Chapter 2: Processes

Sukomal Pal, CSE, IIT(BHU)

Chapter 2. Processes, Threads and their Scheduling

- Processes: Definition, Process Relationship, Different states of a Process, Process State transitions, Process Control Block (PCB), Context switching
- Thread: Definition, Various states, Benefits of threads, Types of threads, Concept of multithreads
- Process Scheduling: Foundation and Scheduling objectives, Types of Schedulers
- Scheduling criteria: CPU utilization, Throughput, Turnaround Time, Waiting Time, Response Time
- Scheduling algorithms: Pre-emptive and Non pre-emptive, FCFS, SJF, RR;
- Multiprocessor scheduling

PROGRAMS and PROCESSES

- A program is a set of instructions that need to be executed on the processor or CPU of the computer.
- When the program is not run, it is a passive entity.
- From the point of view of an OS, a program is any *executable* file (a.out in UNIX or .exe in Windows).
- *source codes* like .c or .java files which are written in high-level languages and therefore not executable and not programs
- When a program is executed, it becomes a process.
- *A process is a program in execution.* It is an active entity and dynamically changes.
- Each process holds some attributes assigned by the OS as follows.
  - **Process-id:** a process identifier (pid).
  - **User-id:** the process is owned by a specific user (uid).
  - **Process Group-id:** Every process is supposed to belong to a group, based on the task. The group has a process group identifier (pgid).
  - **Address space:** main memory space (known as *process address space*) where it stores

PROGRAMS and PROCESSES

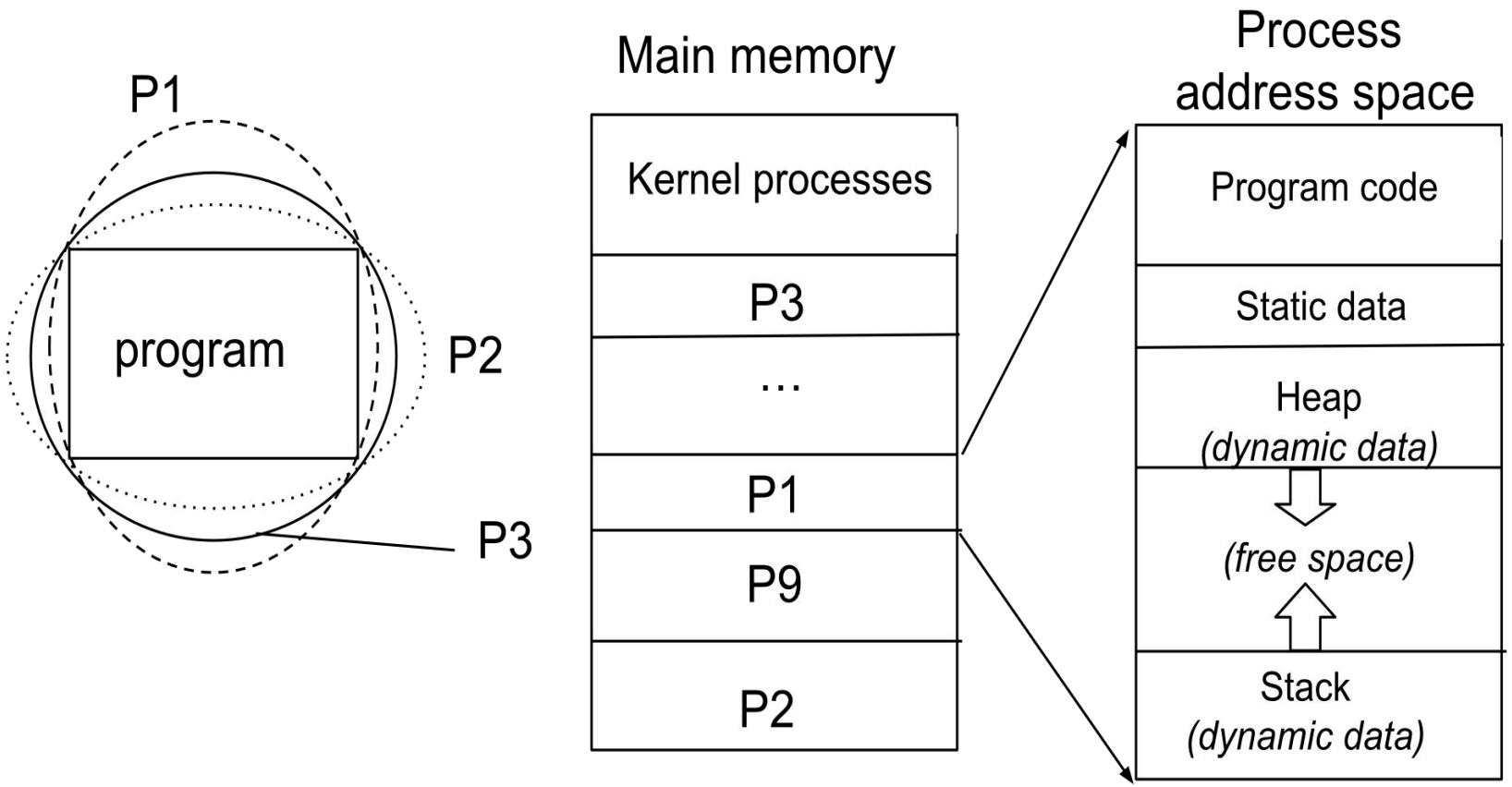


Fig 2.1: Program having 3 processes running with other processes

Process Address Space

- Memory allocated to a process when a program executes.
- **Text Section:**
  - Stores the **executable** program code.
  - Read-only section to prevent accidental modification.
- **Data Section:**
  - Stores **global and static variables**.
  - Further classified into:
  - **Static Data:** Bound at compile time.
  - **Dynamic Data:** Allocated at runtime.
- **Heap Section:**
  - Used for **dynamic memory allocation** (malloc() in C).
  - Can **grow or shrink** based on program execution.
- **Stack Section:**
  - Stores **function call-related data** (arguments, local variables, return addresses).
  - Grows **downward** as new function calls are made.
  - Shrinks when functions return.

Process Address Space

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

const int x=5;
int y = 15;
int z;


int main(int argc, char *argv[])
{
   int *val;
   int i;

   val = (int *)malloc(sizeof(int)*10);

   for(i = 0; i < 10; i++)
      val[i] = i;

   return 0;
}
```

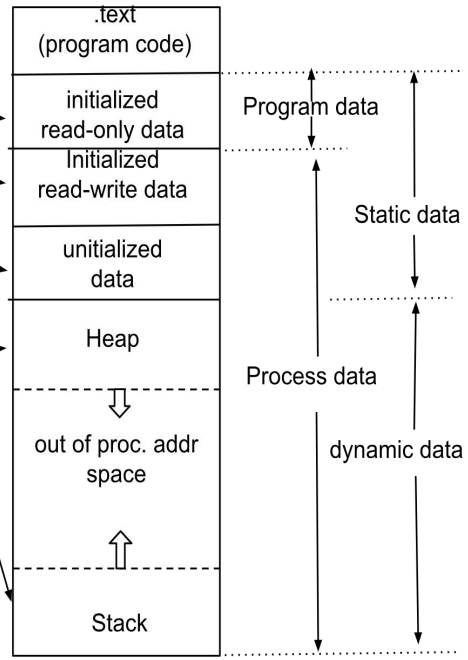


Fig 2.2: Process Address space

```
a.out  :
section                size     addr
…
.text                   437     4192
…
.rodata                   8     8192
…
.data                    20    16384
…
Total                  2265
```

The given example C-source code (Fig 2.2), when compiled using GNU C compiler

PROCESS RELATIONSHIP

- All processes in an OS are created by another process.
- **Exception**: The first process (init or systemd), which is created during bootstrapping.

- Each process has a number of links. It has a link to its parent process, its own children and sibling processes.
- **Parent Process**: The process that creates a new process.
- **Child Process**: A process created by another process.
- **Sibling Processes**: Processes that share the same parent

PROCESS RELATIONSHIP

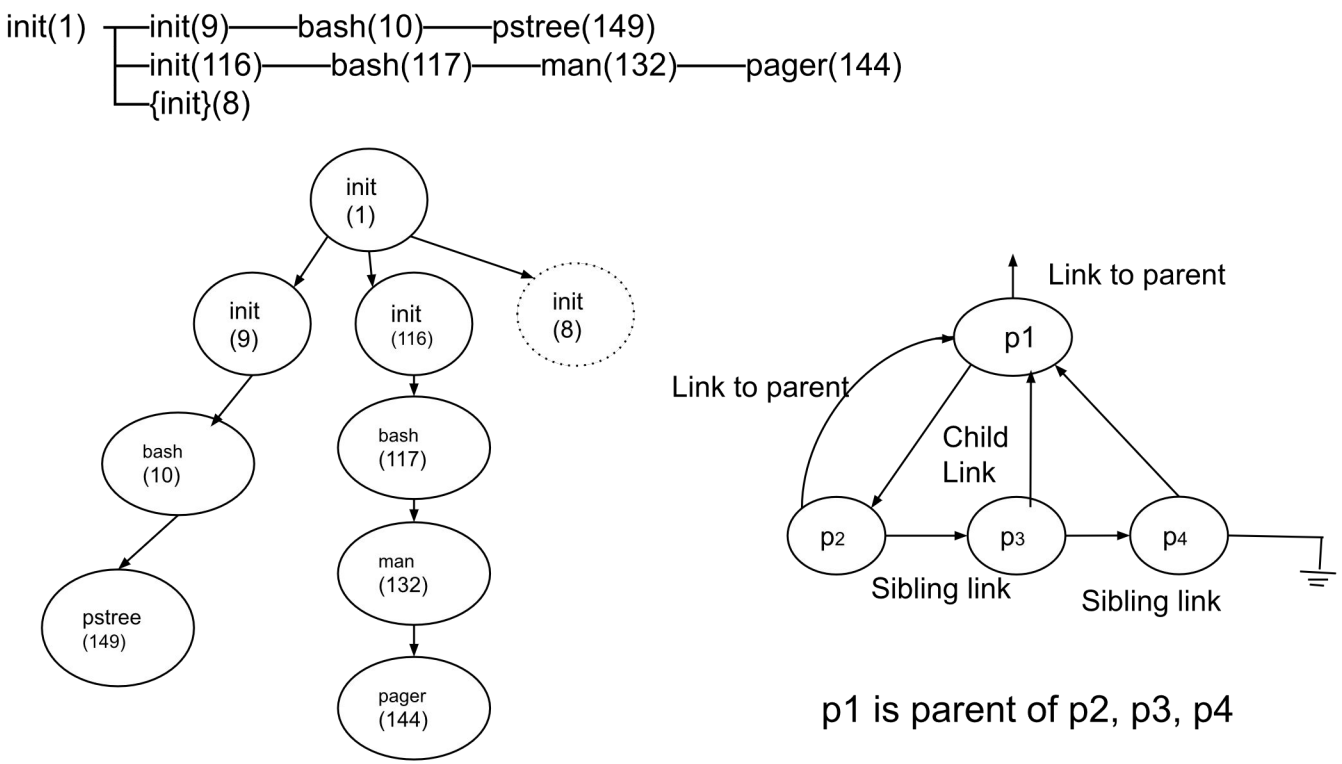


Fig 2.3: Process Tree

Child processes

- A process can create another process using a system call.
- A fork () creates a new process (child) as a duplicate of the calling process (parent).
- On success, it returns two different integer values:
  - The parent gets the child's process-id
  - The child gets a value zero (0).
  - On failure, no child process is created and the calling process gets a negative return value.
- getpid() and getppid() return process id and parent's process-id to the calling process respectively.

Child processes

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

int main(){
int pr_id;
if ((pr_id = fork()) == 0)
     printf("\nfrom child with pid=%d, ppid=%d and fork-return
value=%d\n", getpid(), getppid(), pr_id);

else if (pr_id >0)
     printf("\nfrom parent with pid=%d, ppid=%d and fork-return
value=%d\n", getpid(), getppid(),pr_id);

else
     printf("\nerror in in fork pid=%d, ppid=%d and fork-return
value=%d\n", getpid(),getppid(), pr_id);
}
```

The results obtained

```
from parent with pid=264, ppid=10 and fork-return value=265
from child with pid=265, ppid=264 and fork-return value=0
```

Fig 2.4: Example of process creation

PROCESS STATES and their TRANSITIONS

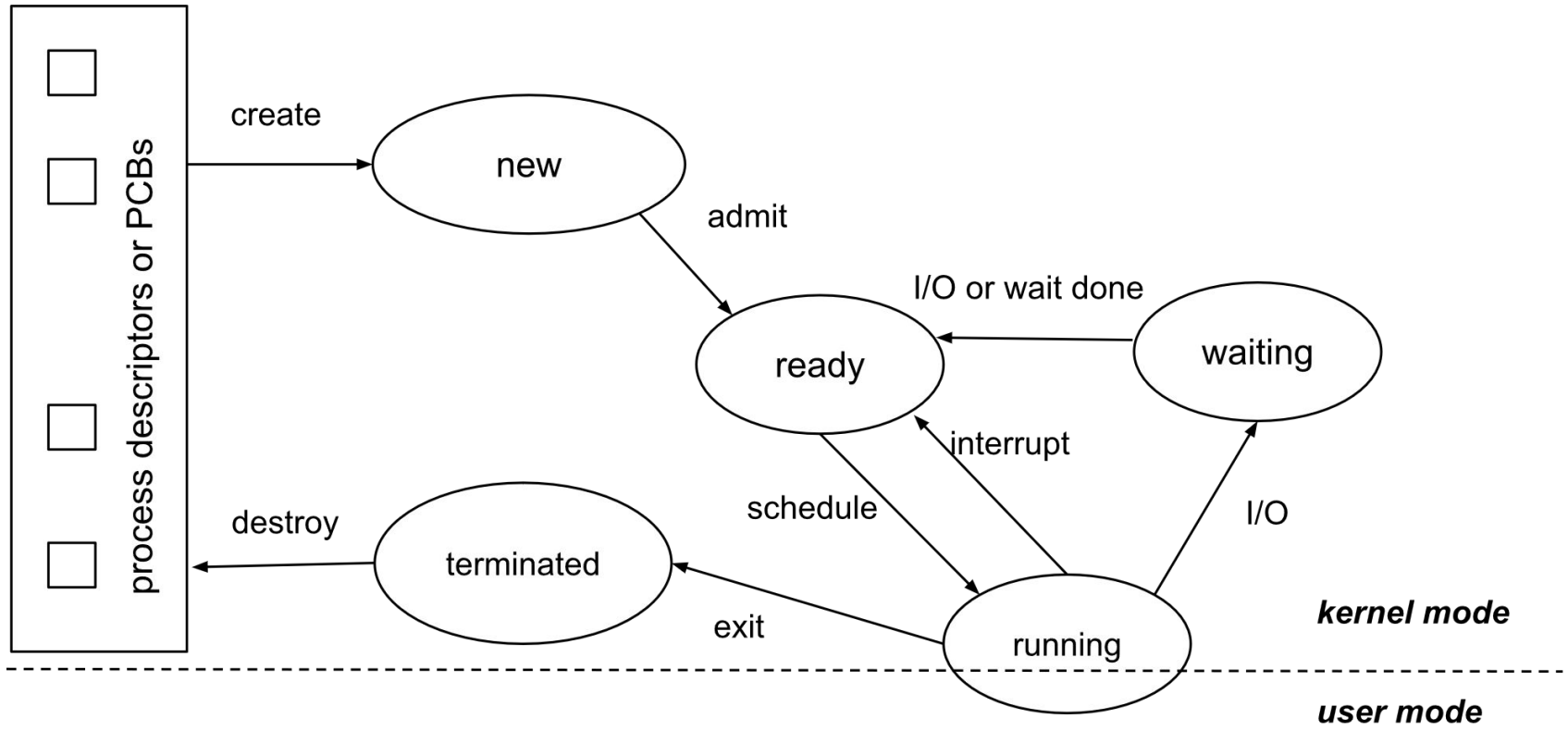


Fig 2.5: Process states(Life cycle of Process)

PROCESS CONTROL BLOCK (PCB)

- The OS kernel maintains a special data-structure called process control block (PCB) or process descriptor in its kernel space for each live process.
- This is a per-process data structure that stores the context of a process.
- The PCB has a number of attributes.
  - **Process id:** a unique identifier.
  - **User id:** The owner of the process.
  - **Process state:** Kept track of that can be new, ready, running, waiting, terminated etc.
  - **Scheduling information:** Process priority, pointers to scheduling queues, and other scheduling parameters need to be maintained.
  - **Memory-management information:** Value of the base and limit registers and the page tables, or segment tables.
  - **Accounting information:** Amount of CPU time used, wait time, time limits etc.
  - **Software context:**
  - **Hardware context:** Program counter, Stack pointer, Other CPU registers, I/O devices.
  - **Pointers to different data structures:**

Process Table

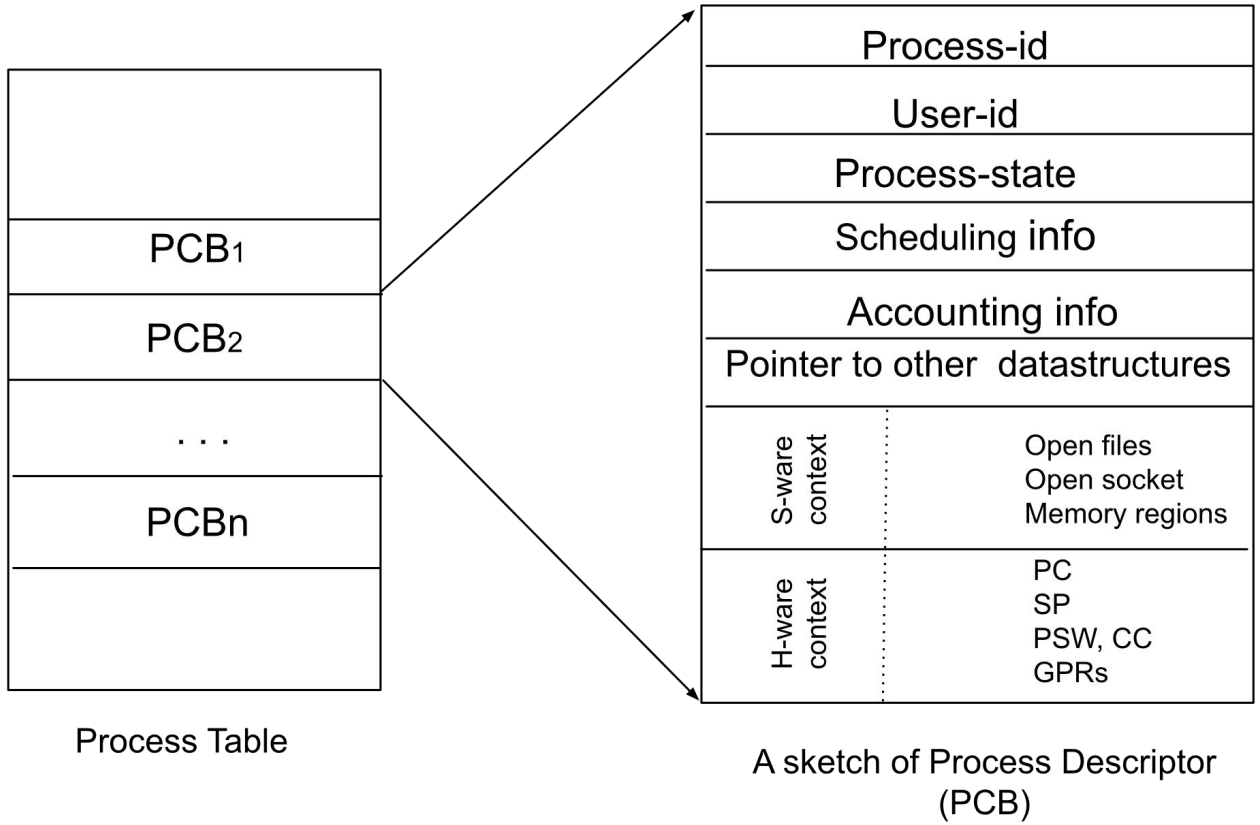


Fig 2.6: Process table and PCB

CONTEXT SWITCH

- When the CPU is changed from one process to another, the context of the first process is saved and that of the second process is loaded into appropriate registers and other data structures.
- We call this context switching or process switching.
- **Who causes the context switch and when?**
  1. **Interrupts (Asynchronous Events)**
     - **Timer Interrupt:** Occurs when the time slice of a running process expires.
     - **I/O Interrupt:** Triggered when an I/O task completes, requiring CPU attention.
     - If an interrupt is unblocked, the CPU switches to a kernel process to handle it.
  2. **System Calls (Synchronous Events)**
     - Happens when a process requests **privileged operations** (e.g., accessing I/O, memory).
     - Context of the running process is saved, and a suitable kernel process is executed to meet the requirement.
     - **Interrupts** come from **external devices** to the CPU.
     - **System Calls** originate from **running processes** to the OS.
  3. **Traps / Exceptions (Error Handling)**
     - Occurs when a process encounters an **error or illegal operation.**
     - The OS **switches to a kernel process** to handle the trap.

CONTEXT SWITCH

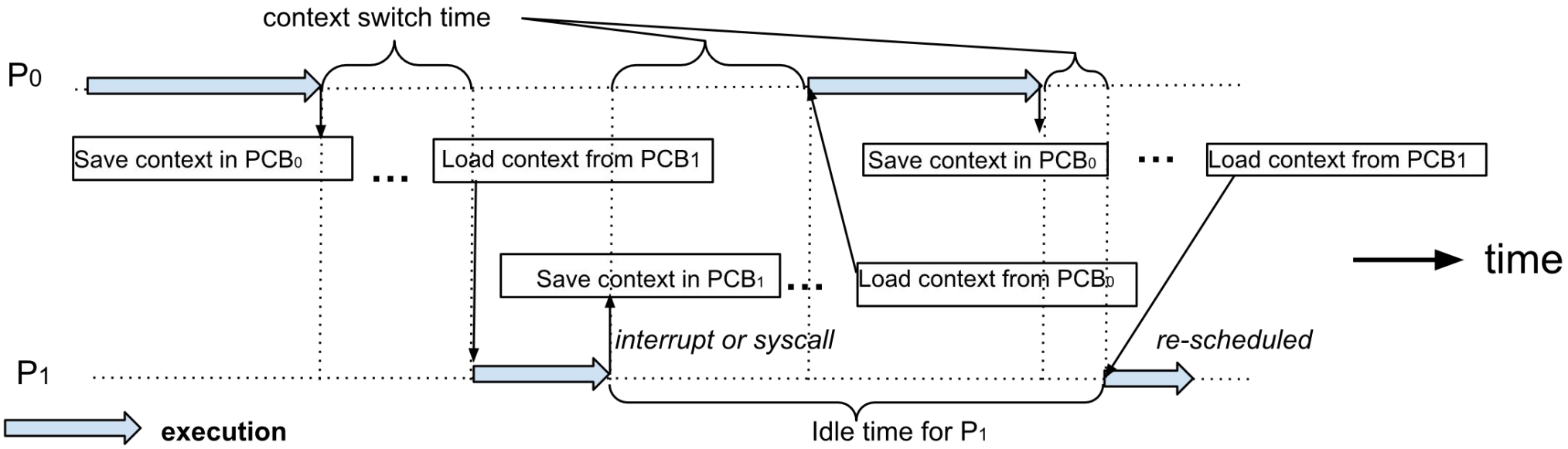


Fig 2.7: Context switching between Two Processes

THREADS

- A thread is a single flow of execution and the basic unit of CPU utilization.
- A process can have one or more threads.
- Threads share code, data, and resources but have their own stack and registers.

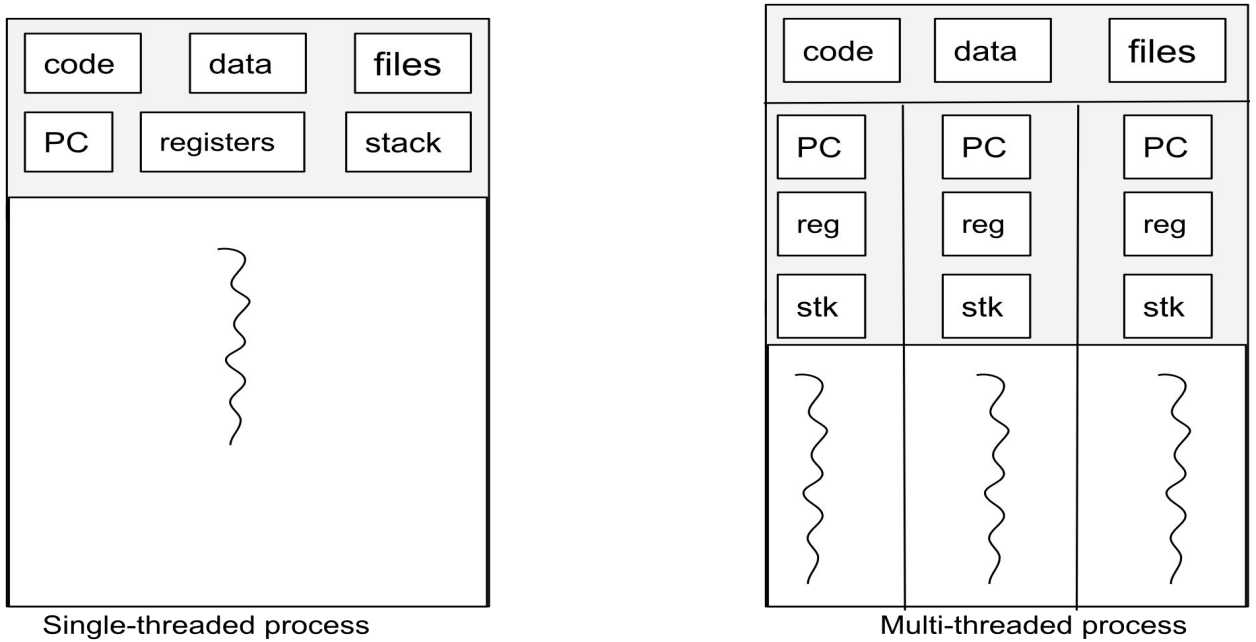


Fig 2.8: Relation between process and Thread

Thread States

- **Thread States:**
  1. **Ready:** Thread is prepared but waiting for CPU.
  2. **Running:** Thread executes instructions.
  3. **Blocked:** Thread is waiting for an event.
- **Thread Operations:**
  1. **Spawn:** A process creates threads.
  2. **Block:** A thread waits for an event.
  3. **Unblock:** A waiting thread moves to Ready state.
  4. **Finish:** Thread execution completes.

Pros and Cons of Threads

- **Advantages**:
  - Improved Performance: Tasks run in parallel, reducing wait times.
  - Resource Sharing: Threads share memory/resources efficiently.
  - Low Cost: Thread creation is 10x faster than process creation.
  - Scalability: Threads can execute on multiple CPU cores.
- **Disadvantages:**
  - Increased Stack Space Usage: Each thread needs a separate stack.
  - Increased Complexity: Debugging multi-threaded programs is difficult.

Types of Threads

1.User-Level Threads (ULTs):

- User level threads exist in the user space.
- The kernel may not be aware of the ULTs.
- When the OS does not inherently support multi-threading, ULTs are managed by the threads library in the user space only.

2. Kernel-Level Threads (KLTs):

- All threads are managed by the OS kernel itself and there is no thread management necessary at the application level.

3. Mixed Approach (Hybrid Model):

- The combination of both ULTs and KLTs.
- ULTs are managed at application level, and they are mapped to a few KLTs.
- ULTs do not directly get attached to KLTs but through an intermediary called lightweight processes (LWPs).
- ULTs are equal to KLTs or less in number.
- There are different types of mapping possible between ULTs and LWPs

Types of Threads

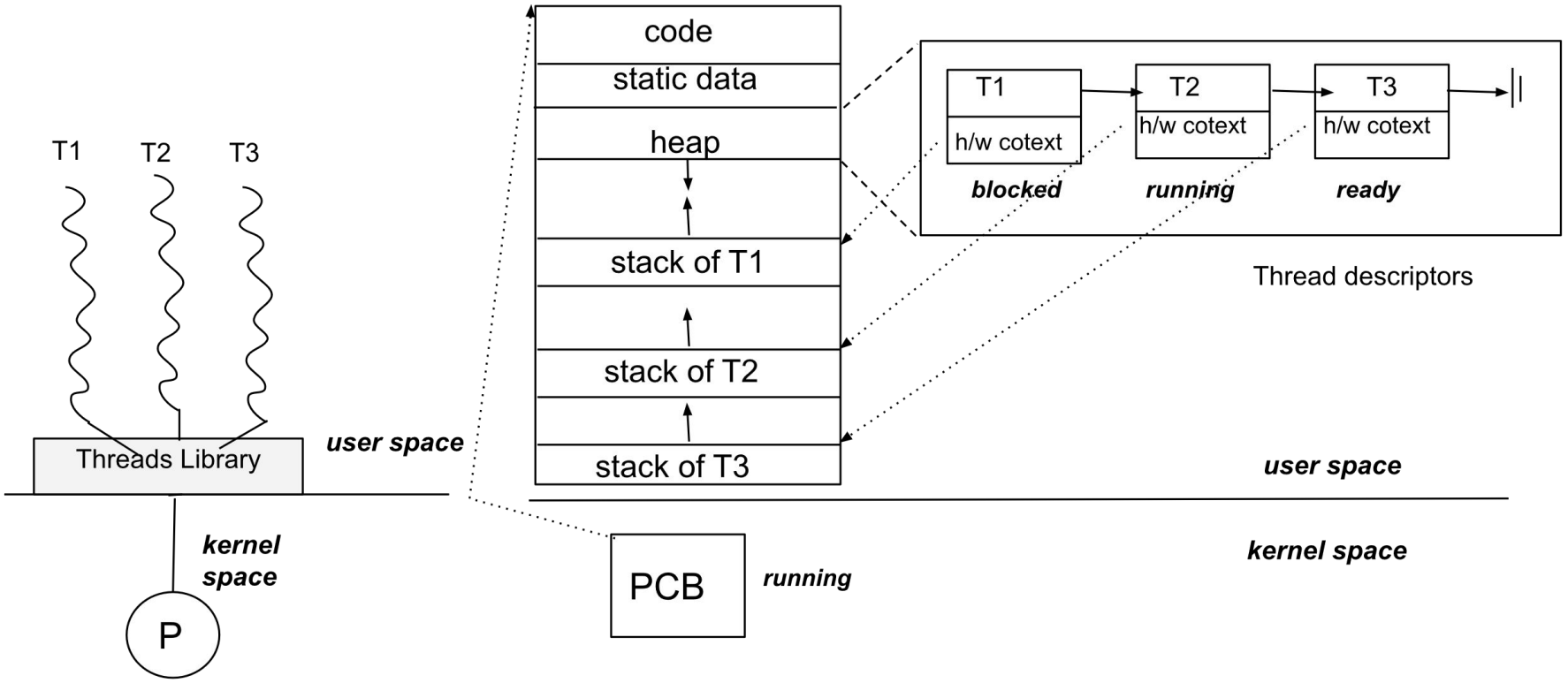


Fig 2.9: User Level Thread

Types of Threads

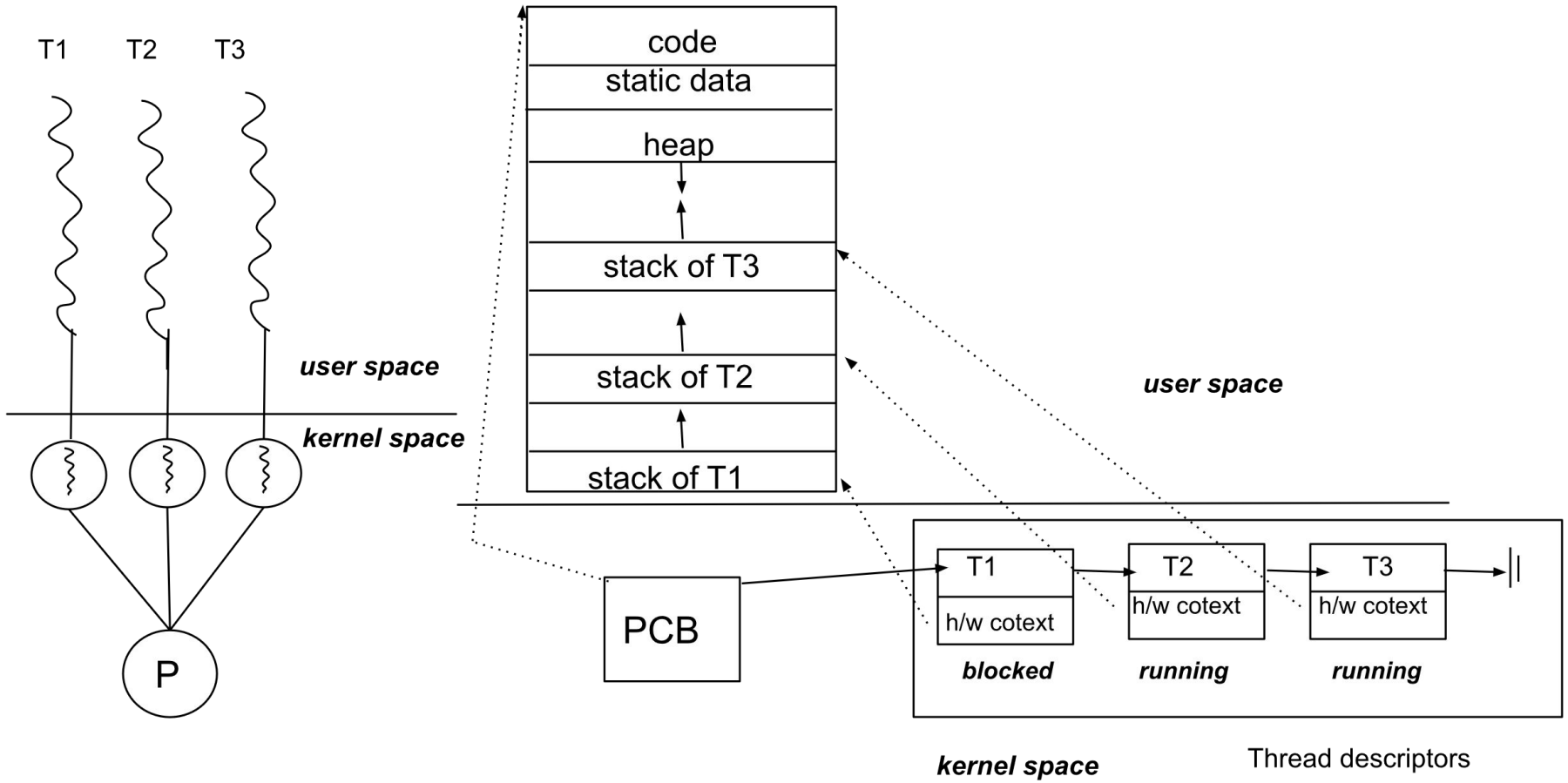


Fig 2.10: Kernel Level Thread

Types of Threads

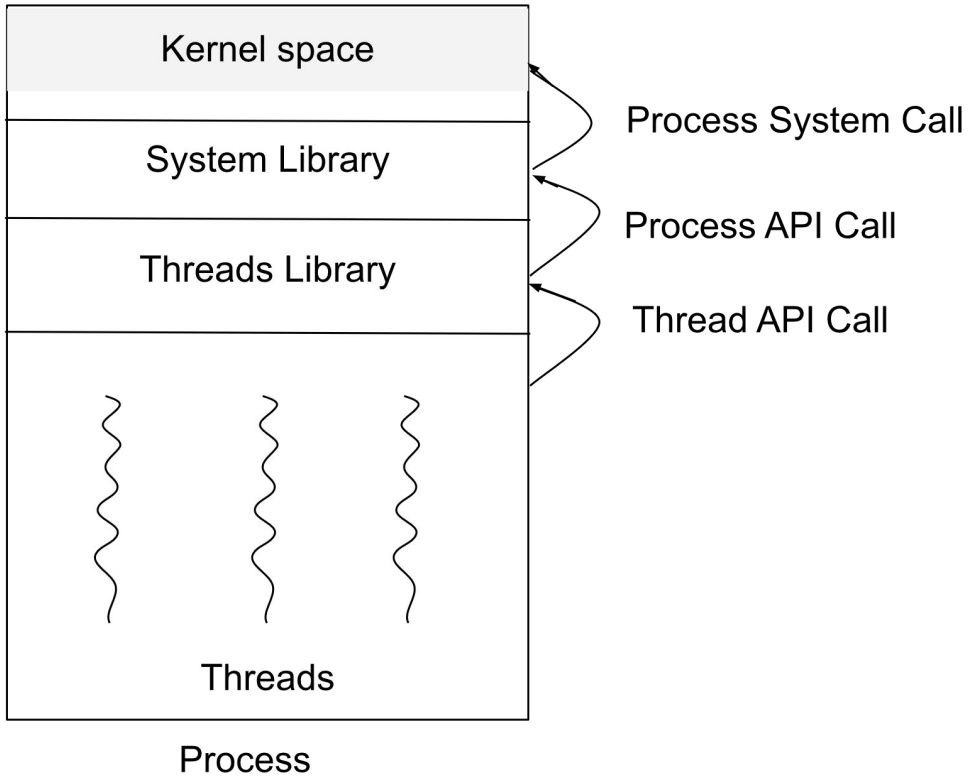


Fig 2.11: ULT interaction with kernel

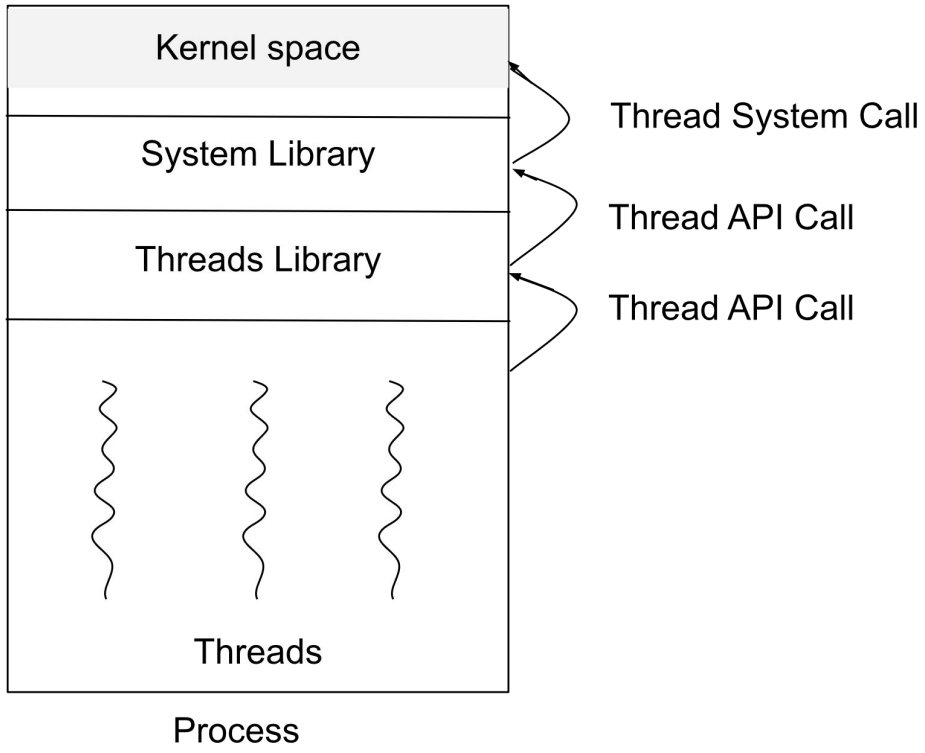


Fig 2.12: User application Interaction with Kernel in KLT

Types of Threads

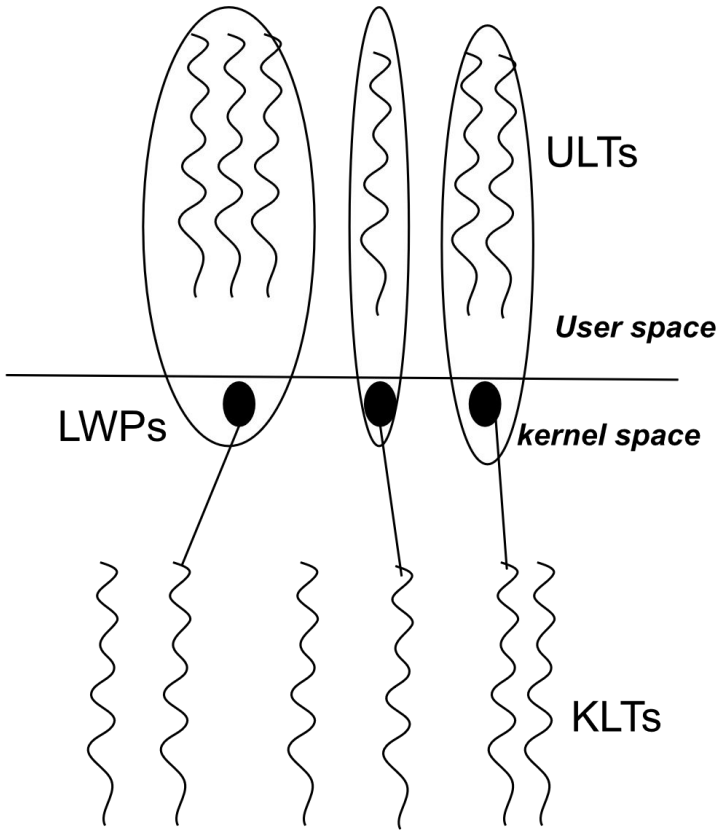


Fig 2.13: Mixed Approach

- There are different types of mapping possible between ULTs and LWPs
  - **a. one-to-one (1:1):**
  - **b. many-to-one (M:1)**:
  - **c. many-to-many (M:N):**

Concept of Multithreading

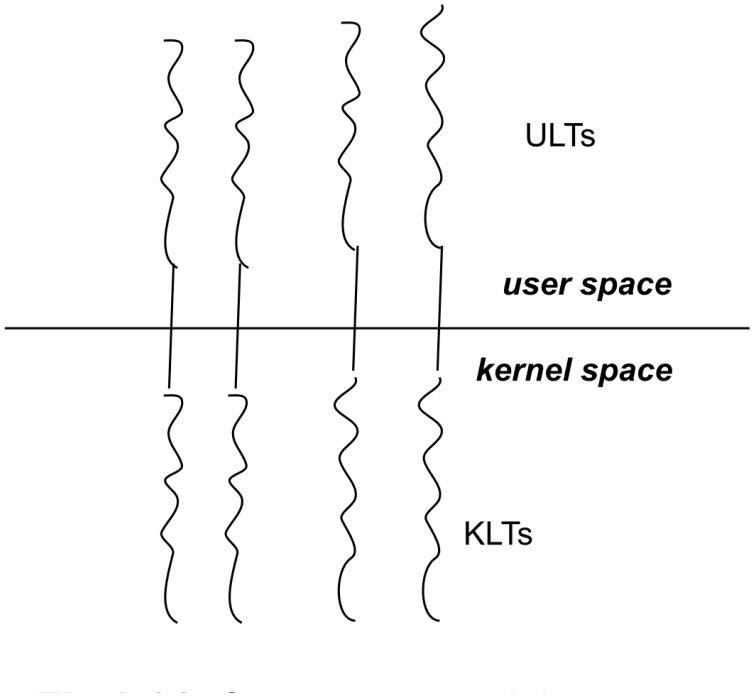


Fig 2.14: One to One Model

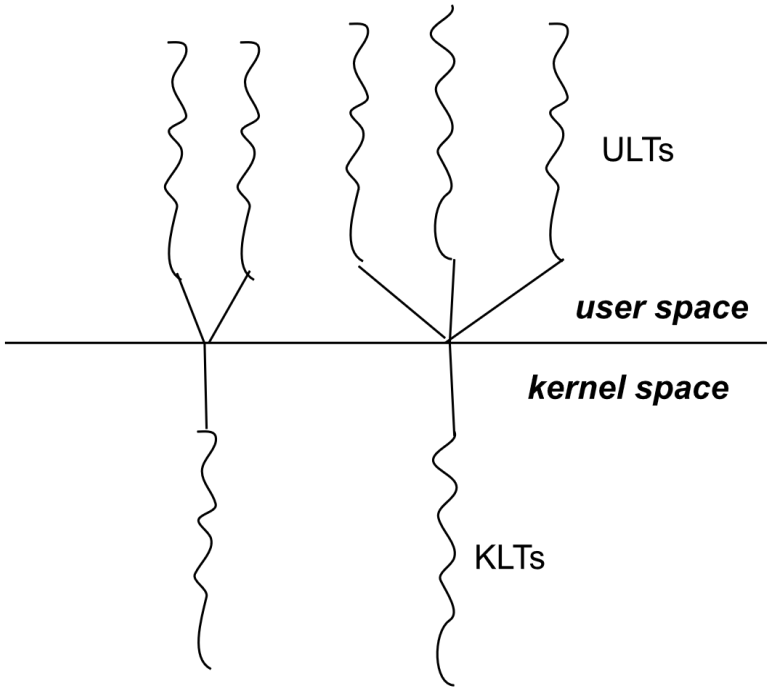


Fig 2.14: Many to One Model

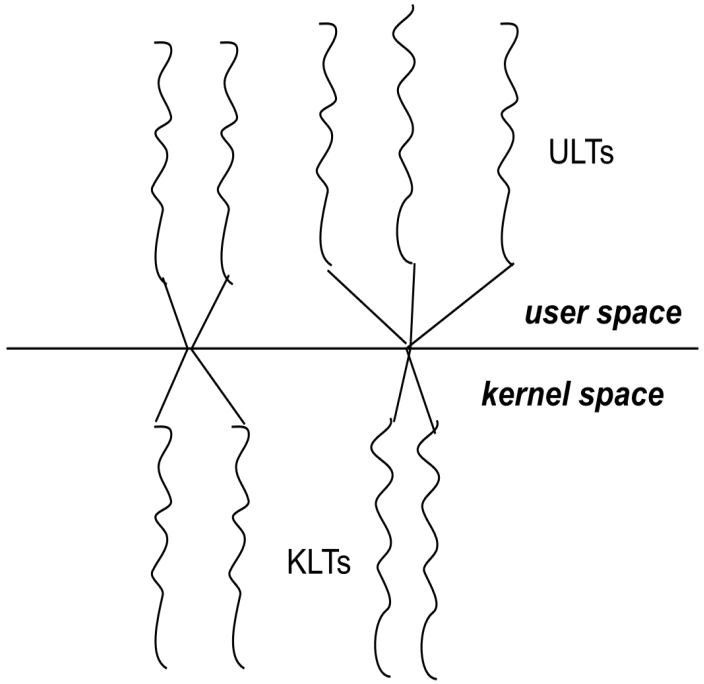


Fig 2.14: Many to Many Model

Amdahl's Law & Threading Speed-Up

speed-up

=

time to execute a program using a single threadtime to execute the same program using multiple thre

$$= \frac{1}{s + (1-s)/N}$$

where *s* is the fraction of serial code that cannot be parallelized, and *N* is the number of threads.

Concept of Multithreading

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

void *print_function( void *ptr ){
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}

main()
{
    pthread_t thread1, thread2, thread3;
    char *message1 = "From Thread 1";
    char *message2 = "From Thread 2";
    char *message3 = "From Thread 3";

    int  ret1, ret2, ret3;

   /* Create independent threads each of which will execute function */

    ret1 = pthread_create( &thread1, NULL, print_function, (void*) message1);
    ret2 = pthread_create( &thread2, NULL, print_function, (void*) message2);
    ret3 = pthread_create( &thread3, NULL, print_function, (void*) message3);

    pthread_join( thread1, NULL);   /* Wait till threads are complete */
    pthread_join( thread2, NULL);  /*otherwise process can terminate */
    pthread_join( thread3, NULL);  /* before threads are complete */

    printf("Thread 1 returns: %d\n",ret1);
    printf("Thread 2 returns: %d\n",ret2);
    printf("Thread 3 returns: %d\n",ret3);

    exit(0);
}
```

Compile it with `gcc -pthread <file_name.c>` and run as `./a.out`.

Fig 2.19: An example of multithreading using POSIX threads library

PROCESS SCHEDULING

- Every process needs a CPU core non-sharably to execute its code.
- Process scheduling is the method by which the OS allocates CPU cores to processes.
- In multiprogramming systems, multiple processes compete for the CPU.
- The CPU should never remain idle if there are processes waiting.
- **Scheduling objectives:**
  - Orderly allocation of CPU to all processes that require it.
  - Increased overall performance of the system in terms of throughput.
  - Increased degree of multiprogramming.

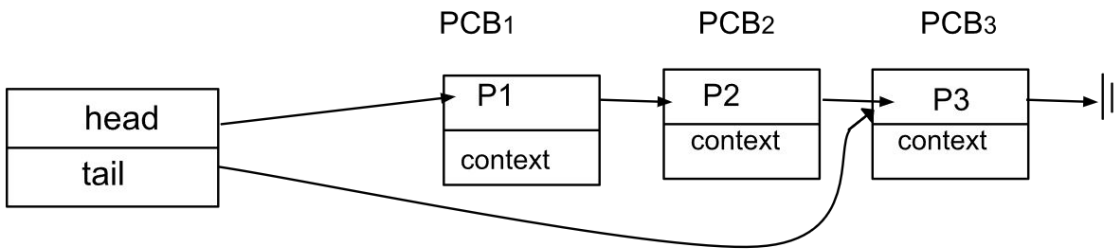


Fig 2.20: Queue of Processes waiting for accessing a Resource

PROCESS SCHEDULING

- **Types of Schedulers:**
  - **Long-term scheduler:** The scheduler decides how many processes and exactly which processes will be brought in the ready queue of a CPU.
  - **Short-term scheduler:** Once processes are brought into the CPU ready queue, which process out of them will be assigned the CPU next.

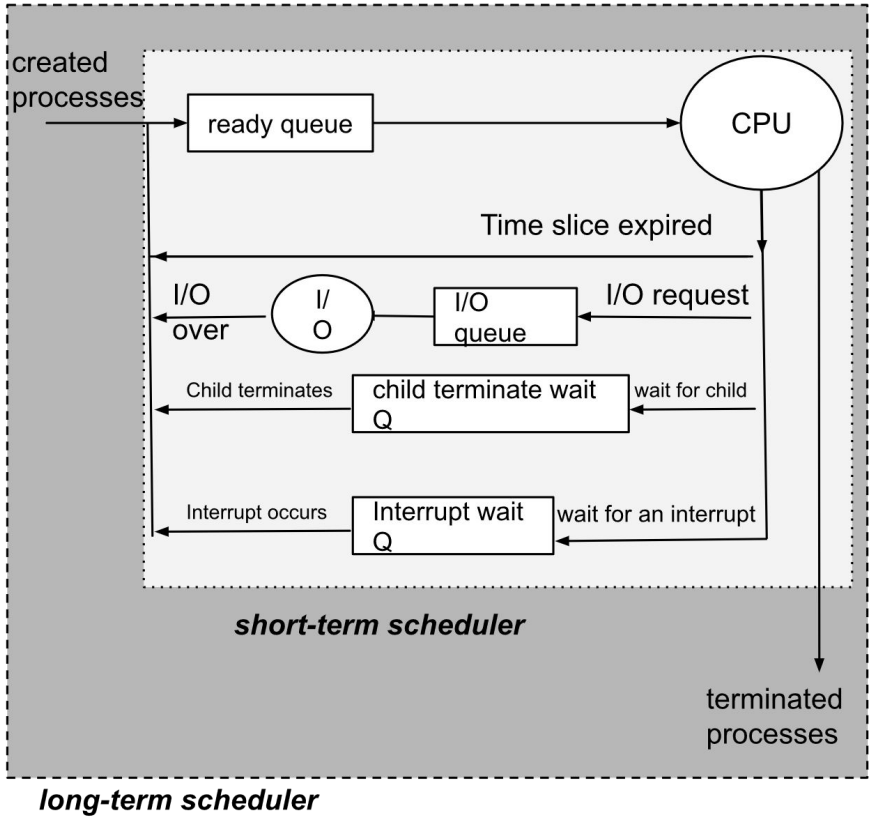


Fig 2.21: Scheduler and Different Queues

Scheduling Criteria

- **CPU Utilization:**
  - $CPU\ Utilization = \frac{\text{CPU busy time}}{\text{CPU total time}} = \frac{CPU\ busy\ time}{CPU\ busy\ time + CPU\ idle\ time}$
- **Throughput**: To measure the performance of any system in terms of units of work or task accomplished in unit time.
- **Turnaround Time:**
  - $Turnaround\ (TA)\ time = \ total\ wait\ time\ \ in\ ready\ Q\ + total\ execution\ time\ in\ CPU\ + \ total\ wait\ time\ in\ I/O\ Q + \ \ time\ for\ doing\ I/O.$
  - $TA\ time = \sum CPU\ bursts\ + \sum I/O\ bursts + \sum I/O\ bursts$
- **Waiting Time (WT):**Time spent in the Ready Queue
- **Response Time:** The time spent in getting the first response from the system.
- **Burst time :** The time spent for executing the activity in the burst excluding the wait time in the queue.

Scheduling Algorithms

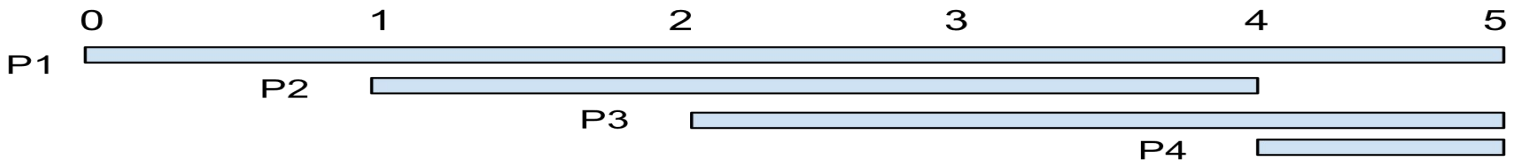
- Which process will get the chance first and next, when and for how long.
- CPU scheduling is needed under the following circumstances.
  1. A newly created process joins the ready queue, and the process needs to be immediately executed.
  2. The time slice allocated to a process is over and another process needs to get the CPU.
  3. A process needs an I/O before it can proceed any further.
  4. A process waits for its children to complete first before it proceeds further.
  5. A process waits for some interrupt (other than timer) and the interrupt occurs.
  6. A process completes its execution.

First-Come-First-Served (FCFS) Algorithm

Example 1: *Consider the following set of processes, with the arrival times and the CPU-burst times given in milliseconds. Find the average waiting time and average turnaround time in the FCFS algorithm.*

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	5
P2	1	3
P3	2	3
P4	4	1

Soln. Following is the arrival time CPU burst times of different processes.



According to FCFS, P1 will execute undisturbed, followed by P2, then P3 and P4.

Hence CPU will be held by them as per following timing diagram (called *Gantt chart*).

A process waits from arrival till it gets the CPU. Hence wait times are as follows:
for P1 = 0 ms
P2 = (5-1) = 4ms
P3 = (8-2) = 6ms
P4 = (11-4) = 7ms

Avg wait-time = (0+4+6+7)/4 = **4.25ms**

Turnaround time for each process is its time of completion minus its time of arrival.

Hence, TA times are as follows:
for P1 = (5-0) = 5ms
P2 = (8-1) = 7ms
P3 = (11-2) = 9ms
P4 = (12-4) = 8ms

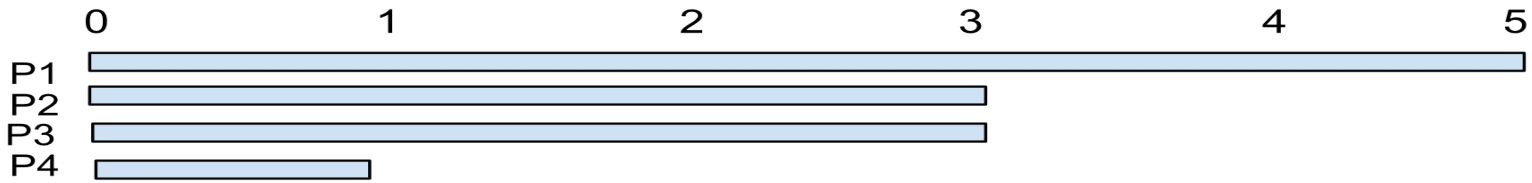
Avg TA-time = (5+7+9+8)/4 = **7.25ms**

Shortest-Job-First (SJF) Algorithm

Example 2: *The following set of processes arrive at the same time, however in the following order with the given CPU-burst times in milliseconds. Find the average waiting time and average turnaround time in the SJF algorithm.*

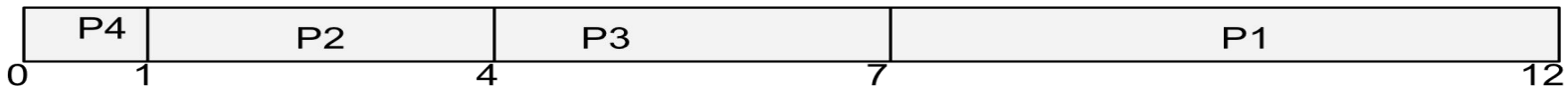
Process	Arrival Time (ms)	Burst Time (ms)
P1	0	5
P2	0	3
P3	0	3
P4	0	1

Soln. Following is the arrival time CPU burst times of different processes.



According to SJF, P4 will execute first, followed by P2, then P3 (since P2 & P3 have same CPU bursts, arrival order is given preference) and P1 at last.

Hence corresponding *Gantt chart* looks like the following.



A process waits from arrival till it gets the CPU.
Hence wait times are as follows:
for P1 = (7-0) = 7 ms
P2 = (1-0) = 1ms
P3 = (4-0) = 4ms
P4 = 0ms

Avg wait-time = (7+1+4+0)/4 = **3.0 ms**

Turnaround time for each process is its time of completion since its time of arrival.

Hence, TA times are as follows:
for P1 = (12-0) = 12ms
P2 = (4-0) = 4ms
P3 = (7-0) = 7ms
P4 = (1-0) = 1ms

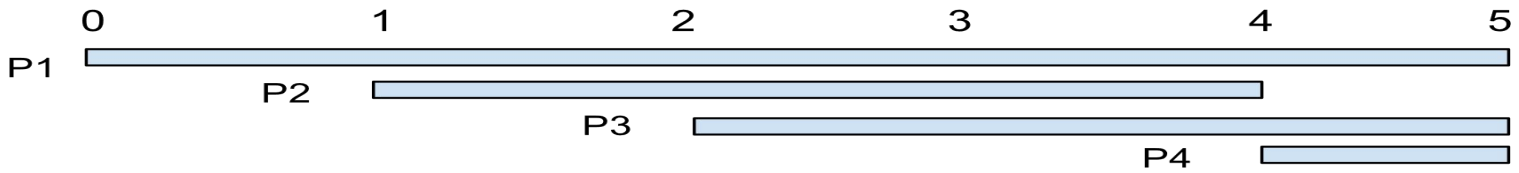
Avg TA-time = (12+4+7+1)/4 = **6.0 ms**

Shortest-Job-First (SJF) Algorithm

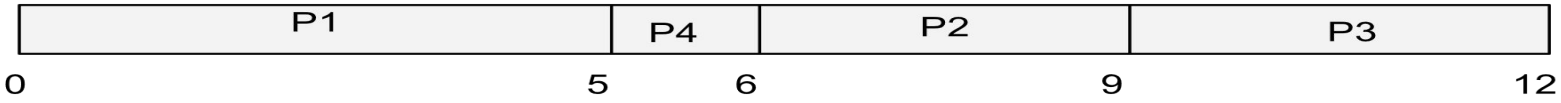
Example 3: *Consider the same problem as in* **Example 1.** *Find the average waiting time and average turnaround time in the SJF algorithm (non-preemptive).*

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	5
P2	1	3
P3	2	3
P4	4	1

Soln. Following is the arrival time & CPU burst times of different processes.



According to SJF (non-preemptive), P1 will execute first undisturbed, since other processes did not arrive then. But before P1 finishes, P2, P3, P4 arrived. Hence, P4 will execute next, followed by P2 and then P3 (though P2 & P3 are of same length, P2 is given preference for arriving early). Hence *Gantt chart* will look like the following.



A process waits from arrival till it gets the CPU.
Hence wait times are as follows:
for P1 = 0 ms
P2 = (6-1) = 5ms
P3 = (9-2) = 7ms
P4 = (5-4) = 1ms

Avg wait-time = (0+5+7+1)/4 = **3.25ms**

Turnaround time for each process is its time of completion since its time of arrival.

Hence, TA times are as follows:
for P1 = (5-0) = 5ms
P2 = (9-1) = 8ms
P3 = (12-2) = 10ms
P4 = (6-4) = 2ms

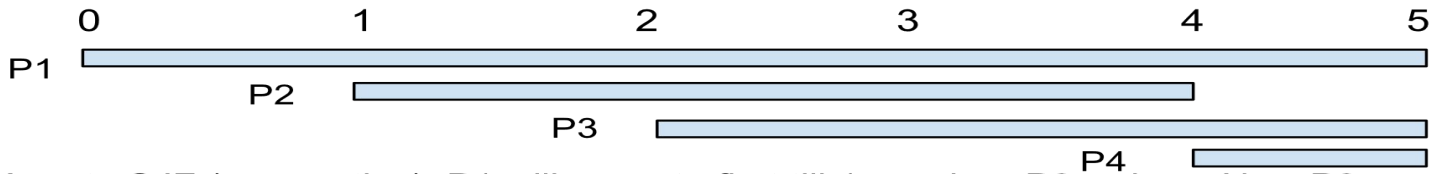
Avg TA-time = (5+8+10+2)/4 = **6.25ms**

Shortest-Job-First (SJF) Algorithm

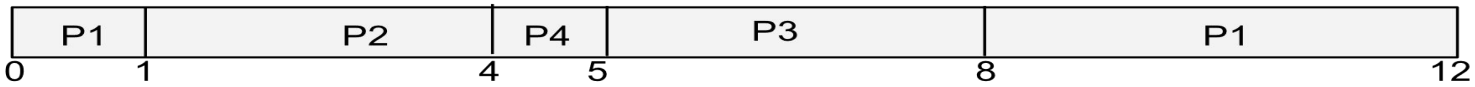
Example 4: *Consider the same problem as in* **Example 1.** *Find the average waiting time and average turnaround time in the SJF algorithm (preemptive).*

Soln. Following is the arrival time & CPU burst times of different processes.

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	5
P2	1	3
P3	2	3
P4	4	1



Acc. to SJF (preemptive), P1 will execute first till 1ms when P2 arrives. Now P2 needs 3ms of CPU whereas P1 needs 4ms. Hence P2 will execute till 2ms when P3 arrives. Here P2 has the smallest CPU burst (3-1 = 2ms) whereas P1 has = (5-1) = 4ms, and P3 has 3ms. At time = 4ms, P2 completes and P4 arrives with CPU burst 1ms (the smallest). Hence P4 will execute next. P3 will start after P4 as its CPU burst time (3ms) is smaller than than that of P1 (4ms). Hence *Gantt chart* will look like the following.



A process after arrival waits whenever it is not using CPU.
Hence wait times are as follows:
for P1 = (8-1) =7ms
P2 = (1-1) =0 ms
P3 = (5-2) = 3ms
P4 = (4-4) = 0ms

Avg wait-time = (7+0+3+0)/4 = **2.5ms**

Turnaround time for each process is its time of completion since its time of arrival.

Hence, TA times are as follows:
for P1 = (12-0) = 12ms
P2 = (4-1) = 3ms
P3 = (8-2) = 6ms
P4 = (5-4) = 1ms

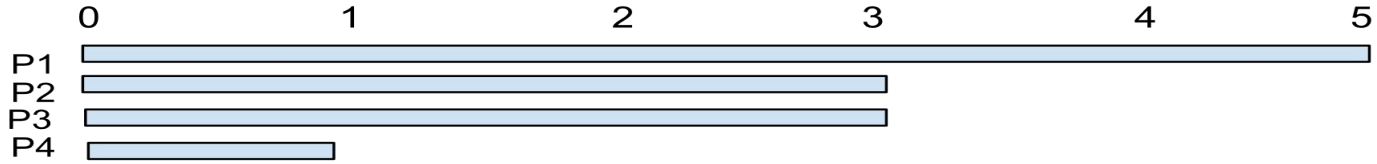
Avg TA-time = (12+3+6+1)/4 = **5.5ms**

Round-Robin (RR) Algorithm

Example 5: *The following set of processes arrive at the same time, however in the following order with the given CPU-burst times in milliseconds. Find the average waiting time and average turnaround time in the RR algorithm with time-slice of 1ms.*

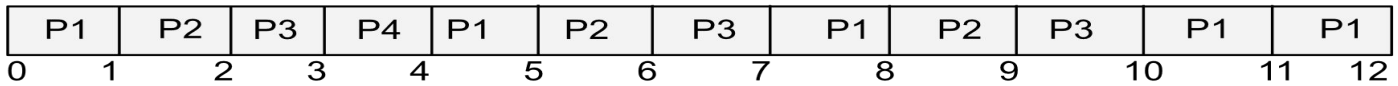
Process	Arrival Time (ms)	Burst Time (ms)
P1	0	5
P2	0	3
P3	0	3
P4	0	1

Soln. Following is the arrival time CPU burst times of different processes.



According to RR, P1 will execute first for 1ms, followed by P2, P3 and P4 each for 1ms in the order. Then again the next cycle will start with P1 and so on. Since P4 will complete in the 1st cycle itself, it will not feature in the 2nd cycle.

Hence corresponding *Gantt chart* looks like the following.



A process waits from arrival till it gets the CPU. Here, wait happens multiple times for a process.

Hence wait times are as follows:
for P1 = (3+2+2) = 7 ms
P2 = (1+3+2) = 6ms
P3 = (2+3+2) = 7ms
P4 = 3ms

Avg wait-time = (7+6+7+3)/4 = **5.75 ms**

Turnaround time for each process is its time of completion since its time of arrival.

Hence, TA times are as follows:
for P1 = (12-0) = 12ms
P2 = (9-0) = 9ms
P3 = (10-0) = 10ms
P4 = (4-0) = 4ms

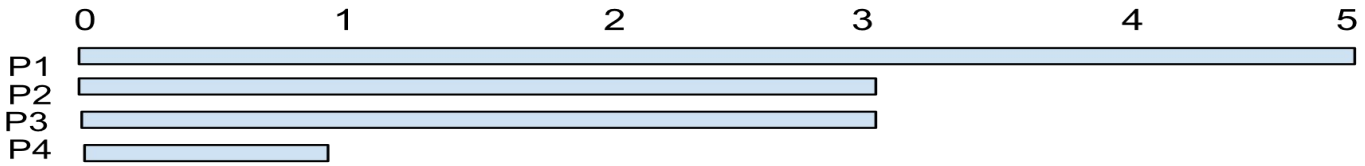
Avg TA-time = (12+9+10+4)/4 = **8.75 ms**

Round-Robin (RR) Algorithm

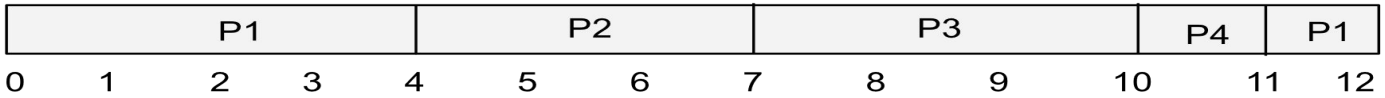
Example 6: *The following set of processes arrive at the same time, however in the following order with the given CPU-burst times in milliseconds. Find the average waiting time and average turnaround time in the RR algorithm with time-slice of 4ms.*

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	5
P2	0	3
P3	0	3
P4	0	1

Soln. Following is the arrival time CPU burst times of different processes.



According to RR, P1 will execute first for 4ms, followed by P2, P3 and P4 each for maximum 4ms in the order. Then again, the next cycle will start with P1 and so on. Since P2, P3, and P4 have burst times less than the time quantum (4ms), they will complete in the 1st cycle itself except P1 that will feature in the 2nd cycle. Hence corresponding *Gantt chart* looks like the following.



Wait time is the total time spent in ready queue of the CPU. Here, wait happens multiple times.

Hence wait times are as follows:
for P1 = (11-4) = 7 ms
P2 = (4-0) = 4ms
P3 = (7-0) = 7ms
P4 = (10-0)=10ms

Avg wait-time = (7+4+7+10)/4 = **7.0 ms**

Turnaround time for each process is its time of completion since its time of arrival.

Hence, TA times are as follows:
for P1 = (12-0) = 12ms
P2 = (7-0) = 7ms
P3 = (10-0) = 10ms
P4 = (11-0) = 4ms

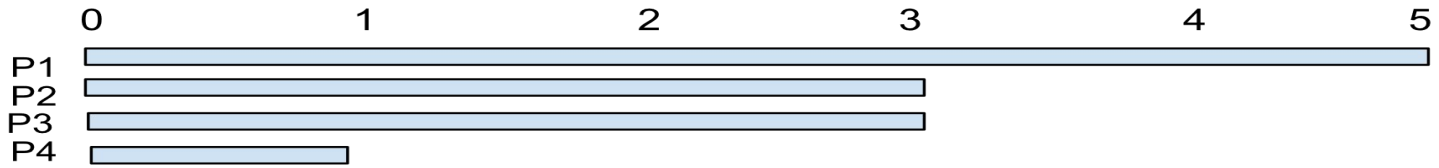
Avg TA-time = (12+7+10+4)/4 = **8.25 ms**

Priority-based Scheduling Algorithm

Example 7: *The following set of processes arrive at the same time, however in the following order with the given CPU-burst times in milliseconds. Find the average waiting time and average turnaround time in according to the priority based scheduling.*

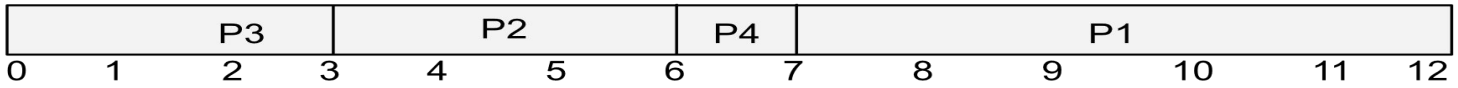
Process	Burst Time (ms)	Priority
P1	5	4
P2	3	2
P3	3	1
P4	1	3

Soln. Following is the arrival time CPU burst times of different processes.



According to priority-based scheduling, P3 will execute first undisturbed, followed by P2, P4 and P1 at the last.

Hence corresponding *Gantt chart* looks like the following.



A process waits from arrival till it gets the CPU.

Hence wait times are as follows:
for P1 = (7-0) = 7 ms
P2 = (3-0) = 4ms
P3 = 0ms
P4 = (6-0)= 6ms

Avg wait-time = (7+4+0+6)/4 = **4.25 ms**

Turnaround time for each process is its time of completion since its time of arrival.

Hence, TA times are as follows:
for P1 = (12-0) = 12ms
P2 = (6-0) = 6ms
P3 = (3-0) = 3ms
P4 = (7-0) = 7ms

Avg TA-time = (12+6+3+7)/4 = **7.0 ms**

Thread Scheduling

- In most modern OSs, threads are considered units of work and can be independently scheduled.

1. **One-level thread scheduling**
   - Threads are directly assigned to CPU cores.
   - Similar to process scheduling: Uses CPU scheduling algorithms (FCFS, RR, etc.).
   - Efficient but limited control over how threads are scheduled.

2. **Two-Level Thread Scheduling**
   - CPUs are assigned to processes, and thread scheduling is managed within each process.
   - Two possible implementations:
   - Single Scheduler: OS-level scheduling only (Kernel-level threads).
   - Dual Scheduler: Application-level scheduler (for ULTs) & OS scheduler (for KLTs).

Multiprocessor Scheduling

- Multiple CPU cores allow parallel execution of several processes and/or several threads simultaneously.
- This makes CPU scheduling more complex.
- A multiprocessor system may also refer to any of the following system architectures:
  a) Homogeneous multiprocessing
     i) a multi-core CPU or a set of multicore CPUs
     ii) Multi-threaded cores
     iii) Non-Uniform Memory Architecture (NUMA) systems
  b) Heterogeneous multiprocessing.

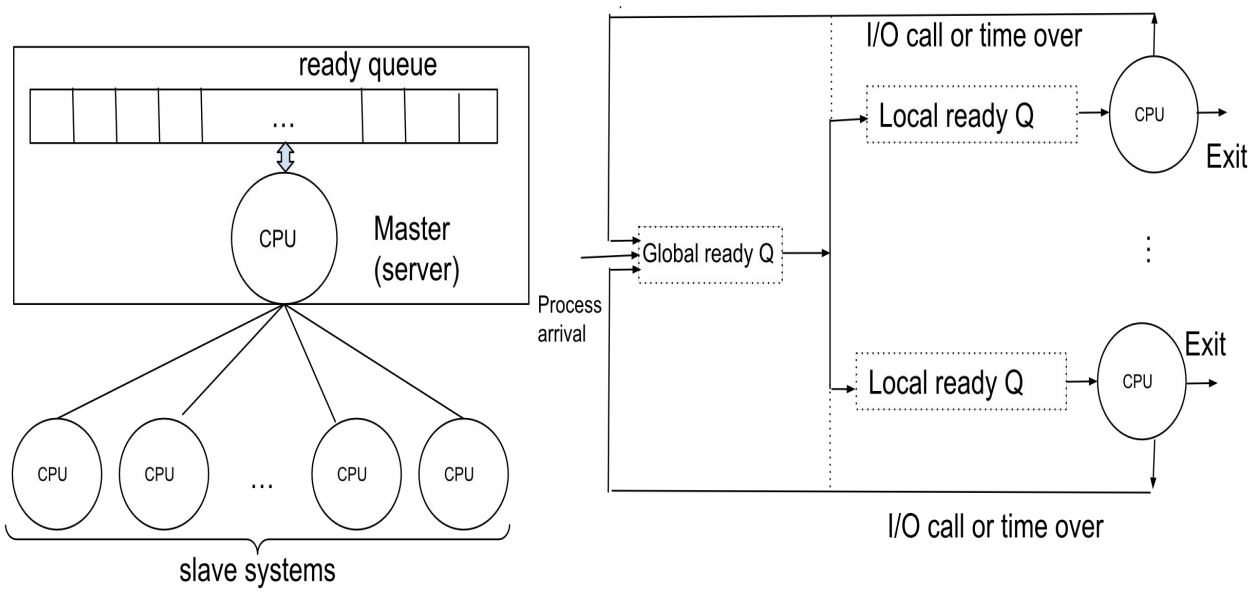
Homogeneous multiprocessing

i) Asymmetric multiprocessing:

- A processor acts as the master server while others as the slaves.
- The master takes all the scheduling decisions.
- The slaves execute only user code.

i) Symmetric multiprocessing (SMP):

- Each processor can schedule on its own.
- They can either have a single global ready queue shared by all the processors or each processor can have their own local and private ready queue
- Issues in SMP Scheduling
  - A. Load Balancing
  - B. Processor Affinity



a) Asymmetric multiprocessing *b) Symmetric multiprocessing*

Fig 2.22: Homogeneous multiprocessing Scheduling

Homogeneous multiprocessing

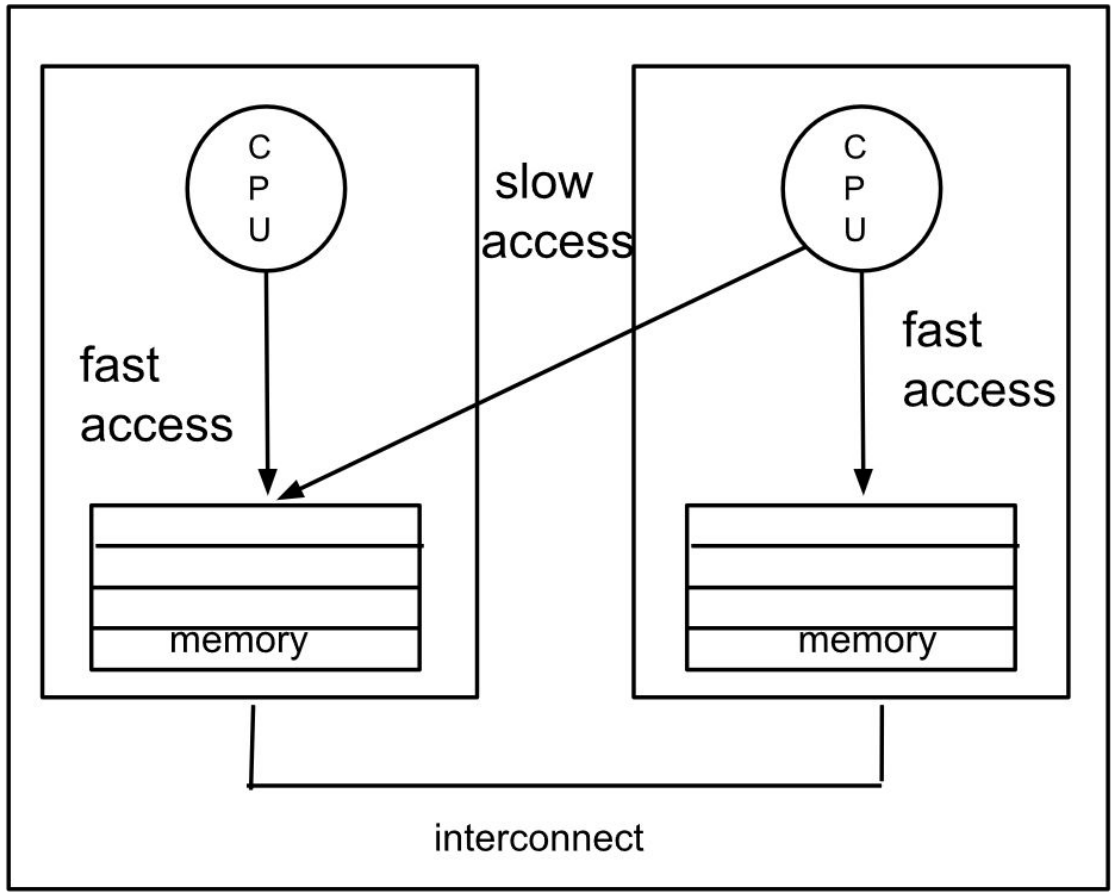


Fig 2.23: NUMA Architecture

Heterogeneous multiprocessing

- Some of the present day mobile devices use multicore processors of different processing attributes (clock speed, power requirement etc.). Such systems are called heterogeneous multiprocessing (HMP).
- Mainly used to save battery power for long hours.
- The littles save energy while the bigs deliver performance.
- Windows 10 supports HMP scheduling.

Real-Time Scheduling

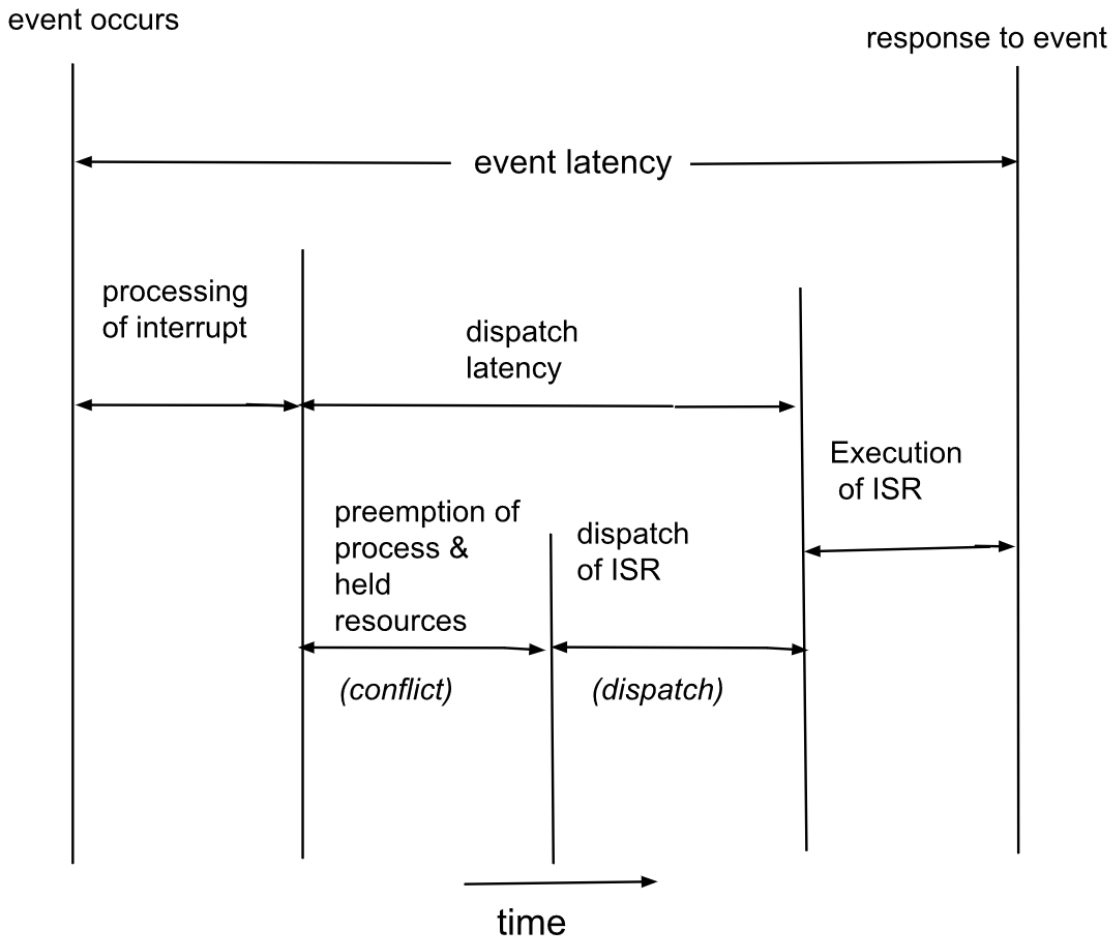


Fig 2.24: Real-Time Scheduling: Important Latencies

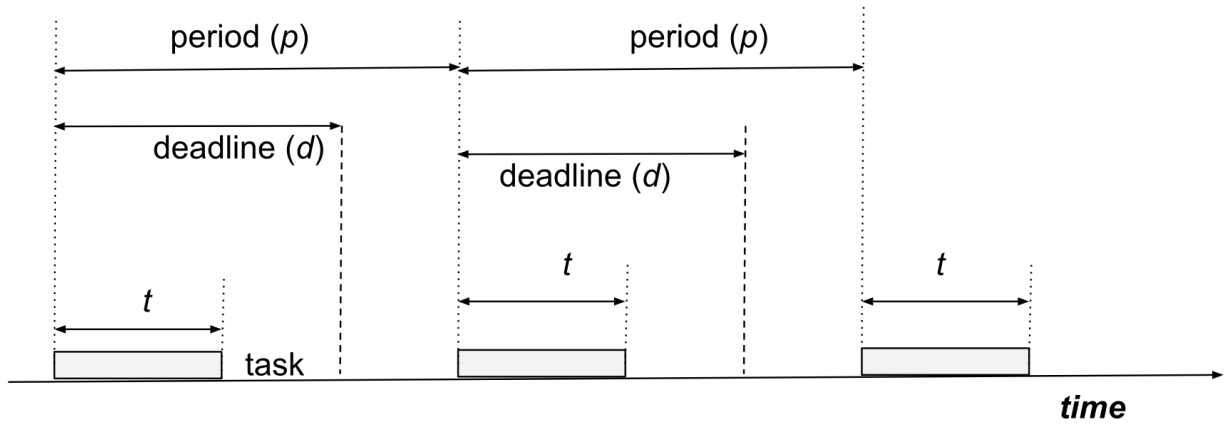


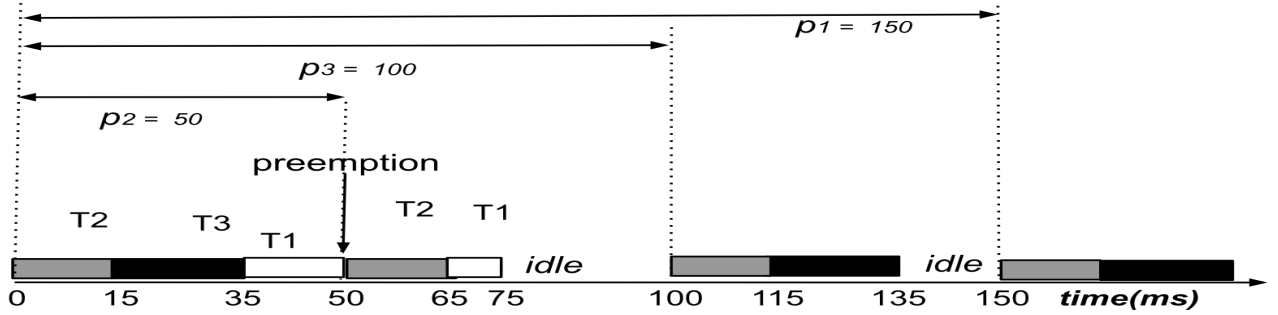
Fig 2.24: Periodic Task with deadlines in Real-Time Scheduling

Rate Monotonic (RM) Scheduling Algorithm

Example 8: *A set of periodic processes arrive at the same time with the following execution times and periods. Deadlines for each process can be considered as before the start of next period. Show their possible scheduling according to the RM algorithm.*

tasks	Exec Time (c) (ms)	Period (p)
T1	25	150
T2	15	50
T3	20	100

Soln. Here, T2 has the shortest period, followed by T3 and then T1.
Hence, the scheduling order will be T2 ⟶ T3 ⟶ T1.



At *t=0ms,* tasks T1, T2 and T3 arrive. According to RM algorithm, T2 gets CPU and run entirely.
At *t= 15ms,* T2 completes and releases CPU. T3 gets the CPU as it has the next highest priority, and can complete as it needs 20ms of execution, well within the first period of 50ms.
At *t=35ms,* T3 completes, leaves the CPU and T1 can run for 15ms till 50ms.

At t=50ms, T2 arrives for the second time with the highest priority. T1 is therefore preempted and T2 starts.
At *t =65ms,* T2 completes and leaves the CPU. T1 is the only remaining task that can run the remaining part of T1 (10ms) and completes at 75 ms. The CPU remains idle for next 25ms.

At *t=100ms,* both T2 and T3 arrive. T2 run first for 15 ms, followed by T3 for 20 ms.
At *t=135ms,* T3 leaves CPU and there is no task remaining. The CPU remains idle for 15 ms.

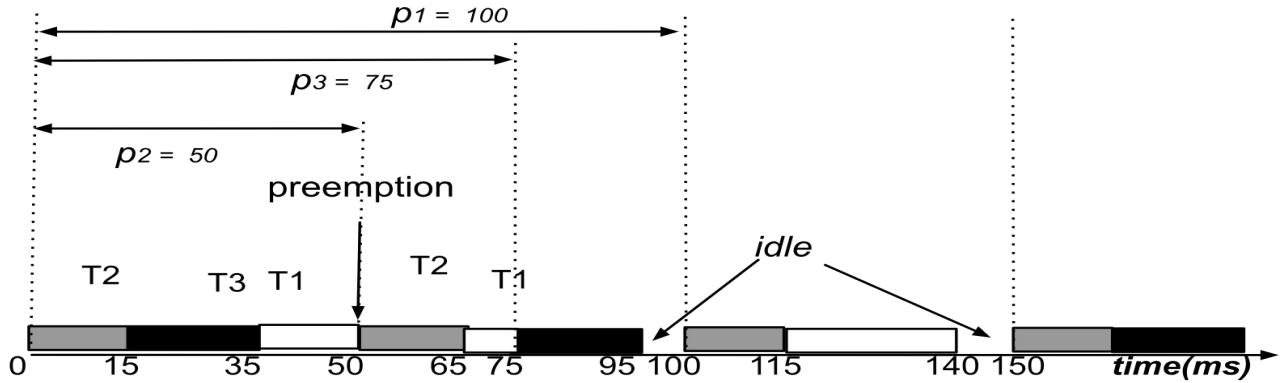
At *t=150ms*, all of T1, T2 and T3 arrive again. Here, the scenario beginning at *t=0* starts repeating.

Rate Monotonic (RM) Scheduling Algorithm

Example 9: *A set of periodic processes arrive at the same time with the following execution times and periods. Deadlines for each process can be considered as before the start of the next period. Show their possible scheduling according to the any algorithms.*

tasks	Exec Time (c) (ms)	Period (p)
T1	25	100
T2	15	50
T3	20	75

Soln. Let us first check the schedulability. According to schedulability criterion, for 3 processes, this bound is 3(2^(⅓)-1)= 0.779. Here, the sum of utilizations is (25/100+15/50 + 20/75) =0.817 which is greater than the bound (0.779). Hence, we are not guaranteed to find a schedule with all hard deadlines met according to the RM algorithm. However, let us see below.



At *t=0ms,* tasks T1, T2 and T3 arrive. According to RM algorithm, T2 gets CPU and run entirely.
At *t= 15ms,* T2 completes and releases CPU. T3 gets the CPU as it has the next highest priority, and can complete as it needs 20ms of execution, well within the first period of 50ms.
At *t=35ms,* T3 completes, leaves the CPU and T1 can run for 15ms till 50ms.

At t=50ms, T2 arrives for the second time with the highest priority. T1 is therefore preempted and T2 starts.
At *t =65ms,* T2 completes and leaves the CPU. T1 is the only remaining task that can run the remaining part of T1 (10ms) and completes at 75 ms.

At *t=75ms,* T3 arrives. T3 runs for 20 ms till 95 ms. No other process is there and CPU remains idle for 5ms.
At *t=100ms,* T1 and T2 arrive. Acc to RM, T2 runs entirely (till 115ms) followed by T1 running completely till 140ms. The CPU can remain idle for 10 ms.

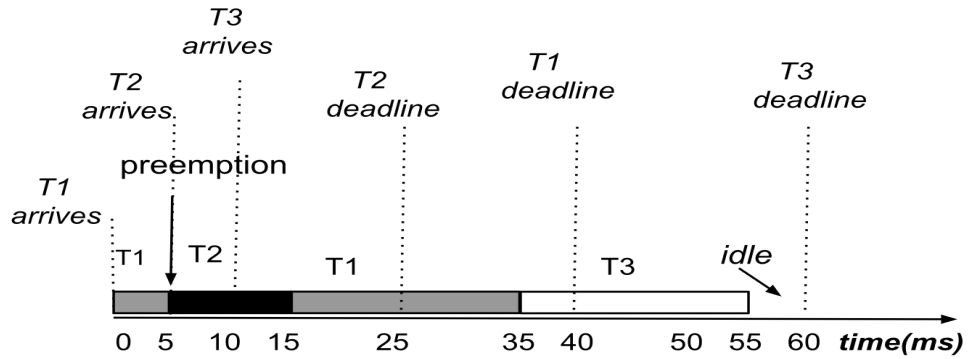
At *t=150ms*, T2 and T3 arrive again. Both can comfortably complete as sum of their execution times (15+20) is less than 50, before any new period starts (T1, T2 come at t = 200ms).
Hence, we can even find a schedule according to RM algorithm, even if schedulability criterion is not satisfied.

Earliest Deadline First (EDF) Scheduling Algorithm

Example 10: *The following set of aperiodic processes arrive at the mentioned time with the given execution times and deadlines. Show their possible scheduling according to the EDF algorithm.*

Tasks	Arrival time (ms)	Exec Time (c) (ms)	End-deadline (ms)
T1	0	25	40
T2	5	10	20
T3	10	20	50

Soln. Here, T1 appears first when there are no other processes. Hence, it can start and execute till 5ms when T2 arrives.



At *t=5ms,* task T2 arrives with deadline at (5+20)=25ms, earlier than that of T1 [at 40ms], hence T1 is preempted and T2 starts running.

At *t= 10ms,* T3 arrives with deadline at (10+50) = 60 ms, later than that of T2. Hence T2 continues till its end, i.e. till 15ms.

At *t= 15ms,* T2 leaves CPU. Now T1 has deadline earlier than that of T3, hence T1 gets CPU and completes rest of its execution (20ms) till (15+20)=35ms.

At *t=35ms,* T1 completes, leaves the CPU and T3 can run for 20ms till 55ms.

At *t=55ms,* T3 completes its execution comfortably as it had the latest deadline at 60 ms.

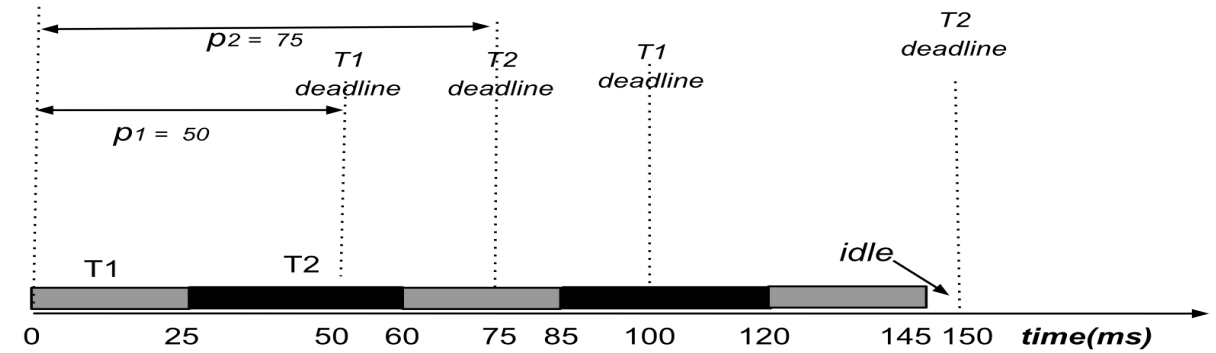
Hence, all three tasks complete well within their end-deadlines according to the EDF algorithm.

Earliest Deadline First (EDF) Scheduling Algorithm

Example 11: *The following pair of periodic processes arrive at the same time with the given execution times and periods. Deadlines for each process can be considered as before the start of next period. Show their possible scheduling according to the EDF algorithm.*

tasks	Exec Time (c) (ms)	Period (p)
T1	25	50
T2	35	75

Soln.



At *t = 0ms,* tasks T1 and T2 arrive simultaneously. According to EDF algorithm, T1 with earlier deadline (at 50ms) gets the CPU first and run entirely.
At *t= 25ms,* T1 releases CPU and T2 starts.
At *t=50ms,* T1 comes again for the next period. But T1 has the next deadline at 100ms, while T2 at 75ms. Hence T2 continues.

At *t=60ms,* T2 completes, leaves the CPU and T1 can start.

At *t=75ms*, deadline for T2 strikes, but T2 already completed at 60ms. T2 comes for the second cycle, but T1 has earlier deadline at 100ms and so continues.
At *t = 85ms,* T1 completes and leaves the CPU. T2 for the second cycle starts.

At *t=100ms,* T1 comes again for the third cycle with its deadline at 150ms, while T2 also has at 150ms. Hence T2 continues applying FCFS.
At *t=120ms,* T2 completes it burst of 35ms and leaves CPU. T1 starts and continues for 25ms of its burst.
At *t=145ms,* T1 completes 25ms of burst and leaves CPU. CPU remains idle for 5ms.

At *t=150ms*, both T1 and T2 arrive again. Here, the scenario beginning at *t=0* starts repeating.

Reference

[1] “OPERATING SYSTEMS”, Author: Dr. Sukomal Pal Associate Professor Department of Computer Science & Engineering IIT (BHU), Varanasi, UP