel level thread has to be created, which is a task with great overhead.



iii) Many - to - ~~One~~ Many Model: Model Maps many ULTs to lesser of equal number of KLTs.

The number of KLTs may be specified by the application.

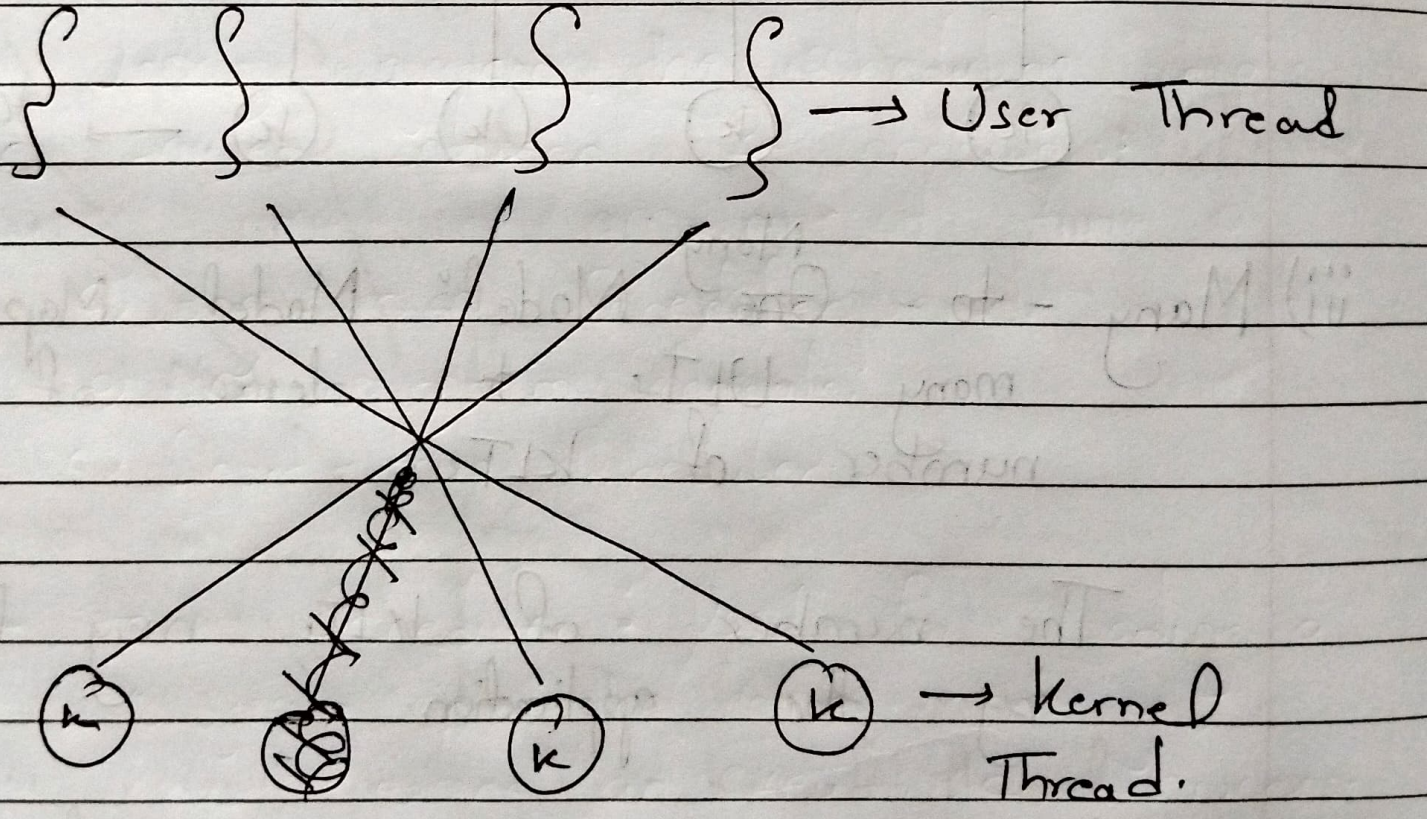
- The ~~Open~~ Many - to - One approach did not achieve true concurrency, because the no. of KLTs is always 1, thus only 1 thread can be scheduled at a time.
- The One - to - One approach provided great concurrency, but the programmer

has to be careful not to create too many ~~KLTs~~ ULTs (as they would create KLTs)

- The Many-to-Many suffers from neither of the above disadvantages.

In Many-to-Many, developers can create as many ULTs as needed.
The corresponding KLTs will run parallel on the processors.

When a thread performs a blocking system call, the kernel can schedule another thread for execution.



2.2

1) → Multicore Processors and threads

- Multicore : Multiple Processors / Computing Core are place inside a single chip.
- Concurrency: Ability of different parts of program to be executed out of order without affecting the outcome.
- Parallelism: Ability to execute independant tasks of a program in the same instant of time by using multiple processors.

In older systems, programs were executed by rapidly switching between processes, thus allowing multiple processes to progress at the same time.
This system was concurrent but not parallel.

→ Types of Parallelism:

a) Data Parallelism: Same data is given to both cores and same operation has to be performed on both cores.

Eg: find the sum of 1 to n.
Thread Core A finds sum from 1 to $n/2$
Thread Core B finds sum from $n/2$ to n.

b) Task Parallelism: Distribution of task among different cores.

Eg: One thread performs a sorting task, while the other thread performs a searching task.

2) ~~Scheduler~~ Scheduling: Uniprocessor Scheduling

Types of Scheduling:

The aim of processor scheduling is to assign process so that they ~~are~~ have ideal response time, throughput & processor efficiency.

In many system, the scheduling responsibilities are shared between 3 schedulers:

a) Long-term Scheduler
b) Medium-term Scheduler
c) Short-term Scheduler.

a) Long-term Scheduler: Used for loading a program into the main memory, thus creating an executable process.

These processes are thus active.

b) Medium-time Scheduler: The purpose of the medium-term scheduler is to manage the swapping ~~for~~ of process from the suspended state (which is

only partially in the main memory) into the main memory fully so that they are ready for execution (Ready Queue).

This partial storage in main memory is possible only using Virtual Memory.

c) Short term Scheduling:

Executes more frequently than long-term & medium term scheduler. It decides which process to execute next.
The short term scheduler invokes when the system tells that blocking of current process & running another process is favorable, after some event has occurred.

This event can be:
i) Clock interrupts
ii) I/O interrupts
iii) Operating System Calls
iv) Signals.

3) → Scheduling Algorithms:

- Non-Preemptive: Once a process in the running state, it continues to execute until it terminates or it blocks itself to wait for I/O or to request some OS service. This process cannot be stopped from execution otherwise.

- Preemptive: The currently executing process ~~in the running state~~ may be interrupted and moved to the ready state by the OS. This could happen when a new more higher priority enters, or when the maximum computation time for a single process is exceeded, etc.

- Turnaround Time: The amount of time between the submission of a process and its completion

- Response Time: The amount of time between submission of a request and receiving the response.

- Waiting Time: The amount of time a ~~processor~~ process spends in the waiting queue.

Scheduling Algorithms:

a) First-Come-First-Serve (FCFS):
Also known as First-In-First-Out or strict queuing ~~theme~~. scheme.

As each process gets ready, they join the ready queue. When currently running process stops execution, the next job starts executing.

Process	Arrival Time	Service Time (T_s)	Start Time	Finish Time	Turnaround Time (T_r)	T_r/T_s
W	0	1	0	1	1	1
X	1	100	1	101	100	1
Y	2	1	101	102	100	100
Z	3	100	102	202	199	1.99
Mean					100	26

Disadvantages of FCFS:

{{Table on page 429}}

i) Some short processes may join the queue after a bigger process has joined, increasing its turnaround time ridiculously, for just a small CPU time required.
Eg: Y has $T_r/T_s = 100$, meaning it is in the system for 100x time more than it required.

ii) FCFS favors processor bound processes over I/O bound processes.
When a processor bound process is executing, there may be several I/O bound processes waiting in queue and during this time, the I/O module may be sitting idle.
This is very inefficient indeed.

b) Shortest Job/ Process Next:

A non-preemptive policy in which the process with the shortest expected processing time is selected next.
Thus, a short process will jump ahead of the queue past longer jobs.

Disadvantage:

i) Variability of response time has increased.
ii) We need to know or at least estimate the required processing time of each process
iii) Risk of Starvation.

c) Shortest Remaining Time:
(SRTN or SRTF)

It is the preemptive version of SPN.
The OS always choose the process that has the shortest remaining ~~time~~ processing time.
If a new process with shorter remaining time than currently executing process enters, the current process is preempted.

- There is a risk of starvation in SRT.
- Elapsed service times must be recorded, contributing to overhead.

Gives better turnaround time than SPN, because shorter jobs are immediately executed.

d) Round Robin:

Uses preemption based on clock.
The clock interrupt is generated at a regular interval. When the interrupt occurs, the currently executing process is placed in the ready queue & the next job is selected on an FCFS basis.

Also called time slicing.

Disadvantage:

Processor bound processes get more better portion of processor time than I/O bound process. This is because when I/O bound processes start executing, they may be done with the ~~process~~ needed processor computation required before requesting an I/O access, thus blocking that process, wasting the remaining CPU time it had. Whereas the processor bound process utilizes the time fully.

c) Multilevel Queue Scheduler:

A type of data Structure.

Main Task — Assign processor so that starvation is avoided.

↓

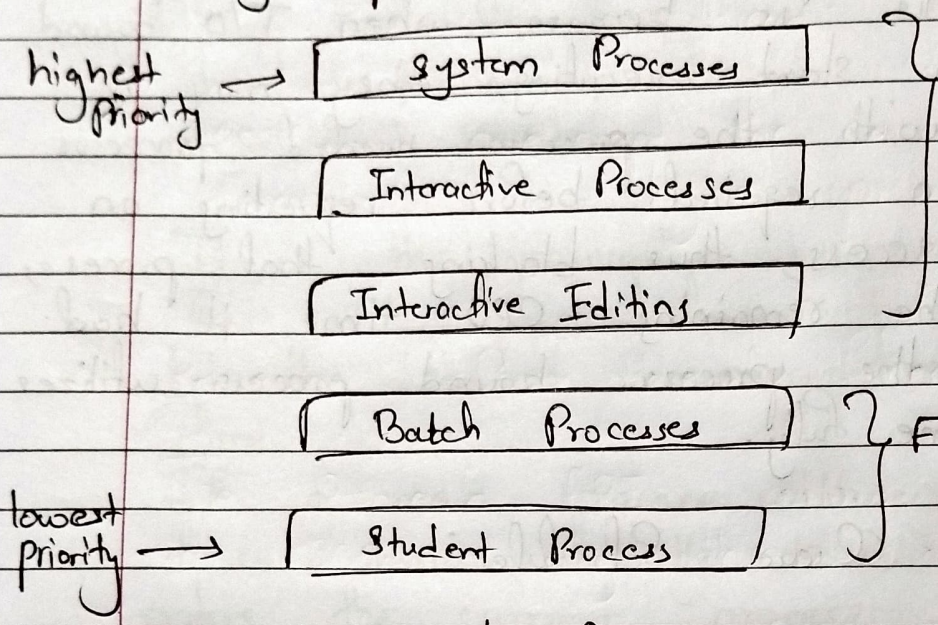
Multilevel Queue, Feedback Multilevel Queue.

Foreground — Interactive tasks
(Uses Round Robin)
Eg: keyboard input

Background — Processor Bound task
(Uses FCFS)

- Foreground process have higher priority than background process.

→ Divided into 5 queues
The purpose of this division is to reduce the overhead of searching for ready processes.



Prone to starvation

Multilevel feedback Queue

A process can move between layers.
A job is first sent in first layer, and the CPU burst used is calculated. As time passes the process is pushed below if it doesn't complete execution in ~~that~~ ~~ti~~ the limited it is pushed downwards till the FCFS level.

But to prevent starvation, each process ~~an~~ below top priority is given a timestamp. According to the aging of a process it is pulled back Up.

23 Introduction to Thread Scheduling:

On operating Systems that support user and kernel level threads, It is the kernel level threads that are being scheduled and not the processes themselves.
To run on a CPU, the user level threads must ultimately be mapped to a kernel level thread. This can be done indirectly via a Light Weight Process (LWP).
We now explore scheduling issues involving user and kernel level threads.

a) Contention Scope:

i) Process Contention scope: (PCS)

On systems with many-to-one and many-to-many models, the threads library schedules a ULT to run on an available LWP.

Known as PCS because competition for the CPU takes place among the threads belonging to the same process.

ii) System Contention Scope (SCS):

To decide which KLT to schedule onto a CPU. Competition for the CPU takes place among all the threads in the system.

Typically, PCS is done based on priority. The thread with the highest pridrity is selected to run.

The priority is not set by the threads library, instead the programmer selects the priority of a thread.

PCS will pre-empt the currently executing thread for a higher priority thread.

b) Pthread (Posix thread) Scheduling:

The Pthread API allows specifying PCS or SCS during thread creation.

The API allows specifying:

- PTHREAD_SCOPE_PROCESS: Schedules threads using PCS scheduling.
- PTHREAD_SCOPE_SYSTEM: Schedules threads using SCS scheduling.

On many-to-many models, PTHREAD_SCOPE_PROCESS schedules user-level thread onto a LWP.

The PTHREAD_SCOPE_SYSTEM policy will create & bind an LWP to a ULT, creating a one-to-one

policy, even on many-to-many systems.

** There are 2 main functions used by the Pthread IPC, for getting & setting the contention scope:

- pthread_attr_setscope (pthread_attr_t * attr, int scope);

i) The first parameter of the function contains a pointer to the attribute set of the thread, this set can include information like the scheduling policy, the size, priority, etc (of the thread).

ii) The second parameter of the function is the contention policy (PTHREAD_SCOPE_PROCESS or PTHREAD_SCOPE_SYSTEM) that is to be assigned to the thread.

- pthread_attr_getscope (pthread_attr_t * attr, int scope);

i) The first para meter holds same info as the getscope function.

ii) The second ~~parmet~~ parameter holds the pointer to the int

value that is set to the contension scope of the thread. This integer will return 0 in case of an error.

* Contension Scope is referring to whether it is system or process (PTHREAD scope)

2.4 Linux Scheduling

a) Before version 2.5 the Linux kernel ran on a variation of UNIX scheduling algorithm. But this algorithm wasn't written with multiple processors in mind, thus giving poor performance in multiprocessor systems.

b) In version 2.5, a new algorithm was introduced which was known as O(1), which, unsuprisingly, gave constant time complexity no matter the number of tasks. O(1) scheduler also provided increased support for multiprocessor systems. Processor affinity & Load Balancing between processors.

c) However, the O(1) scheduling algorithm led to poor response times for interactive processes. Thus, in 2.6, a new scheduling algorithm was introduced: Completely Fair Scheduler (CFS).

Processor Affinity:

A process has an affinity towards the processor it is currently running on (in multiprocessor system) and doesn't migrate to a different processor in between.

This is done because the cache that is created in a processor while executing a process has to be invalidated when a process migrates from that processor and the processor to which this process has migrated to has to repopulate its cache for the new process. This migration is thus extremely costly.

CFS:

The Linux CFS scheduler is an efficient algorithm to select which task is to be executed next. The tasks are arranged in an RBT based on the value of the runtime executed for every task.

Thus the task with the smallest executed runtime is found at the leftmost node of the tree. Thus to find the task with the smallest executed runtime, the no. of operations required is O(log n), but the scheduler caches a pointer pointing

to the leftmost value node to save up time.

d) Scheduling in Linux systems are based on scheduling classes. Each scheduling class has a different priority and tasks are allotted to these different scheduling classes.

The scheduler chooses the highest priority task from the highest priority class to execute.

There are 2 basic scheduling classes:

i) Default Scheduling Class - CFS is used
ii) Real time scheduling.

I believe that real-time scheduling class would get higher priority, but it depends on system to system.

e) i) Nice Value: Each task is assigned a 'nice value' which ranges from -20 to +19. The lower the 'nice value', the higher the priority of task.

ii) The CFS scheduler assigns a proportion of the CPU to each process / task. The higher the priority of task (or lower the nice value), the greater the CPU time

allotted will be.

ii) Targeted Latency: This is the interval of time during which each runnable task should execute at least once. This period increases or decreases based on the number of tasks in the system.

f) Why is it Completely fair?

Let's take an example of 2 processes: 1 I/O bound and another CPU bound.

The I/O bound process will use short CPU usage & get pre-empted when it needs an I/O access. Now the CPU proce bound process will run & get its job done ~~in that~~ almost completely.

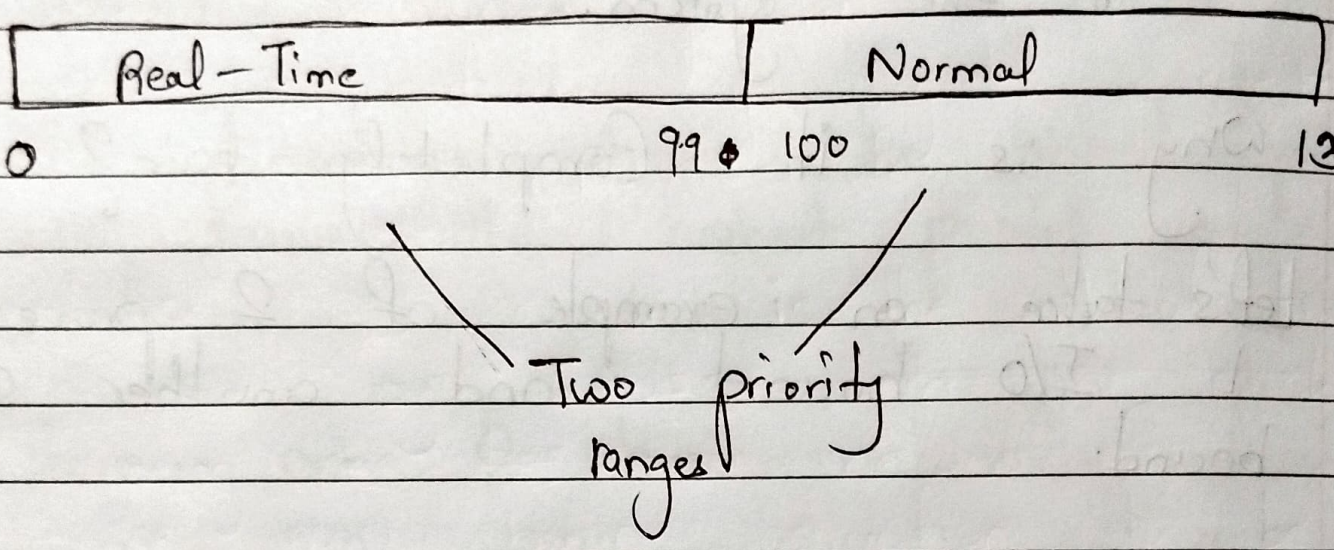
Now the I/O bound process is ready to execute again, but it might have more CPU time remaining required than the processor bound.

The CFS, as mentioned before, keeps track of the amount of CPU time a task has used. The lesser the ~~runn~~ executed time, the higher the priority. Thus in the above scenario, the

I/O bound process will pre-empt the pro CPU bound process.

Thus both the types of processes will be done executing at around the same time, aint that fair!

g)



0-99: Real time tasks
100-139: Normal tasks

Real time tasks have static priority within 0 to 99.
Normal tasks have dynamic priority based on the nice value, ranging from (100-139)