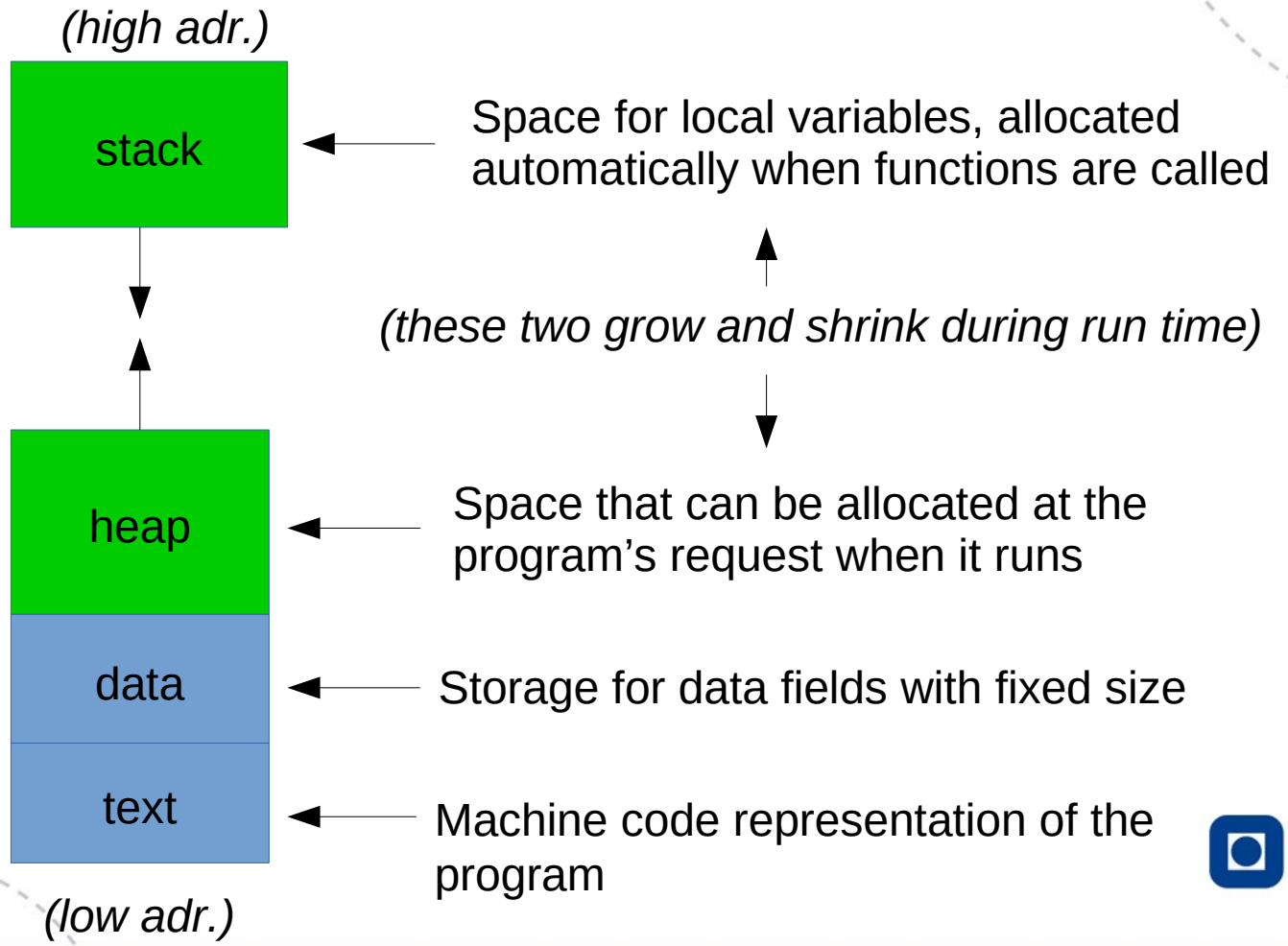
NTNU – Trondheim
Norwegian University of
Science and Technology

Pthreads introduction

Jan.Christian.Meyer@ntnu.no

Back to the process image

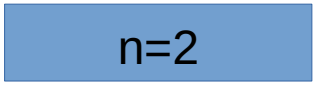


The purpose of the stack

- The stack supports function calls:

```
fib (n) {
  if (n>1)
    return fib(n-1) + fib(n-2)
  else
    return 1
}

main() {
  print fib(2)
}
```

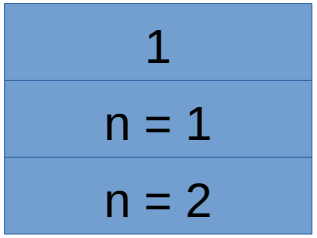


Stack

Anatomy of function calls

```
fib (n) {
  if (n>1)
    return fib(n-1) + fib(n-2)
  else
    return 1
}

main() {
  print fib(2)
}
```

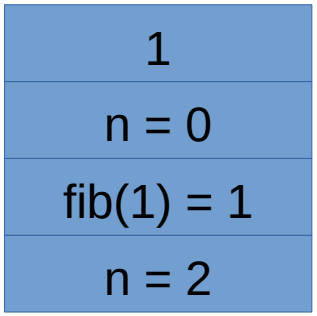


Stack

Anatomy of function calls

```
fib (n) {
  if (n>1)
    return fib(n-1) + fib(n-2)
  else
    return 1
}

main() {
  print fib(2)
}
```

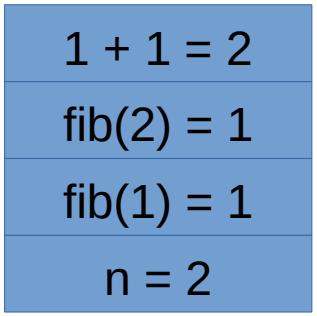


Stack

Anatomy of function calls

```
fib (n) {
  if (n>1)
    return fib(n-1) + fib(n-2)
  else
    return 1
}

main() {
  print fib(2)
}
```



Stack

Anatomy of function calls

```
fib (n) {
  if (n>1)
    return fib(n-1) + fib(n-2)
  else
    return 1
}

main() {
  print fib(2)
}
```

2

Stack

...and the number “2” appears on screen.

The purpose of the heap

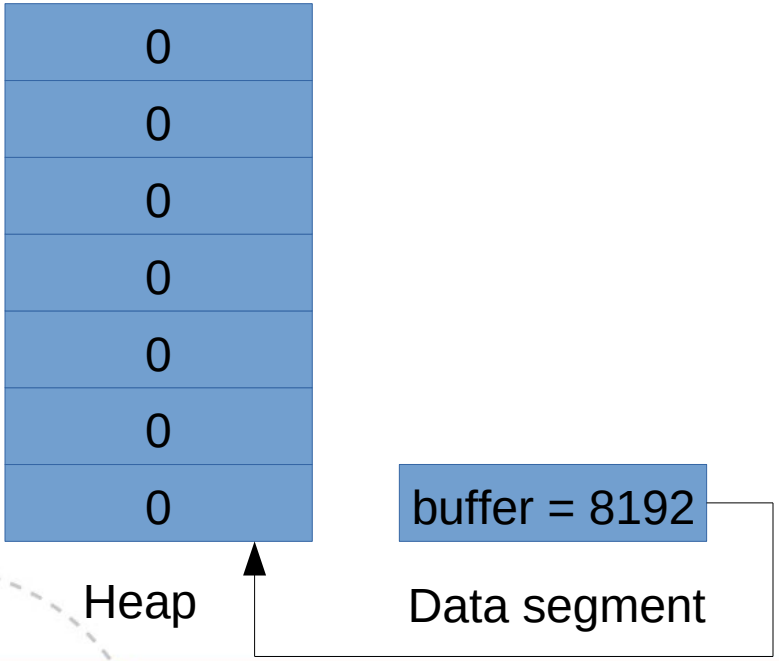
- The heap supports explicit program allocations:

```
*buffer = 0
main() {
  buffer = allocate (7)
  buffer[0] = ‘H’
  buffer[1] = ‘e’
  buffer[2] = ‘l’
  buffer[3] = ‘l’
  buffer[4] = ‘o’
  buffer[5] = ‘!’
  buffer[6] = 0
  print ( buffer )
  deallocate( buffer )
}
```

Heap

Data segment

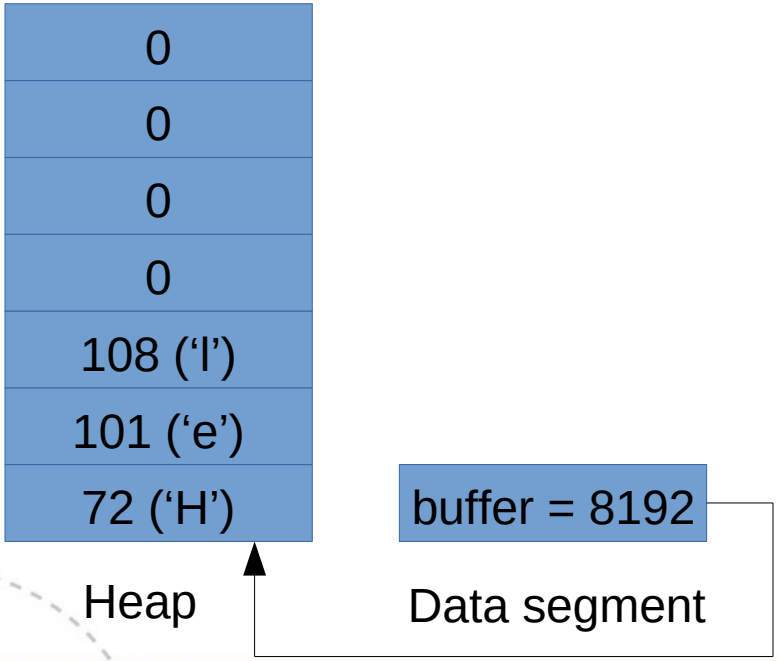
Anatomy of an explicit allocation




```
*buffer = 0
main() {
  buffer = allocate (7)
  buffer[0] = 'H'
  buffer[1] = 'e'
  buffer[2] = 'l'
  buffer[3] = 'l'
  buffer[4] = 'o'
  buffer[5] = '!'
  buffer[6] = 0
  print ( buffer )
  deallocate( buffer )
}
```

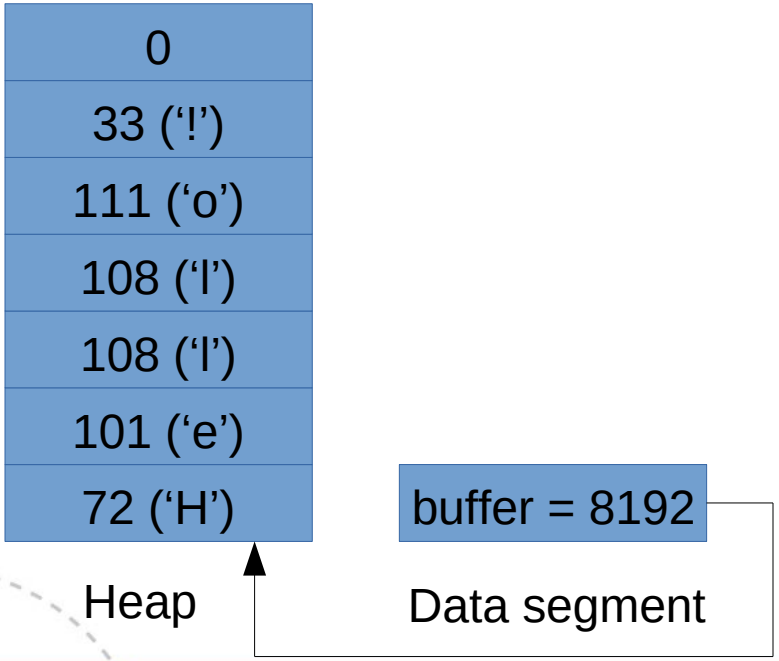
(this number is an arbitrary address inside the range of the heap)

Anatomy of an explicit allocation



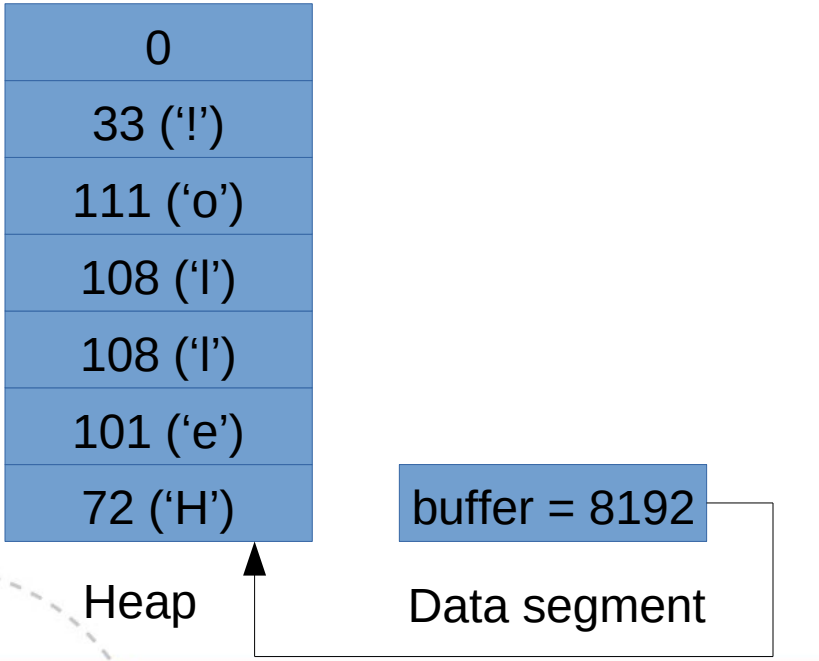

```
*buffer = 0
main() {
  buffer = allocate (7)
  buffer[0] = ‘H’
  buffer[1] = ‘e’
  buffer[2] = ‘l’
  buffer[3] = ‘l’
  buffer[4] = ‘o’
  buffer[5] = ‘!’
  buffer[6] = 0
  print ( buffer )
  deallocate( buffer )
}
```

Anatomy of an explicit allocation




```
*buffer = 0
main() {
  buffer = allocate (7)
  buffer[0] = ‘H’
  buffer[1] = ‘e’
  buffer[2] = ‘l’
  buffer[3] = ‘l’
  buffer[4] = ‘o’
  buffer[5] = ‘!’
  buffer[6] = 0
  print ( buffer )
  deallocate( buffer )
}
```

Anatomy of an explicit allocation




```
*buffer = 0
main() {
  buffer = allocate (7)
  buffer[0] = ‘H’
  buffer[1] = ‘e’
  buffer[2] = ‘l’
  buffer[3] = ‘l’
  buffer[4] = ‘o’
  buffer[5] = ‘!’
  buffer[6] = 0
  print ( buffer )
  deallocate( buffer )
}
```

“Hello!” appears on screen

dheim
versity of
chnology

Anatomy of an explicit allocation

```
*buffer = 0
main() {
  buffer = allocate (7)
  buffer[0] = ‘H’
  buffer[1] = ‘e’
  buffer[2] = ‘l’
  buffer[3] = ‘l’
  buffer[4] = ‘o’
  buffer[5] = ‘!’
  buffer[6] = 0
  print ( buffer )
  deallocate( buffer )
}
```

buffer = 8192

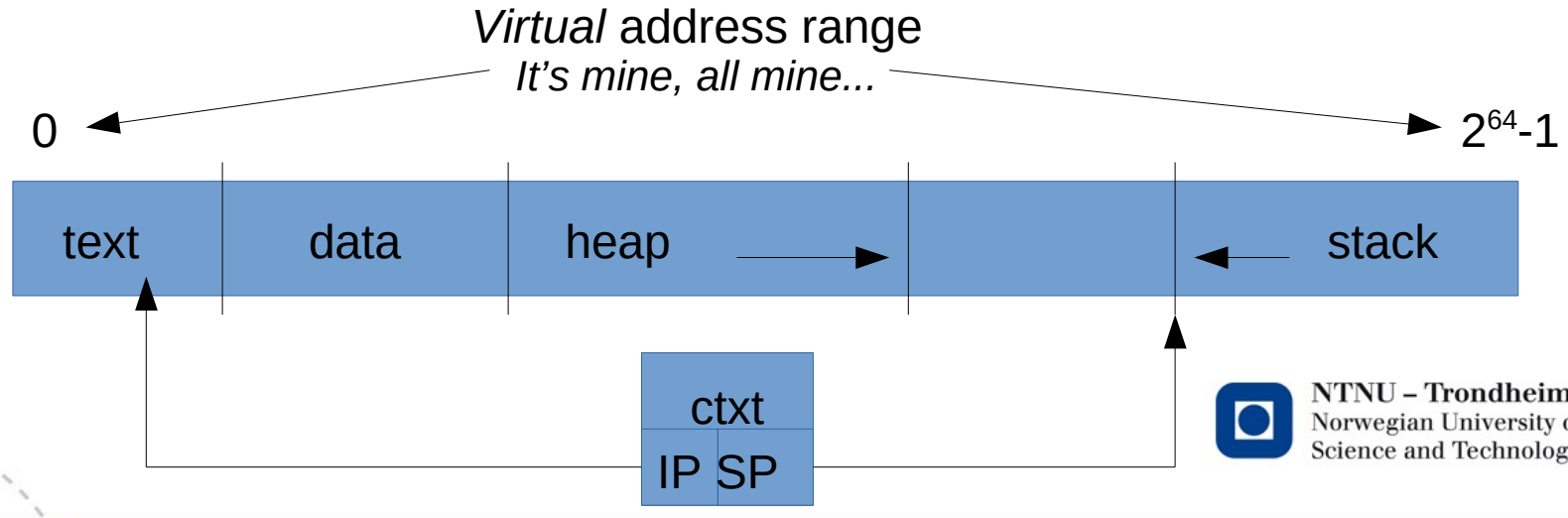
Heap

Data segment

Memory at 8192 can be re-used for a different allocation later

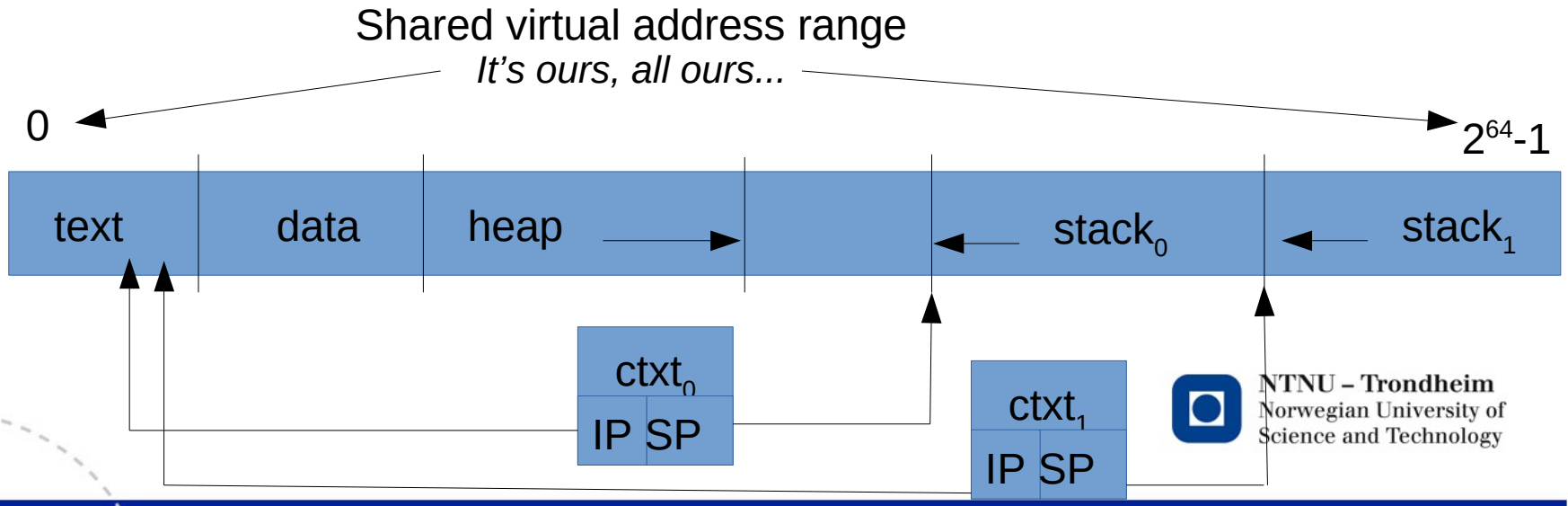
One process and its thread

- In a sequential program, we follow one path through the program text
- It's enough to have 1 of everything we have examined so far:



One process and two threads

- In a threaded program, we follow several independent paths through the program text
- This requires each thread to track its position, and the state of its own stack
- Data and heap segments are shared between these contexts, so if one thread alters a variable stored on the heap, that variable changes value for the other thread also



NTNU – Trondheim
Norwegian University of Science and Technology

The concept of concurrency

- The central idea is that we can improve programs by relaxing the restriction that their statements must only ever be executed in 1 specific order
- This can be done in many ways, and at many different levels of abstraction
  - *“one step of the program”* might mean one database transaction, one web page served, one image rendered, *etc. etc.*
- When our goal is to compute some number, individual machine operations make a natural unit step, so that we can rearrange the calculation in detail

Software that enables concurrency

- Threads are a software construct
  - They only augment the program with information about which steps depend on a fixed sequence, and which ones don't
- The hardware and O/S get to decide whether or not to do anything with that information
  - Among the many valid execution orders of a threaded program, there is still the option of running all parts one after the other, like a sequential program
- For (TDT4200) *parallel computing*, we would like to have a maximal number of operations executed simultaneously

The hardware that executes our program

- Mapping of threads to physical processing cores is trivial when you have the same number of threads and cores
- *Oversubscription* can occasionally be useful
  - It can keep the cores occupied when some threads need to block because they're waiting for resources
  - It is absolutely instrumental to using GPUs effectively, which we can return to later on
- For applications where the threads are constantly calculating something, multiple threads per core only delay the computation by adding the overhead of scheduling switches between them

Shared memory parallelism

- A set of thread contexts map onto a set of processing cores
  - Often a 1-1 mapping, but not by necessity
- Each thread has its own private work space
  - That is, the local variables in the stack space of its function call(s)
- All threads have a shared, global workspace
  - The data and heap segments can be used for inter-thread communication and collaboration

What could possibly go wrong?

- Any shared work space can become an arena for conflicts
  - These are often caused by disagreements over the management of shared resources
- Parallel computers don’t grow sentimental about such conflicts, but they can still have them
  - What if two (or more) independent threads simultaneously try to assign a number to a shared memory location?
- We must decide on some kind of contract or policy to resolve these situations

Theoretically speaking

- The PRAM model* is a minimialistic, abstract machine which proposes 4 candidate policies:
  - *Common Value* admits the assignment and allows execution to proceed if (and only if) all attempts to make it are trying to assign the same value
  - *Arbitrary Value* assigns the value from one of the attempts, and the program must assume that it can be any of them
  - *Priority* requires each thread to have a ranking among its peers, and the highest ranked assignment wins
  - *Reduction* assigns the result of applying some commutative operation to all the attempts (sum, product, logical AND, *etc.*)

* *Introduction to the Theory of Complexity*, D.P. Bovet & P. Crescenzi, Prentice Hall Europe, 1994

Practically speaking

- The only policy I have ever seen realized in actual hardware is the *Arbitrary Value* policy
- The value is simply determined by handling the assignments in the order they arrive at the memory banks
- This makes the program
  - Behave non-deterministically in the best case
  - Crash and Burn™ in the worst (and by far most common) case

The *Read-Modify-Write* cycle (again)

- We talked about this in the context of the von Neumann computer, but I'm repeating it now
- A minimal statement in a programming language might look like this:

```
total_sum += my_value;
```

- After the compiler is finished with it, we get a short sequence of smaller operations, *e.g.*

```
load my_value into register B
load total_sum into register A
add register B to register A
store register A in total_sum
```

- Programs produce this pattern all the time

Off to the races

- Suppose we have two threads with different numbers (4,6) as their private versions of my_value
- We want their sum (10) in a shared location
- If we try to run their `total_sum += my_value` statements simultaneously

  (or even just *almost* simultaneously)

  the answer comes out wrong:

Time step	total_sum	my_value #1	Thread #1	my_value #2	Thread #2
1	0	4	B ← 4	6	
2	0	4	A ← 0	6	B ← 6
3	0	4	A ← A+B = 4	6	A ← 0
4	4	4	total_sum ← 4	6	A ← A+B = 6
5	6	4		6	total_sum ← 6

Race conditions

- This sort of thing is called a *race condition*
  - So named because just as in a horse race, the result is determined by which participant finishes first
- It can occur when there is contention for any shared thing that requires exclusive access to end well
- When the contested object is the value of a datum in memory, we can also call them *data races*
- The code segment that must not overlap is called a *critical section*

Instruments of protection

- Several mechanisms have been invented to ensure exclusive access in a critical section
  - Atomic operations
  - Load Linked & Store Conditional instructions
  - Mutex (lock) and semaphore data structures
  - Higher-level programming model constructs
- All of these need some hardware support for their underlying operations
  - Well, *almost*…
  - There is actually a short and unreadable 1965 treatment by Dijkstra*, which proves that you can implement mutual exclusion purely in software
  - If you try that method, you will discover that it's dreadfully slow in practice

* *Solution of a problem in Concurrent Programming Control*,
E. W. Dijkstra, Communications of the ACM, Vol. 8, No. 9, 1965

Pthreads are quite minimalistic

- They really only have 5 (or 6) things they can do:
  - Start
  - Stop
  - Set and release a memory lock
  - Wait until some other thread wakes them up
  - Wait for every other thread at a barrier (just like MPI can do with processes)
- It's not super productive to write pthreads code explicitly, because it makes you push many keys on the keyboard in order to solve conceptually simple problems
- We cover them anyway, because OpenMP (our next topic) is mostly implemented in terms of pthread operations if you closely inspect how it works