Concurrency

Johan Montelius

KTH

2021

What is concurrency?

What is concurrency?

Concurrency: (the illusion of) happening at the same time.

What is concurrency?

Concurrency: (the illusion of) happening at the same time.

A property of the programing model.

What is concurrency?

Concurrency: (the illusion of) happening at the same time.

A property of the programing model.

Why would we want to do things concurrently?

What is parallelism?

What is parallelism?

Parallelism: the ability to do several things at the same time.

What is parallelism?

Parallelism: the ability to do several things at the same time.

A property of the execution.

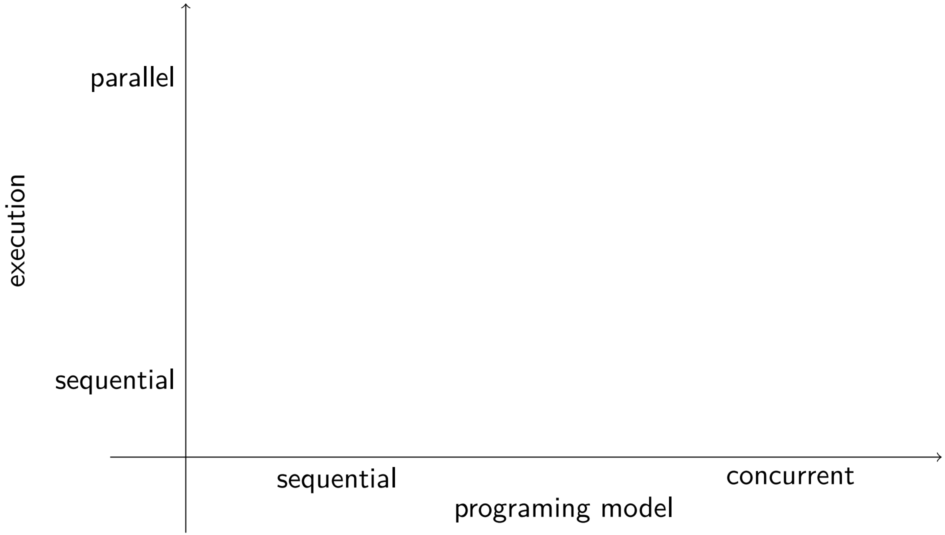
What is parallelism?

Parallelism: the ability to do several things at the same time.

A property of the execution.

Why would we want to do things in parallel?

concurrency vs parallelism



Why in this course?

The problem of concurrency was first encountered in the implementation of operating systems. It has since been a central part in any course on operating systems.

Why in this course?

The problem of concurrency was first encountered in the implementation of operating systems. It has since been a central part in any course on operating systems.

Today - concurrency is such an important topic that it could (and often do) fill up a course of it's own.

What is the problem?

If concurrent activities are not manipulating a shared resource then it's not a problem.

What is the problem?

If concurrent activities are not manipulating a shared resource then it's not a problem.

We often want to share resources between concurrent activities.

What is the problem?

If concurrent activities are not manipulating a shared resource then it's not a problem.

We often want to share resources between concurrent activities.

What do two UNIX processes share?

A process

As we have learned - the unit of a computation.

A process

As we have learned - the unit of a computation.

- a program

A process

As we have learned - the unit of a computation.

- a program
- an instruction pointer

A process

As we have learned - the unit of a computation.

- a program
- an instruction pointer
- a computation stack

A process

As we have learned - the unit of a computation.

- a program
- an instruction pointer
- a computation stack
- a data segment for static data structures

A process

As we have learned - the unit of a computation.

- a program
- an instruction pointer
- a computation stack
- a data segment for static data structures
- a heap for dynamic data structures

A process

As we have learned - the unit of a computation.

- a program
- an instruction pointer
- a computation stack
- a data segment for static data structures
- a heap for dynamic data structures
- a file table of open files

A process

As we have learned - the unit of a computation.

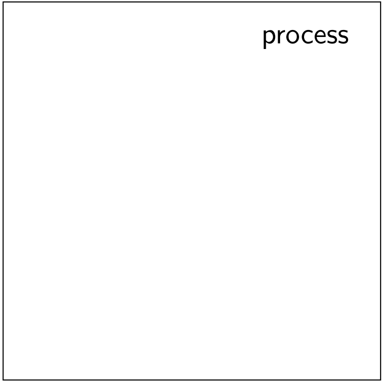
- a program
- an instruction pointer
- a computation stack
- a data segment for static data structures
- a heap for dynamic data structures
- a file table of open files
- signal handlers, ...

A process

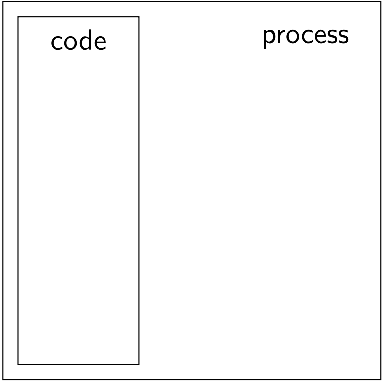
As we have learned - the unit of a computation.

- a program
- an instruction pointer
- a computation stack
- a data segment for static data structures
- a heap for dynamic data structures
- a file table of open files
- signal handlers, ...

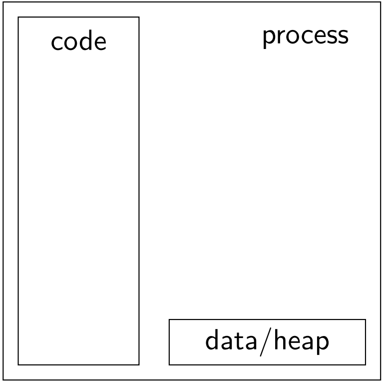
A thread



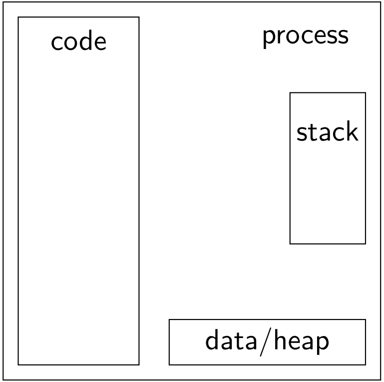
A thread



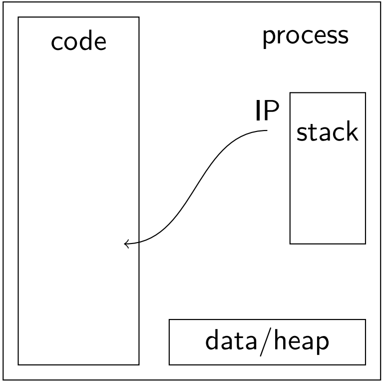
A thread



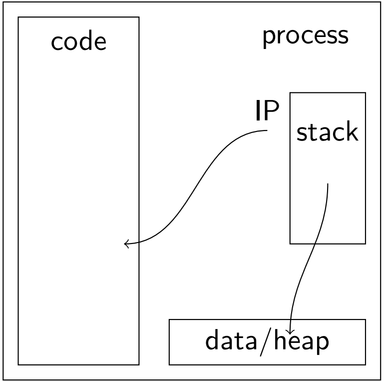
A thread



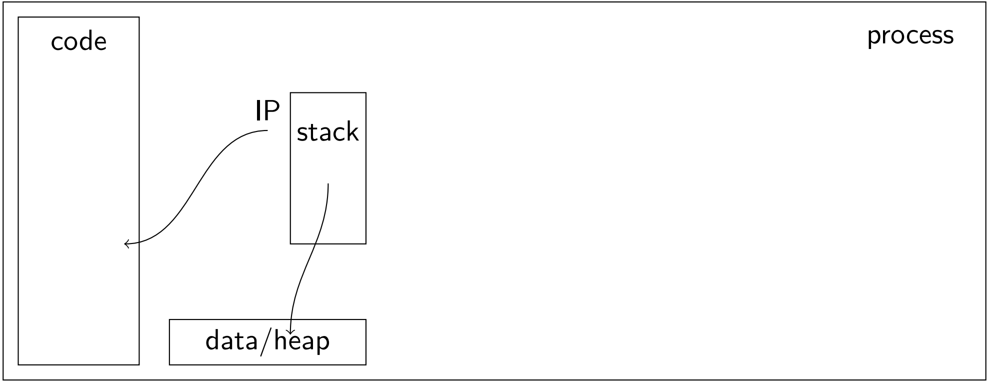
A thread



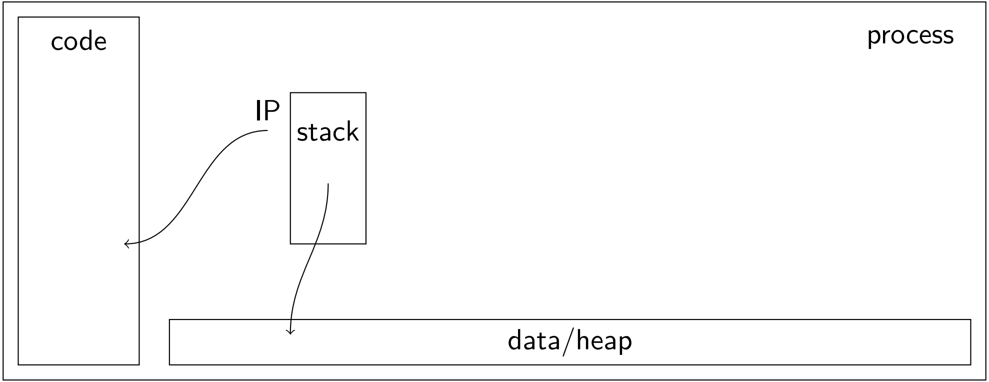
A thread



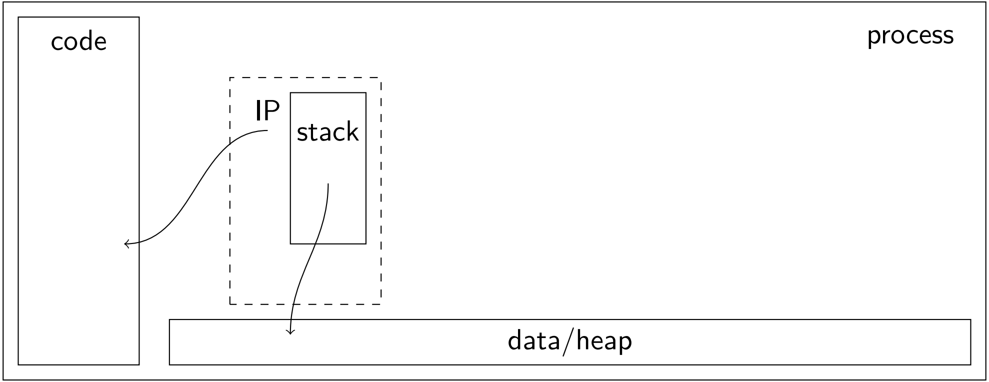
A thread



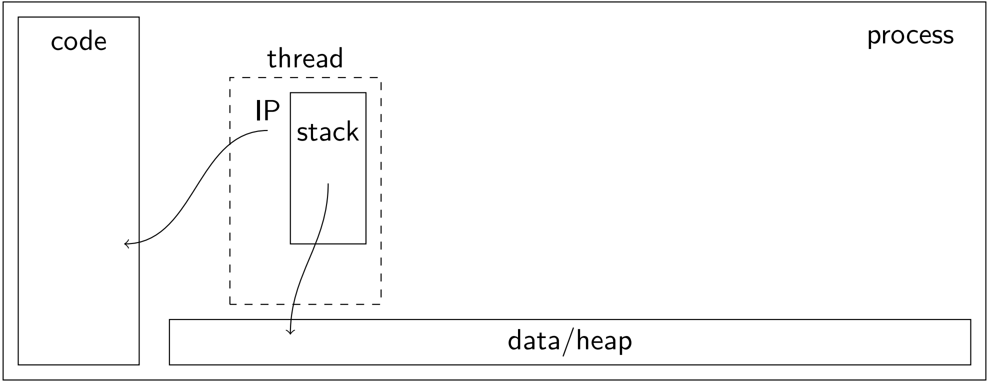
A thread



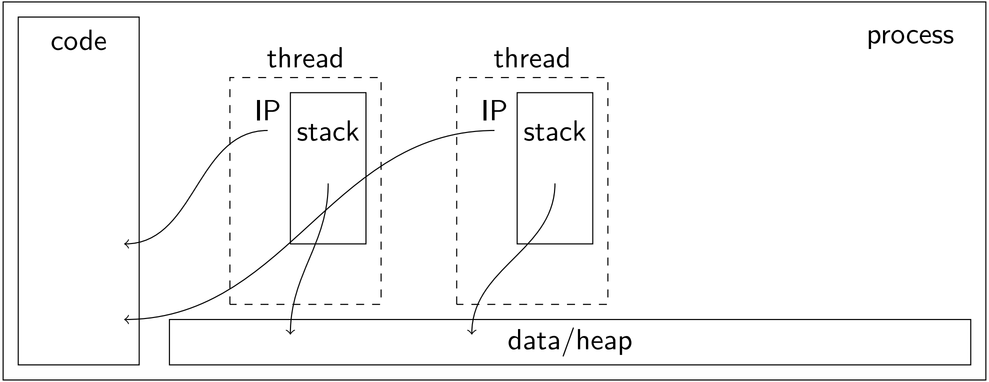
A thread



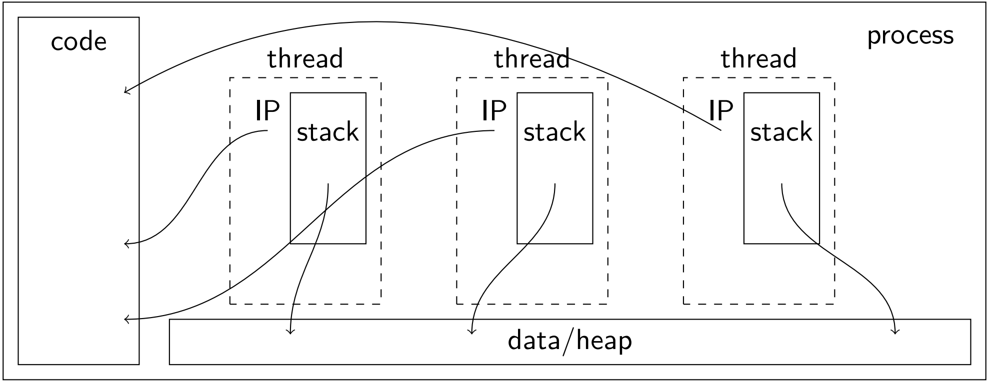
A thread



A thread



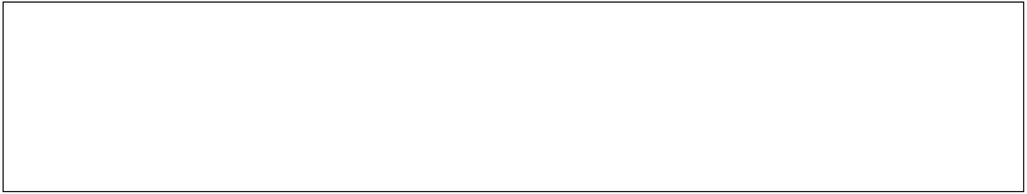
A thread



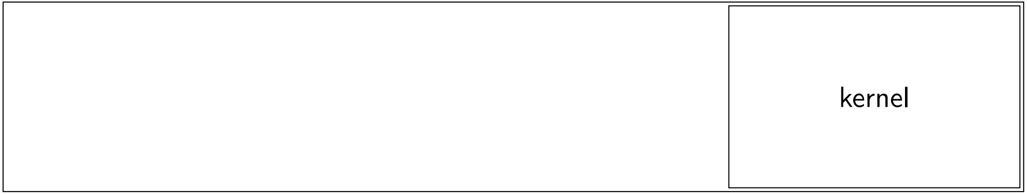
code
process
thread
thread
thread
IP
stack
IP
stack
IP
stack
data/heap

Virtual memory layout

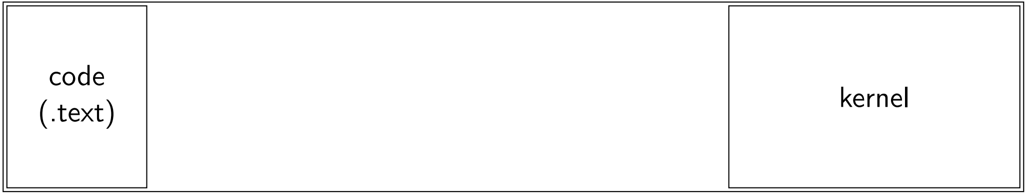
Virtual memory layout



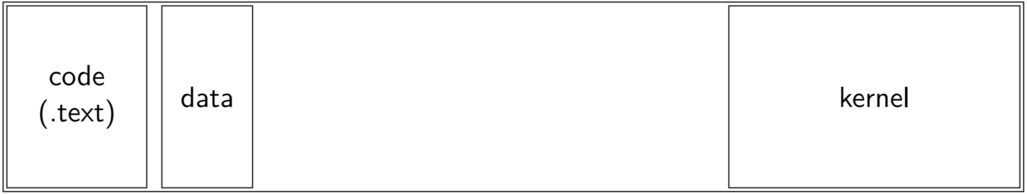
Virtual memory layout



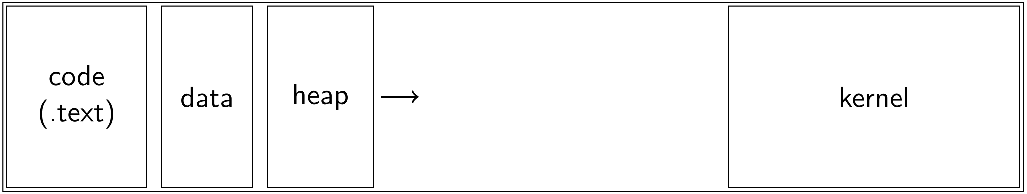
Virtual memory layout



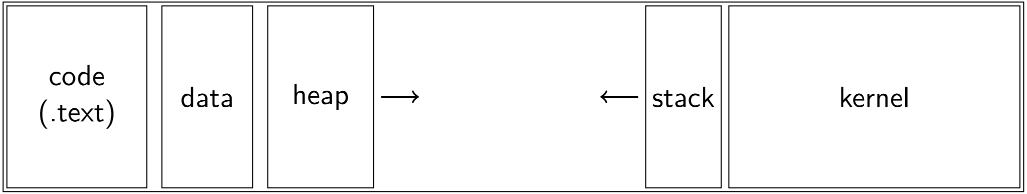
Virtual memory layout



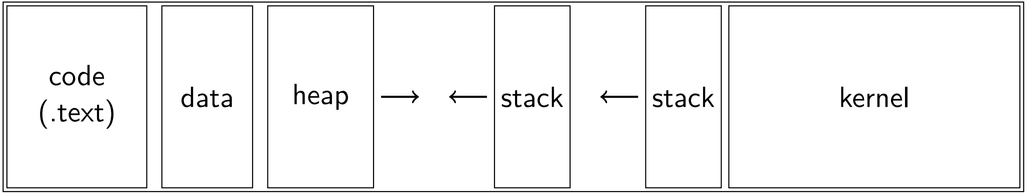
Virtual memory layout



Virtual memory layout



Virtual memory layout



```
#include <pthread.h>
#include <stdio.h>

int loop = 10;
int count = 0;

void *hello(char *name) {
  for(int i = 0; i < loop; i++) {
    count++;
    printf("hello %s %d\n", name, count);
  }
}

int main() {
  pthread_t p1;
  pthread_create(&p1, NULL, hello, "A");
  pthread_join(p1, NULL);
  return 0;
}
```

Concurrency

Concurrency

What is the problem?

Cache coherence

Cache coherence

The CPU uses caches to improve performance, a cache protocol must provide *coherence*.

Cache coherence

The CPU uses caches to improve performance, a cache protocol must provide *coherence*.

- All write operations to a single memory location:
  - are atomic,

Cache coherence

The CPU uses caches to improve performance, a cache protocol must provide *coherence*.

- All write operations to a single memory location:
  - are atomic,
  - performed in program order and

Cache coherence

The CPU uses caches to improve performance, a cache protocol must provide *coherence*.

- All write operations to a single memory location:
  - are atomic,
  - performed in program order and
  - seen by all processes in a total order.

Cache coherence

The CPU uses caches to improve performance, a cache protocol must provide *coherence*.

- All write operations to a single memory location:
    - are atomic,
    - performed in program order and
    - seen by all processes in a total order.

The C compiler can do optimizations that we are not prepared for.

Cache coherence

The CPU uses caches to improve performance, a cache protocol must provide *coherence*.

- All write operations to a single memory location:
  - are atomic,
  - performed in program order and
  - seen by all processes in a total order.

The C compiler can do optimizations that we are not prepared for.

There are several alternatives of how coherence is defined, this is one example

More problems

More problems

What is the expected outcome of an execution?

Sequential consistency

The outcome is the same as if all the operations of the program were executed:

Sequential consistency

The outcome is the same as if all the operations of the program were executed:

as atomic operations in some sequence,

Sequential consistency

The outcome is the same as if all the operations of the program were executed:

as atomic operations in some sequence,

consistent with the *program order* of each thread.

the code

```
int loop = 10;
int count = 0;

void *hello(void *) {
      :
  for(int i = 0; i < loop; i++) {
      count++;
  }
      :
}
```

the code

```
int loop = 10;
int count = 0;

void *hello(void *) {
      :
  for(int i = 0; i < loop; i++) {
      count++;
  }
      :
}
```

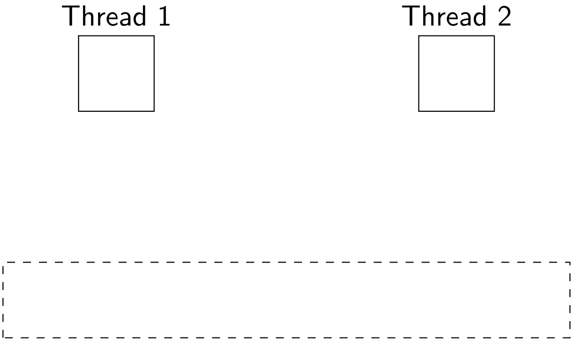
```
.L3:
        movl    count(%rip), %eax
        addl    $1, %eax
        movl    %eax, count(%rip)
        addl    $1, -4(%rbp)
        movl    loop(%rip), %eax
        cmpl    %eax, -4(%rbp)
        jl      .L3
```

What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
    :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```

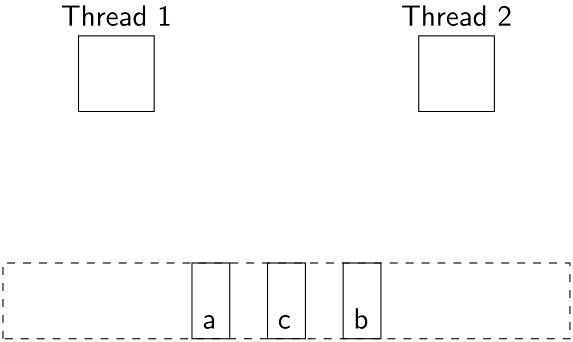
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
   :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



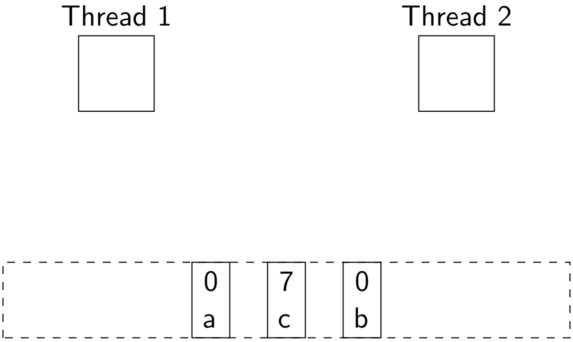
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
   :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



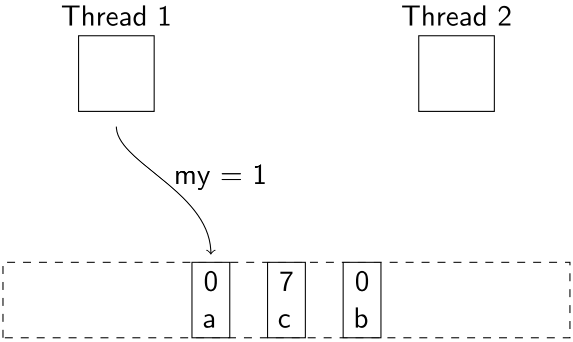
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
   :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



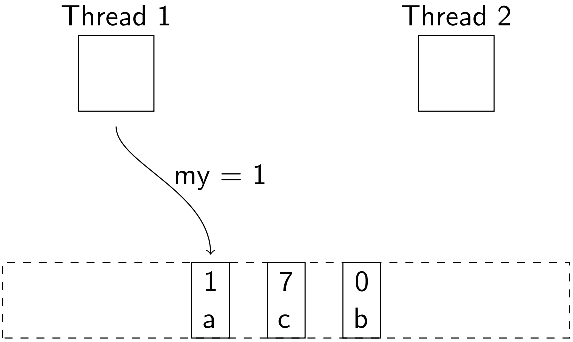
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
   :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



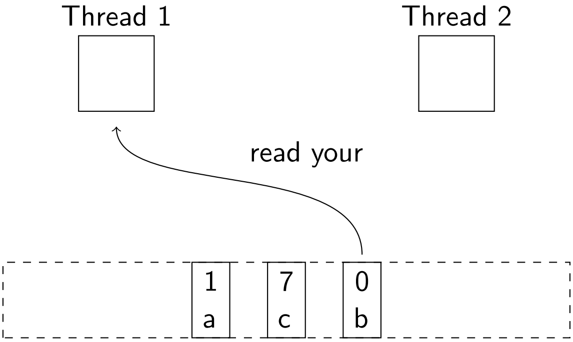
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
    :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



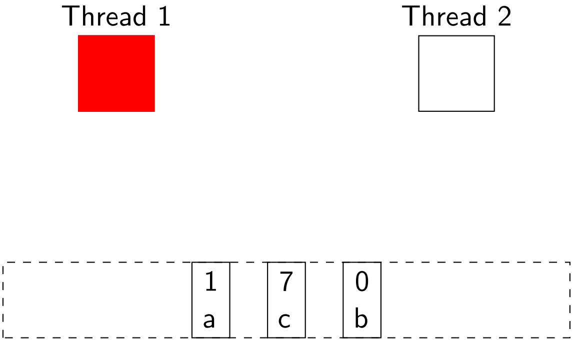
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
    :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



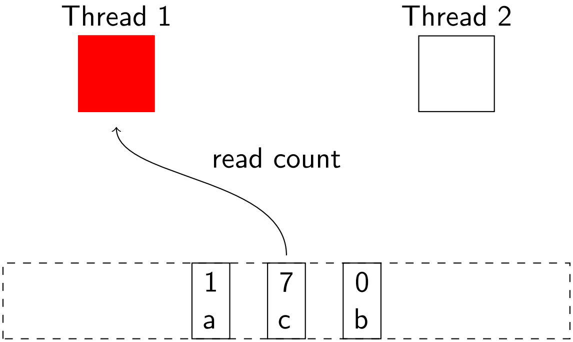
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
    :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



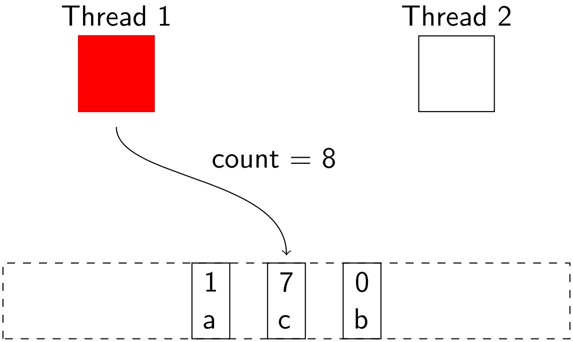
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
   :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



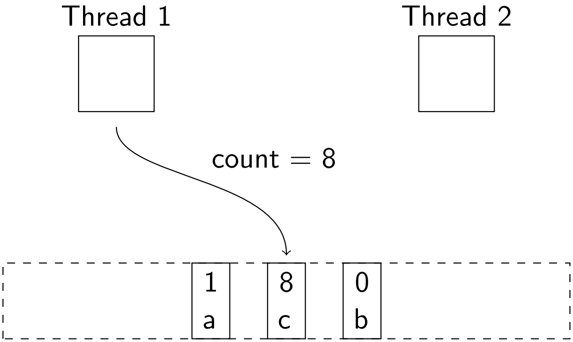
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
   :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



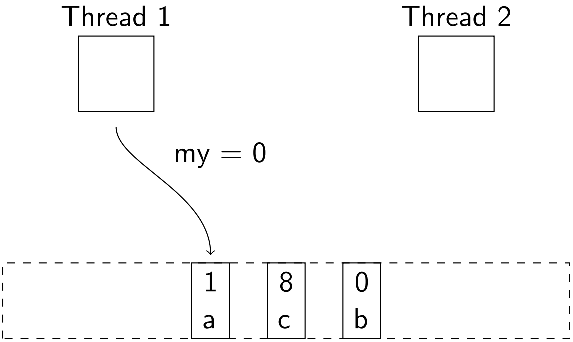
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
   :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



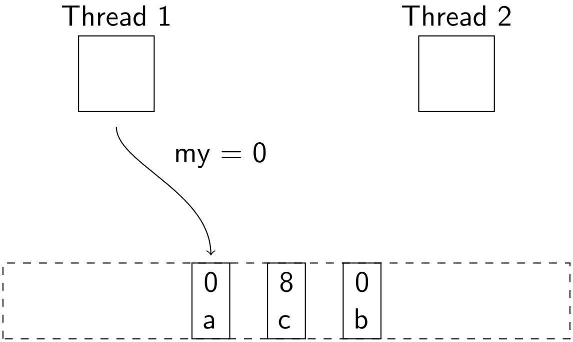
What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
    :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



What about this?

```
int count = 7;
volatile int a = 0;
volatile int b = 0;

void critical( .... ) {
   :
  while(1) {
    my = 1;
    if(your == 0) {
      count++;
      my = 0;
      break;
    } else {
      my = 0;
    }
  }
}
```



Total Store Order (TSO)

Total Store Order (TSO)

Modern CPU:s do not provide *sequential consistency*, they only provide *Total Store Order*.

Total Store Order (TSO)

Modern CPU:s do not provide *sequential consistency*, they only provide *Total Store Order*.

- Write operations are performed in a total order.

Total Store Order (TSO)

Modern CPU:s do not provide *sequential consistency*, they only provide *Total Store Order*.

- Write operations are performed in a total order.
- A process will immediately see its own store operations but,

Total Store Order (TSO)

Modern CPU:s do not provide *sequential consistency*, they only provide *Total Store Order*.

- Write operations are performed in a total order.
- A process will immediately see its own store operations but,
- ... a read operation might *bypass* a write operation of another memory location.

Total Store Order (TSO)

Modern CPU:s do not provide *sequential consistency*, they only provide *Total Store Order*.

- Write operations are performed in a total order.
- A process will immediately see its own store operations but,
- ... a read operation might *bypass* a write operation of another memory location.

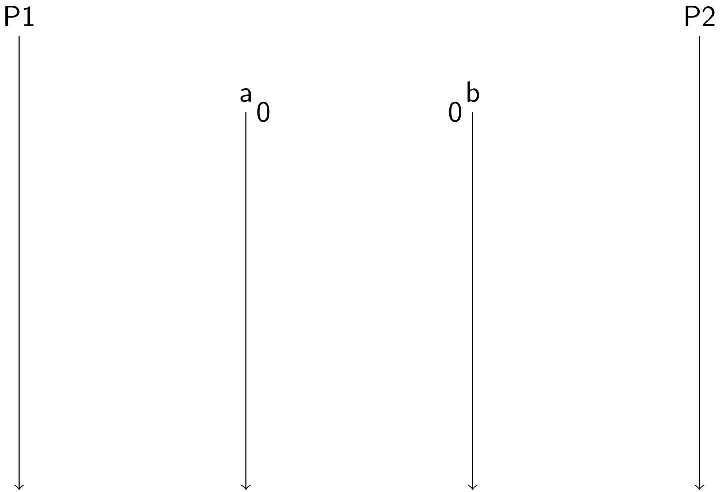
There are operations provided by the hardware that will give us better guarantees.

Total Store Order

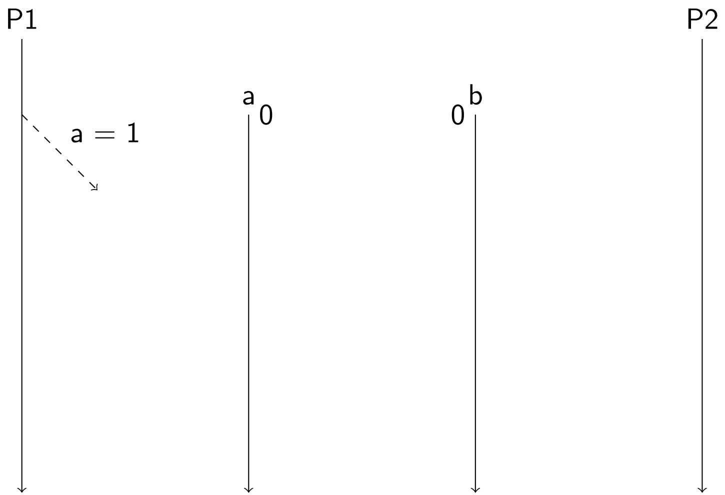
WARNING: the following sequence contains scenes that some viewers may find disturbing.

Total Store Order

WARNING: the following sequence contains scenes that some viewers may find disturbing.

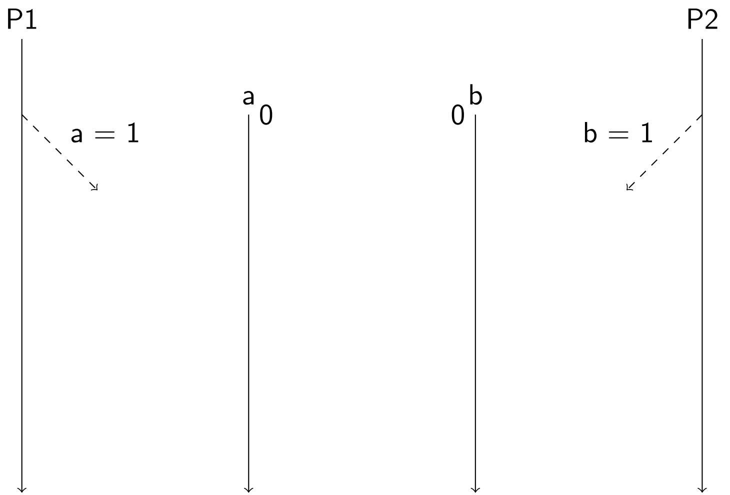


WARNING: the following sequence contains scenes that some viewers may find disturbing.



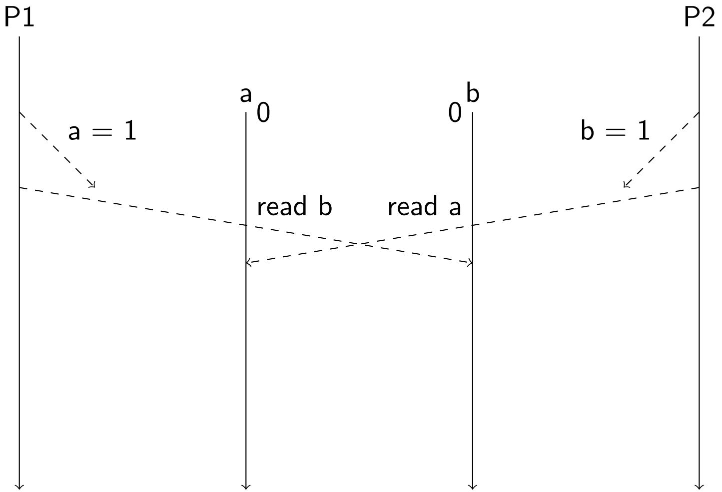
Total Store Order

WARNING: the following sequence contains scenes that some viewers may find disturbing.



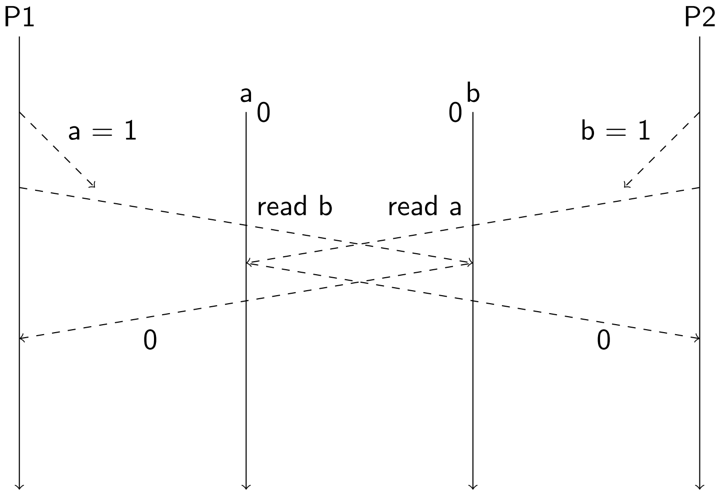
Total Store Order

WARNING: the following sequence contains scenes that some viewers may find disturbing.



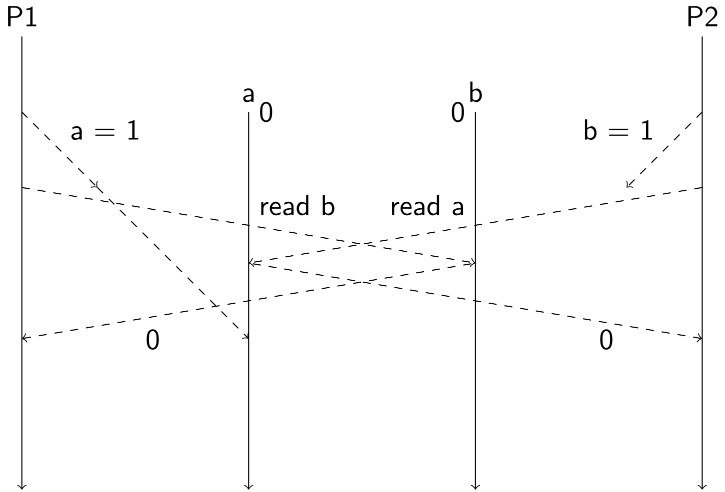
Total Store Order

WARNING: the following sequence contains scenes that some viewers may find disturbing.



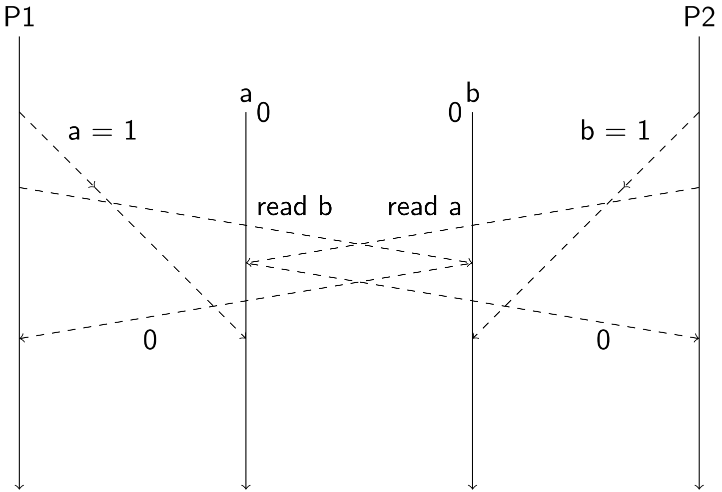
Total Store Order

WARNING: the following sequence contains scenes that some viewers may find disturbing.



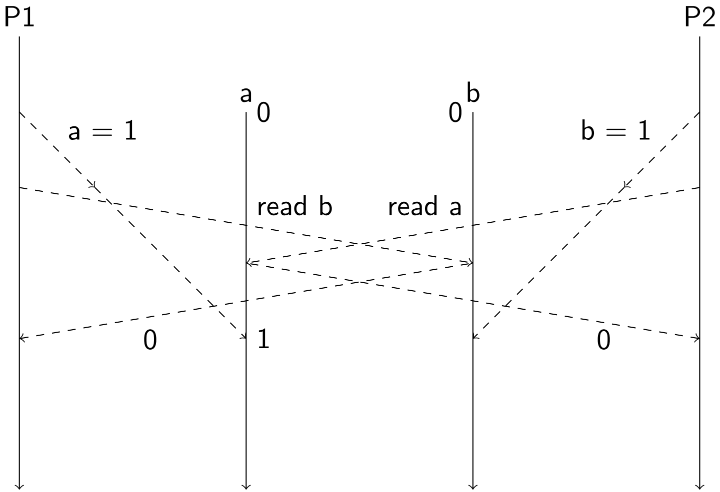
Total Store Order

WARNING: the following sequence contains scenes that some viewers may find disturbing.



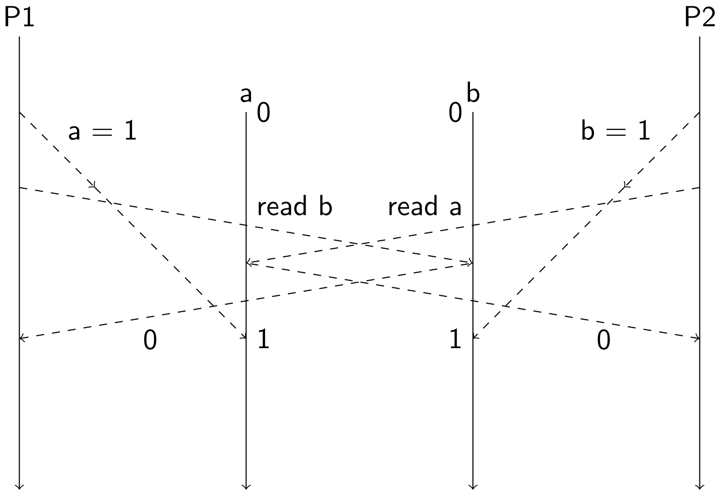
Total Store Order

WARNING: the following sequence contains scenes that some viewers may find disturbing.



Total Store Order

WARNING: the following sequence contains scenes that some viewers may find disturbing.



Hardware support - TGH

Hardware support - TGH

TGH - Thank God for Hardware

Hardware support - TGH

TGH - Thank God for Hardware

- **Fences, barriers etc**: all load and store operations before a fence are guaranteed to be performed before any operations after the fence.
- **Atomic-swap, test-and-set etc**: an instructions that reads and writes to a memory location in one atomic operation.

Hardware support - TGH

TGH - Thank God for Hardware

- **Fences, barriers etc**: all load and store operations before a fence are guaranteed to be performed before any operations after the fence.
- **Atomic-swap, test-and-set etc**: an instructions that reads and writes to a memory location in one atomic operation.

Modern CPU:s provide very weak consistency guarantees if these operations are not used. Don't rely on the program order of your code.

Hardware support - TGH

TGH - Thank God for Hardware

- **Fences, barriers etc**: all load and store operations before a fence are guaranteed to be performed before any operations after the fence.
- **Atomic-swap, test-and-set etc**: an instructions that reads and writes to a memory location in one atomic operation.

Modern CPU:s provide very weak consistency guarantees if these operations are not used. Don't rely on the program order of your code.

Better still - if possible, use a library that handles synchronization.

How to synchronize

How to synchronize

Next week.

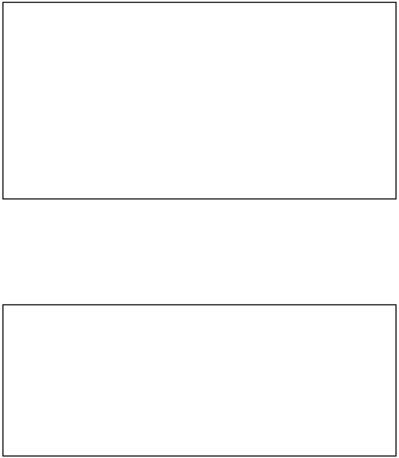
How to implement threads

threads in user space

threads in kernel space

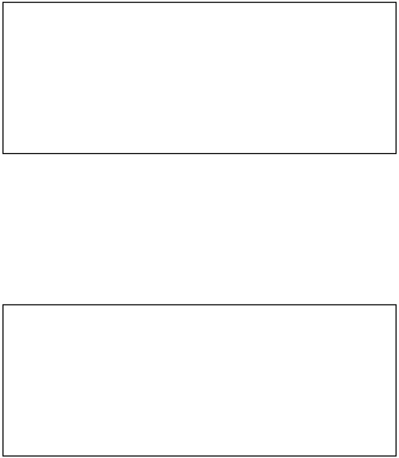
How to implement threads

threads in user space



kernel

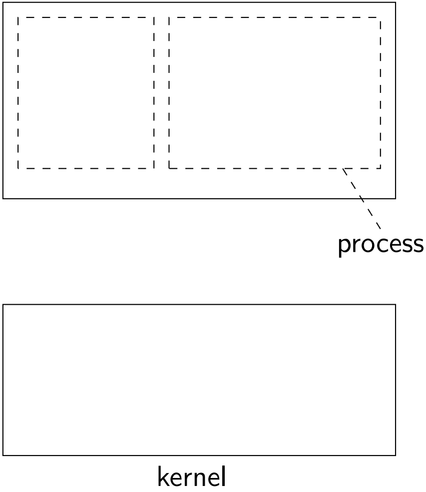
threads in kernel space



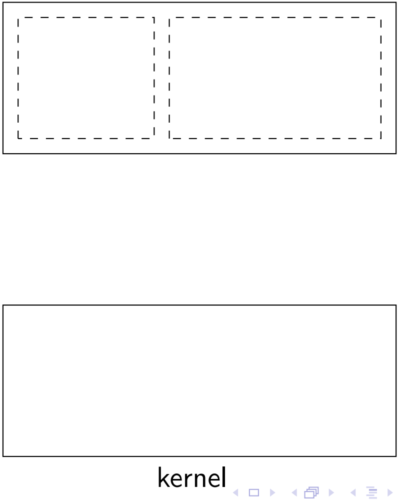
kernel

How to implement threads

threads in user space

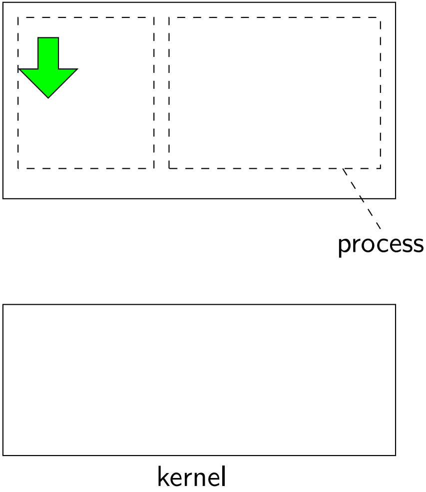


threads in kernel space

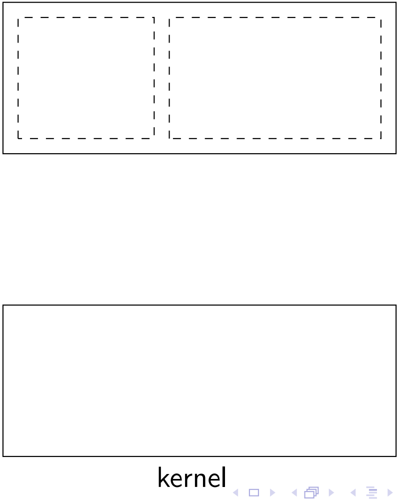


How to implement threads

threads in user space

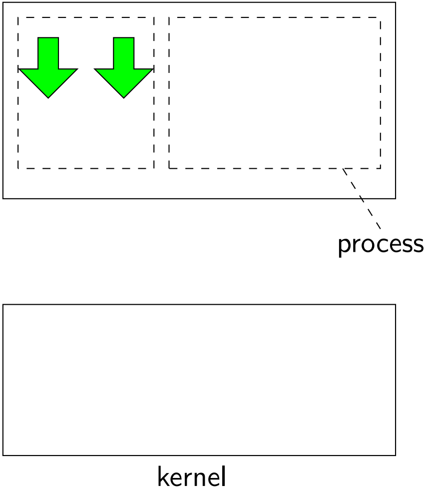


threads in kernel space

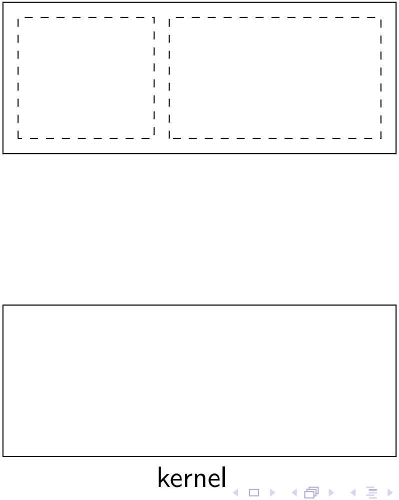


How to implement threads

threads in user space

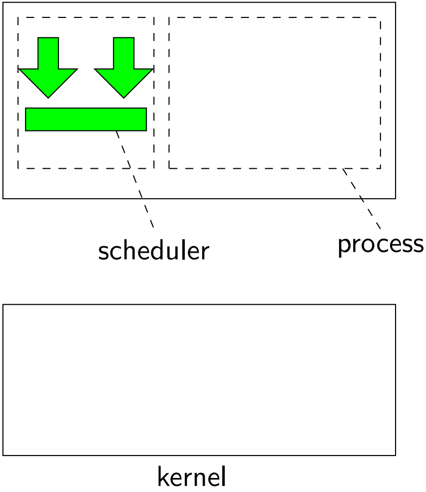


threads in kernel space

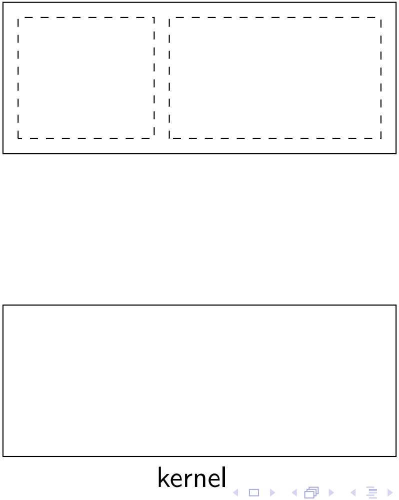


How to implement threads

threads in user space

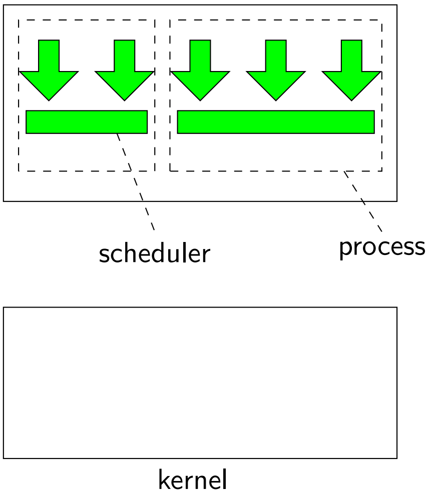


threads in kernel space

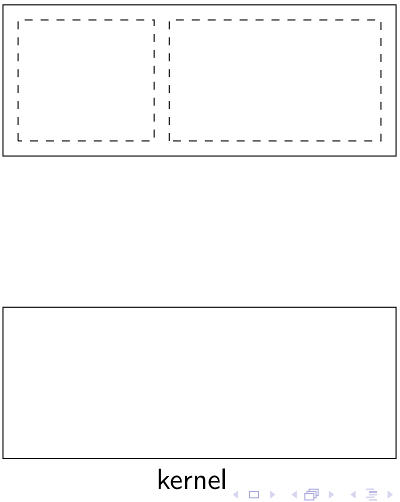


How to implement threads

threads in user space

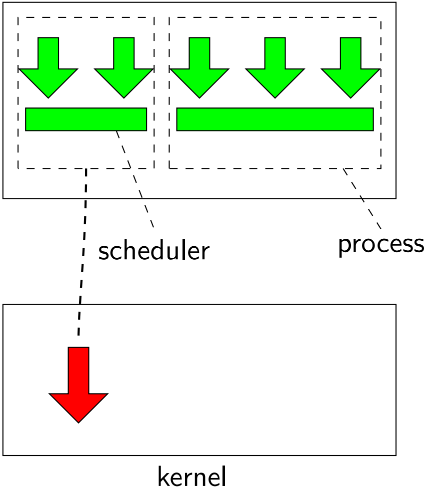


threads in kernel space

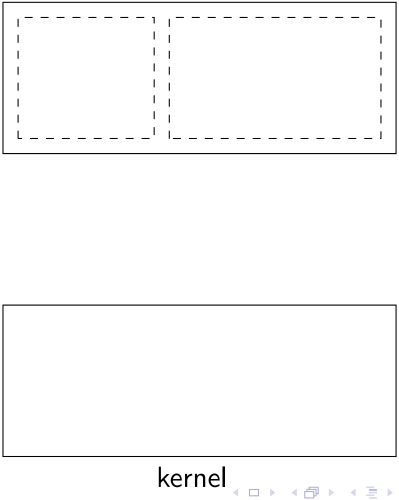


How to implement threads

threads in user space

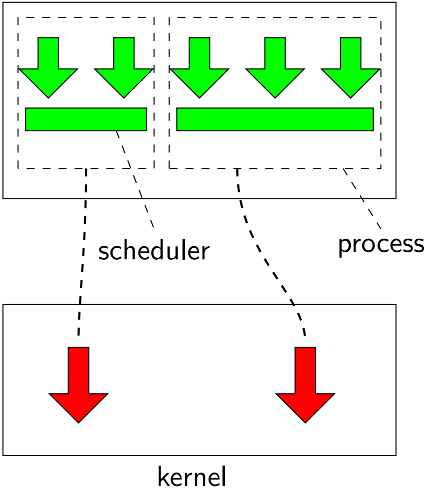


threads in kernel space

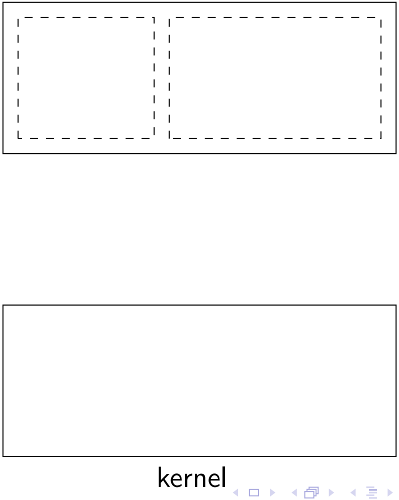


How to implement threads

threads in user space

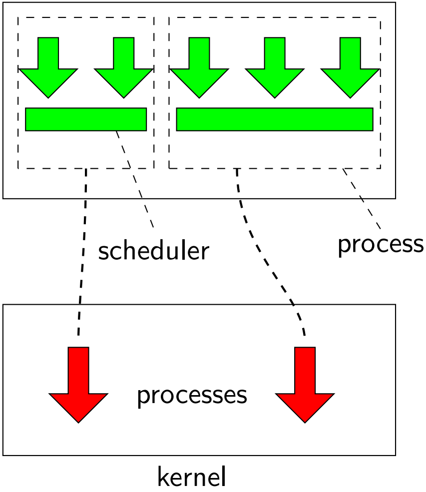


threads in kernel space

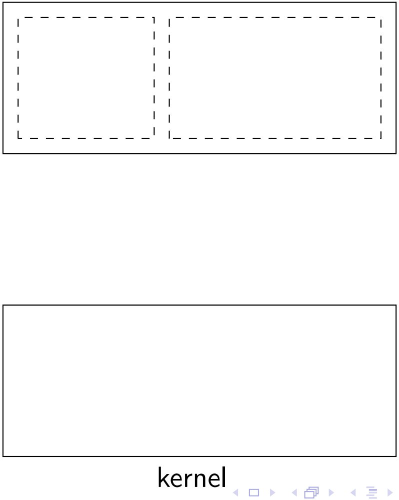


How to implement threads

threads in user space

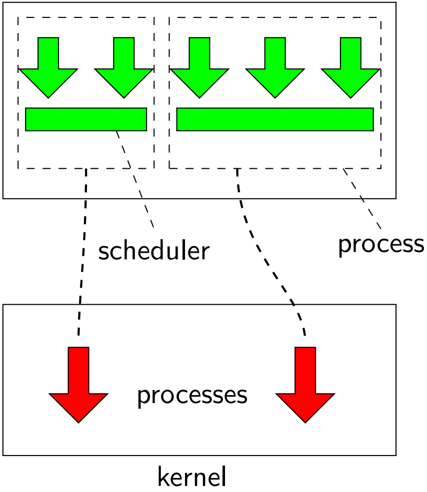


threads in kernel space

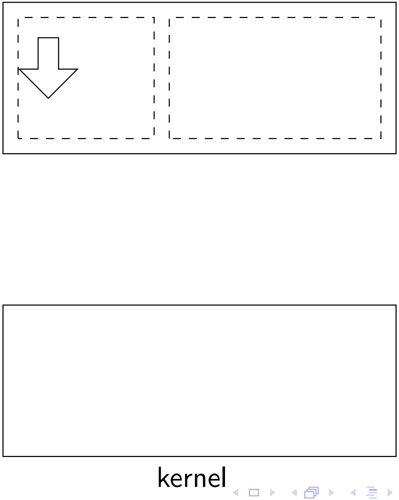


How to implement threads

threads in user space

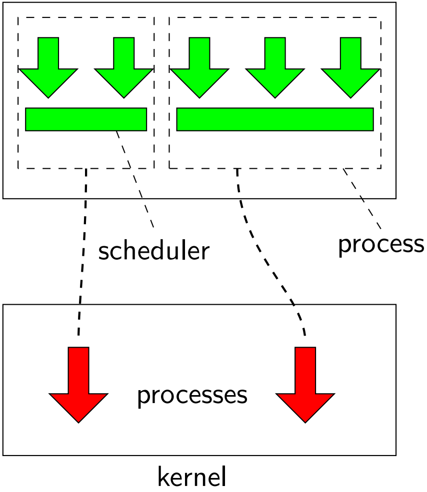


threads in kernel space

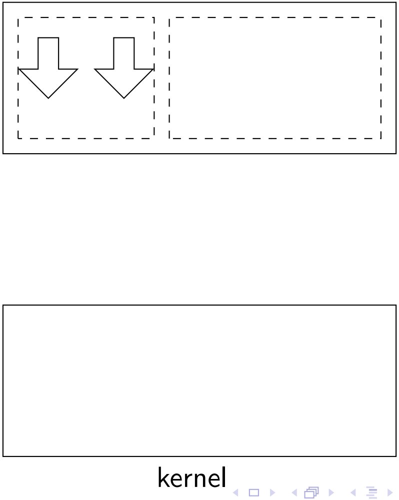


How to implement threads

threads in user space

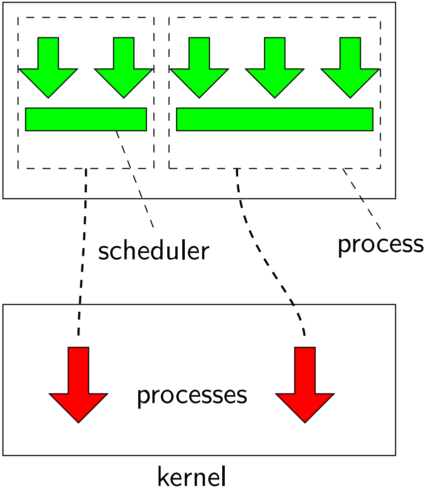


threads in kernel space

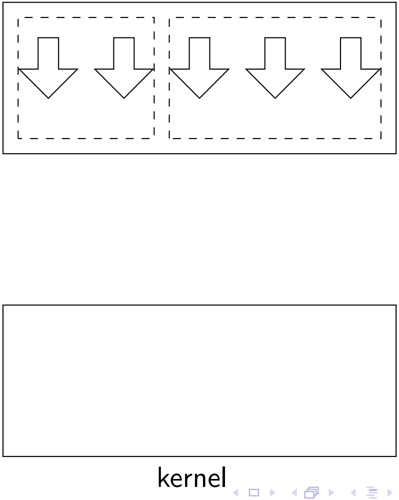


How to implement threads

threads in user space

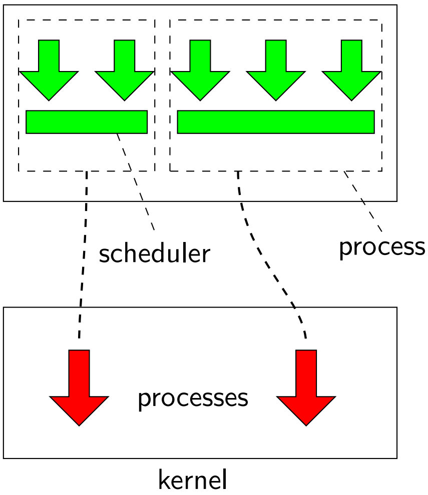


threads in kernel space

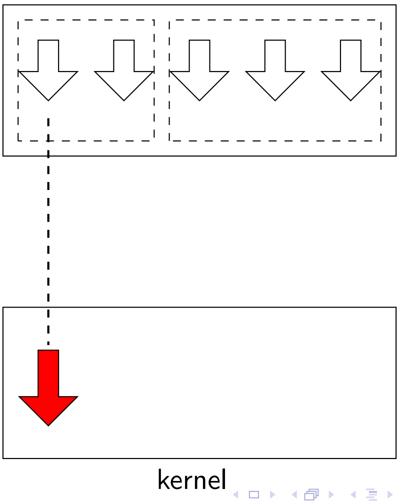


How to implement threads

threads in user space

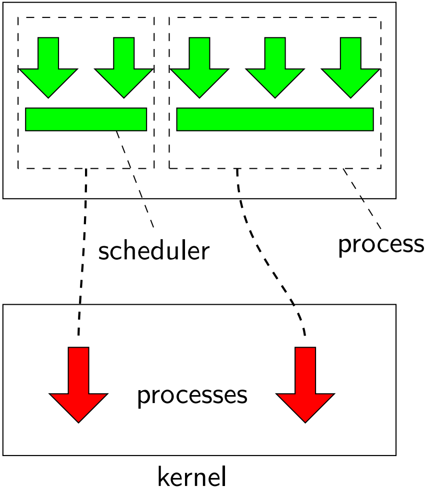


threads in kernel space

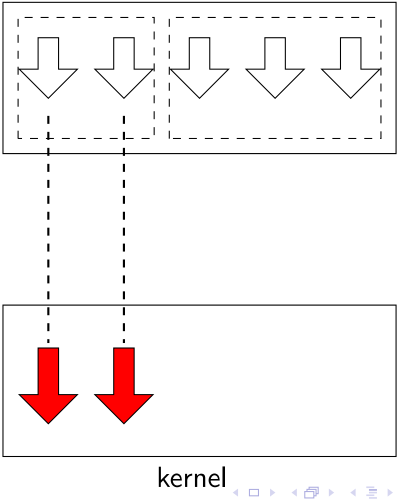


How to implement threads

threads in user space

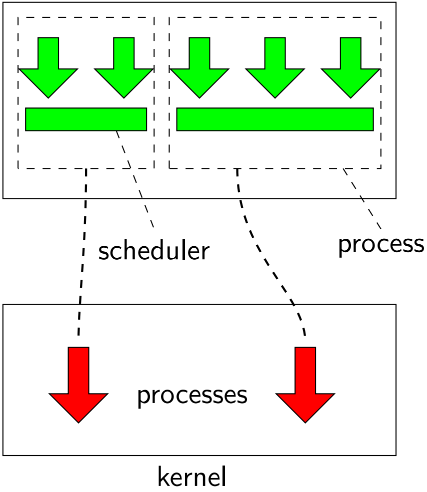


threads in kernel space

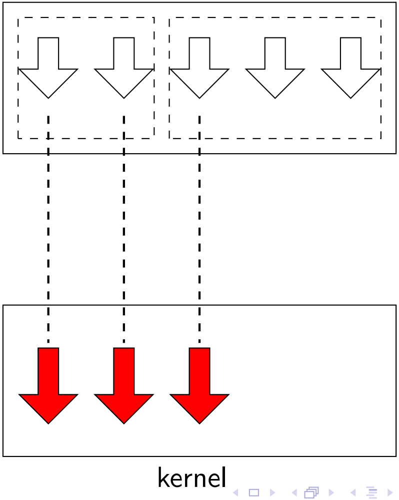


How to implement threads

threads in user space

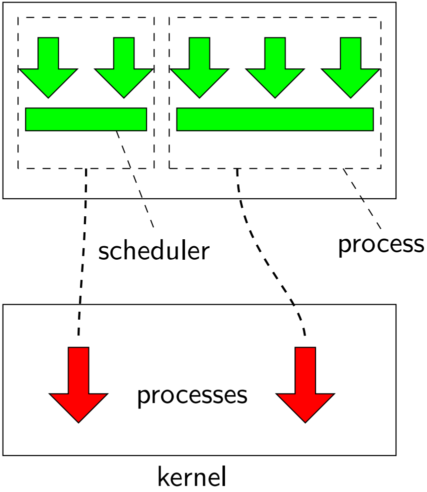


threads in kernel space

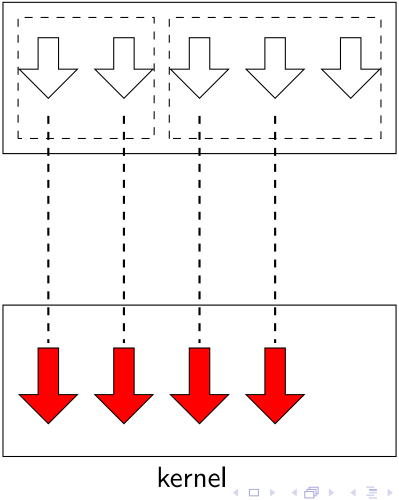


How to implement threads

threads in user space

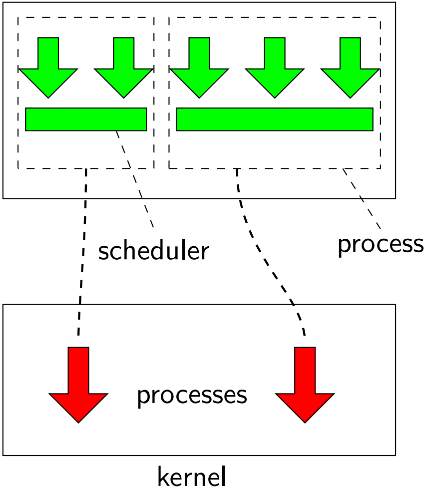


threads in kernel space

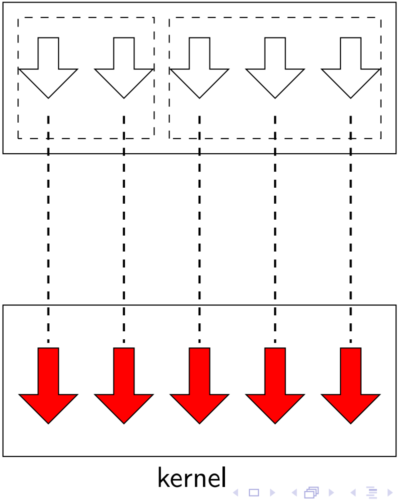


How to implement threads

threads in user space



threads in kernel space



pros and cons

Threads in user space:

- + You can change scheduler.

pros and cons

Threads in user space:

- + You can change scheduler.
- + Very fast task switching.

pros and cons

Threads in user space:

- + You can change scheduler.
- + Very fast task switching.
- - If the process is suspended, all threads are.

pros and cons

Threads in user space:

- + You can change scheduler.
- + Very fast task switching.
- - If the process is suspended, all threads are.
- - A process can not utilize multiple cores.

pros and cons

Threads in user space:

- + You can change scheduler.
- + Very fast task switching.
- - If the process is suspended, all threads are.
- - A process can not utilize multiple cores.

Threads in kernel space:

- + One thread can suspend while other continue to execute.

pros and cons

Threads in user space:

- + You can change scheduler.
- + Very fast task switching.
- - If the process is suspended, all threads are.
- - A process can not utilize multiple cores.

Threads in kernel space:

- + One thread can suspend while other continue to execute.
- + A process can utilize multiple cores.

Threads in user space:

- + You can change scheduler.
- + Very fast task switching.
- - If the process is suspended, all threads are.
- - A process can not utilize multiple cores.

Threads in kernel space:

- + One thread can suspend while other continue to execute.
- + A process can utilize multiple cores.
- - Thread scheduling requires trap to kernel.

pros and cons

Threads in user space:

- + You can change scheduler.
- + Very fast task switching.
- - If the process is suspended, all threads are.
- - A process can not utilize multiple cores.

Threads in kernel space:

- + One thread can suspend while other continue to execute.
- + A process can utilize multiple cores.
- - Thread scheduling requires trap to kernel.
- - No way to change scheduler for a process.

pros and cons

Threads in user space:

- + You can change scheduler.
- + Very fast task switching.
- - If the process is suspended, all threads are.
- - A process can not utilize multiple cores.

Threads in kernel space:

- + One thread can suspend while other continue to execute.
- + A process can utilize multiple cores.
- - Thread scheduling requires trap to kernel.
- - No way to change scheduler for a process.

pros and cons

Threads in user space:

- + You can change scheduler.
- + Very fast task switching.
- - If the process is suspended, all threads are.
- - A process can not utilize multiple cores.

Threads in kernel space:

- + One thread can suspend while other continue to execute.
- + A process can utilize multiple cores.
- - Thread scheduling requires trap to kernel.
- - No way to change scheduler for a process.

Which approach is taken by GNU/Linux?

Java, Haskell and Erlang

How is this handled in high level languages?

Java, Haskell and Erlang

How is this handled in high level languages?

- Java: each Java thread mapped to one operating system thread.

Java, Haskell and Erlang

How is this handled in high level languages?

- Java: each Java thread mapped to one operating system thread.
- Erlang and Haskell: Language threads scheduled by the virtual machine. The virtual machine will use several operating system threads to have several outstanding system calls, utilize multiple cores etc.

Java, Haskell and Erlang

How is this handled in high level languages?

- Java: each Java thread mapped to one operating system thread.
- Erlang and Haskell: Language threads scheduled by the virtual machine. The virtual machine will use several operating system threads to have several outstanding system calls, utilize multiple cores etc.

Java, Haskell and Erlang

How is this handled in high level languages?

- Java: each Java thread mapped to one operating system thread.
- Erlang and Haskell: Language threads scheduled by the virtual machine. The virtual machine will use several operating system threads to have several outstanding system calls, utilize multiple cores etc.

Java originally had user space threads, and introduced the name, "green threads". This was later replaced by "native threads" i.e. each Java thread attached to a kernel operating system thread.

an experiment

How long time does it take to send a message around a ring of a hundred threads?

pthread_create() - from man pages

```
#include <pthread.h>
```

pthread_create() - from man pages

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

pthread_create() - from man pages

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

- `pthread_t *thread` : a pointer to a *thread structure*.

pthread_create() - from man pages

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

- `pthread_t *thread` : a pointer to a *thread structure*.
- `const pthread_attr_t *attr` : a pointer to a structure that are the *attributes* of the thread.

pthread_create() - from man pages

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

- `pthread_t *thread` : a pointer to a *thread structure*.
- `const pthread_attr_t *attr` : a pointer to a structure that are the *attributes* of the thread.
- `void *(*start_routine) (void *)` : a pointer to a function that takes one argument, `(void*)`, with return value `void*`.
- `void *arg` : the arguments to the function, given as a a `void *`.

pthread_create() - from man pages

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

- `pthread_t *thread` : a pointer to a *thread structure*.
- `const pthread_attr_t *attr` : a pointer to a structure that are the *attributes* of the thread.
- `void *(*start_routine) (void *)` : a pointer to a function that takes one argument, `(void*)`, with return value `void*`.
- `void *arg` : the arguments to the function, given as a a `void *`.

Compile and link with `-lpthread`.

Pthreads in Linux

How do we implement threads in Linux?

Pthreads in Linux

How do we implement threads in Linux?

In Linux, both `fork()` and `pthread_create()` are implemented using the system call `clone()`.

Pthreads in Linux

How do we implement threads in Linux?

In Linux, both `fork()` and `pthread_create()` are implemented using the system call `clone()`.

What is clone()?

clone() - from man pages

```
Unlike fork(2), clone() allows the child process to share parts of
its execution context with the calling process, such as the memory
space, the table of file descriptors, and the table of signal
handlers.
```

clone() - from man pages

```
Unlike fork(2), clone() allows the child process to share parts of
its execution context with the calling process, such as the memory
space, the table of file descriptors, and the table of signal
handlers.
```

The system call `clone()` allows us to define how much should be shared:

- fork(): copy table of file descriptors, copy memory space and signal handlers i.e a perfect copy
- pthread_create(): share table of file descriptors and memory, copy signal handlers

clone() - from man pages

`Unlike fork(2), clone() allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.`

The system call `clone()` allows us to define how much should be shared:

- fork(): copy table of file descriptors, copy memory space and signal handlers i.e a perfect copy
- pthread_create(): share table of file descriptors and memory, copy signal handlers

Using clone() directly you can pick and choose of more than twenty parameters what the clone should share.

Thread Local Storage (TLS)

All threads have their own stack, the heap is shared.

Thread Local Storage (TLS)

All threads have their own stack, the heap is shared.

Would it not be nice to have some *thread local storage*?

Thread Local Storage (TLS)

All threads have their own stack, the heap is shared.

Would it not be nice to have some *thread local storage*?

```
__thread int local = 42;
```

TLS implementation

```
__thread int local = 0;

int global = 1;

void *hello(void *name) {

  int stk = 2;
  int sum = local + global + stk;
}
```

TLS implementation

```c
__thread int local = 0;

int global = 1;

void *hello(void *name) {

  int stk = 2;
  int sum = local + global + stk;
}
```

```
pushq   %rbp
movq    %rsp, %rbp
movq    %rdi, -24(%rbp)
movl    $2, -8(%rbp)
movl    %fs:local@tpoff, %edx
movl    global(%rip), %eax
addl    %eax, %edx
movl    -8(%rbp), %eax
addl    %edx, %eax
movl    %eax, -4(%rbp)
nop
popq    %rbp
ret
$
```

The TLS is referenced using the segment selector `fs:`.

The TLS is referenced using the segment selector `fs:`.

When we change thread, the kernel sets the `fs` selector register.

The TLS is referenced using the segment selector `fs:`.

When we change thread, the kernel sets the `fs` selector register.

The TLS has an original copy that is copied by each thread (even the mother thread) before any write operations.

remember segmentation

The TLS is referenced using the segment selector `fs:`.

When we change thread, the kernel sets the `fs` selector register.

The TLS has an original copy that is copied by each thread (even the mother thread) before any write operations.

You can take an address of a TLS structure and pass it to another thread.

remember segmentation

The TLS is referenced using the segment selector `fs:`.

When we change thread, the kernel sets the `fs` selector register.

The TLS has an original copy that is copied by each thread (even the mother thread) before any write operations.

You can take an address of a TLS structure and pass it to another thread.

Summary

Summary

- Concurrency vs parallelism?

Summary

- Concurrency vs parallelism?
- What is a thread?

Summary

- Concurrency vs parallelism?
- What is a thread?
- What do threads of process share?

Summary

- Concurrency vs parallelism?
- What is a thread?
- What do threads of process share?
- Sequential Consistency vs Total Store Order

Summary

- Concurrency vs parallelism?
- What is a thread?
- What do threads of process share?
- Sequential Consistency vs Total Store Order
- Threads in kernel or user space?

Summary

- Concurrency vs parallelism?
- What is a thread?
- What do threads of process share?
- Sequential Consistency vs Total Store Order
- Threads in kernel or user space?
- Threads in GNU/Linux and `clone()`.

Summary

- Concurrency vs parallelism?
- What is a thread?
- What do threads of process share?
- Sequential Consistency vs Total Store Order
- Threads in kernel or user space?
- Threads in GNU/Linux and `clone()`.
- What is Thread Local Storage?

Summary

- Concurrency vs parallelism?
- What is a thread?
- What do threads of process share?
- Sequential Consistency vs Total Store Order
- Threads in kernel or user space?
- Threads in GNU/Linux and `clone()`.
- What is Thread Local Storage?

Summary

- Concurrency vs parallelism?
- What is a thread?
- What do threads of process share?
- Sequential Consistency vs Total Store Order
- Threads in kernel or user space?
- Threads in GNU/Linux and `clone()`.
- What is Thread Local Storage?