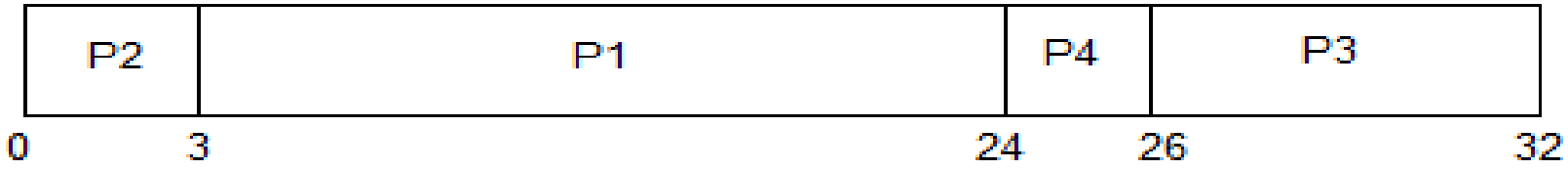
Priority Scheduling

- Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Priority Scheduling

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,

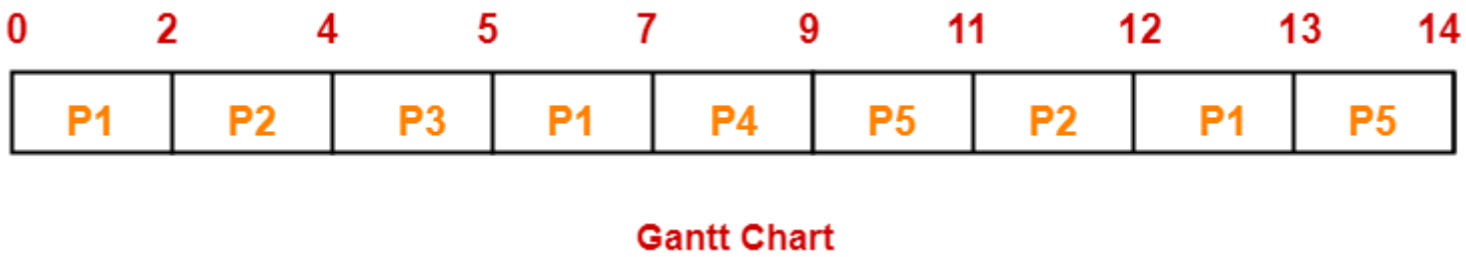


The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

Round Robin Scheduling

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preemptied and other process executes for given time period.
- Context switching is used to save states of preemptied processes.

Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3



Gantt Chart

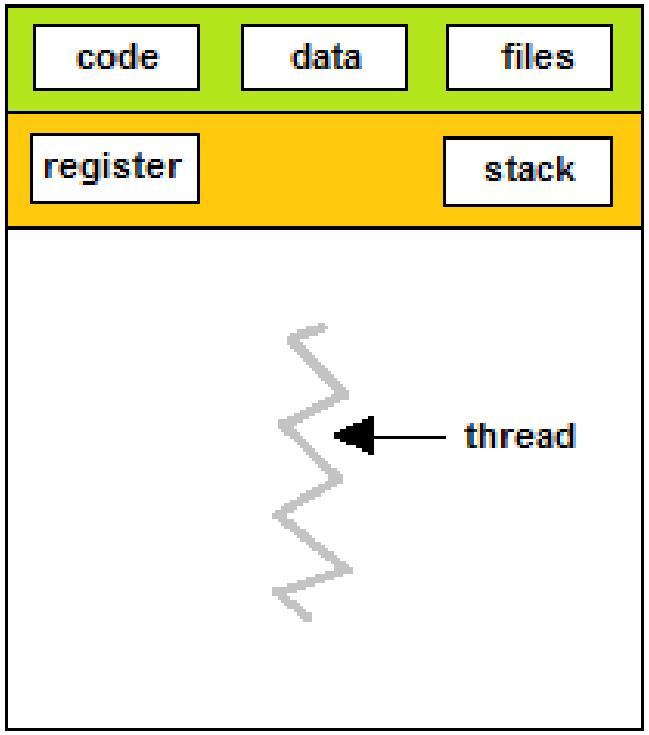
Process Id	Exit time	Turn Around time	Waiting time
P1	13	13 – 0 = 13	13 – 5 = 8
P2	12	12 – 1 = 11	11 – 3 = 8
P3	5	5 – 2 = 3	3 – 1 = 2
P4	9	9 – 3 = 6	6 – 2 = 4
P5	14	14 – 4 = 10	10 – 3 = 7

- Average waiting time = (8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit

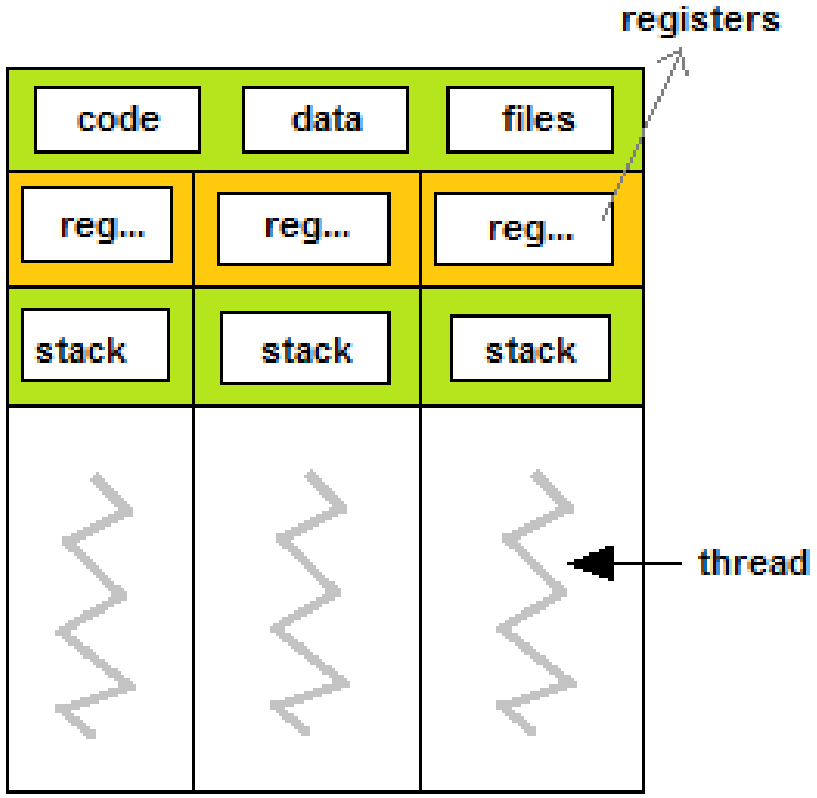
What is Thread?

- **Thread** is an execution unit which consists of its own program counter, a stack, and a set of registers.
- Threads are also known as Lightweight processes.
- Threads are popular way to improve application through parallelism.
- The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.
- As each thread has its own independent resource for process execution, multpile processes can be executed parallely by increasing number of threads.

- **Program counter** that keeps track of which instruction to execute next,
- **System registers** which hold its current working variables,
- **Stack** which contains the execution history.



single-threaded process



multithreaded process

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Types of Thread

There are two types of threads:

User threads,

Are above the kernel and without kernel support.

These are the threads that application programmers use in their programs.

Kernel threads

are supported within the kernel of the OS itself.

All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

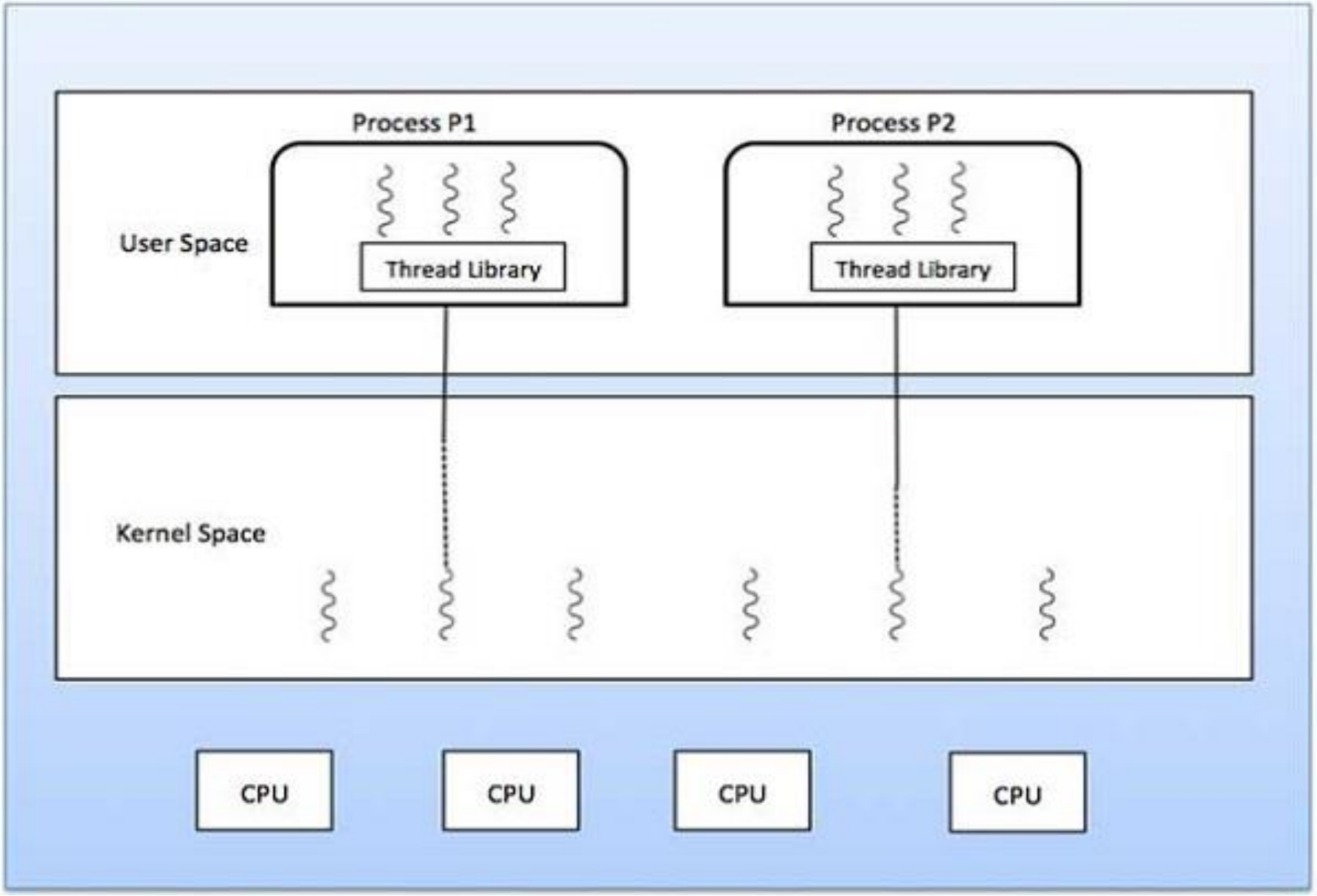
Multithreading Models

The user threads must be mapped to kernel threads, by one of the following strategies:

- Many to One Model
- One to One Model
- Many to Many Model

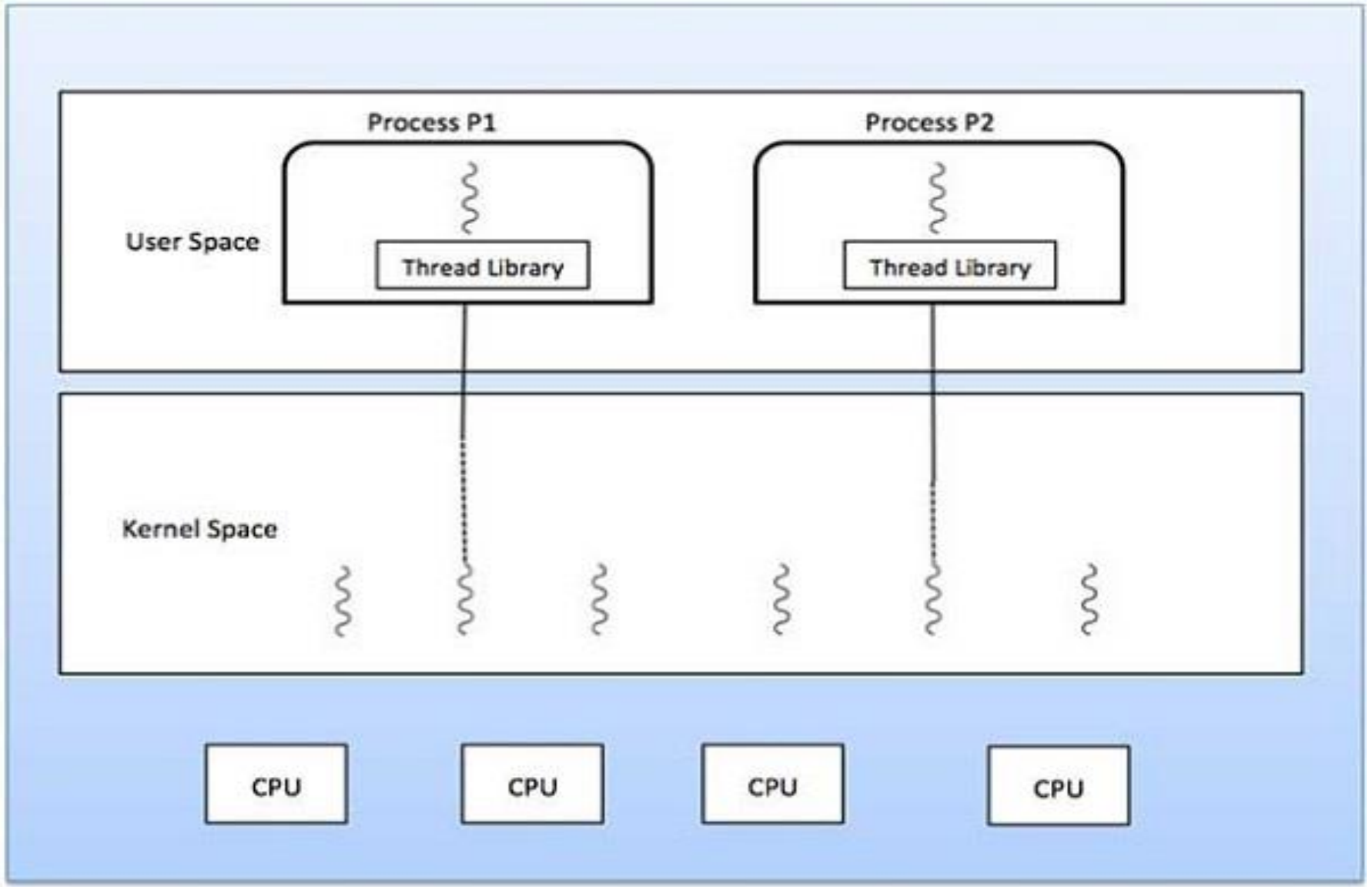
Many to One Model

- In the **many to one** model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is efficient in nature.


Process P1
Process P2
User Space
Thread Library
Thread Library
Kernel Space
CPU
CPU
CPU
CPU

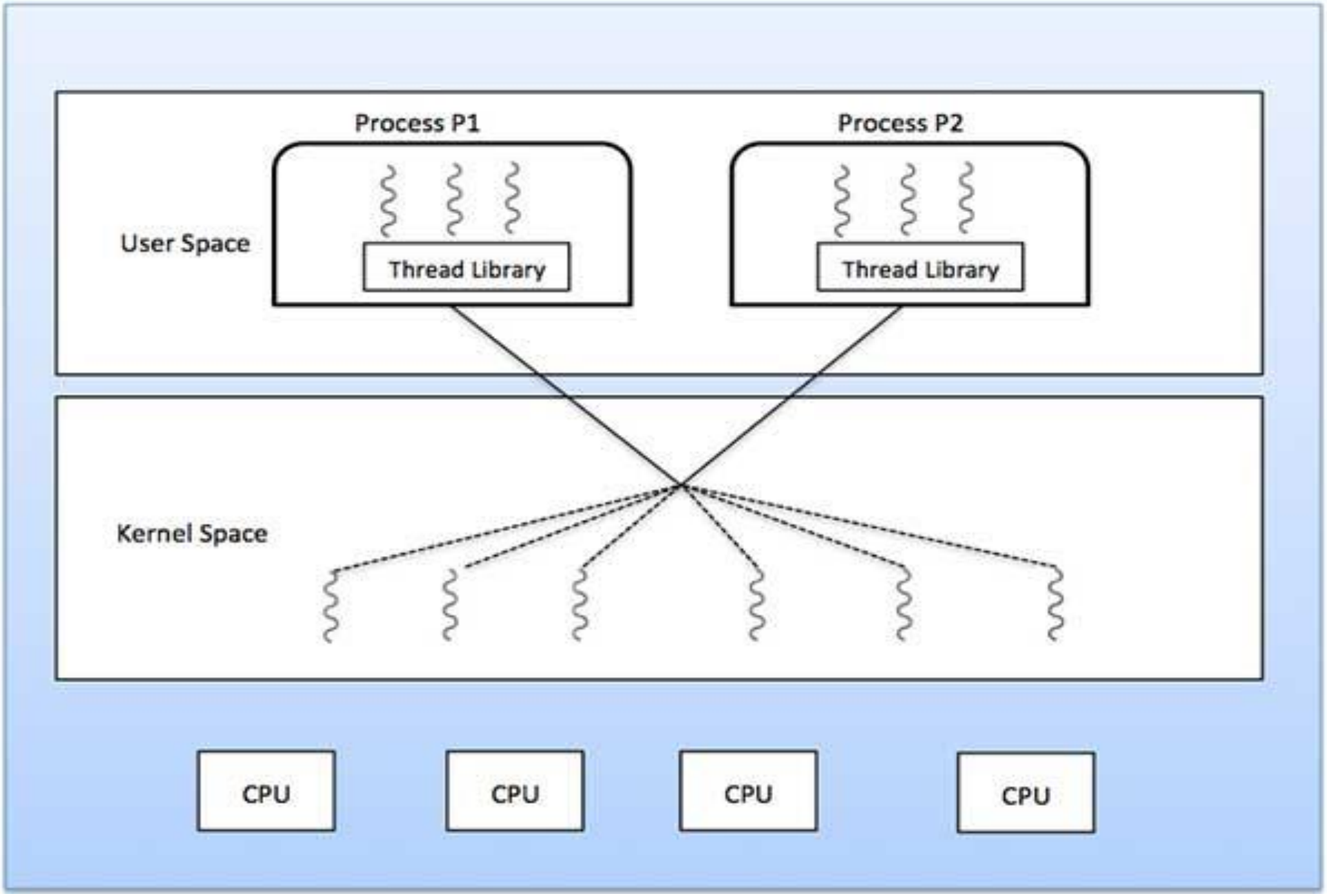
One to One Model

- The **one to one** model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.


Process P1
Process P2
User Space
Thread Library
Thread Library
Kernel Space
CPU
CPU
CPU
CPU

Many to Many Model

- The **many to many** model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.


Process P1
Process P2
User Space
Thread Library
Thread Library
Kernel Space
CPU
CPU
CPU
CPU

Benefits of Multithreading

- Responsiveness
- Resource sharing, hence allowing better utilization of resources.
- Economy. Creating and managing threads becomes easier.
- Scalability. One thread runs on one CPU.
- In Multithreaded processes, threads can be distributed over a series of processors to scale.
- Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

Multithreading Realtime Examples

- Background jobs like running application servers like Oracle application server, Web servers like Tomcat etc which will come into action whenever a request comes.
- Typing MS Word document while listening to music.
- Games are very good examples of threading. You can use multiple objects in games like cars, motor bikes, animals, people etc. All these objects are nothing but just threads that run your game application.
- Railway ticket reservation system where multiple customers accessing the server.
- Multiple account holders accessing their accounts simultaneously on the server. When you insert a ATM card, it starts a thread for perform your operations.

fork()

- Fork system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process).
- After a new child process is created, both processes will execute the next instruction following the fork() system call.
- A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.

Zero: Returned to the newly created child process.

Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

To install C on Centos

 # yum groups install Development Tools

How to create programme

nano fork.c

To compile

gcc fork.c

To run

#./a.out

fork() in C

There are no arguments in fork() and the return type of fork() is integer. You have to include the following header files when fork() is used:

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

When working with fork(), <sys/types.h> can be used for type ***pid_t*** for processes ID's as pid_t is defined in <sys/types.h>.

The header file <unistd.h> is where fork() is defined so you have to include it to your program to use fork().

The return type is defined in <sys/types.h> and fork() call is defined in <unistd.h>. Therefore, you need to include both in your program to use fork() system call.

Example 1: Calling fork()

Consider the following example in which we have used the fork() system call to create a new child process:

CODE:

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

int main()
{
    fork();
    printf("Using fork() system call\n");
    return 0;
}
```

OUTPUT:

```
Using fork() system call
Using fork() system call
```

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

int main()
{
    pid_t p;
    p = fork();
    if(p==-1)
    {
        printf("There is an error while calling fork()");
    }
    if(p==0)
    {
        printf("We are in the child process");
    }
    else
    {
        printf("We are in the parent process");
    }
    return 0;
}
```

OUTPUT:

We are in the parent process
We are in the child process

To create Zombie process

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

int main()
{
int pid=fork(); //create new process

if(pid>0)
{
        sleep(60); //parent sleeps for 60 sec

}
else
{       exit(0);  // child exits before the parent & child becomes zombie

}
 return 0;
}
```

OUTPUT

 # ./a.out &

[1] 9006

 # pstree -p 9006

a.out(9006) ------ âââa.out(9007) // 9007 is child PID

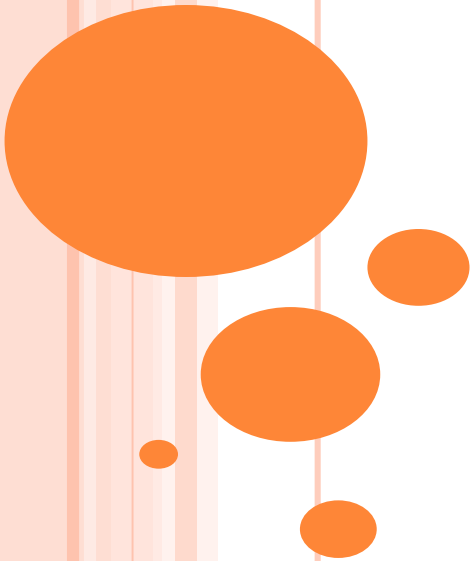
 # ps -aux | grep 9007

user1 9007 0.0 0.0 0 0 pts/0 Z 12:19 0:00 [a.outct>

To create Orphan process

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

int main()
{
int pid=fork(); //create new process

if(pid>0)
{
        exit(0);  //parent exit before child

}
else if(pid==0)
{

sleep(60);  // child sleeps for 60 second
}
 return 0;
}
```

```
[user1@localhost shell]$ ./a.out &
[1] 10506
[1]+  Done                    ./a.out

[user1@localhost shell]$ ps -aux | grep a.out

user1     10510  0.0  0.0   4212    88 pts/0    S    12:32   0:00 ./a.out
user1     10515  0.0  0.0 112812   984 pts/0    R+   12:33   0:00 grep --color=auto a.out

[user1@localhost shell]$ ps -o ppid 10510
 PPID
    1

[user1@localhost shell]$ pstree -p 1

systemd(1)─┬─ModemManager(6586)─┬─{ModemManager}(6639)
           │                    └─{ModemManager}(6642)
           ├─NetworkManager(6726)─┬─dhclient(7036)
           │                      ├─{NetworkManager}(6736)
           │                      └─{NetworkManager}(6738)
           ├─VGAuthService(6584)
           ├─a.out(10510)
           ├─abrt-watch-log(6588)
           ├─abrt-watch-log(6590)
```



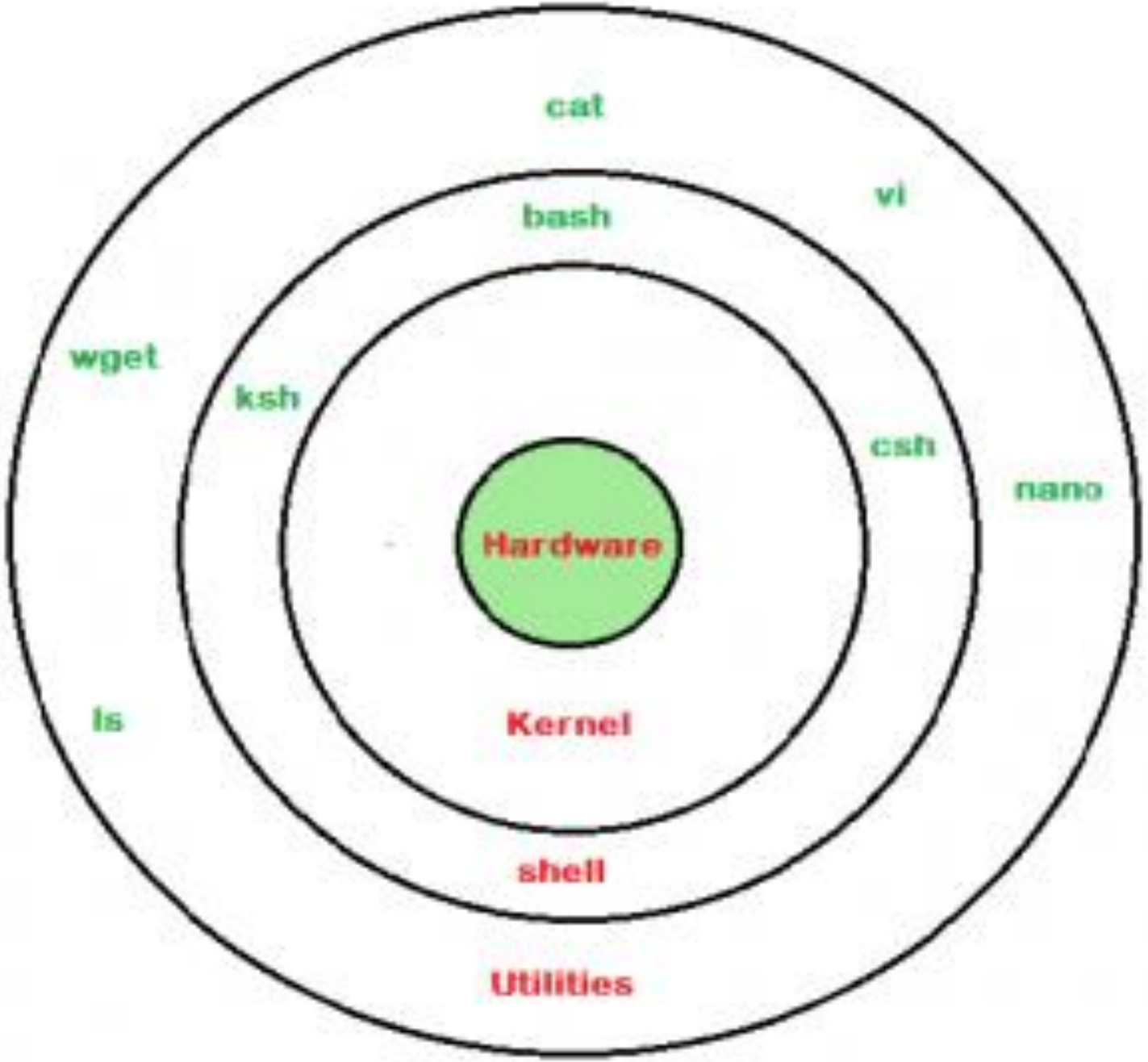
Shell Scripting

What is Shell

- A shell is special user program which provide an interface to user to use operating system services.
- Shell accept human readable commands from user and convert them into something which kernel can understand.
- It is a command language interpreter that execute commands read from input devices such as keyboards or from files.

What is Shell

- A shell is special user program which provide an interface to user to use operating system services.
- Shell accept human readable commands from user and convert them into something which kernel can understand.
- It is a command language interpreter that execute commands read from input devices such as keyboards or from files.


cat
vi
bash
wget
ksh
csh
nano
Hardware
ls
Kernel
shell
Utilities

Types of shells

There are four shells

- Bourne shell(sh),
- Korn shell(ksh),
- C shell(csh) and
- Bourne Again Shell (bash).

Basic Shell Programming

- A script is a file that contains shell commands
  - data structure: variables
  - control structure: sequence, decision, loop
- Shebang line for bash shell script:

```
#! /bin/bash
#! /bin/sh
```

- to run:
  - make executable: `% chmod +x script`
  - invoke via: `% ./script`

Bash shell programming

- Input
  - prompting user
  - command line arguments
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select
- Functions
- Traps

Variable

- A variable is a character string to which we assign a value.
- The value assigned could be a number, text, filename, device, or any other type of dat
- Valid variables
- _abc
- Ab_c
- Ab_1
- Invalid Variables
- 1_ab
- -ab
- Ab-cd
- Ab_c!

SPECIAL SHELL VARIABLES

Parameter	Meaning
$0	Name of the current shell script
$1-$9	Positional parameters 1 through 9
$#	The number of positional parameters
$*	All positional parameters, “$*” is one string
$?	Return status of most recently executed command
$$	Process id of current process

Examples: Command Line Arguments

```
% set tim bill ann fred
      $1  $2   $3  $4
% echo $*
tim bill ann fred
% echo $#
4
% echo $1
tim
% echo $3 $4
ann fred
```

The ‘set’ command can be used to assign values to positional parameters

User input

- shell allows to prompt for user input

Syntax:

```
read varname [more vars]
```

- or

```
read -p "prompt" varname [more vars]
```

- words entered by user are assigned to **varname** and "**more vars**"
- last variable gets rest of input line

User input example

```
#! /bin/bash
read -p "enter your name: " first last

echo "First name: $first"
echo "Last name: $last"
```

OPERATORS

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

Arithmetic Operators

- shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr or bc**.
- C=`expr 1 + 1`
- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between ` `, called the backtick.

Floating Point

- Bc - **bc** command is used for command line calculator.
- **Input : $ echo "12+5" | bc**
- **Output : 17**
- **Input : $ echo "10/2" | bc**
- **Output : 5**
- **Input: $ echo "5.1*2.1" | bc**
- **Output : 10.7**
- **Input: $ echo “scale=1;5/2" | bc**
- **Output : 10.7**
- **[root@localhost ~]# div=`echo "scale=1;5/2" | bc`**
- **[root@localhost ~]# echo $div**
- **[root@localhost ~]# 2.5**

- **How to store the result of complete operation in variable?**

  Example:

- **Input:**
- **$ x=`echo "12+5" | bc`**
- **$ echo $x**
- **Output:17**

USIING EXPR

```
#!/bin/bash

read -p "enter two numbers=  " a b

add=`expr $a + $b`

echo "add=$add"

dev=`expr $a / $b`

echo "dev=$dev
```

USING BC

```
#!/bin/bash

read -p "enter two numbers=  " a b

add=`echo “$a+$b” | bc`

echo "add=$add"

dev=`echo “scale=1;$a/$b” | bc`

echo "dev=$dev
```

Arithmetic Operators

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b`
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b`
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b`
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a`
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a`
= (Assignment)	Assigns right operand in left operand	a = $b
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b]
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b]