Operating System Concepts

Lecture 14: Threads

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

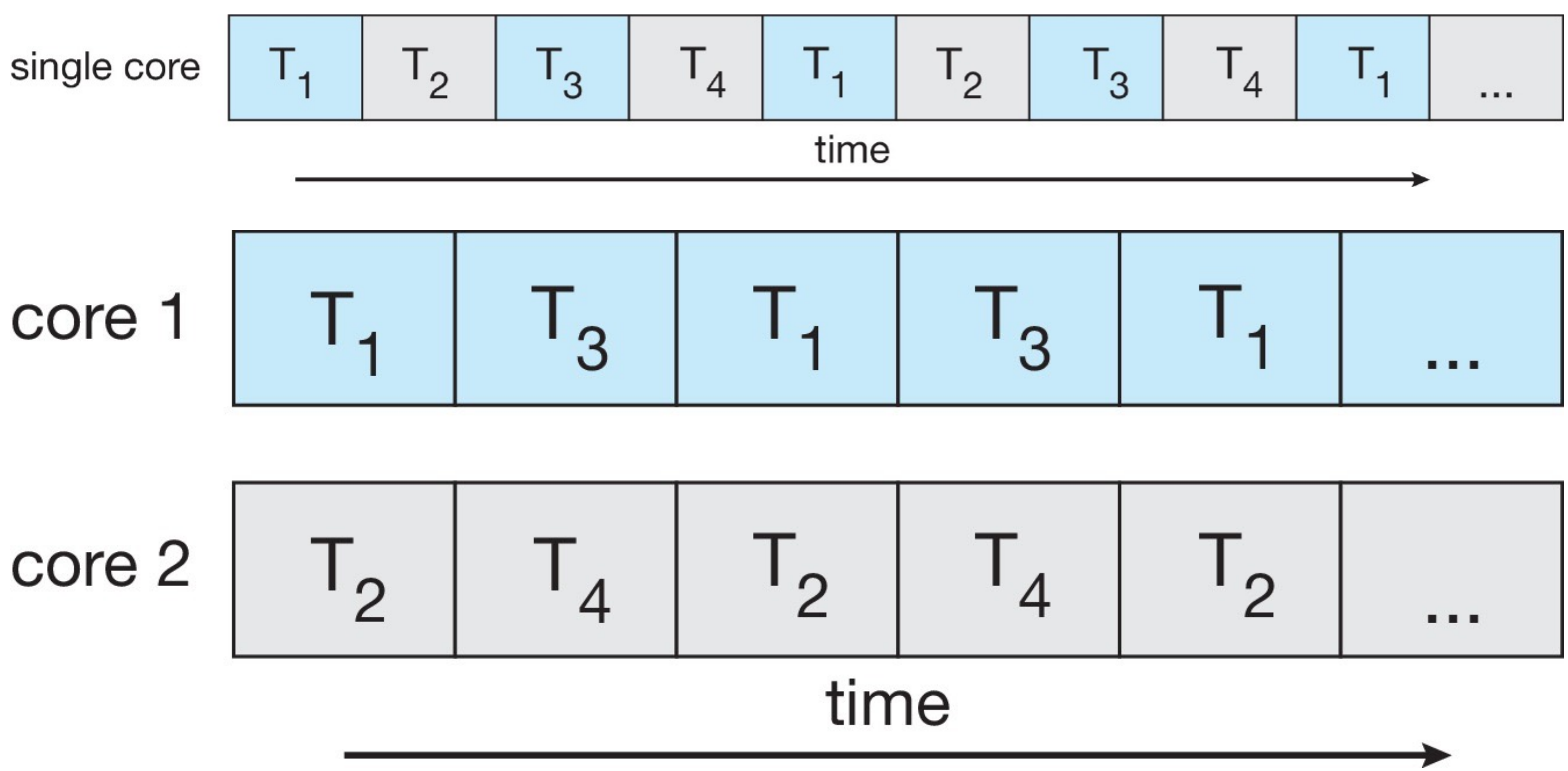
MWF 12:00-12:50 VVC 2 215

Today's class

- Multithreading: why and how
- Difference between user threads and kernel threads
- Threading issues

Thread motivation

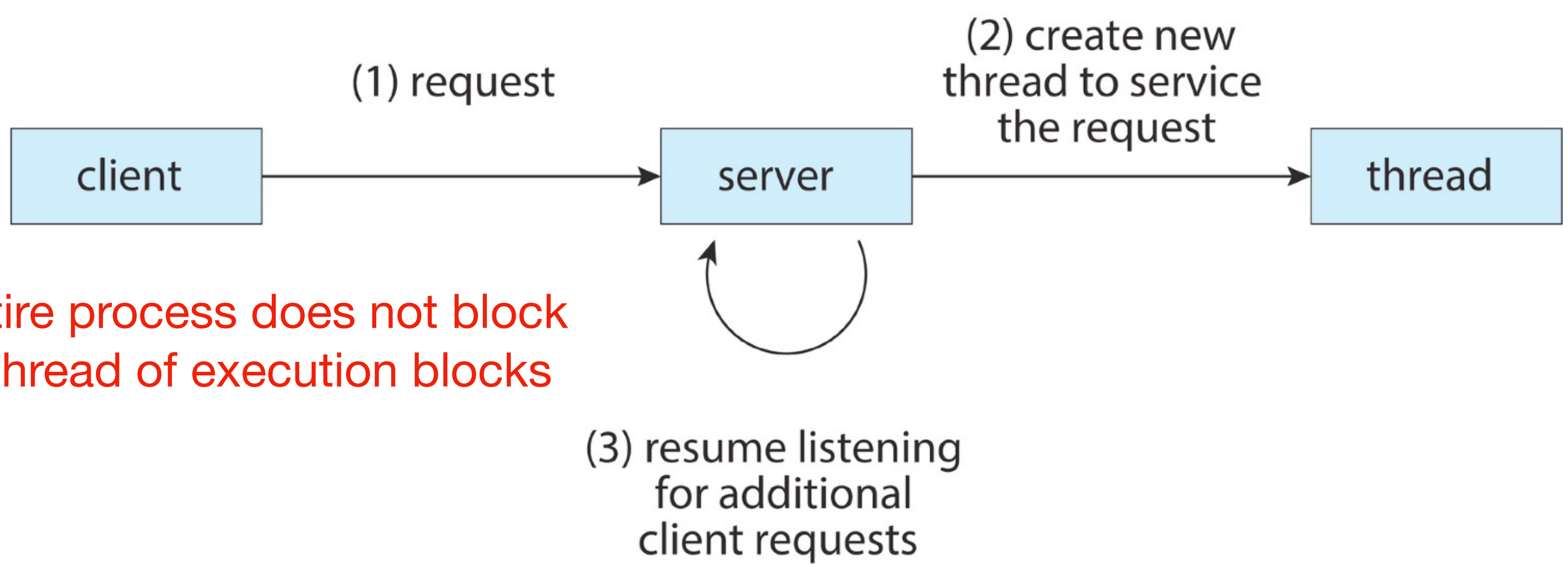
- modern systems have multiple processing cores
- Operating Systems need to be able to handle many things at once
- many other software applications running on multicore systems are multi-threaded
  - to achieve better performance or responsiveness



Why multithreading?

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

- a web browser might have a thread to display text and images, another thread to retrieve data from the network, and a thread for responding to user events (keystrokes, clicks, etc.)
- a web server accepting requests from hundreds of clients concurrently
- 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



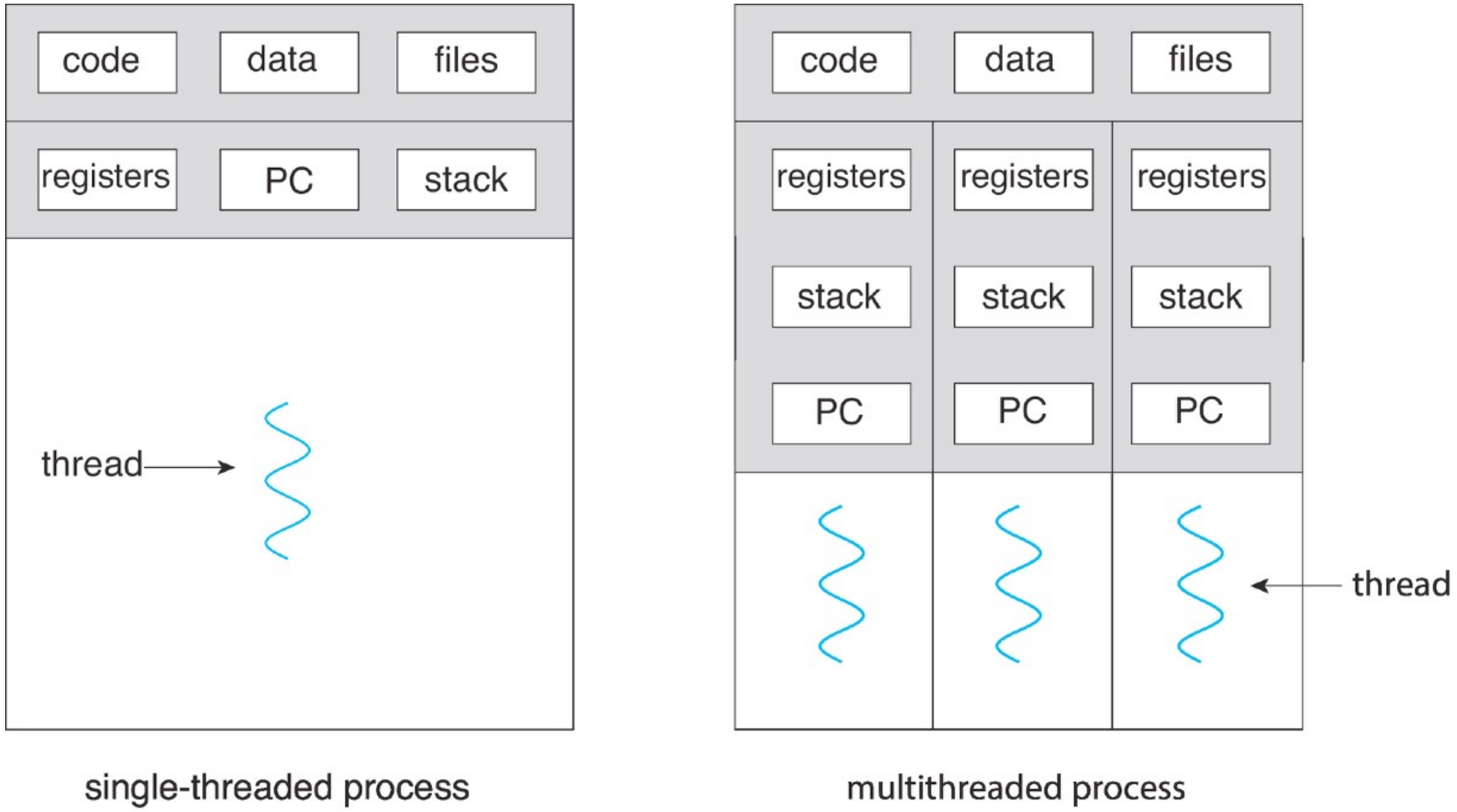
the entire process does not block if one thread of execution blocks

Thread abstraction

- Definition: a single execution stream within a process that represents a separately schedulable task
  - a process defines the protection domain
  - a 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
    - 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 local storage (TLS) which is specific to it
    - different from local variables in a function because TLS data are accessible across several function invocations

Single and multithreaded processes

- each process may have multiple threads of control within it
  - the address space of the process is shared among its threads (many threads per protection domain)
  - no system calls are required for cooperation between threads
  - hence it is simpler than message passing and shared-memory approaches



Benefits of multithreading

- responsiveness to user
  - especially important in the design of a user interface

Benefits of multithreading

- responsiveness to user
  - especially important in the design of a user interface
- 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

Benefits of multithreading

- responsiveness to user
  - especially important in the design of a user interface
- 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
  - it is less costly to create threads and context switch threads
  - in Linux, switching between processes takes 3-4 μsec while switching between threads takes 100 ns
  - can save on required memory by having multiple threads instead of multiple processes

Benefits of multithreading

- responsiveness to user
  - especially important in the design of a user interface
- 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
  - it is less costly to create threads and context switch threads
  - in Linux, switching between processes takes 3-4 μsec while switching between threads takes 100 ns
  - can save on required memory by having multiple threads instead of multiple processes
- scalability
  - threads can run in parallel on different processing cores; this is key for multiprocessor architecture

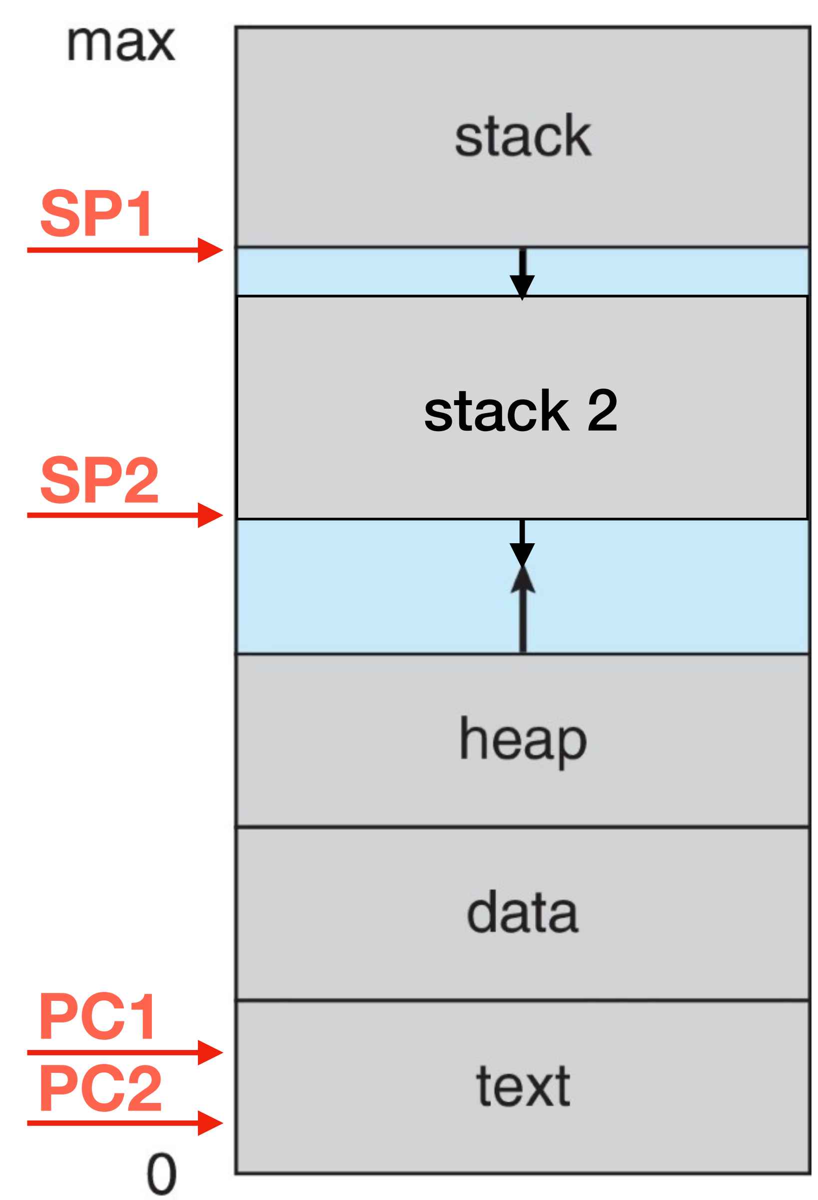
Concurrency versus parallelism

- 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

Modifying code to enable multithreading

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

void producer( ) {
    …
}
void consumer( ) {
    …
}
int main( ) {
    …
    in = 0; out = 0;
    fork_thread(producer( ));
    fork_thread(consumer( ));
    …
}
```



- forking a thread can be a system call to the kernel, or a procedure call to a thread library (user code)

Challenges of programming for multicore systems

- identifying and splitting tasks
  - tasks must be independent of one another and can run in parallel
  - tasks should perform equal work of equal value
  - task parallelism concerns the distribution of tasks across multiple cores

Challenges of programming for multicore systems

- identifying and splitting tasks
  - tasks must be independent of one another and can run in parallel
  - tasks should perform equal work of equal value
  - task parallelism concerns the distribution of tasks across multiple cores
- data splitting
  - data required by separate tasks must be spliced
  - data parallelism concerns the distribution of data across multiple cores

Challenges of programming for multicore systems

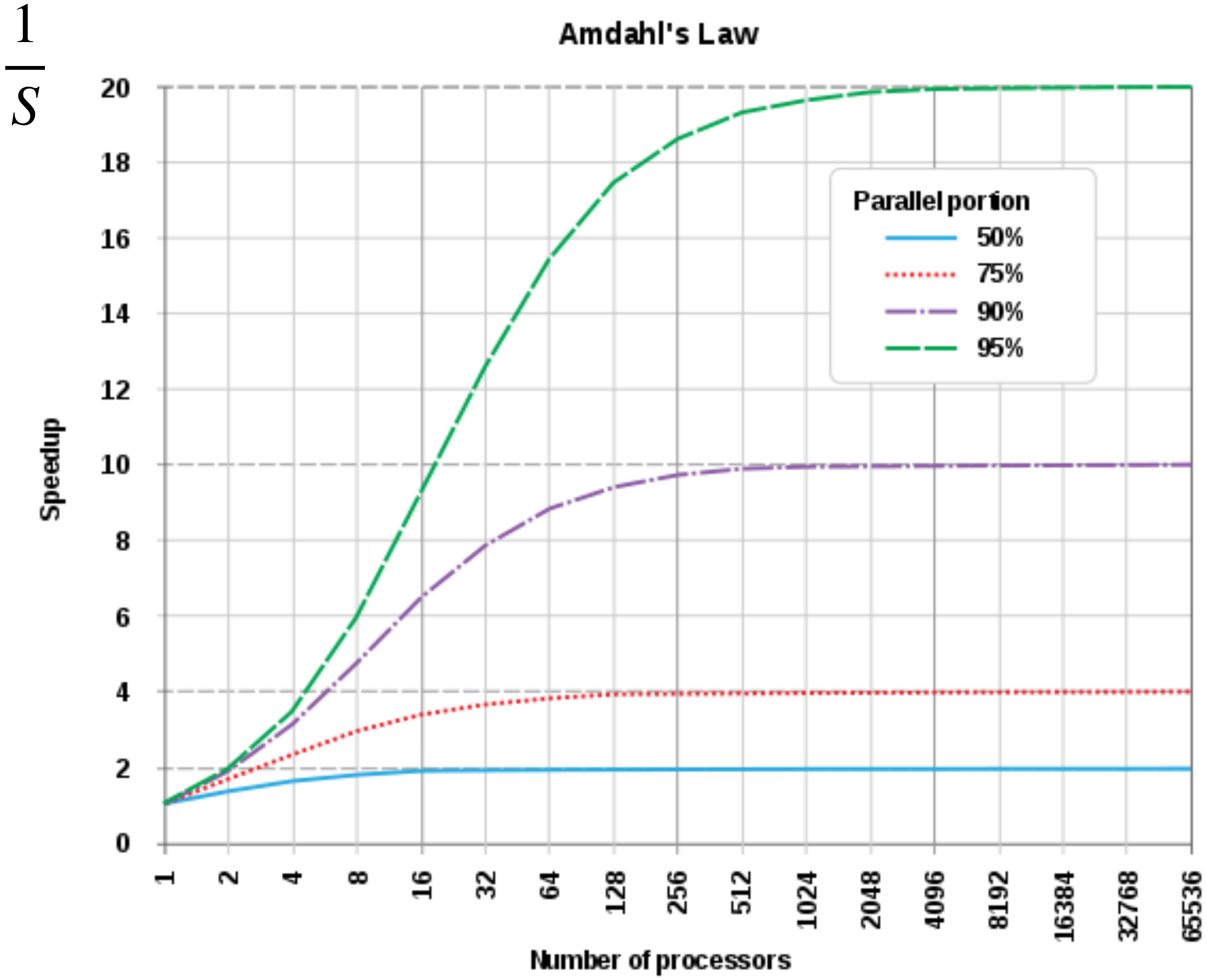
- <u>identifying and splitting tasks</u>
  - tasks must be independent of one another and can run in parallel
  - tasks should perform equal work of equal value
  - task parallelism concerns the distribution of tasks across multiple cores
- <u>data splitting</u>
  - data required by separate tasks must be spliced
  - data parallelism concerns the distribution of data across multiple cores
- <u>data dependency</u>
  - task execution must be synchronized if there is a dependency between data they access

Challenges of programming for multicore systems

- identifying and splitting tasks
  - tasks must be independent of one another and can run in parallel
  - tasks should perform equal work of equal value
  - task parallelism concerns the distribution of tasks across multiple cores
- data splitting
  - data required by separate tasks must be spliced
  - 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)
    - scheduler can run threads in any order
    - scheduler can switch threads at any time

Performance gain from adding additional computing cores

- consider an application that has S% serial component and (1-S)% parallel component
- Amdahl's law explains the potential performance gain from adding additional computing cores to this application
    - theoretical speedup = $\frac{1}{(S + \frac{(1-S)}{N})}$
      where N is the number of processing cores
- as N approaches infinity, the speedup converges to $\frac{1}{S}$

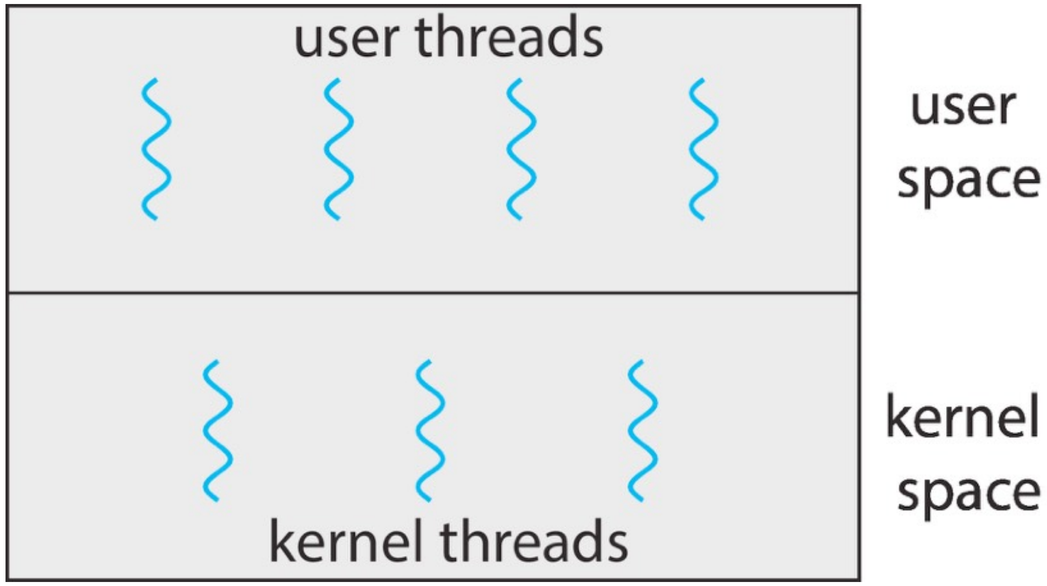


Kernel threads

- a kernel thread, also known as a lightweight process, is a thread directly managed (scheduled) 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
  - just 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

- example: the C-Threads package
- 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 to manage threads
  - create and delete them, synchronize them, and schedule them
  - user threads can be scheduled non-preemptively (only switch on yield)



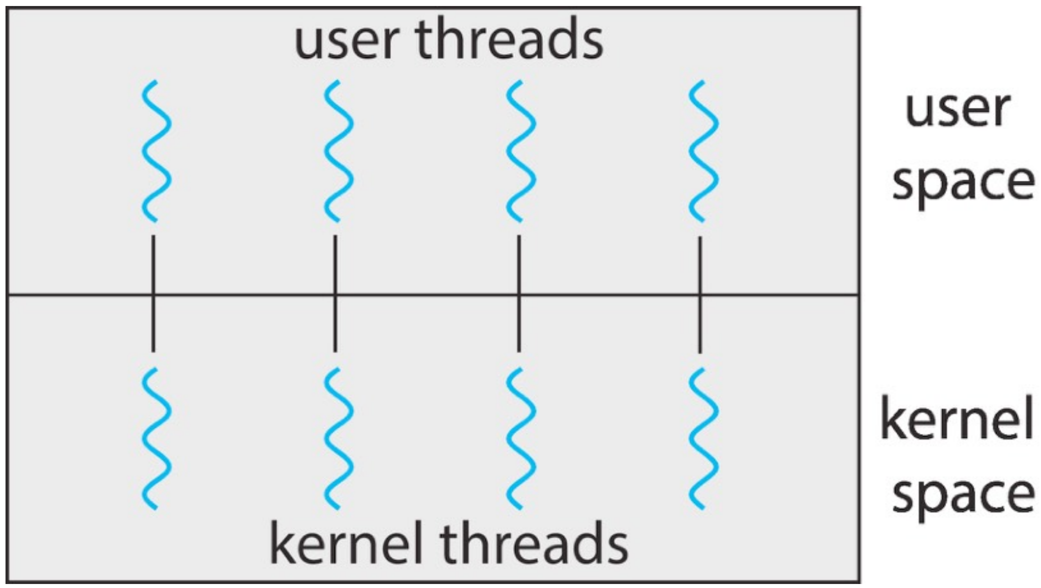
Advantages of user-level threads

- there is no context switch involved when switching threads
- user-level thread scheduling is more flexible
  - a user-level code can define a problem-dependent thread scheduling policy
  - 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
- 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
- user-level threads are typically much faster than kernel threads

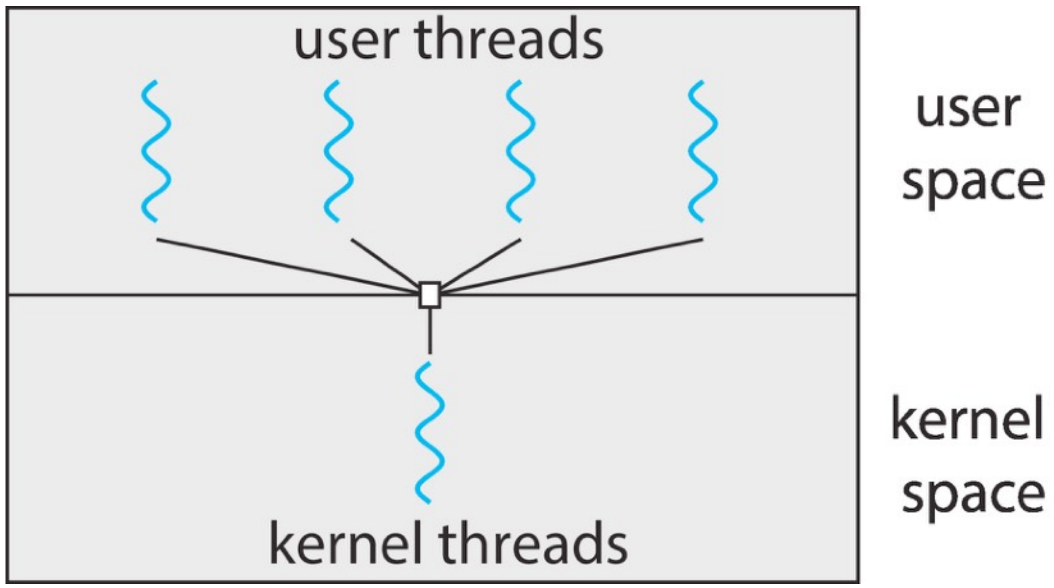
Disadvantages of user-level threads

- since the OS does not know about the existence of the user-level threads, it may make poor scheduling decisions
  - it schedules the process the same way as other processes, regardless of the number of user threads
    - multiple user-level threads are unable to run in parallel on multicore systems
  - it may run a process that only has idle threads
  - 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

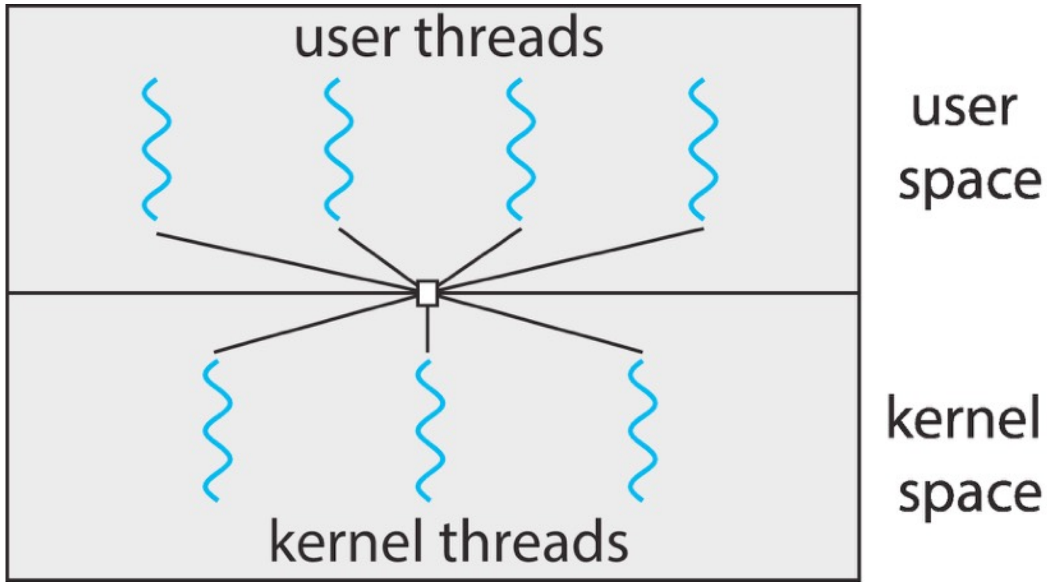
Threading models



one-to-one

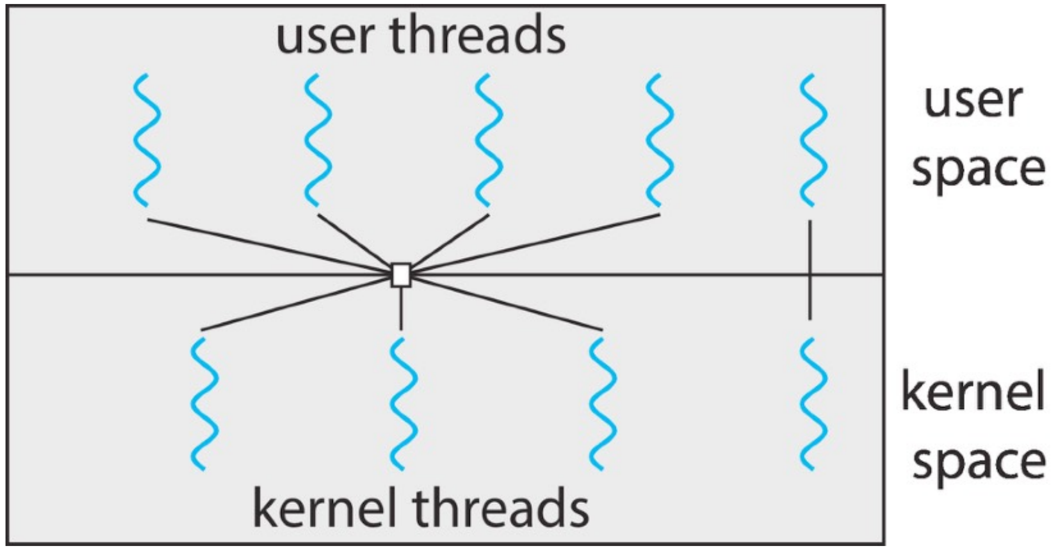


many-to-one



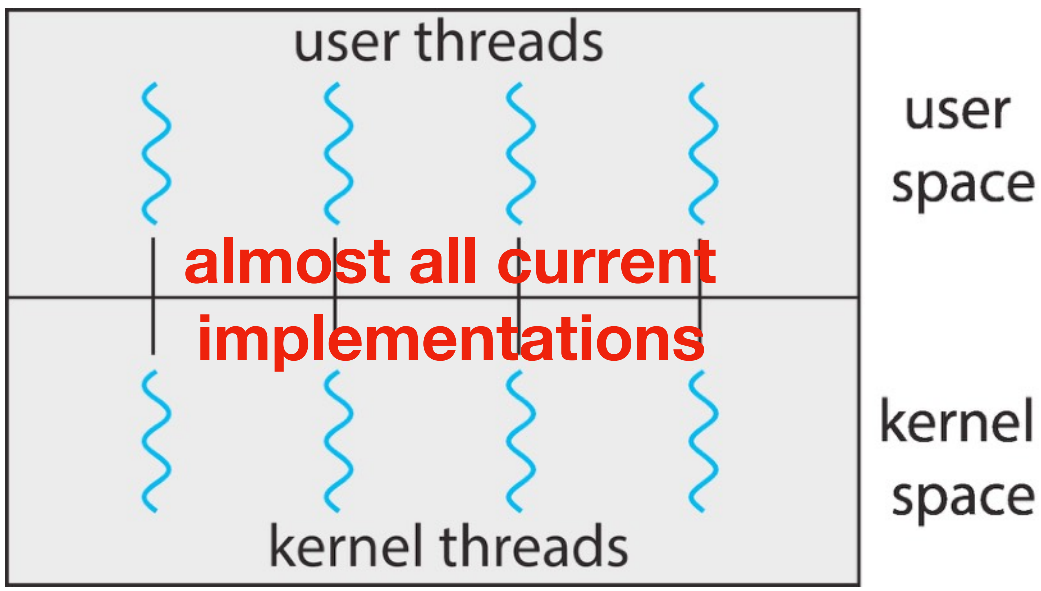
many-to-many

a smaller or equal number of kernel threads

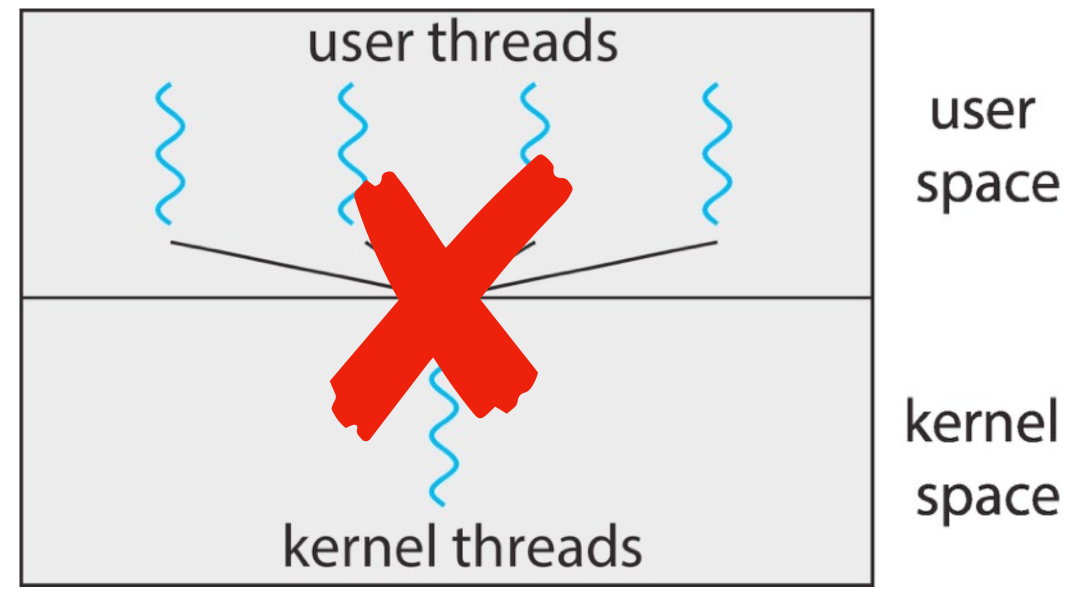


two-level

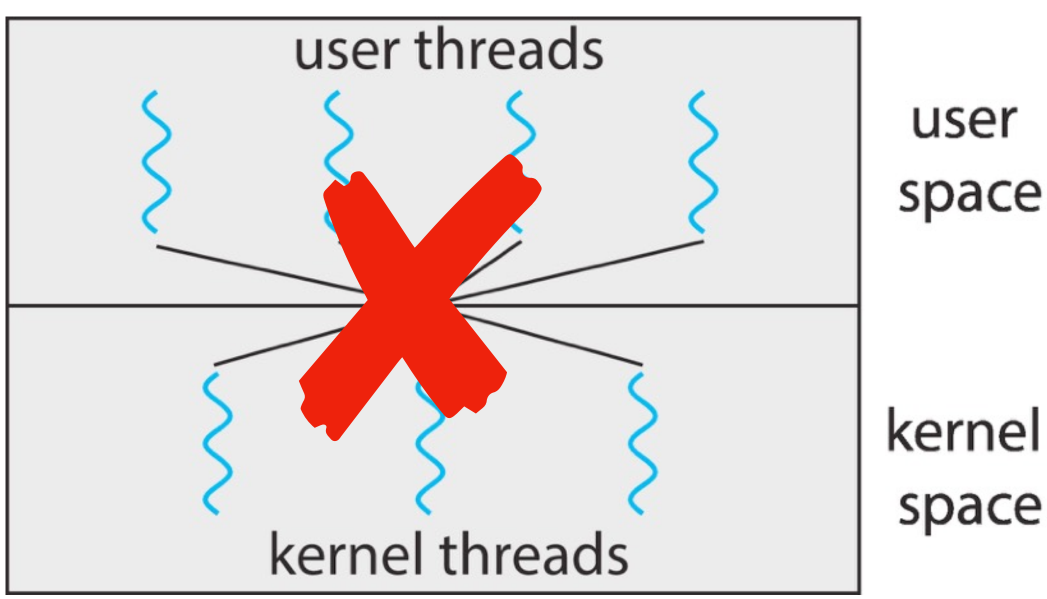
Threading models



one-to-one

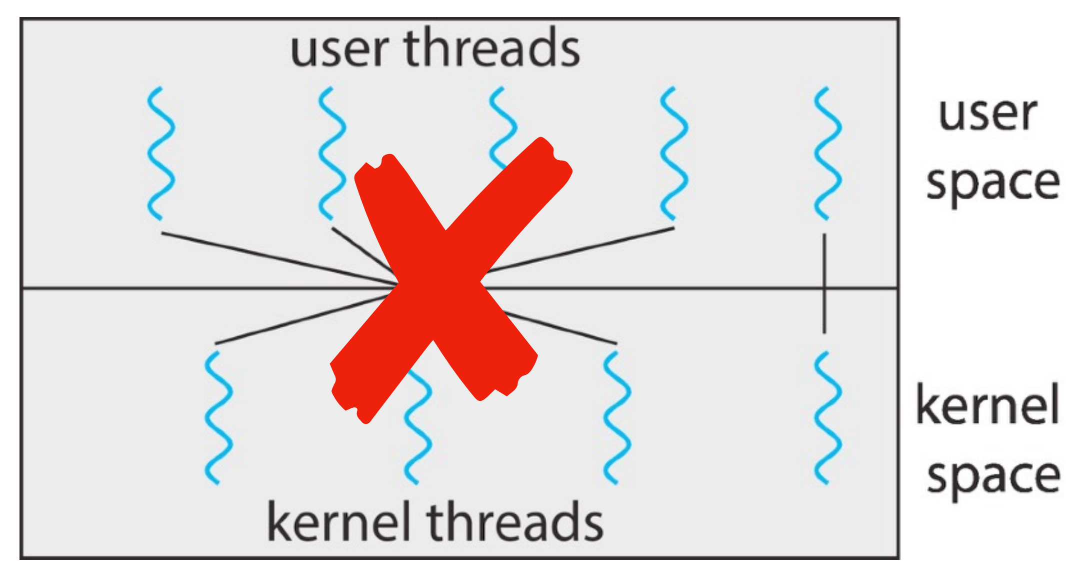


many-to-one



many-to-many

a smaller or equal number of kernel threads



two-level