Operating Systems
CS 415

Lecture 7:

Threads

Suyash Gupta

Assistant Professor

Distopia Labs and ONRG

Dept. of Computer Science

(E) suyash@uoregon.edu

(W) gupta-suyash.github.io

UNIVERSITY OF OREGON

Announcements

- **Suyash Gupta**
  - Office Hours: Thursday, 10-11am, Deschutes 334

- **Nihal Balivada (TA)**
  - Office Hours: Wednesday/ Friday, 11-12pm, Deschutes 335

- **Ranjitha Rani (TA)**
  - Office Hours: Monday /Tuesday, 1-2pm, Deschutes 229

Assignment 1 Due Today!

- **Deadline → April 21, 2026 at 11:59pm PST**
- Please start working and talk to us if you are stuck.

- **Midterm → April 28, 2026 (in-class)**
  - Closed book, no cheat sheets, no discussions.

Last Class

- Process
- Chapter 3 is now complete!

Process and Threads

Process and Threads

- Each process starts with a single thread of control.
- A process can contain multiple threads of control.

Threads

- **Each thread owns?**

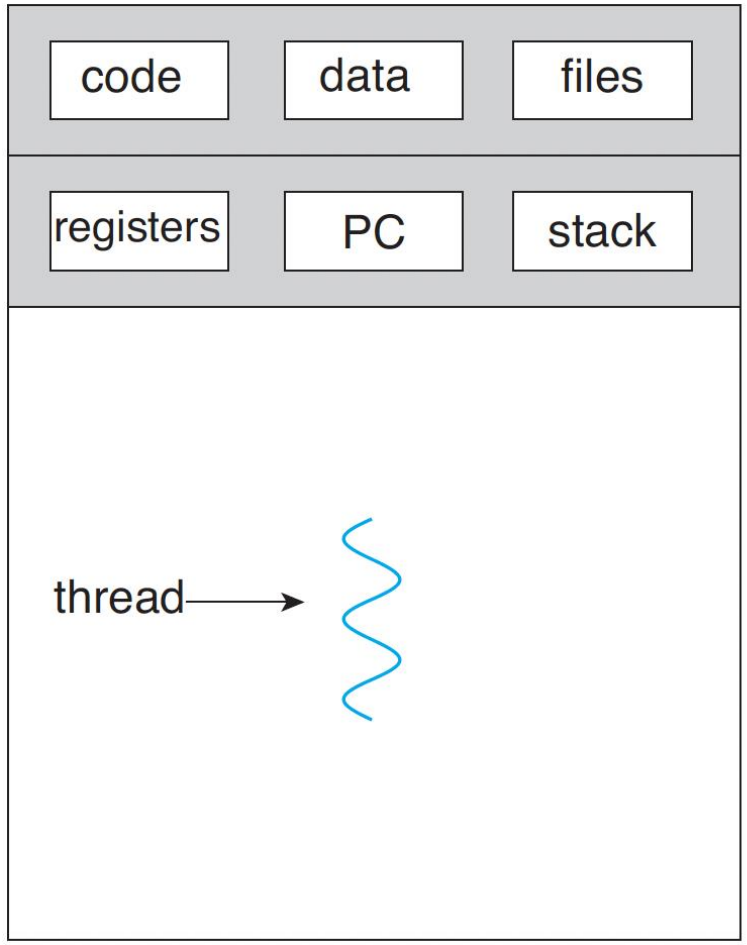
- **Threads share?**

Threads

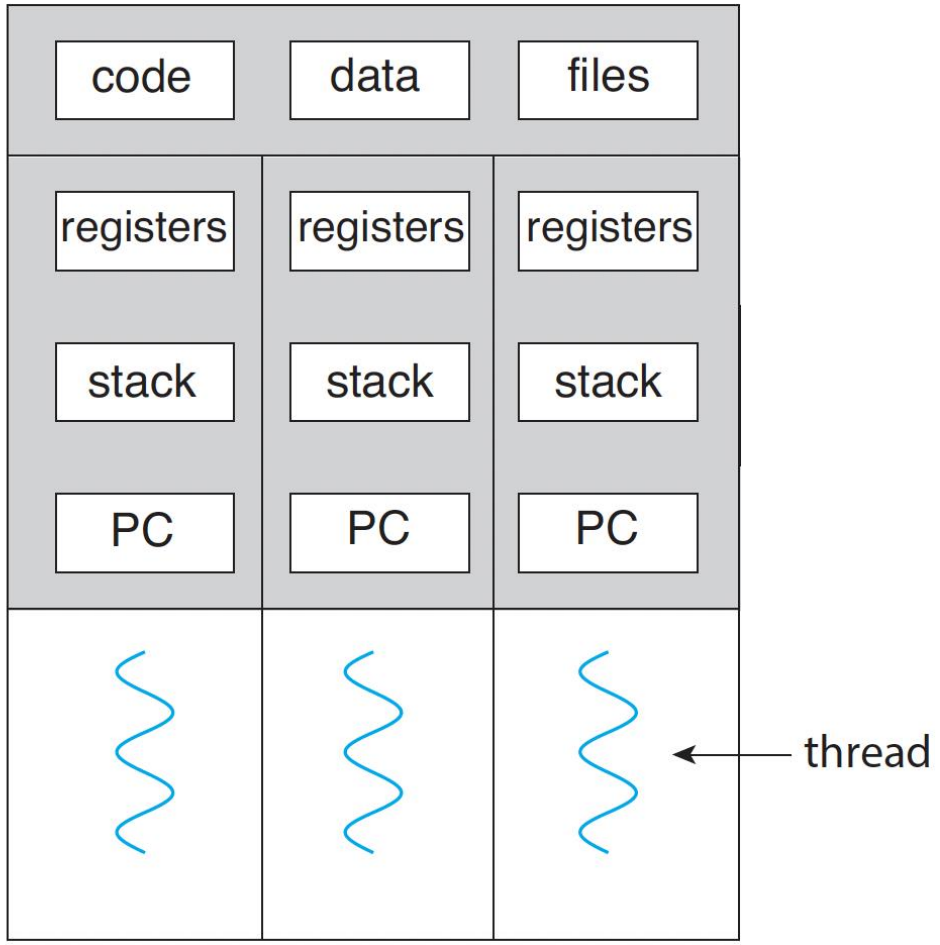
- **Each thread owns:**
  - Registers, stack, Program Counter, Thread ID

- **Threads share:**
  - Code and data.
  - Other operating-system resources, such as open files and signals.

Threads



single-threaded process



multithreaded process

Examples of Multithreaded Applications

Examples of Multithreaded Applications

- **Web browser:**
  - One thread to display images or text.
  - One thread to retrieve data from the network.

- **Word processor:**
  - One thread for displaying graphics.
  - One thread for responding to keystrokes from the user.
  - One thread to perform spelling and grammar checking in the background.

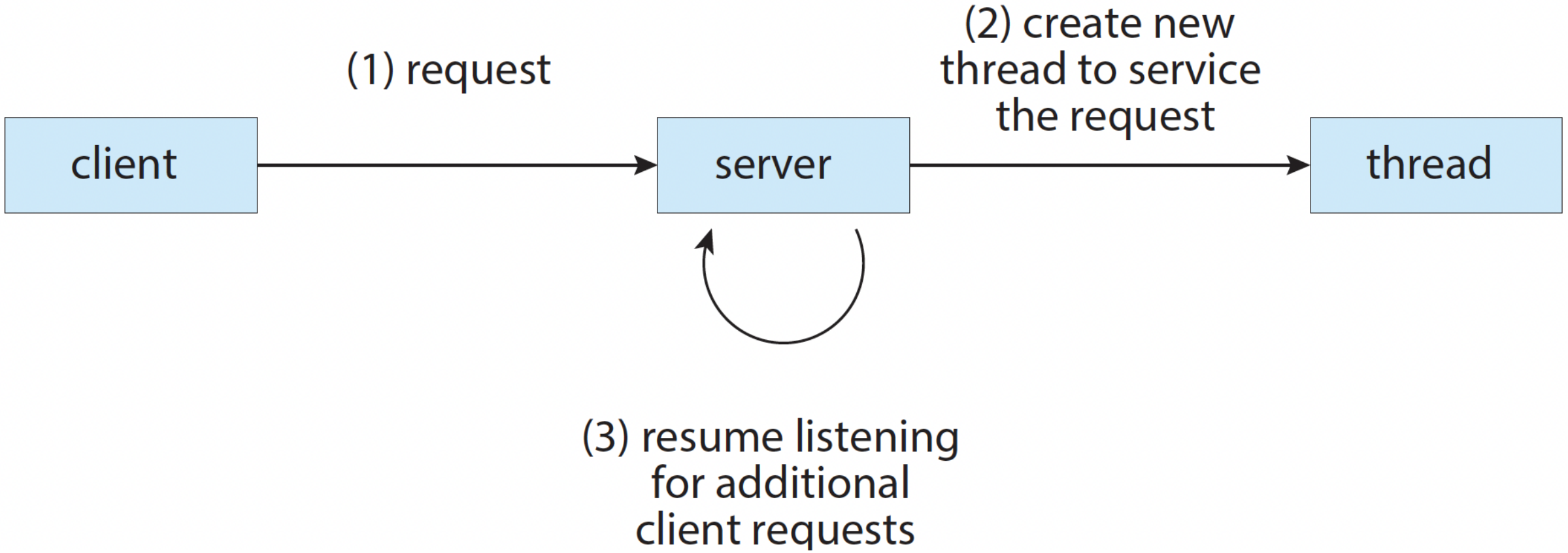
Multithreading

- Multiple threads share process resources, including memory!

- Why would you want to do that?
  - Concurrency!

- How do you get concurrency?
  - One way is to create multiple processes
  - Use IPC to support process-level concurrency.
  - But IPC is unnecessary and costly if you only want to finish small concurrency tasks in current applications.

Multithreading



Which is cheaper?

- Create new process or create new thread (in existing process)?
- Context switch between processes or threads?
- Interprocess or interthread communication?
- Sharing memory between processes or threads?
- Terminating a process or terminating a thread (not the last one)?

Threads vs. Process

- Easier to create than a new process.
- Less time to terminate a thread than a process.
- Less time to switch between two threads.
  - Within the same process
- Less communication overheads
  - Communicating between the threads of one process is simple because the threads share everything.
  - Address space is shared, and thus memory is shared.

Threads vs. Process

Process creation method	Time (sec), elapsed (real)
fork()	22.27 (7.99)
vfork() (faster fork)	3.52 (2.49)
clone()	2.97 (2.14)

Time to create 100,000 processes (Linux 2.6 kernel, x86-32 system).

clone() creates a lightweight Linux process (thread).

Threads vs. Process

- Consider a web server on a Linux platform.
  - Cost of each fork() operation → 0.22 ms
  - Maximum of (1000 / 0.22) = 4545.5 connections/sec
  - 0.45 billion connections per day per machine
- Fine for most servers, but too slow for super-high-traffic web services.

- Facebook serves O(750 billion) page views per day.
  - Guess ~1-20 HTTP connections per page
  - Would need 3,000 - 60,000 machines just to handle fork(), without doing any work for each connection!

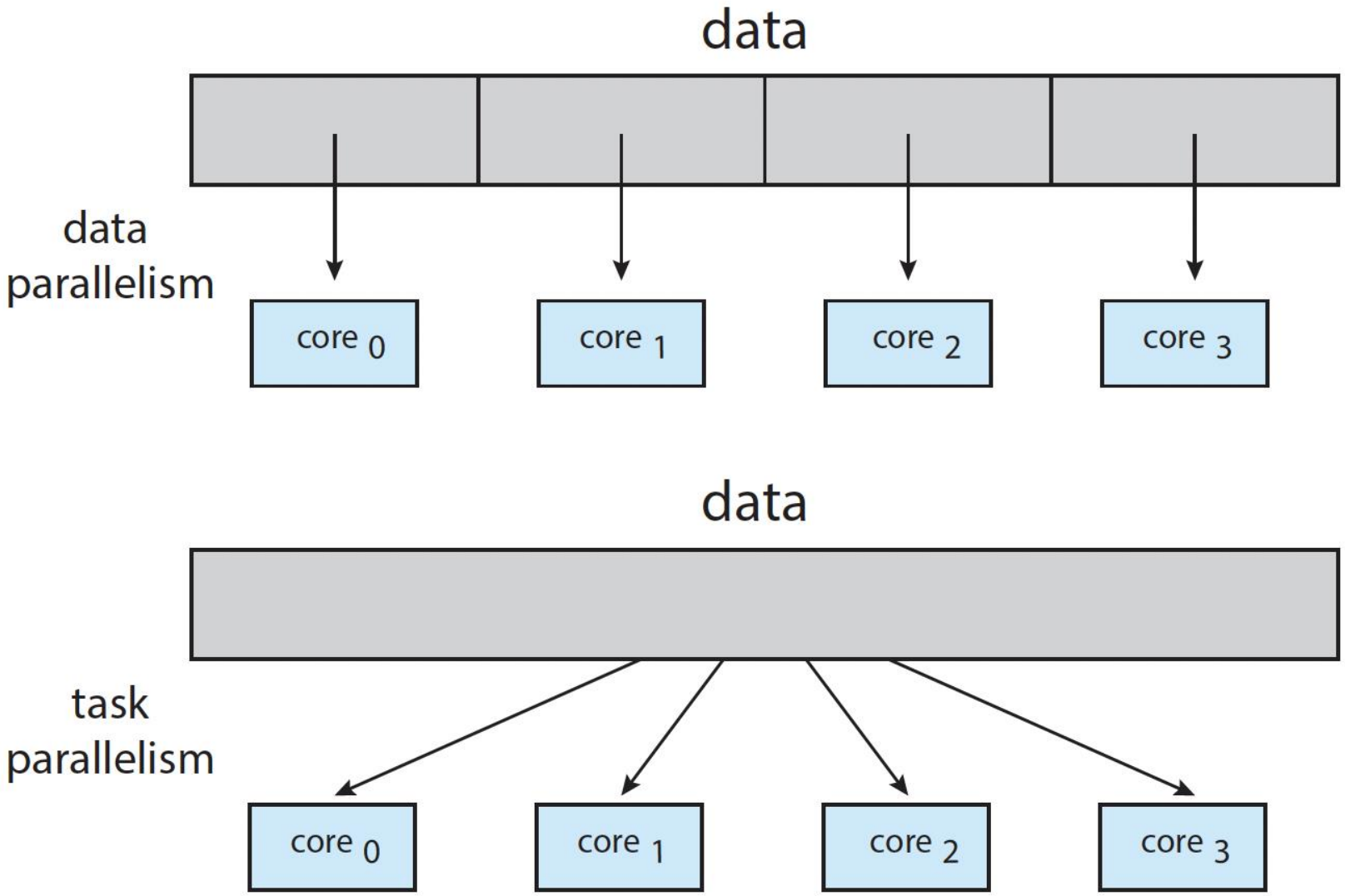
Concurrency vs. Parallelism

Concurrency vs. Parallelism

- A concurrent system allows more than one tasks to make progress.
- A parallel system can perform more than one task simultaneously.
- Thus, it is possible to have concurrency without parallelism.
- Before the advent of multiprocessor and multicore architectures:
  - CPU schedulers provided the illusion of parallelism by rapidly switching between processes,
  - Such processes were running concurrently, but not in parallel.

Types of Parallelism

Types of Parallelism



Types of Parallelism

- **Data parallelism**
  - Distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

- **Task parallelism**
  - Distributing not data but tasks (threads) across multiple computing cores.
  - Each thread is performing a unique operation.
  - Different threads may be operating on the same data, or they may be operating on different data.

Types of Parallelism Examples

Types of Parallelism Examples

- **Data parallelism Example:**
  - Summing the contents of an array of size N.
  - On a single-core system, one thread would simply sum the elements [0] . . . [N − 1].
  - On a dual-core system:
    - thread A, running on core 0, could sum the elements [0] . . . [N/2 − 1]
    - thread B, running on core 1, could sum the elements [N/2] . . . [N − 1].

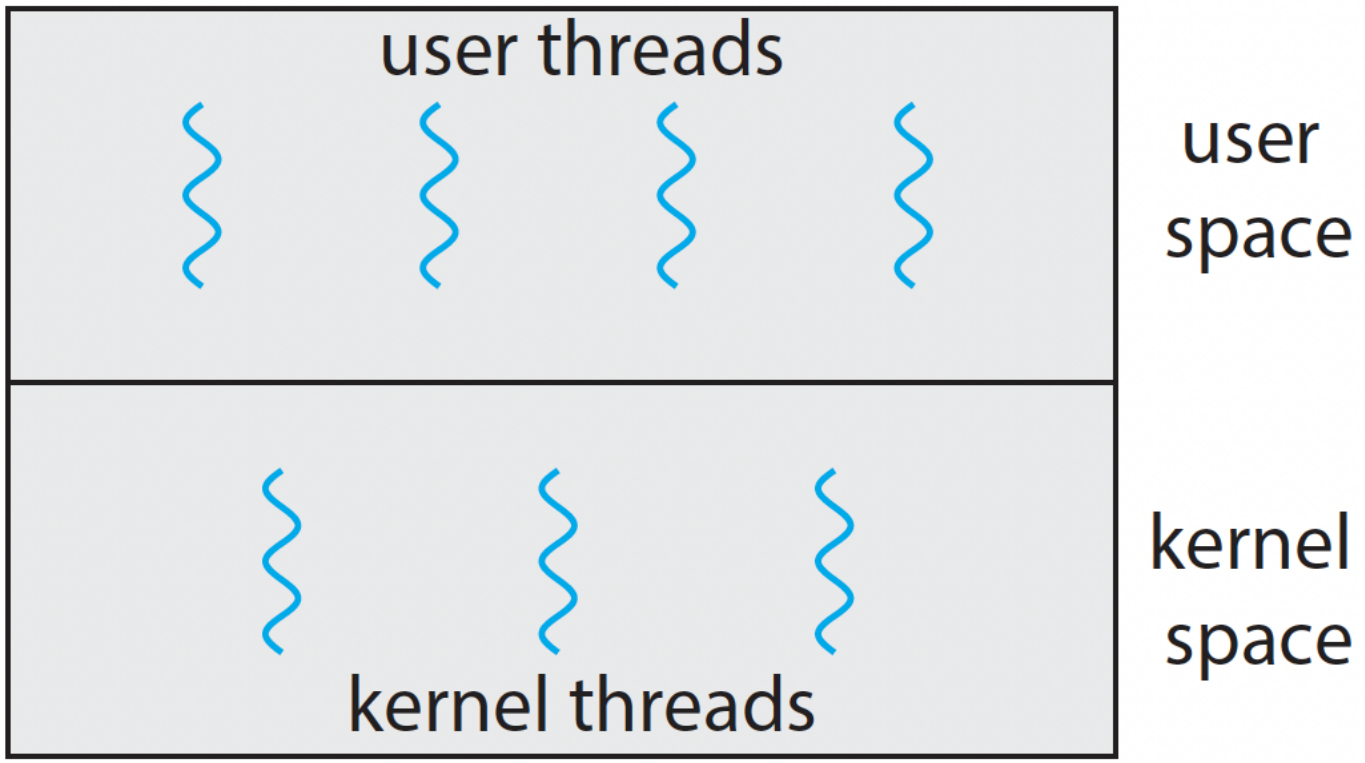
- **Task parallelism Example**
  - Two threads, each performing a unique statistical operation on the array of elements.

Kernel Threads

Kernel Threads

- OS also employs multi-threading.
- Kernel threads are background workers → threads that work in kernel mode.
- They help OS finish work that doesn't have to run right now but still must run in the kernel.
- Tasks include:
  - Flushing dirty pages to disk, write-back daemons, journaling.
  - Memory management tasks like page reclamation, swap daemons.
  - Housekeeping: cleaning up zombies, reaping resources, load-balancing, etc.

Kernel Threads



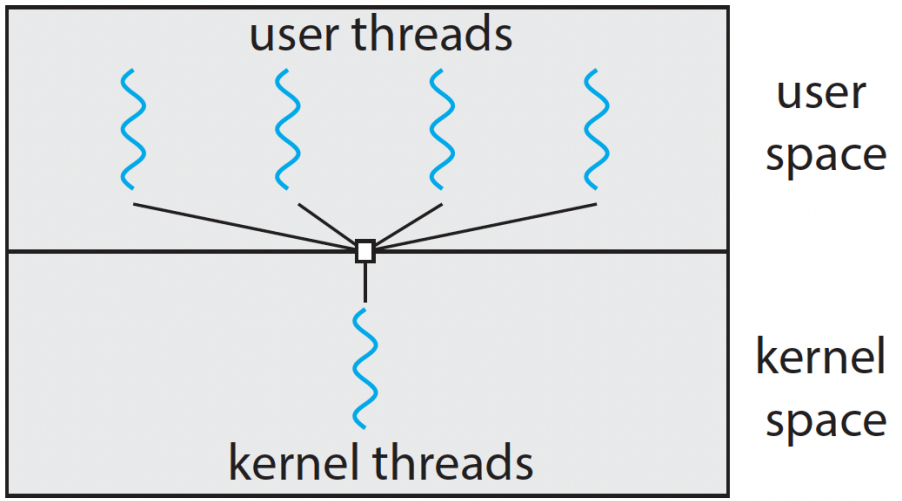
Relationship Models between User Threads and Kernel Threads

Relationship Models between User Threads and Kernel Threads

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

Many-to-One

- Maps many user-level threads to one kernel thread.
- User thread management is done by the thread library in user space.
- The entire process blocks if a user thread makes a blocking system call.
- Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

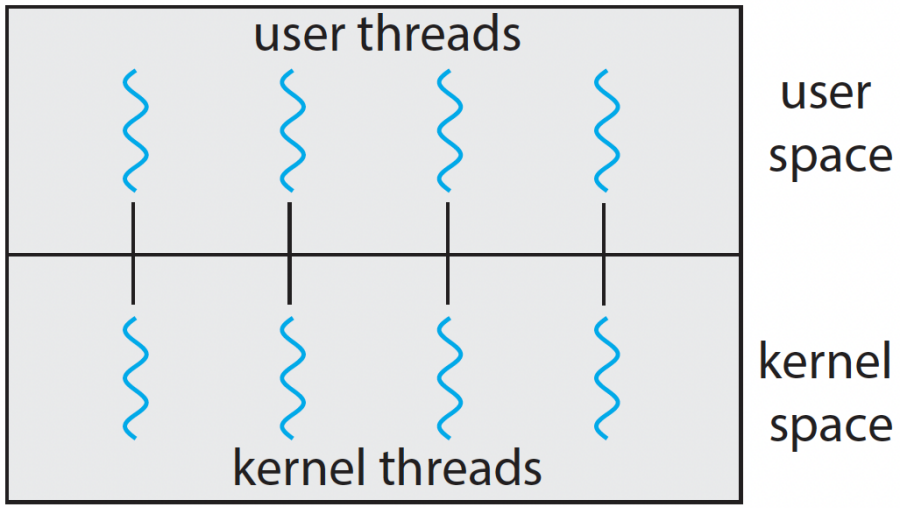


One-to-One

- Maps each user thread to a kernel thread.

- Provides more concurrency by allowing another thread to run when a thread makes a blocking system call.
  - It also allows multiple threads to run in parallel on multiprocessors.

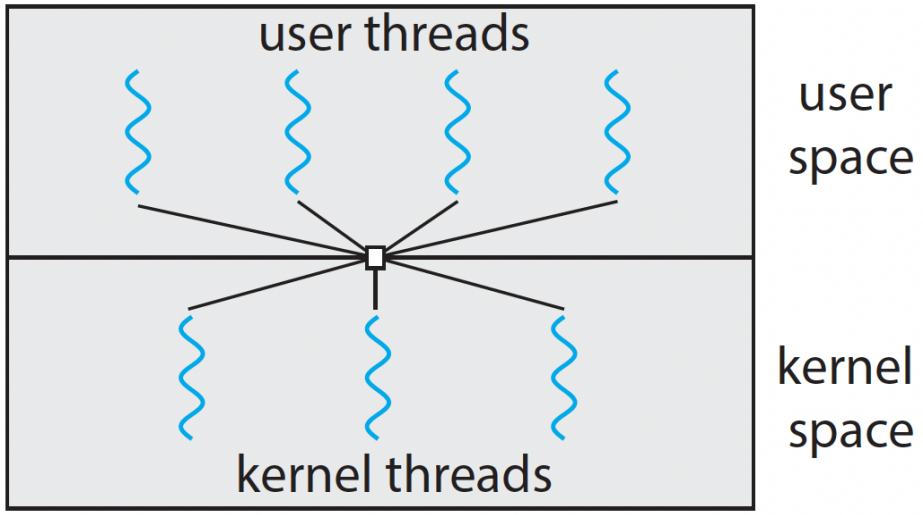
- Creating a user thread requires creating the corresponding kernel thread.
  - Large number of kernel threads may burden the performance of the system.



Many-to-Many

- Multiplexes many user-level threads to a smaller or equal number of kernel threads.
  - The number of kernel threads may be specific to a particular application or machine.

- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.



Thread Libraries

Thread Libraries

- Provides the programmer with an API for creating and managing threads.
- Two ways of implementing a thread library.

- **User Space 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 results in a system call to the kernel.

Thread Creation

Thread Creation

- **Asynchronous threading:**
  - Once the parent creates a child thread, the parent resumes its execution.
  - The parent and child execute concurrently and independently.
  - Little data sharing between them.
  - E.g., For designing responsive user interfaces.

- **Synchronous threading:**
  - The parent thread creates one or more children and then waits for all of its children to terminate before it resumes.
  - Children threads can perform work concurrently.
  - Significant data sharing among threads.
  - E.g., the parent thread combines the results calculated by its children.

Pthreads

Pthreads

- Posix Threads → POSIX standard (IEEE 1003.1c)
- For defining an API for thread creation and synchronization.
- This is a specification for thread behavior, not an implementation.
- OS designers may implement the specification in any way they wish.
- Pthreads implementations: UNIX-type systems, Linux and macOS.

Pthreads

```
#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);
}
```

Pthreads: Joining Multiple Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Unlimited Threads?

Unlimited Threads?

- Creating threads although cheaper than creating processes, has problems:
  - The amount of time required to create the thread + the thread will be discarded once it has completed its work.
  - Allowing each concurrent request to be serviced in a new thread, can lead to unlimited threads!
  - Can exhaust system resources, such as CPU time or memory.

Thread Pools

- **Thread pool →**
  - Create a number of threads at start-up and place them into a pool.
  - Threads in pool sit and wait for work.
  - When a server receives a request, it submits the request to the thread pool.
  - If there is a thread in the pool, it is awakened, and the request is serviced.
  - If the pool contains no thread, the task is queued until one becomes free.
  - Once a thread completes its service, it returns to the pool.

Threads Communication

Threads Communication

- Threads in the same process can communicate both by shared memory or by message passing built on top of that shared memory.

- **Shared memory**
  - Any variable can be seen by any thread → very fast, but easy to get wrong (races, deadlocks).
  - Use locks, mutexes, and atomic operations.
  - Shared counter with and without a mutex.

- **Message passing**
  - Threads don't touch each other's state, they send messages."
  - Usually built with a thread-safe queue or producer/consumer patterns.
  - Often simpler reasoning: fewer shared mutable structures.
  - Log thread: other threads push log entries into a queue; only the log thread writes to disk.