PARALLEL PROGRAMS, PROCESSES AND THREADS

AGENDA

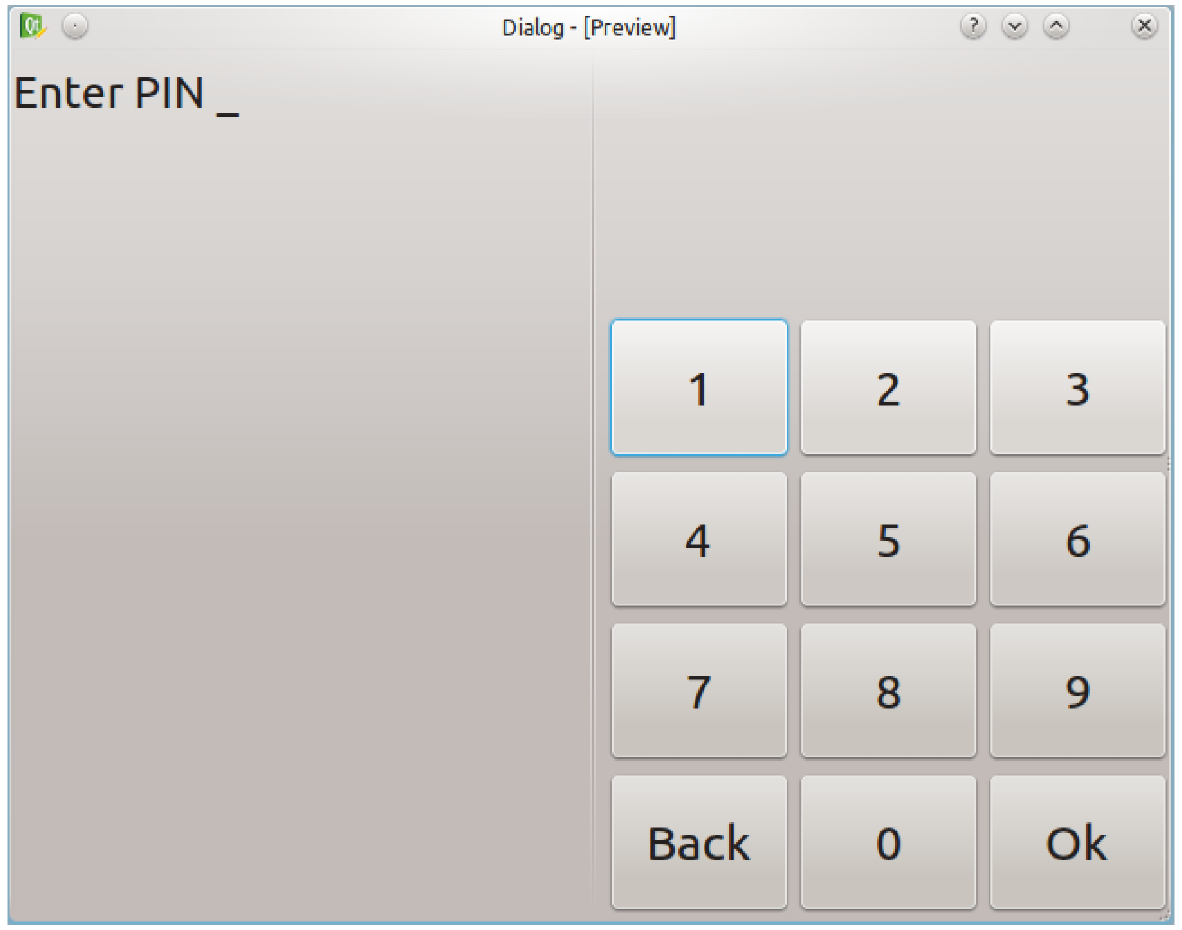
- The problem - A Case
- A Solution - Parallelism
- Processes and Threads in Linux
- Advantages & Disadvantages with multitasking

CASE

CASE

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

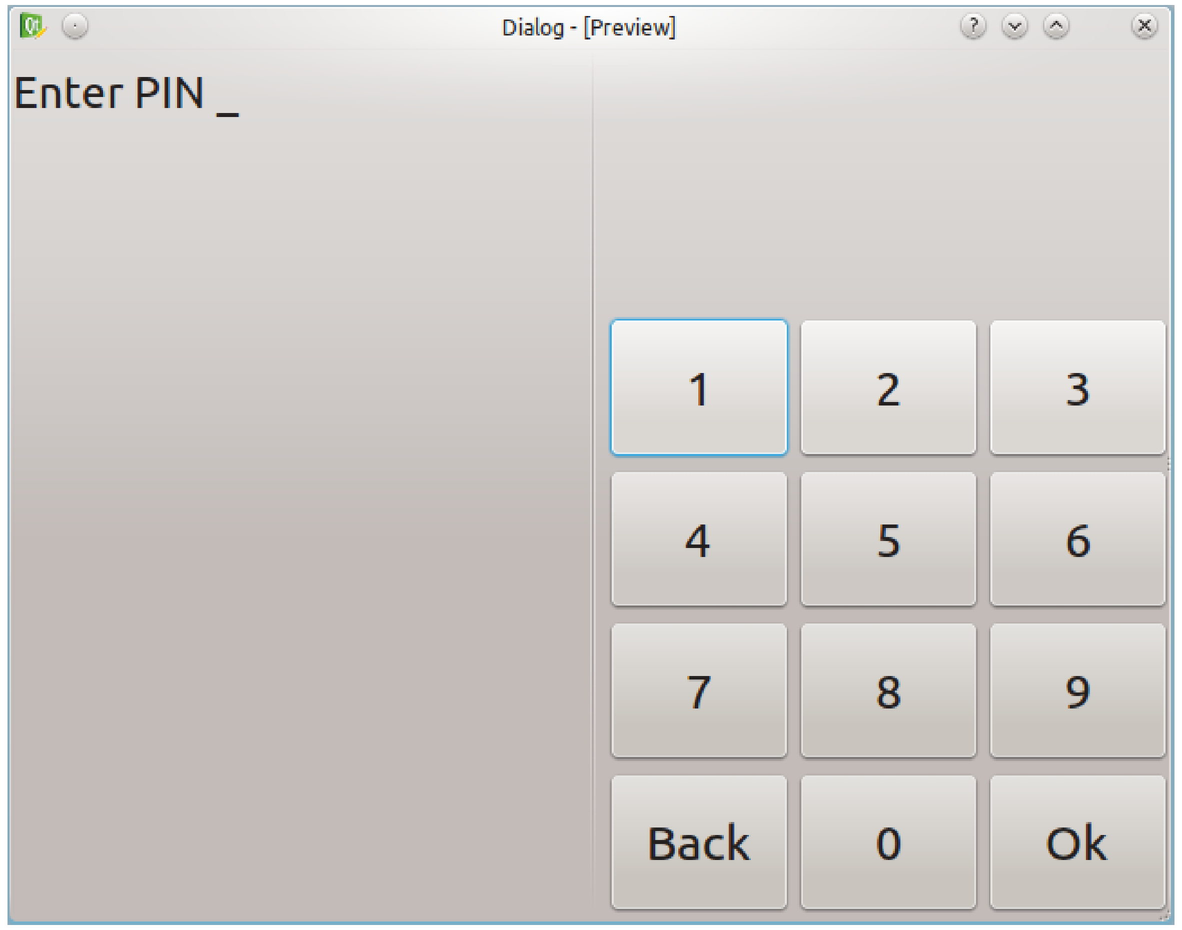
BANK TERMINAL



CASE

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

BANK TERMINAL



POSSIBLE IMPLEMENTATION

```
void main()
{
  print("ENTER PIN:");
  ch = getKey();
  while(ch != "OK")
  {
    input += ch
    compare(input, pin)
    ...
}
```

CASE

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

BANK TERMINAL

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

CASE

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

BANK TERMINAL

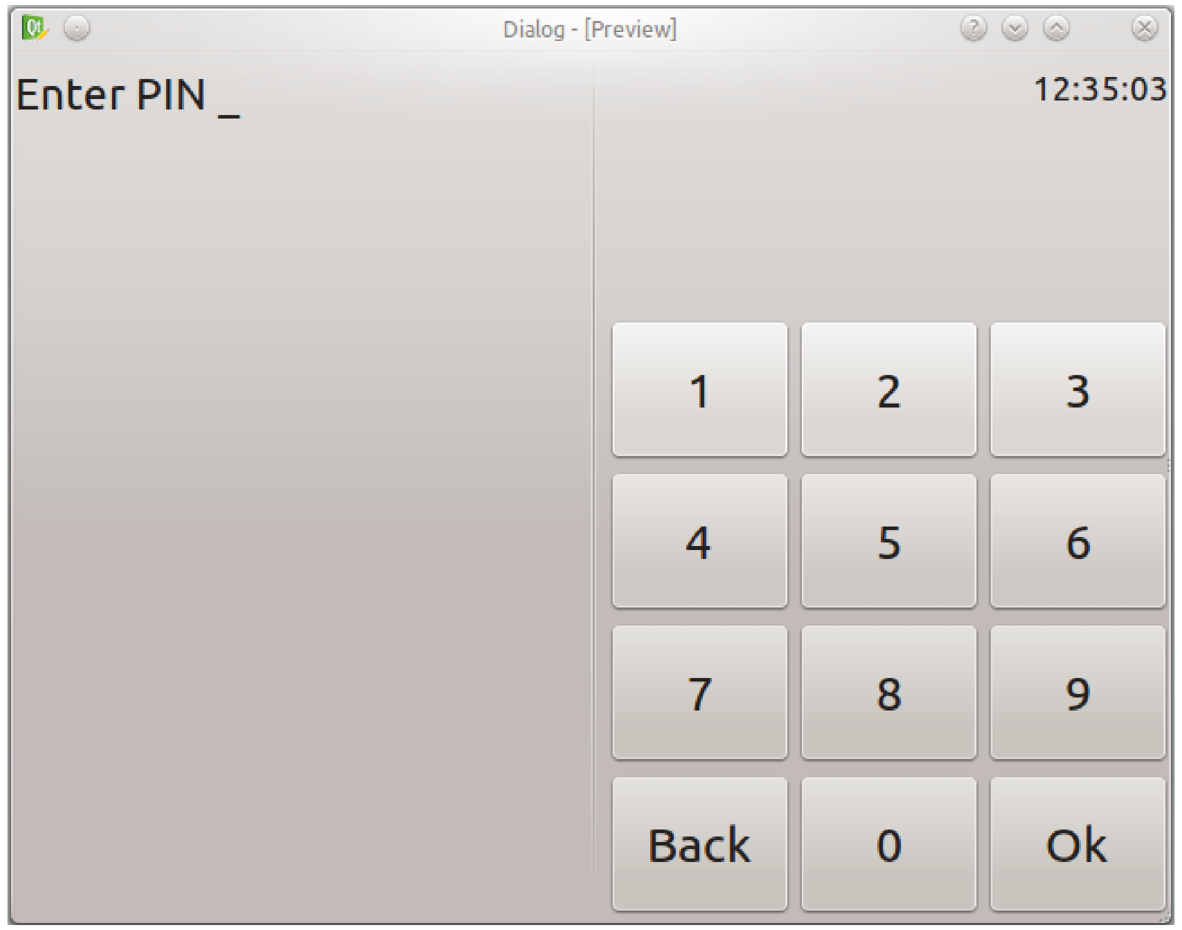
POSSIBLE IMPLEMENTATION

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

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

CASE - SOLUTION

BANK TERMINAL



CASE - SOLUTION

BANK TERMINAL

USER INPUT THREAD

- Waiting on blocking call

```
void userInput()
{
  print("ENTER PIN:");
  ch = getKey();
  while(ch != ”OK”)
  {
    input += ch
    compare(input, pin)
    ...
  }
}
```

CASE - SOLUTION

BANK TERMINAL

USER INPUT THREAD

- Waiting on blocking call

```
void userInput()
{
  print("ENTER PIN:");
  ch = getKey();
  while(ch != ”OK”)
  {
    input += ch
    compare(input, pin)
    ...
  }
}
```

CLOCK/TIME THREAD

- Delay is blocking

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

CASE - SOLUTION

BANK TERMINAL

USER INPUT THREAD

- Waiting on blocking call

```
void userInput()
{
  print("ENTER PIN:");
  ch = getKey();
  while(ch != ”OK”)
  {
    input += ch
    compare(input, pin)
    ...
  }
}
```

CLOCK/TIME THREAD

- Delay is blocking

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

STARTS THREADS

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

PARALLEL PROGRAMS - A SOLUTION

- *Parallel programs* are programs
  - where the work done is parallelised
- Parallelisation can take the form of
  - Multiple processes working in conjunction
  - Multiple threads, jobs or tasks in process working in conjunction

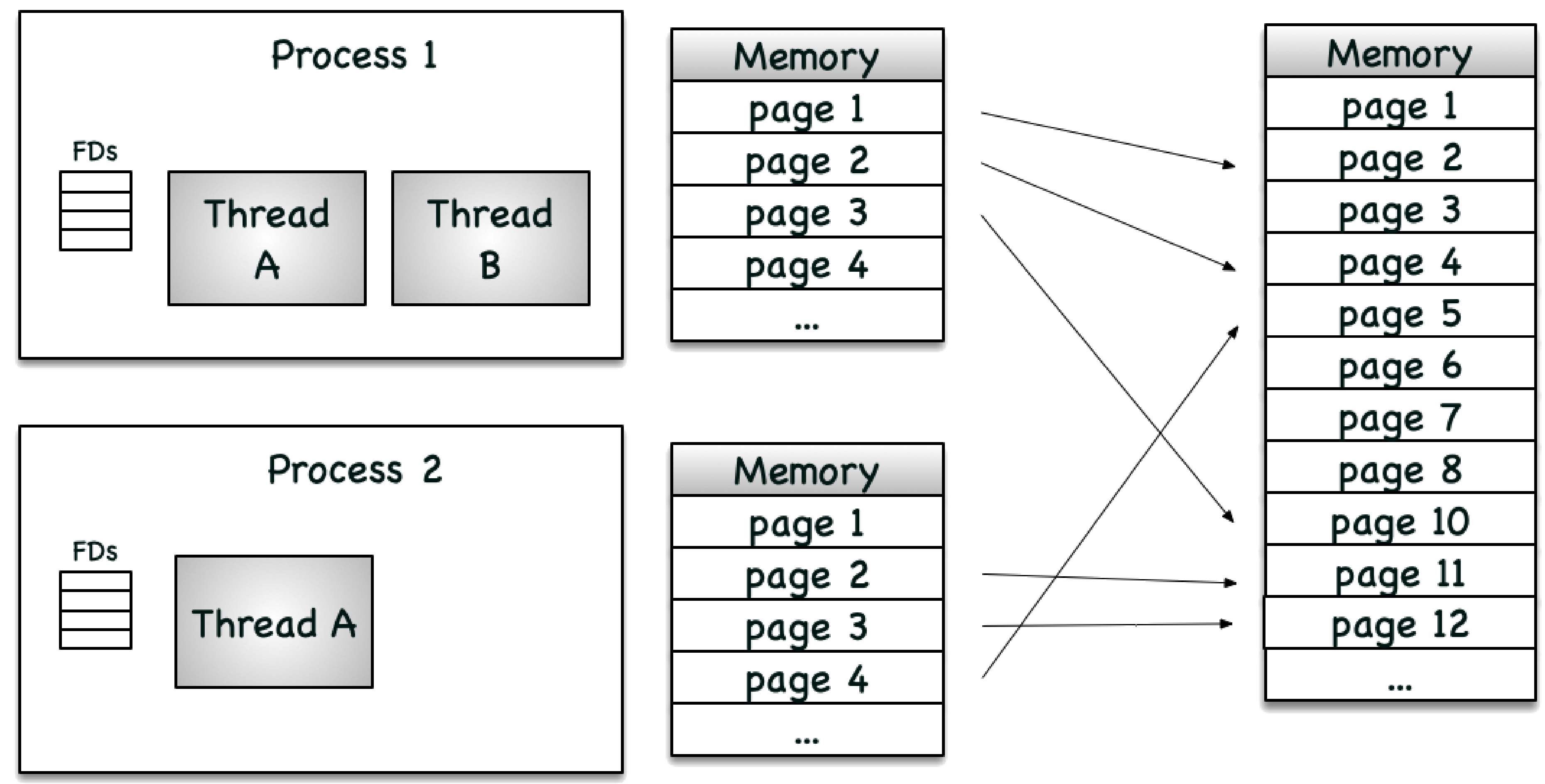
PROCESS & THREADS IN LINUX

PROCESS & THREADS IN LINUX

- Processes and the OS (Kernel)
- Process anatomy
- Shared memory

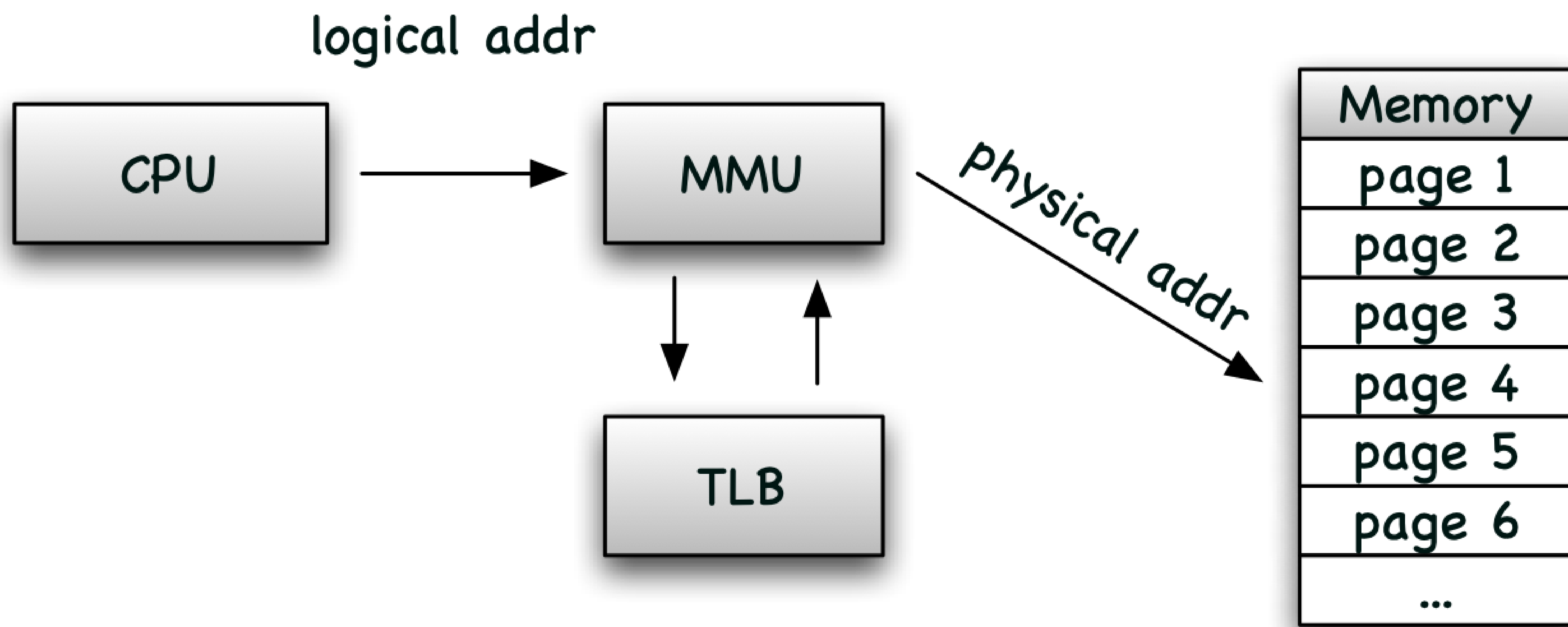
PROCESSES AND THE OS (KERNEL)

- 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

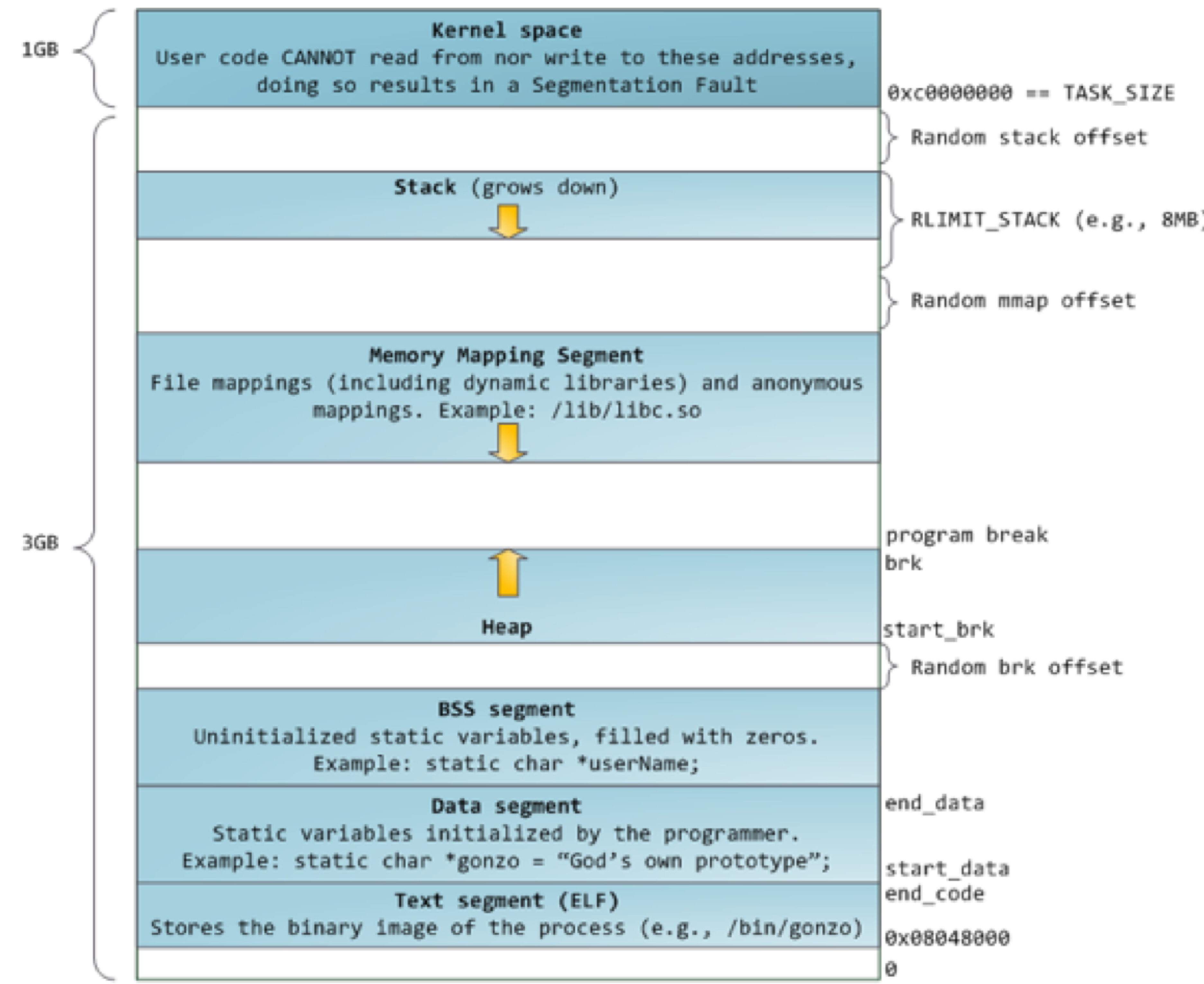


MMU: Memory Management Unit
TLB: Translation Lookaside Buffer

PROCESS ANATOMY

PROCESS - A PROGRAM BEING EXECUTED IN LINUX

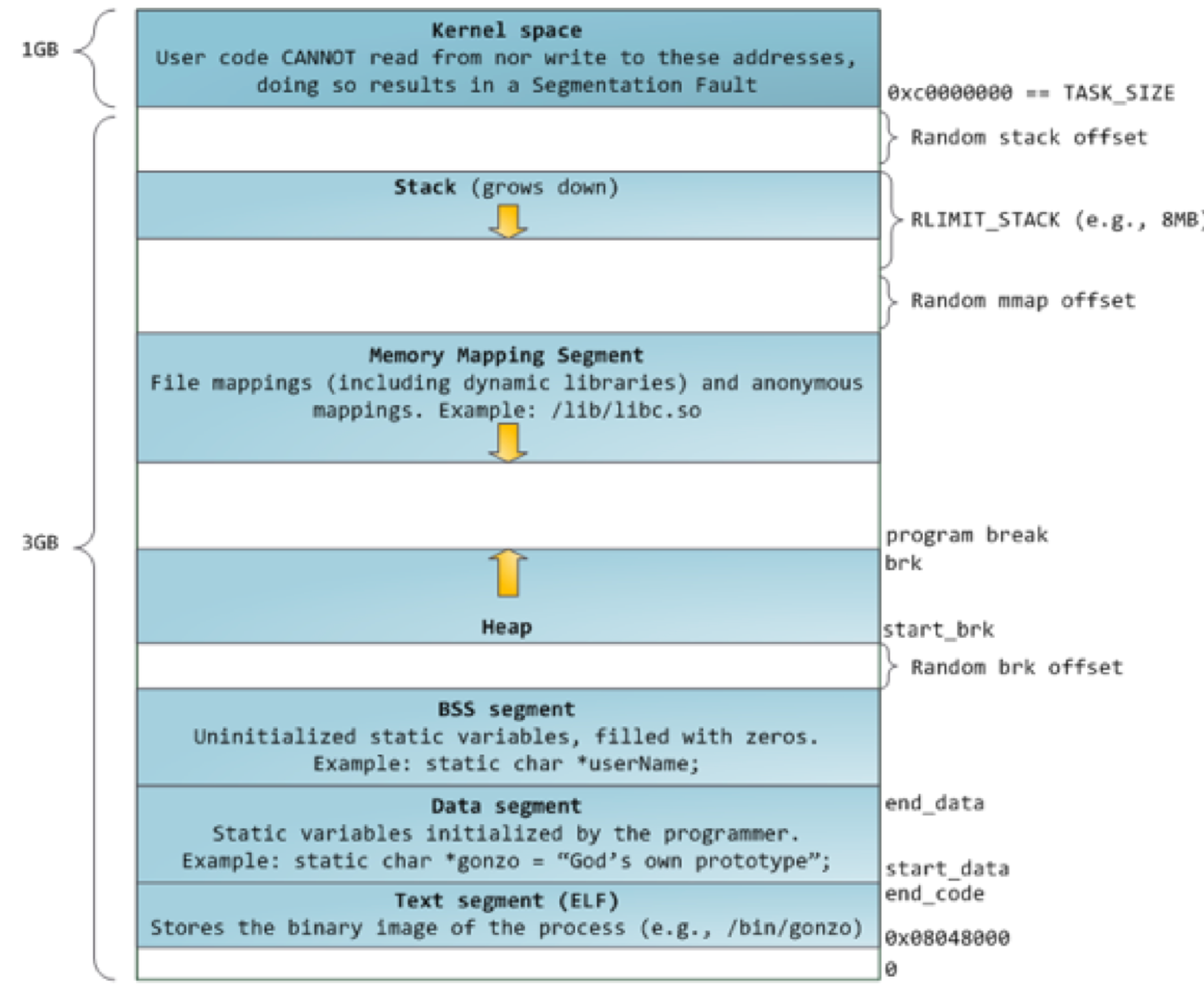
- Stack
  - Local variables
  - Function return values
  - LIFO



PROCESS ANATOMY

PROCESS - A PROGRAM BEING EXECUTED IN LINUX

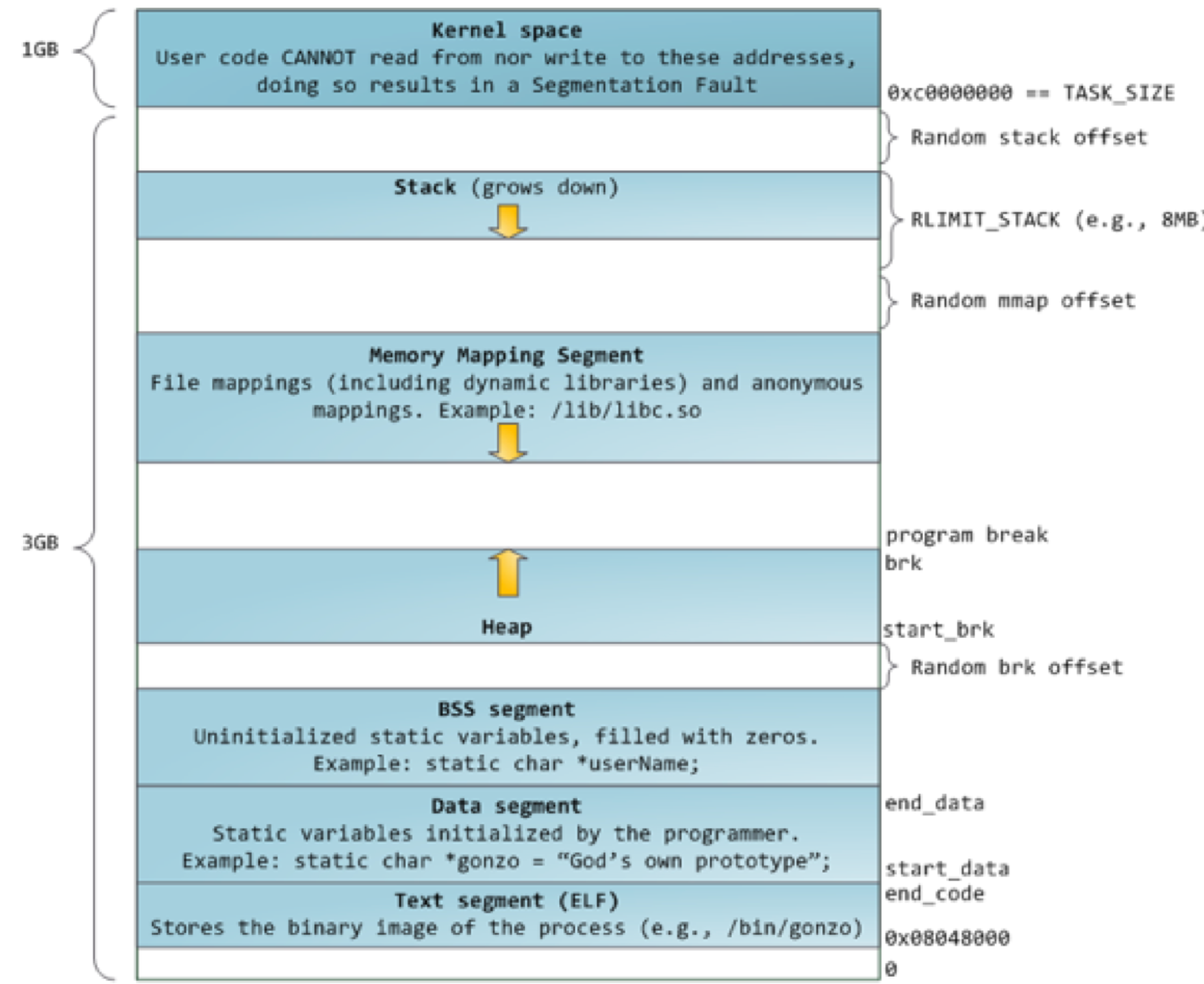
- Stack
  - Local variables
  - Function return values
  - LIFO
- Heap
  - "Free-store"
  - Dynamically allocated memory



PROCESS ANATOMY

PROCESS - A PROGRAM BEING EXECUTED IN LINUX

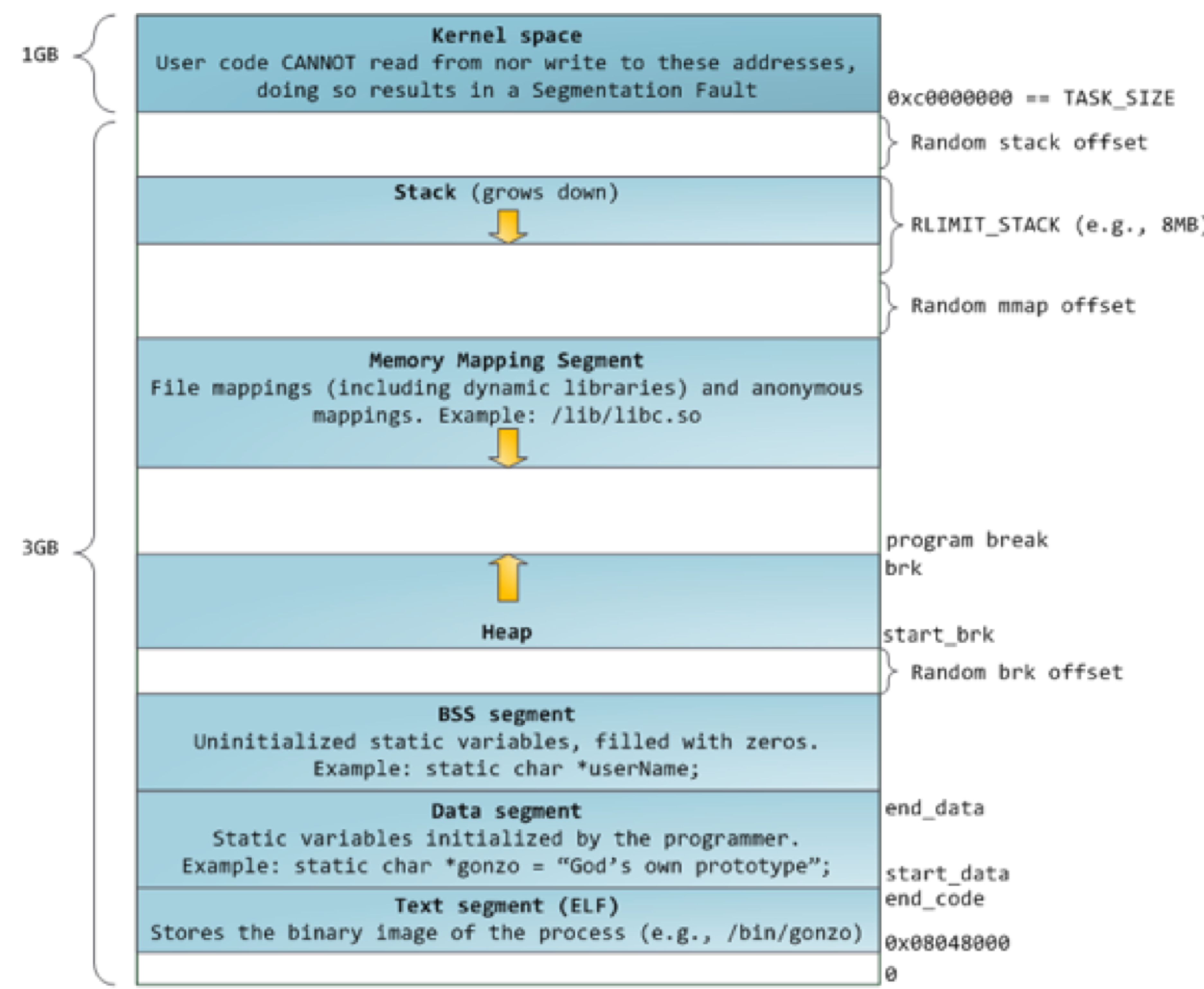
- Stack
  - Local variables
  - Function return values
  - LIFO
- Heap
  - "Free-store"
  - Dynamically allocated memory
- Memory Mapping
  - File mapped in memory
  - Includes dyn libs
  - Sharing memory between processes



PROCESS ANATOMY

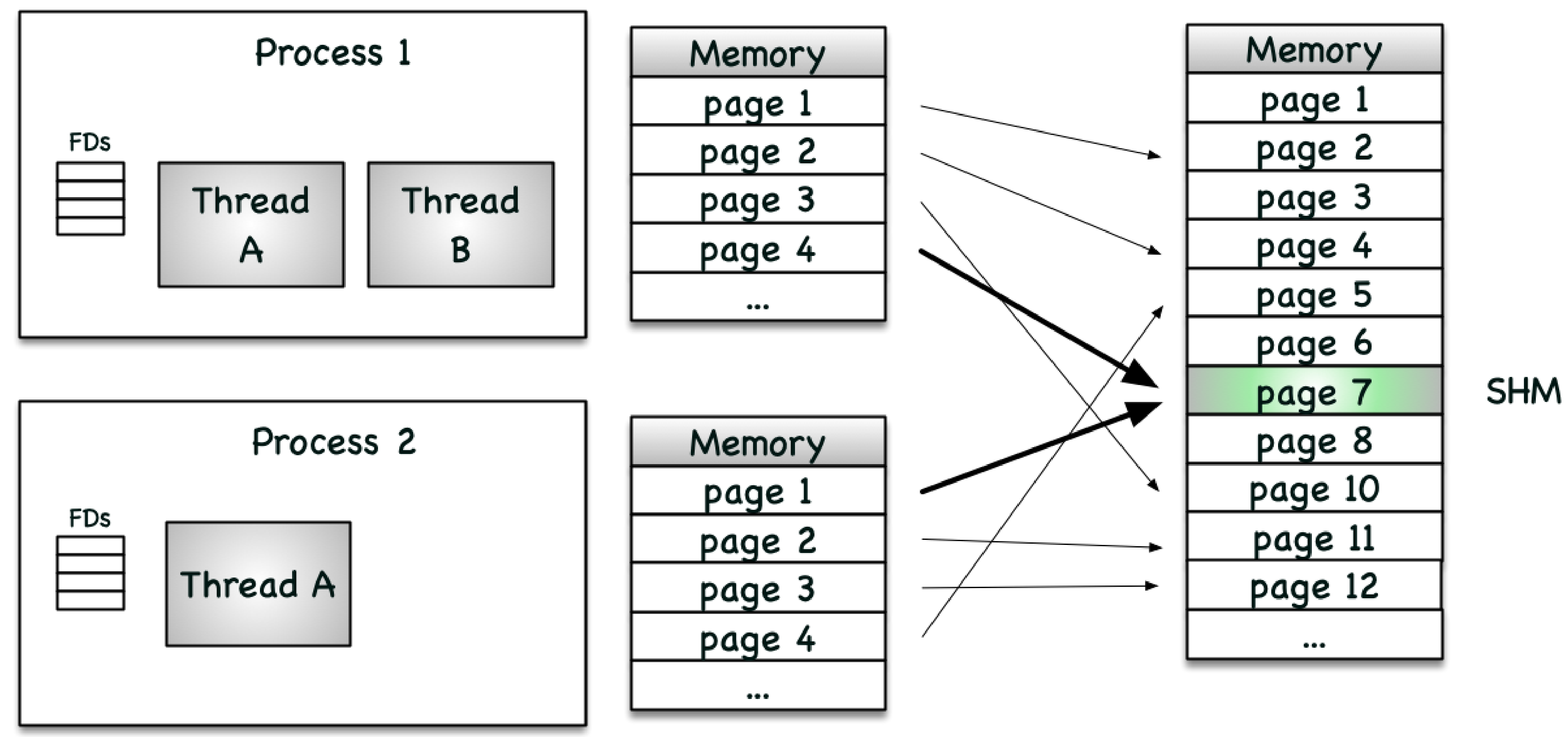
PROCESS - A PROGRAM BEING EXECUTED IN LINUX

- 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 processes
- Variables and ELF



SHARED MEMORY

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



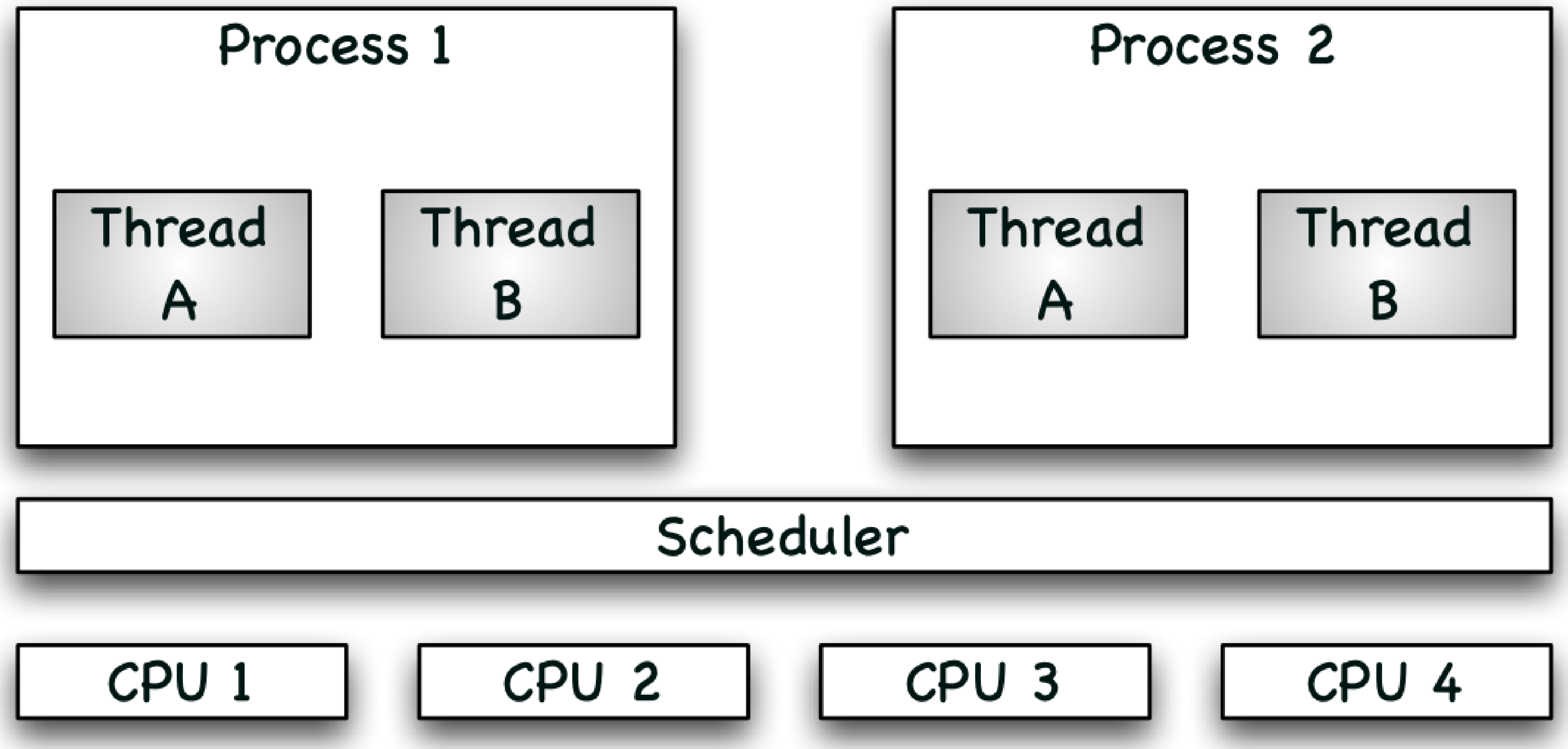
THREADING MODELS

THREADING MODELS

- Threads of execution and the OS (Kernel)
- User level threading
- Kernel level threading
- Hybrid level threading

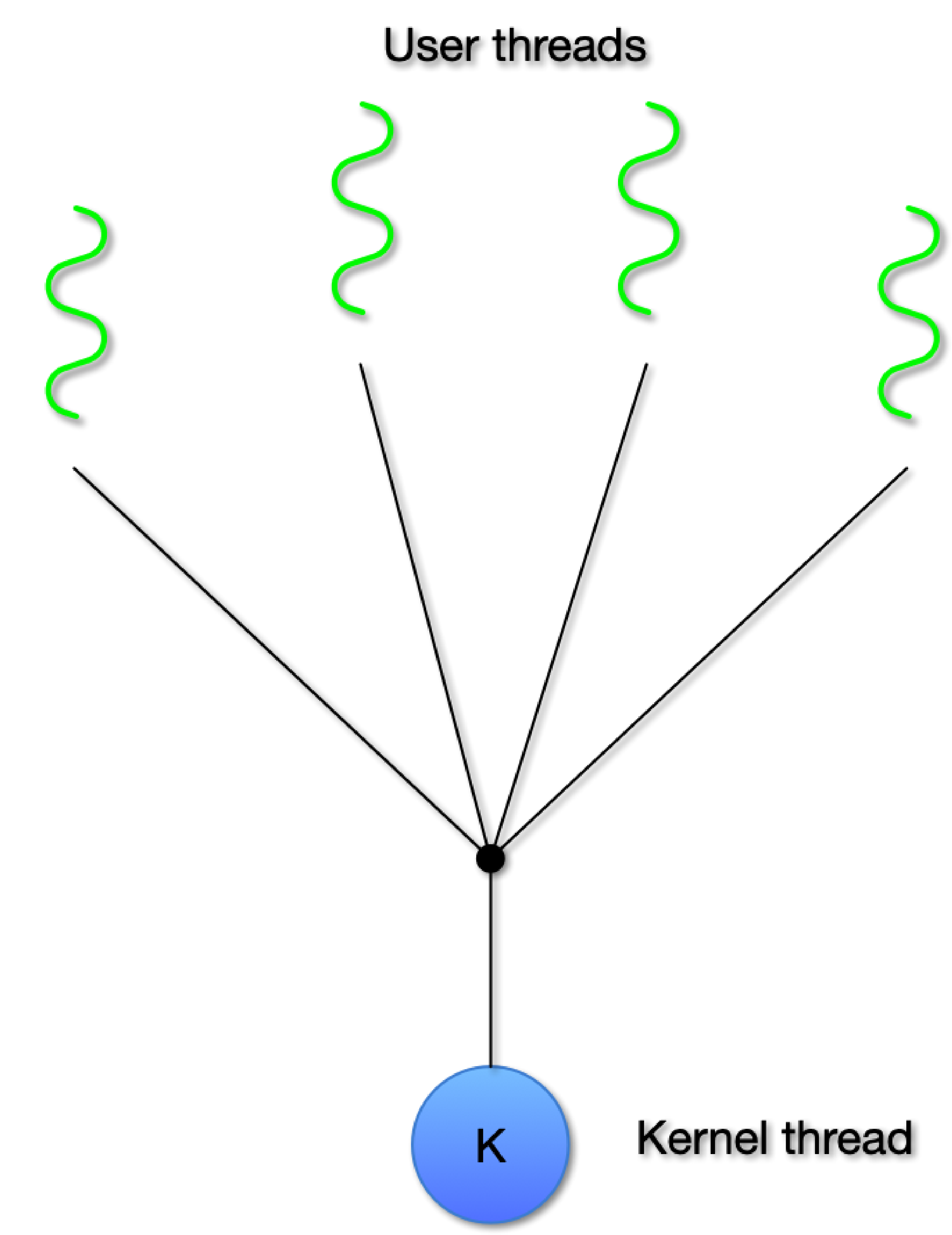
THREADS OF EXECUTION AND THE OS (KERNEL)

- How are threads mapped for execution?
  - By the scheduler
  - Using one of three different models
    - User level threading
    - Kernel level threading
    - Hybrid level threading



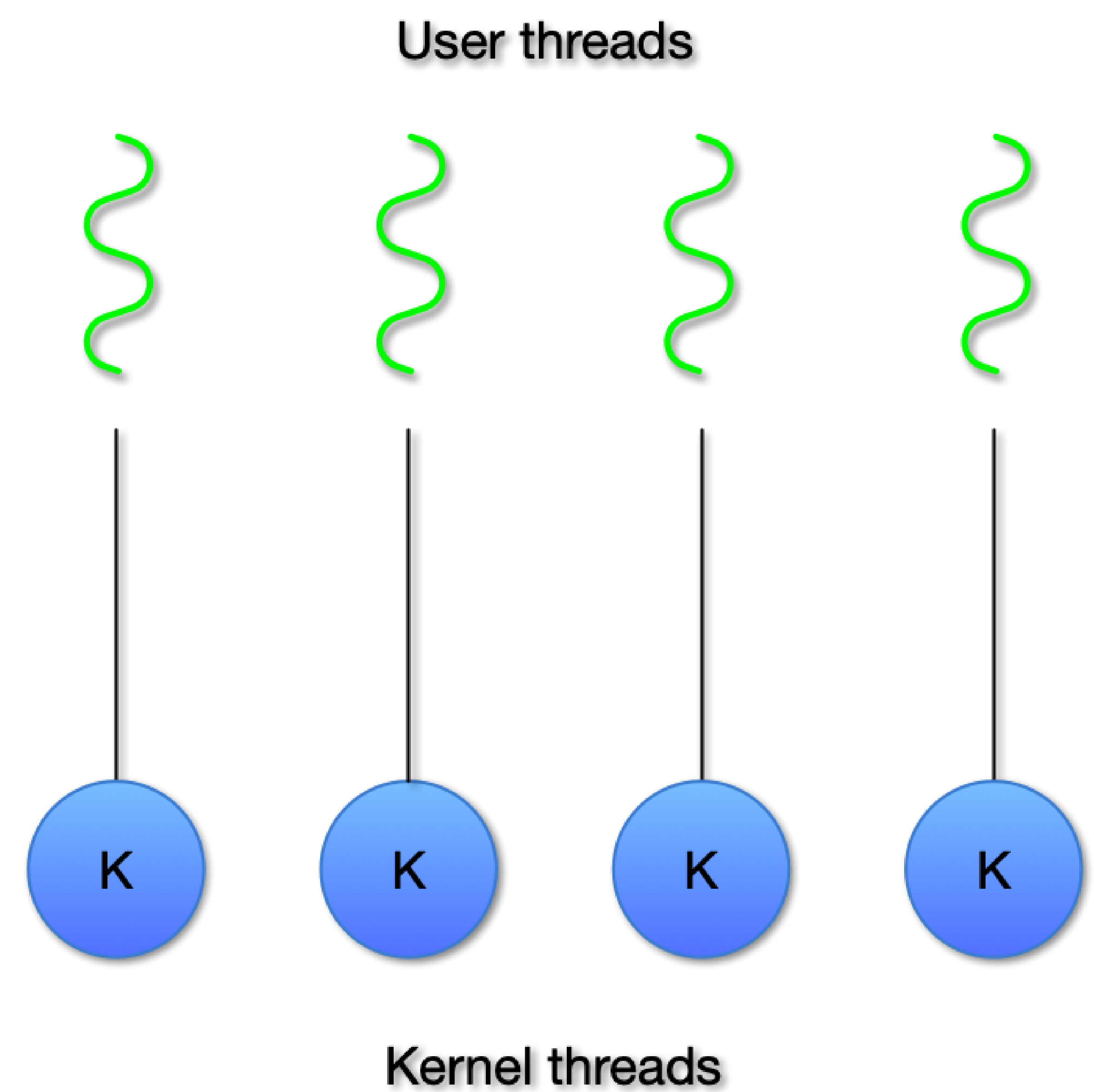
USER-LEVEL THREADING

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



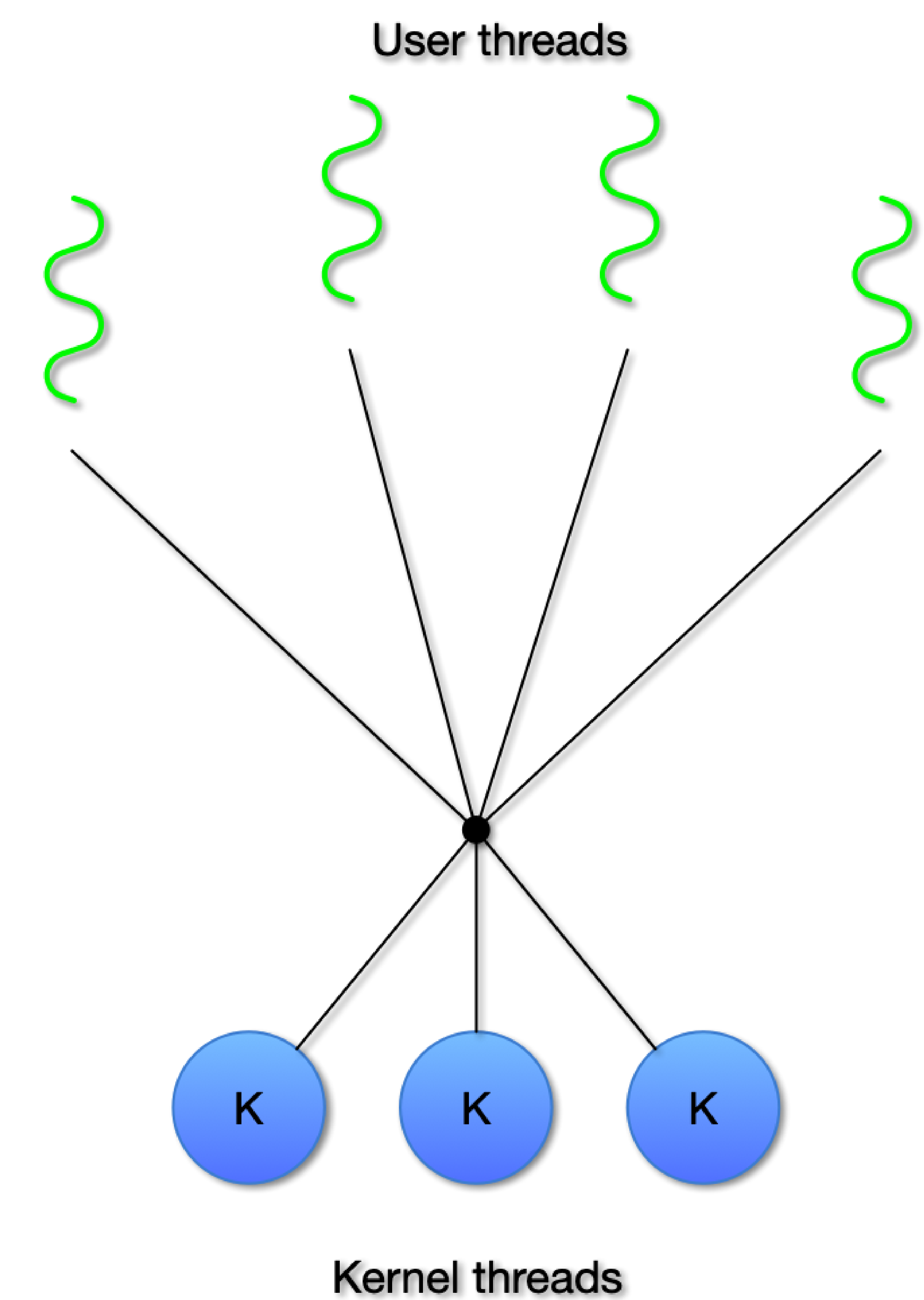
KERNEL LEVEL THREADING

- 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.



HYBRID LEVEL THREADING

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



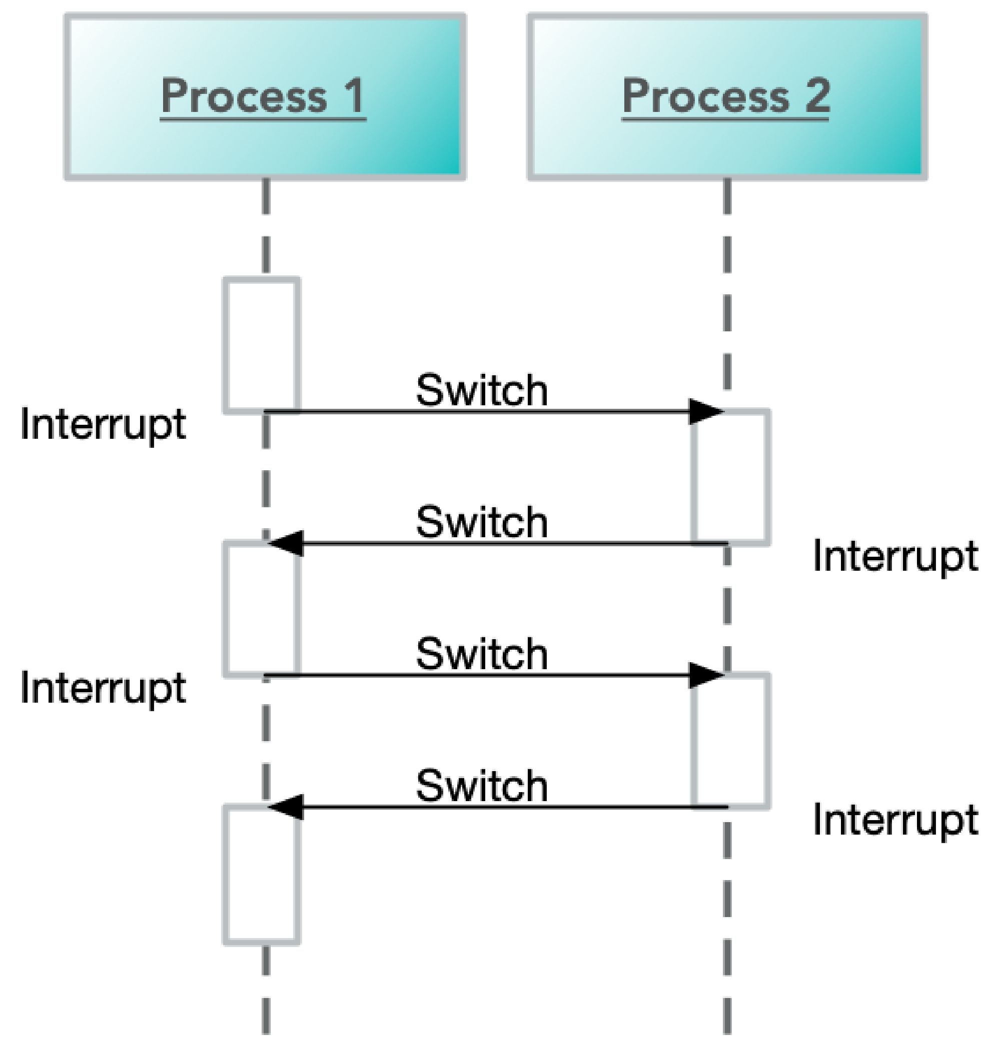
CONTEXT SWITCHING

CONTEXT SWITCHING

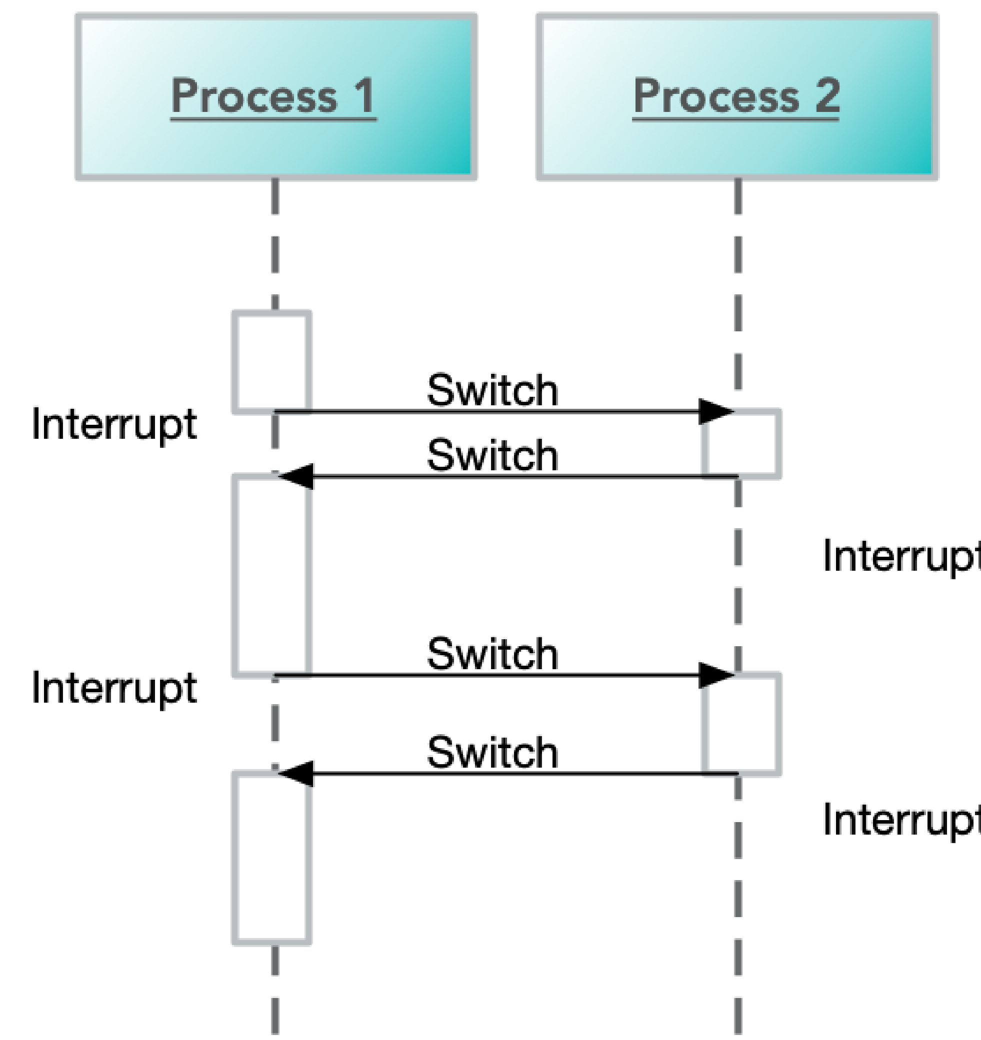
- A context is the environment of the currently running process
- A context switch is performed by the OS to suspend the currently running process and resume another process
- General steps:
  - Interrupt current process
  - Save context of current process (SP, PC, registers, ...)
  - Restore context 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
  - Non-preemptive scheduling: Tasks voluntarily yield the CPU
- Linux supports both - Kernel configuration options



pre-emptive

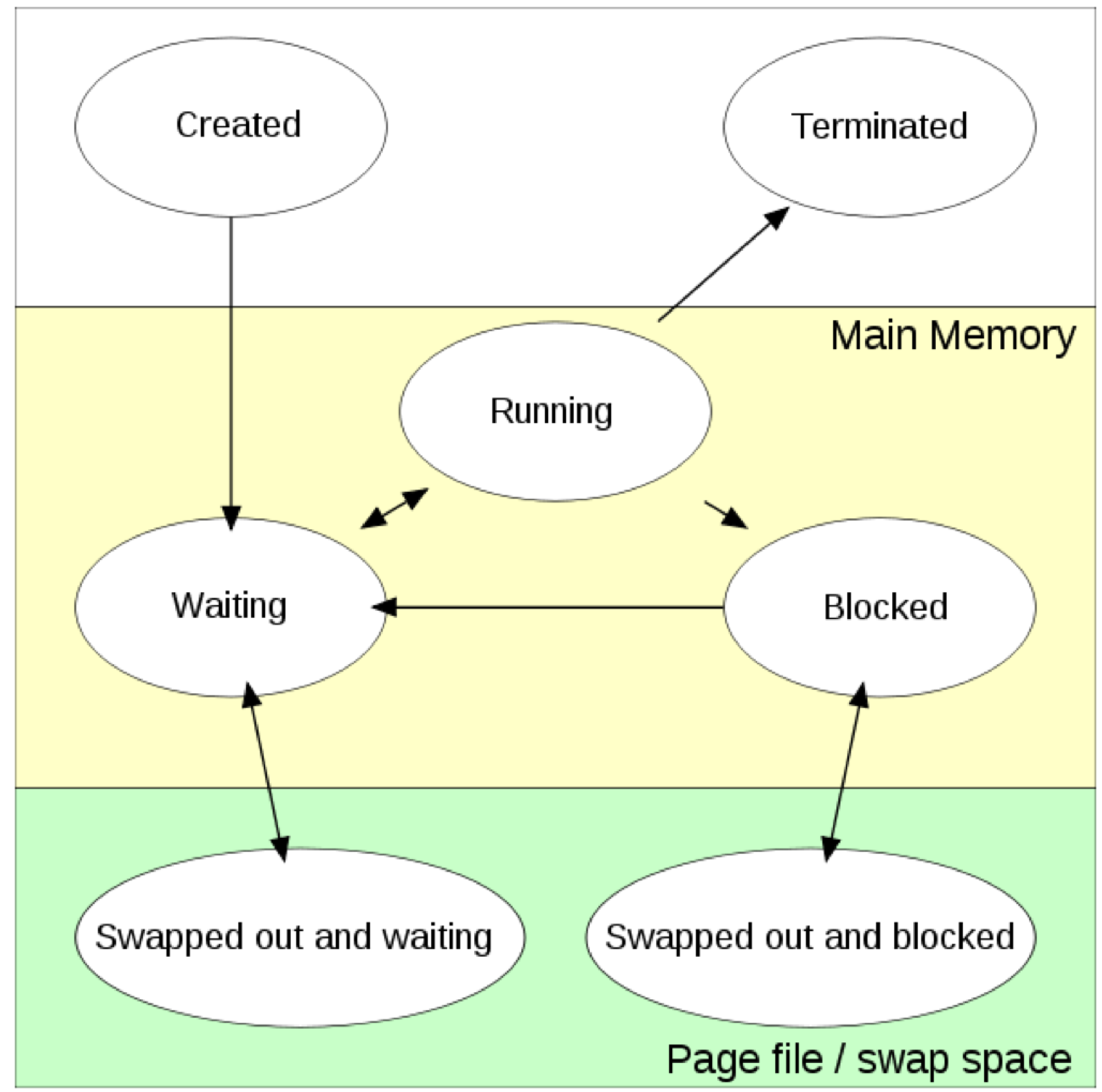


non pre-emptive

THE LIFE AND DEATH OF A PROCESS

THE LIFE AND DEATH OF A PROCESS

- Waiting
  - In queue to run on processor
- Running
  - Running on the processor
- Blocked
  - Waiting on
    - Mutex
    - File
    - Connection...
- Swapped out ...
  - Placed on disk temporarily



MULTI-THREADED SYSTEMS

MULTI-THREADED SYSTEMS

- Advantages
- Disadvantages
  - Shared data
  - Starvation

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

DISADVANTAGES

SHARED DATA

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();
}
```

DISADVANTAGES

SHARED DATA

Zooming in...

Program

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

sleep instruction

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
```

DISADVANTAGES

SHARED DATA

- Whats the difference between two execution scenarios?
- Who controls which scenario plays out?
- Which values can `shared` be?

Task 1

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

Task 2

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

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
```

DISADVANTAGES

SHARED DATA

```
struct Position
{
  double x, y, z;
};
```

DISADVANTAGES

SHARED DATA

```
struct Position
{
  double x, y, z;
};
```

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

DISADVANTAGES

SHARED DATA

```
struct Position
{
  double x, y, z;
};
```

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

```
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;
}
```

DISADVANTAGES

SHARED DATA

```
struct Position
{
  double x, y, z;
};
```

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

```
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;
}
```

GLOBAL COMMON VARIABLE

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

TASK 1 (T1)

```
newPos(p, 11, 22, 33);
```

TASK 2 (T2)

```
printPos(p);
```

NON-INTERLEAVED

```
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;
```

INTERLEAVED

```
T1  pos.x = x;

T2  std::cout << "X: " << pos.x << std::endl; // X: 11
T2  std::cout << "Y: " << pos.y << std::endl; // Y: 20
T2  std::cout << "Z: " << pos.z << std::endl; // Z: 30

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

DISADVANTAGES

SHARED DATA

NON-INTERLEAVED

```
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;
```

- Result
  - Ok, everthing completes in *time*
  - Data is consistent

INTERLEAVED

```
T1  pos.x = x;

T2  std::cout << "X: " << pos.x << std::endl; // X: 11
T2  std::cout << "Y: " << pos.y << std::endl; // Y: 20
T2  std::cout << "Z: " << pos.z << std::endl; // Z: 30

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

- Result
  - Data skewed
    - Result should be have either
      - 10 & 20 & 30
      - 11 & 22 & 33
  - Unreliable

DISADVANTAGES

SHARED DATA

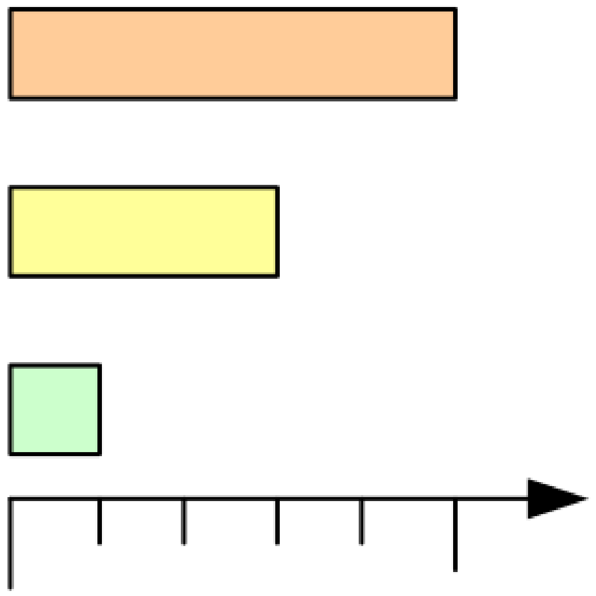
- The shared data problem is inherent in any preemptive multithreaded system
- Very cumbersome to find a good software solution to the problem
  - Peterson's solution: 2 interest flags, 1 will-wait flag
  - Does not scale
- We need a way to define critical sections of program
  - Sections in which the thread is guaranteed to be allowed to execute uninterrupted
- In a later lecture!

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

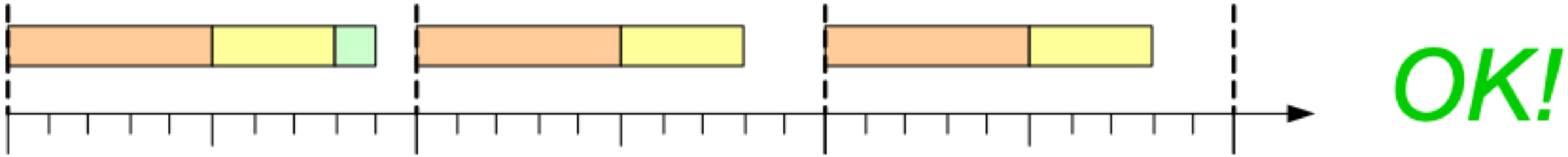
DISADVANTAGES

STARVATION

- 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



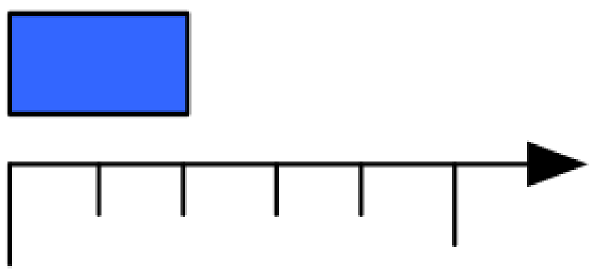
SCHEDULE?



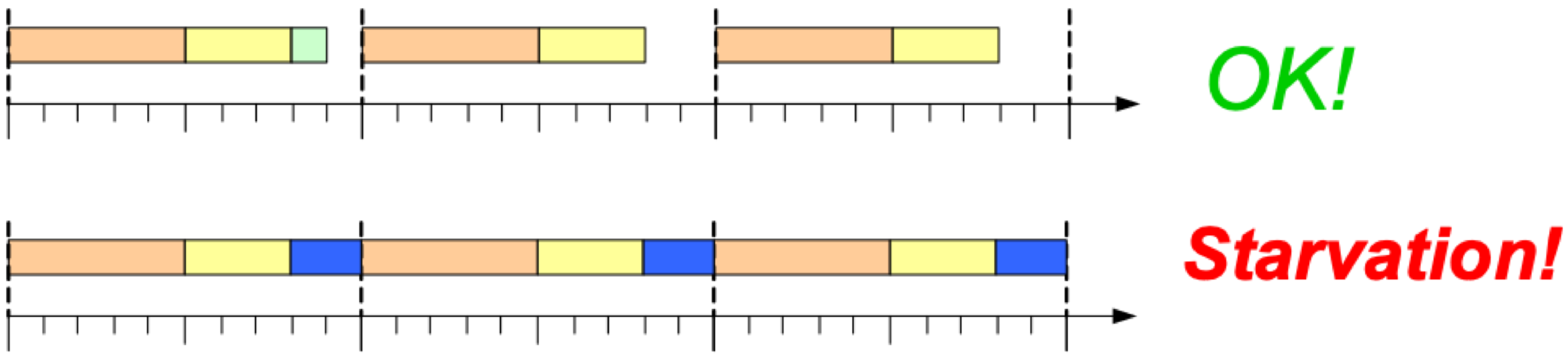
DISADVANTAGES

STARVATION

- Assume we add another MP thread, MP2
  - MP2 takes 2 µs, may run once every 10 µs



SCHEDULE?



DISADVANTAGES

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

PROGRAMMING WITH THREADS

- Summary
  - Threads
  - Processes
- Getting started

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

GETTING STARTED

- C++ does have the concept of threads built in
  - But we will not use it :-)
- 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`

GETTING STARTED

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