Operating System Concepts

Lecture 14: Threads

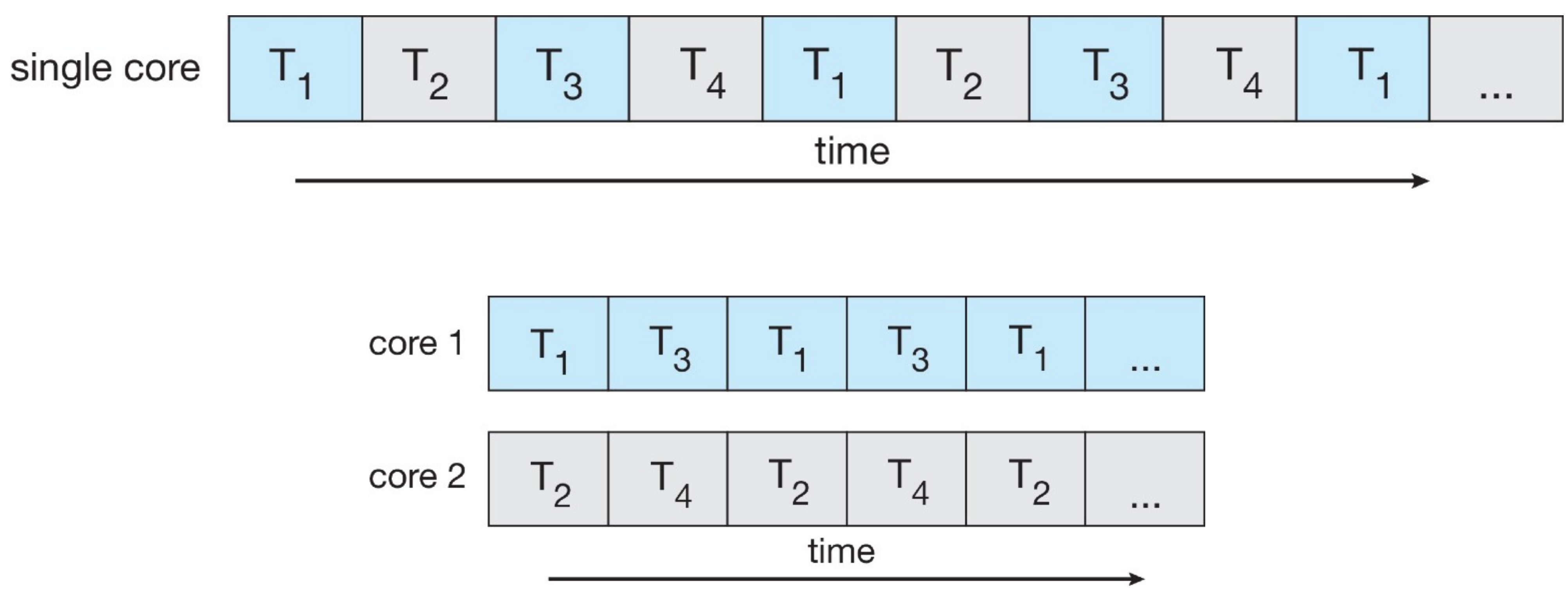
Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Today's class

- Multithreading
  - thread vs. process
  - user threads vs. kernel threads
- Threading issues

Motivation

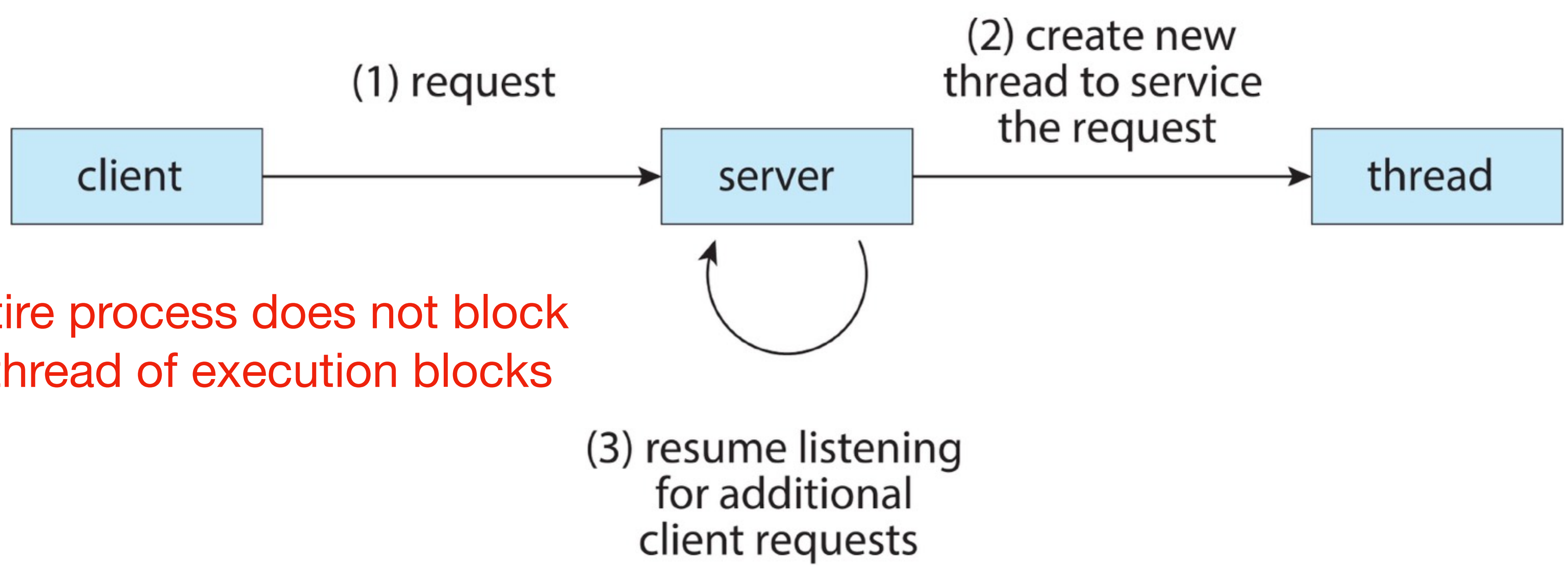
- Modern systems have multiple processing cores
  - can speed up the process through **parallelism**
- Many applications running on multicore systems are multi-threaded to achieve better performance
  - e.g. one thread is doing computation while another thread waits for I/O operation



Multithreading in the real world

A process with multiple threads of control can perform multiple tasks in parallel

- a web server accepting requests from hundreds of clients concurrently, using one thread per connection
- a web browser might have a thread to display text and images, a thread to receive data from network, and a thread to respond to user events (keystrokes, clicks, etc.)
- a kernel is also multithreaded, each performing a specific task, e.g., device management, memory management, interrupt handling, etc.
  - to display kernel threads on a linux system, run `ps -ef`
  - the kernel thread daemon `kthreadd` is the parent of all other kernel threads



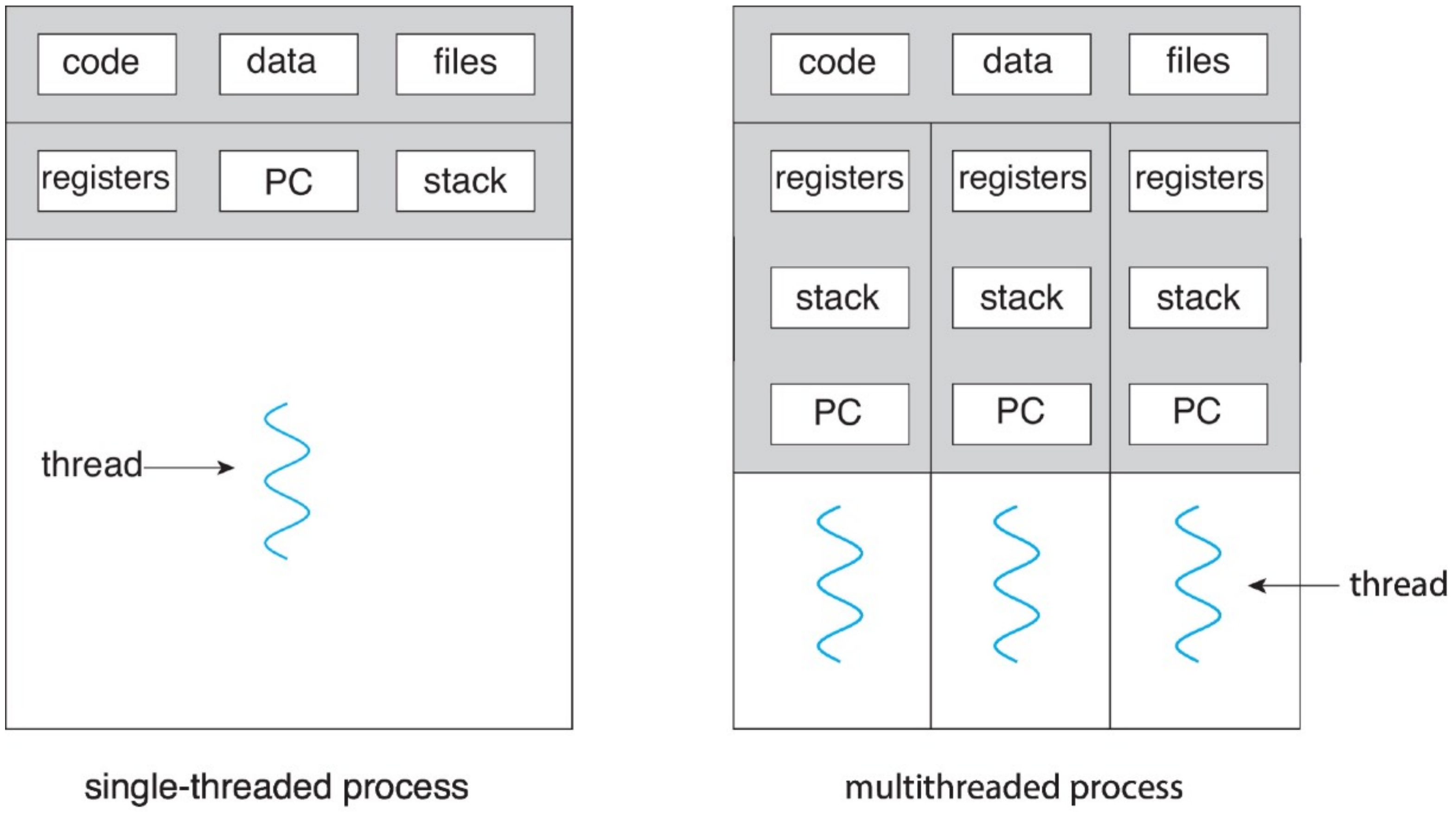
the entire process does not block if one thread of execution blocks

Thread is another abstraction

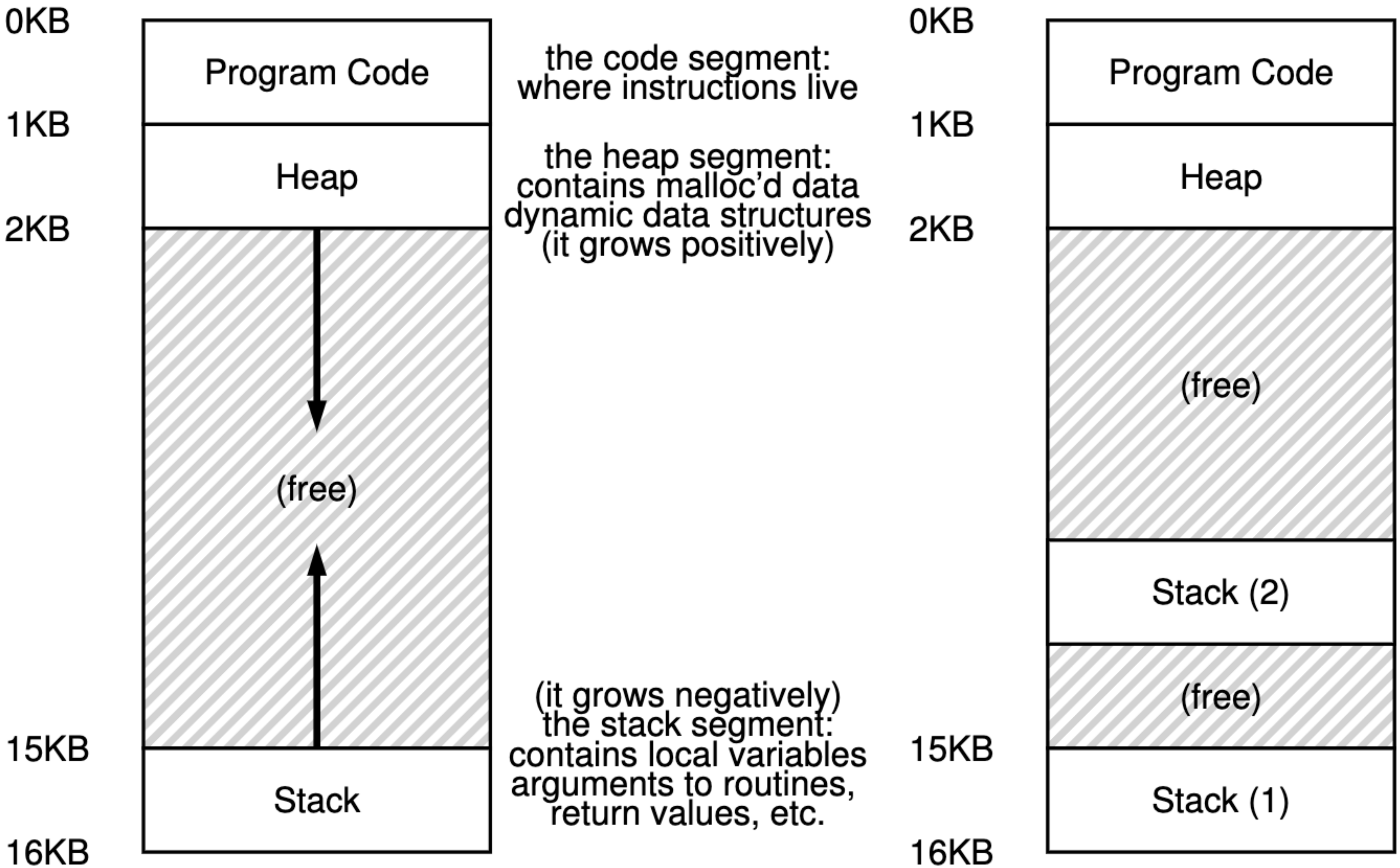
- A single execution stream within a process representing an independently schedulable task
  - process defines the protection domain
- Each thread has a thread ID, a program counter, a stack pointer, a register set which are kept in the **thread control block** (TCB)
  - TCB also contains scheduling info (priority) and a pointer to the PCB
  - a thread has its own stack
    - but it shares with other threads belonging to the same process its code section, data section, and other OS resources (i.e. open files and signals)
  - a thread can have **Thread-Local Storage** (TLS)
    - different from local variables in a function because TLS data are accessible across function invocations

Single and multithreaded processes

- Each process may have multiple threads of control (multiple points of execution)
  - the address space of the process is **shared** among its threads (many threads per protection domain)
  - a system call is not required for cooperation between threads, hence it is easier to share data between threads of a process than using message passing and shared-memory system calls to share data between processes
  - since threads of a process have their own PC and registers, a **context switch** is still required if they run on a single processor successively
    - but the address spare remains the same unlike the context switch that happens between processes



Layout of a single vs. multithreaded process



Benefits of multithreading

- Faster response to user
  - especially important in user interface design
- Resource sharing
  - threads run within the same address space and therefore share memory and other process resources by default while processes have to use IPC
- Economy
  - can save on the required memory by having multiple threads instead of multiple processes
  - it is less costly to create threads and context switch between them
    In Linux, switching between processes takes 3-4µs while switching between threads of a process takes 100ns
- Scalability
  - threads can run in parallel on different processing cores; this is key for multiprocessor architecture

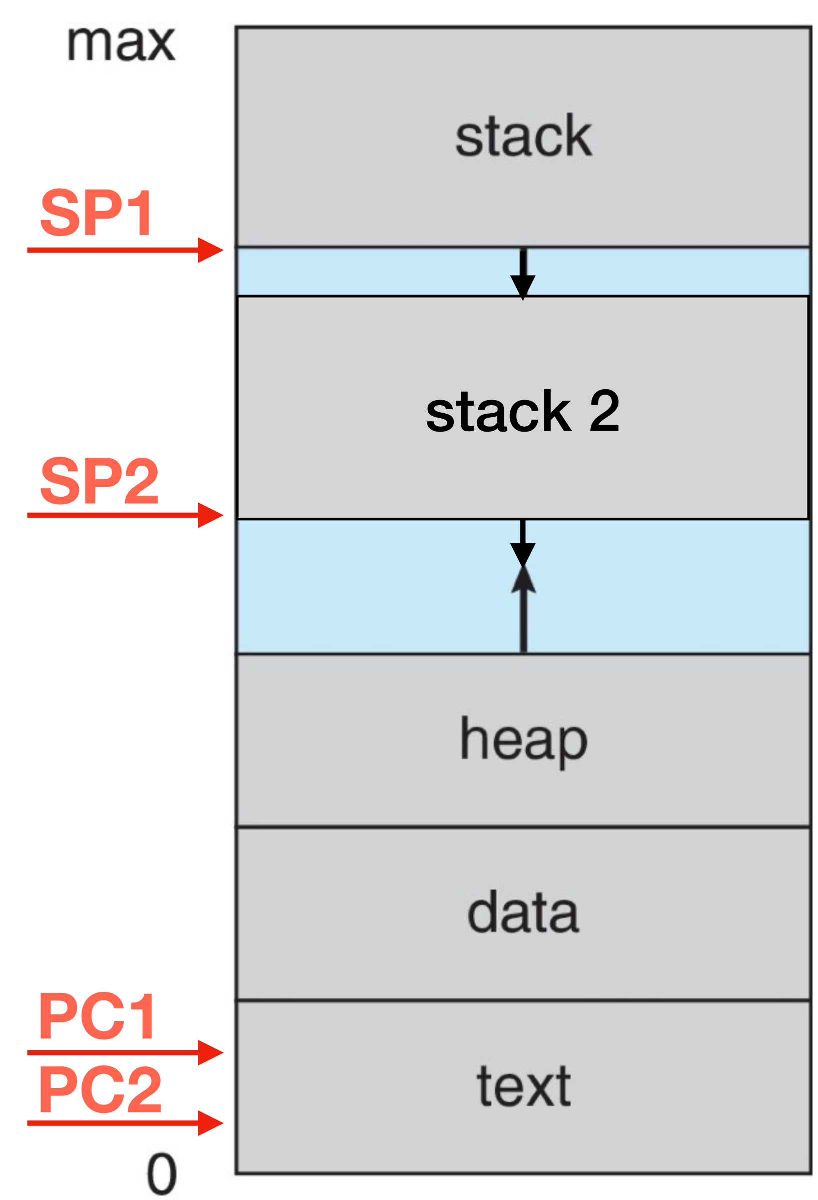
Concurrency vs. parallelism

- Concurrency: all tasks can make progress
  - can happen by switching between processes rapidly
  - does not need multithreading
- Parallelism: the system can perform more than one task at a time
  - multithreading is a way to improve parallelism
- It is possible to have concurrency but not parallelism

Example of a multithreaded program (loose syntax)

```
#define N 100;
int in, out;
int buffer[N];

void producer( ) {
    …
}
void consumer( ) {
    …
}
int main( ) {
    in = 0; out = 0;
    fork_thread(producer( ));   // e.g. pthread_create takes a function pointer
    fork_thread(consumer( ));  // e.g. pthread_create takes a function pointer
    …
}
```



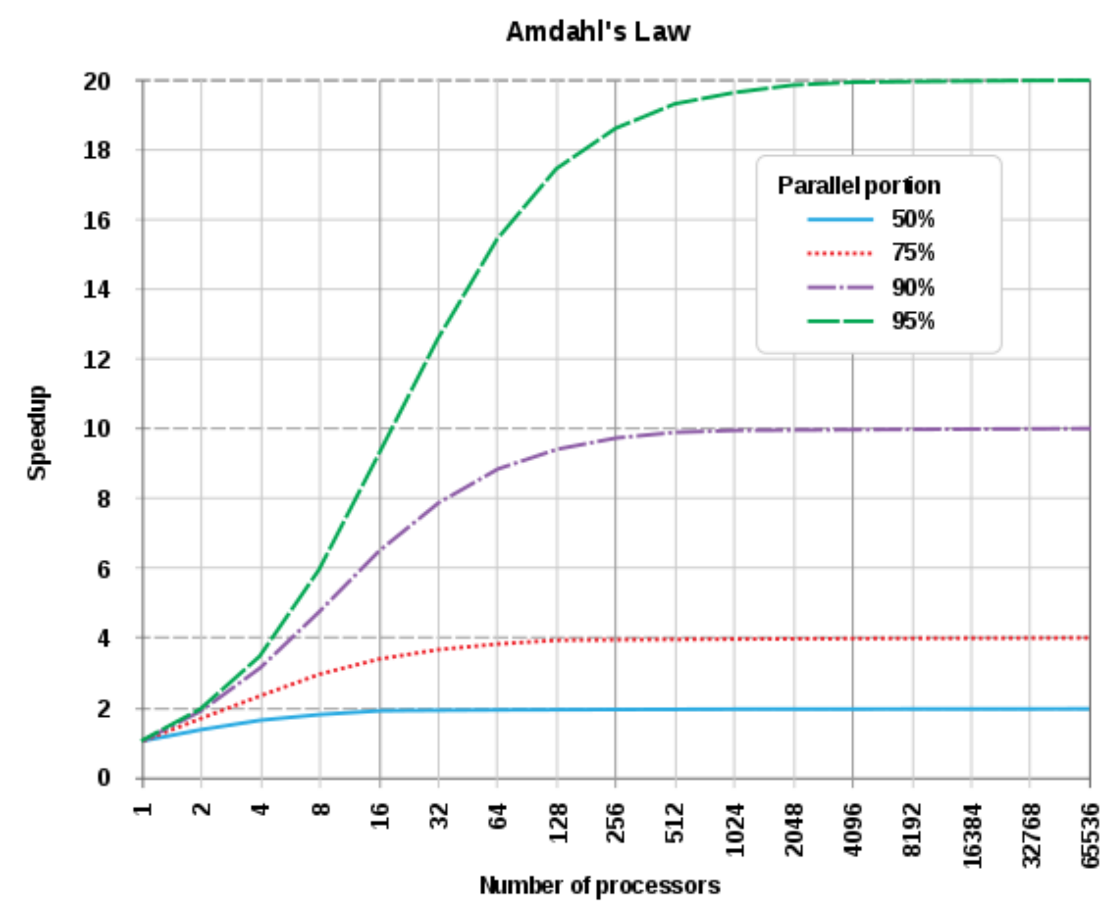
- We will have 3 threads after calling fork twice: the main thread and two newly created threads
  - they run in an arbitrary order (what if they all update a global variable?)
  - the main thread can execute concurrently with forked threads (**asynchronous threading**) or wait for all of them to terminate (**synchronous threading**)

Programming challenges in multicore systems

- Identifying and splitting tasks
  - tasks must be independent of one another and can run in parallel
  - tasks should perform equal amount of work (of equal value)
  - task parallelism concerns the distribution of tasks across multiple cores
- Data splitting
  - data required by separate tasks must be divided
  - data parallelism concerns the distribution of data across multiple cores
- Data dependency
  - task execution must be **synchronized** if there is a dependency between data they access
- Testing and debugging
  - multithreaded applications are inherently more difficult to develop and debug due to non-determinism and multiple execution paths
    - scheduler can run threads in any order and switch threads at any time
      e.g. incrementing a variable requires mov, add, mov operations and a context switch may happen in the middle of these operations!

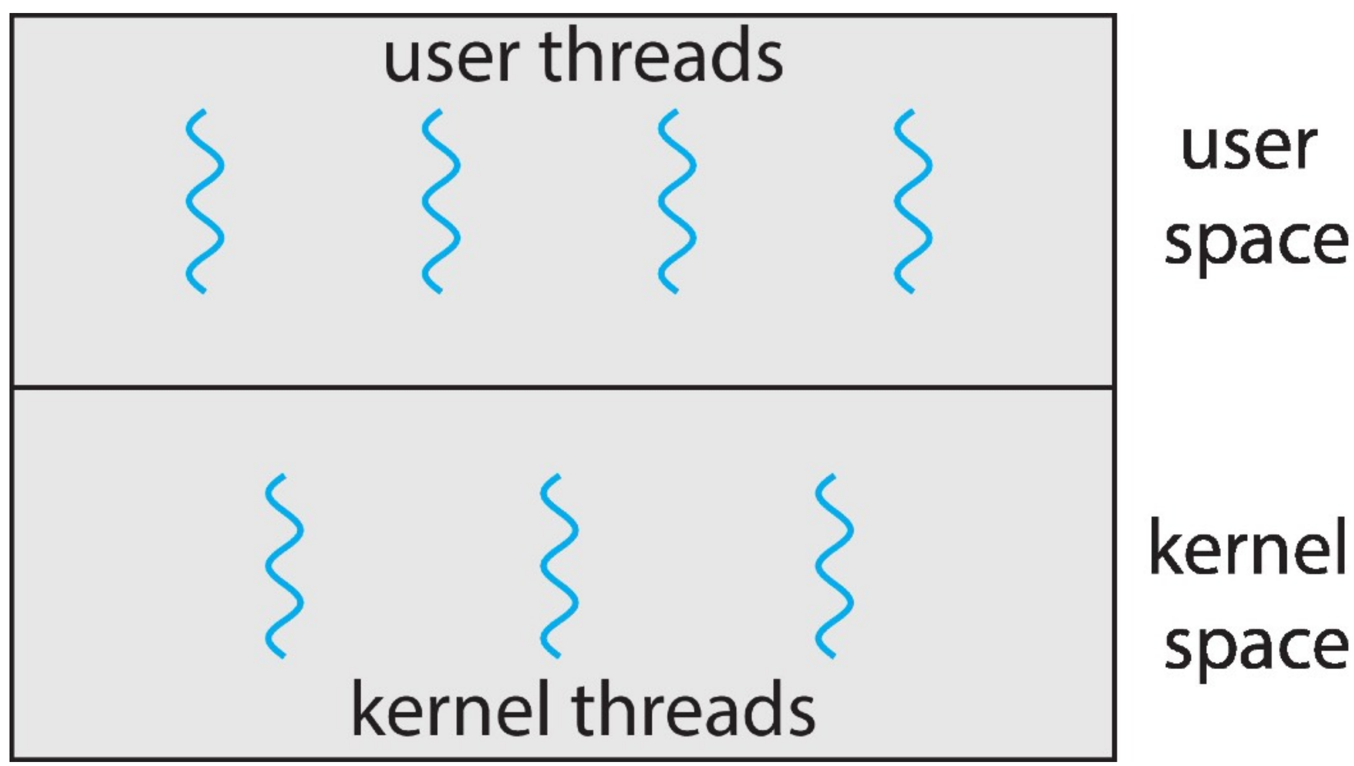
Performance gain from additional computing cores

- Speedup in latency is the ratio of latency on a system with fewer resources (e.g. processors) to latency on a system with more resources
- Consider an application that has S% serial components and (1-S)% parallel components; **Amdahl's law** explains the potential performance gain from adding additional computing cores to this application
  - speedup $\leq \dfrac{1}{S + \dfrac{(1-S)}{N}}$ where N is the number of processing cores in the system
  - as $N \to \infty$ theoretical speedup converges to $\dfrac{1}{S}$



Multithreading models

- Kernel threads supported at the kernel level
  - all contemporary operating systems support kernel threads
- User threads supported at the user level by a thread library, i.e. without kernel support



Kernel threads

- A kernel thread, also known as a lightweight process, is a thread directly managed by the OS
  - the kernel must manage and schedule threads/processes
- Switching between kernel threads of the same process requires a small context switch, hence it is slightly faster than switching between processes
  - only the values of registers, program counter, and stack pointer must be updated
  - memory management information does not need to be changed since threads share the process address space

User-level threads

- A user-level thread is a thread that the OS does not know about
  - the OS only knows about the process containing the threads
  - the OS only schedules the process (or kernel-level thread), not the user-level threads within the process
- The programmer uses a thread library (e.g. C-Threads) to manage threads
  - create and delete them, synchronize them, and schedule them
  - user threads can be scheduled non-preemptively (only switch on yield)

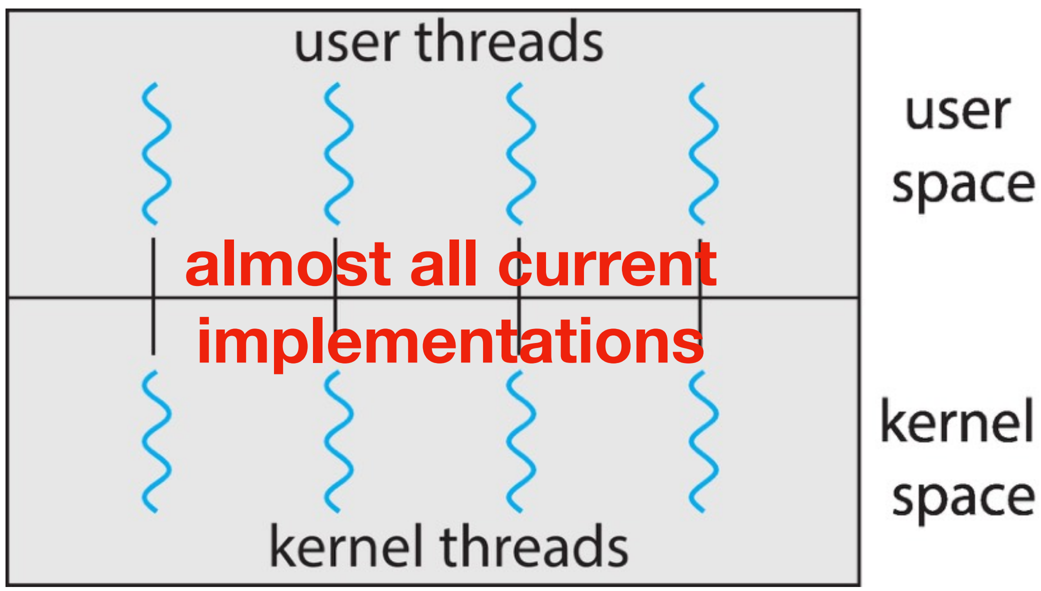
Advantages of user-level threads

- User-level thread scheduling is more flexible
  - allows a **problem-dependent** thread scheduling policy so each process might use a different scheduling algorithm for its own threads
  - a thread can voluntarily give up the processor by telling the scheduler that it **yields** to other threads of that process
- User-level threads do not require system calls to create them or context switches to move between them
  - thread management calls are library calls and much faster than system calls made by kernel threads

Disadvantages of user-level threads

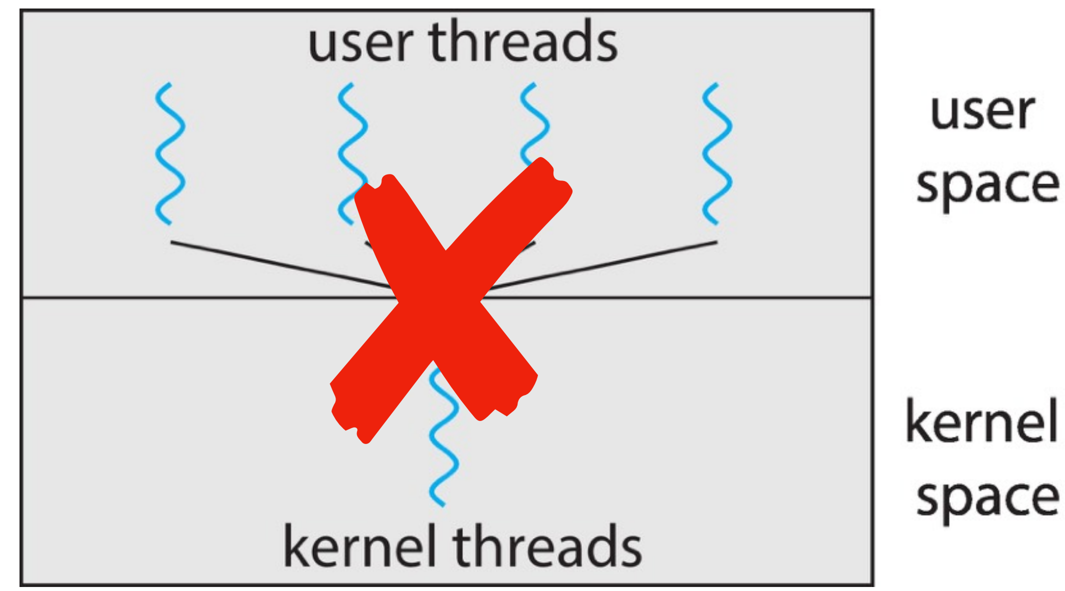
- Since the OS does not know about the existence of the user-level threads, it makes poor scheduling decisions
  - it schedules the process the same way as other processes, regardless of the number of user threads that it contains
    - so multiple user-level threads are unable to run in parallel on multicore systems
  - it may run a process that has idle threads only
  - if a user-level thread makes a blocking system call (e.g., waits for I/O), the entire process blocks
- Solving this problem requires communication between the kernel and the user-level thread manager
  - for kernel threads, the more threads a process creates, the more time slices the OS will dedicate to it

Threading models



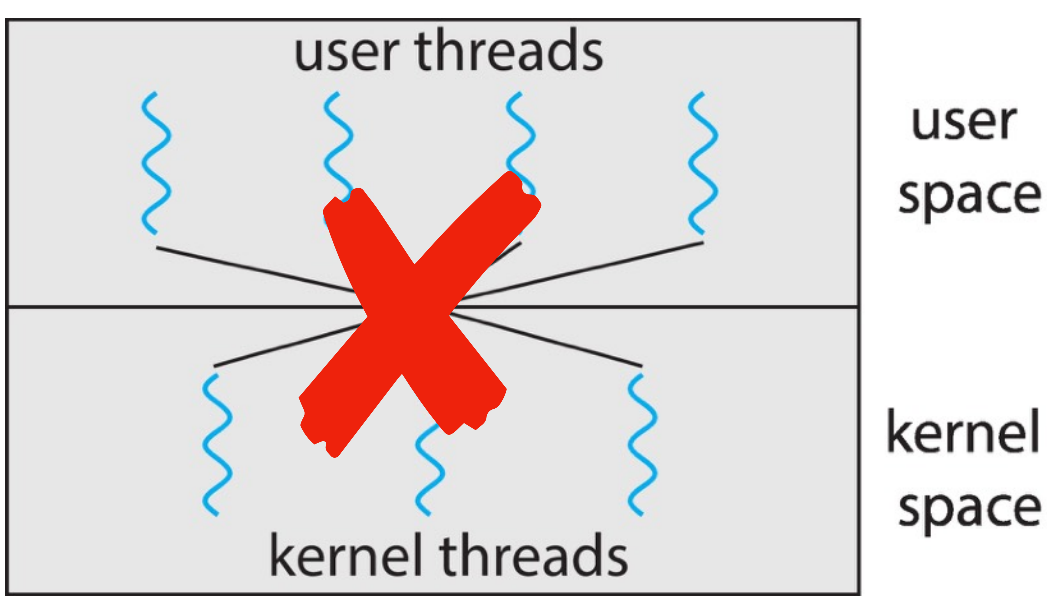
one-to-one

higher parallelism/concurrency



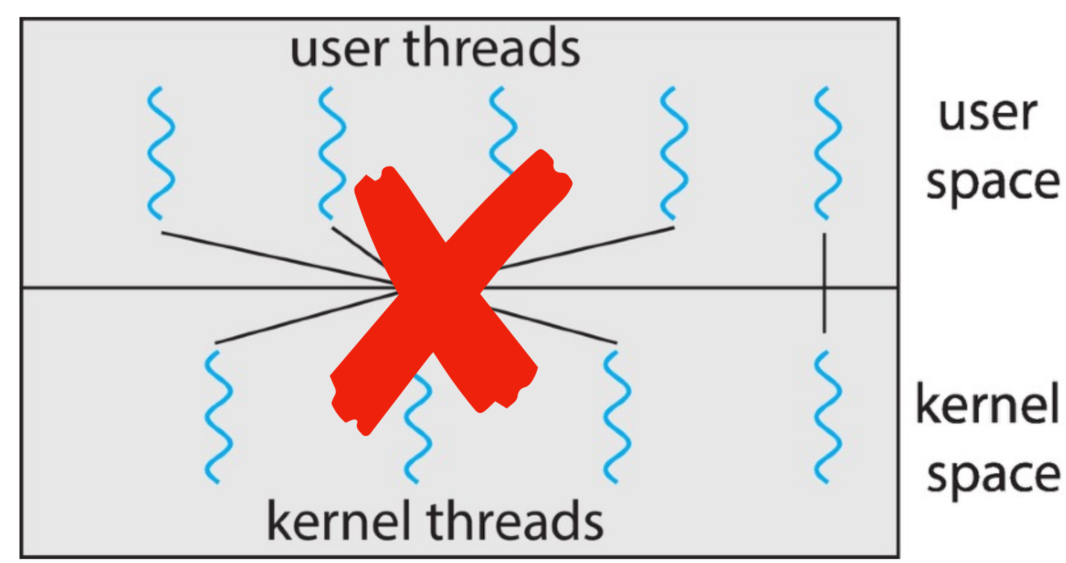
many-to-one

not suitable for multicore systems as threads of the same process can't run in parallel



many-to-many

a smaller or equal number of kernel threads



two-level