Concurrency

Johan Montelius

KTH

2019
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?
Concurrency: (the illusion of) happening at the same time.
Concurrency: (the illusion of) happening at the same time.

A property of the programming model.
What is concurrency?

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

A property of the programming model.

Why would we want to do things concurrently?
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?
Parallelism: the ability to do several things at the same time.
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?
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.
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.
If concurrent activities are not manipulating a shared resource then it’s not a 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.
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
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
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
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, ...
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

process
A thread

code

process
A thread

code

process

data/heap
A thread

code

process

stack

data/heap
A thread

- Code
- Process
- Stack
- IP
- Data/Heap
A thread

- Code
- Process
- Stack
- IP
- Data/Heap
A thread

code

IP

stack

data/heap

process
A thread

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

code

IP

stack

data/heap

process
A thread

code

thread

IP

stack

data/heap

process
A thread

- Code
- Stack
- IP
- Thread
- Process
- Data/Heap
A thread
Virtual memory layout
Virtual memory layout
Virtual memory layout

code
(.text)

kernel
Virtual memory layout

code (.text)  data  kernel
Virtual memory layout

code (.text)  data  heap  kernel
Virtual memory layout

code (.text)  data  heap  →  ↓  stack  ↓  kernel
Virtual memory layout

code (.text)  data  heap  stack  stack  kernel
#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;
}
What is the problem?
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.
The CPU uses caches to improve performance, a cache protocol must provide *coherence*. 
The CPU uses caches to improve performance, a cache protocol must provide *coherence*.

- All write operations to a single memory location:
  - are atomic,
The CPU uses caches to improve performance, a cache protocol must provide\textit{coherence}.

- All write operations to a single memory location:
  - are atomic,
  - performed in program order and
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 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
What is the expected outcome of an execution?
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:
The outcome is the same as if all the operations of the program were executed:

as atomic operations in some sequence,
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.
int loop = 10;
int count = 0;

void *hello(void *) {
    for(int i = 0; i < loop; i++) {
        count++;
    }
}
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)
    j1 .L3
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;
        }
    }
}
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;
        }
    }
}
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;
        }
    }
}
```c
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;
        }
    }
}
```
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;
        }
    }
}
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;
        }
    }
}
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;
    }
  }
}
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;
        }
    }
}
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;
    }
}

Thread 1

Thread 2

read count

1 a 7 c 0 b
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;
    }
  }
}
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;
        }
    }
}
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;
    }
  }
}
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;
        }
    }
}
Modern CPUs 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.
Modern CPU:s do not provide *sequential consistency*, they only provide *Total Store Order*. 
Modern CPU:s do not provide *sequential consistency*, they only provide *Total Store Order*.

- Write operations are performed in a total order.
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,
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.
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.*
WARNING: the following sequence contains scenes that some viewers may find disturbing.
WARNING: the following sequence contains scenes that some viewers may find disturbing.

P1

a

0

P2

0

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

\[ a = 1 \]
WARNING: the following sequence contains scenes that some viewers may find disturbing.

\[ a = 1 \]

\[ b = 1 \]
WARNING: the following sequence contains scenes that some viewers may find disturbing.

P1

a = 1
read b

P2

b = 1
read a

a = 1

0

0
WARNING: the following sequence contains scenes that some viewers may find disturbing.

\[
\begin{align*}
\text{P1} & : \quad a = 1 \\
\text{P2} & : \quad b = 1
\end{align*}
\]

**Sequence:**
- **P1:**
  - Read b
  - 0
- **P2:**
  - 0
  - Read a
  - a = 1

**Diagram:**
- Two processes, P1 and P2, interacting through variables a and b.
- a is initialized to 1 by P1.
- b is initialized to 1 by P2.
- Process P1 reads variable b.
- Process P2 reads variable a.

**Timeline:**
1. P1 reads b.
2. P2 reads a.
3. P1 sets a to 1.
4. P2 sets b to 1.

This diagram illustrates a race condition in concurrent programming, where the order in which variables are read and written can lead to unexpected outcomes.
WARNING: the following sequence contains scenes that some viewers may find disturbing.

\[ a = 1 \]

\[ b = 1 \]

read b

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

P1

\( a = 1 \)

read b

\[ 0 \]

P2

\( b = 1 \)

read a

\[ 0 \]
WARNING: the following sequence contains scenes that some viewers may find disturbing.
WARNING: the following sequence contains scenes that some viewers may find disturbing.

```
P1

a = 1

read b

0 1

P2

a 0

read a

0 b

b = 1

1 0
```
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.
TGH - Thank God for Hardware
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.
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.*
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

Next week.
How to implement threads

threads in user space

threads in kernel space
How to implement threads

threads in user space

threads in kernel space

kernel

kernel
How to implement threads

threads in user space

process

threads in kernel space

kernel
How to implement threads

threads in user space

threads in kernel space

process
How to implement threads

threads in user space

threads in kernel space

process

kernel

kernel
How to implement threads

threads in user space

scheduler  process

kernel

threads in kernel space

kernel
How to implement threads

threads in user space

scheduler

process

threads in kernel space

kernel

kernel
How to implement threads

threads in user space

threads in kernel space

scheduler

process

kernel

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

- scheduler
- processes

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

scheduler

process

processes

kernel

kernel
How to implement threads

threads in user space

threads in kernel space

scheduler

process

processes

kernel

kernel
How to implement threads

threads in user space

threads in kernel space

scheduler
process

processes

kernel

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

scheduler

process

processes

kernel

kernel
How to implement threads

threads in user space

threads in kernel space

scheduler

process

processes
Threads in user space:

- + You can change scheduler.
Threads in user space:

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

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?
Threads in user space:

- + You can change scheduler.
- + Very fast task switching.
- - If the process is suspended, all threads are.
Threads in user space:
- You can change scheduler.
- Very fast task switching.
- If the process is suspended, all threads are.
- A process cannot 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 cannot 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.

Which approach is taken by GNU/Linux?
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.
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 cannot 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.
 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?
How is this handled in high level languages?
How is this handled in high level languages?

- Java: each Java thread mapped to one operating system thread.
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.
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.
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?
#include <pthread.h>
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
#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**.
#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.
#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 *.
#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 -pthread.
How do we implement threads in Linux?
How do we implement threads in Linux?

In Linux, both fork() and pthread_create() are implemented using the system call clone().
How do we implement threads in Linux?

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

What is `clone()`?
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.
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
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.
All threads have their own stack, the heap is shared.
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;
__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;
}
```

```asm
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
```

30 / 32
The TLS is referenced using the segment selector $fs:$. 

remember segmentation
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.
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.
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

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?
Concurrency vs parallelism?
- Concurrency vs parallelism?
- What is a thread?
Summary

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

What is a thread?

What do threads of process share?

Sequential Consistency vs Total Store Order
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?
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?
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?