Threads and Concurrency

CSD2180 Operating Systems
BSc in Computer Science (IMGD / RTIS)
Singapore Institute of Technology / DigiPen Institute of Technology
September 2021

Attendance Taking

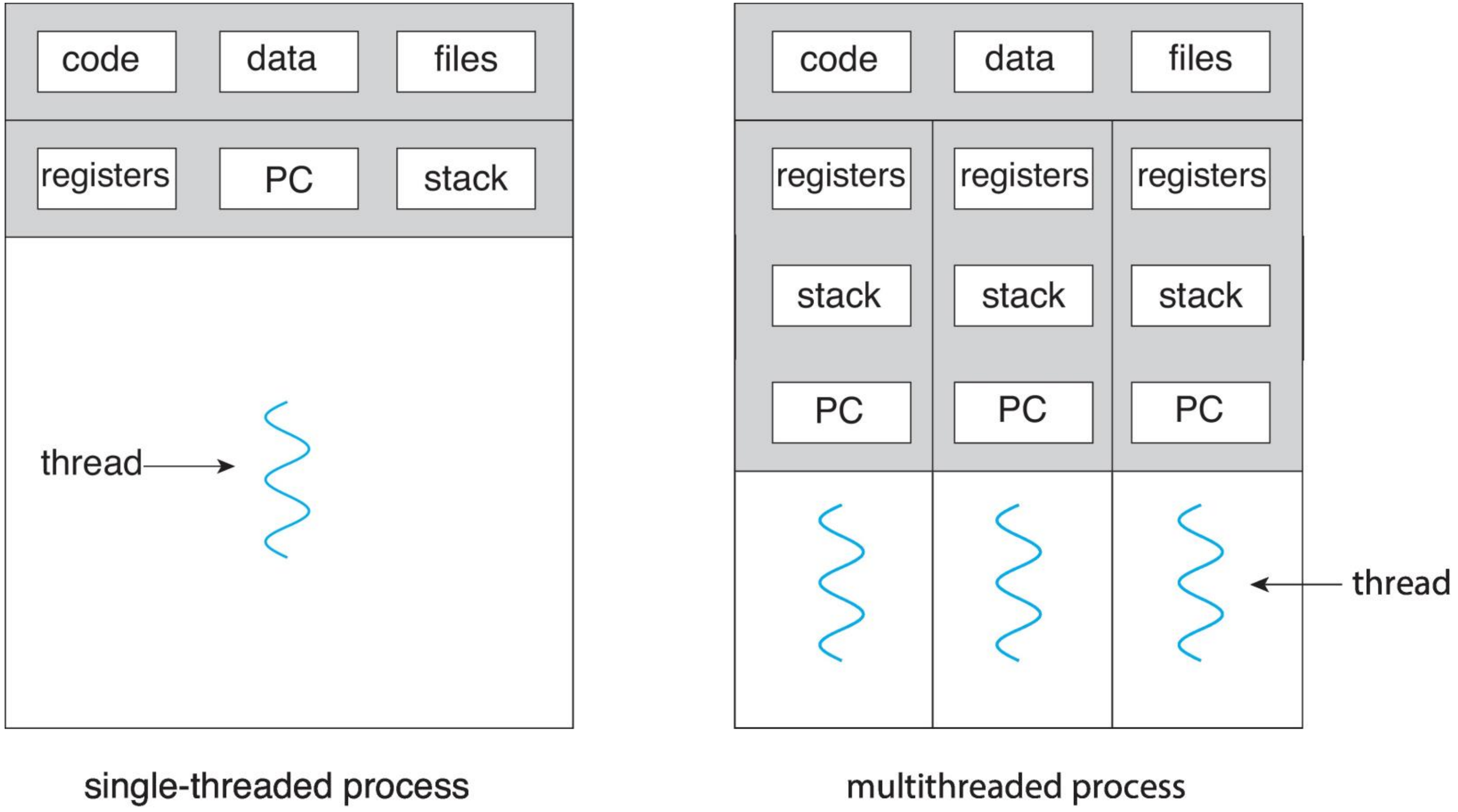
https://forms.office.com/r/aXuxCWp4FY

- Log in to your SIT account to submit the form
- You can only submit the form once
- The codeword is **coffee**

Overview

Of Single and Multithreaded Processes

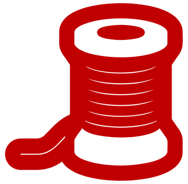
- A process has at least a single thread of control
- If a process has multiple threads of control, it can perform more than one task at a time



A thread is a basic unit of CPU utilization 🧵

Comprises a thread ID, a program counter (PC), a register set, and a stack

Shares its code and data section, and other operating system resources with other threads from the same process



A process can have one or more threads; a process without a thread can do nothing

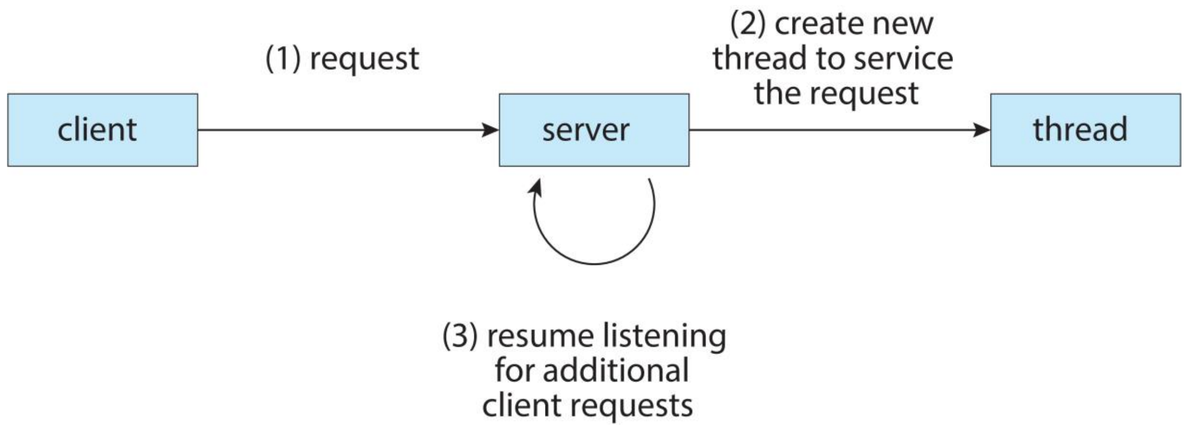
Process vs Thread

Process	Thread
Heavyweight operations	Lighter weight operations
Has its own memory space	Uses the memory space of the process they belong to
Context switching between processes is more expensive	Context switching between threads of the same process is less expensive
In general, processes do not share memory with other processes	Threads share memory with other threads of the same process
As each process have a different memory space, inter-process communication is slow(er)	Because threads of the same process share the same memory space with the process they belong to, inter-thread communication can be fast(er)

Multithreading

Most modern applications are multithreaded

An application typically is implemented as a process with several threads of control



Motivation for Multithreading

Multiple tasks with the application can be implemented by separate threads, e.g., update display, fetch data, spell checking, answer a network request

Process creation is heavy-weight while thread creation is light-weight

Can simplify code, increase efficiency

Benefits of Multithreading

Responsiveness

May allow continued execution if part of process is blocked, especially important for user interfaces

Economy

Cheaper than process creation, thread switching lower overhead than context switching

Resource sharing

Threads share resources of process, easier than shared memory or message passing

Scalability

Process can take advantage of multicore architectures

Threading and Multicore Programming

Multicore Programming Challenges

- **Dividing activities**, i.e., examining applications to find areas that can be divided into separate, concurrent tasks; ideally, tasks are independent of one another and thus can run in parallel on individual cores
- **Balance**, i.e., ensure that the tasks perform work of equal value; using a separate core to run tasks of little value may not be worth the cost

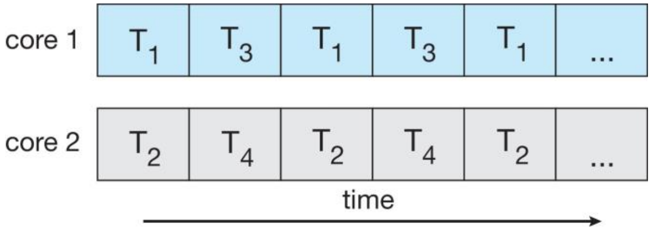
- **Data splitting**, i.e., data accessed and manipulated by the tasks must be divided to run on separate cores
- **Data dependency**, i.e., data accessed by the tasks must be examined for dependencies between two or more tasks; when one task depends on data from another, must ensure that the execution of the tasks are synchronized to accommodate the dependency

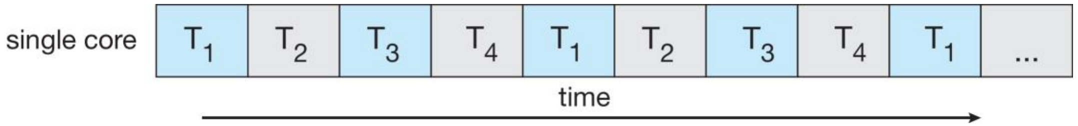
- **Testing and debugging**, i.e., as many different execution paths are possible, testing and debugging such concurrent programs is inherently more difficult

Concurrency and Parallelism Recap

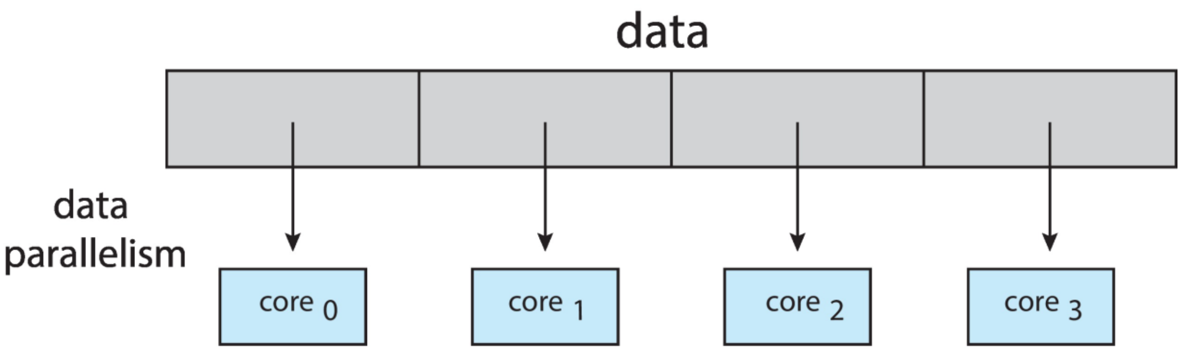
Parallelism implies a system can perform more than one task simultaneously

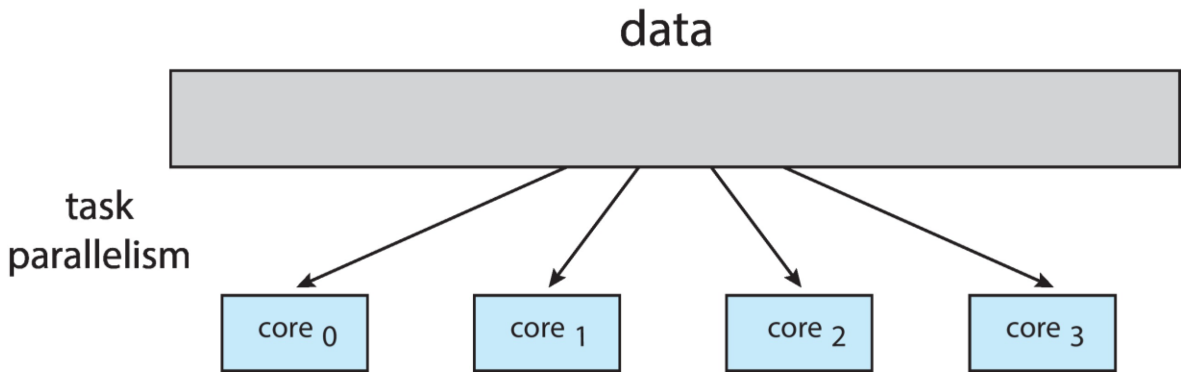
Concurrency supports more than one task making progress; e.g., in a single processor / core system, scheduler provides concurrency





Types of Parallelism





Data parallelism, i.e., distributes subsets of the same data across multiple cores, same operation on each

Task parallelism, i.e., distributing threads across cores, each thread performing unique operation

Multithreading Models

We have so far treated threads in a generic sense

However not all threads are the same...

User Threads and Kernel Threads

Kernel threads

A schedulable entity

Managed directly by the operating system; the operating system is aware of each kernel thread

Slightly faster to context switch between kernel threads than between processes

Application does not have control over how threads are managed; the operating system is responsible for scheduling them

Supported by virtually all general-purpose operating systems, e.g., Windows, Linux, etc.

User threads

Supported above the kernel

Managed without kernel support; the operating system does not know about such threads

Even faster context switch between user threads since no extra system calls are required

May impact performance (e.g., when performing I/O, support for CPU parallelism, etc.) depending on how threads are mapped

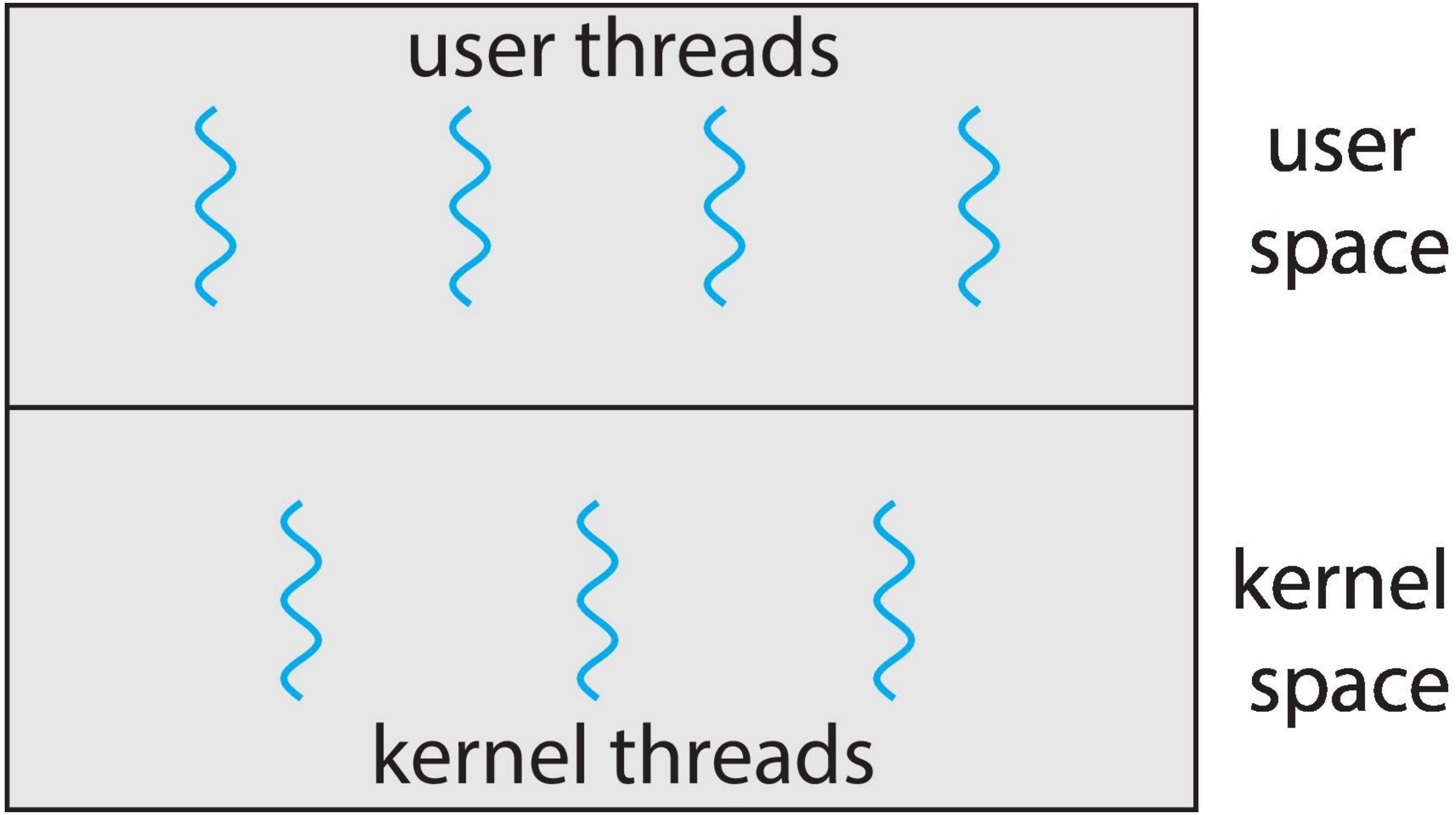
Typically managed using a thread library, e.g., POSIX pthreads, Windows threads, Java threads

Multithreading Models

Ultimately, a relationship must exist between user threads and kernel threads

Three common models

- Many-to-one model
- One-to-one model
- Many-to-many model



Many-to-One Model (N:1)

Many user-level threads mapped to single kernel thread

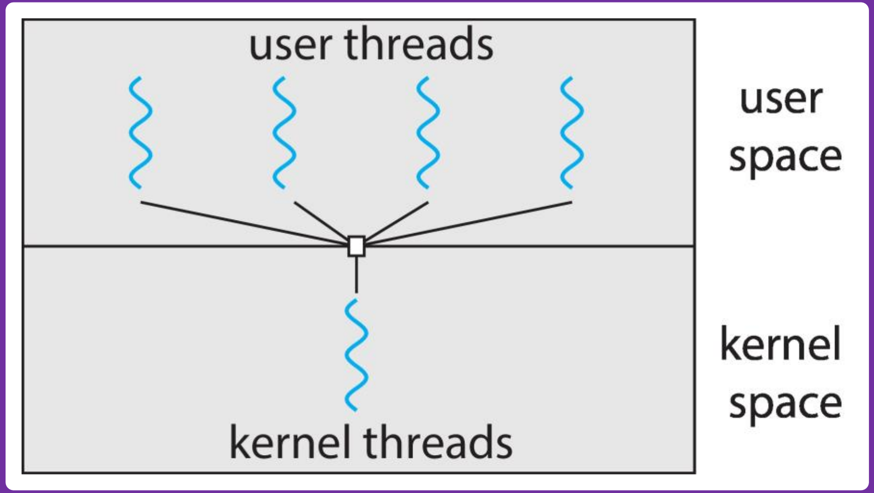
Thread management is done by thread library in user space

Entire process will block if a thread makes a blocking system call

As only **one thread can access the kernel at any one time**, multiple threads may not run in parallel on multicore system

- Hence, very few systems uses this model

Examples: Solaris Green Threads, GNU Portable Threads



One-to-One Model (1:1)

Each user-level thread maps to kernel thread

Creating a user-level thread creates a kernel thread

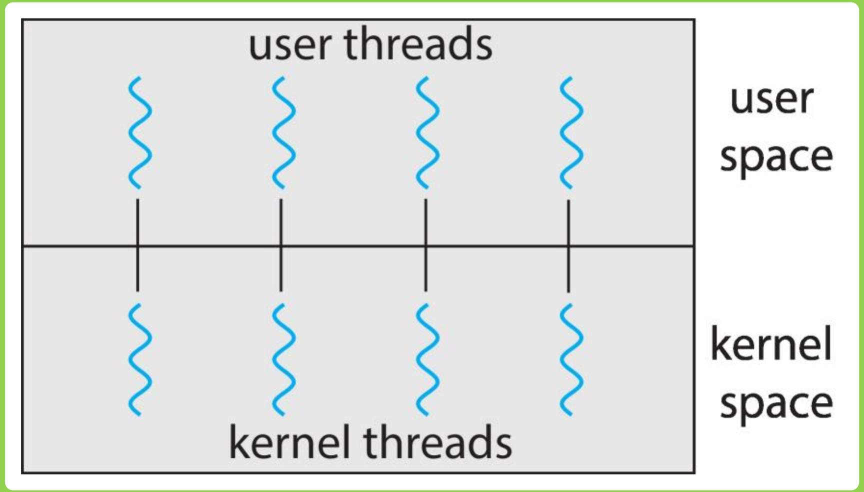
More concurrency than many-to-one

- Other threads can still run when a thread makes a blocking system call
- Multiple threads can run in parallel on multiprocessors systems

Number of **threads per process sometimes restricted** due to overhead

- Creating many kernel threads may burden the performance of a system

Examples: Windows, Linux, most operating systems



Many-to-Many Model (M:N)

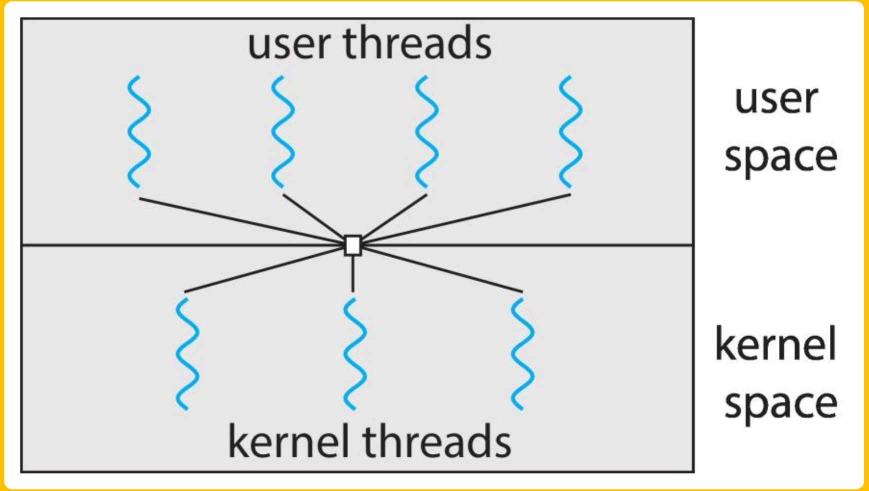
Allows many user level threads to be mapped to many kernel threads

Allows the operating system to create sufficient kernel threads

Number of kernel threads may be specific to a particular application or a particular machine

Not very common nowadays

Example: Windows with the *ThreadFiber* package, FreeBSD 5 / 6



Effects of Multithreading Models on Concurrency

Many-to-one model

- Developers can create as many user threads as they wish, but cannot achieve parallelism because only one kernel thread can be scheduled at a time

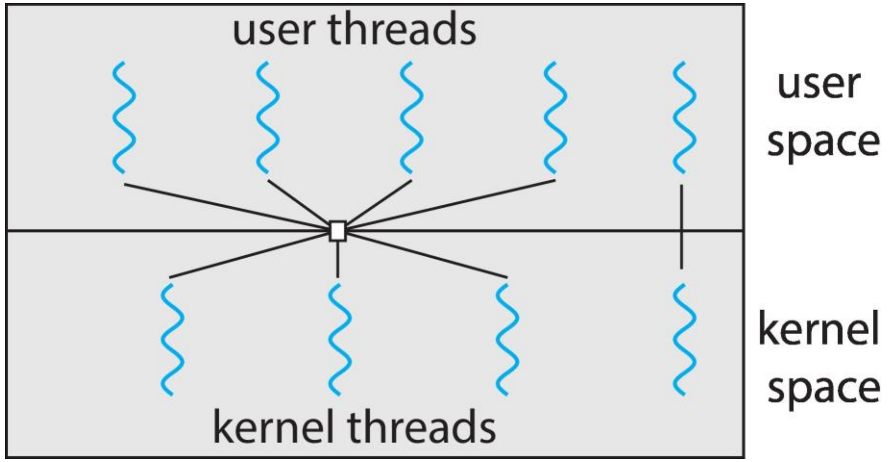
One-to-one model

- Allows greater concurrency, but developers must be careful not to create too many threads

Many-to-many model

- Suffers from neither the above shortcomings
- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel
- Difficult to implement

Two-level model: A variation of the many-to-many model multiplexes many user threads to a smaller or equal number of kernel threads, and allows a user thread to be bound to a kernel thread



Thread Libraries

Thread Libraries

A thread library provide programmers with API for creating and managing threads

Examples of thread libraries in use today

- **POSIX pthreads:** Threads extension of the POSIX standard; may be provided as a user or kernel library
- **Windows thread library:** A kernel-level library on Windows systems
- **Java thread API:** Allows threads to be created and managed directly in Java programs; generally implemented using a thread library available on the host system

Two primary ways of implementing library

- Library entirely in user space with no kernel support
  - All code and data structures for the library exist in user space
  - Invoking a function in the library results in a local function call in user space and not a system call
- Kernel-level library supported by the operating system
  - Code and data structures for the library exist in kernel space
  - Invoking a function in the API for the library typically results in a system call to the kernel

POSIX pthreads

A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

Is a **specification**, not implementation

API specifies behavior of the thread library, implementation is up to developer of the library

Common in UNIX operating systems (Linux, macOS)

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

int sum;                    /* this data is shared by the thread(s) */
void *runner(void *param);  /* threads call this function */

int main(int argc, char *argv[]) {
    pthread_t tid;          /* the thread identifier */
    pthread_attr_t attr;    /* set of thread attributes */

    /* set the default attributes of the thread */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n", sum);
}

/* The thread will execute in this function */
void *runner(void *param) {
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Threading Issues

A Summary of Threading-Related Issues

Semantics of `fork()` and `exec()` system calls

Signal handling (synchronous and asynchronous)

Thread cancellation of target thread (asynchronous or deferred)

Thread-local storage

Scheduler activations

Semantics of `fork()` and `exec()`

If a thread in a program calls `fork()`, does `fork()` duplicate only the calling thread, or all threads?

- Some Unix systems have two versions of fork
  - One duplicates all threads, and another that duplicates only the thread invoking the `fork()` system call

`exec()` usually works as normal, i.e., replace the running process including all threads

Which of the two versions of `fork()` to use?

If `exec()` is called immediately after forking, then duplicating all threads is not productive as the program specified in the parameters to `exec()` will replace the process; in this case, duplicating only the calling thread is appropriate

If the forked process does not call `exec()` after forking, the separate process should duplicate all threads

Questions?

Thank You!

[Weihan.Goh {at} Singaporetech.edu.sg](mailto:Weihan.Goh@Singaporetech.edu.sg)

https://www.singaporetech.edu.sg/directory/faculty/weihan-goh

https://sg.linkedin.com/in/weihan-goh