Threads

Module 4

Lecture

Operating systems 2019

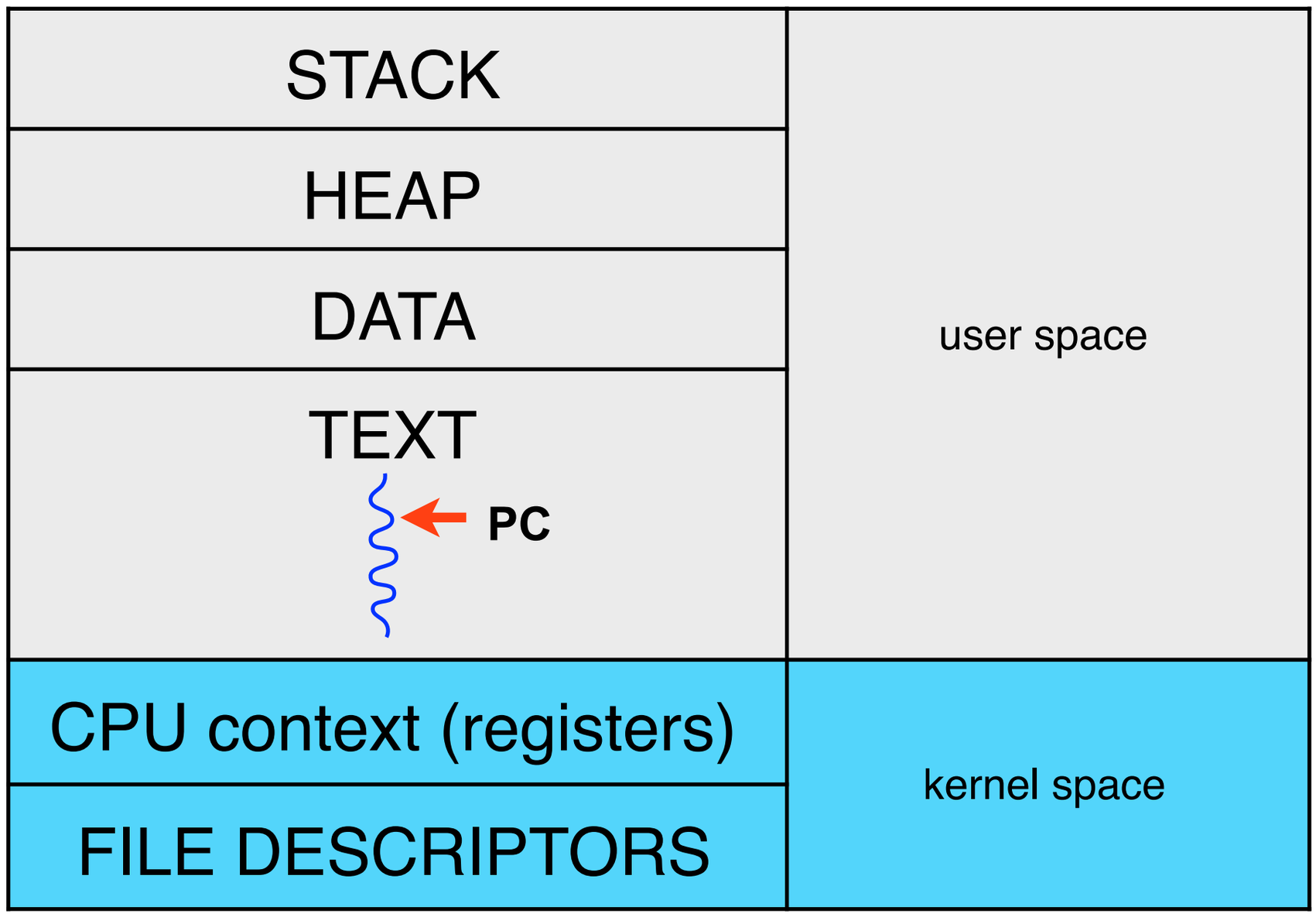
1DT044, 1DT096 and 1DT003

February 2019 karl.marklund@it.uu.se Uppsala university

Background

A process

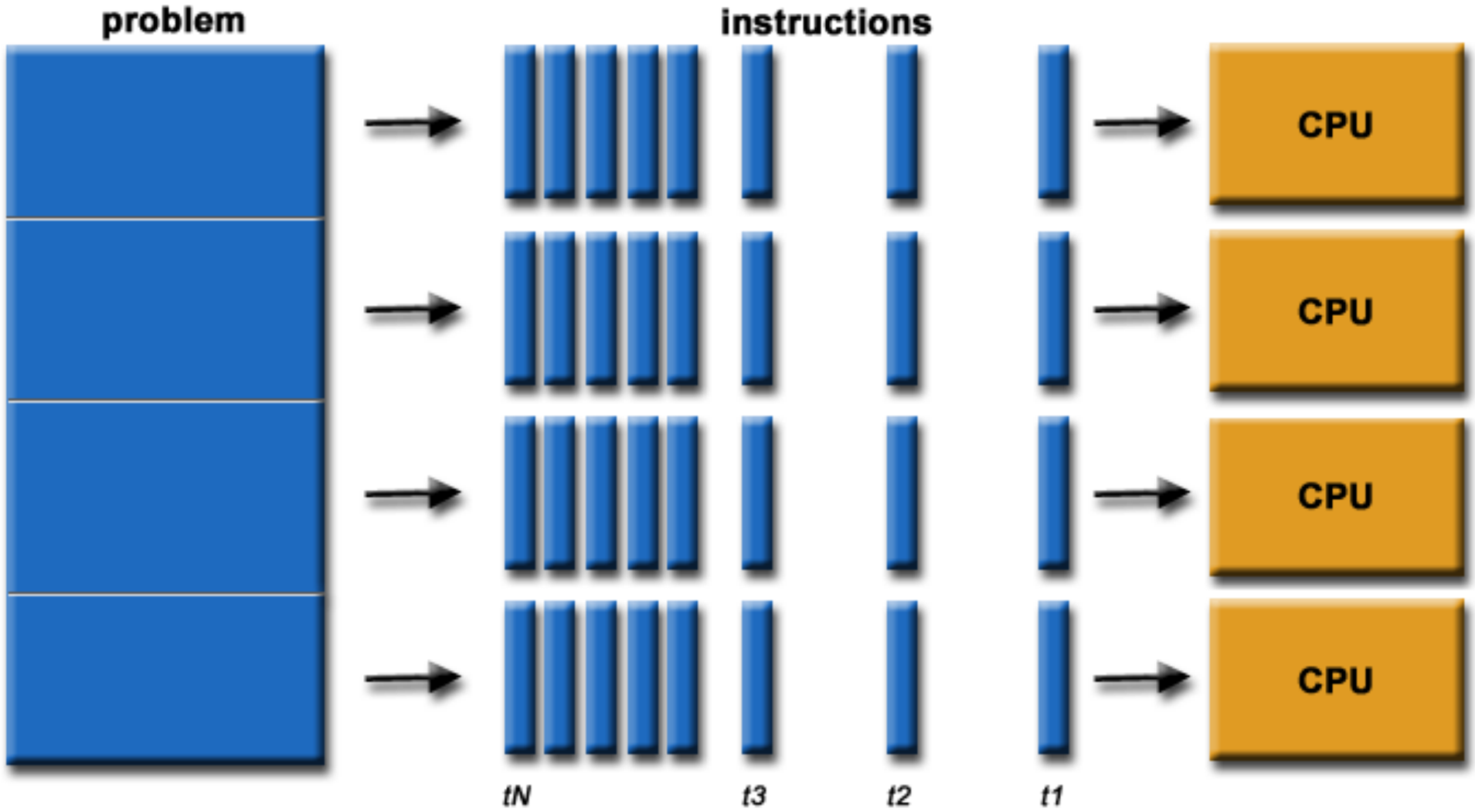
A process as we know it has a single program counter (PC), i.e, a single thread of control.



Parallel computing

A type of computation in which many **calculations** are carried out **simultaneously**.

Large problems can often be divided into smaller ones, which are then solved in **parallel** (at the same time) on multiple physical CPUs.



This is not the same as multithreading. Multithreading can be done on a single core CPU. In such a case, two threads can never execute at the same time on the CPU.

Concurrency

The ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.

This allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems.

Source: https://en.wikipedia.org/wiki/Concurrency_(computer_science) 2018-02-08

Concurrency

≠

Parallelism

Concurrency is often referred to as the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units.

Source: Lamport, Leslie (July 1978). "Time, Clocks, and the Ordering of Events in a Distributed System"

Parallelism ≠ Concurrency

In parallel systems, two tasks are **actually** performed **simultaneously**.

Parallelism is when tasks **literally** run at the **same time**, eg. on physical separate cores in a multicore processor.

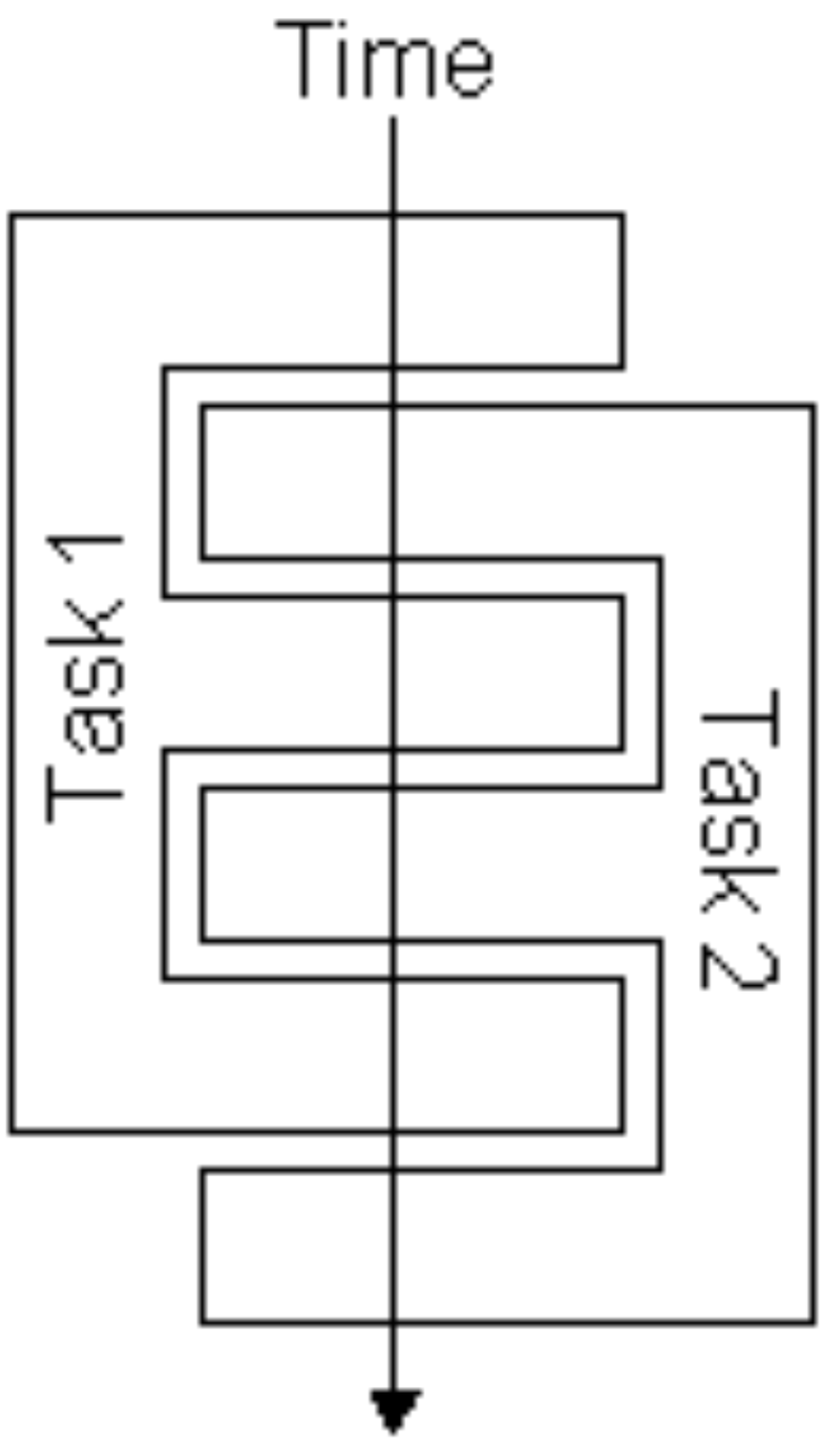
Two tasks are concurrent if the **order** in which the two tasks are executed in time is **not predetermined**.

The **appearance** of several tasks executing at once.

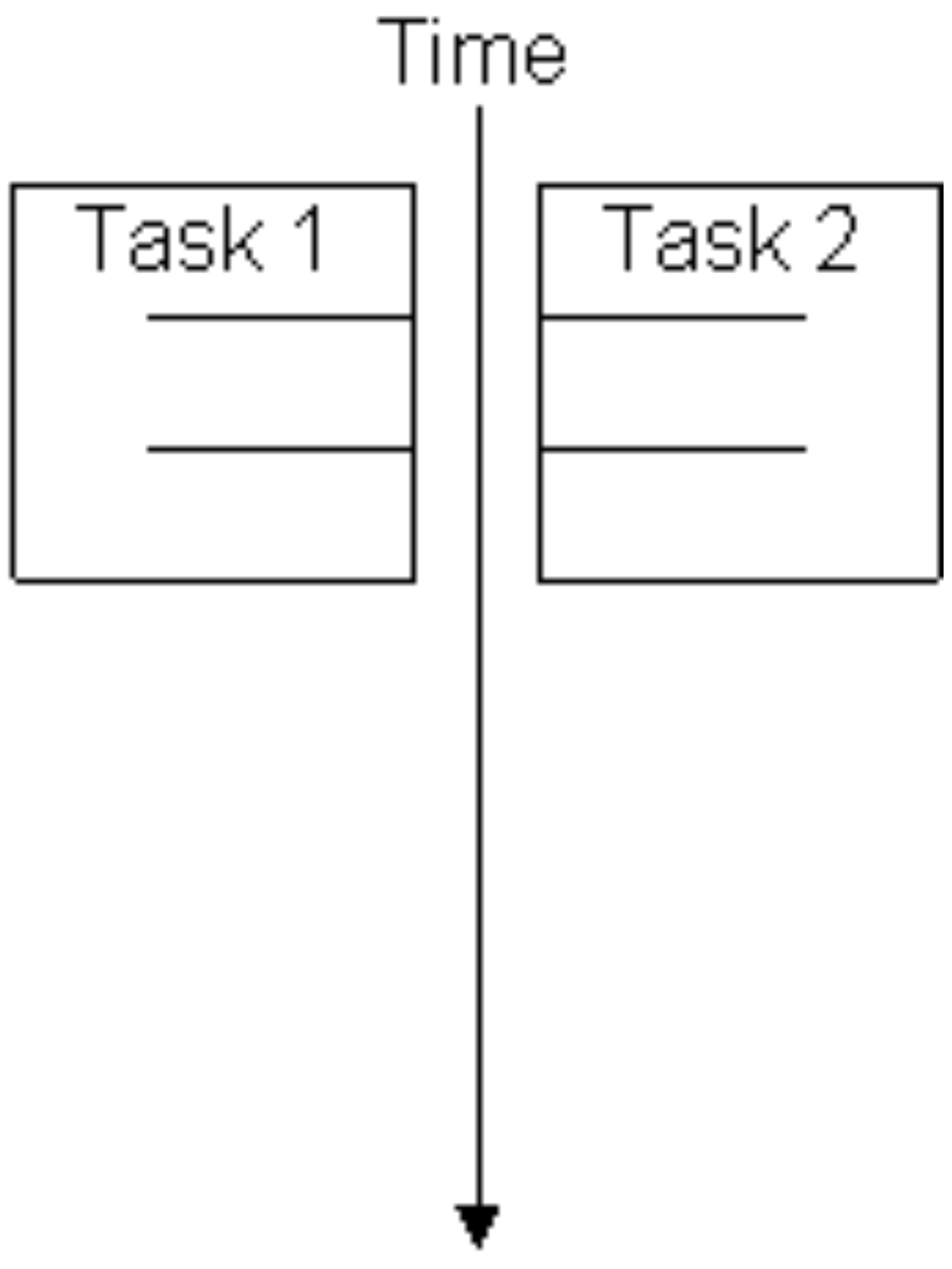
Tasks are actually split up into chunks that share the processor with chunks from other tasks by interleaving the execution in a time-slicing way.

Tasks can start, run, and complete in **overlapping** time periods.

Concurrency



Parallelism



Concurrent programming

- Programming **languages** and **algorithms** used to implement concurrent systems.
- Usually considered to be **more general** than parallel programming.
- Can involve **arbitrary** and **dynamic** patterns of **communication** and **interaction**,
  - In contrast, parallel systems generally have a predefined and well-structured communications pattern.

Source: https://en.wikipedia.org/wiki/Concurrency_(computer_science) 2016-02-06

OS, OSPP and DSP

In the OS (1DT044), OSPP (1DT096) and DSP (1DT003) courses we will focus on concurrent programming.

Other courses

Parallel programming course:

- ★ Low-level parallel programming (1DL550)

Parallel and concurrent programming courses:

- ★ Introduction to parallel programming (1DL530)
- ★ Parallel programming for efficiency (1DL560)
- ★ Testing concurrent and parallel software (1DL570)
- ★ Project in concurrent and parallel programming (1DL580)

Motivating examples

Untitled 1

Default Style | Liberation Serif | 12

Page 1 of 1 | 0 words, 0 characters | Default Style | English (USA) | 100%

A word processor may have to perform the following tasks *”at the same time”*:

- Displaying graphics (GUI).
- Responding to key strokes.
- Spelling and grammar checking in the background.

How can this be accomplished?

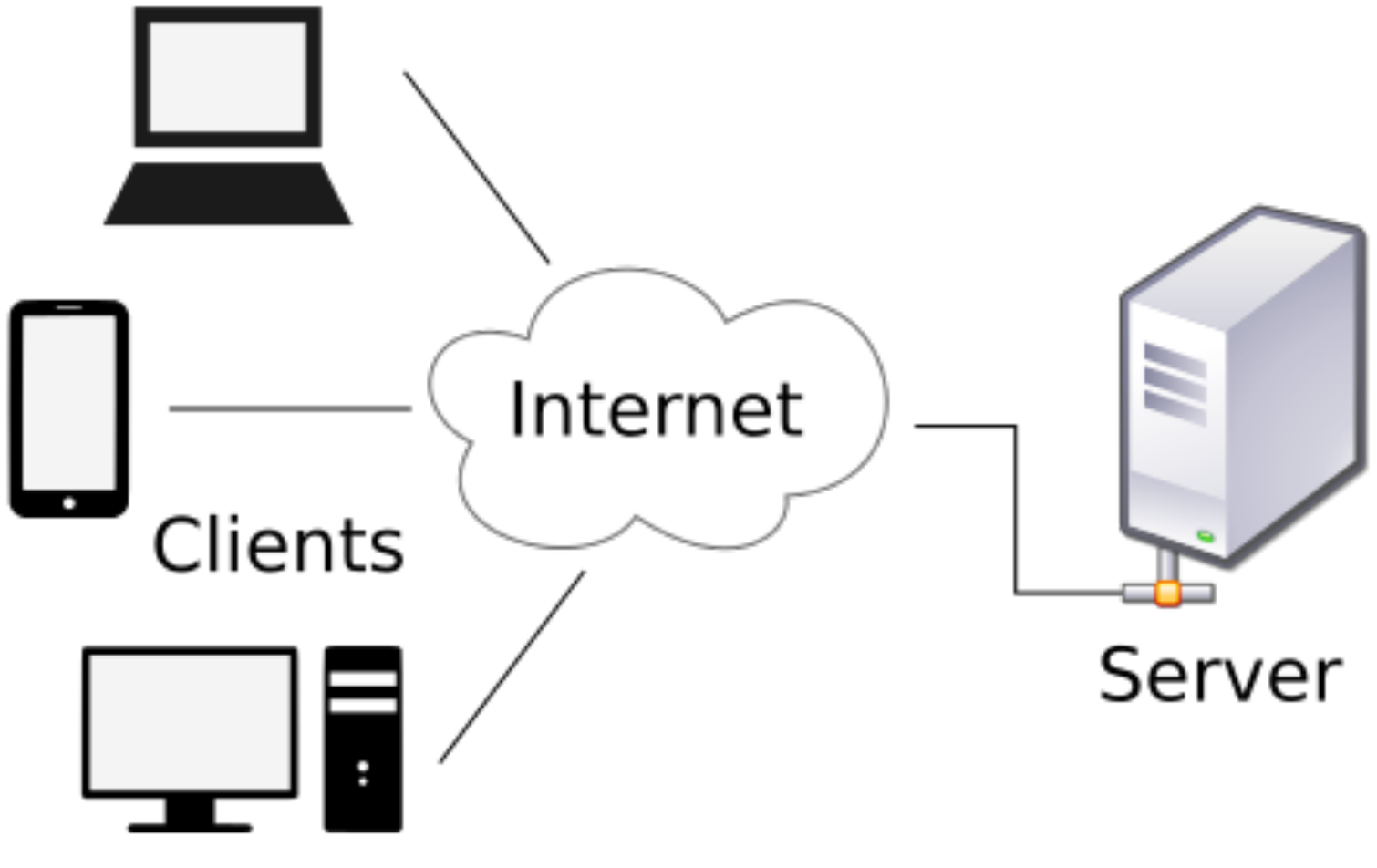
- Multiple processes?

Remark 1

Cannot easily use multiple processes since processes don’t share data.

Client — Server model

A distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.



- Often clients and servers communicate over a computer network on separate hardware …
- … but both client and server may reside in the same system.
- A **server** host runs one or more server programs which **share** their **resources with clients**.
- **A client does not share any of its resources**, but requests a server's content or service function.
- Clients therefore initiate communication sessions with servers which await incoming requests.

Source: https://en.wikipedia.org/wiki/Client-server_model 2016-02-06

Concurrency and response time

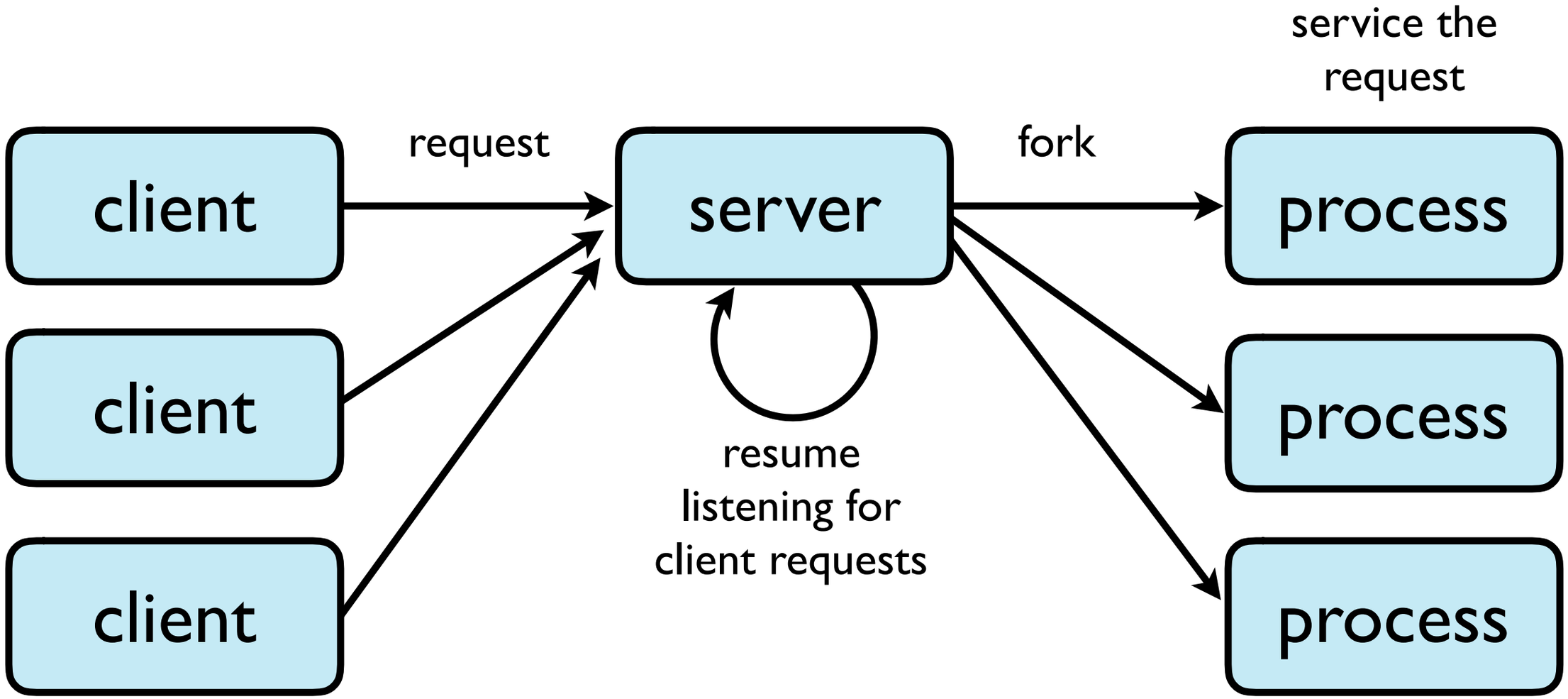
A server can exploit concurrency for increasing its response time.

Server

Needs to listen for client requests.

Needs to service each request.

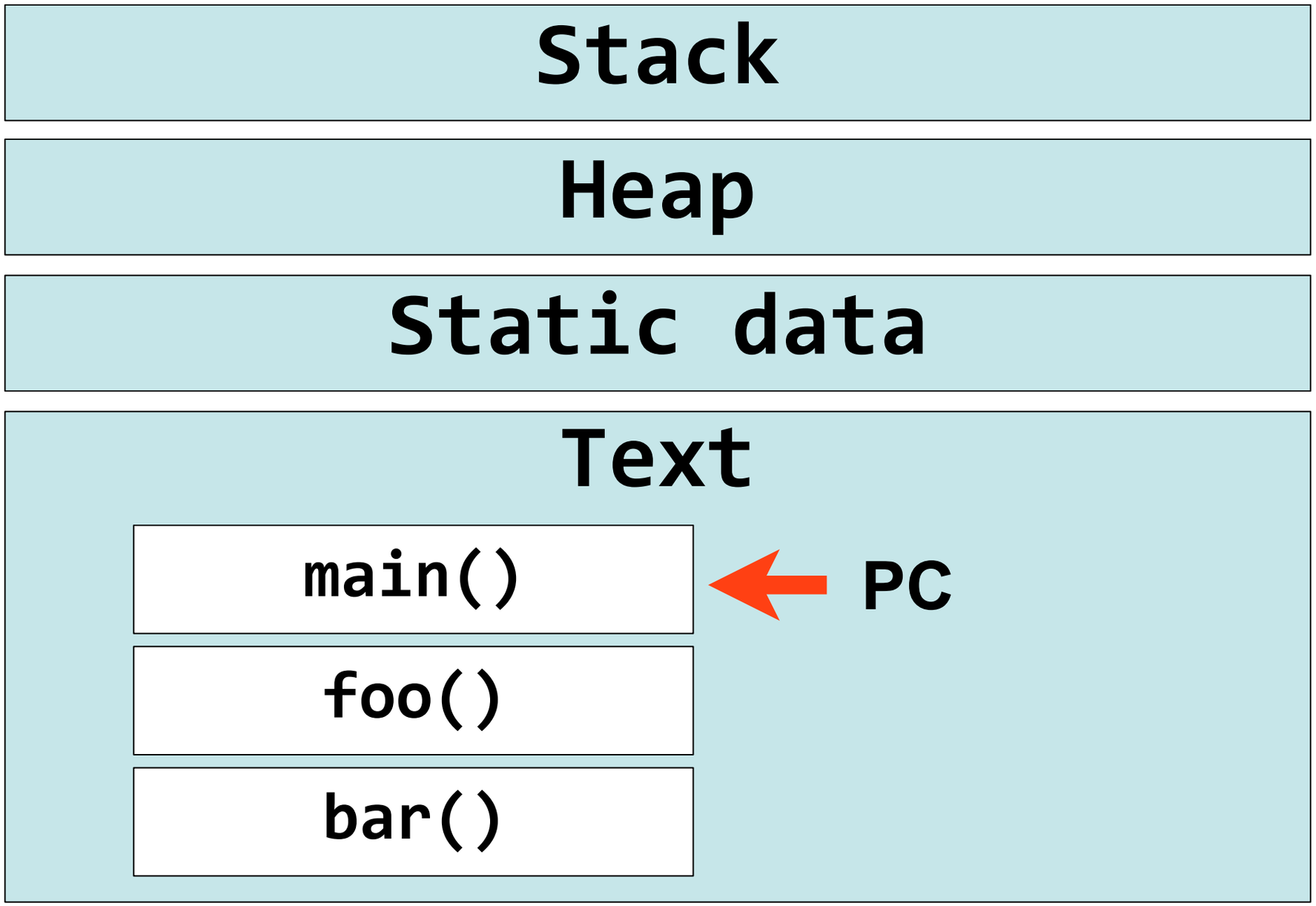
If these activities can be executed **concurrently**, the server may resume listening for new requests while previous requests are served (handled).



Remark 2

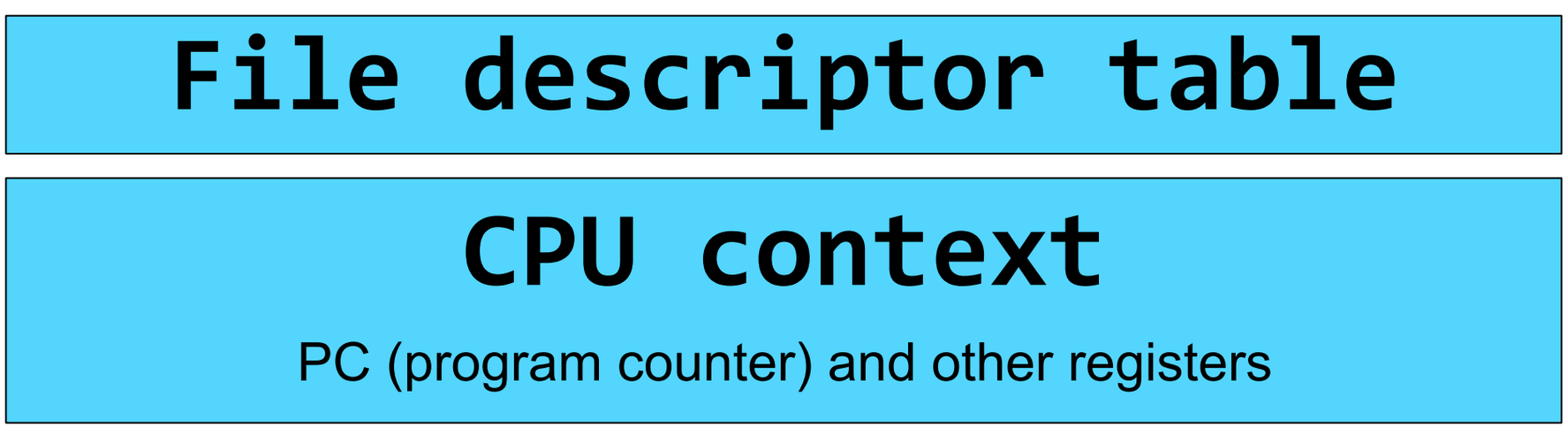
Creating a new process is time consuming and resource intensive.

The process



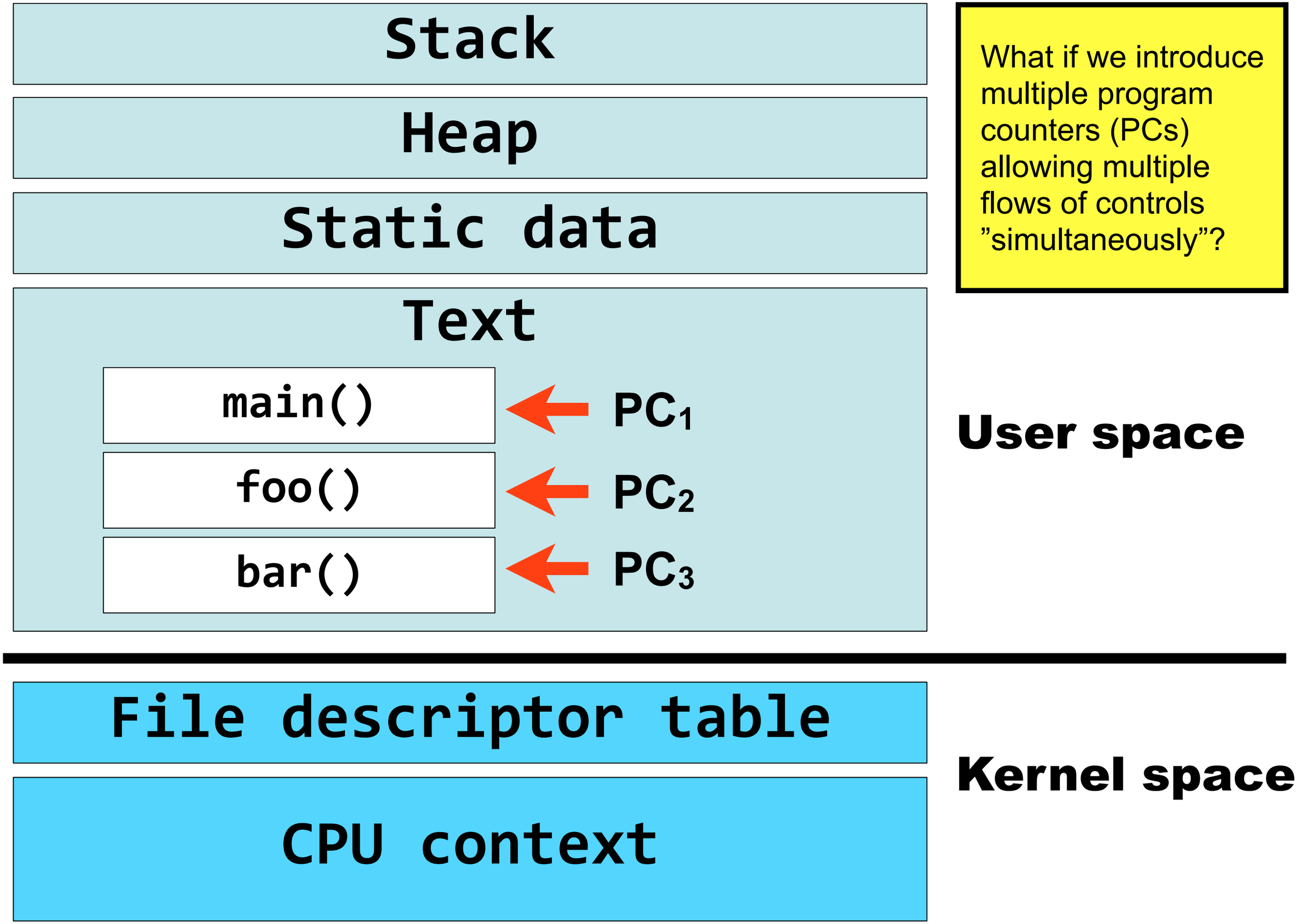
A process as we know it has a single program counter (PC), i.e, a single thread of control.

User space



Kernel space

What if ...


Stack
Heap
Static data
Text
main()
foo()
bar()
PC1
PC2
PC3
What if we introduce multiple program counters (PCs) allowing multiple flows of controls ”simultaneously”?
User space
File descriptor table
CPU context
Kernel space

Stack ?

Heap ?

Static data ?

Text

main() ← PC_1

foo() ← PC_2

bar() ← PC_3

If multiple program counters are used, how does this affect the use of the **stack**, **heap**, **static data**, **file descriptors** and **registers** (CPU context)?

User space

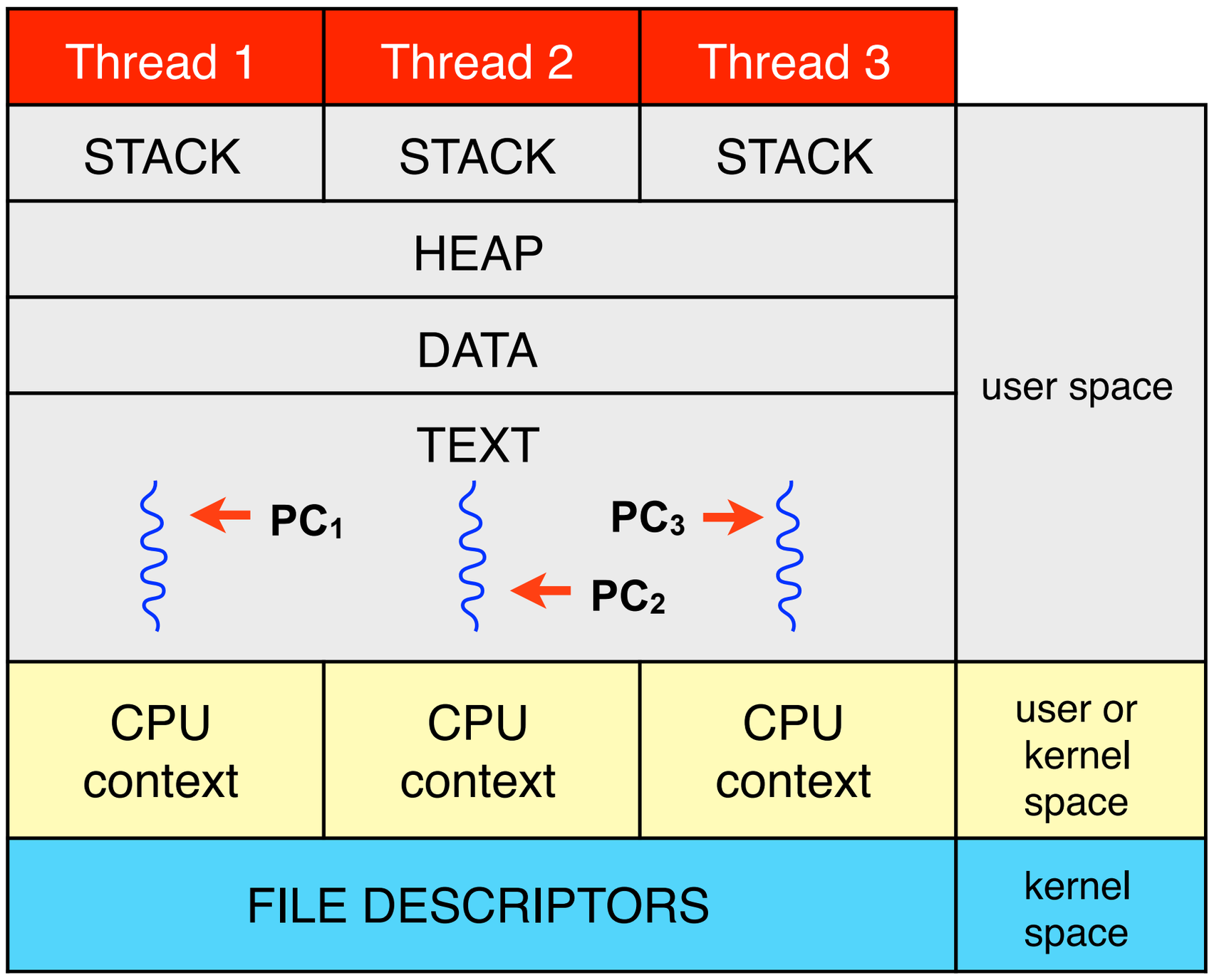
File descriptor table ?

CPU context ?

Kernel space

A process with three threads

Each thread must have a private stack. Depending on how threads are implemented, storage for the CPU context (registers) for each thread can be kept in either user space or kernel space.



Multithreaded processes

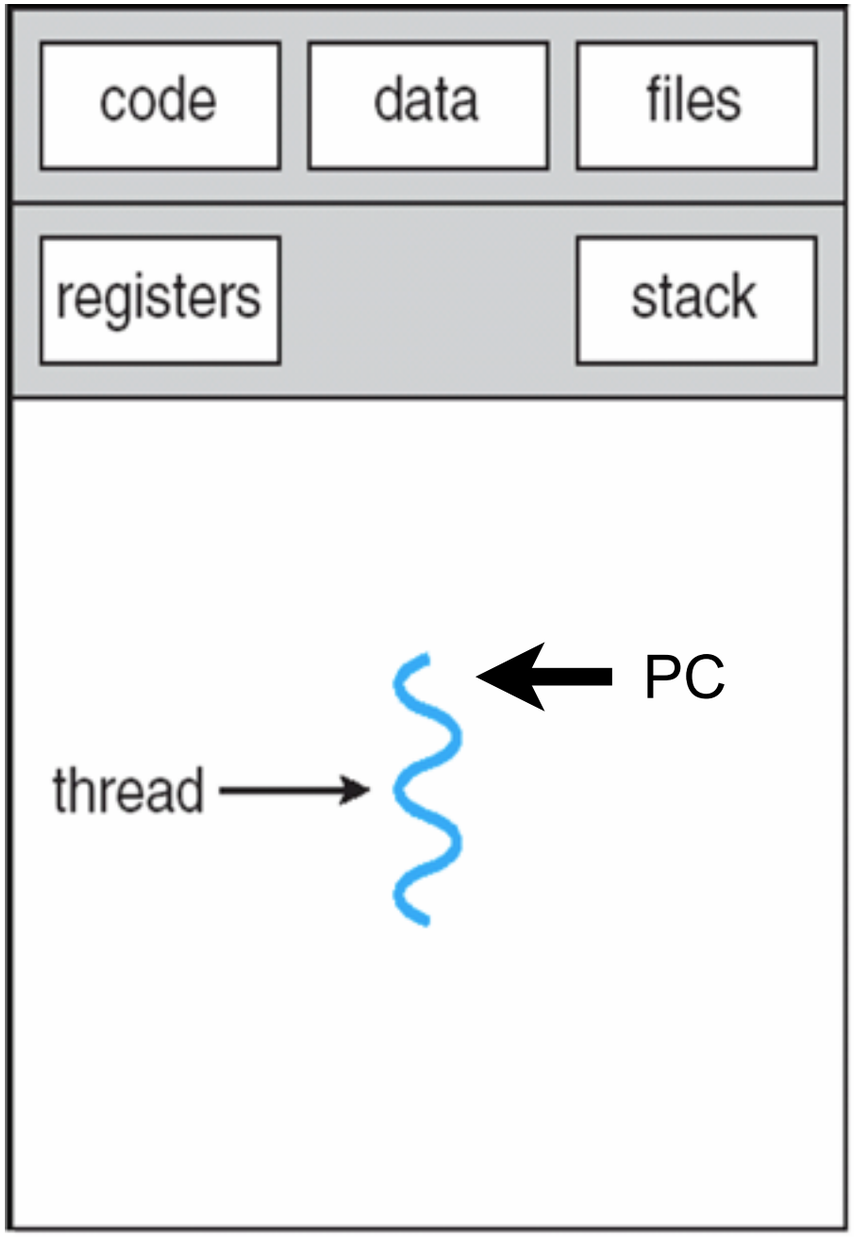
In a multithreaded process there are more than one thread of control.

Each thread uses a separate stack.

The CPU context (register values) of each thread must be saved to, and restored from separate memory locations.

Depending on how threads are implemented, storage for the CPU context for each thread can be kept in either user space or kernel space.

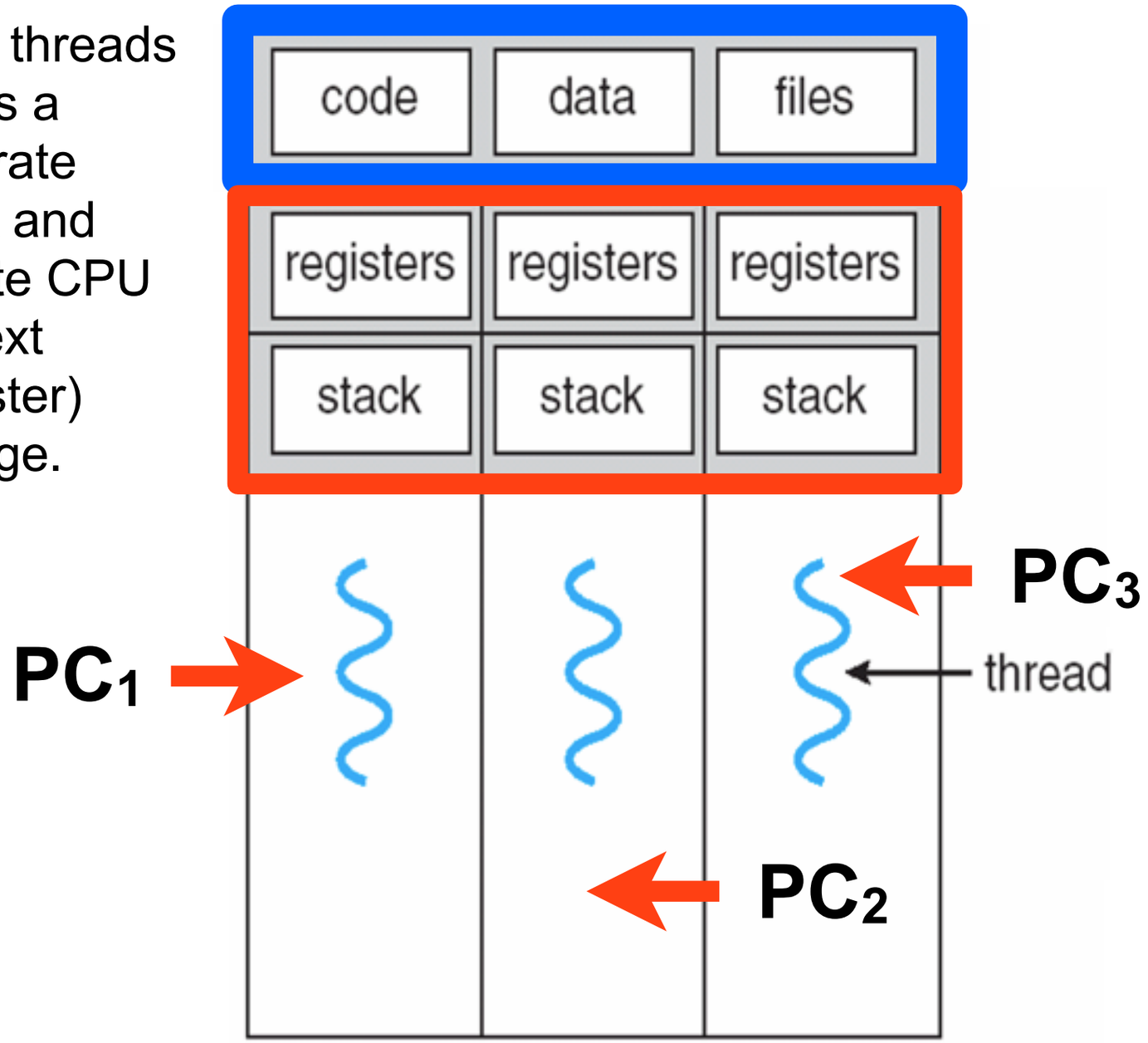
Single and multithreaded processes



single-threaded process

Each threads needs a separate stack and private CPU context (register) storage.

Threads share code, data and open files.

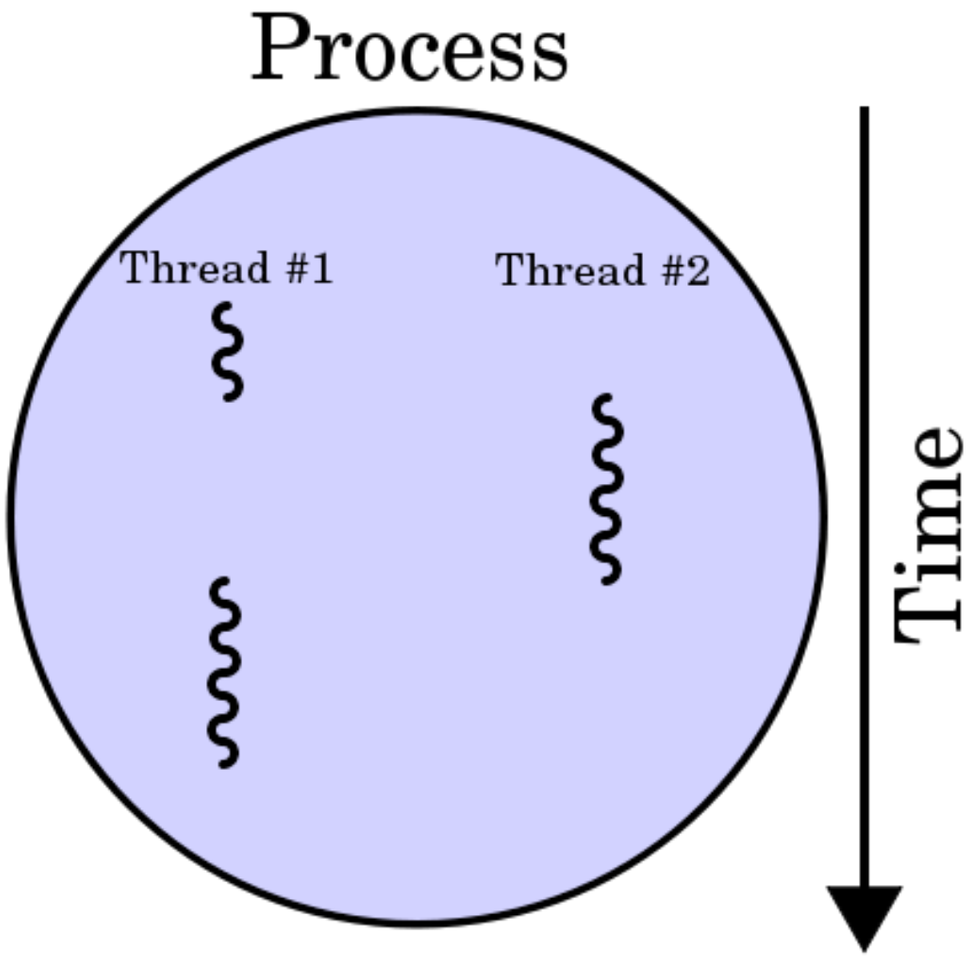


multi-threaded process

Note: this is an example of a process with three threads. Generally, a process may have n > 0 threads.

Thread

A thread of execution is the **smallest sequence of instructions** that can be **managed independently by a scheduler**, which is typically a part of the operating system.

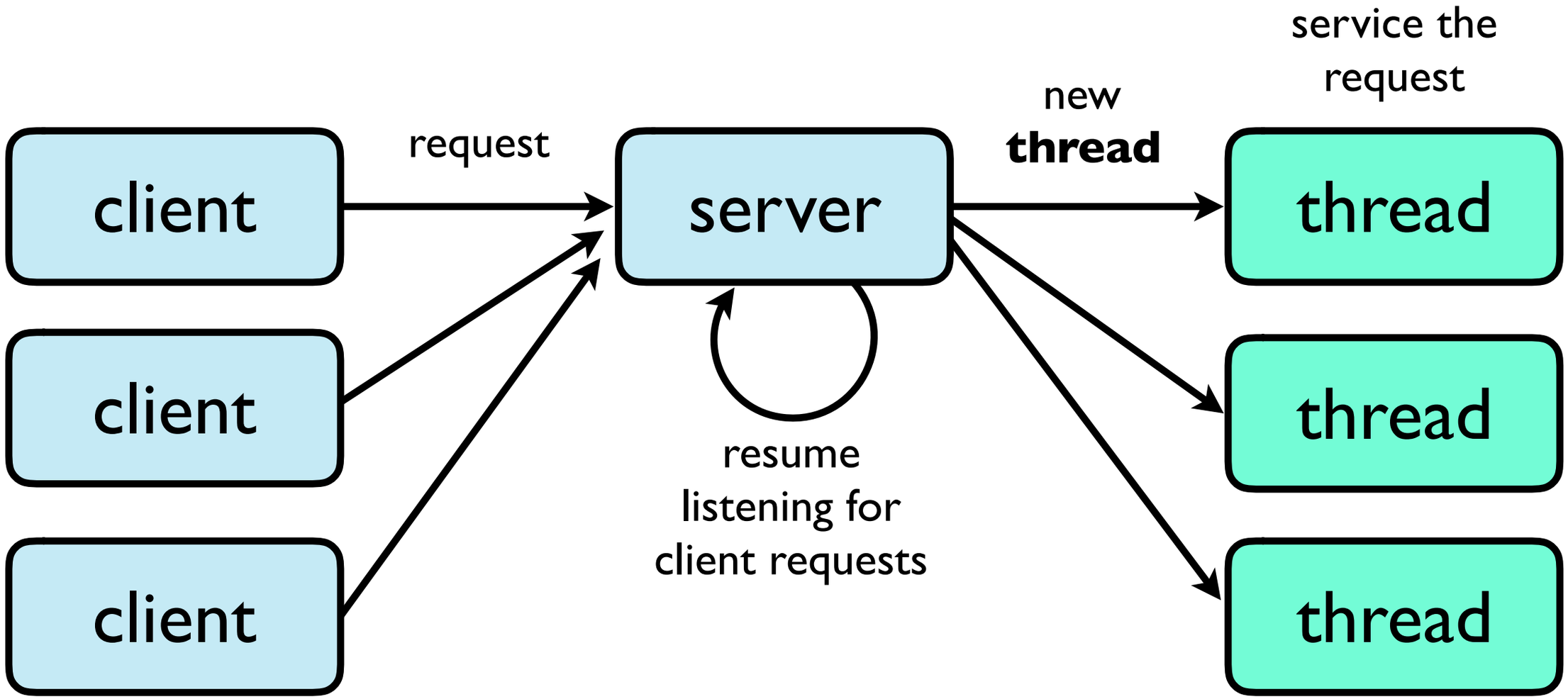


Multiple threads can exist within the same process:

- ★ **Executing concurrently** (one starting before others finish)
- ★ **Sharing** resources such as **memory**, while different processes do not share these resources.
- ★ **Sharing** instructions (executable **code**) and memory (the values of its variables at any given moment).

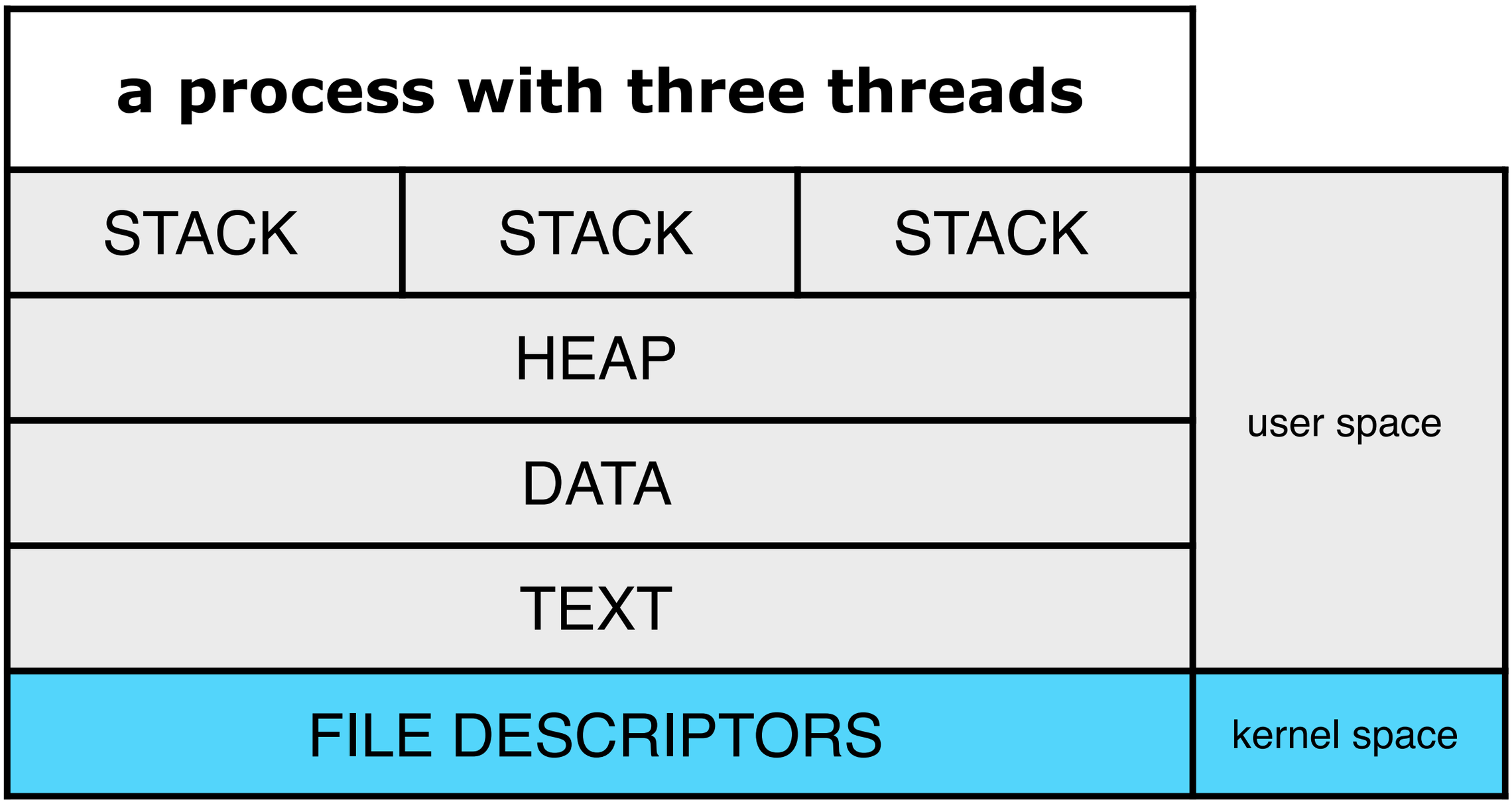
A word processor may have to perform the following tasks *"at the same time"*, **using separate threads** for:

- Displaying graphics (GUI).
- Responding to key strokes.
- Spelling and grammar checking in the background.


client
client
client
request
server
resume
listening for
client requests
new
thread
service the
request
thread
thread
thread

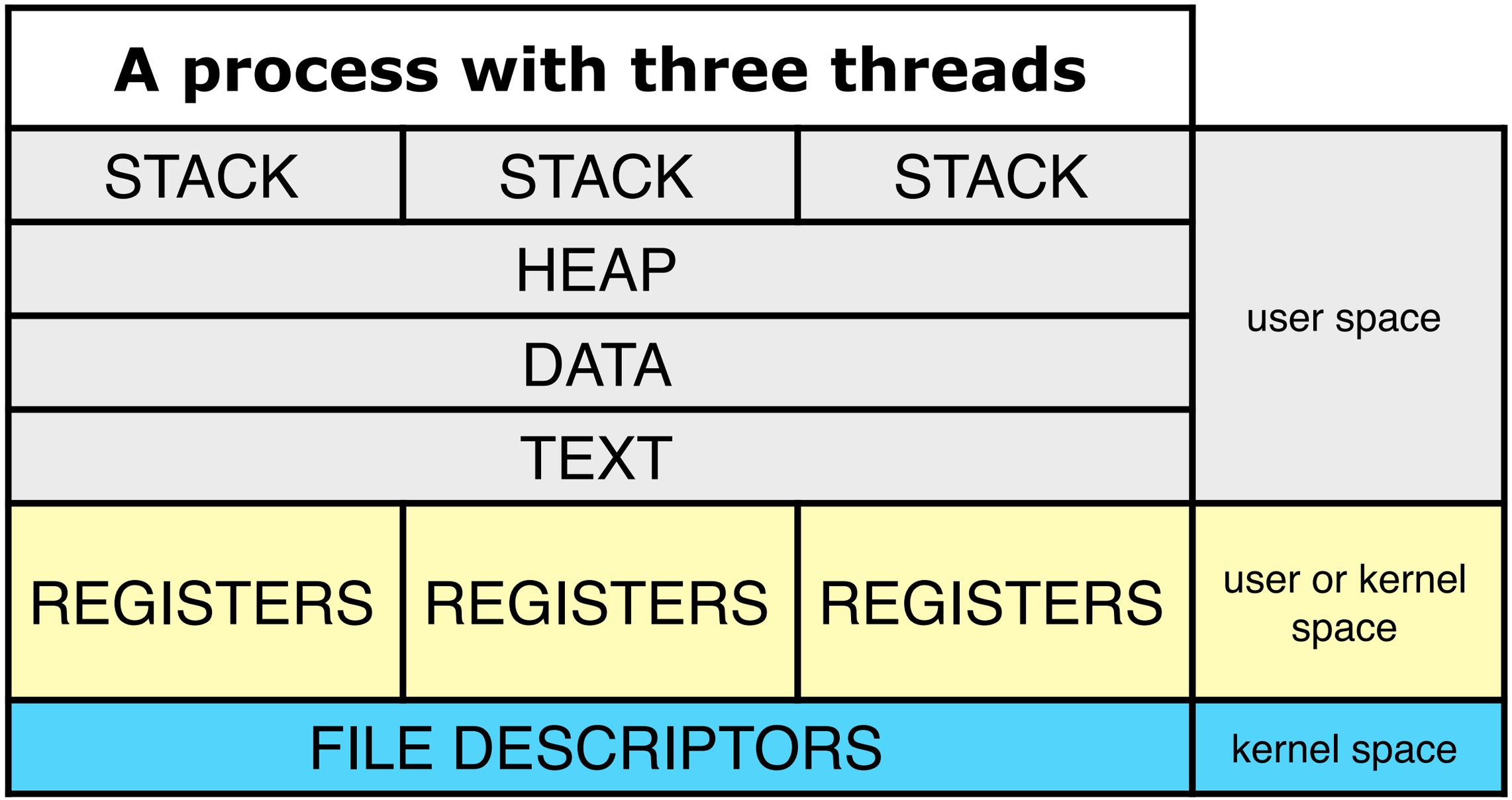
Thread resource management

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.



CPU context (register values)

Depending on how threads are implemented, storage for the CPU context (registers) for each thread can be kept in either user space or kernel space.



Benefits of using threads

Responsiveness

Multithreading an interactive application may allow a program to continue running even if part of it is blocking or performing a lengthy operation, thereby increasing responsiveness to the user. '

Resource sharing

Processes may only share resources through techniques such as shared memory or message passing. Threads (by default) share memory and resources which allows an application to have several threads of execution in the same address space.

Economy

Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is (at least in theory) more economical to create and context-switch threads.

Scalability

The benefits of multithreading can be greatly increased in a multiprocessor (multicore) architecture, where threads may be running in parallel on different processors (cores).

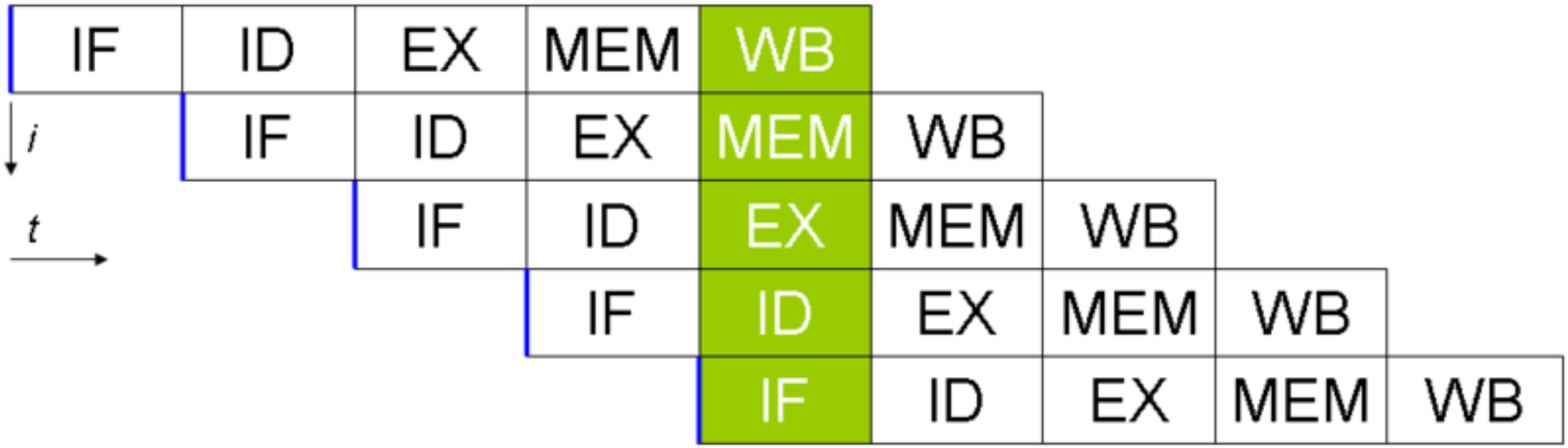
Single-Core CPUs

The core is the part of the processor that actually performs the reading and executing of instructions.

- ★ Processors were originally developed with only one core.
- ★ A **Single-core** processor can process only **one instruction at a time**.

CPU pipeline

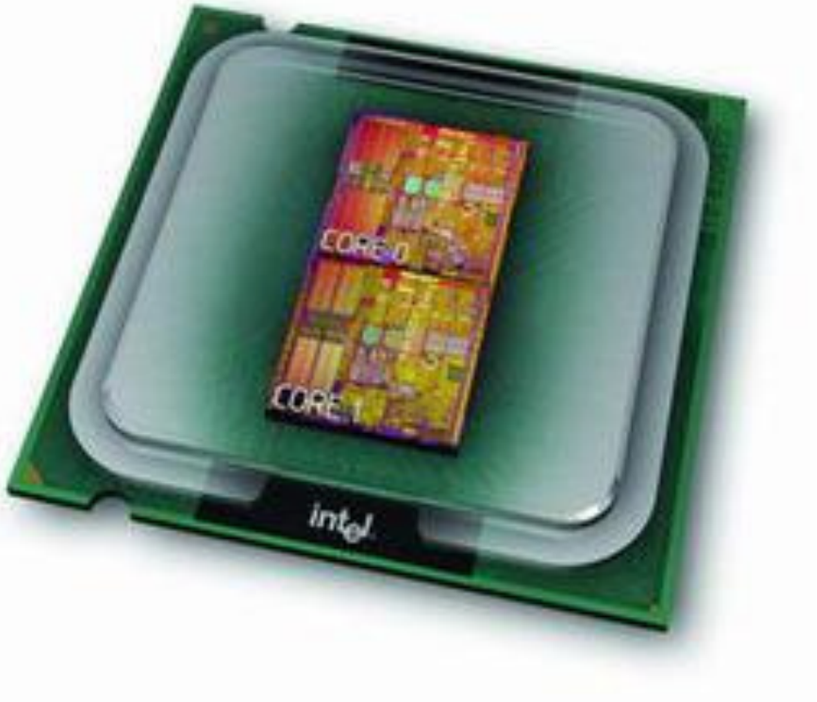
To improve efficiency, processors commonly utilize **pipelines** internally.



★ Allows several instructions to be processed together.

★ Instructions are still consumed into the pipeline one at a time.

Multi-Core CPUs

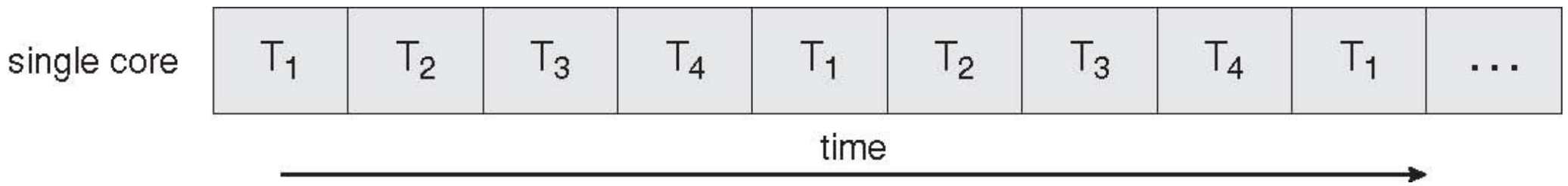


- A multi-core processor is composed of two or more independent cores.
- One can describe it as an integrated circuit which has two or more individual processors (called cores in this sense).

To make full use of multicore computers, programmers will need to learn how to use concurrent programming.

Concurrent execution of threads

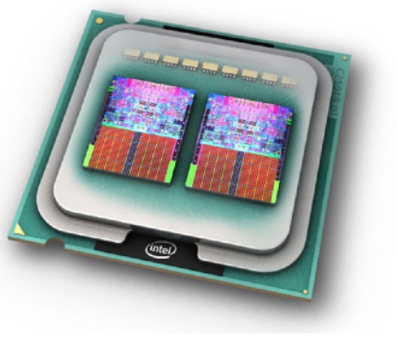
On a single core CPU

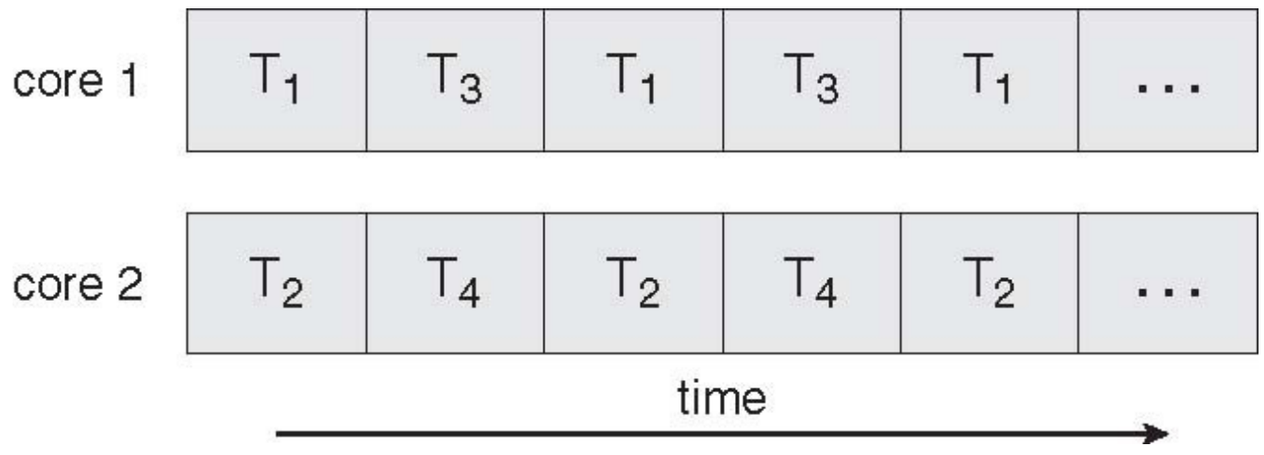


Threads take turn executing on the single CPU core. By switching fast enough between the threads they appear to be executing ”at the same time”.

concurrent ≠ parallel

On a dual core CPU





Remark 3

At most two threads can execute in parallel (truly at the same time) on a dual core CPU. On every core, threads take turn executing just as on a single core CPU.

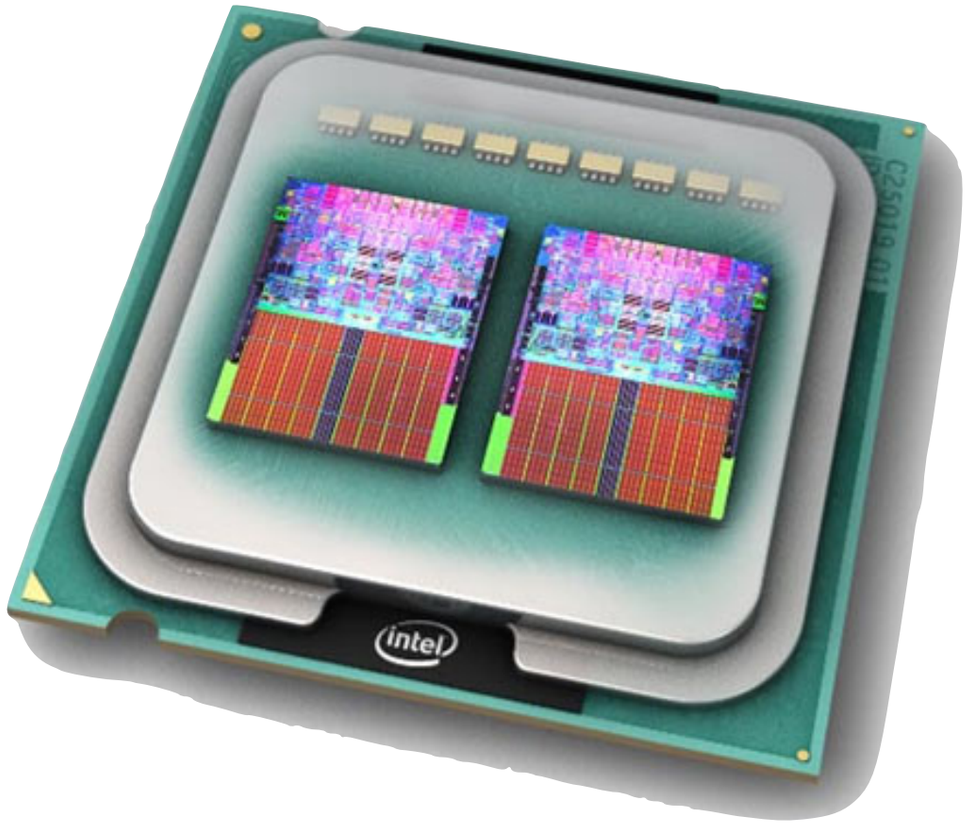
Challenges when using threads

Challenges when using threads

Multi-core give us more computing power - do programmers use threads to utilize this?

Challenges

- ★ Dividing activities
- ★ Balance
- ★ Data splitting
- ★ Data dependency
- ★ Testing and debugging
- ★ Use of shared resources



Intel quad core

Dividing activities

Finding areas in an application that can be divided into separate, concurrent tasks and potentially run in parallel on individual cores.

Balance

If tasks can be found to executed concurrently, programmers also must ensure the tasks to perform equal work of equal value.

Using a separate execution core for a task that don’t contribute much value to the overall process may not be worth the cost.

Data splitting

Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

Data dependency

The data accessed by the tasks must be examined for dependencies between two or more tasks.

In instances where one task depends on data from another, programmers must ensure that the execution of the tasks is **synchronized** to accommodate the data dependency.

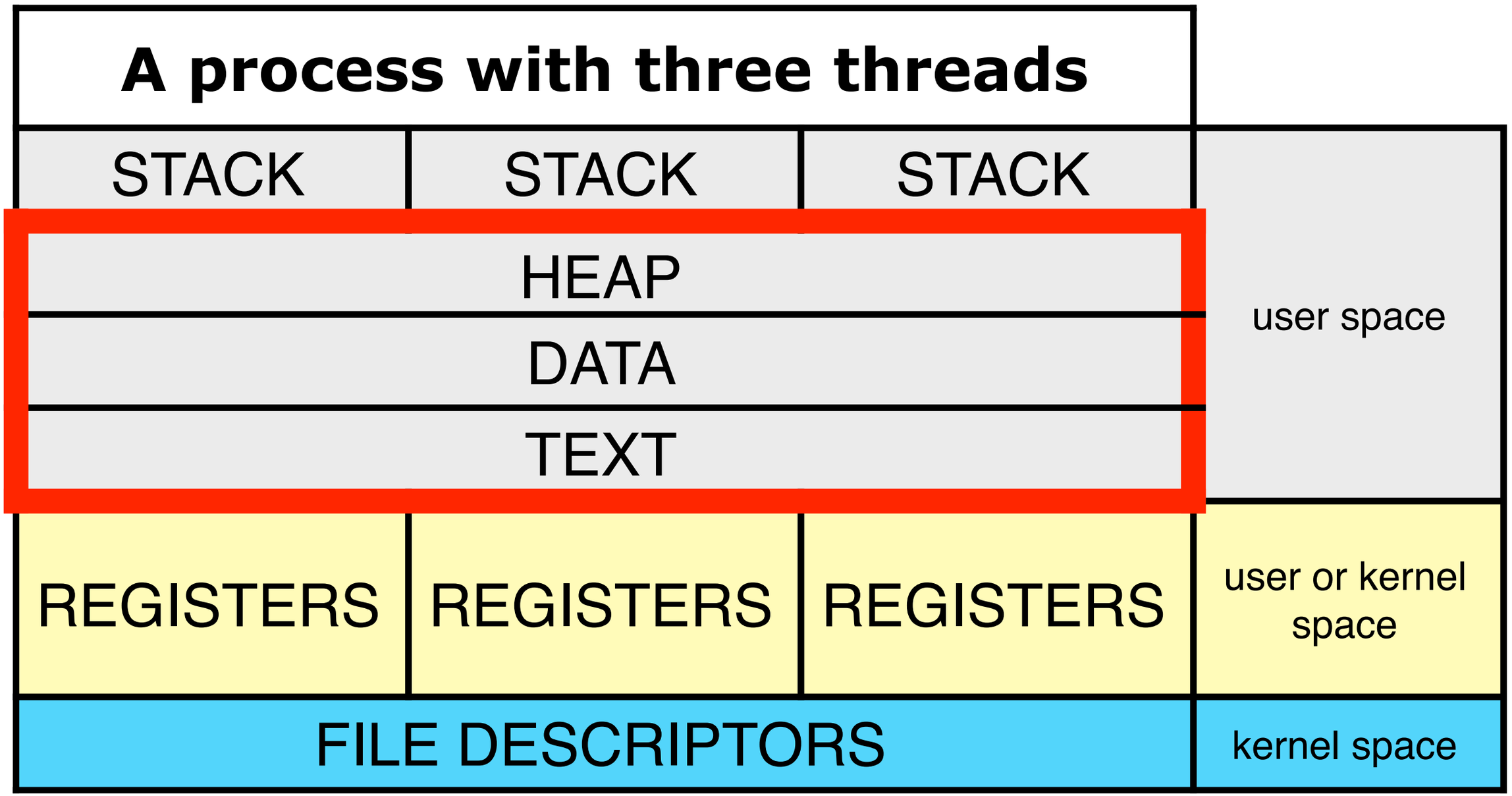
Testing and debugging

When a program is running concurrently on multiple cores, there are many different execution paths.

Testing and debugging such concurrent programs is inherently **more difficult** than testing and debugging single-threaded applications.

Threads share memory

Threads share heap, data and text segments



Bank account example

Concurrent modifications to a shared bank account

shared state

```
#define N 10000

int   BALANCE   0;
```

Thread A

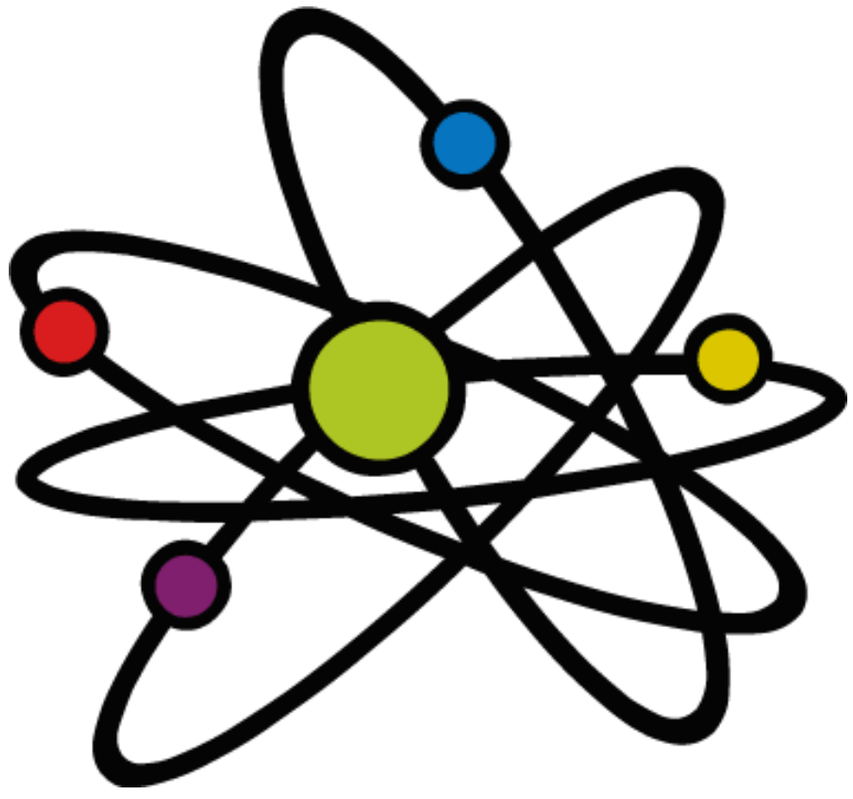
```
for (int i = 0; i < N; i++) {
    BALANCE++;
}
```

Thread B

```
for (int i = 0; i < N; i++) {
    BALANCE--;
}
```

BALANCE == ?

Atomic operations



Atomic operations

- In concurrent programming, an operation (or set of operations) is **atomic** if it appears to the rest of the system to occur at once without being interrupted.

- Other words used synonymously with atomic are: linearizable, indivisible or uninterruptible.

- Additionally, atomic operations commonly have a succeed-or-fail definition—they either successfully change the state of the system, or have no apparent effect.

Non-atomic operations

When the compiler translates the **`BALANCE++`** and **`BALANCE--`** statements, these will be translated to a series of machine instructions (depending on the CPU architecture). On a load/store architecture (for example MIPS):

`BALANCE++`

1) **load** BALANCE from memory into CPU register
2) **increment** BALANCE and save result in register
3) **store** result back to memory

`BALANCE--`

1) **load** BALANCE from memory into CPU register
2) **decrement** BALANCE and save result in register
3) **store** result back to memory

Interleaving of executing threads

Thread A (increment)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	OP	Operands	$t0
			0			
lw	$t0, BALANCE	0	0			
addi	$t0, $t0, 1	1	0			
sw	$t0, BALANCE	1	1			
			1	lw	$t0, BALANCE	1
			1	addi	$t0, $t0, -1	0
			0	sw	$t0, BALANCE	0

If the instructions are executed in this order, the increment and decrement cancel each other and the resulting **BALANCE** is **0**.

Interleaving of executing threads

Thread A (increment)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	OP	Operands	$t0
			0			
lw	$t0, BALANCE	0	0			
			0	lw	$t0, BALANCE	0
addi	$t0, $t0, 1	1	0			
			0	addi	$t0, $t0, -1	-1
			-1	sw	$t0, BALANCE	-1
sw	$t0, BALANCE	1	+1			

Both threads tries to access and update the shared memory location **BALANCE** concurrently. Updates are not atomic and the result depends on the particular order in which the data accesses take place. In this example the resulting **BALANCE** is **+1**.

Interleaving of executing threads

Thread A (increment)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	OP	Operands	$t0
			0			
lw	$t0, BALANCE	0	0			
			0	lw	$t0, BALANCE	0
addi	$t0, $t0, 1	1	0			
sw	$t0, BALANCE	1	1			
			1	addi	$t0, $t0, -1	-1
			-1	sw	$t0, BALANCE	-1

Both threads tries to access and update the shared memory location **BALANCE** concurrently. Updates are not atomic and the result depends on the particular order in which the data accesses take place. In this example the resulting **BALANCE** is **-1**.

Race condition

A race condition or race hazard is the **behaviour** of an electronic, software or other system where the **output** is **dependent** on the sequence or **timing** of other **uncontrollable** events.

It becomes a bug when events do not happen in the intended order.

The term originates with the idea of two signals racing each other to influence the output first.

Data race

A data race occurs when two instructions from different threads access the same memory location and:

- ★ at least one of these accesses is a **write**
- ★ and there is **no synchronization** that is mandating any particular order among these accesses.

shared state

```
#define N 10000

int  BALANCE  0;
```

Thread A

```
for (int i = 0; i < N; i++) {
   BALANCE++;
}
```

Thread B

```
for (int i = 0; i < N; i++) {
   BALANCE--;
}
```

BALANCE == ?

Thread A (increment)				Thread B (decrement)		
OP	**Operands**	**$t0**	**BALANCE**	**OP**	**Operands**	**$t0**
			0			
lw	$t0, BALANCE	0	0			
			0	lw	$t0, BALANCE	0
addi	$t0, $t0, 1	1	0			
sw	$t0, BALANCE	1	1			
			1	addi	$t0, $t0, -1	-1
			-1	sw	$t0, BALANCE	-1

Data race

★ Two threads may read and write the same variable **BALANCE** without synchronization.

Race condition

★ The final value of the variable **BALANCE** depends on how the threads are scheduled.

shared state

```
#define N 10000

int  BALANCE  0;
```

Thread A

```
for (int i = 0; i < N; i++) {
    BALANCE++;
}
```

Thread B

```
for (int i = 0; i < N; i++) {
    BALANCE--;
}
```

BALANCE == ?

Generally, the OS gives no guarantees regarding the interleaving of the threads. Depending on the non deterministic interleaving of the threads, the program may give different results every time it is executed.

shared state

```
#define N 10000

int  BALANCE  0;
```

Thread A

```
for (int i = 0; i < N; i++) {
    BALANCE++;
}
```

Thread B

```
for (int i = 0; i < N; i++) {
    BALANCE--;
}
```

BALANCE == ?

Even if the final value of BALANCE is not deterministic, there is a lower and an upper bound for the final value of BALANCE.

shared state

```
#define N 10000

int  BALANCE  0;
```

Thread A

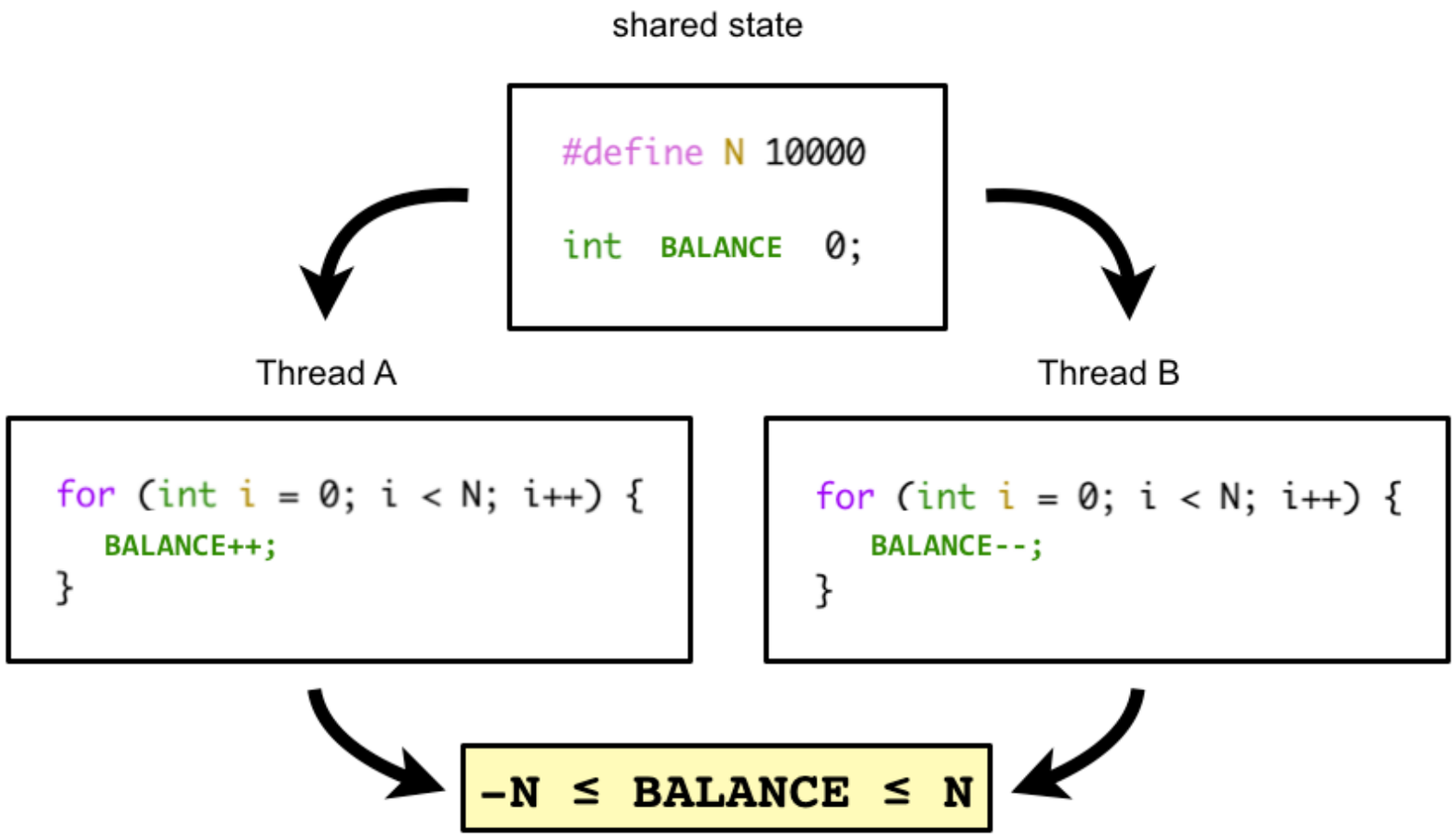
```
for (int i = 0; i < N; i++) {
    BALANCE++;
}
```

Thread B

```
for (int i = 0; i < N; i++) {
    BALANCE--;
}
```

$-N \leq BALANCE \leq N$

Even if the final value of BALANCE is not deterministic, there is a lower and an upper bound for the final value of BALANCE.



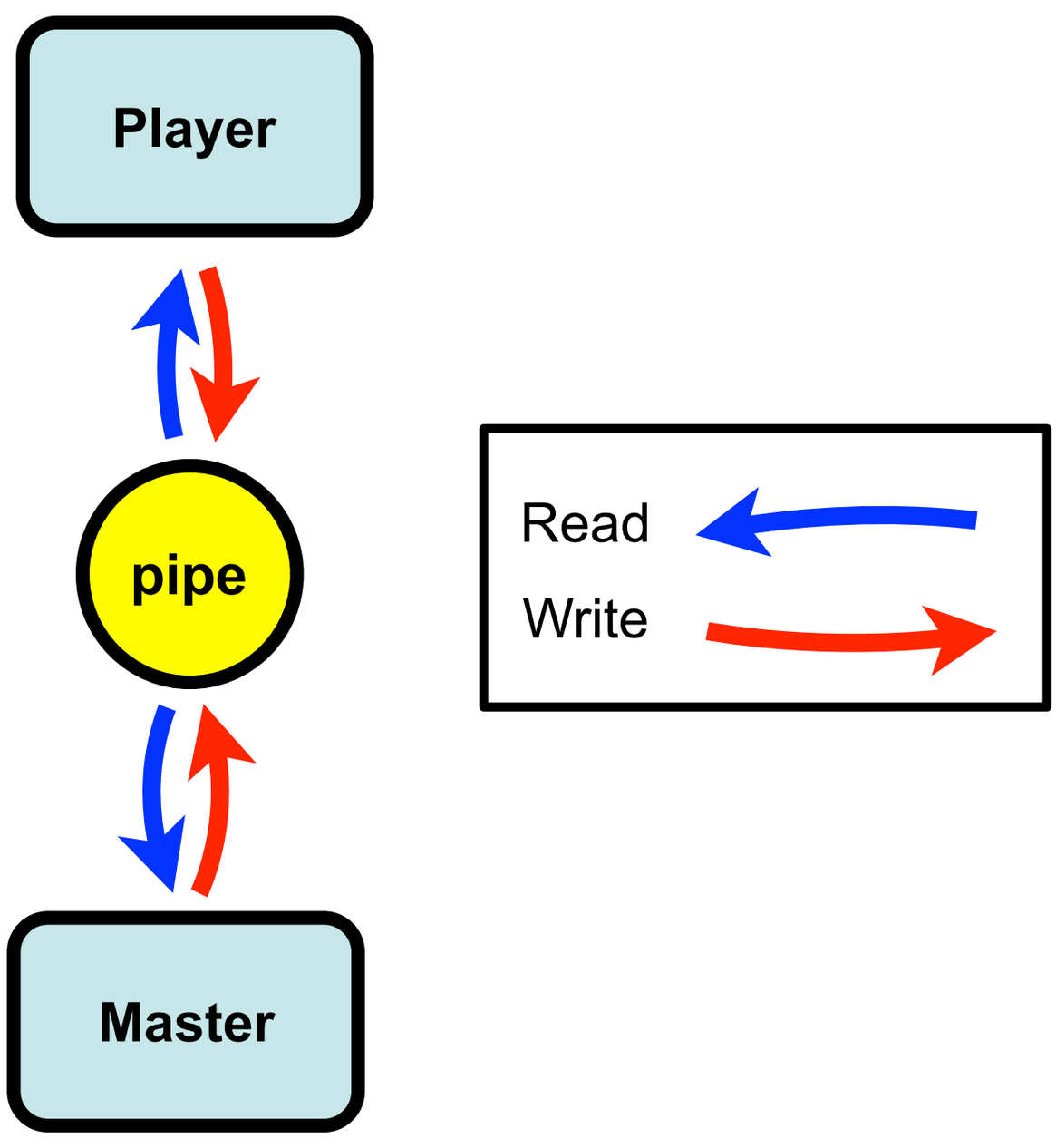
Generally, the OS gives no guarantees regarding the interleaving of the threads. Depending on the non deterministic interleaving of the threads, the program may give different results every time it is executed.

Testing and debugging	When a program is executing concurrently, there are many different execution paths. Testing and debugging concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

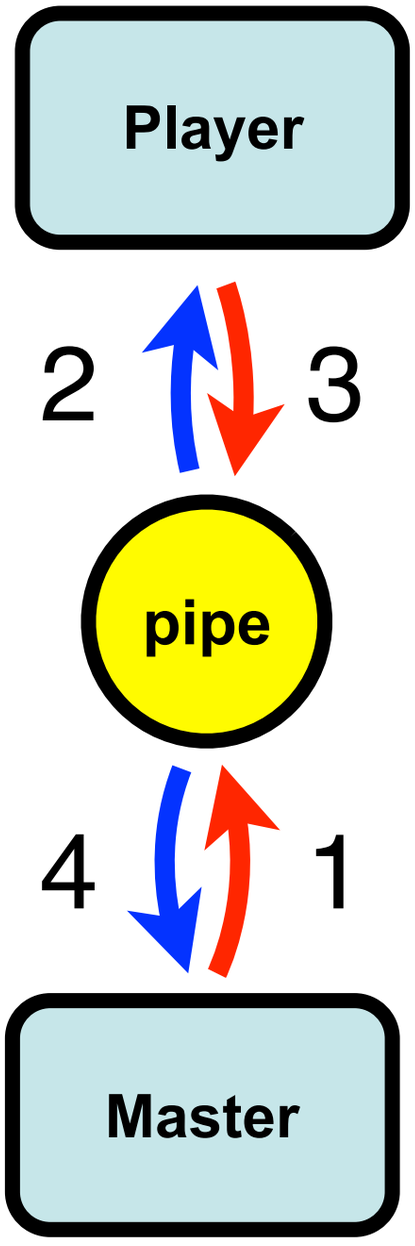
Pipe example

Pipes example

Two processes: a master and player share a single pipe.

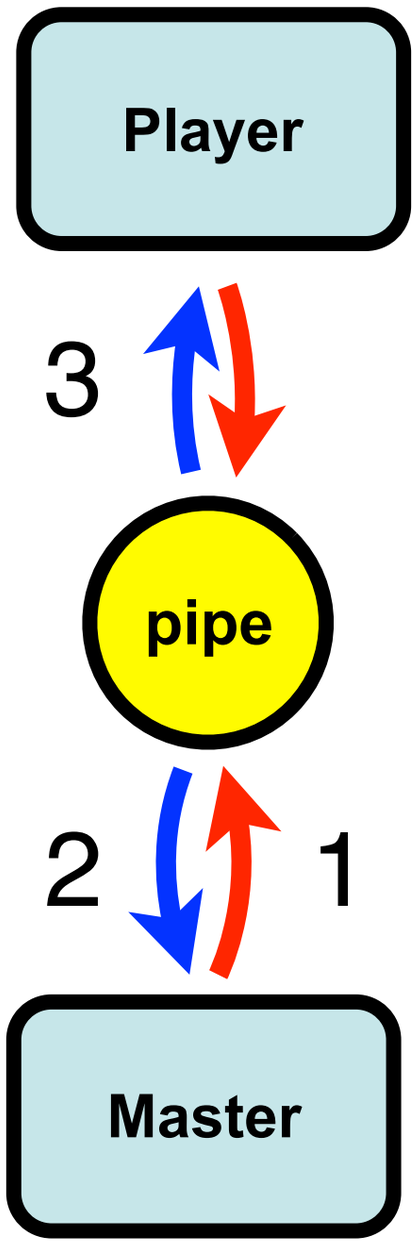


The master sends a seed to the player. The player uses the seed to generate a random number and sends the random number back to the master.



The desired sequence of events

1) Master writes the seed to the pipe.
2) Player reads the seed from the pipe.
3) Player writes the result to the pipe.
4) Master reads the result from the pipe.



Undesired sequence of events

1) Master writes the seed to the pipe.

2) Master reads the "result" from the pipe, i.e., reads the seed.

3) Player attempts to read the seed from the pipe. The pipe is empty and the player is **blocked**.

If pipes are used for bidirectional communication, **race conditions** may occur.

Race condition

≠

Data race

Race conditions and data races are not a subset of one another, neither the necessary, nor the sufficient condition for one another.

Race condition ≠ Data race

There is considerable overlap:

- ★ many race conditions are due to data races
- ★ many data races lead to race conditions.

On the other hand:

- ★ we can have race conditions without data races
- ★ and data races without race conditions.

Source: http://stackoverflow.com/a/18049303/3834375 2016-02-06

http://blog.regehr.org/archives/490 2016-02-06

```
1   transfer1 (amount, from, to) {
2       if (from.balance < amount) return NOPE;
3       to.balance   += amount;
4       from.balance -= amount;
5       return YEP;
6   }
```

Data races

- ★ Multiple threads may read and write the same account balances without synchronization.

Race conditions

- ★ Money on bank accounts may erroneously get added or removed.
- ★ A key invariant (conservation of money) is broken.

```
1  transfer2 (amount, from, to) {
2      atomic { bal = from.balance; }
3      if (bal < amount) return NOPE;
4      atomic { to.balance   += amount; }
5      atomic { from.balance -= amount; }
6      return YEP;
7  }
```

No data races

- ★ Multiple threads may not read and write the same account balances without synchronization.

Race conditions

- ★ Money on bank accounts may erroneously get added or removed.
- ★ Threads can observe memory states where a key invariant (conservation of money) is broken.

```
1  transfer3 (amount, from, to) {
2      atomic {
3          if (from.balance < amount) return NOPE;
4          to.balance   += amount;
5          from.balance -= amount;
6          return YEP;
7      }
8  }
```

No data races

- ★ Multiple threads may not read and write the same account balances without synchronization.

No race conditions

- ★ A key invariant (conservation of money) is not broken.

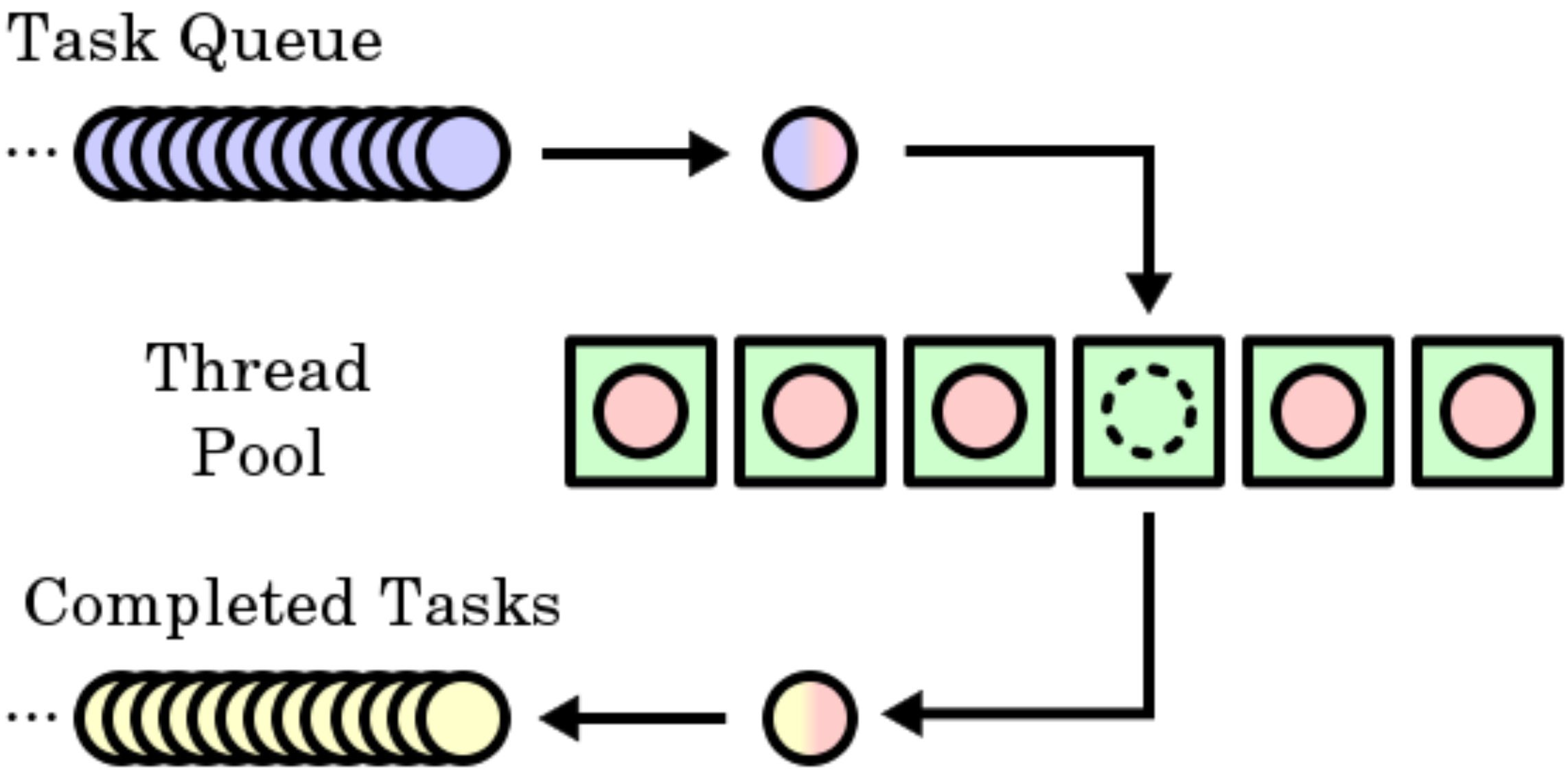
Thread creation and destruction latency

Threads are "cheaper" to create compared to creating processes. Nevertheless, creating a thread requires resources to be allocated and terminating a thread requires these resources to be deallocated.

Sometimes the time used to create and destroy threads becomes a bottle neck.

Thread pools

The idea is to create a pool of threads. Once the thread pool has been created the threads are reused without any overhead for creation and destruction.



Source: https://en.wikipedia.org/wiki/Thread_pool 2018-02-08

- By maintaining a pool of threads, the model increases performance and avoids latency in execution due to frequent creation and destruction of threads for short-lived tasks.

- The number of available threads is tuned to the computing resources available to the program, such as parallel processors, cores, memory, and network sockets.

- Allows the **number of threads** in the application(s) to be **bound** to the size of the pool.

- Thread pools are appropriate when we know there's a stream of jobs to process, even though there could be some time when there are no jobs.

Threading issues

Threading issues

- ★ Semantics of the `fork()` and `exec()` system calls
- ★ Thread cancellation of target thread
  - ◉ Asynchronous
  - ◉ Deferred
- ★ Signal handling
- ★ Thread-specific data

Semantics of `fork()` when using threads

Does `fork()` duplicate only the calling thread or all threads?

- ★ A new process shall be created with a single thread.
- ★ If a multi-threaded process calls `fork()`, the new process shall contain a replica of the calling thread and its entire address space.

Source: http://stackoverflow.com/a/18049303/3834375 2016-02-06

Semantics of exec() when using threads

If a process has multiple threads and one thread calls **`exec()`**, what happens?

- ★ If a thread invokes **`exec()`**, the program specified in the parameter to **`exec()`** will replace the entire process - including all threads.

Thread cancellation

Thread cancellation is the task of terminating a thread before it has completed.

Example 1

- Multiple threads concurrently **searching** through a **database** and one threads returns the result, the remaining threads might be canceled.

Example 2

- Often, a **web page** is loaded using several threads - each **image** is loaded in a separate thread etc. If a user presses the stop button in the browser, how can we cancel all the threads?

Thread cancellation issues

What can possible go wrong when a thread is cancelled? The difficulty with cancellation occurs in situations where:

- ★ resources have been allocated to a canceled thread
- ★ a thread is canceled while in the midst of updating data it is sharing with other threads.

Asynchronous cancellation

When a target thread is **cancelled** (terminated) **immediately** by another thread, this is called asynchronous cancellation. The cancellation is caused by something external to the target thread.

- ★ Often the OS will reclaim system resources from a canceled thread but will not reclaim all resources.
- ★ Cancelling a thread asynchronous may not free a necessary system-wide resource.

Deferred cancellation

Allows the target thread to periodically check if it should be cancelled.

- ★ The thread can perform the check for cancelation at a point at which it can be canceled safely.
- ★ The Pthreads threading API refers to such points as cancellation points.

Signals (1)

Signals are a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems

- ★ A signal is an **notification** sent to a process in order to notify it of an event that occurred.

- ★ When a signal is sent, the operating system **interrupts** the target process' **normal flow of execution** to deliver the signal.

- ★ If the process has previously registered a signal **handler**, that routine is executed. Otherwise, the default signal handler is executed.

Signals (2)

Signals can be synchronous or asynchronous.

Synchronous signals

- ★ Are delivered to the same process that performed the operation that caused the signal.
- ★ Examples of synchronous signals: **illegal memory** access and **division by zero**.

Asynchronous signals

- ★ Are generated by an event external to a running process.
- ★ Typically, an asynchronous signal is sent to another process.
- ★ Examples of asynchronous signals: **terminating** a process (ctrl-c), **timer expire**.

Threads and signal handling

Handling signals is more complicated in a multithreaded process - where should a signal be handled?

Options

- ★ Deliver the signal to the thread to which the signal applies.
- ★ Deliver the signal to every thread in the process.
- ★ Deliver the signal to certain threads in the process.
- ★ Assign a specific thread to receive all signals for the process.

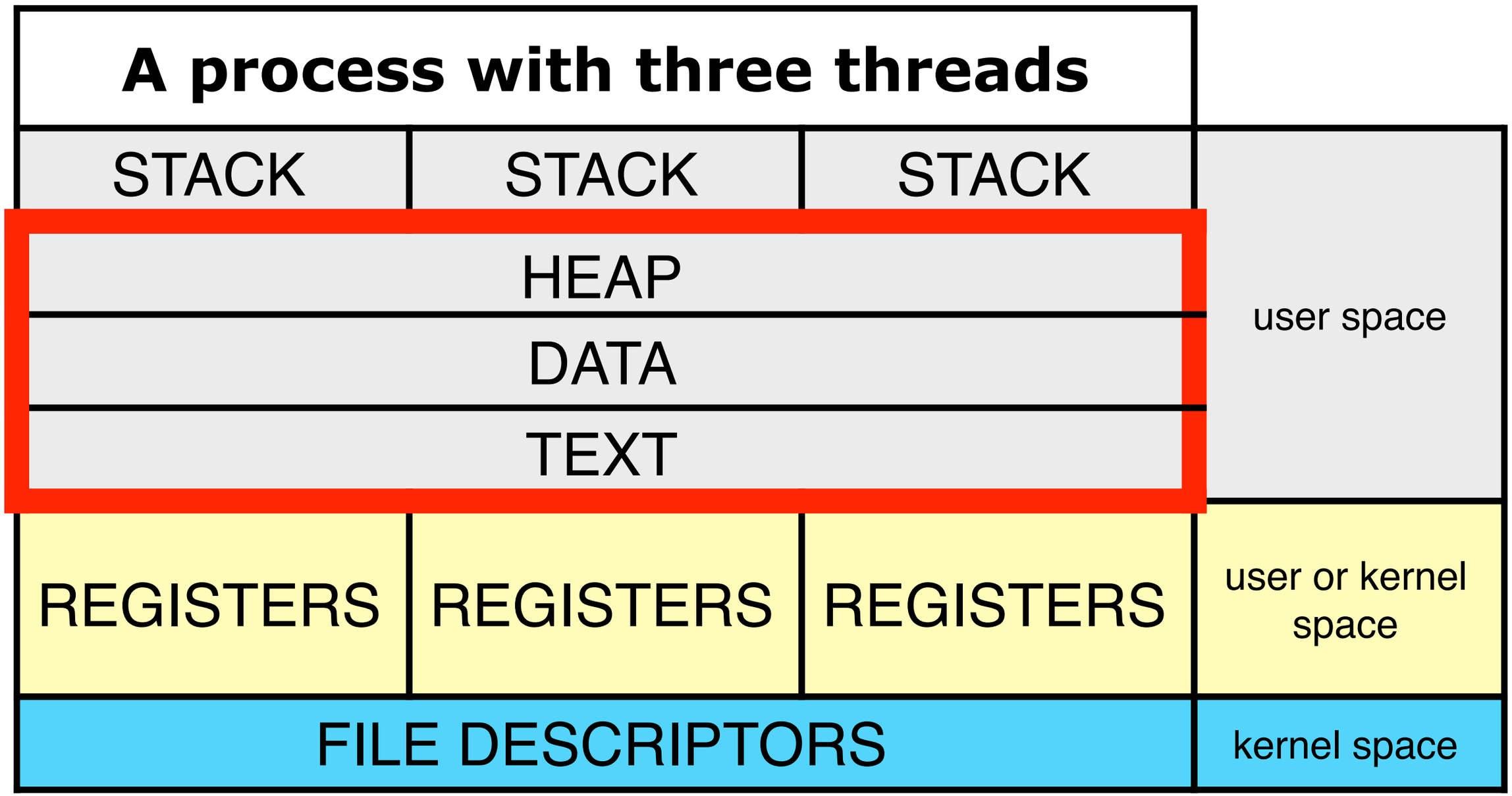
Synchronous signals need to be delivered to the thread causing the signal.

Some asynchronous signals such as ctrl-c (terminate) should be sent to all threads.

Thread specific data

Sometimes we need to maintain data on a per thread basis for data that we don't intend to share with other threads.

If `n` = number of threads, we could allocate an array `data[n]` with per thread data and **trust thread** `i` to only access `data[i]` and not access any other elements in `data[]`.



Most thread libraries - including Win32 and Pthreads provides some sort of support for thread-specific data. Java also provides support for thread-specific data.

- ★ Useful when you do not have control over the thread creation process (i.e., when using a thread pool).
- ★ All threads in a process have access to the entire address space of the process. There is no way for one thread to prevent another from accessing its data. This is true even for thread-specific data.
- ★ Even though the underlying implementation doesn't prevent access, the functions provided in the API to manage thread-specific data promote data separation among threads.

Java threads

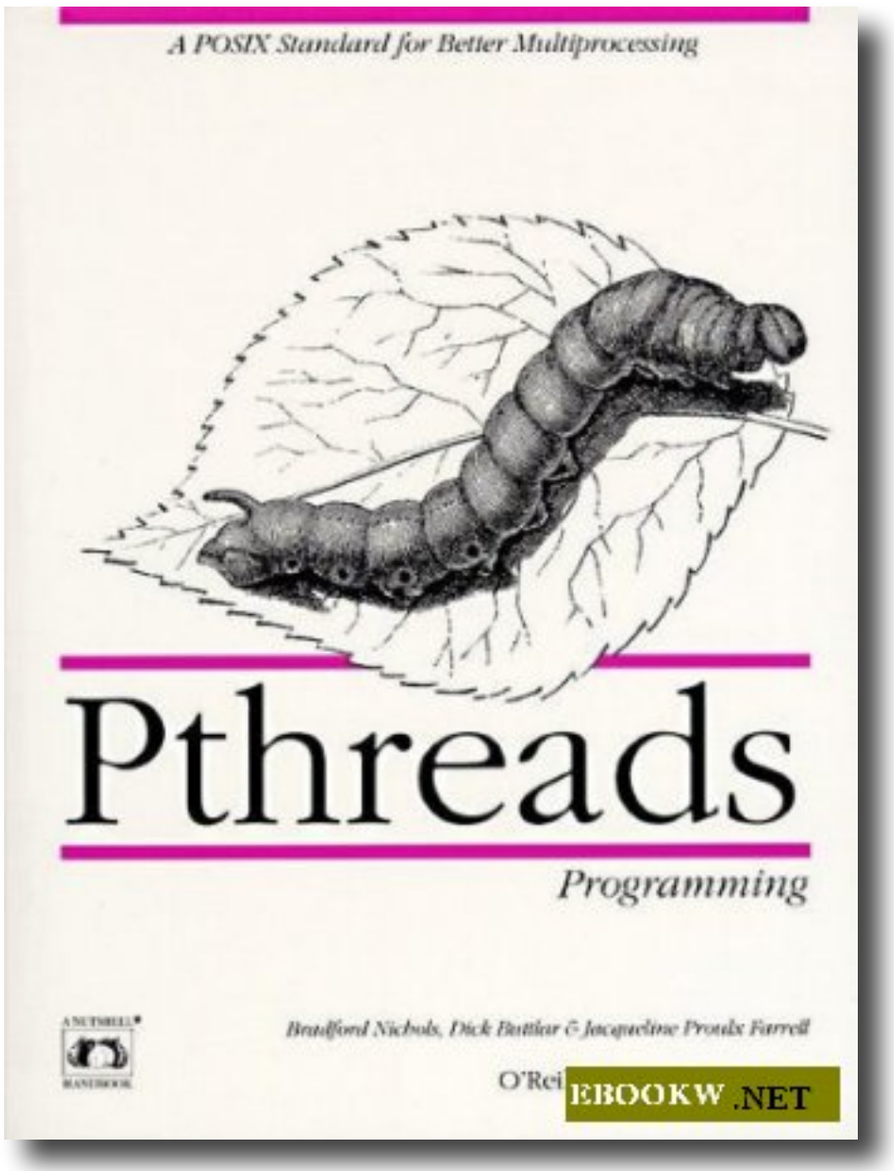
Java threads are managed by the JVM. Typically implemented using the thread model provided by underlying OS.

Java threads may be created by:

- ★ extending the Thread class
- ★ implementing the Runnable interface.

Pthreads

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



- ★ May be provided either as user-level or kernel-level.
- ★ API specifies behaviour of the thread library, implementation depends on the specific library.
- ★ Common in UNIX operating systems (Solaris, Linux, Mac OS X).

More details in the following self study material

Implementing threads

Module 4 self study material

Operating systems 2019

1DT044, 1DT096 and 1DT003

February 2019 karl.marklund@it.uu.se Uppsala university

More details in the following self study material

Pthreads

Module 4 self study material

Callbacks

Pointers to functions

`pthread_create()`

`pthread_exit()`

`pthread_join()`

Semantics of `fork()` when using threads

Concurrent data modification and data races

Operating systems 2018

1DT044, 1DT096, 1DT003

February 2019 karl.marklund@it.uu.se Uppsala university