CSE321: Operating Systems

Topic: Threads

Prepared by:
Saad Bin Sohan

BRAC University

Email: sohan.academics@gmail.com
GitHub: https://github.com/saad-bin-sohan

Threads

- Process is a program in execution.

  A program can be associated with multiple processes.

  ie: Chrome(program) → 3 tabs (~3 process)

- Similarly a single process can have multiple threads.

  → if you divide a process into small seperate work units, those units are called 'threads'.

→ if a process is divided into multiple threads, each thread can be executed independently (of each other) by a scheduler, in parallel.

(so far we studied

- batch
- multiprogramming
- multitasking
- parallel
- distribution)

since we have multiple processors, if we divide a process into multiple threads, and since each thread can execute independently in different core/CPUs, execution becomes faster. So for max CPU utilization, we introduce 'Threads'.

[→ previously, process এর মাঝে context switch এ time ও overhead হতো

→ context switch among threads require much much smaller time]

multi-tasking → time-share করে processগুলো মাঝে frequently context

switch হয়?

→ processes are interactive

multi-programming →

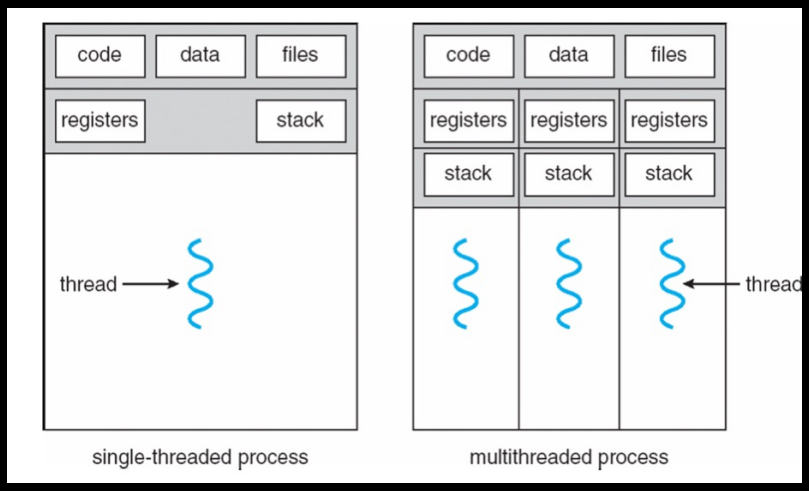
- previously এত process single threaded হতো। now they are multi-threaded.

- Thread creation is less expensive than process. Because each process needs seperate resources, but the the threads under a single process share those resources.

For a Multi-threaded process, all the threads under that process share the code section, data section, OS resources.

→ Each thread has its own -

- Thread ID
- PC
- Register set
- Stack



- Threads cannot exist outside of process. If a process dies, so does their its threads

Multicore Processing

→ introduction of multicore processors brought challenges for programmers

Concurrency vs Parallelism

Concurrency

→ Multiple processes appear to run simultaneously by rapidly (context) switching execution on a single CPU core

Parallelism:

→ Multiple processes run at the same time on multiple CPU cores (requires multi-core processors)

Data Parallelism

→ single instruction এর জন্য

subset of same data gets distributed across multiple cores

→ multiple workers doing the same work (~ multiple processors performing same jobs)

Task Parallelism

→ Multiple instruction on same data

or

Multiple instruction on multiple data

→ works on their own task with same objective in mind

Amhdal's Law

- a program with both serial and parallel parts.

  if an additional core is added it's performance gain,

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Types of threads

2 types:

i) user threads:

→ thread library use করে user program এ thread create, manage করা

→ requires no support from kernel for creation/management

ii) Kernel Threads:

→ threads created and managed by the OS.

- Kernel must support multithread models for multi-threaded user threads to run

- CPU can only execute Kernel threads directly.

So there must be proper relation between user threads and kernel

threads for user threads to execute. That relationship is called 'Multithreading Models'.

3 types ≈ there can be 3 types of relationships between user threads and kernel threads.

i) Many to One Model:

→ thread is managed by thread library in user space

→ if a user threads performs block() system call, it blocks the entire process

→ multiple threads are unable to run in parallel

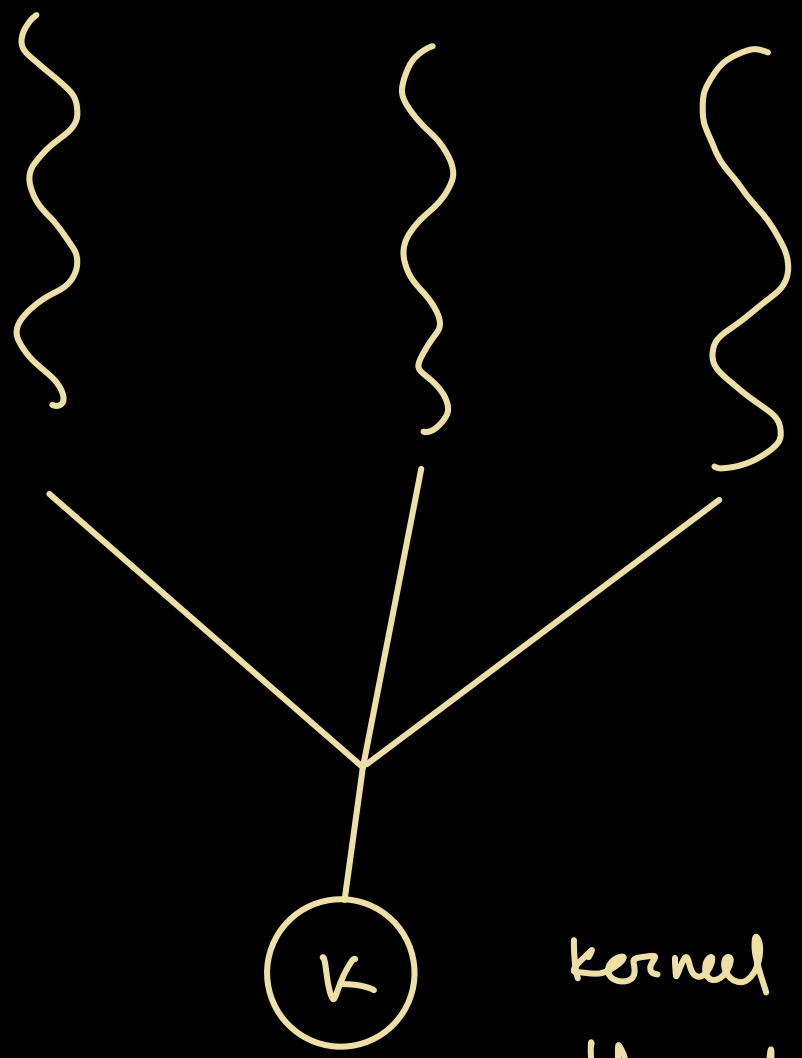


figure: many to one model

One to One model

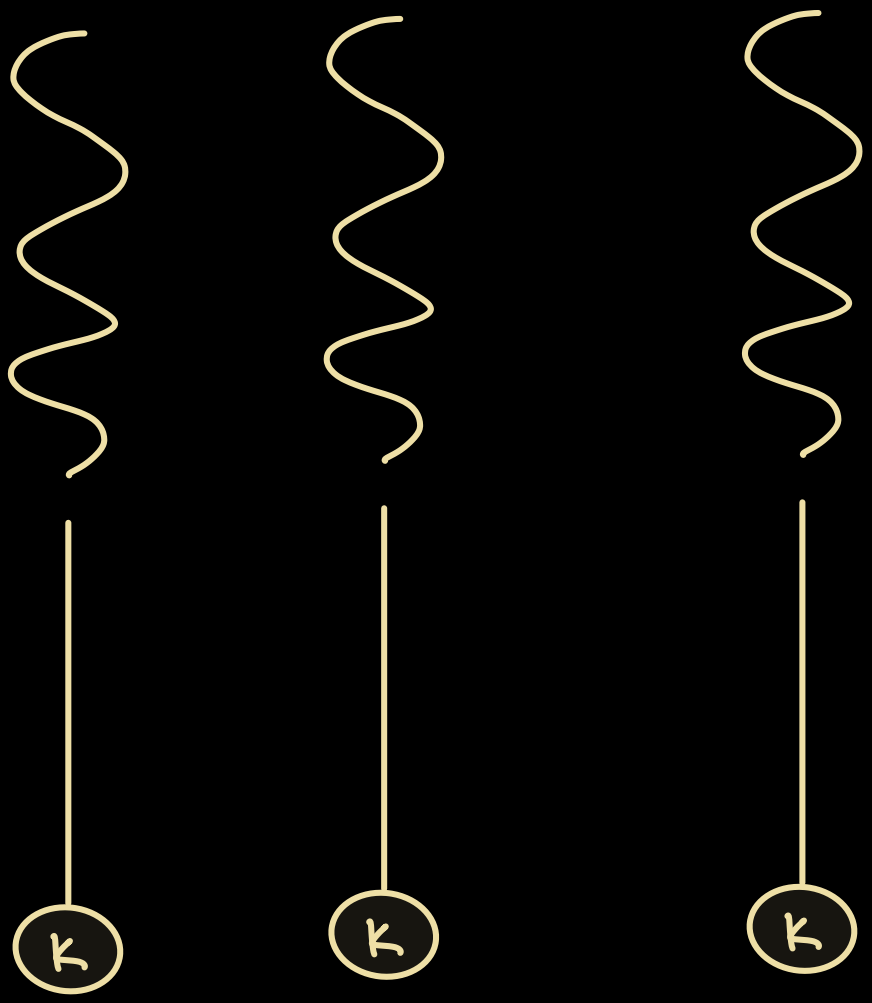
One user thread —mapped to→ exactly one kernel thread

→ creating a user thread creates a kernel thread

→ allows more concurrency: If a thread is blocking a system call, another thread is allowed to run in

their own corresponding Kernel thread

→ multiple threads can run in parallel



Many to Many Model

to solve problem of one to one model,

→ multiplexes many user level threads to a smaller or equal number of kernel threads

→ user can create as many user threads as he wants.
but depending on CPU / core count, they will be multiplexed.

Two level Models

→ (many to many) + (1 to 1)

supports both

Thread Libraries

→ Thread library provides programmers with the API for creating and managing threads

⊞ two ways of implementing a library

way 1) Implement a user level library with no support required from kernel. Thread creation and management is done in user space.

No system call is required.

way 2) Implement a kernel level library

(code and data structures exist in the kernel space)

Major 3 thread libraries:

i) POSIX threads

→ used in both user and kernel level

ii) Win32

→ used only in kernel levels

iii) JAVA

→ OS যে threading structure follow করে, সেই structure অনুযায়ী multithreading implement করে

Pthreads (library)

→ usable at both user level and kernel level

→ an extension of POSIX, used for thread creation and synchronization

thread তুলো কোন ore

→ specification for thread behaviors (its not an implementation

Implicit Threading

↳ popular nowadays (like JAVA threading library)

→ multiple threads have been extracted from one single sequential program.

implicit multithreading is the concurrent execution of those threads

→ creation and management of threads are done by compilers and libraries and not by programmers

3 methods of implicit threading:

i) Thread Pools

example: webpages loading from webservers

→ a collection of (a fixed number of) pre-created threads

→ a queue that holds the tasks waiting to be executed

→ Thread manager assigns tasks to available threads and handles thread creation/destruction

How thread pool works:

→ a new task arrives (eg: user clicks a button on a website)

→ the task is placed in task queue

→ the thread manager checks for idle threads in the pool

→ if there's an idle thread, the manager assigns tasks to that thread

→ the the thread executes the task

→ when task is completed the thread returns to the pool and became idle

→ if all threads are busy, and the task queue is full, the system may reject new task, wait until a thread becomes availabe, or create new threads upto its max limit

ii) OpenMP

iii) Grand Central Dispatch

Threading Issues

i) fork() and exec():

↳ creates a child process

↳ the program specified in exec() will replace the entire process

issue: does fork() duplicate that specific thread or all the threads of that process?

solution: have two version of fork(): one that duplicates all

threads, or that specific

fork called thread.

so which version of fork() to use and

when?

=> 1) fork() ২০ immediate ৩৭০

exec() চালালে duplicate single

thread

2) না চালালে duplicate all

threads

Thread Cancellation

"Target Thread" => thread to be cancelled

2 approaches:

1) Asynchronous cancellation:

→ one thread terminates the 'target thread'

2) Deferred cancellation:

→ allows the target threads to periodically check if it

should be cancelled

ii) Signal Handling :

→ Signals are generated if something goes wrong in the system

→ s

iii) Thread local storage