# Embedded Software

Parallel programs, processes and threads



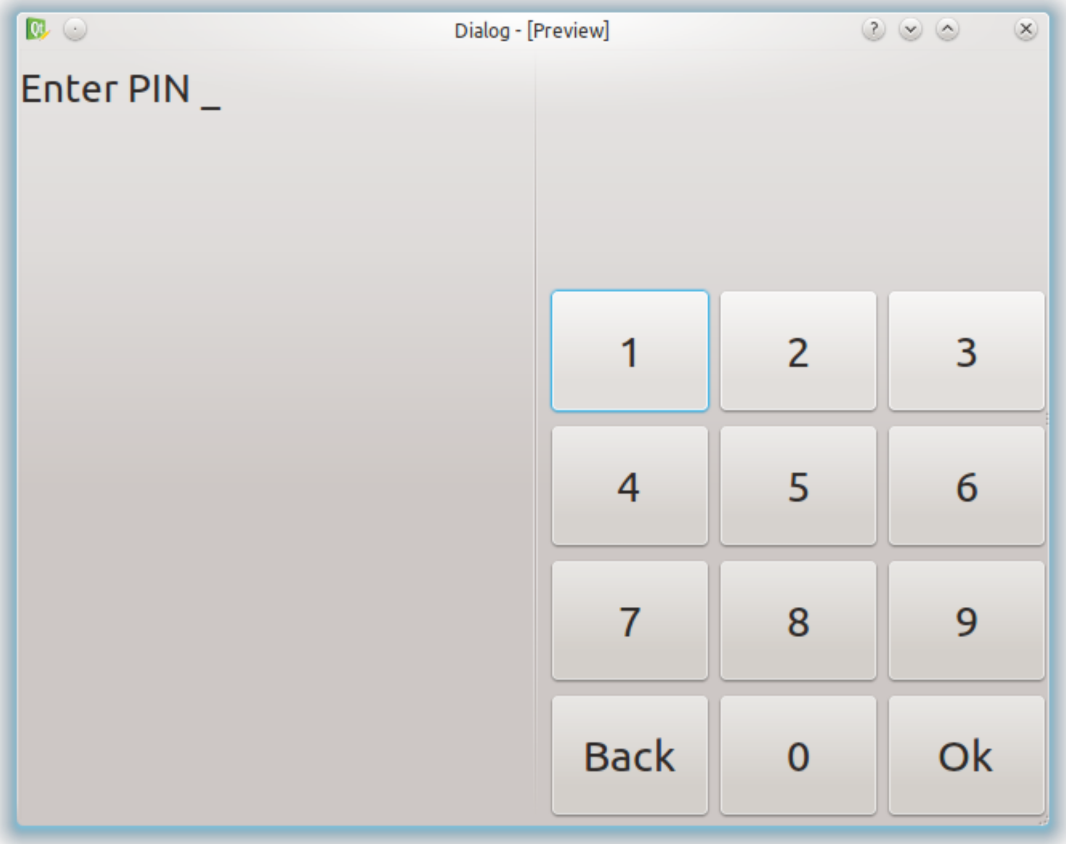
AARHUS UNIVERSITY

# Agenda

- The problem - A Case
- A Solution - Parallelism
- Processes and Threads in Linux
- Advantages & Disadvantages with multitasking

# Case

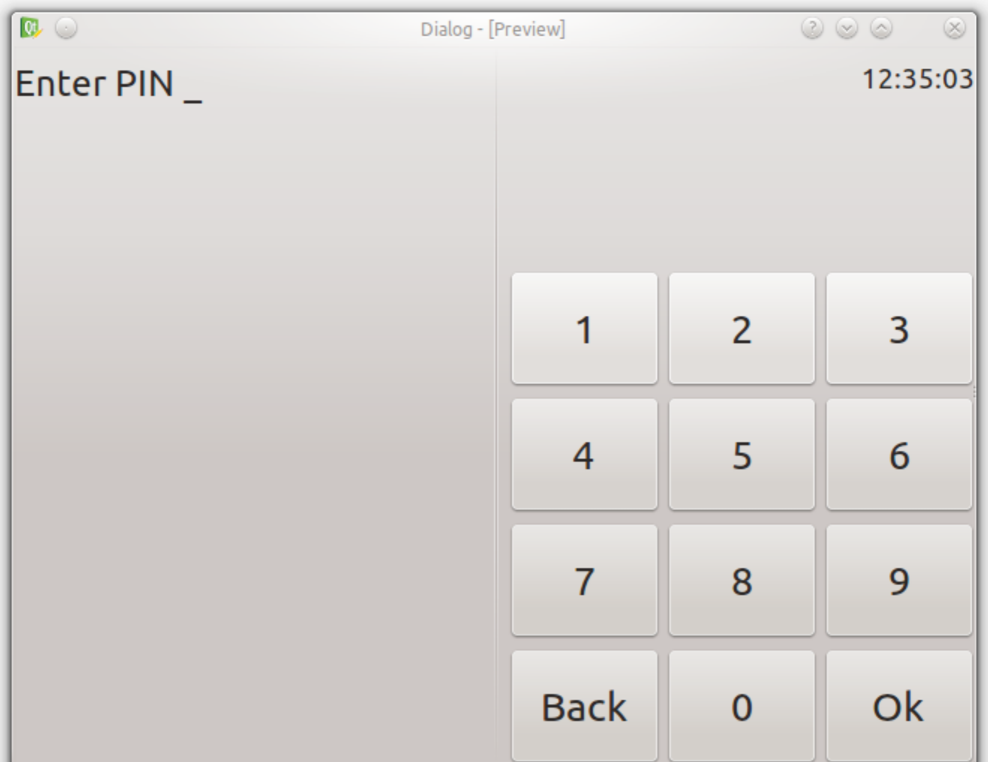
- Consider a system that allows a user to enter a PIN.
  - ***How would you implement this?***




```
void main()
{
   print ”ENTER PIN:”
   get key -> ch
   while(ch != ”OK”)
    input += ch
    compare(input, pin)
    ...
}
```

# Case

- Now consider the same system, but now with a clock.
  - ***How would you implement this?***

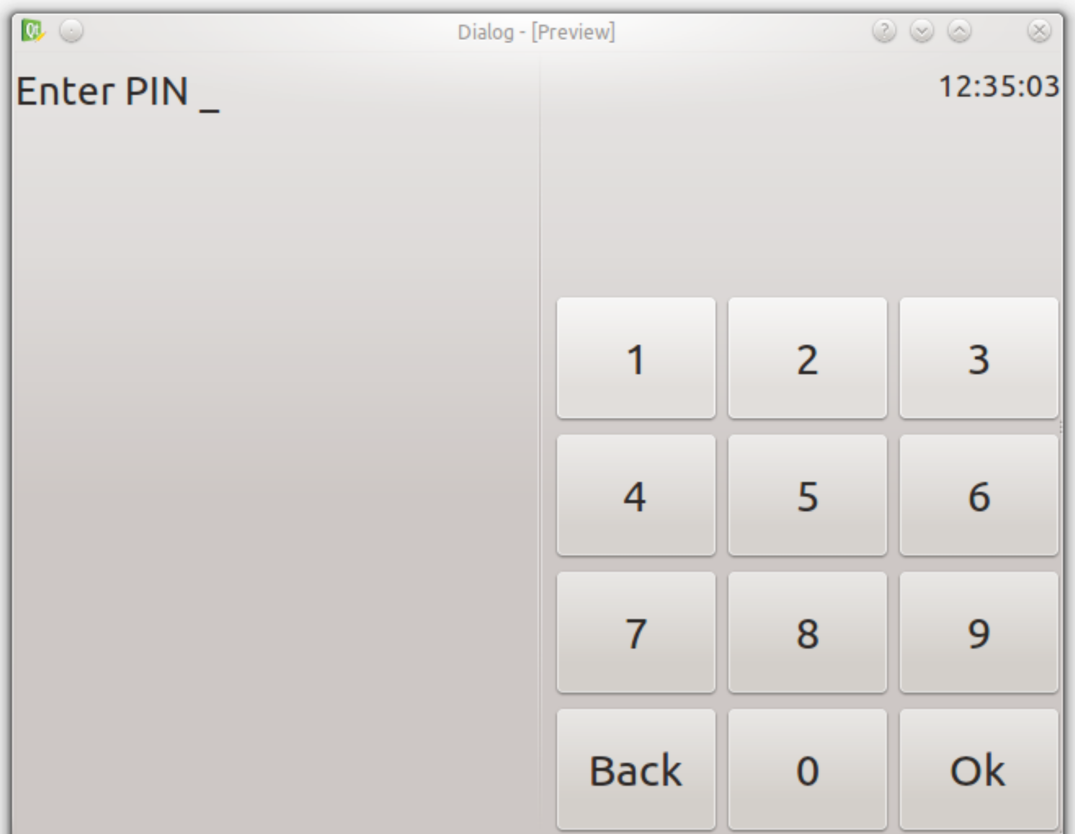



```
void main()
{
   ???
}
```

- How would you update the clock while waiting for input?
- How would you capture key presses while updating the clock?

# Case - Solution

- With multiple threads, this is trivial:




Handles user input as before

```
void userInput()
{
   print ”ENTER PIN:”
   get key -> ch
   while(ch != ”OK”)
    input += ch
    compare(input, pin)
    ...
}
```

Handles clock updates

```
void updClock()
{
   while(true)
   {
    display current time
    wait 1s
   }
}
```

Starts threads that run user input and clock update routines

```
int main()
{
   startThread(userInput)
   startThread(updClock)
}
```

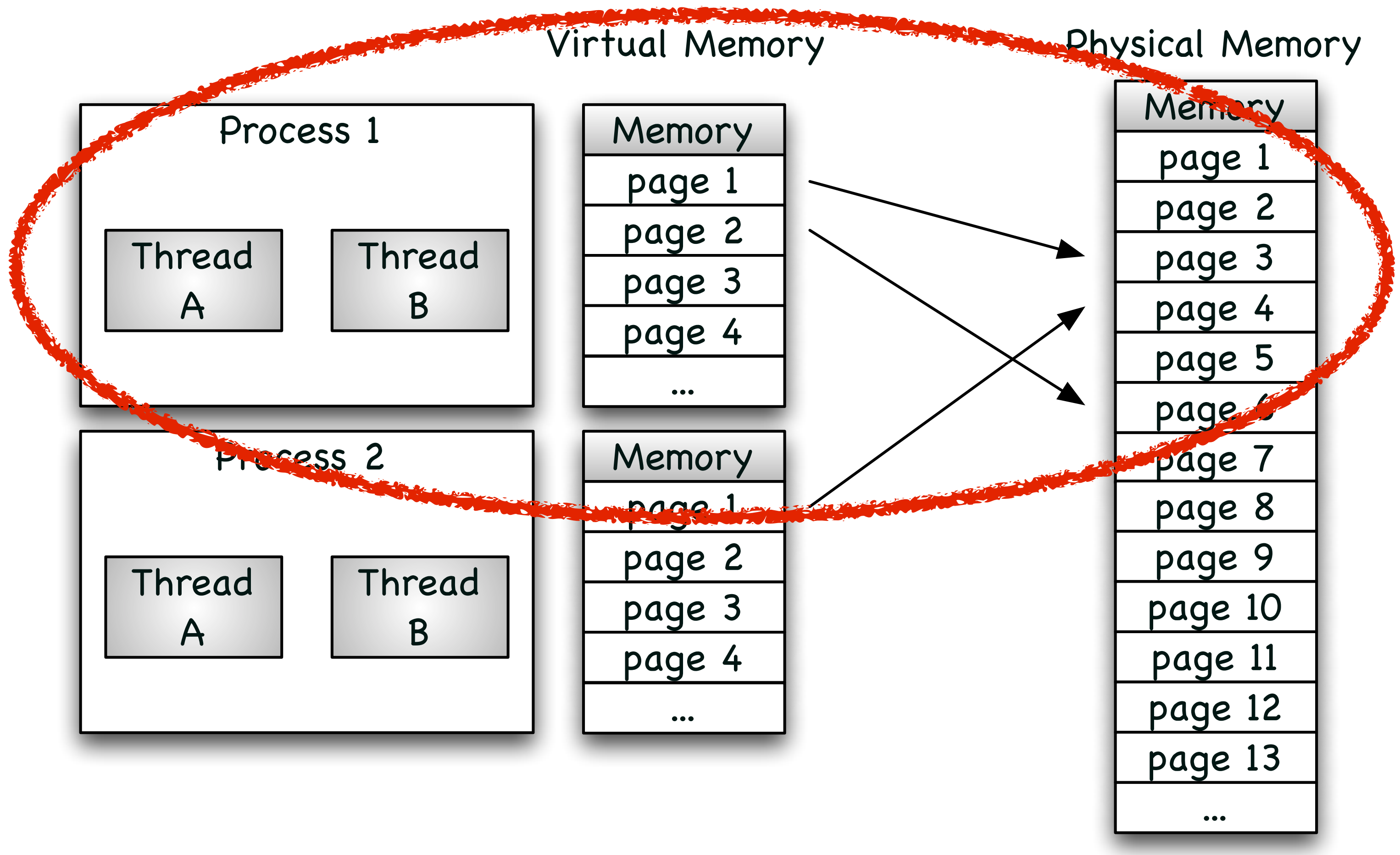
# Parallel programs - A Solution

- *Parallel programs* are programs that are divided into execution units that can execute concurrently
  - Processes
  - Threads, jobs or tasks

# Process & Threads in Linux

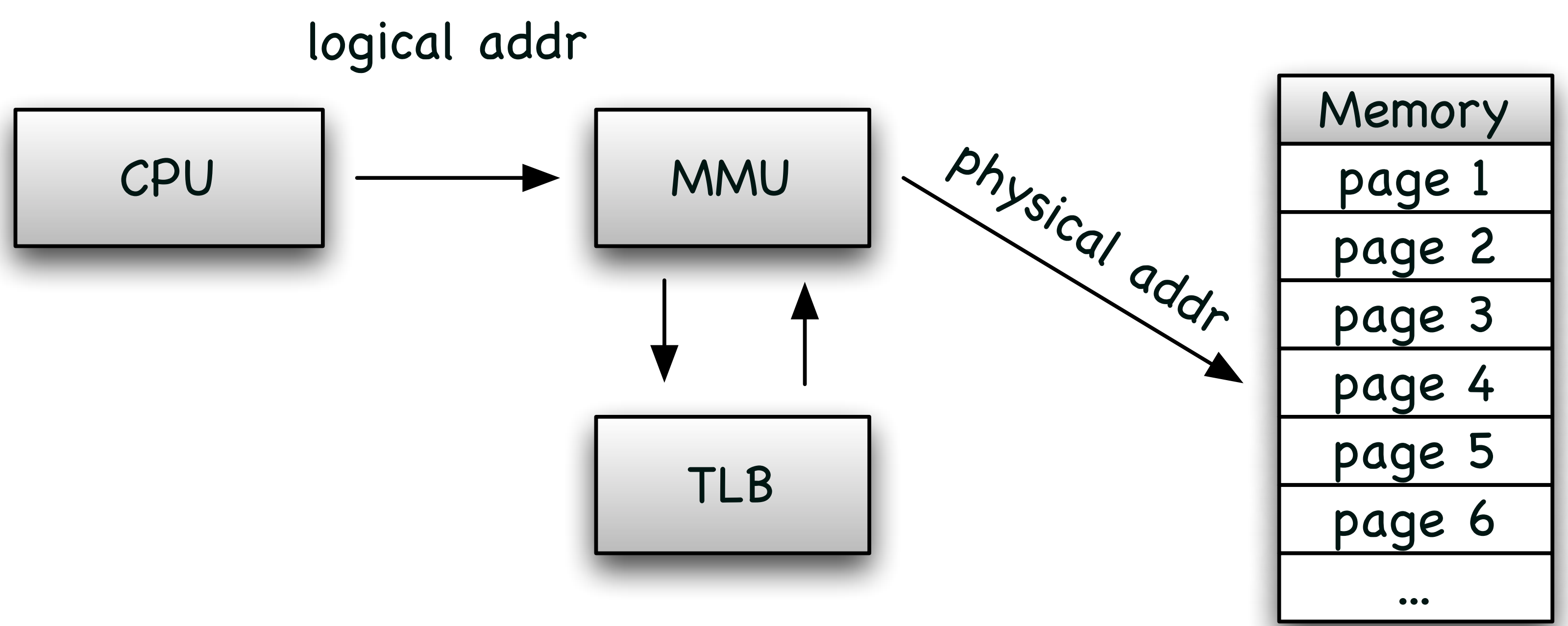
# Processes and the OS (Kernel)

- Threads, Processes and Memory mapping
  - Each process has its own memory space
    - A mapping exists between virtual and physical memory
    - Not possible for one process to write in another address space
  - Threads share data space
    - Care must be taken *NOT* to destroy other threads data



# Processes and the OS (Kernel)

- MMU - Memory Management Unit

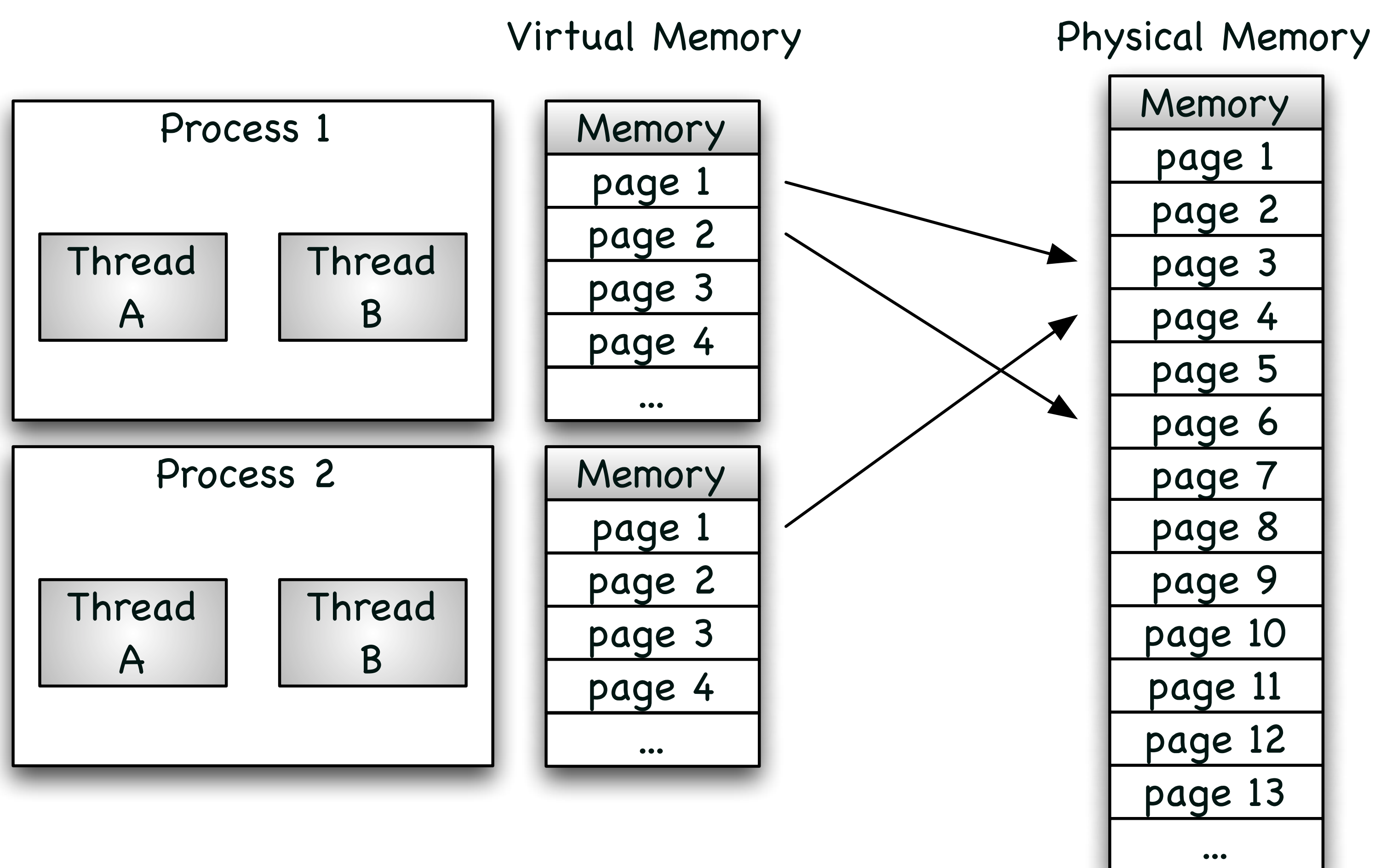



MMU: Memory Management Unit
TLB: Translation Lookaside Buffer

*More to follow in MPS later*

# Processes and the OS (Kernel)

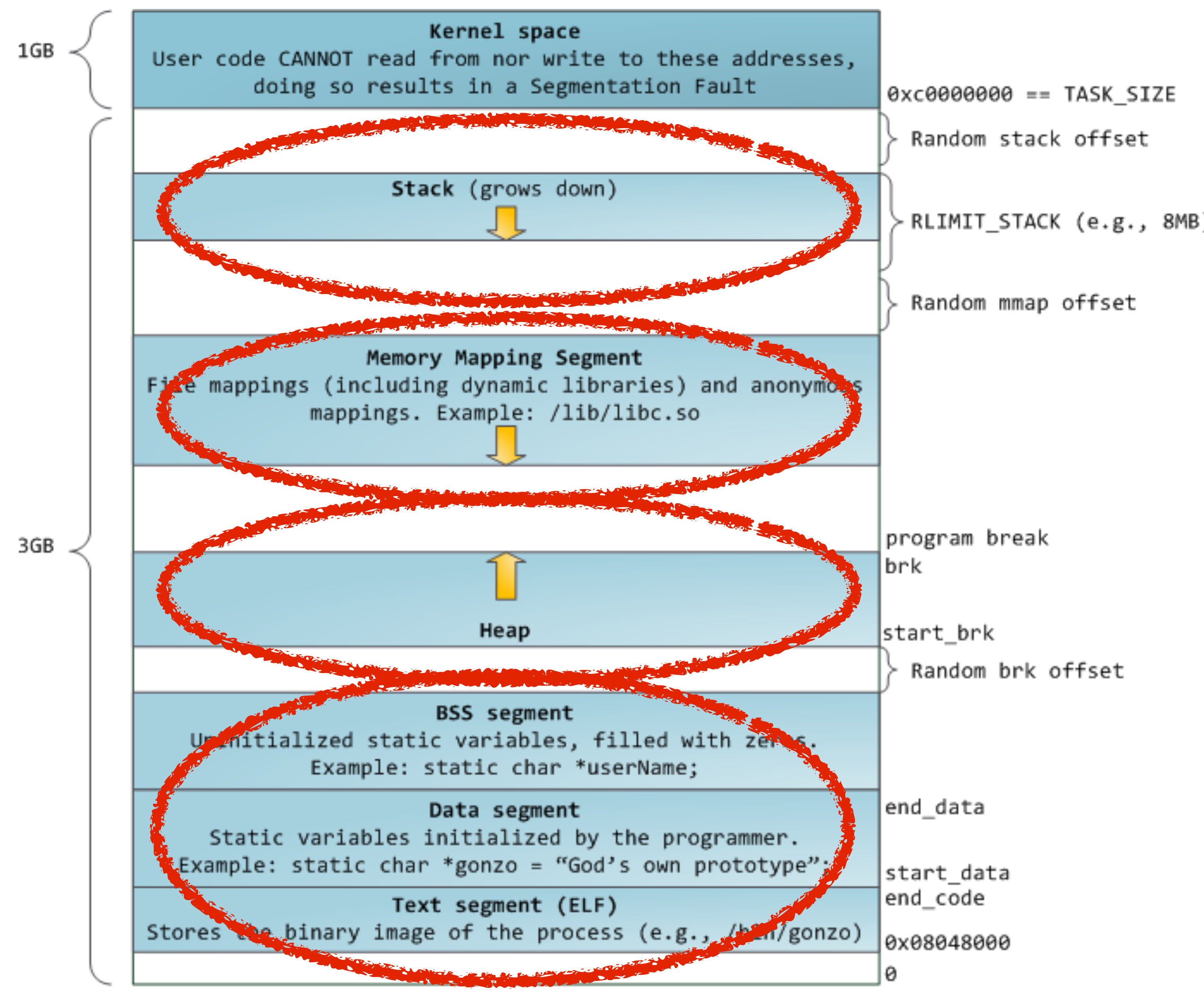
- MMU - Memory Management Unit



# Process - What is it?

- Process - A program being executed in Linux
  - Stack
    - Local variables
    - Function return values
    - LIFO
  - Heap
    - “Free-store”
    - Dynamically allocated memory
  - Memory Mapping
    - File mapped in memory
    - Includes dyn libs
    - Sharing memory between processe
  - Variables and ELF

Process Memory Layout

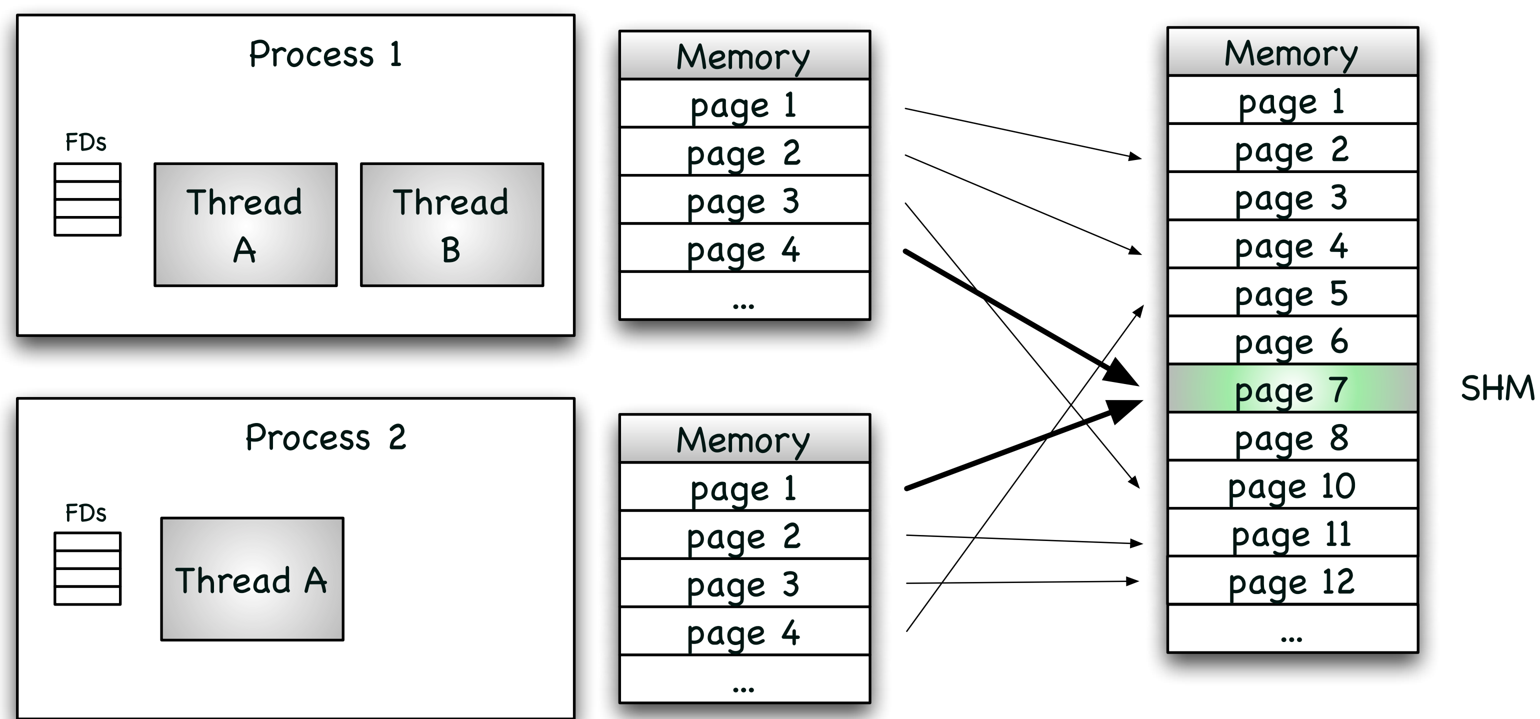



http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory


AARHUS UNIVERSITY

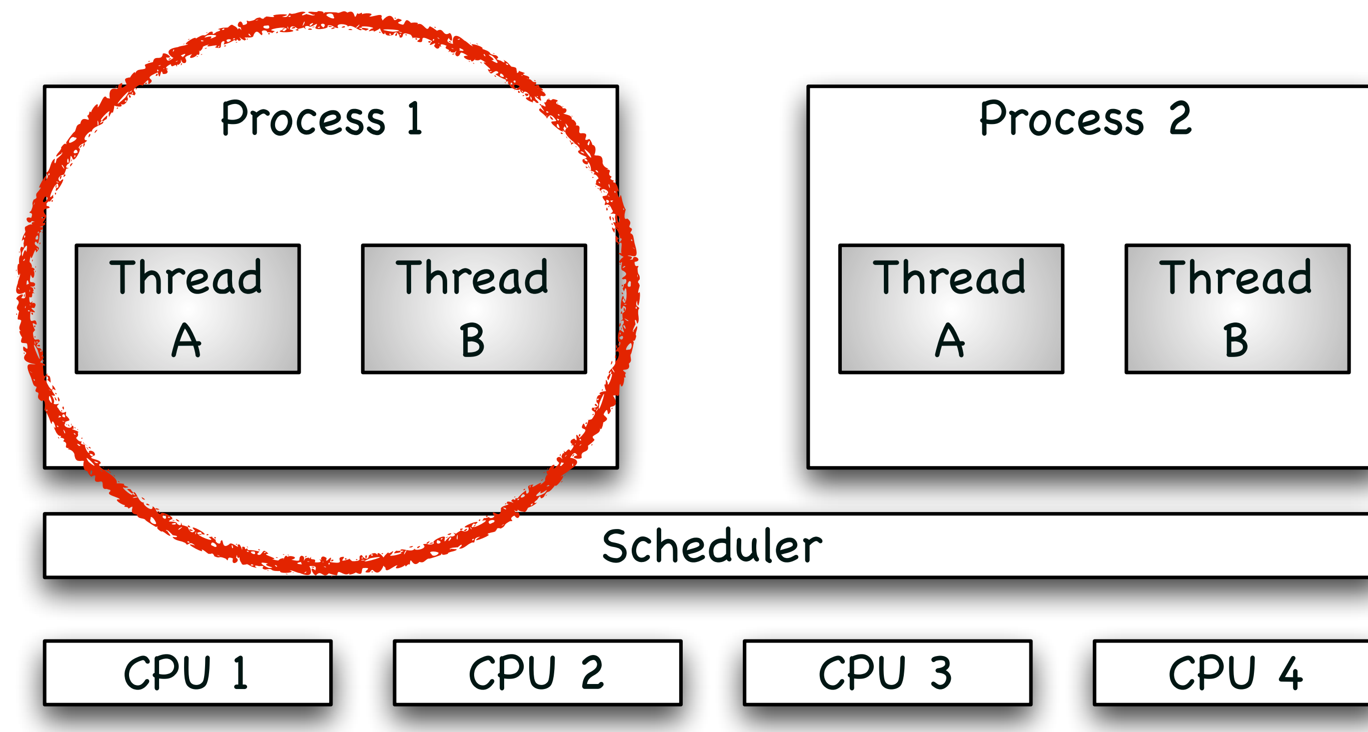
# Processes and the OS - Sharing memory - IPC

- POSIX Shared Memory
  - Accessing memory affects other process
  - Pro
    - Speed/Performance
  - Cons
    - Fragile
      - Death of process
      - Data must abide certain principles
    - Challenge ensuring synchronicity



# Threads of execution and the OS (Kernel)

- How are threads mapped for execution?
  - Three different models
    - User level threading
    - Kernel level threading
    - Hybrid level threading

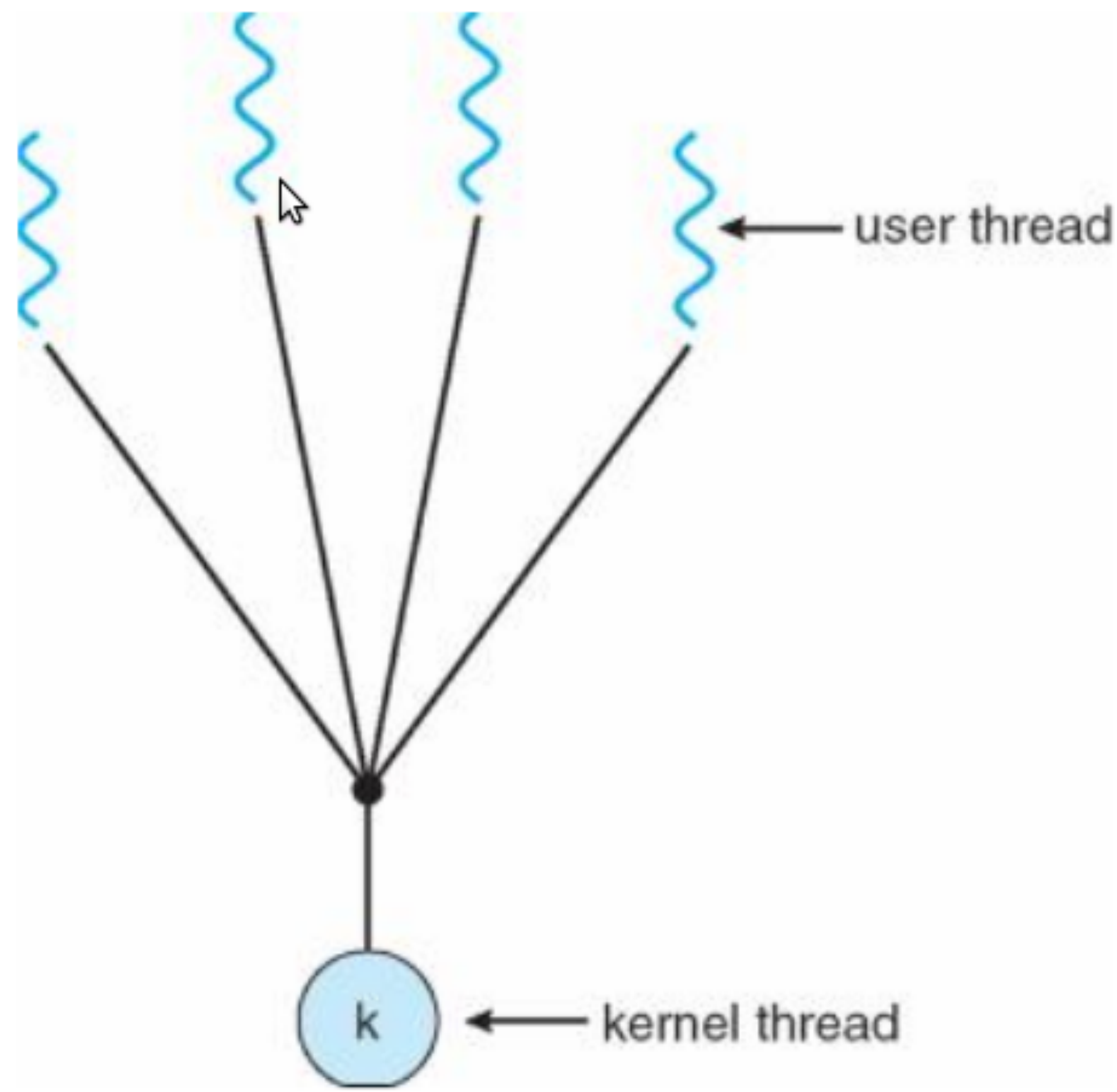





AARHUS UNIVERSITY

# Threading Model

- User-Level Threading
  - Simple implementation no kernel support for threads
  - Very quick thread context switch (no kernel handling needed)
  - Not possible to handle multicores

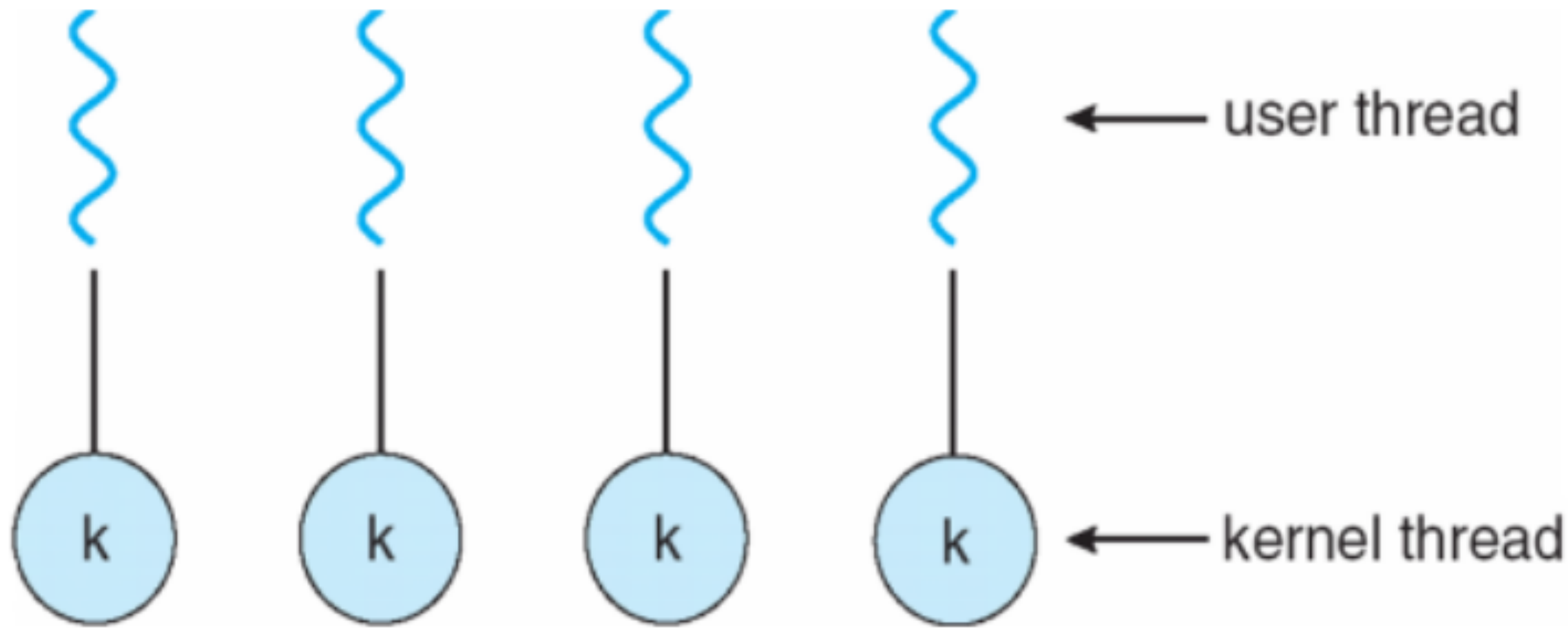





AARHUS UNIVERSITY

# Threading Model

- Kernel Level Threading
  - Need thread awareness in kernel
  - Maps directly to threads which the scheduler can control
  - Efficient multicore usage
- OS
  - Linux (NPLT), Win32 etc.

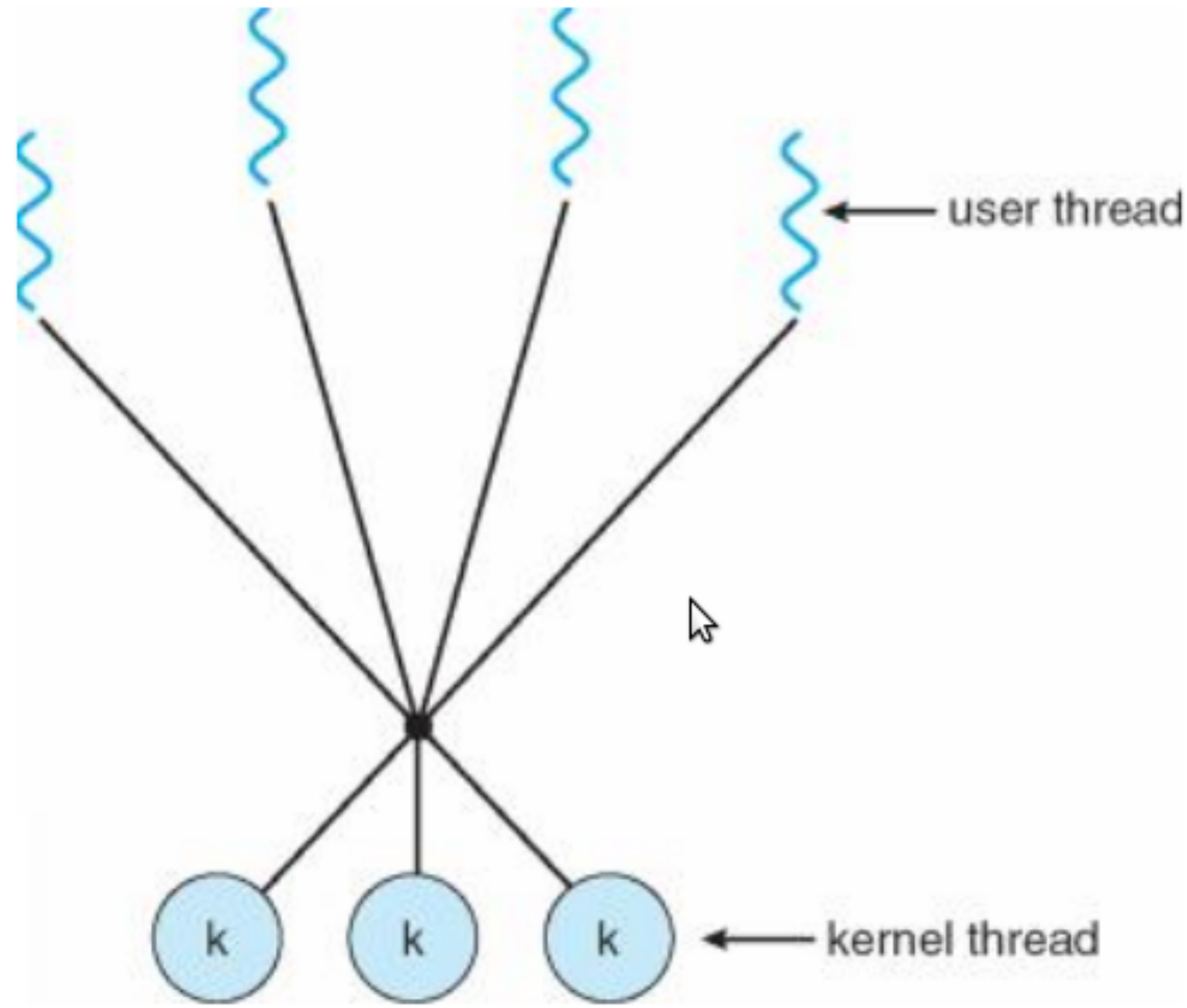





AARHUS UNIVERSITY

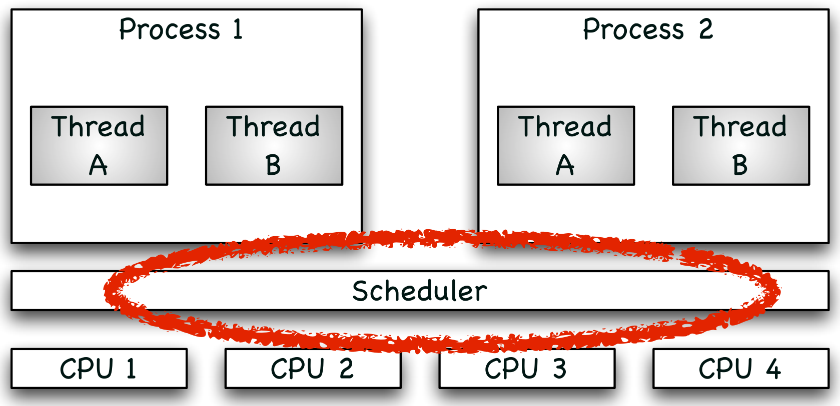
# Threading Model

- Hybrid Level Threading
  - Complex implementation
  - Requires good coordination between user land and kernel land scheduler
    - Otherwise suboptimal resource usage
- OS
  - Windows 7



# Context switching

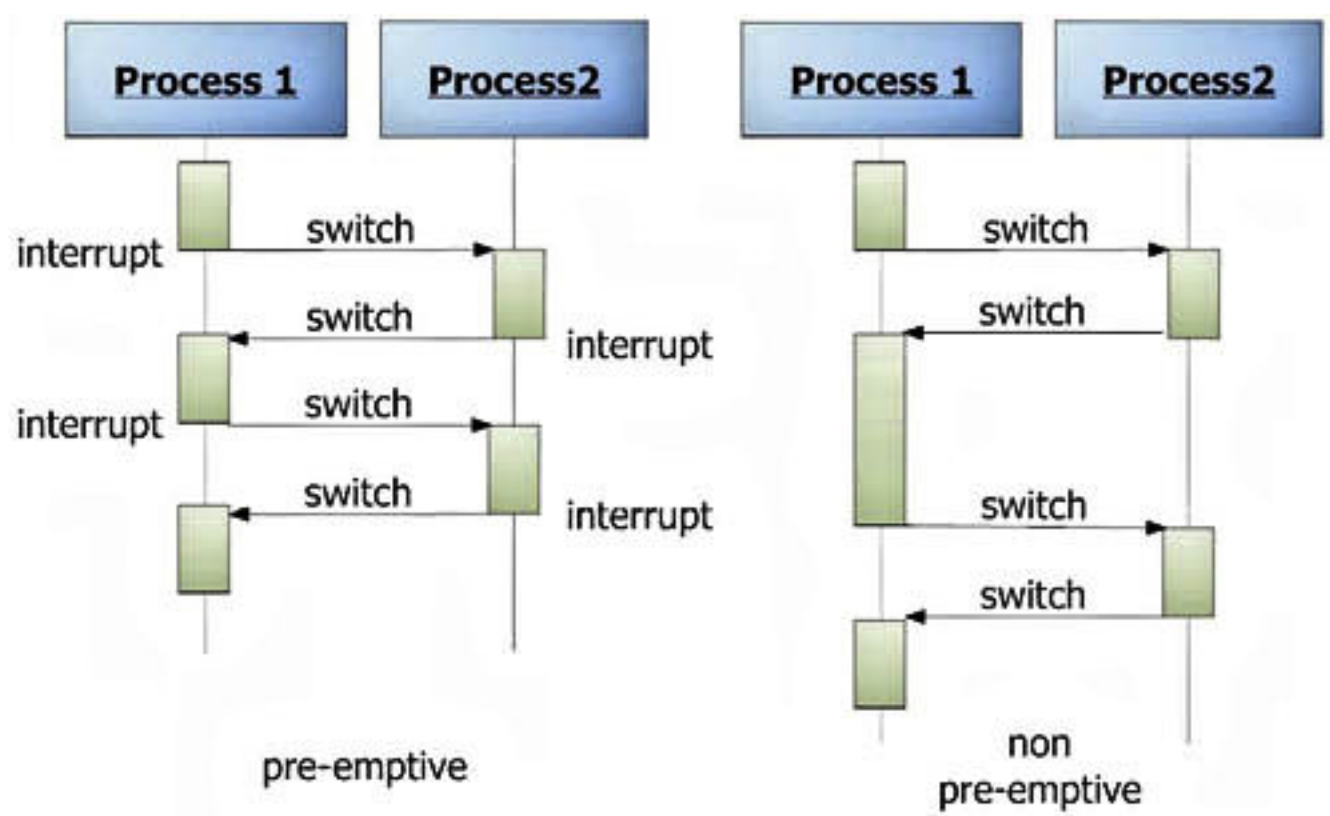
- A *context* is the environment of the currently running process
- A context switch is performed by the OS to suspend the currently running process and resume another process
- General steps:
  - Interrupt current process
  - Save context of current process (SP, PC, registers, …)
  - Restore context of next process
  - Resume execution of next process



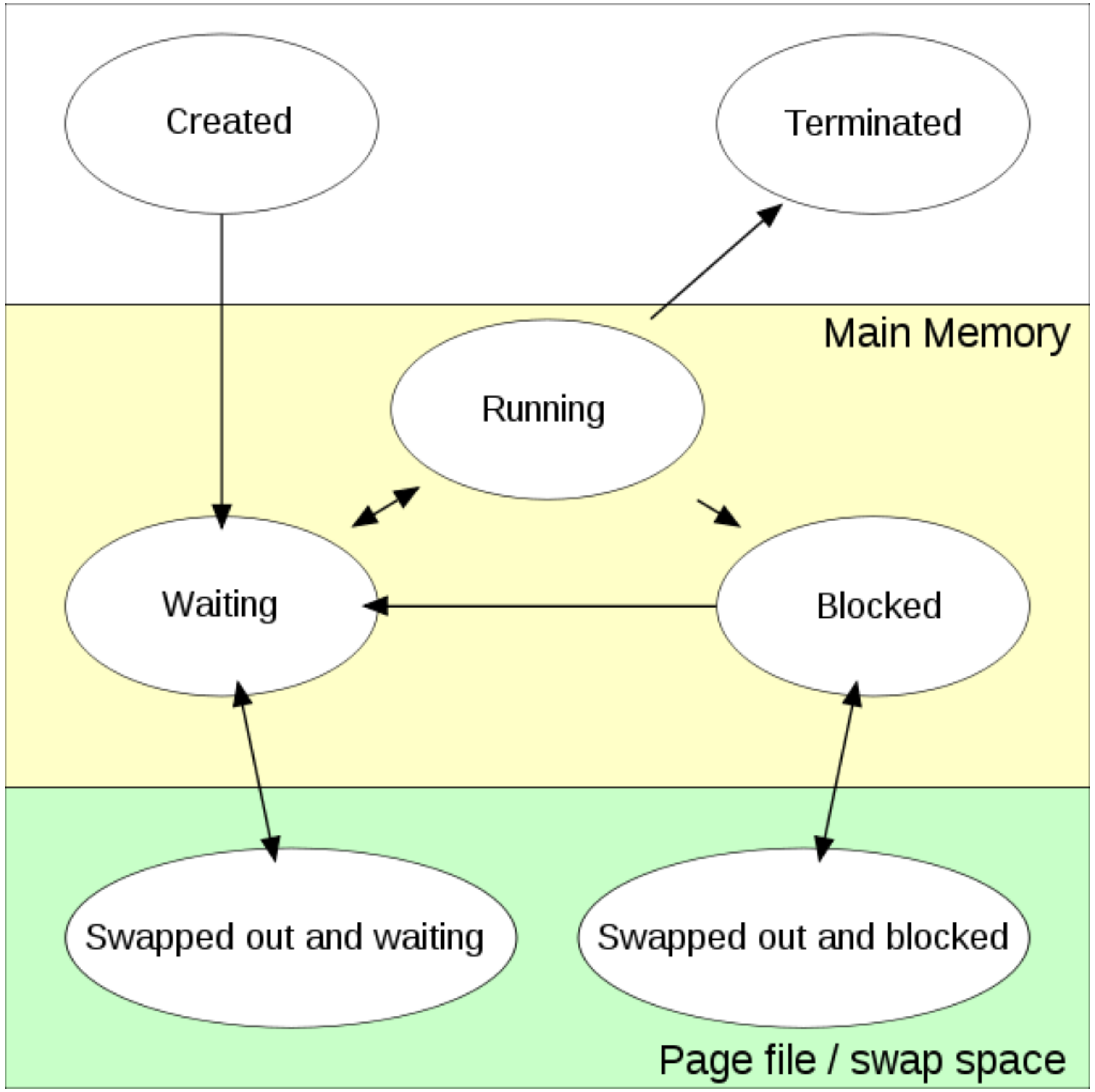
# Context switching

- The *operating system* schedules tasks for execution
  - Preemptive scheduling: Tasks can be interrupted at any time
  - Nonpreemptive scheduling: Tasks voluntarily yield the CPU

- *Linux supports both - Kernel configuration options*



# The life and death of a process



# Threads - Summary

- Threads are *strands of execution* – each process has at least one thread
  - AKA *light-weight processes*
  - Also called tasks or jobs

- Threads of the same process share memory space (e.g. global variables)

- Threads *can harm* each other

- We will often work with several *threads* in the same *process*

# Processes - Summary

- A process is an instance of a program that is being executed
  - Image of program (segment),
  - Stack, heap, registers, file descriptors, …
- Processes have their own individual memory spaces
  - Process A cannot write in memory of process B – they are safe from each other
- Processes may only communicate through IPC mechanisms controlled by the OS.
- Processes may spawn other processes, which may execute the same or other programs
- A process may also spawn threads within its own memory space

# Multitasking systems: Advantages

- What are the advantages of multiple tasking a system?
  - Prioritization – the highest-priority task gets to run
  - Modularization – wrap concurrent activities in a task
  - Resource utilization – Don’t spend CPU time waiting for I/O etc.
  - …
- However, the use of multiple tasks in a system creates a number of problems for us
- We must know the problems to be able to identify and counter or avoid them

# Multithreaded systems: Disadvantages

- Consider the following code. What is the value of shared after 10 seconds?

```
unsigned int shared;
void taskfunc()
{
   while(true)
   {
    shared++;                  // Increment i, then wait
    sleep(ONE_SECOND);         // 1 second
   }
}

int main()
{
   shared = 0;
   createThread(taskFunc);   // Start two identical threads
   createThread(taskFunc);   // that run the same function
   while(true)
     sleep();
}
```



AARHUS UNIVERSITY

# The shared data problem

- Let’s zoom in:

Program

```
while(true)
{
   shared++;
   sleep(ONE_SECOND);
}
```

Pseudo-machine instructions

```
LOAD R1, shared
INC R1
STORE shared, R1
```

Task 1

```
LOAD R1, shared
INC R1
STORE shared, R1
```

Task 2

```
LOAD R1, shared
INC R1
STORE shared, R1
```



AARHUS UNIVERSITY

# The shared data problem

Task 1

```
LOAD R1, shared
INC R1
STORE shared, R1
```

Task 2

```
LOAD R1, shared
INC R1
STORE shared, R1
```

What value is shared suppose to have?

Non-interleaved instructions

```
LOAD R1, shared    // shared = 0
INC R1
STORE shared, R1   // shared = 1
LOAD R1, shared    // shared = 1
INC R1
STORE shared, R1   // shared = 2
```

Interleaved instructions

```
LOAD R1, shared    // shared = 0
LOAD R1, shared    // shared = 0
INC R1
STORE shared, R1   // shared = 1
INC R1
STORE shared, R1   // shared = 1
```

Which values can shared in fact have?

# The shared data problem

```
Position pos { 10, 20, 30 };
```

Task 1

```
void newPos(Position& ps, float x, float y, float z)
{
  pos.x = x;
  pos.y = y;
  pos.z = z;
}
```

Task 2

```
void printPos(Position& pos)
{
  std::cout << "X: " << pos.x << std::endl;
  std::cout << "Y: " << pos.y << std::endl;
  std::cout << "Z: " << pos.z << std::endl;
}
```

Non-interleaved instructions

```
T1  pos.x = x;
T1  pos.y = y;
T1  pos.z = z;

T2  std::cout << "X: " << pos.x << std::endl;
T2  std::cout << "Y: " << pos.y << std::endl;
T2  std::cout << "Z: " << pos.z << std::endl;
```

newPos called with 11, 22, 33

std::cout prints
X: 10
Y: 20
Z: 30

Interleaved instructions

```
T1  pos.x = x;

T2  std::cout << "X: " << pos.x << std::endl;
T2  std::cout << "Y: " << pos.y << std::endl;
T2  std::cout << "Z: " << pos.z << std::endl;

T1  pos.y = y;
T1  pos.z = z;
```

Pseudo'ish code

std::cout prints
***X: 11***
Y: 20
Z: 30

***Hey WHAT???***

# The shared data problem

- The shared data problem is inherent in any preemptive multithreaded system

- Very cumbersome to find a good software solution to the problem
  - Peterson's solution: 2 interest flags, 1 will-wait flag
  - Does not scale

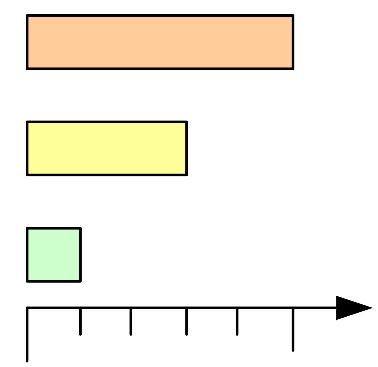
- We need a way to define critical sections of program
  - Sections in which the thread is guaranteed to be allowed to execute uninterrupted

- In a later lecture!

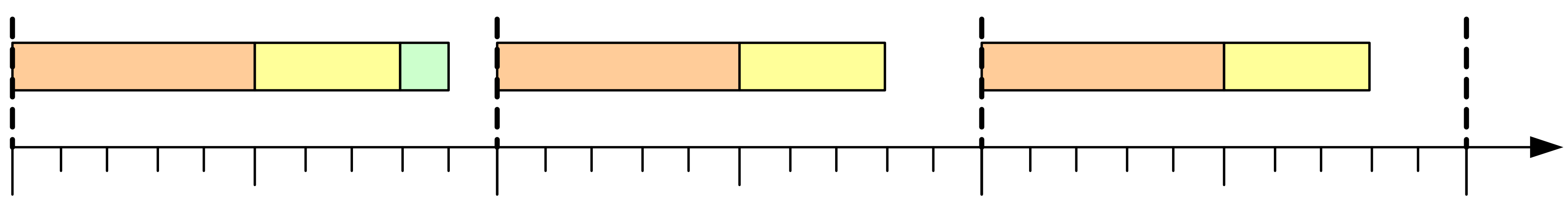
```
void taskfunc()
{
   while(true)
   {
    enterCriticalSection();
    shared++;
    exitCriticalSection();
    sleep(ONE_SECOND);
   }
}
```

# Multithreaded systems: Disadvantages

- Consider a system with three threads, HP, MP and LP:
  - HP takes 5 μs, must run once every 10 μs
  - MP takes 3 μs, must run once every 10 μs
  - LP takes 1 μs, must run once every 1000 μs



- Schedule?



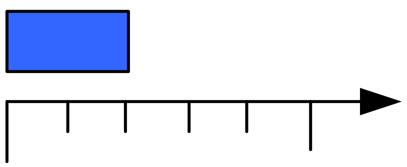
OK!



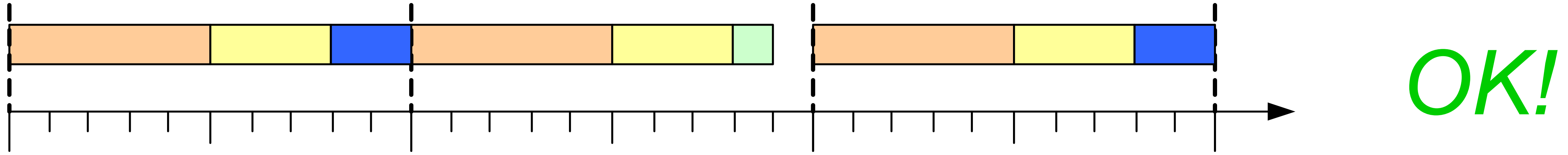
AARHUS UNIVERSITY

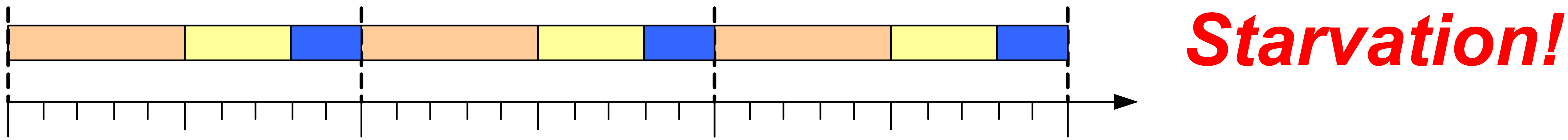
# Starvation

- Assume we add another MP thread, $MP_2$
  - $MP_2$ takes 2 μs, may run once every 10 μs



- Schedule?





# Starvation

- Starvation is an inherent problem in any priority-based system
- It occurs when the schedule is so tight that LP threads are never allowed to run because higher-priority threads “hog” the CPU
- Starvation can be very hard to predict and detect
  - Might only occur in very special situations

# Programming with threads

- C++ does not have the concept of threads built in
  - Relies on 3rd party libraries for this
  - *Not entirely true see new ratified C++ standard*

- The POSIX (Portable OS Interface for uniX) library is the most widely used threading library
  - Others include boost, Qt, …

- The POSIX library has the thread type pthread which we will use.
  - Include pthread.h, link with library pthread

# PThread functions

- Family of pthread functions you are to use...
  - pthread_create()
  - pthread_join()
  - pthread_exit()
  - pthread_* (and more)
- How they work is for next session