CMPT 300 D100
Operating System I
Threads – Chapter 4

Dr. Hazra Imran

Threads (Having concurrency within a single process)

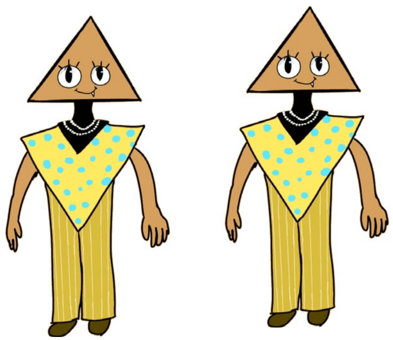
- Understand what a thread is and how it differs from a process.
- Consider the pros and cons of user threads vs kernel threads.
- Write multithreaded code

Threads

- Goal is to build applications that full utilizes many CPUs

Method 1

- Build applications in which many processes communicates.
- Example: Browser (process per tab)
- Communication methods: Pipes, sharing memory
- Issues?



Method 2

- Use new abstraction : **threads**
- Threads are just like processes, but they share the address space.

Threads

- ... a sequence of instructions.
- A normal sequential program consists of a single thread of execution.
- Threads allow a single process to do multiple things at once.
- Two uses for threads:
  - Multiple tasks at once (e.g., UI thread, computation thread)
  - Performance (multicore, multiprocessor)
- In threaded concurrent programs there are multiple threads of execution, all occurring at the same time.
- Threads may perform the same task. Threads may perform different tasks

Concurrency vs Parallelism

process

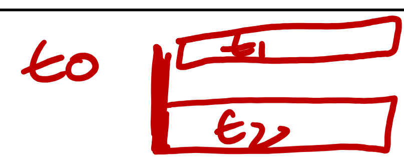
t1 t2 t3

Concurrency

- Concurrency is breaking up a task into smaller tasks (threads) which run independently from one another
- rapid switching of processes onto the CPU, which gives the illusion that multiple processes are executing at once

Parallelism

- Parallelism is when two different tasks (threads) are actively running at the same instant of time
- have hardware support to execute multiple things at once
- **Core:** physical unit of code execution (instruction decoding, registers, etc.)
- **CPU:** Nominally (today), the computing hardware on a single chip.
    - Can contain multiple cores



Thread analogy

Throwing a birthday party

Single-Threaded approach

Prepare a guest list

Book a place

Order a cake

Bake cookies

Cook food items

Enjoy the party and make sure guests are happy and having fun!

Attend the guests

Buy return gifts

Decorate the place

Multi-Threaded approach

Prepare a guest list

Book a place

Order a cake

Buy return gifts

Cook food items

Bake cookies

Decorate the place

Attend the guests

Enjoy the party and make sure guests are happy and having fun!

Note: Ignore synchronisation issues for now

Why threads?

Thread 1 - handle user req + mouse clicks.

T2 → downloading pages + web server

- **Resource Utilization:** blocked/waiting threads give up resources, i.e., the CPU, to others.

T3 → update & display files

- **Parallelism:** multiple threads executing simultaneously; improves performance.

- **Responsiveness:** dedicate threads to UI, others to loading/long tasks.

- **Priority:** higher priority →more CPU time, lower priority→ less CPU time.

- **Modularization:** organization of execution tasks/responsibilities.

divide & conquer.

Threads

Each thread has its own:

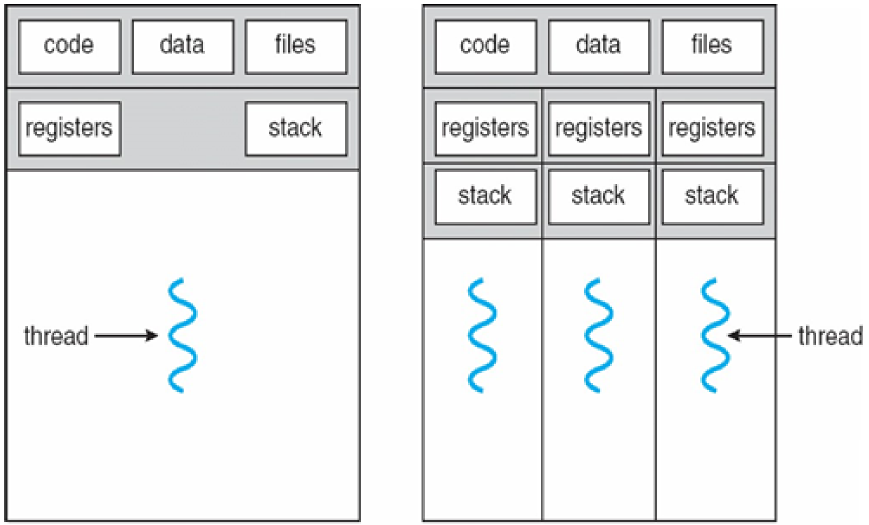
- Thread ID (assigned by the OS)
- Program Counter (which instruction it currently executes)
- Registers Set (which values are stored in registers)
- Stack (bookkeeping of its function/method invocations)

But it shares with other threads in the same process

- the code/text section
- the data segment (global variables)
- the list of open file descriptors (at the moment of thread creation) the heap
- the signal behaviors (handlers)

Processes and Threads

- Processes have one sequential **thread** of execution
- Increasingly, operating systems offer the ability to have multiple concurrent threads of execution in a process
  - Individual threads can execute only one instruction at a time
  - Multiple threads in a process allow multiple tasks to be performed **concurrently**, at the same time



single-threaded process   multithreaded process

Processes and Threads

- **Single-threaded process**
  - Per-process items:
    - Address space / page table
    - Program text (i.e. the code)
    - CPU registers
    - Program counter
    - Stack and stack pointer
    - Global variables
    - Memory heap
    - Signal handlers
    - Open files, sockets, etc.
    - Child processes

- **Multi-threaded process**
  - Per-process items:
    - Address space / page table
    - Program text
    - Global variables
    - Memory heap
    - Signal handlers
    - Open files, sockets, etc.
    - Child processes
  - Per-thread items:
    - CPU registers
    - Program counter
    - Stack and stack pointer

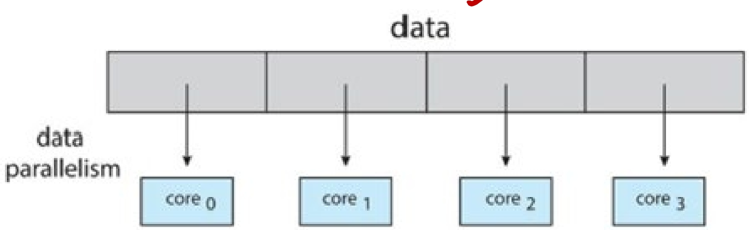
Process and Thread

- A process defines the address space, text, resources, etc.,
- A thread defines a single sequential execution stream within a process (PC, stack, registers).
- Threads extract the thread of control information from the process
- Threads are bound to a single process. Each process may have multiple threads of control within it.
- The address space of a process is shared among all its threads.
- No system calls are required to cooperate among threads.
- Simpler than message passing and shared-memory.

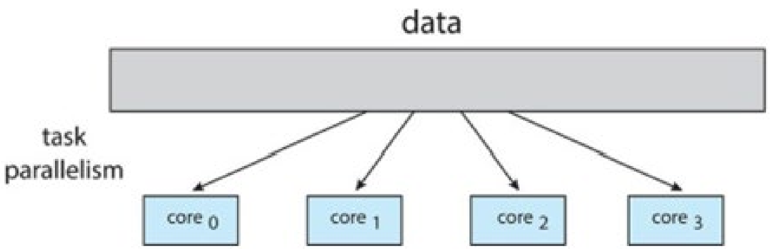
Multi-Thread Programming

Types of parallelism

- **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each.



- **Task parallelism** – distributing threads across cores, each thread performing unique operation.



Clicker

Only the threads can take advantage of multiple CPUs

(A) True
(B) False

Communication between processes and threads

- Process-level operations: fork() and exec()

- Should fork() copy all currently running threads? Or just the one that called fork()?

  - Some OSes provide both types of fork()

  - Which one we choose depends on what the parent/child do next

  →If the child calls exec() immediately after being created, then we probably don't need to copy all of the other threads

Thread

Advantages

1 context switching (low)

2 NO IPC

3 parallilism

Disadvantages

1 shared state

2 faulty thread

3 memory protection (X)

Types of threads

Threads are implemented in following two ways

User Level Threads − User managed threads.

Kernel Level Threads − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

- OS does not know about.
- The OS only knows about the process containing the threads.
- The OS only schedules the process, not the threads within the process.
- The programmer uses a thread library to manage threads (create and delete them, synchronize them, and schedule them).

Kernel Level Threads

(LWP)

- also known as a lightweight process.
- Switching between kernel threads of the same process requires a small context switch.
  - The values of registers, program counter, and stack pointer must be changed.
  - Memory management information does not need to be changed since the threads share an address space.
- The kernel is responsible for scheduling the threads.

User Level Threads – Advantages and Drawbacks

☺ Low overhead

- Fast switching between threads (because no OS involvement)

☺ User-level thread scheduling is more flexible ✓

- Each process might use a different scheduling algorithm for its own threads.
- A thread can voluntarily give up the processor by telling the scheduler it will yield to other threads.

☹ Can't use multiple cores: The OS doesn't know anything about user-level threads

☹ Threads must cooperate: If a user-level thread is waiting for I/O, the entire process will wait.

Kernel Level Threads – Advantages and Drawbacks

☺ Can use multiple cores: Since the OS is aware of all multiple threads, it can put them on different cores

☺ Threads don't have to cooperate: The OS will take care of scheduling

☹ Higher overhead

- System calls required to create/terminate threads

Multithreading Models

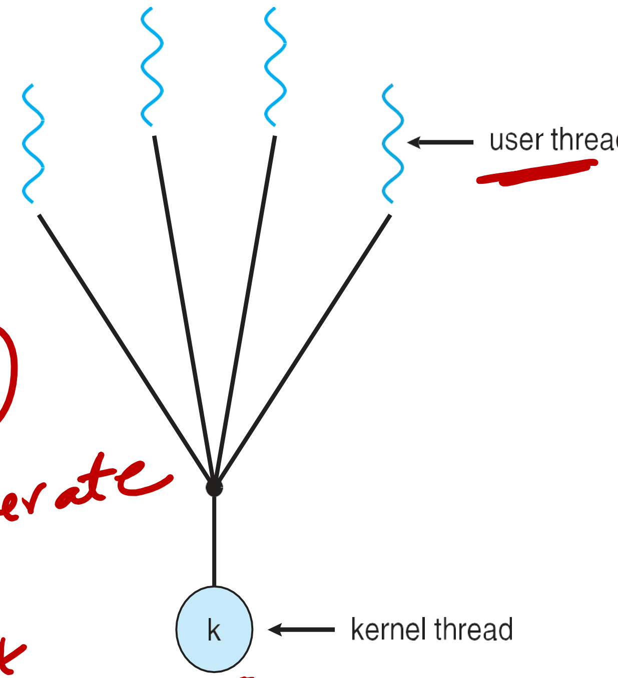
Relationship between user space threads and kernel threads.

Options include:

- Many-to-One
- One-to-One
- Many-to-Many
- Two-Level Model

Many-to-One -(k)

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Few systems currently use this model. Examples:
  - Solaris Green Threads
  - GNU Portable Threads

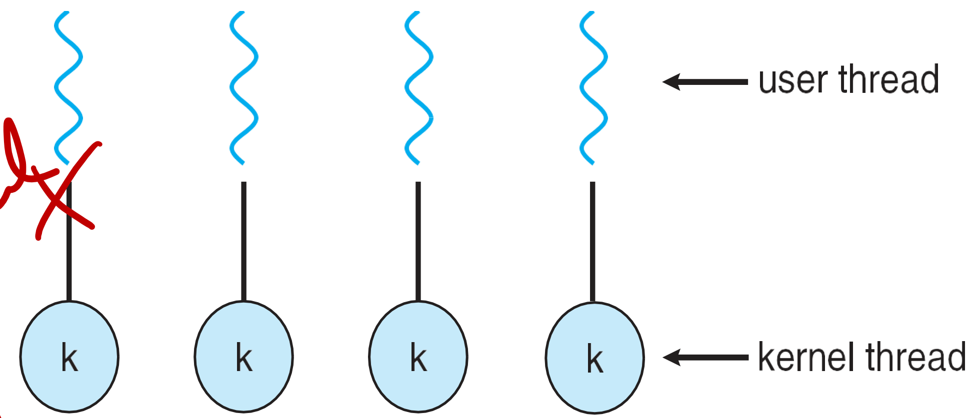


+
1) multi threading is efficient
2) low overhead

−
cooperate
can't use multi core

One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



+

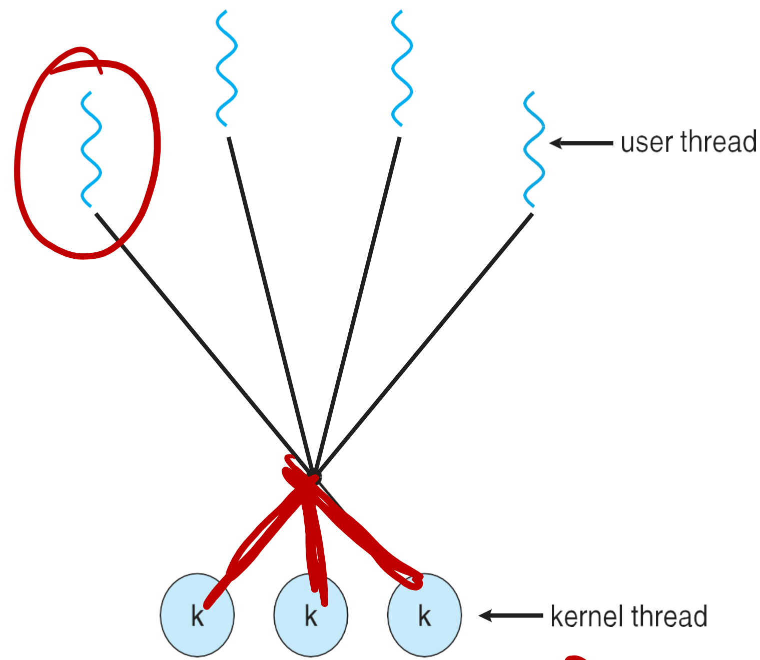
- less prone to failure
- no cooperation require
- multiple core

−

higher overhead

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Example: Solaris 9 and earlier

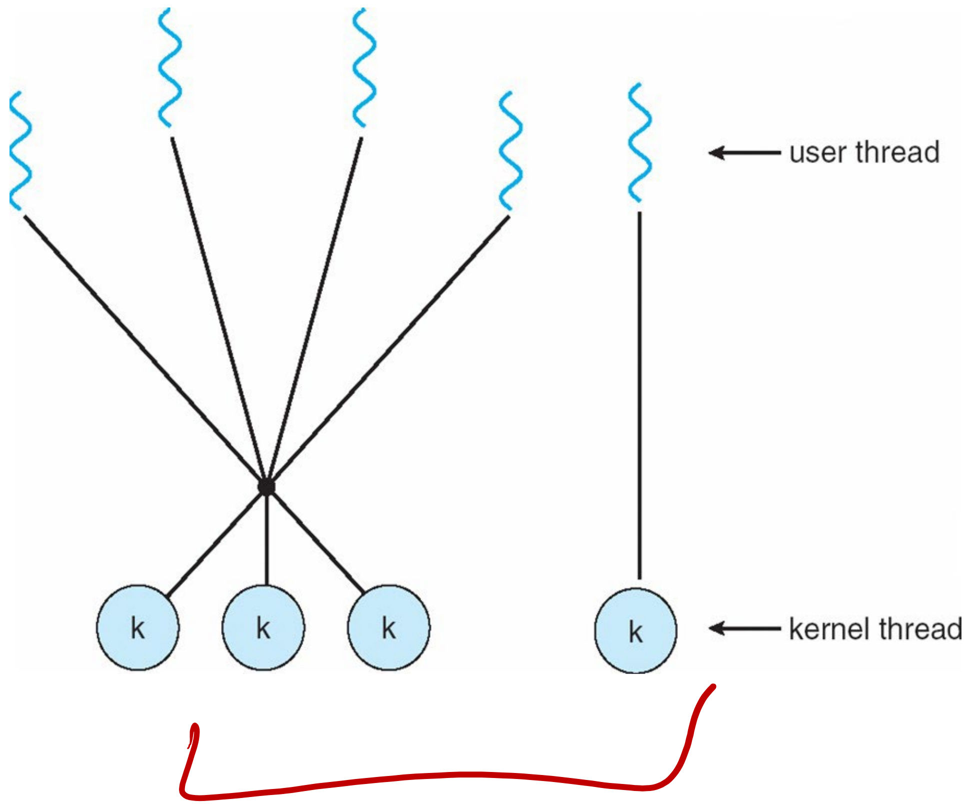


multple core (yes)

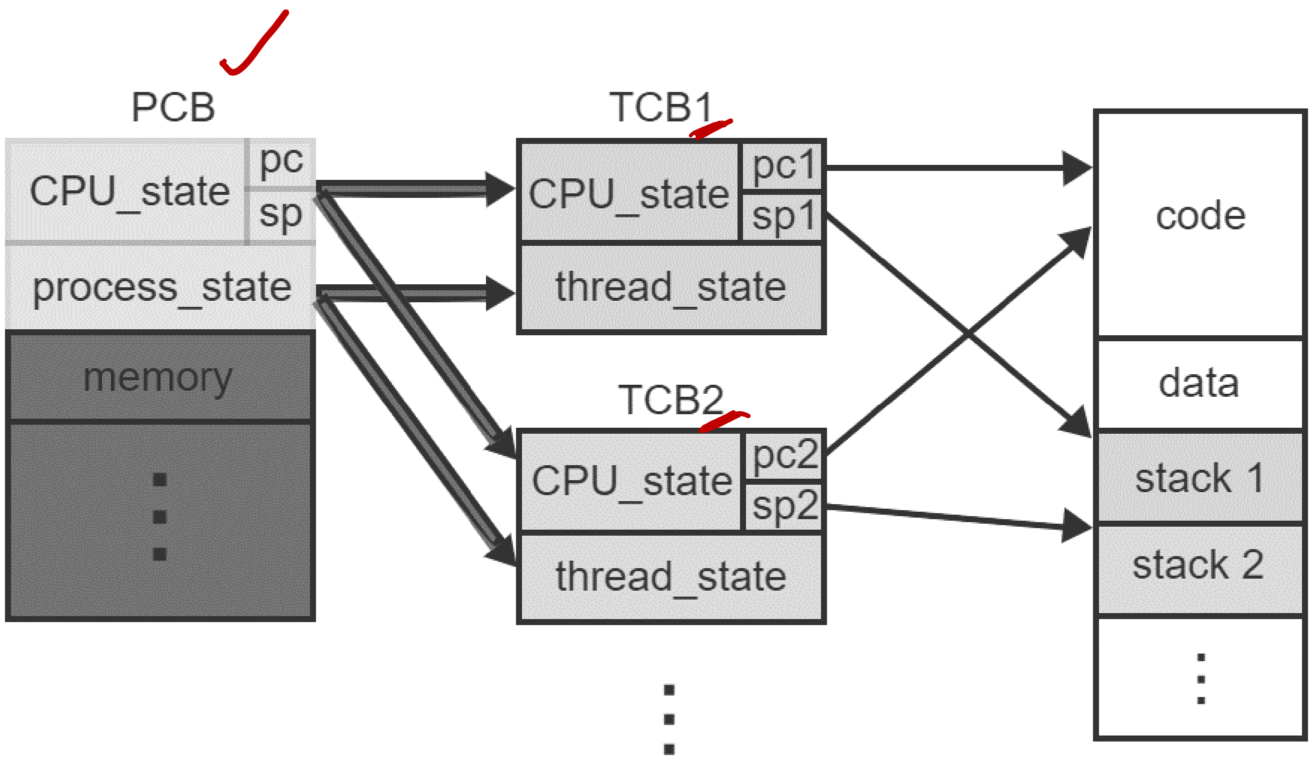
X user thread $\leftrightarrow$ kernel thread

Two level model

Examples: IRIX (obsolete), HP-UX (old ones),



Thread control block — data structure



Clicker

With a multi-threaded process, a context switch between threads is performed by the OS kernel.

with

(A) Only ~~will~~ user-level threads

(B) Only with kernel-level threads

(C) Always

(D) Never

Common operations

- Common thread operations
    - Create
    - Exit
    - Join (instead of wait() for process)

Thread Libraries

Thread libraries provide users with ways to create threads in their own programs

- C/C++: Pthreads (implemented by the kernel)

lib entirely in user space

kernel level library (OS)

- C/C++: OpenMP (layer above Pthreads)

- Java: Java threads (implemented by the JVM, which relies on the kernel threads implementation)

- Python: threading / multiprocessing packages

- JavaScript: no multithreading (the multiprocessing is performed by the "web application" — Check working draft of W3C standard)

Thread Creation

```
#include <pthread.h>

int
pthread_create(      pthread_t*        thread,
               const pthread_attr_t*   attr,
                     void*             (*start_routine)(void*),
                     void*             arg);
```

thread: Reference (or pointer) to the ID of the thread

attr: used to specify any attributes this thread might have. E.g. Stack size, Scheduling priority….

start_routine: The name of the function that the thread starts to execute.

If the function's return type is void *, then its name is simply written; otherwise, it has to be type-cast to void *.

arg: the arguments to be passed to function (start routine).

To pass multiple arguments, send a pointer to a structure.

void pointer allow us to pass in any type of argument

Returns 0 on success, or a positive error number on error

Example

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

void *print_msg( void *ptr )
{
printf("Hello from thread %d\n", *( int *)ptr);
}

int main()
{
        pthread_t tr1, tr2;
        int  first, second, arg1, arg2;

    // Create two threads each of which will execute function
    printf("main: Begin \n");
    arg1=1;
    first = pthread_create( &tr1, NULL, print_msg, (void*) &arg1);
    arg2=2;
    second = pthread_create( &tr2, NULL, print_msg, (void*) &arg2);
    printf("Thread 1 returns: %d\n",first);
    printf("Thread 2 returns: %d\n",second);
    printf("main: End \n");
    return 0;
}
```

```
main: Begin
Thread 1 returns: 0
Thread 2 returns: 0
main: End
```

Create a child thread at print_msg()and pass arg1 as parameter

N C