

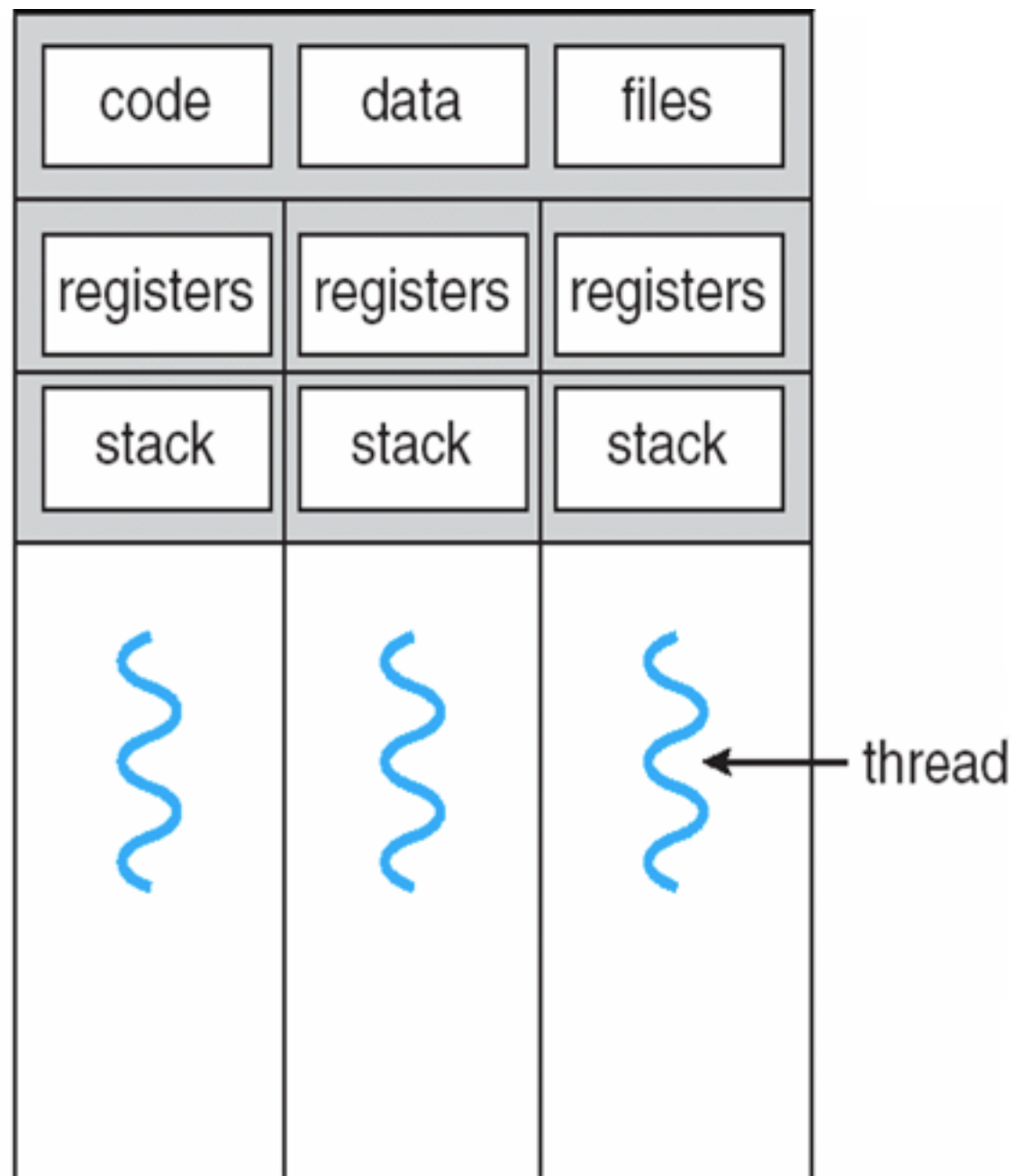
Implementing threads

Module 4 self study material

Operating systems 2020

1DT003, 1DT044 and 1DT096

Implementing threads



multi-threaded process

What part of
threading should
be dealt with in
user space?

What parts of
threading should
be dealt with in
kernel space?

Threading models

Support for threads must be provided either at the **user** level or by the **kernel**.



4.2) Threading Models

Support for threads must be provided either at the user level or by the kernel.

User-level Threads

Supported above the kernel and are managed without kernel support.

User Mode

`mode bit = 1`

How are user level threads implemented?

How does user level threads relate to kernel threads?

Kernel-level Threads

Supported and managed directly by the operating system.

Kernel Mode

`mode bit = 0`

4.3) Thread Libraries

A thread library provides programmers with an API for creating and managing threads.

Two primary ways of implementing a thread library exists:

User-level Threads

All **code** and **data structures** for the library exist in user space.

Invoking a function in the API results in a **local function call** in user **space** and not a system call.

User Mode

`mode bit = 1`

Kernel-level Threads

All **code** and **data structures** for the library exists in **kernel space**.

Invoking a function in the API typically results in a **system call** to the kernel.

Kernel Mode

`mode bit = 0`

4.3) Thread Libraries

A Thread library provides programmers with an API for creating and managing threads.

Two primary ways of implementing a thread library exists:

User-level Threads

Library entirely in user space

Three primary thread libraries:

- ★ POSIX Pthreads
- ★ Win32 threads
- ★ Java threads

User Mode

`mode bit = 1`

Kernel-level Threads

Examples:

- ★ Windows XP/2000
- ★ Solaris
- ★ Linux
- ★ Tru64 UNIX
- ★ Mac OS X

Kernel-level library supported by the OS

Kernel Mode

`mode bit = 0`

Kernel-level threads

(issues)

User space



Kernel space

Kernel-level threads

(issues)

User space

- ★ The kernel knows about and manages all threads.
- ★ One process control block (PCB) per process.
- ★ One thread control block (TCB) per thread in the system.
- ★ Provide system calls to create and manage threads from user space.

Kernel space

Kernel-level threads **advantages**

User space

Advantages

- ★ The kernel has full knowledge of all threads.
- ★ Scheduler may decide to give more CPU time to a process having a large number of threads.
- ★ Good for applications that frequently block.

Kernel space

Kernel-level threads **disadvantages**

User space

Disadvantages

- ★ Kernel manage and schedule all threads.
- ★ Significant overhead and increase in kernel complexity.
- ★ Kernel level threads are slow and inefficient compared to user level threads.
- ★ Thread operations are hundreds of times slower compared to user-level threads.

Kernel space

User-level threads

(issues)

User space



Kernel space

User-level threads

(issues)

User space

- ★ Threads managed entirely by the run-time system (user-level library).
- ★ Ideally, thread operations should be as fast as a function call.
- ★ The kernel knows nothing about user-level threads and manage them as if they where single-threaded processes.

Kernel space

User-level threads **advantages**

Advantages

User space

- ★ Can be implemented on an OS that does not support kernel-level threads.
- ★ Does not require modifications of the OS.
- ★ **Simple representation:** (PC, registers, stack and small thread control block) all stored in the user-level process address space.
- ★ **Simple management:** Creating, switching and synchronizing threads done in user-space without kernel intervention.
- ★ **Fast and efficient:** switching threads not much more expensive than a function call.

Kernel space

User-level threads **disadvantages**

Disadvantages

User space

- ★ Not a perfect solution (a trade off).
- ★ Lack of coordination between the user-level thread manager and the kernel.
- ★ OS may make poor decisions like:
 - ✓ scheduling a process with idle threads
 - ✓ blocking a process due to a blocking thread even though the process has other threads that can run
 - ✓ giving a process as a whole one time slice irrespective of whether the process has 1 or 1000 threads
 - ✓ unschedule a process with a thread holding a lock.
- ★ May require communication between the kernel and the user-level thread manager (scheduler activations) to overcome the above problems.

Kernel space

Implementing user-level threads

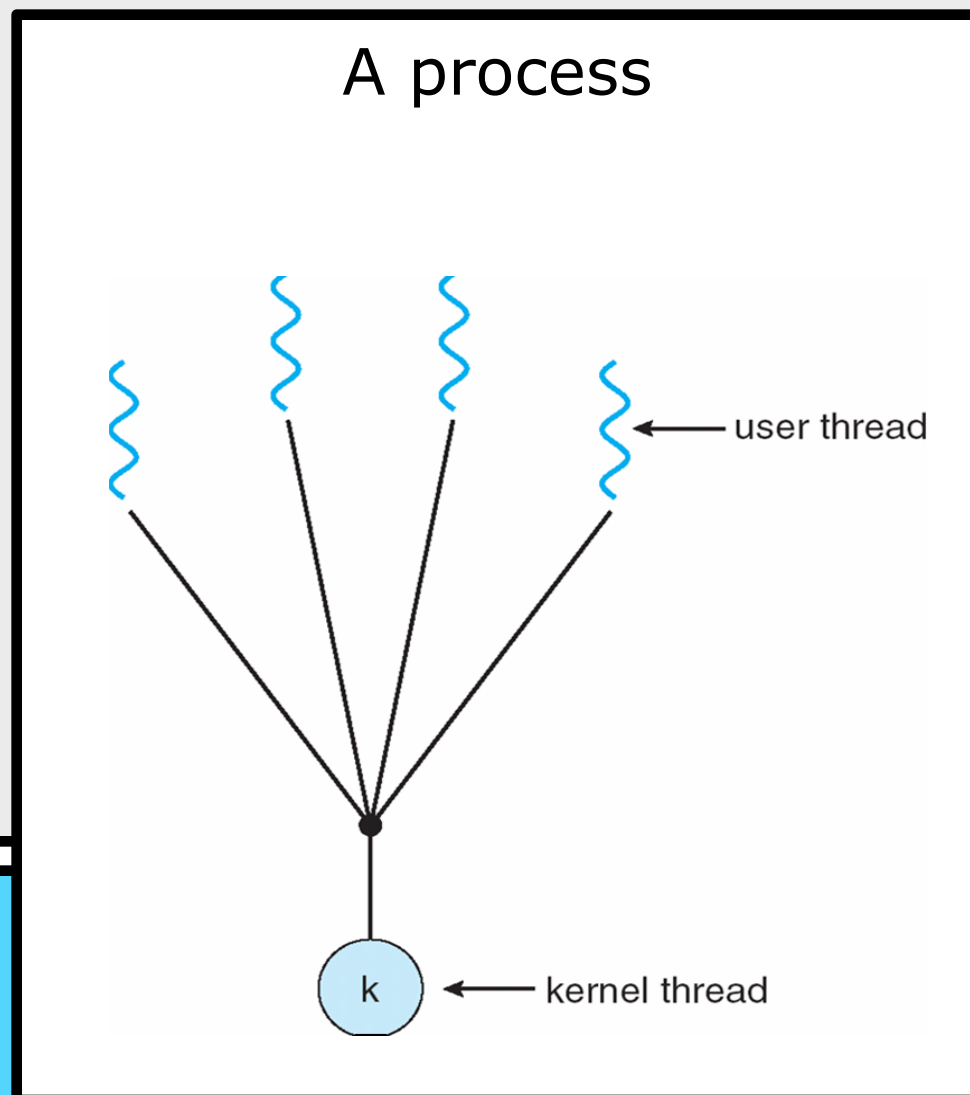
User space



Kernel space

Implementing **many-to-one** user-level threads

User space



If a process has 100,000 user-level threads but only one kernel thread, then the process can only run one user-level thread at a time because there is only one kernel-level thread associated with it.

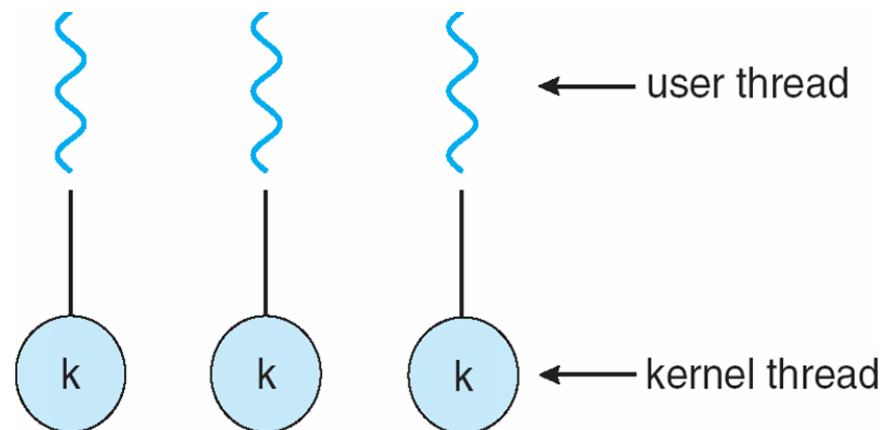
The kernel has no knowledge of user-level threads. From its perspective, a process is an opaque black box that occasionally makes system calls.

Kernel space

Implementing **one-to-one** user-level threads

User space

A process



If a process has multiple kernel-level threads, then it can potentially execute multiple instructions in parallel on a multicore machine.

This model makes thread creation "expensive" and most implementations restrict the number of threads supported by the system.

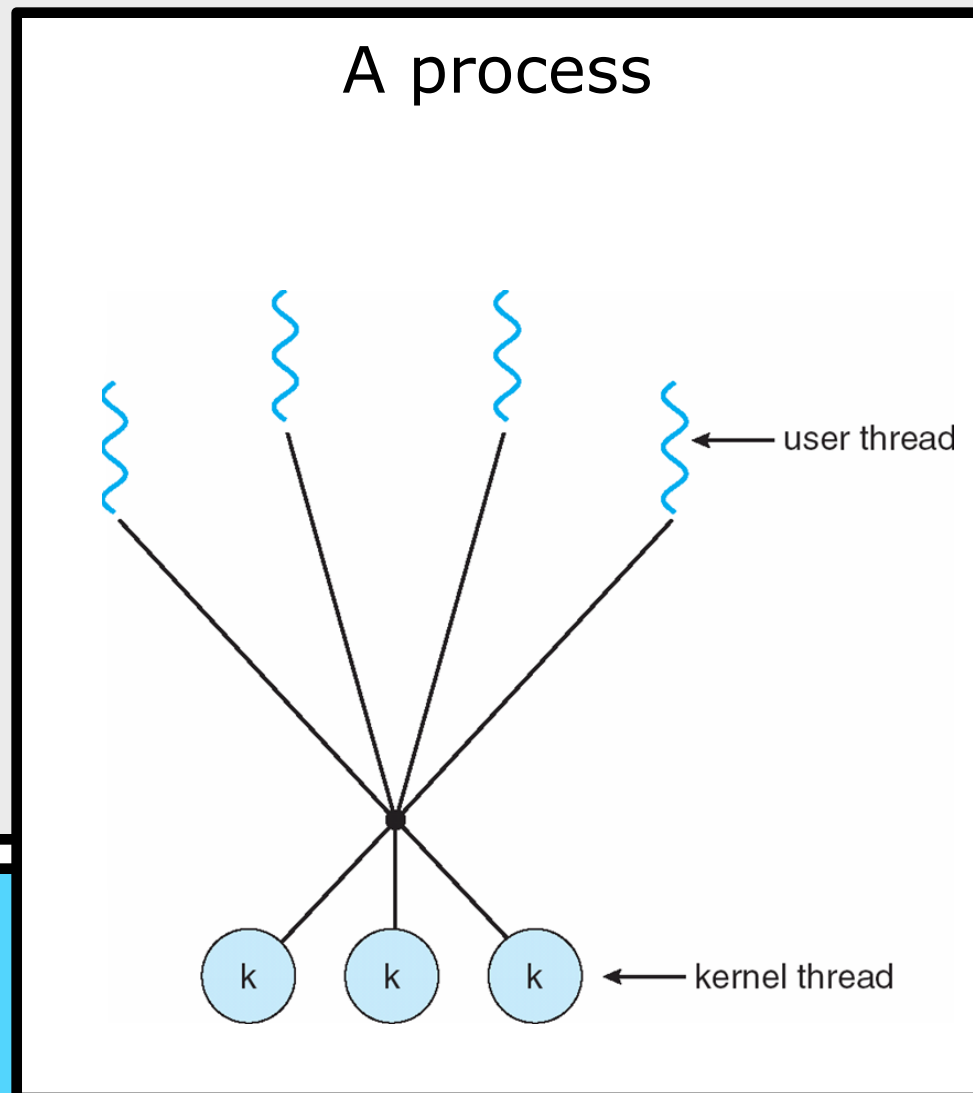
Kernel space

Implementing **many-to-many** user-level threads

User space

A process request some fixed number of kernel-level threads.

Multiple user level threads may execute in parallel, and the process can have fine-grained control over how threads execute.

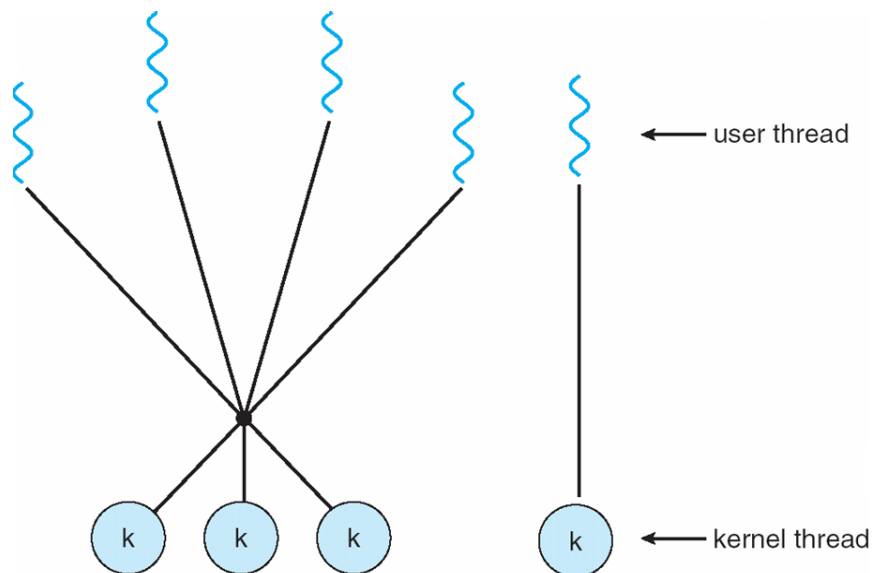


Kernel space

Implementing **two-level** user-level threads

User space

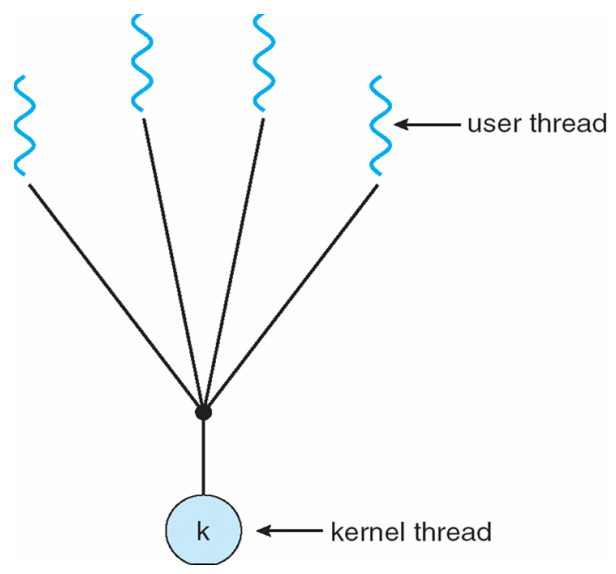
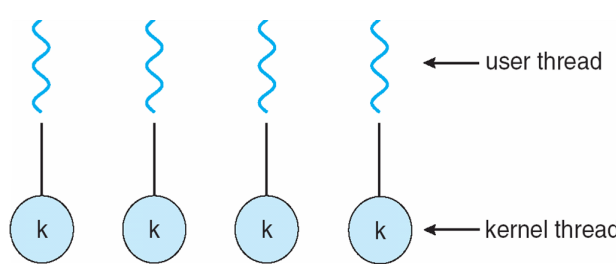
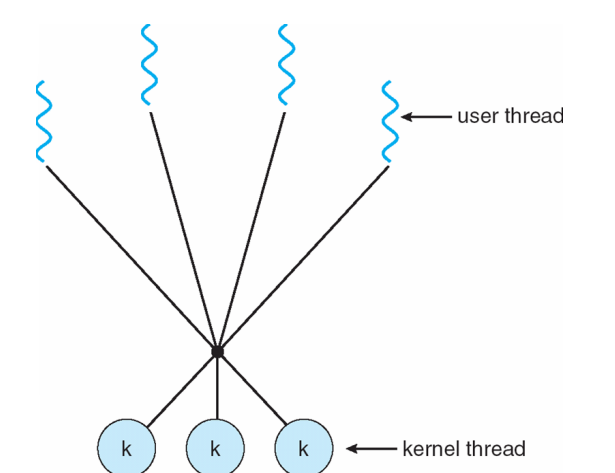
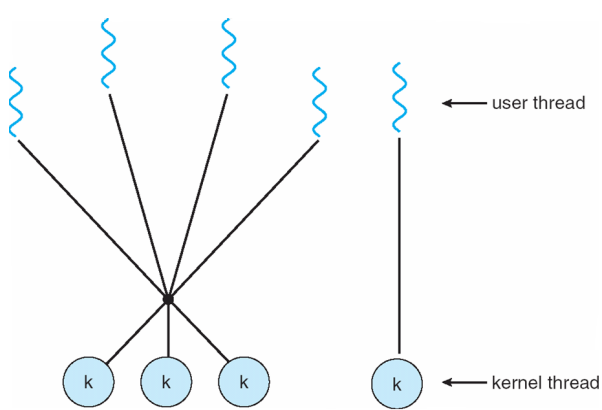
A process



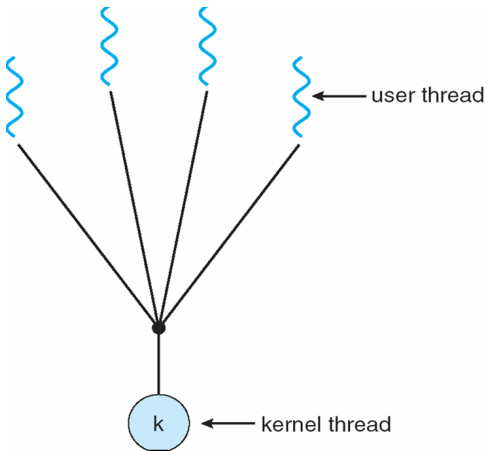
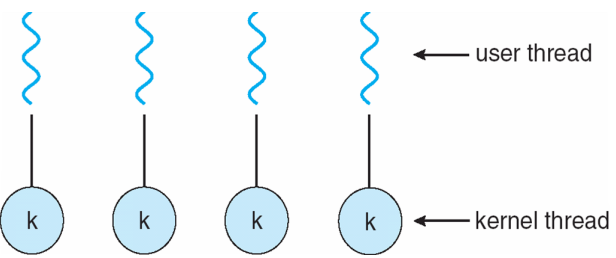
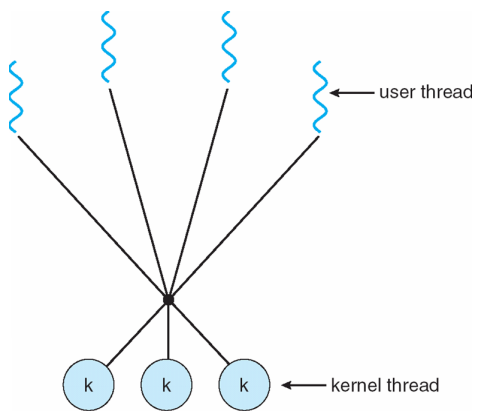
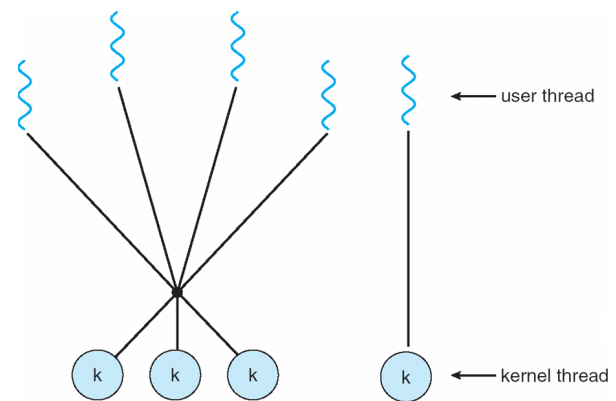
Similar to many-to-many, except that it allows a user thread to be bound to a kernel thread.

Kernel space

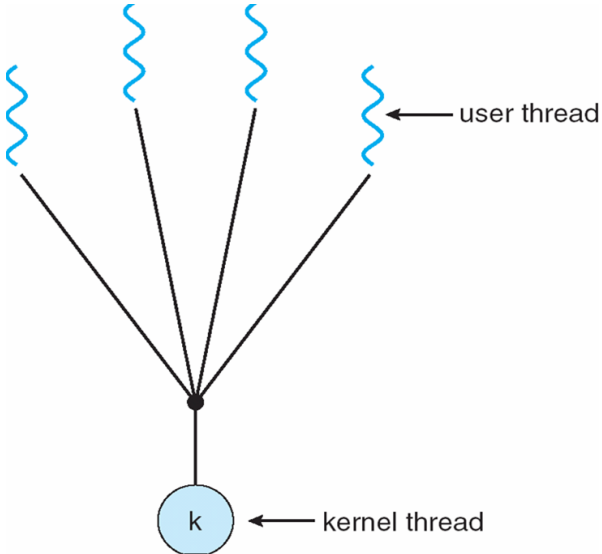
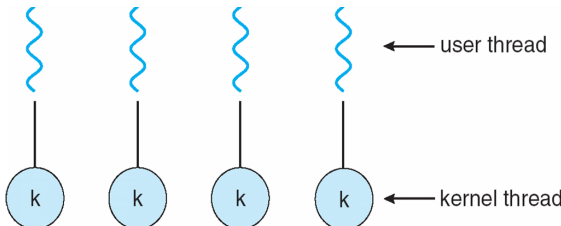
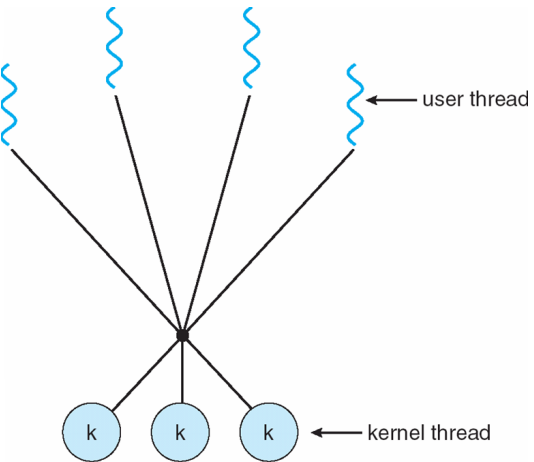
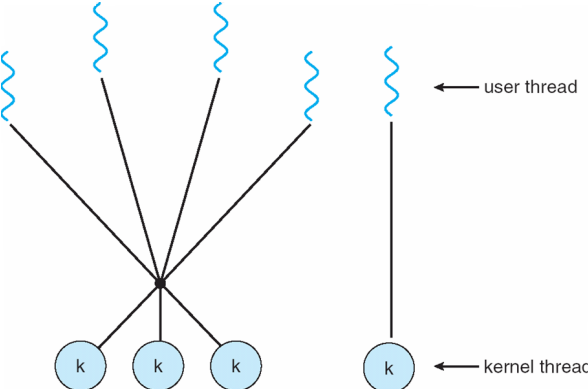
Multithreading models

Many-to-One	One-to-One	Many-to-Many	Two-level-model
 <p>The diagram shows four wavy lines representing user threads converging at a single point, which then connects to a single circle labeled 'k' representing a kernel thread. Labels 'user thread' and 'kernel thread' point to their respective elements.</p>	 <p>The diagram shows four wavy lines representing user threads, each connected by a straight line to a separate circle labeled 'k' representing a kernel thread. Labels 'user thread' and 'kernel thread' point to their respective elements.</p>	 <p>The diagram shows four wavy lines representing user threads converging at a single point, which then branches out to connect to three separate circles labeled 'k' representing kernel threads. Labels 'user thread' and 'kernel thread' point to their respective elements.</p>	 <p>The diagram shows four wavy lines representing user threads. Three of them converge at a single point and connect to three separate circles labeled 'k' representing kernel threads. The fourth wavy line connects directly to a single circle labeled 'k' representing a kernel thread. Labels 'user thread' and 'kernel thread' point to their respective elements.</p>
<p>Many user-level threads mapped to a single kernel thread.</p> <p>Thread management is done by the thread library in user space.</p>	<p>Each user-level thread maps to one kernel thread.</p> <div><p>This models makes thread creation "expensive" and most implementations restrict the number of threads supported by the system.</p></div>	<p>Allows many user level threads to be mapped to many kernel threads.</p> <p>Allows the operating system to create a sufficient number of kernel threads.</p>	<p>Similar to many-to-many, except that it allows a user thread to be bound to a kernel thread.</p>

Multithreading models and blocking

Many-to-One	One-to-One	Many-to-Many	Two-level-model
			
<p>Thread management is done by the thread library in user space but entire process will block if a thread makes a blocking system call.</p> <div>One blocks all</div> <p>Only one thread can access the kernel at a time, hence threads cannot run in parallel on multiprocessors.</p>	<p>Provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.</p> <div>One blocks solely itself</div> <p>Allows for threads to run in parallel on multiprocessors.</p>	<p>Multiplexes many user-level threads to a smaller or equal number of kernel threads.</p> <div>One blocks some</div> <p>A compromise between many-to-one and one-to-one.</p>	<p>Similar to many-to-many, except that it allows a user thread to be bound to a kernel thread.</p>

Multithreading models

Many-to-One	One-to-One	Many-to-Many	Two-level-model
 <p>The diagram shows four wavy blue lines representing user threads converging at a single point, which then connects to a single light blue circle labeled 'k' representing a kernel thread. An arrow points to one of the wavy lines with the label 'user thread', and another arrow points to the circle 'k' with the label 'kernel thread'.</p>	 <p>The diagram shows four wavy blue lines representing user threads, each connected by a straight line to a separate light blue circle labeled 'k' representing a kernel thread. An arrow points to one of the wavy lines with the label 'user thread', and another arrow points to one of the circles 'k' with the label 'kernel thread'.</p>	 <p>The diagram shows four wavy blue lines representing user threads converging at a single point, which then connects to three light blue circles labeled 'k' representing kernel threads. An arrow points to one of the wavy lines with the label 'user thread', and another arrow points to one of the circles 'k' with the label 'kernel thread'.</p>	 <p>The diagram shows five wavy blue lines representing user threads. Four of them converge at a single point connected to three light blue circles labeled 'k' representing kernel threads. The fifth wavy line is connected by a straight line to a separate light blue circle labeled 'k' representing a kernel thread. An arrow points to one of the wavy lines with the label 'user thread', and another arrow points to one of the circles 'k' with the label 'kernel thread'.</p>
<p>Examples:</p> <ul style="list-style-type: none">★ Solaris Green Threads★ GNU Portable Threads	<p>Examples:</p> <ul style="list-style-type: none">★ Windows NT/XP/2000★ Linux★ Solaris 9 and later	<p>Examples:</p> <ul style="list-style-type: none">★ Solaris prior to version 9★ Windows NT/2000 with the ThreadFiber package	<p>Examples:</p> <ul style="list-style-type: none">★ IRIX★ HP-UX★ Tru64 UNIX★ Solaris 8 and earlier

User-level thread scheduling

User space

Scheduling of user-level threads among the available kernel-level threads done by user-level scheduler.

How to decide when to switch threads?

There are two main methods: **preemptive** & **cooperative**.

many-to-one

one-to-one

many-to-many

two-level

Kernel space

Preemptive user level thread scheduling

In the **preemptive** model, you'll have something like a timer signal that causes execution flow to jump to a central dispatcher thread, which chooses the next thread to run.

User space

many-to-one

one-to-one

many-to-many

two-level



Timer

Kernel space

Cooperative user-level thread scheduling

User space



In a cooperative model, threads **yield** to each other, either explicitly (e.g., by calling a `yield()` function you'll provide) or implicitly (e.g., requesting a lock held by another thread).

many-to-one

one-to-one

many-to-many

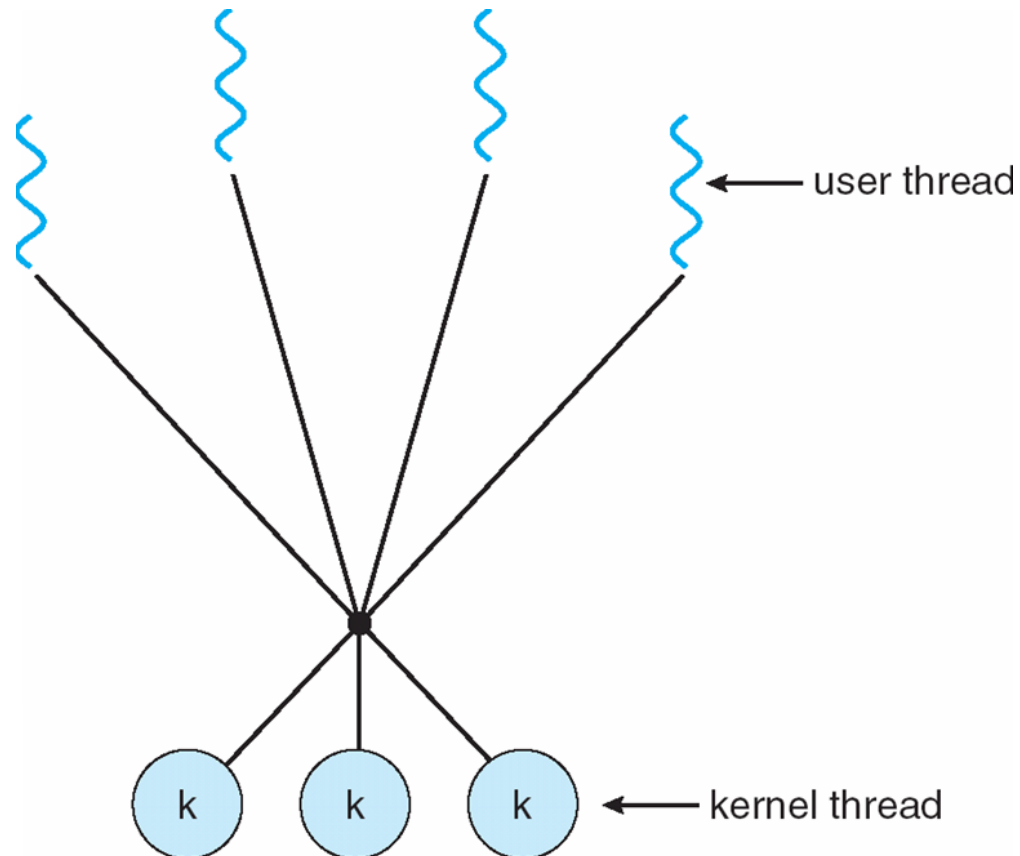
two-level

Kernel space

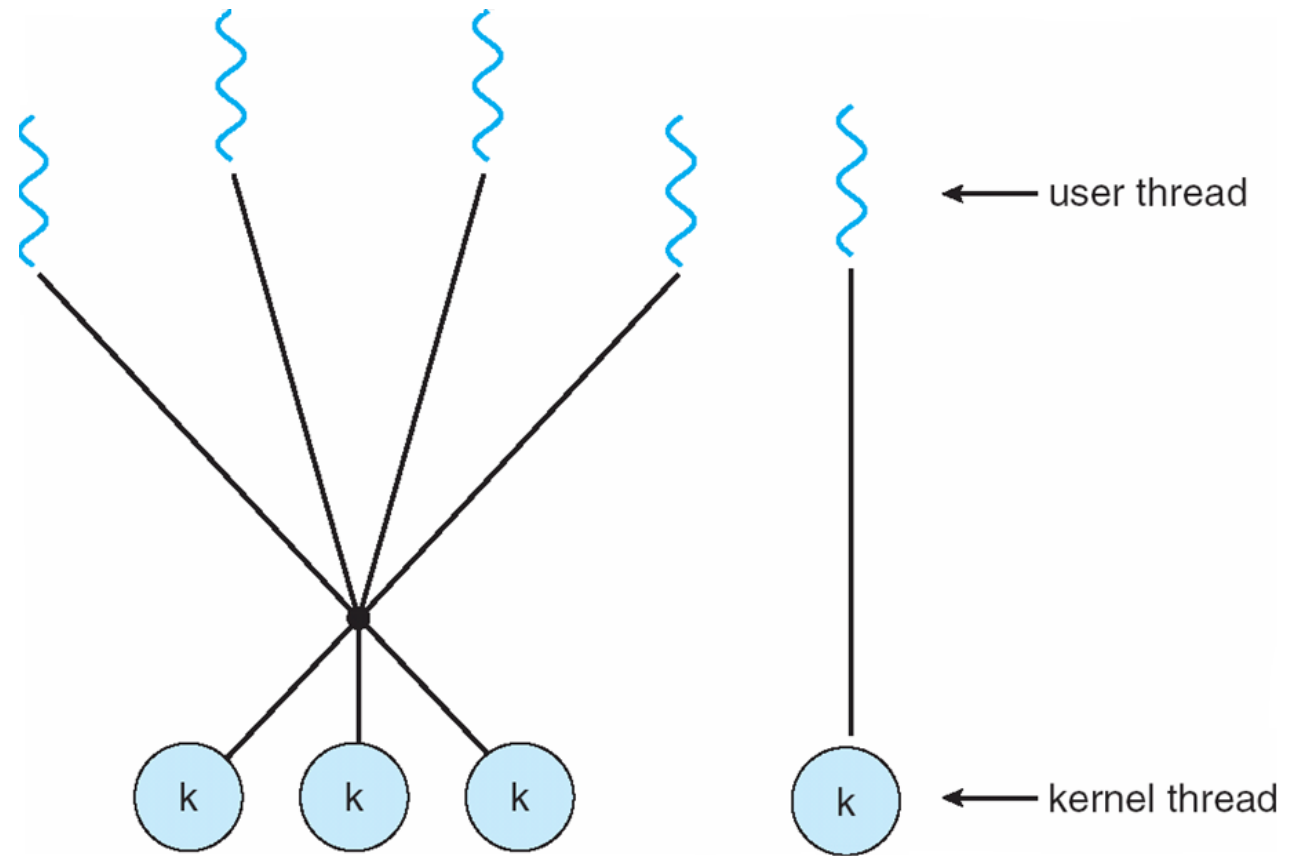
Scheduler activations (1)

Both Many-To-Many and Two-level models require communication from the kernel to the user-level thread manager to maintain an appropriate number of kernel threads allocated to the application.

Many-To-Many



Two-Level

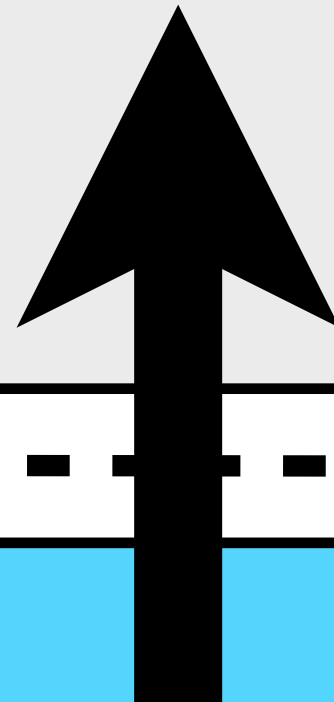


Scheduler activations (2)

A scheduler activation notifies the user-level thread manager of kernel changes.

User space

Scheduler activations allows the user-level thread manager to make better decisions.



Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the user-level thread library.

Kernel space

Scheduler activations (3)

Both Many-To-Many and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.

Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library.

- ★ Such coordination **allows the number of kernel threads to be dynamically adjusted** to help ensure the best performance.
- ★ This communication allows an application to maintain the correct number of kernel threads

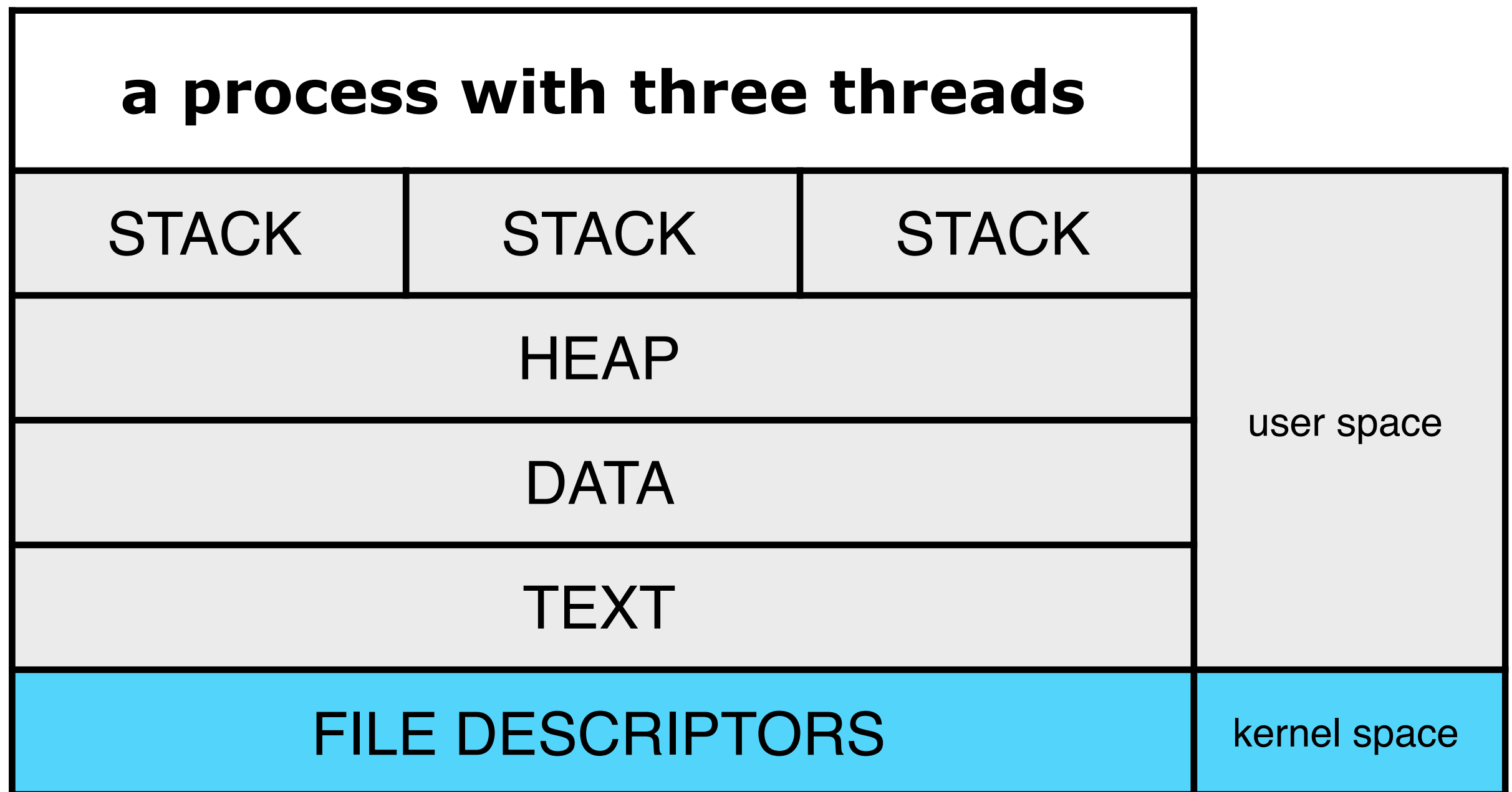
A scheduler activation notifies the user-level thread system of kernel changes:

- ★ Number of processors assigned
- ★ I/O interrupts
- ★ Page faults

Thread resource management

(1)

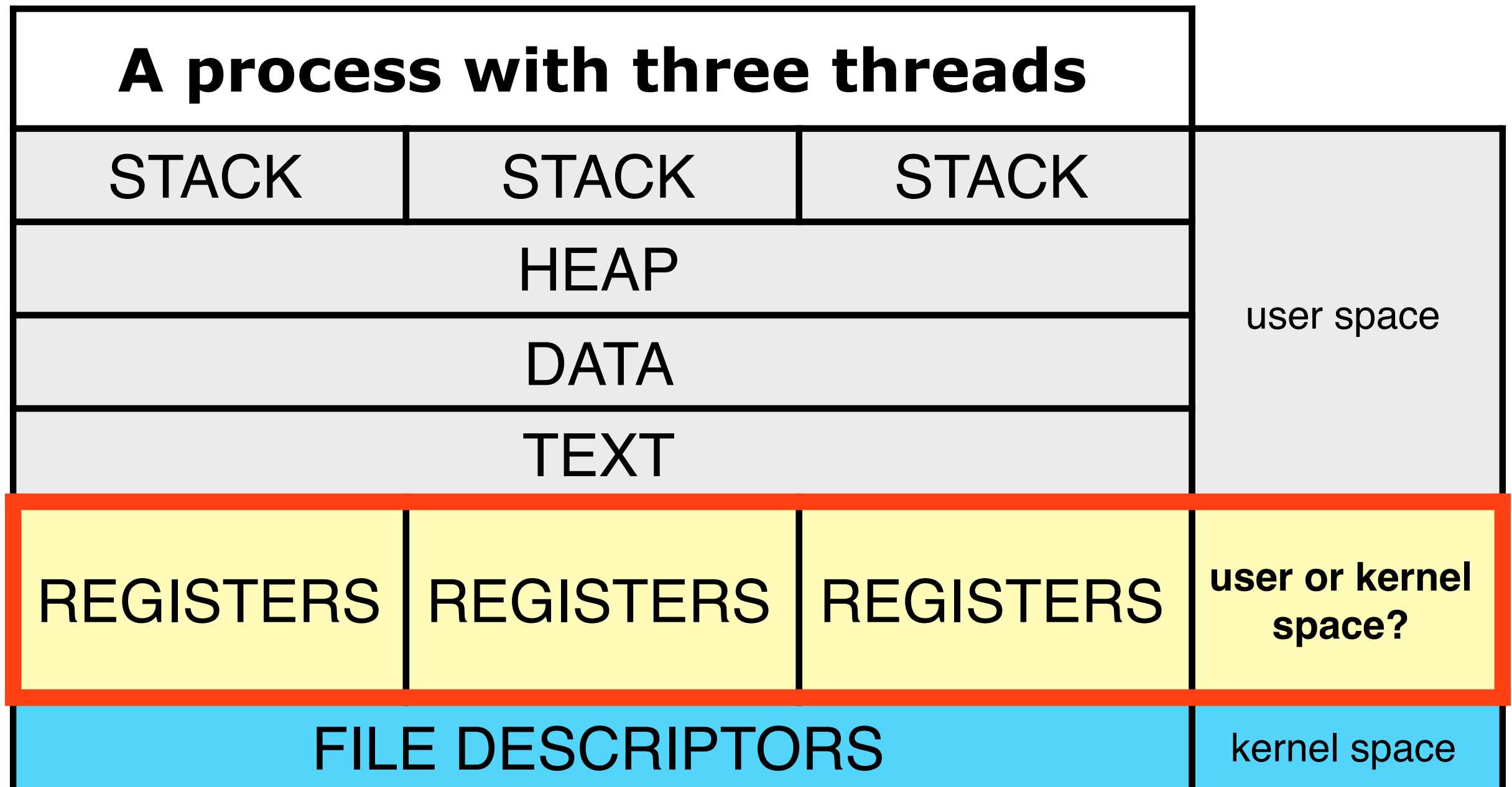
Threads share user space resources (heap, data and text). They also share kernel space resources (file descriptors). Each thread also have a private user level stack.



Thread resource management

(2)

Each thread also needs private storage for registers (context).



User-level thread scheduling

User space

Scheduling of user level thread among the available kernel-level threads done by user-level scheduler.

If scheduling of threads is done in user level, context (registers) must must be saved in user space!

many-to-one

one-to-one

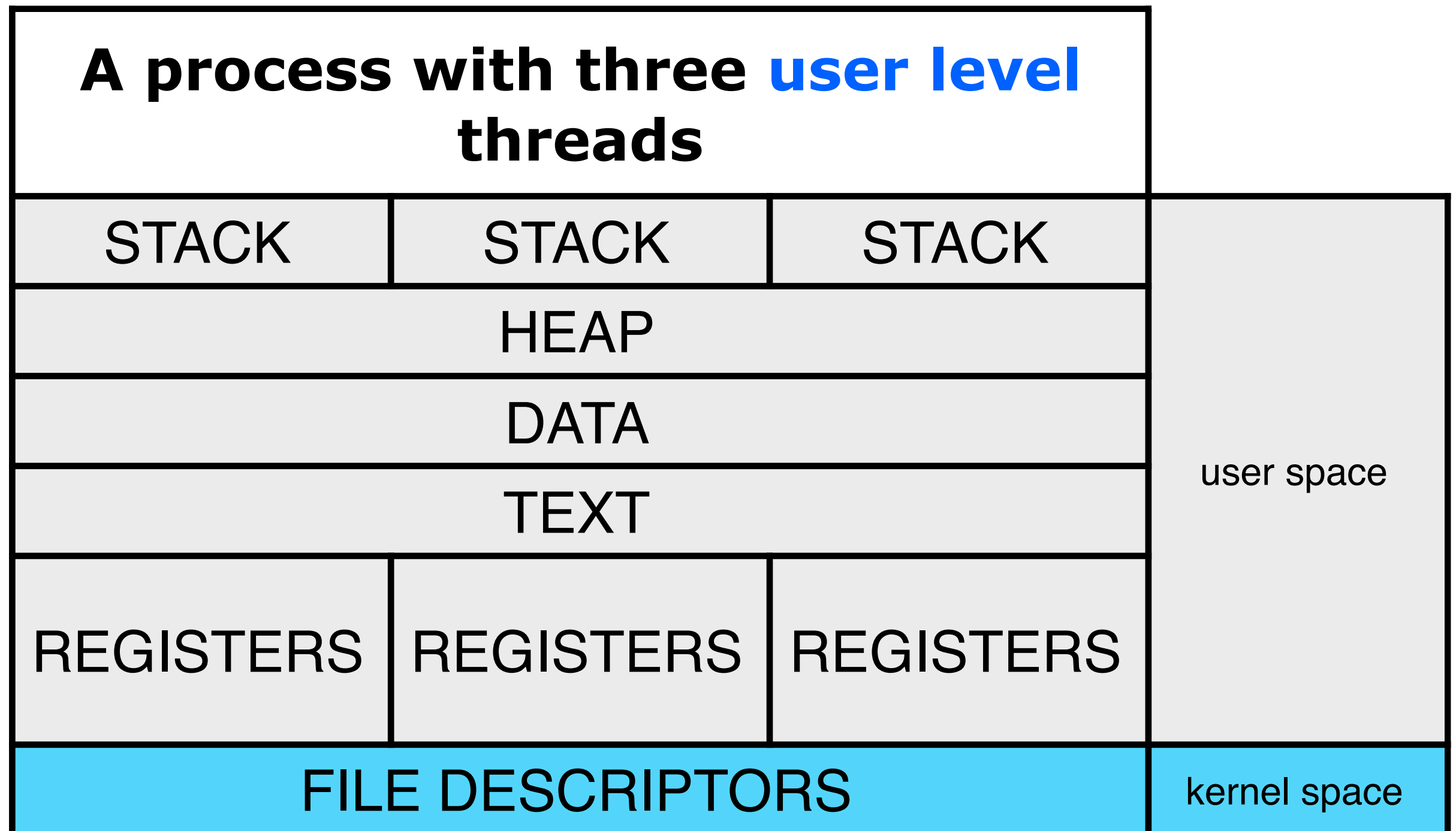
many-to-many

two-level

Kernel space

User level thread context

If scheduling of threads is done in user level, context (registers) must must be saved in user space!

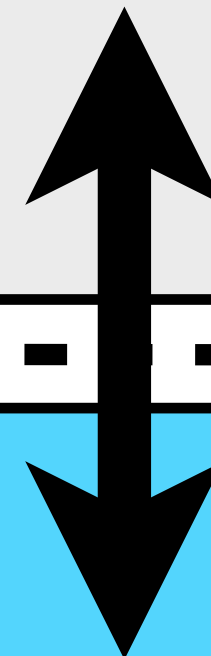
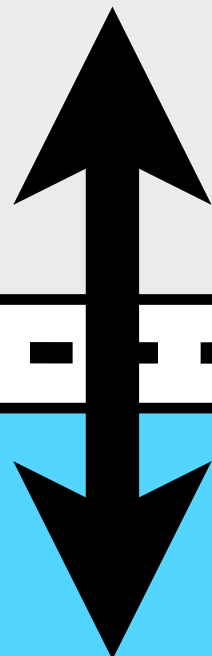


User level thread context

A **machine-specific** representation of the saved **context** (the execution state) including all **registers** and **CPU flags**, the **instruction pointer**, and the **stack pointer**.

Need a **system call** to ask the OS to **save the context** of the currently executing user-level thread **in user space**.

Need a **system call** to ask the OS to **switch context**, i.e., reload a context (saved in user space) and **resume execution** within in the reloaded context.



User space

Kernel space

ucontext_t

The <ucontext.h> header defines the ucontext_t type as a structure that can be used to store the execution context of a user-level thread to user space.

Members of the ucontext_t structure

Type	Name	Description
mcontext_t	uc_mcontext	A machine-specific representation of the saved context (the execution state) including all registers and CPU flags , the instruction pointer , and the stack pointer .
stack_t	uc_stack	The stack used by this context.
sigset_t	uc_sigmask	The set of signals that are blocked when this context is active.
ucontext_t	*uc_link	Pointer to the context that will be resumed when this context returns.

```
int getcontext(ucontext_t *ucp)
```

- ★ The `getcontext()` function shall initialize the structure pointed to by `ucp` to the current user context of the calling thread.
- ★ The `ucontext_t` type that `ucp` points to defines the user context and includes the contents of the calling thread's machine registers, the signal mask, and the current execution stack.
- ★ Upon successful completion, `getcontext()` shall return 0; otherwise, a value of -1 shall be returned.

```
void makecontext(  
    ucontext_t *ucp,  
    void (*func)(),  
    int argc, ...)
```

- ★ Modifies the context specified by **ucp**, which has been initialized using `getcontext()`.
- ★ When this context is **resumed** using `swapcontext()` or `setcontext()`, program execution shall continue **by calling func**, passing it the arguments that follow **argc** in the `makecontext()` call.
- ★ Before a call is made to `makecontext()`, the application shall ensure that the context being modified has a stack allocated for it.
- ★ The application shall ensure that the value of **argc** matches the number of arguments of type `int` passed to **func**; otherwise, the behavior is undefined.


```
int swapcontext(  
    ucontext_t *restrict oucp,  
    const ucontext_t *restrict ucp)
```

- ★ The swapcontext() function shall save the current context in the context structure pointed to by **oucp** and shall set the context to the context structure pointed to by **ucp**.
- ★ When successful, swapcontext() does not return. (But we may return later, in case oucp is activated, in which case it looks like swapcontext() returns 0.) On error, swapcontext() returns -1 and sets errno appropriately.

Source: <http://pubs.opengroup.org/onlinepubs/009695399/functions/makecontext.html>
<http://linux.die.net/man/3/swapcontext>

2016-02-08
2016-02-08

In the C programming language, as of the C99 standard, restrict is a keyword that can be used in pointer declarations. The restrict keyword is a declaration of intent given by the programmer to the compiler. It says that for the lifetime of the pointer, only it or a value directly derived from it (such as pointer + 1) will be used to access the object to which it points. This limits the effects of pointer aliasing, aiding optimizations. If the declaration of intent is not followed and the object is accessed by an independent pointer, this will result in undefined behavior

Source: <https://en.wikipedia.org/wiki/Restrict>

2016-02-08

```
int setcontext(const ucontext_t *ucp)
```

- ★ The setcontext() function shall restore the user context pointed to by **ucp**.
- ★ A successful call to setcontext() shall not return; program execution resumes at the point specified by the ucp argument passed to setcontext().
- ★ The ucp argument should be created either by a prior call to getcontext() or makecontext(), or by being passed as an argument to a signal handler.
- ★ Upon successful completion, setcontext() shall not return; otherwise, a value of -1 shall be returned.