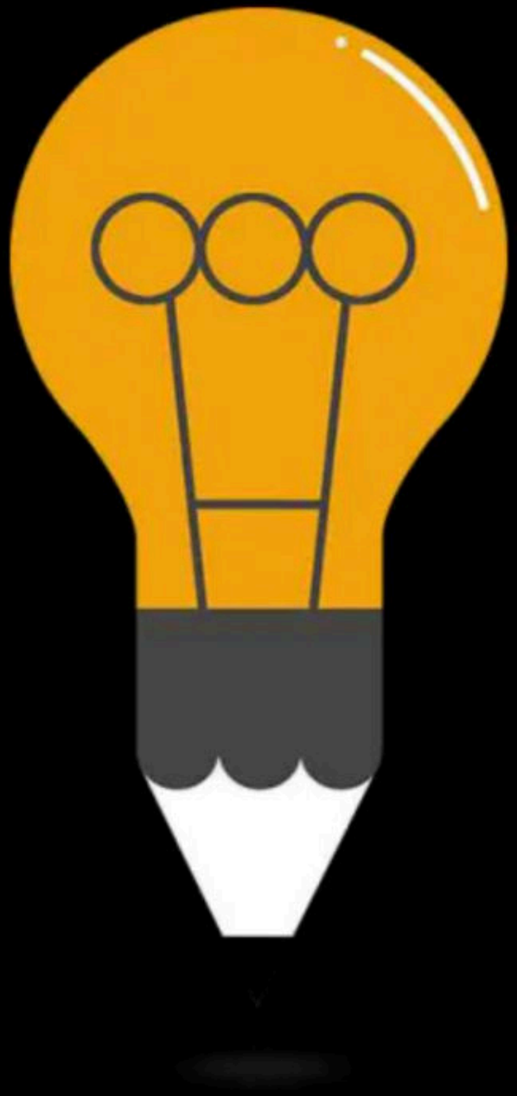
unacademy

Deadlock: Introduction & Prevention

Comprehensive Course on Operating System for GATE - 2024/25

Vishvadeep Gothi • Lesson 19 • May 10, 2023

unacademy



Operating System

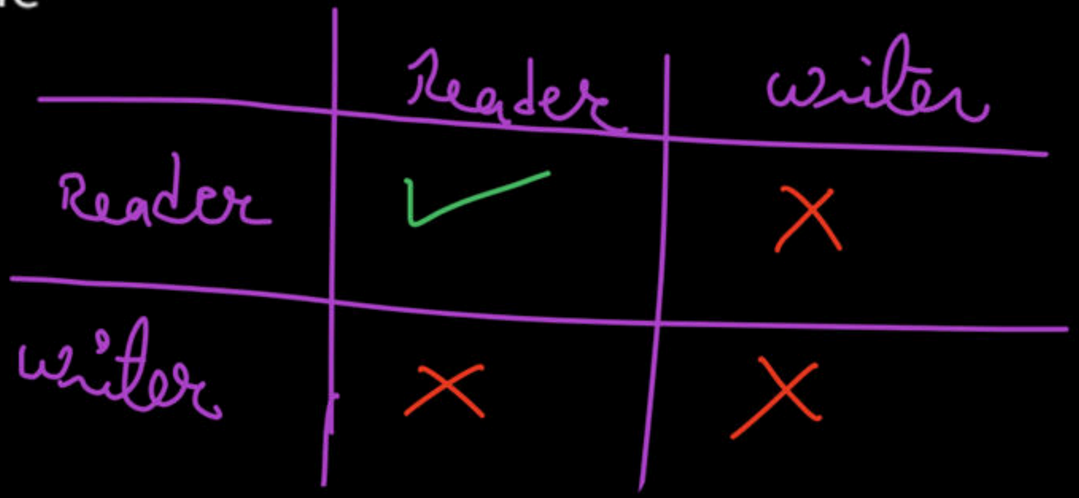
Classical Problems of Synchronization, Multithreading, System Call

By: Vishvadeep Gothi

Reader-Writer Problem

Consider a situation where we have a file shared between many people:

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her
- However, if some person is reading the file, then others may read it at the same time



Reader-Writer Problem: Solution

- If writer is accessing the file, then all other readers and writers will be blocked
- If any reader is reading, then other readers can read but writer will be blocked

Reader-Writer Problem: Solution

- Variables:
  - mutex: Binary Semaphore to provide Mutual Exclusion
  - wrt: Binary Semaphore to restrict readers and writers if writing is going on
  - readcount: Integer variable, denotes number of active readers

- Initialization:
  - mutex: 1
  - wrt: 1
  - readcount: 0

Writer() Process

wait (wrt)

// writing

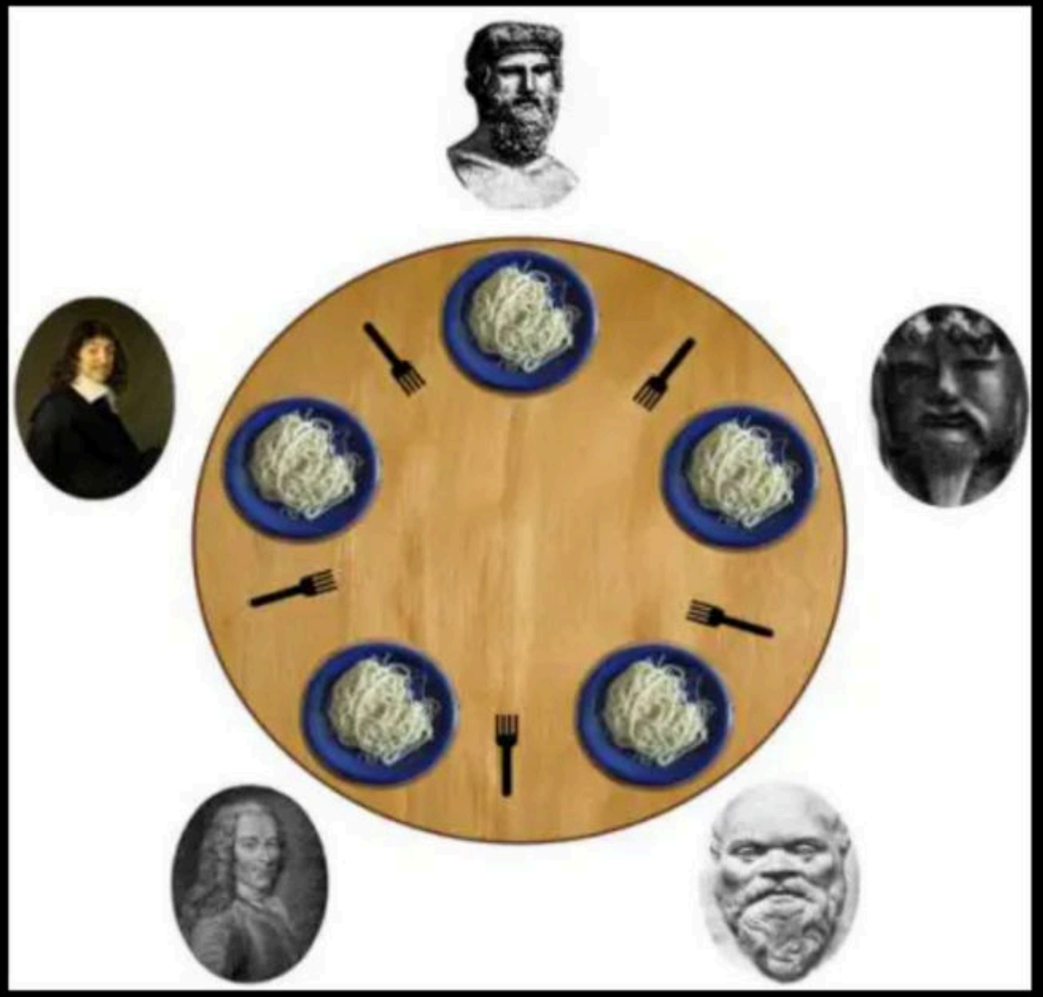
signal (wrt)

Reader() Process

```
wait(mutex)
Readcount ++;
if(Readcount == 1)
    wait(wrt)
Signal(mutex)
    // Reading
wait(mutex)
Readcount --;
if(Readcount == 0)
    signal(wrt)
Signal(mutex)
```

If these 2 statements are removed then at a time only one reader will be allowed.

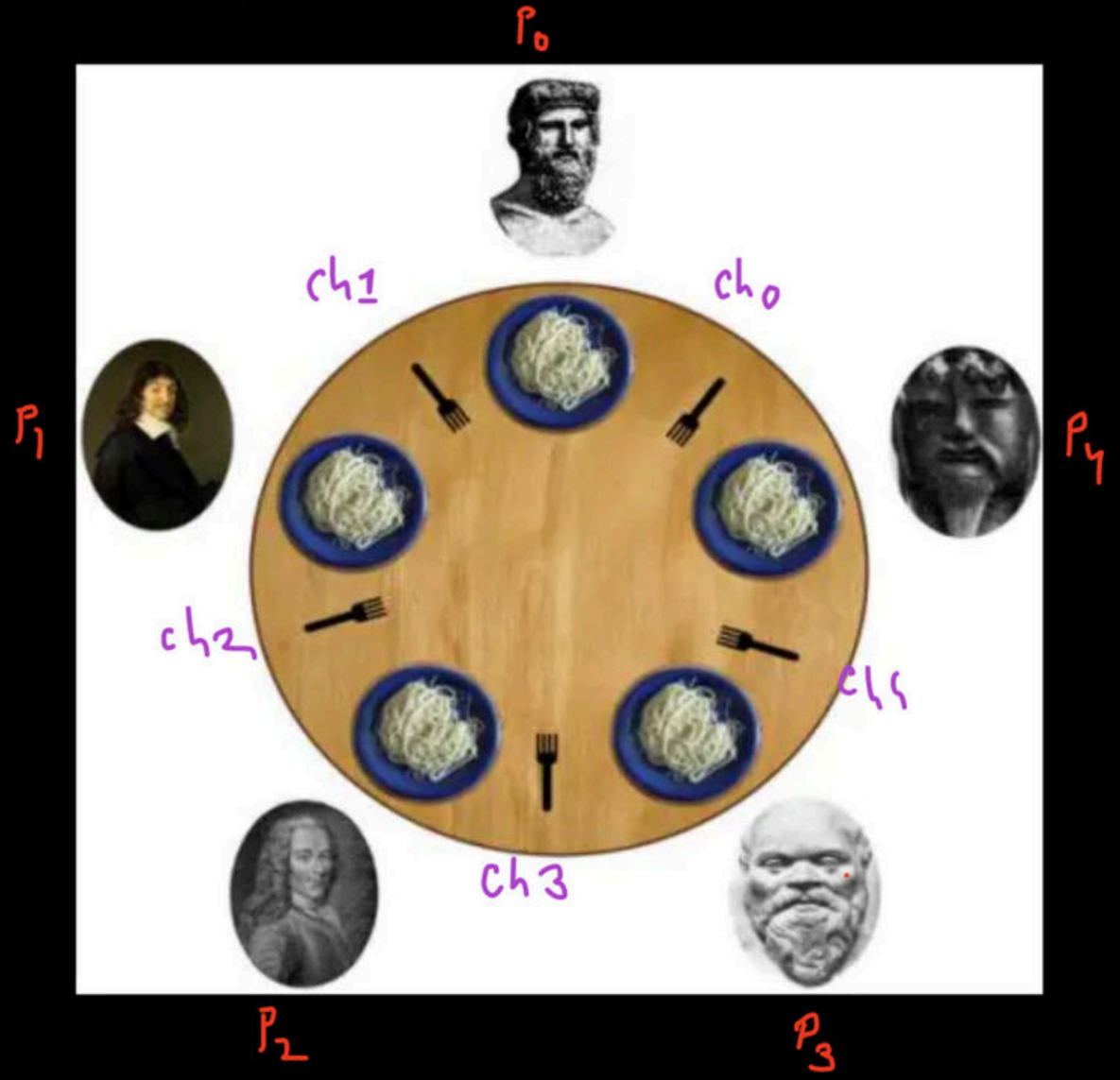
Dining Philosopher Problem



Dining Philosopher Problem

- K philosophers seated around a circular table
- There is one chopstick between each philosopher
- A philosopher may eat if he can pick up the two chopsticks adjacent to him
- One chopstick may be picked up by any one of its adjacent followers but not both

Dining Philosopher Problem: Solution



Dining Philosopher Problem: Solution

An array of binary semaphores of size k, to denote chopsticks

Semaphore chopsticks[k] = {1, 1, ..., 1};

Dining Philosopher Problem: Solution

Process of Philosopher P_i

```
wait(chopstick[i])
wait(chopstick[(i+1) % k])
//Eating
signal(chopstick[i])
signal(chopstick[(i+1) % k])
```

There is a possibility of deadlock.

⇓

when each philosopher picks one chopstick.

Dining Philosopher Problem: Solution

Some of the ways to avoid deadlock are as follows –

1. There should be at most (k–1) philosophers on the table, with k chopsticks

Dining Philosopher Problem: Solution

Some of the ways to avoid deadlock are as follows −

1. There should be at most (k–1) philosophers on the table
2. A philosopher should only be allowed to pick their chopstick if both are available at the same time

Dining Philosopher Problem: Solution

Some of the ways to avoid deadlock are as follows –

1. There should be at most (k–1) philosophers on the table
2. A philosopher should only be allowed to pick their chopstick if both are available at the same time
3. One philosopher should pick the left chopstick first and then right chopstick next; while all others will pick the right one first then left one

Dining Philosopher Problem: Solution

unacademy

Multithreading

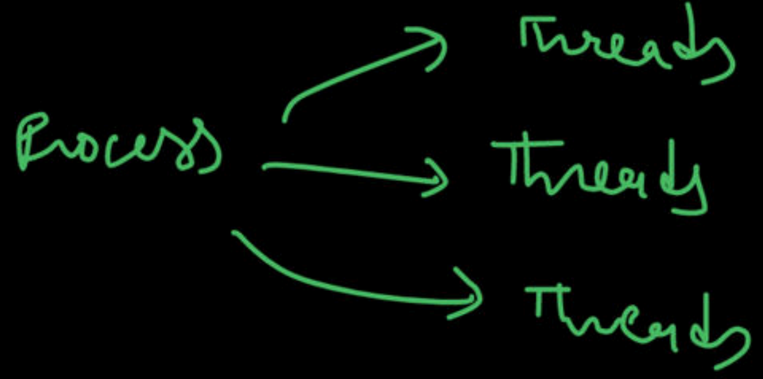
Thread

Component of process
or
Lightweight Process

Threads → some parts of process will be shared

→ some parts will be own.

Provide a way to improve application performance through parallelism



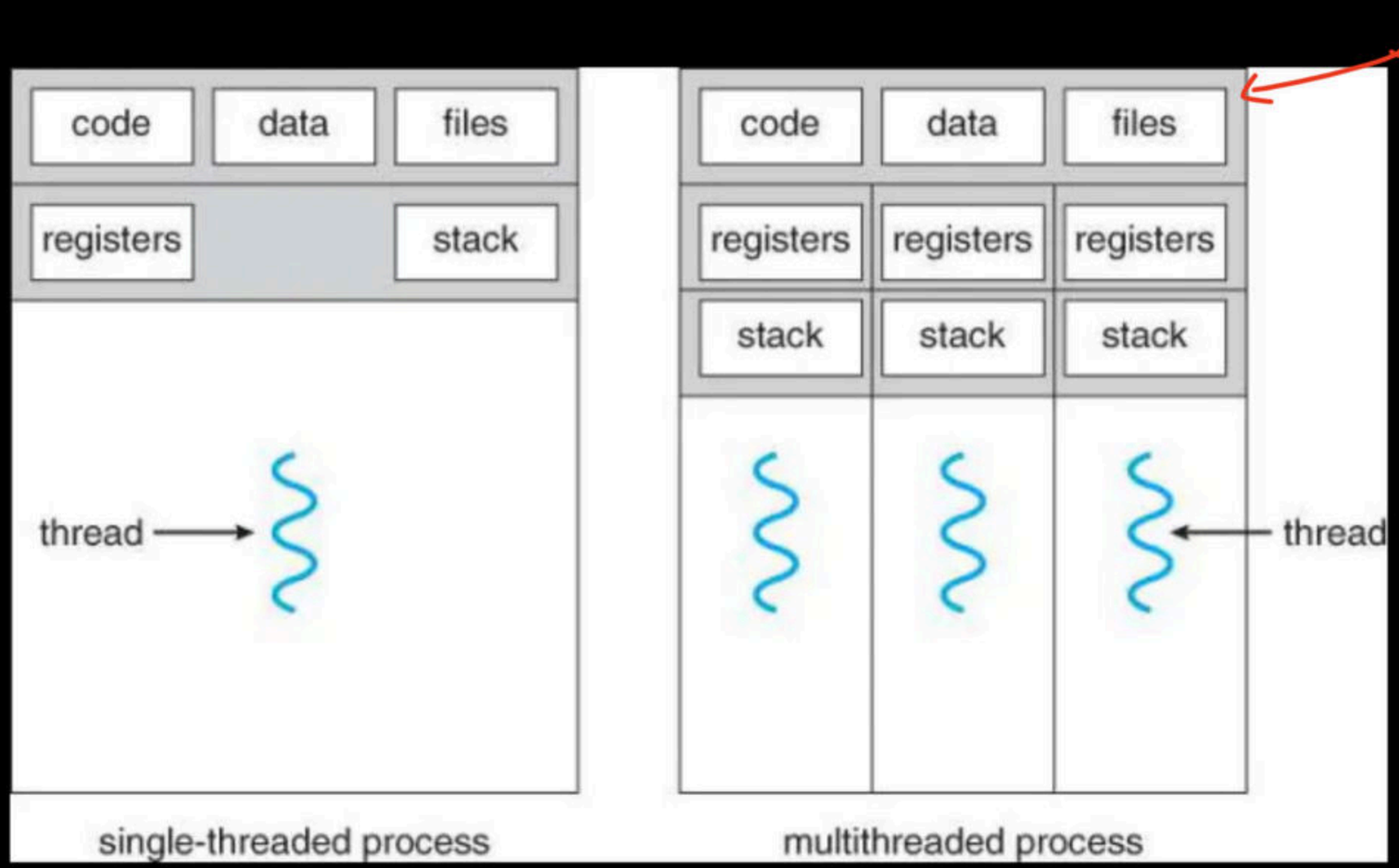
unacademy

Threads

Shared Among Threads	Unique For Each Thread
Code Section	Thread Id
Data Section	Register Set → GPRs, SP, PC, AC, status
OS Resources	Stack ✓
Open Files & Signals	Program Counter

Heap

Threads


shared
code
data
files
registers
stack
thread
code
data
files
registers
registers
registers
stack
stack
stack
thread
single-threaded process
multithreaded process

unacademy

Advantage of Multithreading

- Responsiveness
- Faster Context Switch
- Resource Sharing
- Economy
- Communication
- Utilization of Multiprocessor Architecture

unacademy

Types of Threads

- User Level Thread → multithreading is done in user process
- Kernel Level Thread → —"— —"— OS process

unacademy

Types of Threads

User Threads	Kernel Thread
Multithreading in user process	Multithreading in kernel process
Created without kernel intervention	Kernel itself is multithreaded
Context switch is very fast	Context switch is slow
If one thread is blocked, OS blocks entire process	Individual thread can be blocked
Generic and can run on any OS	Specific to OS
Faster to create and manage	Slower to create and manage

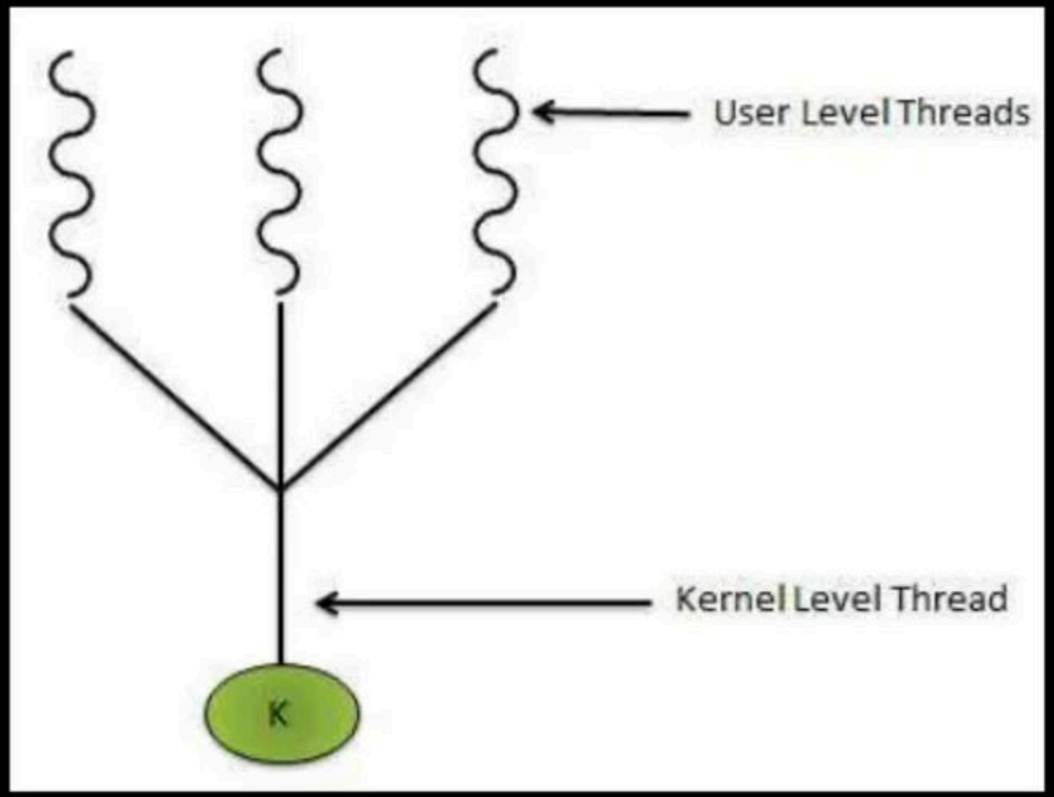
unacademy

Multithreading Model

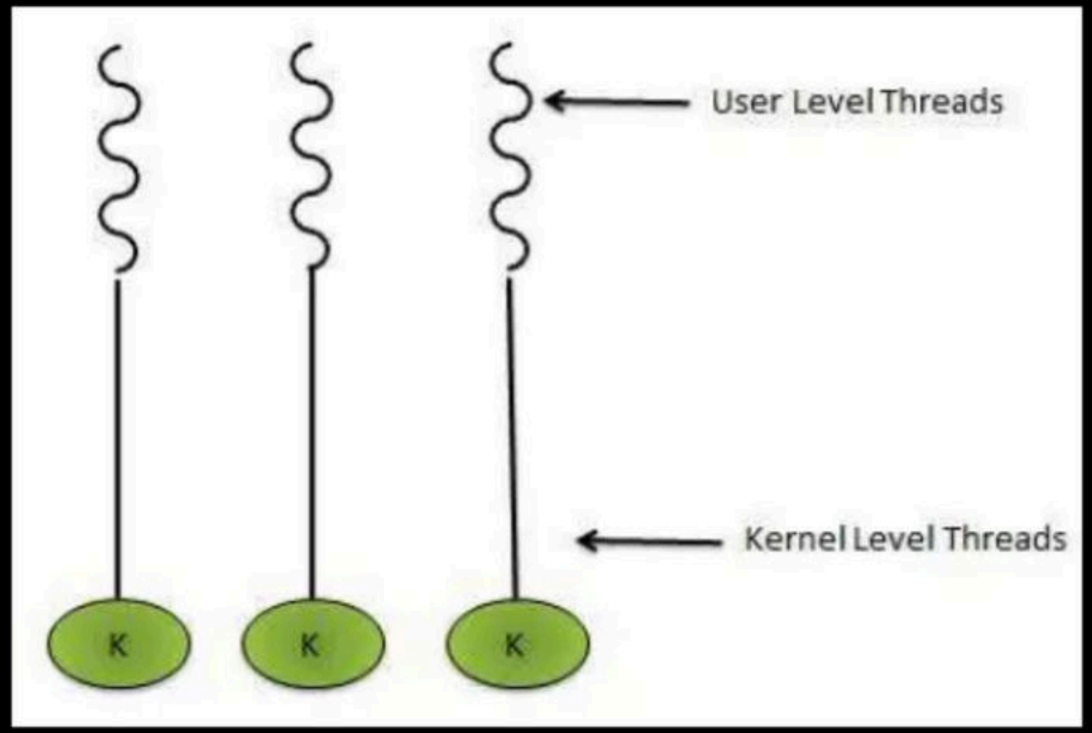
1. Many-to-One Model
2. One-to-One Model
3. Many-to-Many Model

unacademy

Many-to-One

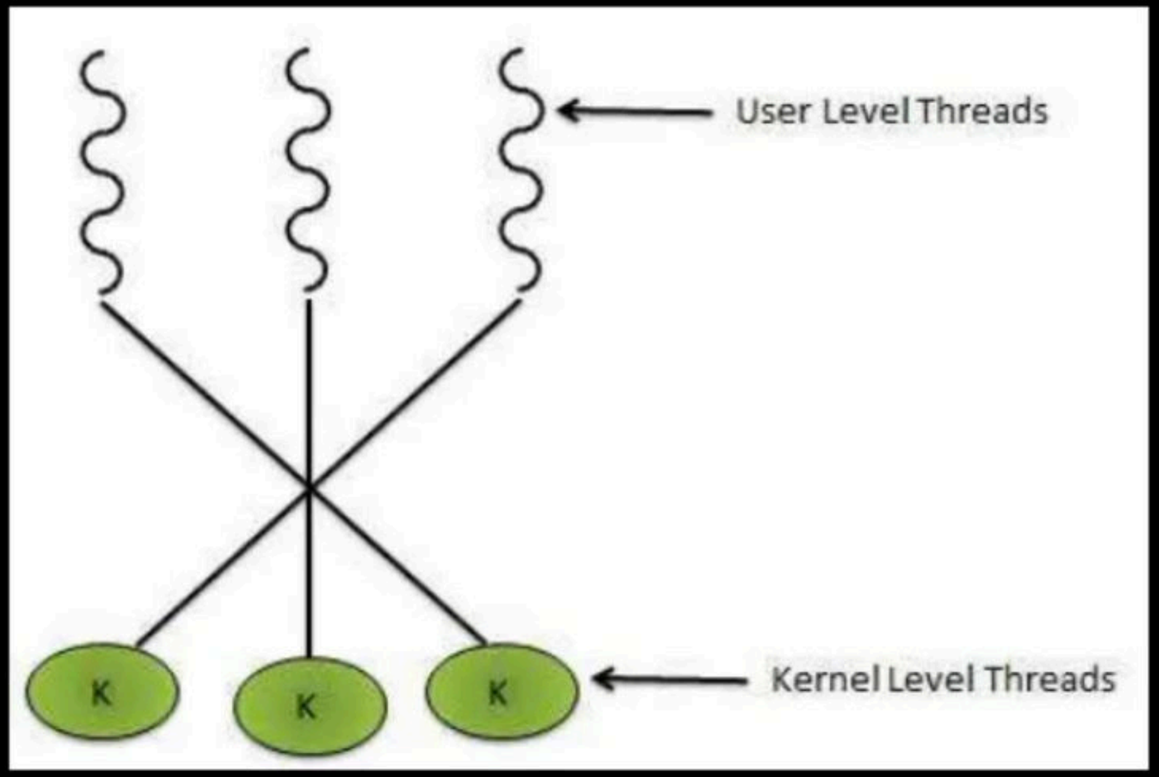


One-to-One



unacademy

Many-to-Many



unacademy

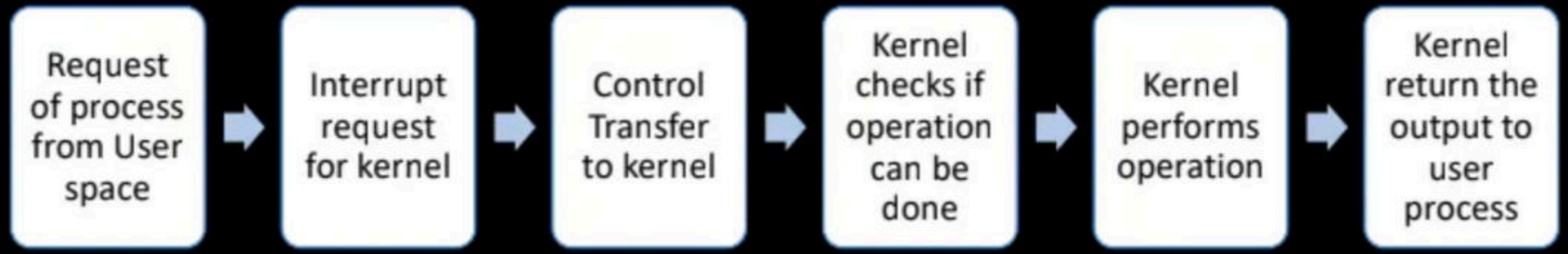
System Call

Programmatic way in which a computer program requests a service from the kernel

unacademy

How System Call Works

s/w



MS word

↓

s/w

System Call

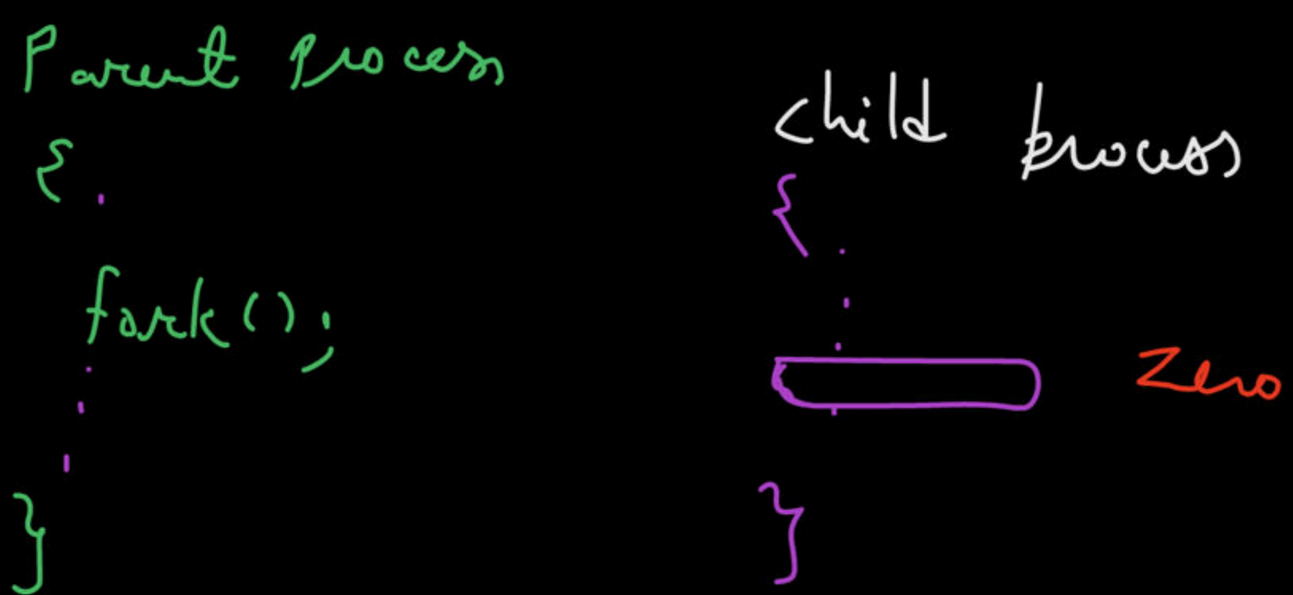
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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

unacademy

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).



unacademy

Parameters and Return Value

It takes no parameters and returns an integer value

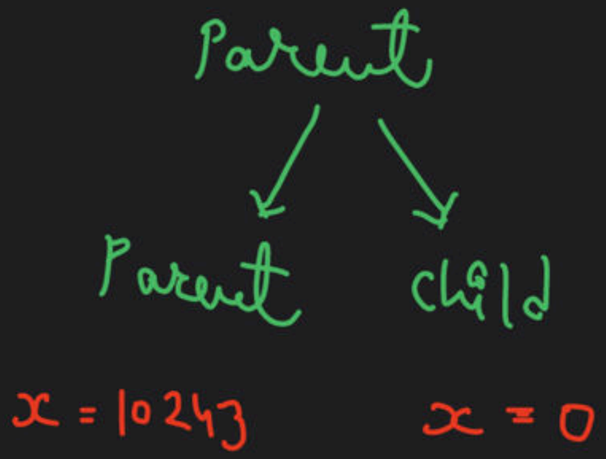
- Negative Value: creation of a child process was unsuccessful
- Zero: Returned to the newly created child process
- Positive value: Returned to parent or caller. The value contains process ID of newly created child process

→ successful

---

child process starts execution ✓ · the fork() call which has created the child. from stmt after

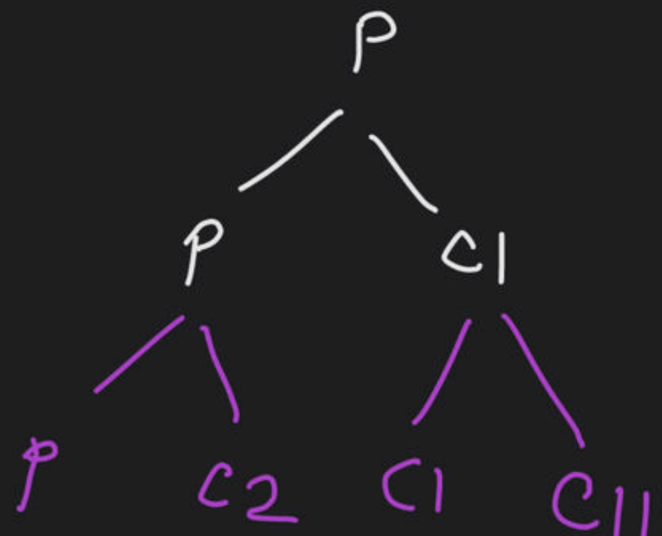
```
void main()
{
  int x;
  x = fork();
  printf("%d\n", x);
}
```



assuming child process id ⇒ 10243

Process

```
fork();
fork();
```



no. of child processes ? => C1, C2, C11

Ans = 3

Total no. of processes = 4

n - fork() calls sequentially

```
fork()
fork()
 ⋮
fork()
```

no. of child processes = $2^n - 1$

Total no. of processes = 2

unacademy

Question GATE-2008

A process executes the code

```
fork ();
fork ();
fork ();
```

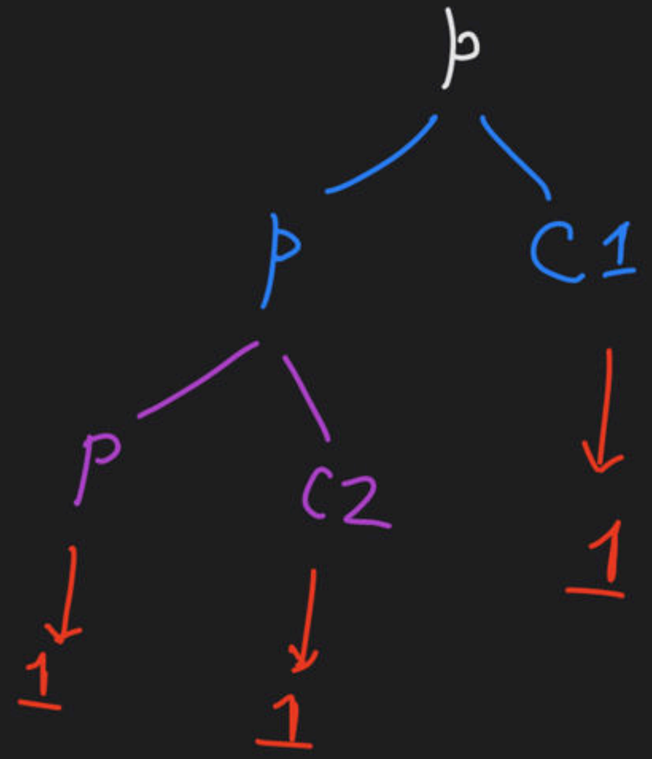
The total number of child processes created is

(A) 3
(B) 4
(C) 7
(D) 8

Question

A process executes the code
fork ();
:
fork ();
There are n such statements. The total number of child processes created is? $= 2^n - 1$

```
void main()
{
  if (fork())
  {
    fork();
  }
  printf("1");
}
```



no. of times 1 printed = ?

3 times

Ans =

unacademy

DPP (1)

```
void main()
{
  if ( fork() || fork())

      fork();

  printf("*");
}
```

no. of times * printed?

unacademy

Happy Learning.!