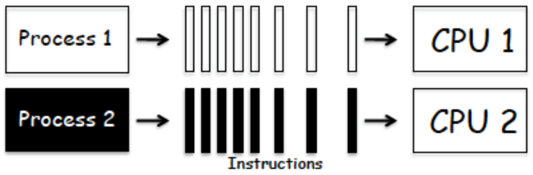
L2

Multiprocessing and Multithreading

C++ Threads

Multiprocessing, parallelism

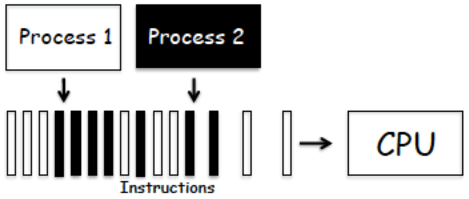
- Many of today's computations can take advantage of
- multiple processing units (through multi-core processors):



- Terminology:
- Multiprocessing : the use of more than one processing unit in a system
- Parallel execution: processes running at the same time

Multitasking, concurrency

- Even on systems with a single processing unit we may give the illusion of that several programs run at once
- The OS switches between executing different tasks



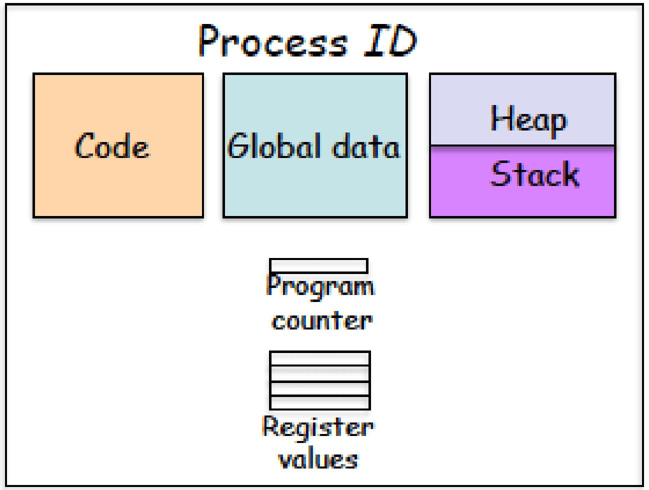
- Terminology:
- Interleaving: several tasks active, only one running at a time
- Multitasking: the OS runs interleaved executions
- Concurrency: multiprocessing, multitasking, or any combination

Processes

- A (sequential) program is a set of instructions
- A process is an instance of a program that is being executed

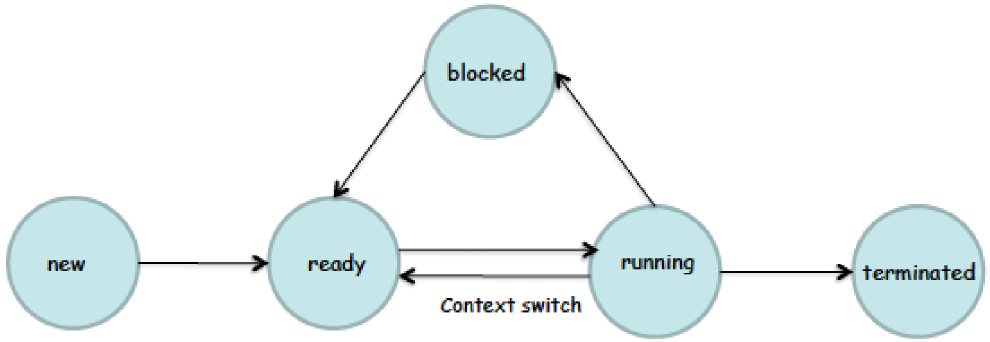
Operating system processes

- How are processes implemented in an operating system?
- Structure of a typical process:
    - Process identifier: unique ID of a process.
    - Process state: current activity of a process.
    - Process context: program counter, register values.
    - Memory: program text, global data, stack, and heap.



The scheduler

- A system program called the scheduler controls which processes are running; it sets the process states:
    - new: being created.
    - running: instructions are being executed.
    - blocked: currently waiting for an event.
    - ready: ready to be executed, but not been assigned a processor yet.
    - terminated: finished executing.

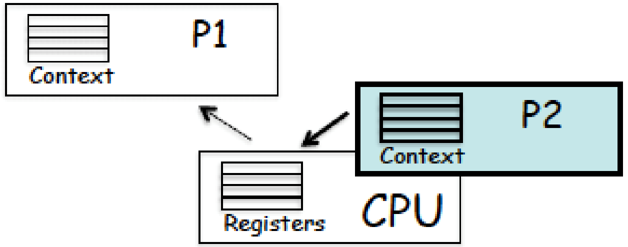


Blocked processes

- A process can get into state blocked by executing special program instructions
- When blocked, a process cannot be selected for execution
- A process gets unblocked by **external events** which set its state to ready again

The context switch

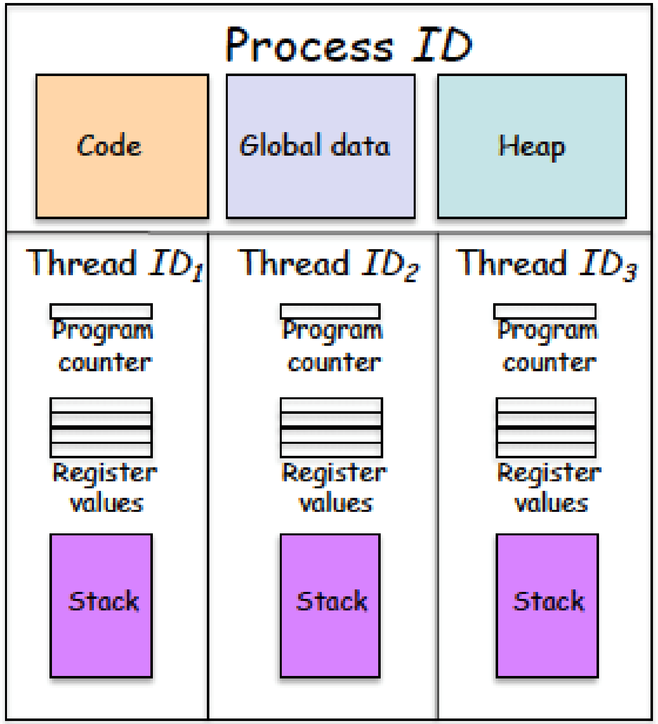
- The swapping of processes on a processing unit by the scheduler is called the context switch



- Scheduler actions when switching processes P1 and P2:
    - P1.state := ready
    - Save register values as P1's context in memory
    - Use context of P2 to set register values
    - P2.state := running

Threads

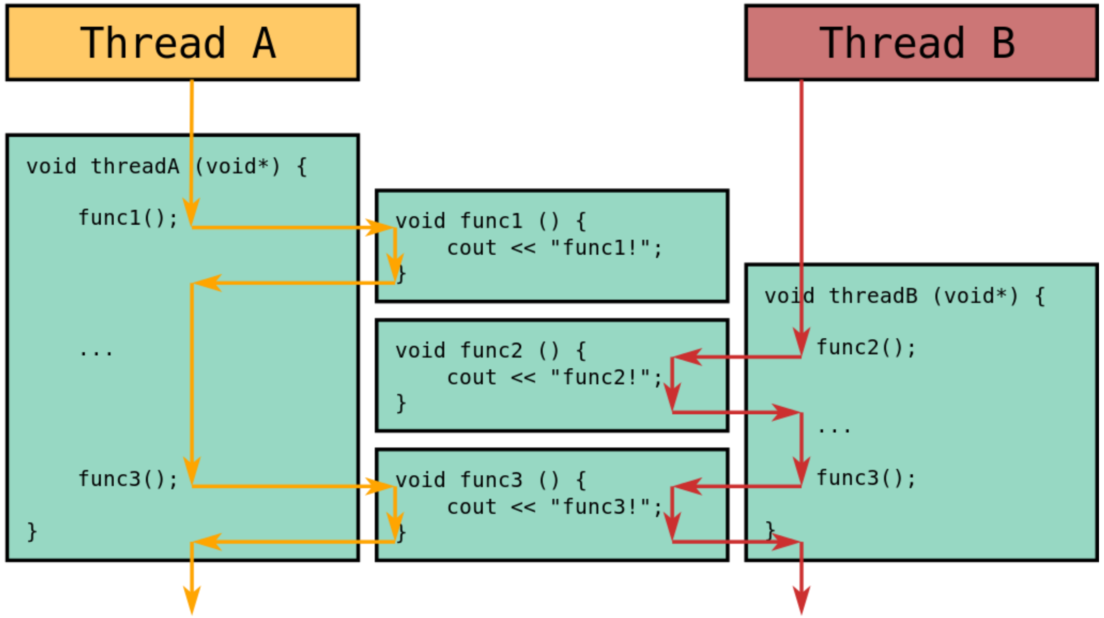
- Make programs concurrent by associating them with threads
- A thread is a part of an operating system process
- Components private to each thread
    - Thread identifier
    - Thread state
    - Thread context
    - Memory: only stack
- Components shared with other threads:
    - Program text
    - Global data
    - Heap



C++ threads (C++11 and further)

- std::thread class
- Each instance of this object
  - represents
  - wraps
  - manages

  a single execution thread.



Source: https://kholdstare.github.io/technical/2012/08/21/objects-and-threads-in-cpp-1.html

- all code is accessible to any thread
- two threads could be executing the same function at the same time.

Simple examples

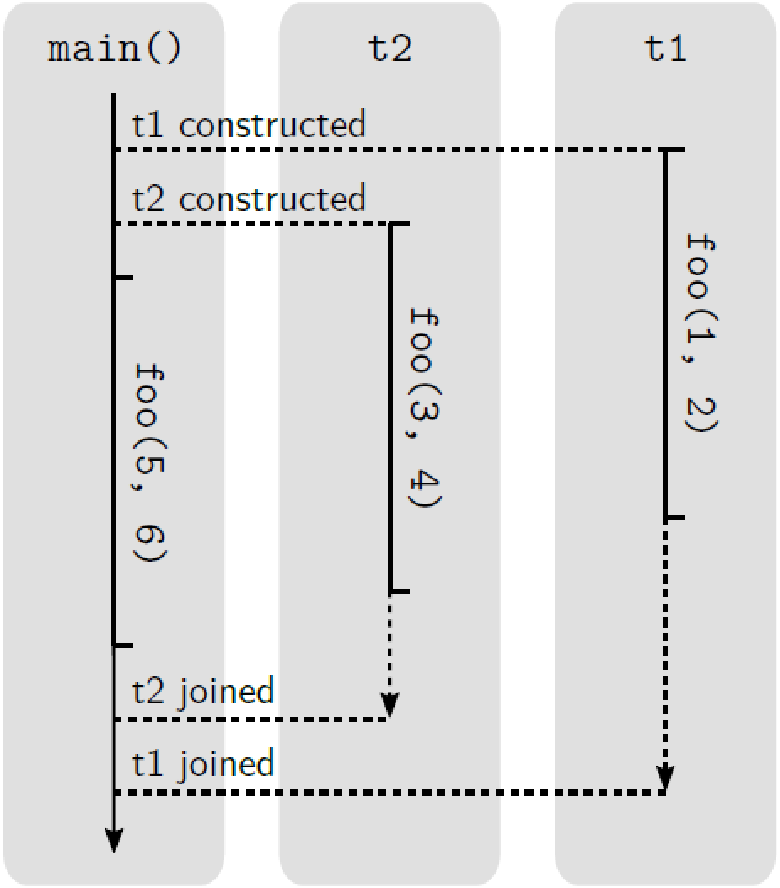
```
#include <iostream>
#include <thread>
void thread_function()
{
   std::cout << "thread function\n";
}
int main() {
   std::thread t(thread_function);   // t starts running
   std::cout << "main thread\n";
   t.join();   // main thread waits for the thread t to finish
   return 0;
}
```

```
#include <iostream>
#include <thread>
#include <string>
void thread_function(std::string s)
{
   std::cout << "thread function ";
   std::cout << "message is = " << s << std::endl;
}
int main() {
   std::string s = ”HPC&BDA";
   std::thread t(thread_function, s );
   std::cout << "main thread message = " << s << std::endl;
   t.join();
//t.detach();
   return 0;
}
```

Thread creation and destruction

```
#include <thread>
voidfoo(int a, int b);
int main(){
	//Pass a function and args
	std::threadt1(foo, 1, 2);
	//Pass a lambda
	std::threadt2([](){ foo(3,4); }); foo(5, 6);
	t2.join();
	t1.join();
}
```

join is mandatory
unless detach is called



Other useful functions

- std::this_thread::sleep_for()

  : Stop the current thread for a given amount of time

- std::this_thread::sleep_until()

  : Stop the current thread until a given point in time

- std::this_thread::yield()

  : Let the operating system schedule another thread

- std::this_thread::get_id()

  : Get the (operating-system-specific) id of the current thread

- std::thread::detach()

  : Separates the thread of execution from the thread object, allowing execution to continue independently.

Sharing data

Global Variables

- All global and static variables that are initialized at compile time can be accessed by threads.
- Since the threads should know the addresses for them.

Passing By Value vs by Reference

- All parameters passed to a function when starting a thread are passed by value!
- For passing by reference, we need to explicitly wrap the arguments in std::ref() .
- Because the thread functions can't return anything, passing by reference is the only way to properly get data out of a thread without using global variables.

- Example:

```
void ref_function(int &a, int b) {
        int val;
        std::thread ref_function_thread(ref_function, std::ref(val),  std::ref(val));
}
```

static and thread_local Variables

```
void method() {
  static int var = 0;
  var++;
}
```

- This does NOT create a separate instance of the static variable per thread instance !!!!
  This is because static variables are initialized once when the compiler goes over their declaration.
- In order to have 'static' variables that are static within the scope of each particular thread, we must use thread_local variables instead.
  Then each thread will have its own version of the static variable, and the static variable will only be destroyed on thread exit.

```
void method() {
  thread_local int var = 0;
  var++;
}
```

race condition

- **race condition** (**race hazard)**
- *a condition of a program where its behavior depends on relative timing or interleaving of multiple threads or processes*

- The *race condition* term was introduced before 1955,
  [e.g. David A. Huffman's doctoral thesis "The synthesis of sequential switching circuits”, 1954]

Race condition

```
void threadfunc(unsigned* x) {
//????????????
   for (int i = 0; i != 10000000; ++i) {
      *x += 1;
   }
}
```

```
int main() {
   std::thread th[4];
   unsigned n = 0;
   for (int i = 0; i != 4; ++i) {
      th[i] = std::thread(threadfunc, &n);
   }
   for (int i = 0; i != 4; ++i) {
      th[i].join();
   }
   printf("%u\n", n);
}
```

A data race occurs when two or more threads access the same variable concurrently, and at least one of the accesses is a write.

critical race condition vs. non-critical race condition

- *critical race condition* = when the order may change the final state

- *non-critical race condition* = when the order do not change the final state

T1:

```
update (){
        a=a+1;
        }
```

T2:

```
update (){
                b=a*2;
}
```

Concurrency

Initialization of global variables

Thread ID

x := 0			
P1		P2	
1 2	x := 0 x := x + 1	1	x := 2

Line numbers

Code of concurrently executed threads

Execution Variants

P1	1	x := 0	x = 0
P2	1	x := 2	x = 2
P1	2	x := x + 1	x = 3

Instruction executed with Thread ID and line number

Variable values after execution of the code on the line

P2	1	x := 2	x = 2
P1	1	x := 0	x = 0
P1	2	x := x + 1	x = 1

P1	1	x := 0	x = 0
P2	1	x := 2	x = 2
P1	2	x := x + 1	x = 3

P1	1	x := 0	x = 0
P1	2	x := x + 1	x = 1
P2	1	x := 2	x = 2

Atomic instruction

- Levels of atomicity

Ex: x:= x + 1

Execution:

```
temp := x                          LOAD REG, x
temp := temp + 1                   ADD  REG, #1
x := temp                              STORE REG, x
```

Variante de executie

x := 0			
P1		P2	
1 2 3 4	x := 0 temp := x temp := temp + 1 x := temp	1	x := 2

- "interleaving"

P1	1	x := 0	x = 0
P1	2	temp := x	x = 0, temp = 0
P2	1	x := 2	x = 2, temp = 0
P1	3	temp := temp + 1	x = 2, temp = 1
P1	4	x := temp	x = 1, temp = 1

Counter -> Details – register level

get this.count from memory into register

add value to register

write register to memory

- Example of interleaving

this.count = 0;

A: reads this.count into a register (0)

B: reads this.count into a register (0)

B: adds value 2 to register

B: writes register value (2) back to memory. this.count now equals 2

A: adds value 3 to register

A: writes register value (3) back to memory. this.count now equals 3

Race Conditions & Critical Sections

- ***A Critical Section*** = code segment where a data-race may occur

```
public class Counter {
    protected long count = 0;
    public void add(long value){
        this.count = this.count + value;
    }
}
```

If an object Counter is used by more threads!

=> Not *thread-safe!*

- Method add() is an example of critical section.

Atomics

- Processors have special, *atomic* instructions that always execute without racing with any other processor's accesses.
- These instructions are "indivisible, i.e., the processor does not internally decompose atomic instructions into smaller micro-code instructions.
- Atomic instructions are generally slower and more expensive (in terms of energy) to execute than normal instructions.

```
void threadfunc(std::atomic* x) {
    for (int i = 0; i != 10000000; ++i) {
        x->fetch_add(1);
        // `*x += 1` and `(*x)++` also work!
    }
}
```

Uses processor lock-prefixed instructions !

Mutual exclusion - Mutexes

- **Mutual exclusion** means that at most one thread accesses the shared data at a time!

```
std::mutex mutex;

void threadfunc(unsigned* x) {
    for (int i = 0; i != 10000000; ++i) {
        mutex.lock();
        *x += 1;
        mutex.unlock();
    }
}
```

Where should be declared?

- A mutex (a kind of a data structure) has an internal state (denoted by state), which can be either locked or unlocked. The semantics of a mutex object is as follows:
- Upon initialization, state = unlocked.
- When a thread call:
  - mutex::lock() method: waits until state becomes unlocked, and then atomically sets state = locked. Note the two steps shall complete in one atomic operation.
  - mutex::unlock() method: asserts that state == locked, then sets state = unlocked.
- Binary semaphore

See also std::unique_lock
a RAII wrapper for exclusive locking

Recursive Mutexes

```
#include <mutex>
std::mutex mutex;
void bar() {
    std::unique_lock lock(mutex);
    // do some work...
}
void foo() {
    std::unique_lock lock(mutex);
    // do some work...
    bar(); // will deadlock
}
```

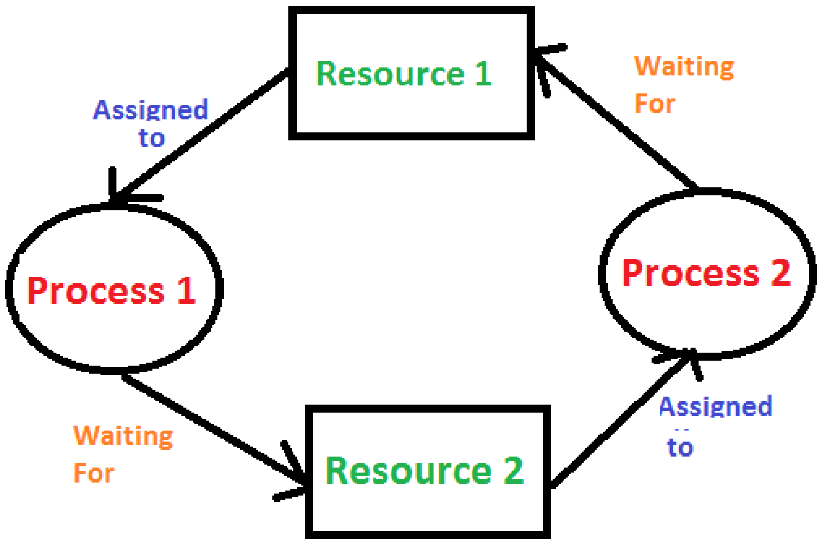
- The code will deadlock since std::mutex can be locked at most once

Recursive Mutexes

- std::recursive_mutex implements recursive ownership semantics
- The same thread can lock an std::recursive_mutex multiple times without blocking
- Other threads will still block if an std::recursive_mutex is currently locked
- Can be used with std::unique_lock just like a regular std::mutex
- Useful for functions that call each other and use the same mutex.

```
#include <mutex>
std::recursive_mutex mutex;
void bar() {
        std::unique_lock lock(mutex);
}
void foo() {
        std::unique_lock lock(mutex);
        bar(); // OK, will not deadlock
}
```

Deadlocks




```
std::mutex m1, m2, m3;
void threadA() {
// ???
std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
void threadB() {
// ???
std::unique_lock l3{m3}, l2{m2}, l1{m1};
}
```

Possible deadlock scenario

1. threadA() acquires locks on m1 and m2
2. threadB() acquires lock on m3
3. threadA() waits for threadB() to release m3
4. threadB() waits for threadA() to release m2

Solution

Deadlocks can be avoided by always locking mutexes in a *globally* consistent order

- Ensures that one thread always “wins”.
- Maintaining a globally consistent locking order requires considerable developer discipline.
- Maintaining a globally consistent locking order may not be possible at all.

```
std::mutex m1, m2, m3;
void threadA() {
    // OK, will not deadlock
    std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
void threadB() {
    // OK, will not deadlock
    std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
```

Condition variables

- A condition variable is a synchronization primitive that allows multiple threads to wait until an (arbitrary) condition becomes true.
- A condition variable uses a mutex to synchronize threads.
- Threads can *wait* on or *notify* the condition variable.
- When a thread waits on the condition variable, it blocks until another thread notifies it.
- If a thread waited on the condition variable and is notified, it holds the mutex.
- A notified thread must check the condition explicitly because *spurious wake-ups* can occur

std::condition_variable header <condition_variable> which has the following member functions:

- wait(): Takes a reference to a std::unique_lock that must be locked by the caller as an argument, unlocks the mutex and waits for the condition variable
- notify_one(): Notify a single waiting thread, mutex does not need to be held by the caller
- notify_all(): Notify all waiting threads, mutex does not need to be held by the caller

Worker threads example

```
#include <condition_variable>

#include <iostream>

#include <thread>


std::mutex a_mutex;

std::condition_variable condVar;

bool dataReady = false;


void waitingForWork(){
        std::cout << "Waiting\n ";
        std::unique_lock<std::mutex>  lck(a_mutex);
         condVar.wait(lck, []{ return dataReady; } );
        std::cout << "Running\n " ;
}
```

```
void setDataReady(){
                {
                std::lock_guard<std::mutex> lck(a_mutex);
                dataReady = true;
                }
                std::cout << "Data prepared\n";
                condVar.notify_one();                 }


int main(){

        std::thread t1(waitingForWork);

        std::thread t2(setDataReady);

        t1.join();  t2.join();
}
```