

System Programming: Threads

Raymond Namyst

Dept. of Computer Science

University of Bordeaux, France

<https://gforgeron.gitlab.io/progsys/>

Communication between processes

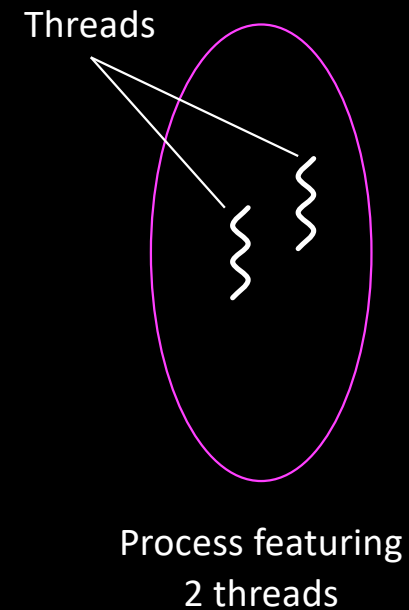
- Processes have private address spaces
 - They don't seem to share any data
 - Actually, they do (mostly in read-only mode, e.g. code)
- Exchanging data between processes is painful... and slow!
 - BTW: Signals are not aimed at communicating rich information
 - Pipes: system calls are slow
- Except with mmap...

Address space and execution flow

- Many applications spawn multiple processes to speed up execution
 - Perform many I/O intensive tasks concurrently
 - Perform tasks in parallel over multicore architectures
- But process creation/destruction is slow
 - Memory allocation + deallocation + initialization
- We only want to start a new activity
 - Sharing data is bonus

Threads

- Threads = Execution flow
- Process = Thread + Address Space
- Several threads can share the same address space



Our first “hello thread” program

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

void *thread_func (void *arg)
{
    printf ("%s from thread!\n", arg);

    return NULL;
}
```

```
int main (int argc, char *argv[])
{
    pthread_t pid;
    pthread_create (&pid, NULL, thread_func, "Hello");

    printf ("Hello from main\n");

    return 0;
}
```

Our first “hello thread” program

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

void *thread_func (void *arg)
{
    printf ("%s from thread!\n", arg);

    return NULL;
}
```

```
int main (int argc, char *argv[])
{
    pthread_t pid;
    pthread_create (&pid, NULL, thread_func, "Hello");

    printf ("Hello from main\n");

    pthread_join (pid, NULL);

    return 0;
}
```

Creating a group of threads

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

int NBTHREADS = 10;

void *thread_func (void *arg)
{
    int me = arg;

    printf ("Hello from thread %d\n", me);

    return NULL;
}
```

```
int main (int argc, char *argv[])
{
    if (argc > 1)
        NBTHREADS = atoi (argv[1]);

    pthread_t pids[NBTHREADS];

    for (int i = 0; i < NBTHREADS; i++)
        pthread_create (&pids[i], NULL, thread_func, i);

    printf ("Hello from main\n");

    for (int i = 0; i < NBTHREADS; i++)
        pthread_join (pids[i], NULL);

    return 0;
}
```

Creating a group of threads

- Useful when decomposing computation is smaller parts
 - Each thread must decide which part it should address
 - Easier if threads are numbered $[0..N-1]$
 - See "spin" kernel, under the EasyPAP environment

Parallelizing computations

- The "spin" kernel involves independent computations on the elements of an array
 - Trivially parallel
- Our first work distribution strategy assigns horizontal stripes of (approximately the same number of) pixels to threads
- TODO: extend spin.c!

Parallelizing computations

- The "*spin*" kernel involves independent computations on the elements of an array
 - Trivially parallel
- Our first work distribution strategy assigns horizontal stripes of (approximately the same number of) pixels to threads

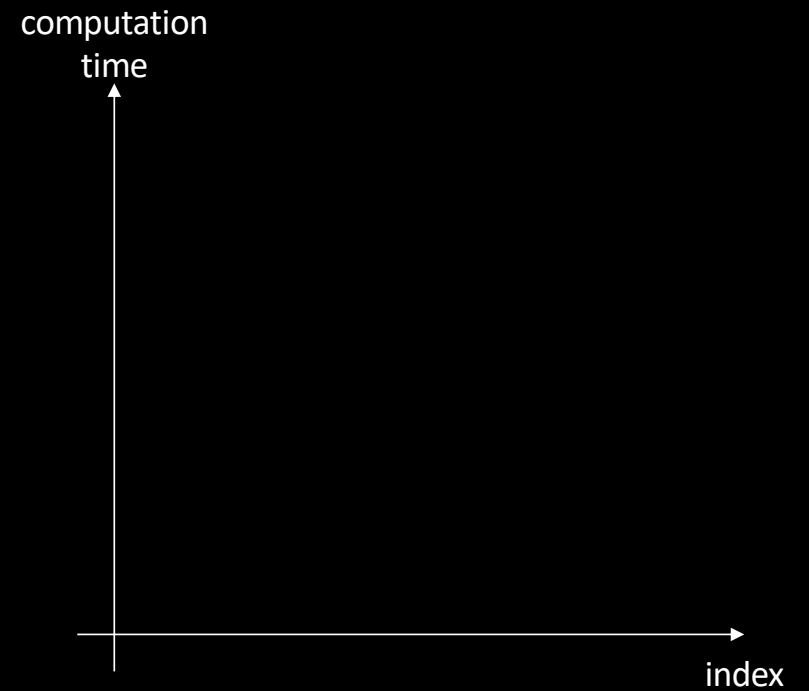
```
void *thread_starter (void *arg)
{
    ...

    for (int i = line; i < line + slice; i++)
        for (int j = 0; j < DIM; j++)
            cur_img (i, j) = compute_color (i, j);

    return NULL;
}
```

Parallelizing computations

- Why did we choose a static *block* distribution?



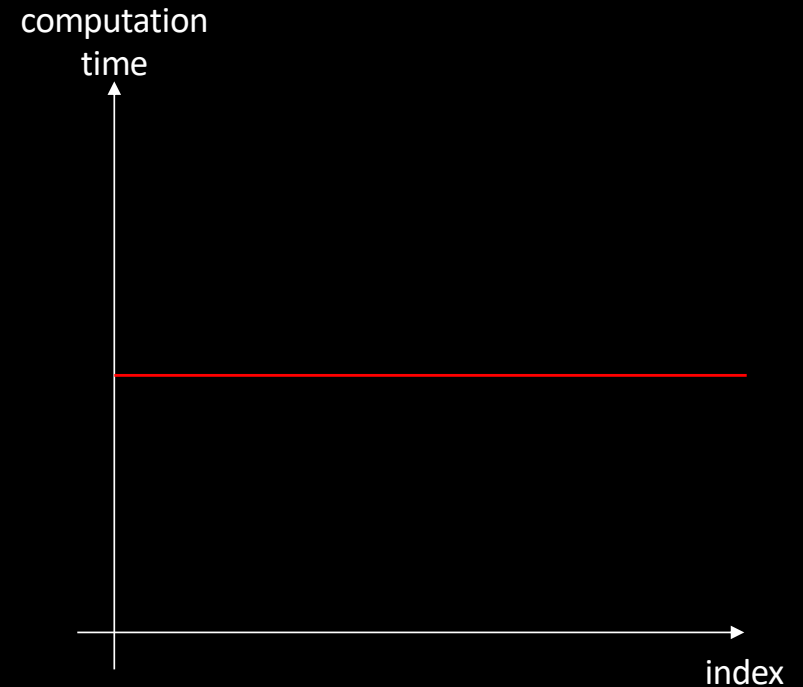
Parallelizing computations

- Why did we choose a static *block* distribution?

- Because we assumed that the computation time of “compute_color” is constant
 - I.e. does not depend on (i, j)

- Let us consider a 1D example

```
float tab [MAX];  
  
for (int i = 0; i < MAX; i++)  
    tab [i] = f (i);
```

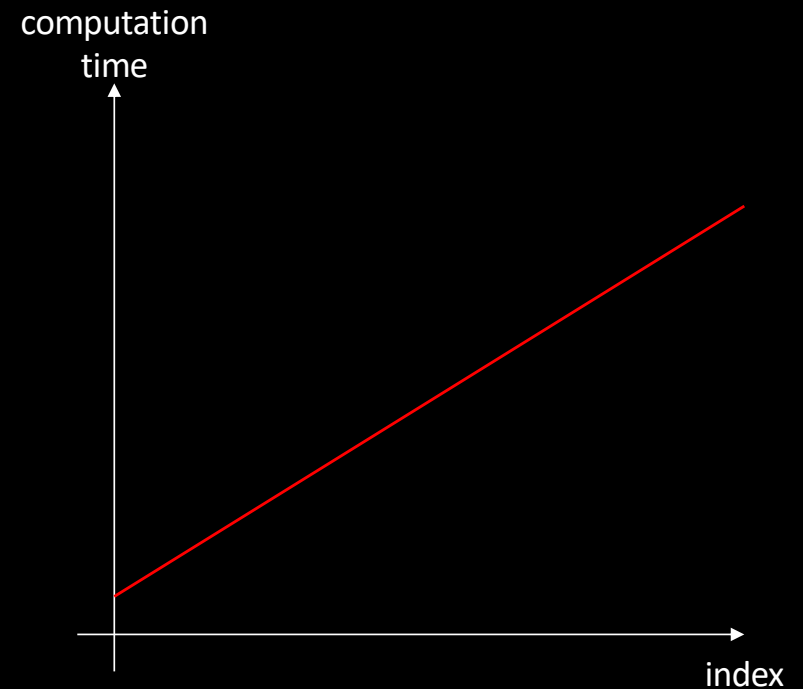


Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];  
  
for (int i = 0; i < MAX; i++)  
    tab [i] = f (i);
```

- What if the computation time is linearly increasing?

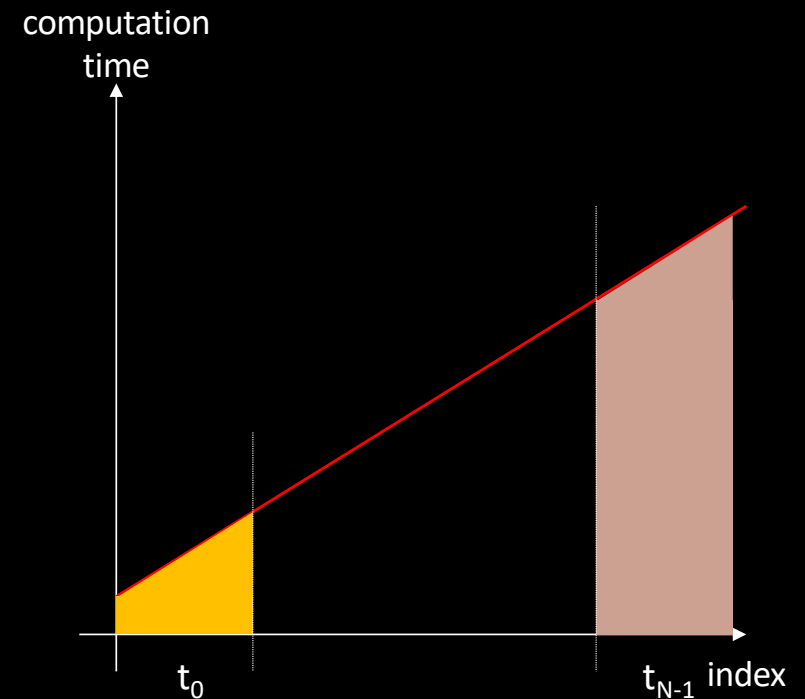


Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];  
  
for (int i = 0; i < MAX; i++)  
    tab [i] = f (i);
```

- What if the computation time is linearly increasing?
 - Our block distribution is no longer relevant

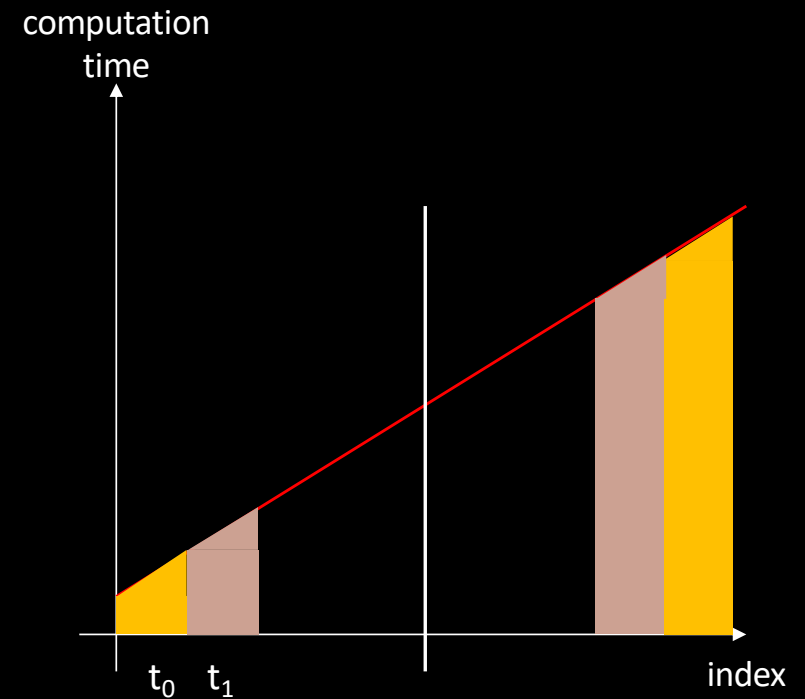


Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];  
  
for (int i = 0; i < MAX; i++)  
    tab [i] = f (i);
```

- What if the computation time is linearly increasing?
 - Our block distribution is no longer relevant
 - Well, using a mirror block distribution assigning two blocks per thread would work...

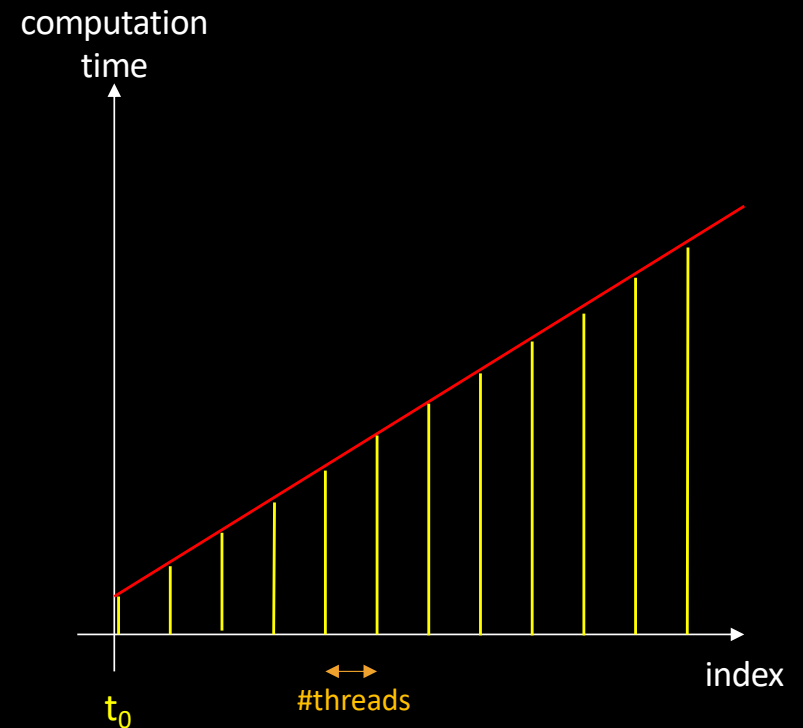


Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];  
  
for (int i = 0; i < MAX; i++)  
    tab [i] = f (i);
```

- What if the computation time is linearly increasing?
 - A cyclic distribution of indexes would be a good option

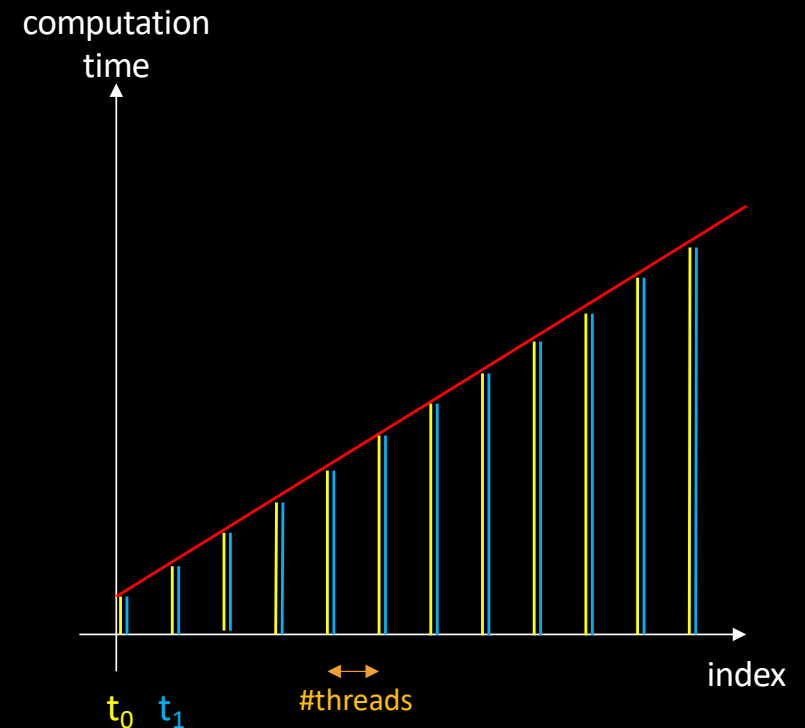


Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];  
  
for (int i = 0; i < MAX; i++)  
    tab [i] = f (i);
```

- What if the computation time is linearly increasing?
 - A cyclic distribution of indexes would be a good option

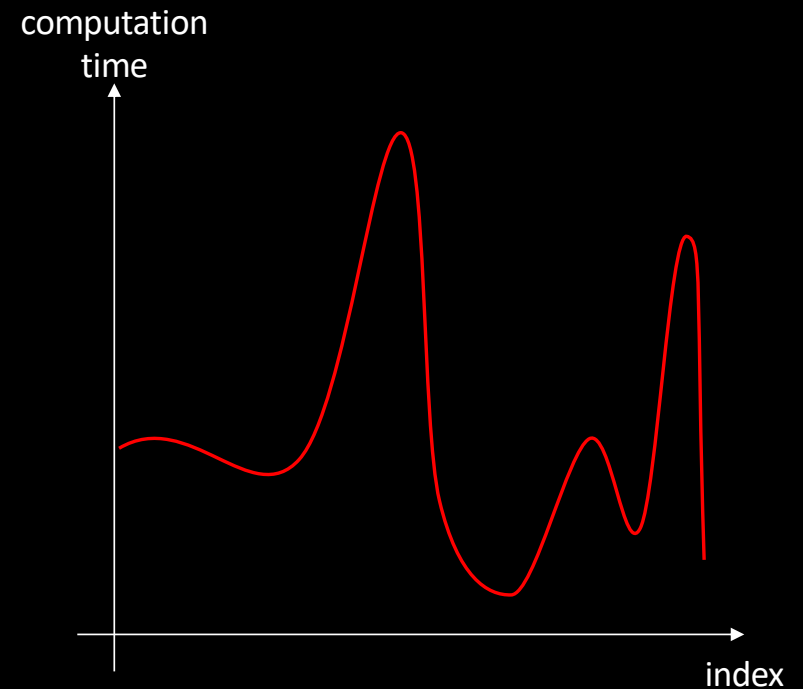


Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];  
  
for (int i = 0; i < MAX; i++)  
    tab [i] = f (i);
```

- What if the computation time is unpredictable?
 - Even the cyclic strategy may fail



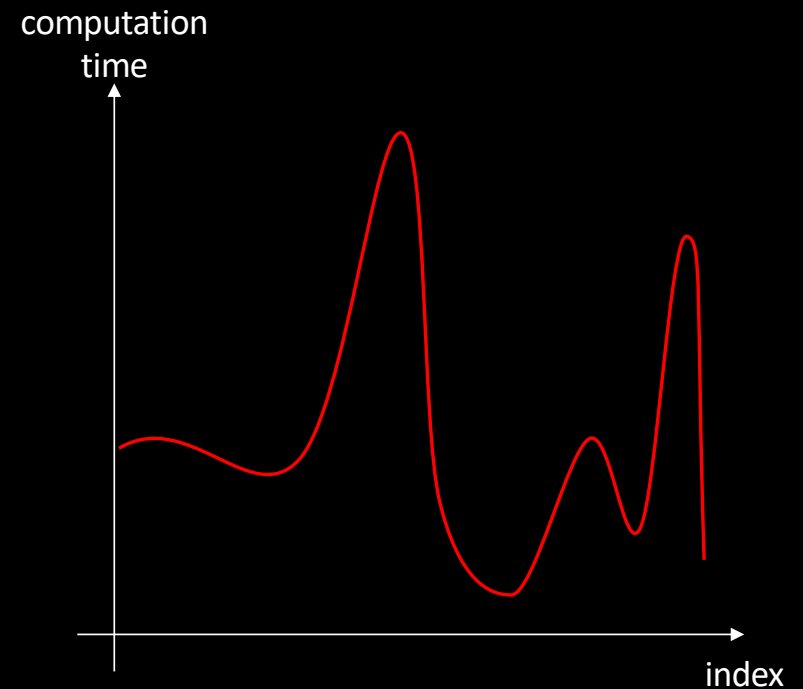
Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];  
  
for (int i = 0; i < MAX; i++)  
    tab [i] = f (i);
```

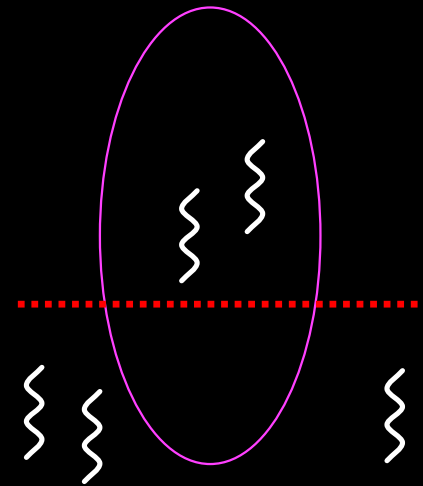
- What if the computation time is unpredictable?

- Dynamic strategy
 - Distribute indexes in a greedy manner
 - To be explored later on

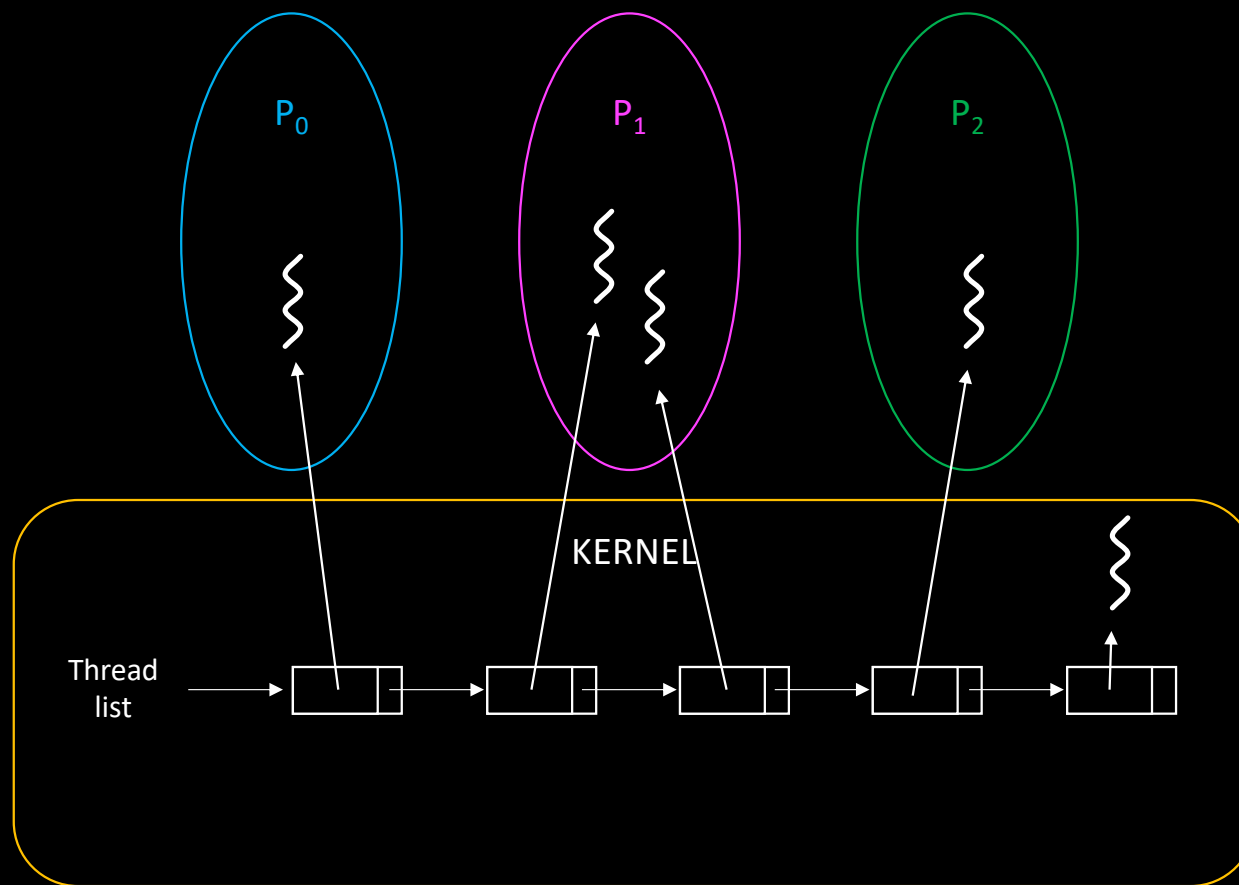


Processes and Threads

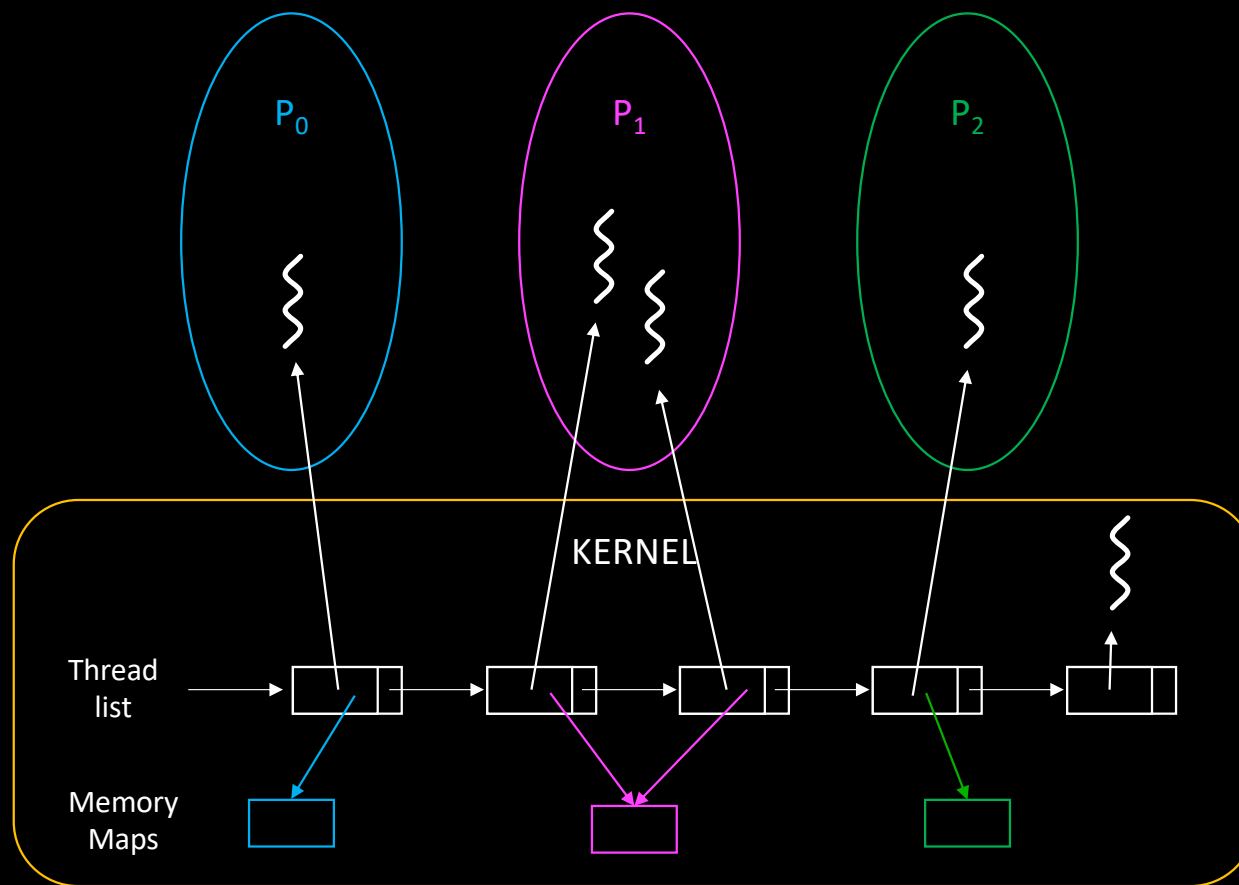
- Some (daemons) threads only run inside the kernel
- Modern kernels only run threads



Processes and Threads: the Big Picture



Processes and Threads: the Big Picture

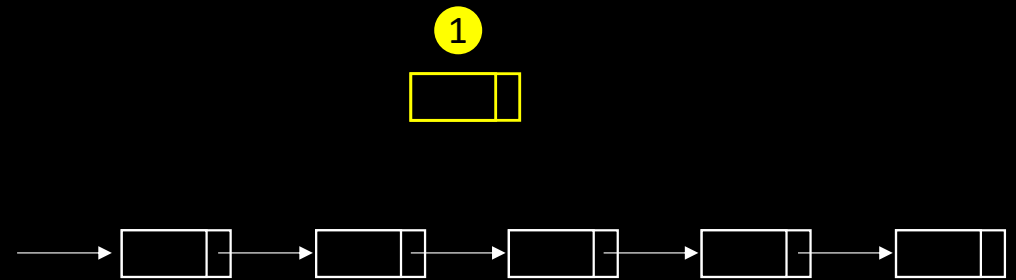


Race conditions

- **Threads can access the same data simultaneously**
 - May lead to undefined behavior, data corruption, ...
 - Think about
 - Linked lists, graphs, hash tables
 - Structures where several fields must be updated consistently
 - Or just integers...
- **When executing kernel code, processes share data as well**
 - So the kernel must enforce synchronization

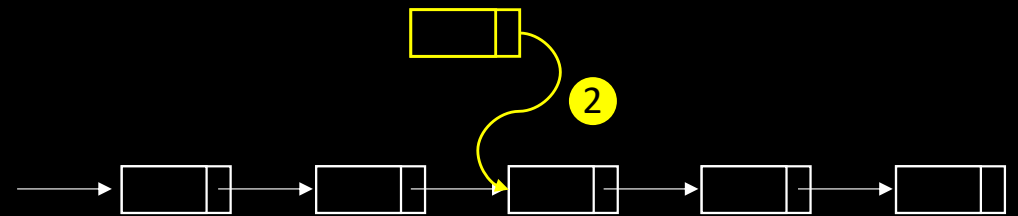
Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous



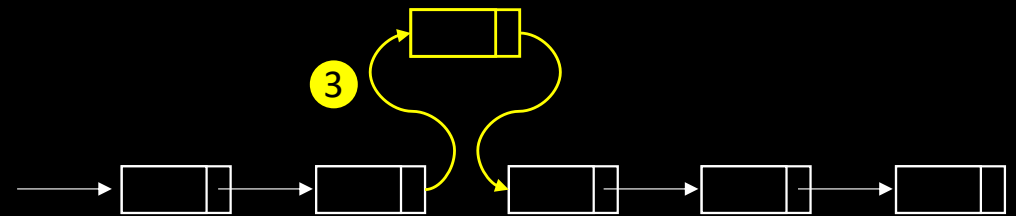
Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous



Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous



Race conditions

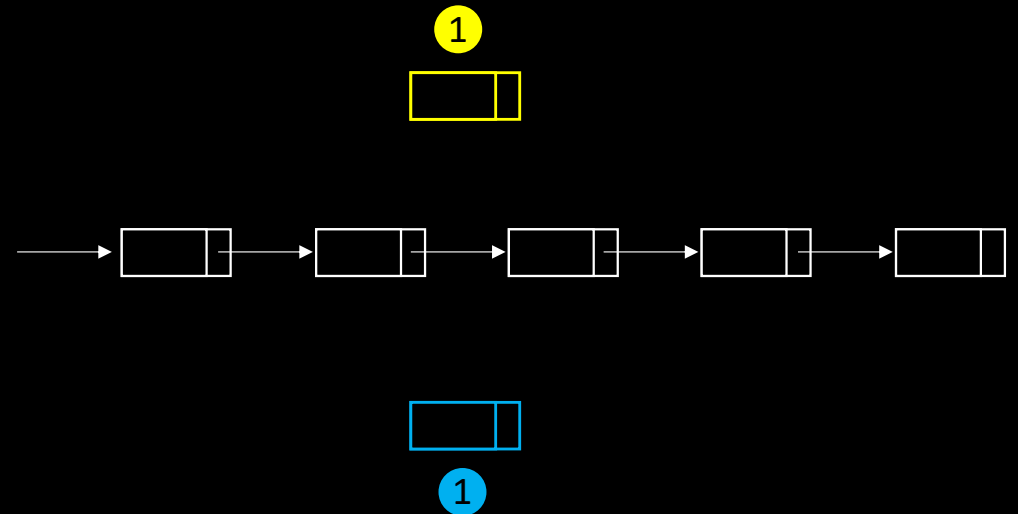
- Example with linked lists

- Insertion of a new element

- 3 steps

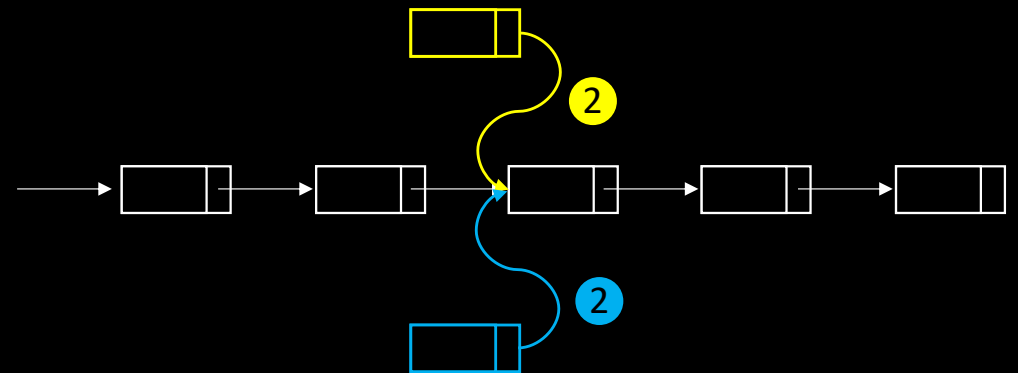
1. Allocate
2. Set next
3. Modify previous

- What if two threads perform an insert simultaneously, at the same position?



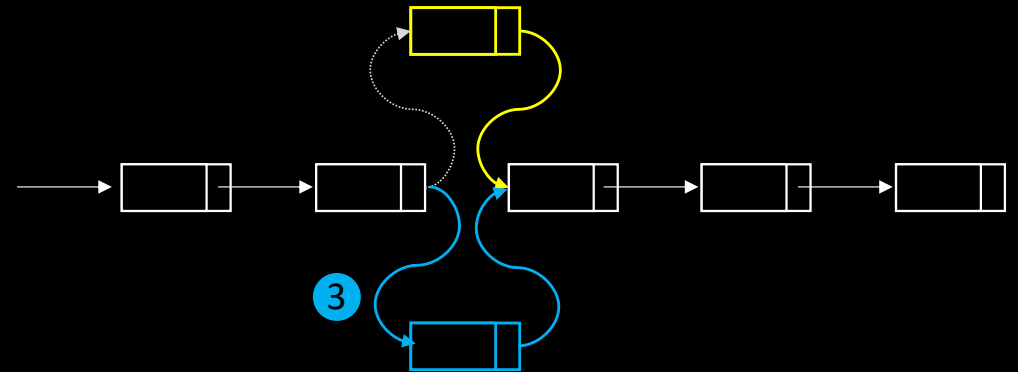
Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous
 - What if two threads perform an insert simultaneously, at the same position?



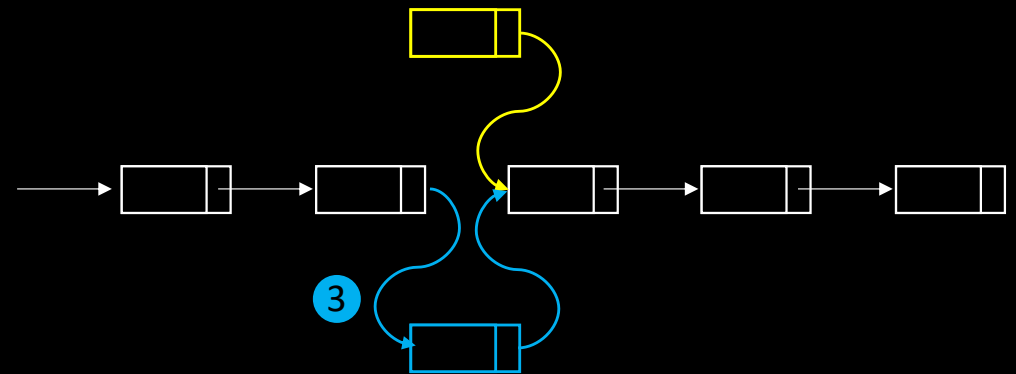
Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous
 - What if two threads perform an insert simultaneously, at the same position?



Race conditions

- Example with linked lists
 - Insertion of a new element
 - 3 steps
 1. Allocate
 2. Set next
 3. Modify previous
 - What if two threads perform an insert simultaneously, at the same position?
 - We may end up with a corrupted list



Race conditions

```
volatile int n = 0;
```

```
for (int i = 0; i < 100; i++)  
    n++;
```

```
for (int i = 0; i < 100; i++)  
    n++;
```

pthread_join

```
printf ("n = %d\n", n);
```

$n = 200$?

Race conditions

```
volatile int n = 0;
```

```
for (int i = 0; i < 100; i++)  
    n++;
```

```
for (int i = 0; i < 100; i++)  
    n++;
```

`pthread_join`

```
printf ("n = %d\n", n);
```

$n \in [100, 200]$?

Possible scenario

$n++ \Leftrightarrow$ `load @n, r1` ; load from memory
`inc r1` ; increment register
`store r1, @n` ; store in memory



$n : 0$

Possible scenario

$n++ \Leftrightarrow$ `load @n, r1` ; load from memory
`inc r1` ; increment register
`store r1, @n` ; store in memory



`load @n, r1`
`inc r1`




← context switch


$n : 0$

Possible scenario

$n++ \Leftrightarrow$ `load @n, r1` ; load from memory
`inc r1` ; increment register
`store r1, @n` ; store in memory

$n : 0 \dots 99$


`load @n, r1`
`inc r1`


`load @n, r1`
`inc r1`
`store r1, @n`
...

} 99x

Possible scenario

$n++ \Leftrightarrow$ `load @n, r1` ; load from memory
`inc r1` ; increment register
`store r1, @n` ; store in memory

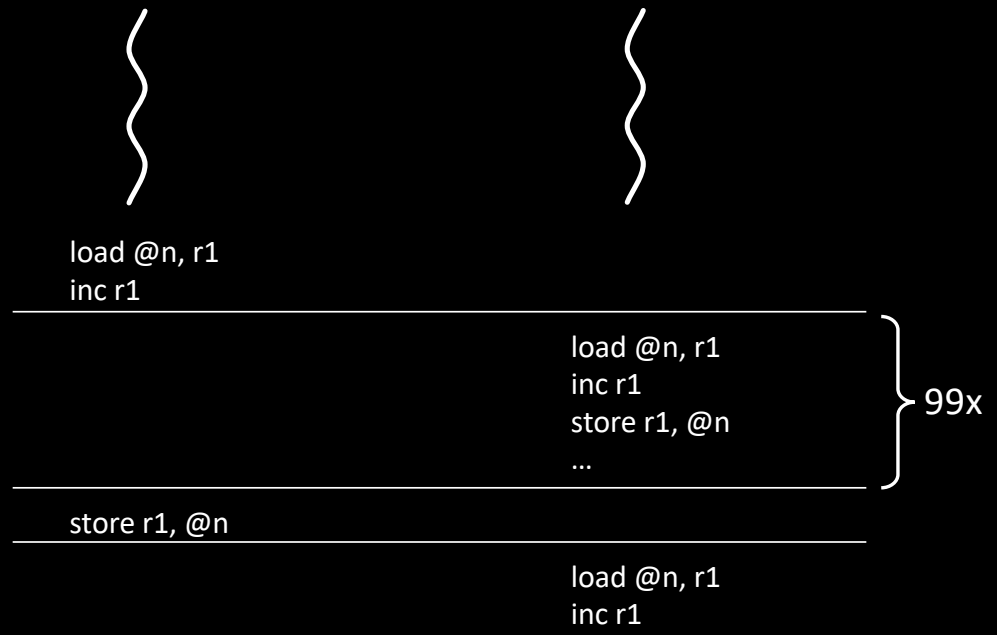
$n : 0 \dots 99$



Possible scenario

$n++ \Leftrightarrow$ `load @n, r1` ; load from memory
`inc r1` ; increment register
`store r1, @n` ; store in memory

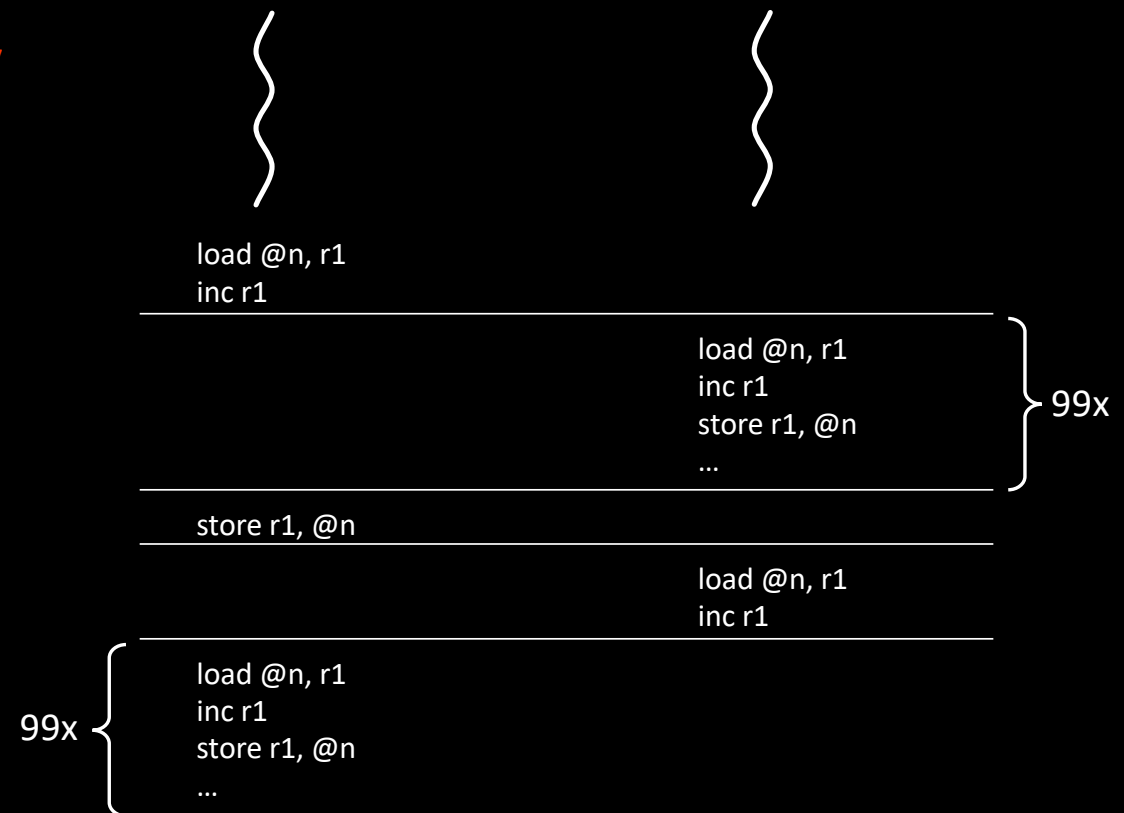
$n : 0 \dots 99$



Possible scenario

$n++ \Leftrightarrow$ `load @n, r1` ; load from memory
`inc r1` ; increment register
`store r1, @n` ; store in memory

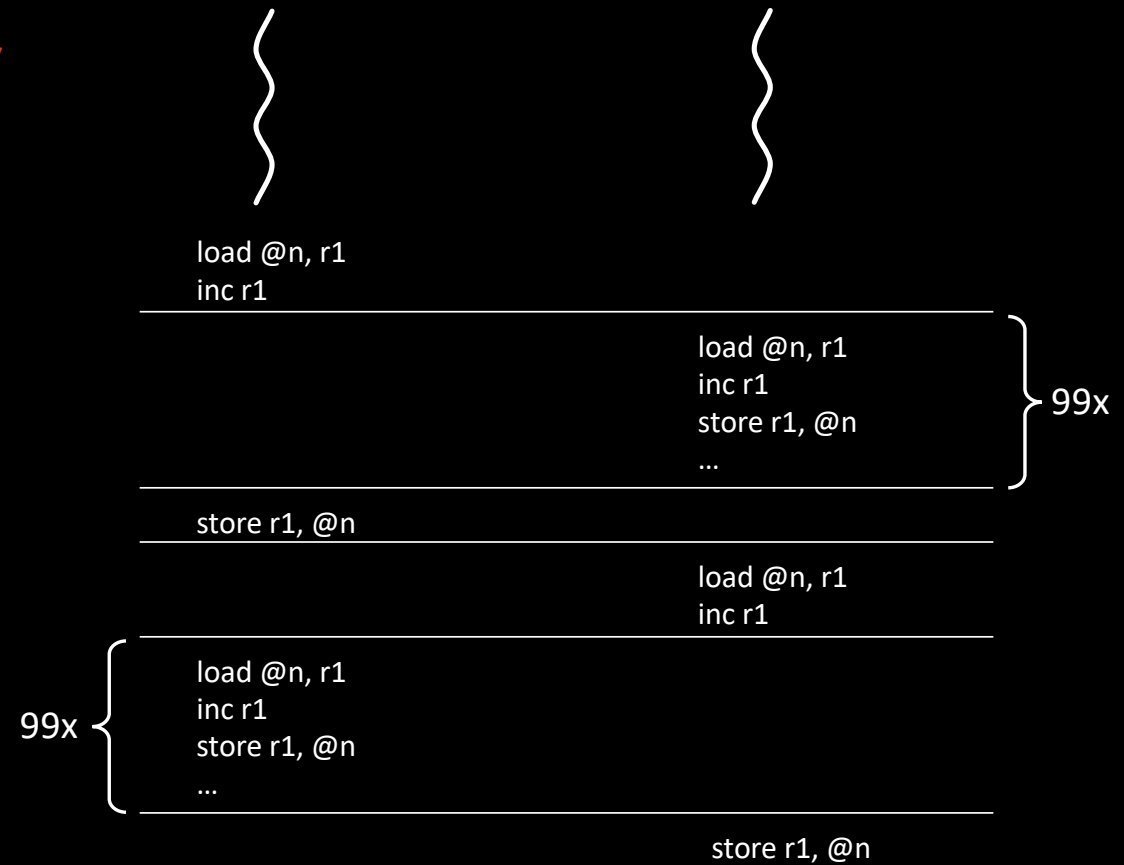
$n : 0 \text{--} 99 \text{--} 1 \text{--} 100$



Possible scenario

$n++ \Leftrightarrow$ `load @n, r1` ; load from memory
`inc r1` ; increment register
`store r1, @n` ; store in memory

$n : 0 \text{ } \cancel{99} \text{ } 1 \text{ } \cancel{100} \text{ } 2$



Possible scenario

$n++ \Leftrightarrow$ `load @n, r1` ; load from memory
`inc r1` ; increment register
`store r1, @n` ; store in memory

$n : 0 \text{ } 99 \text{ } 1 \text{ } 100 \text{ } 2$

$n \in [2, 200] !$

99x {

`load @n, r1`
`inc r1`
`store r1, @n`
...

`load @n, r1`
`inc r1`

`store r1, @n`

`load @n, r1`
`inc r1`
`store r1, @n`
...

} 99x

Race conditions

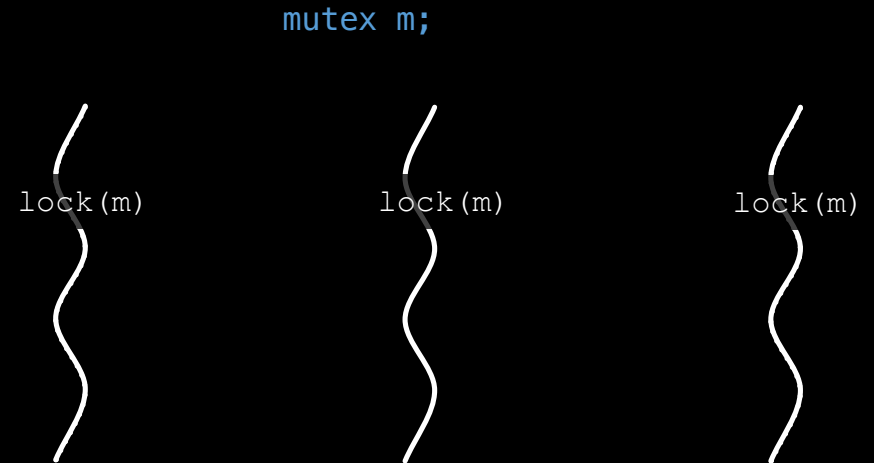
- Even the simple ++ operator is not an *atomic* operation
 - So we must prevent multiple threads to execute this operation concurrently!
- To do so, we need synchronization tools

Mutexes

- A *mutex* is an object intended to ensure **MUTual EXclusion** between threads
 - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Two operations can be performed:
 - `pthread_mutex_lock (pthread_mutex_t *m);`
 - Blocks the caller while lock is not available
 - `pthread_mutex_unlock (pthread_mutex_t *m);`
 - Releases the lock (never blocking)

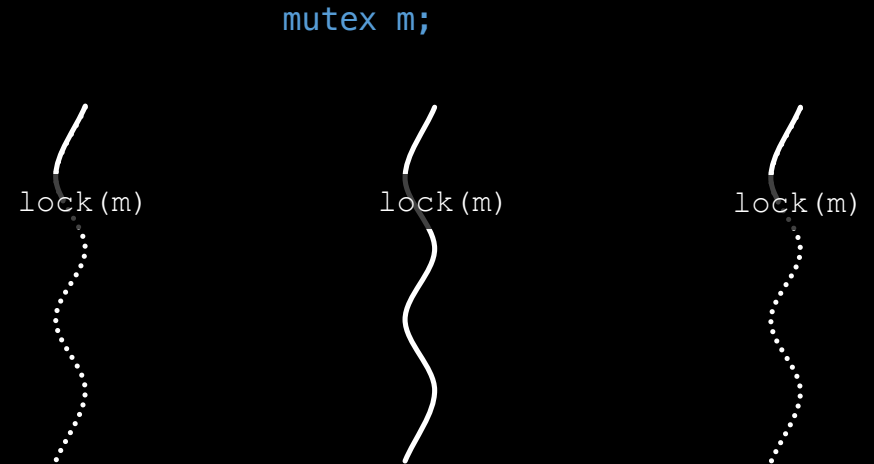
Mutexes

- Let's see how it works on a simple example
 - Three threads
 - Each one calls lock(), then unlock()



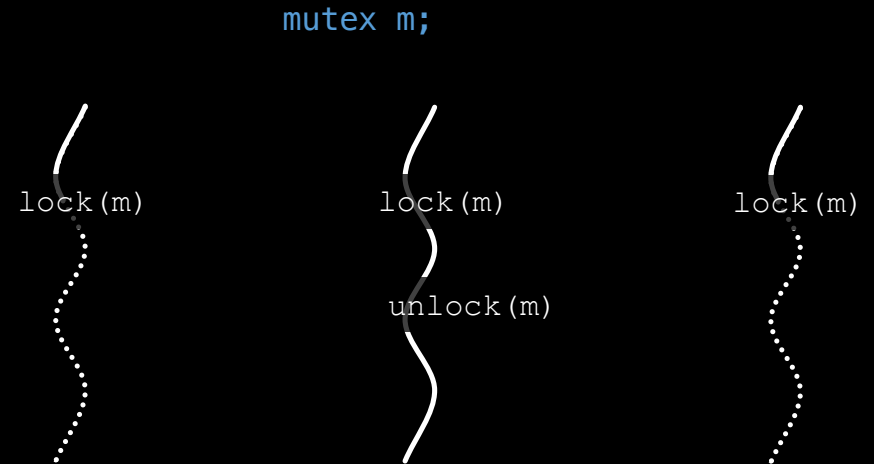
Mutexes

- Let's see how it works on a simple example
 - Three threads
 - Each one calls lock(), then unlock()



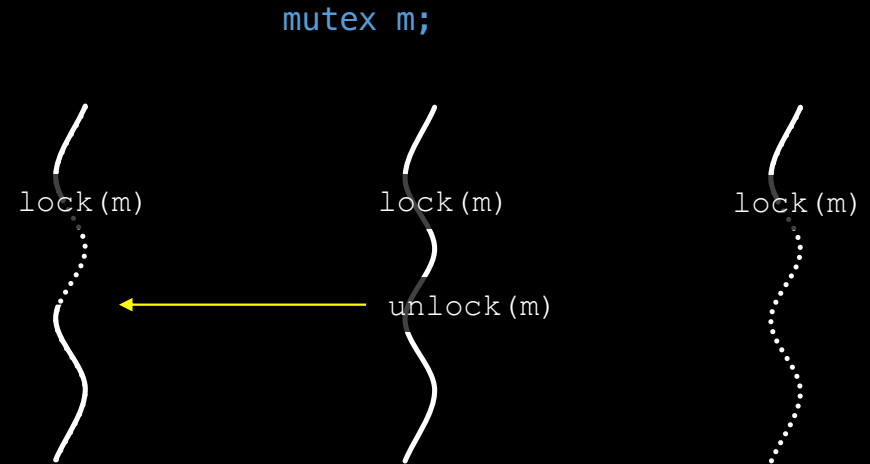
Mutexes

- Let's see how it works on a simple example
 - Three threads
 - Each one calls lock(), then unlock()



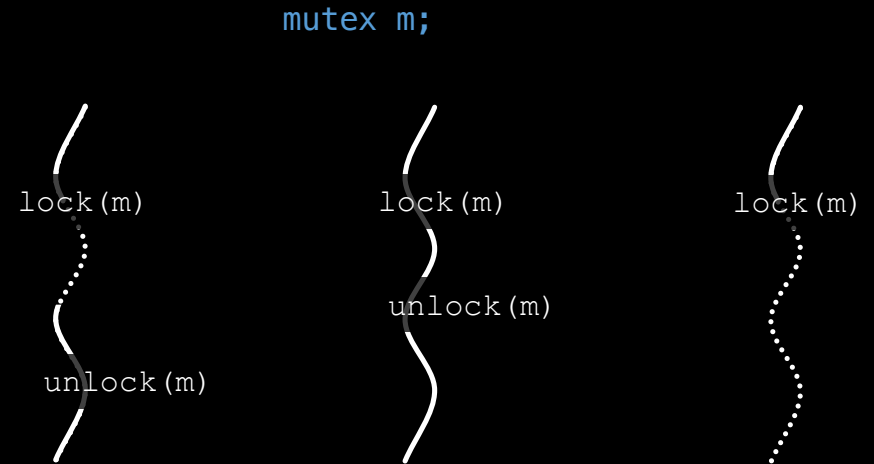
Mutexes

- Let's see how it works on a simple example
 - Three threads
 - Each one calls lock(), then unlock()



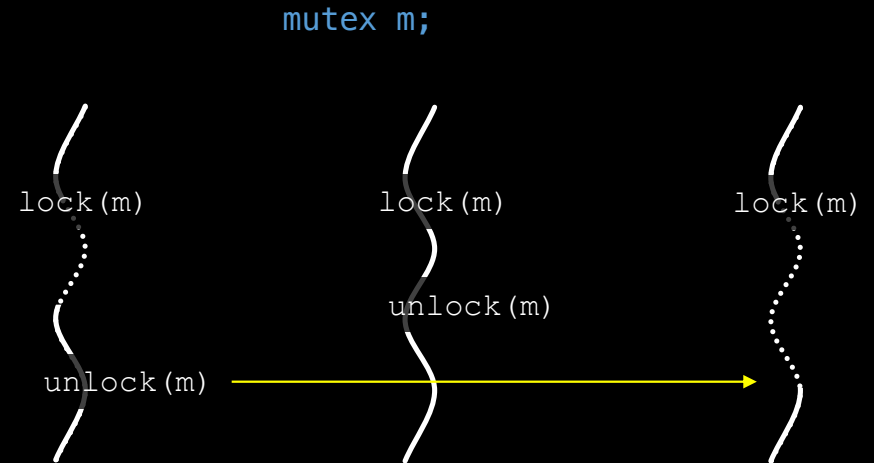
Mutexes

- Let's see how it works on a simple example
 - Three threads
 - Each one calls lock(), then unlock()



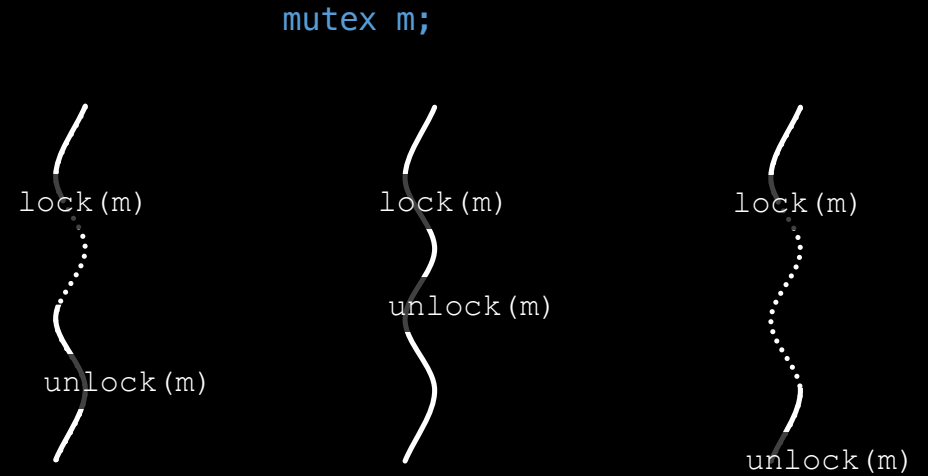
Mutexes

- Let's see how it works on a simple example
 - Three threads
 - Each one calls lock(), then unlock()



Mutexes

- Let's see how it works on a simple example
 - Three threads
 - Each one calls lock(), then unlock()



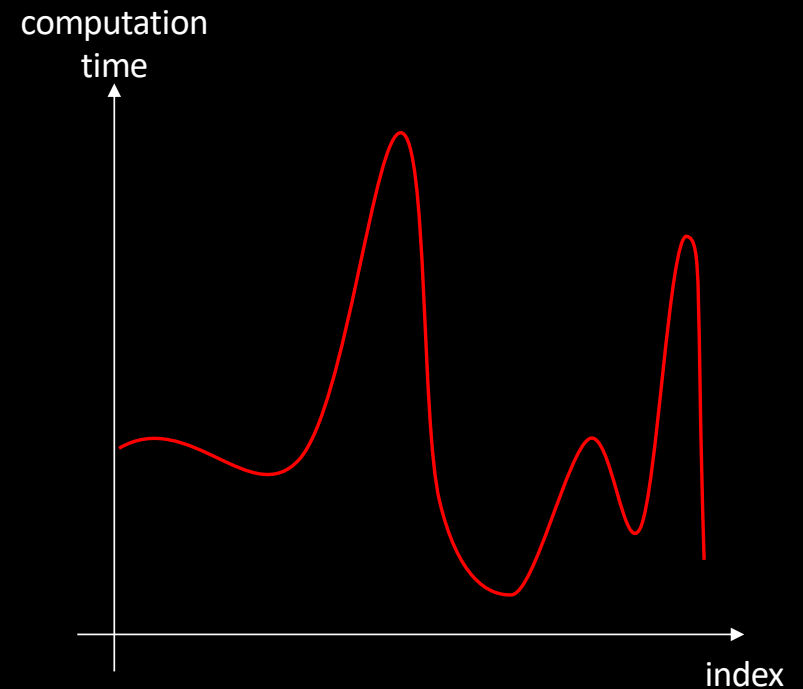
Parallelizing computations

- Back to our 1D example

```
for (int i = 0; i < MAX; i++)  
    tab [i] = f (i);
```

- Dynamic strategy

```
float tab [MAX];  
  
for (;;) {  
    int i = get_index();  
    if (i > MAX)  
        break;  
    tab [i] = f (i);  
}
```



Additional resources
available on

<http://gforgeron.gitlab.io/progsys/>