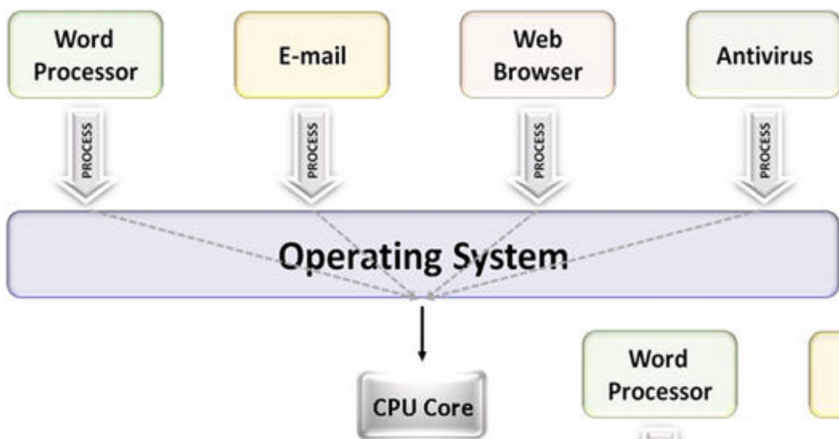
CS35L Software Construction Laboratory

Week 6; Lecture 1

Parallelism
&
Multithreading

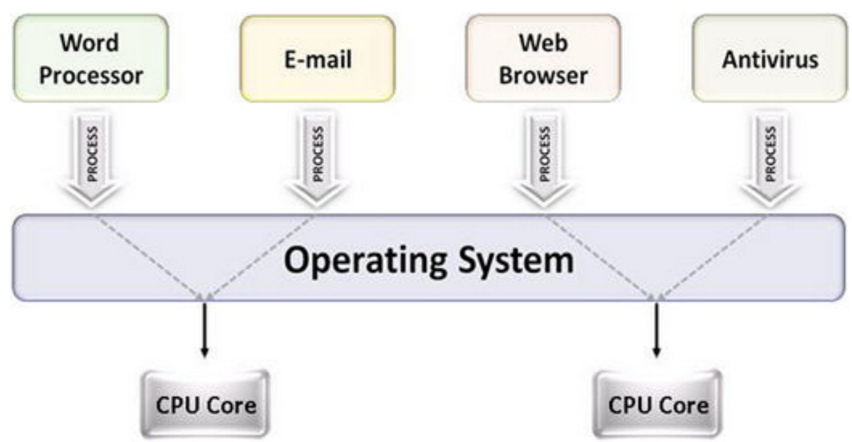
Multiprocessing

- The use of multiple CPUs/cores to run multiple tasks simultaneously

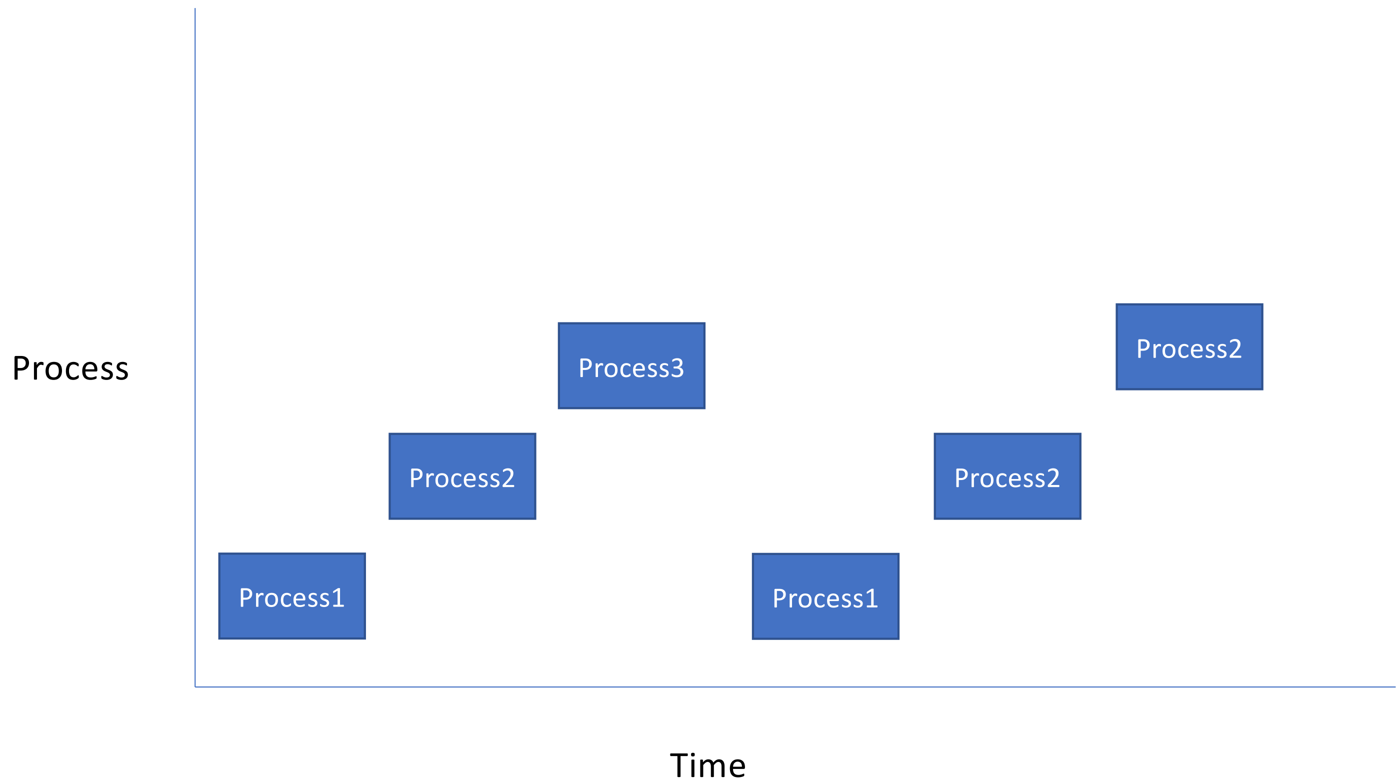


Uniprocessing system

Multiprocessing system

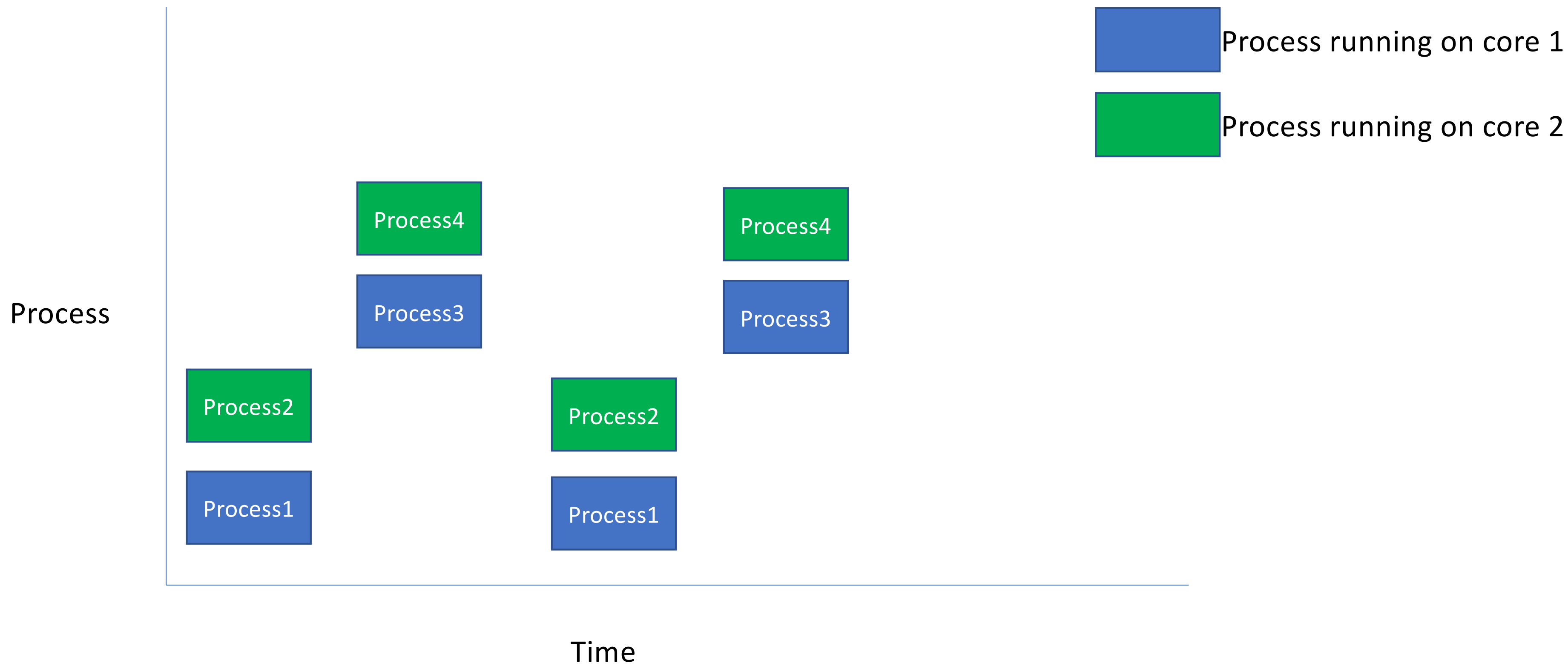


Multi Processing on a single core



Courtesy: Nandan Parikh

Multi Processing on 2 cores



Courtesy: Nandan Parikh

Parallelism

- Executing several computations simultaneously to gain performance
- Different forms of parallelism
  - **Multitasking**
    - Several processes are scheduled alternately or possibly simultaneously on a multiprocessing system
  - **Multithreading**
    - Same job is broken logically into pieces (threads) which may be executed simultaneously on a multiprocessing system

Usecase : Word Processor

Consider a word processor. It has the below features:

- Typing in from keyboard
- Displaying to stdout
- Spellchecker
- Auto-save to disk
- Connect to internet to view different templates

How to run all of this on a 2-core machine? Separate processes?

Enter Threads!

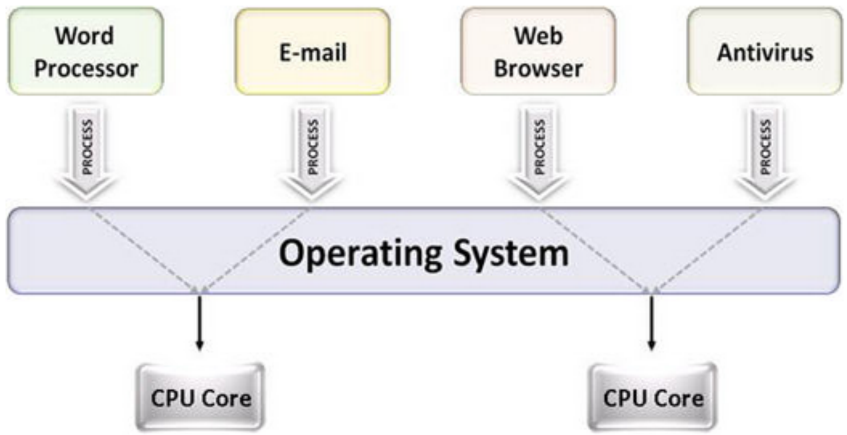
What is a thread?

- A flow of instructions, path of execution within a process
- The smallest unit of processing scheduled by OS
- A process consists of at least one thread
- Multiple threads can be run on:
  - **A uniprocessor (time-sharing)**
    - Processor switches between different threads
    - Parallelism is an illusion
  - **A multiprocessor**
    - Multiple processors or cores run the threads at the same time
    - True parallelism

Process vs Threads

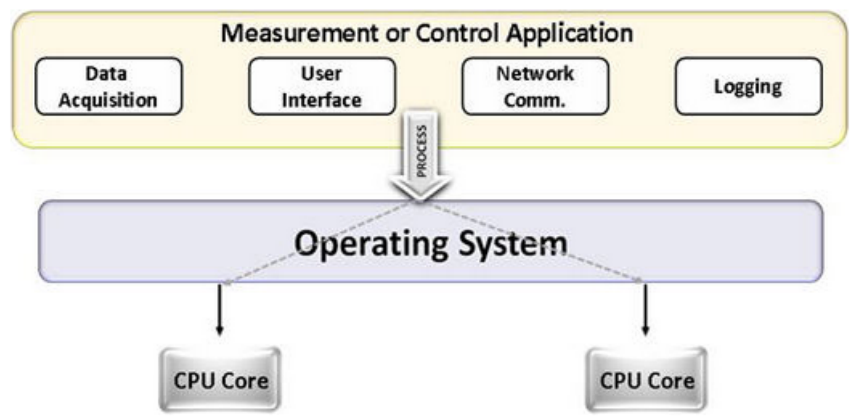
- Different processes see separate address spaces
  - good for protection, bad for sharing
- All threads in the same process share the same memory (except stack)
  - good for sharing, bad for protection
  - each thread can access the data of other thread

Multitasking vs. Multithreading

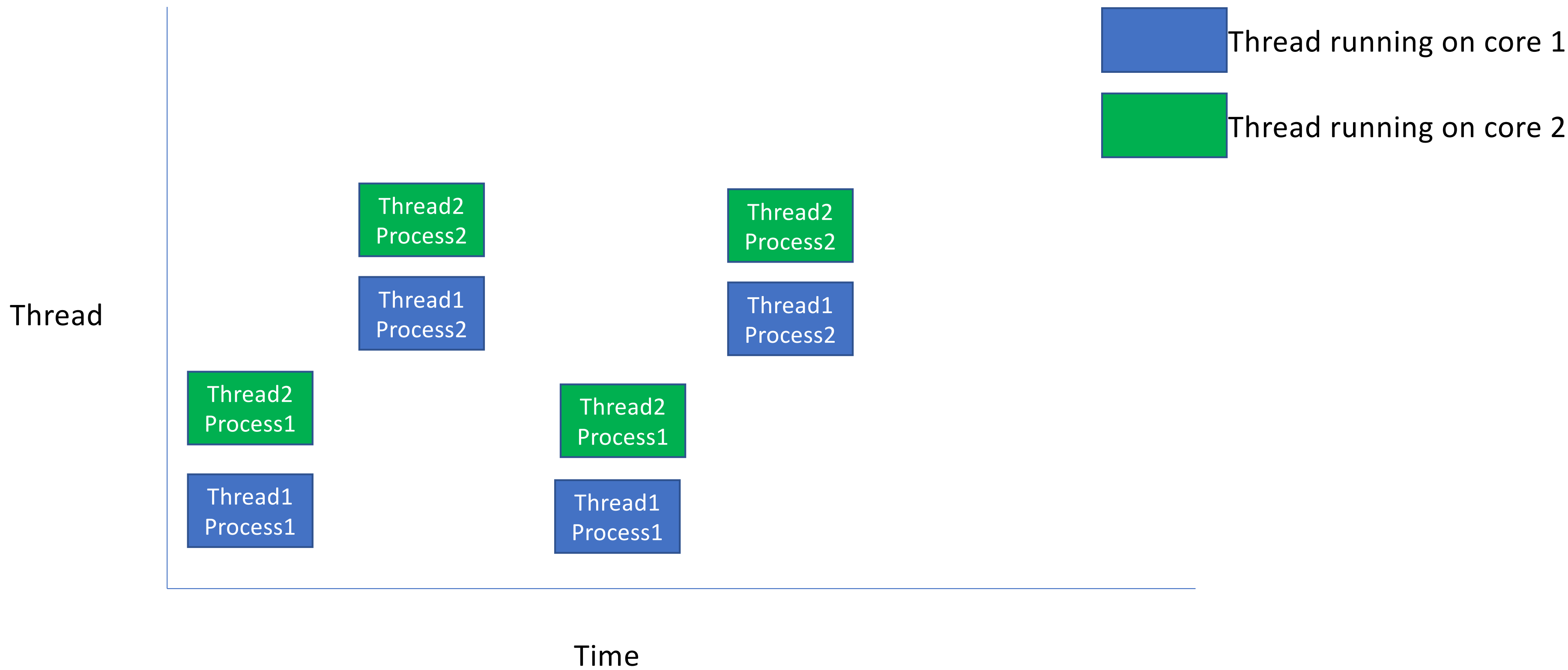


Multitasking

Multithreading



Multi Threading on 2 cores

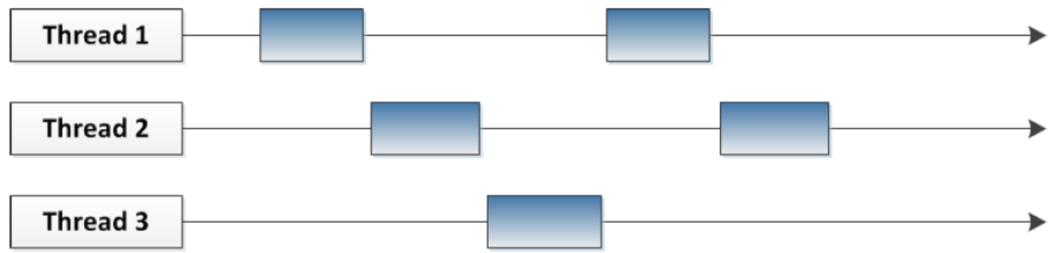


Courtesy: Nandan Parikh

Multithreading & Multitasking: Comparison

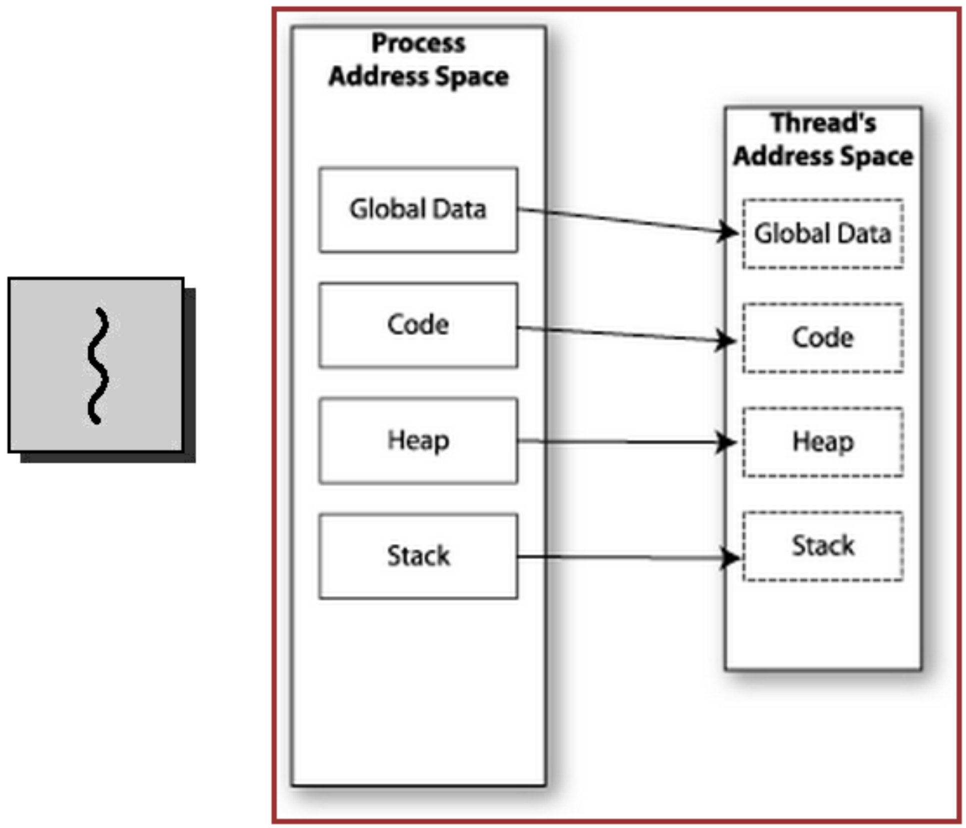
- **Multithreading**
  - Threads share the same address space
    - Light-weight creation/destruction
    - Easy inter-thread communication
    - An error in one thread can bring down all threads in process
- **Multitasking**
  - Processes are insulated from each other
    - Expensive creation/destruction
    - Expensive IPC
    - An error in one process cannot bring down another process

Multiple threads sharing a single CPU

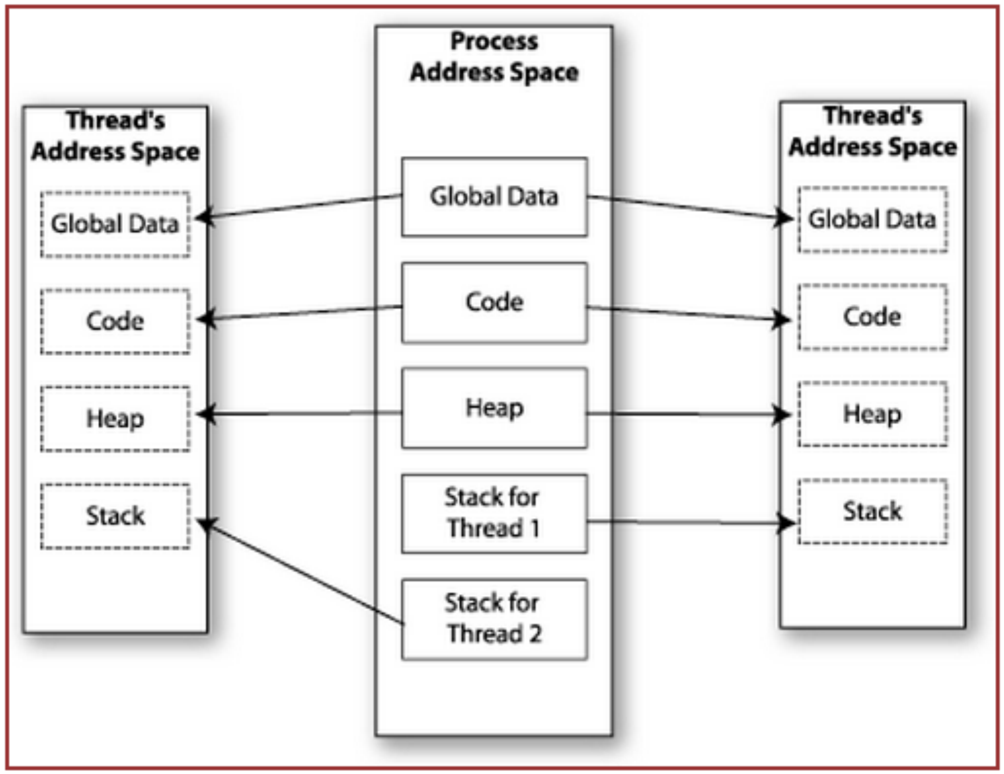


Multiple threads on multiple CPUs

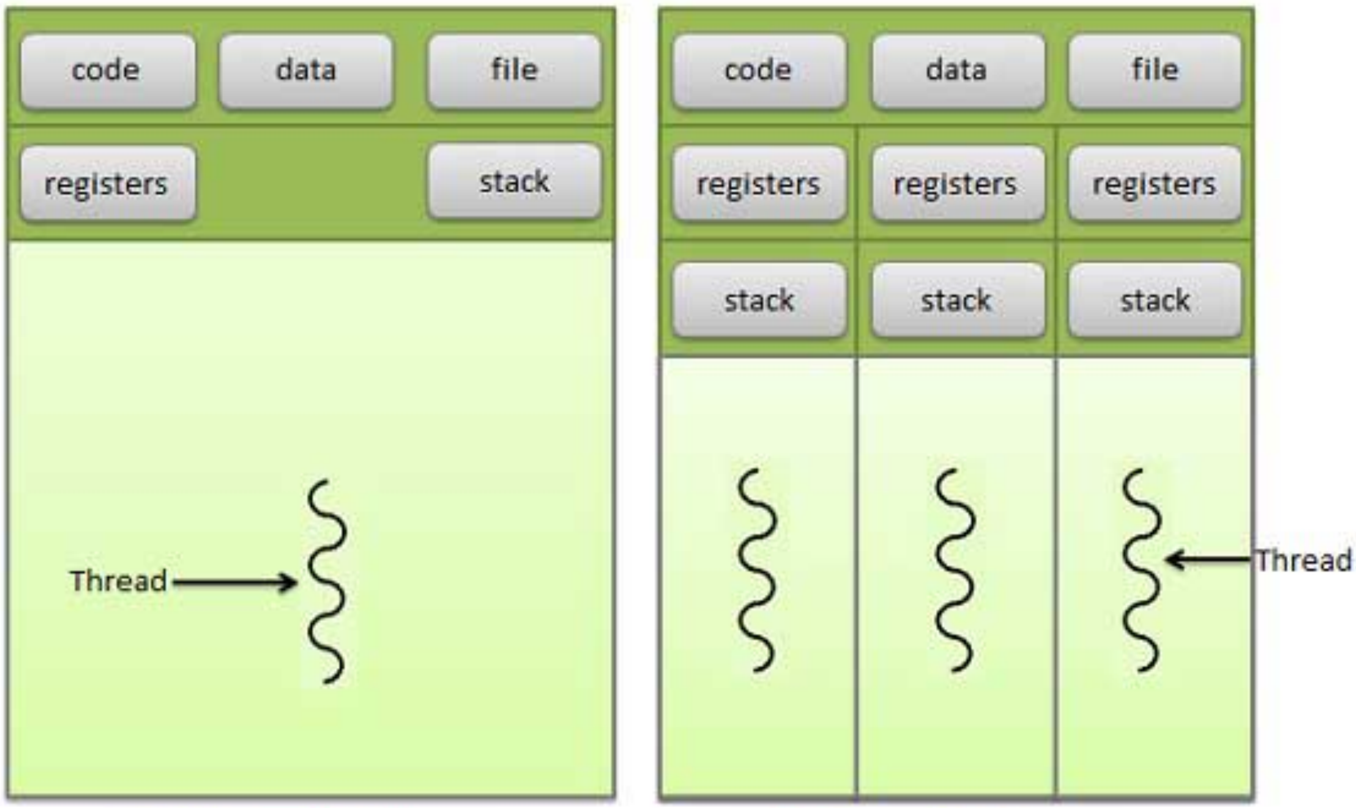
Memory Layout: Single-Threaded Program



Memory Layout: Multithreaded Program



Multithreading memory layout



Single threaded Process

Multi-threaded Process

Shared Memory

- Makes multithreaded programming
  - **Powerful**
    - can easily access data and share it among threads
  - **More efficient**
    - No need for system calls when sharing data
    - Thread creation and destruction less expensive than process creation and destruction
  - **Non-trivial**
    - Have to prevent several threads from accessing and changing the same shared data at the same time (synchronization)

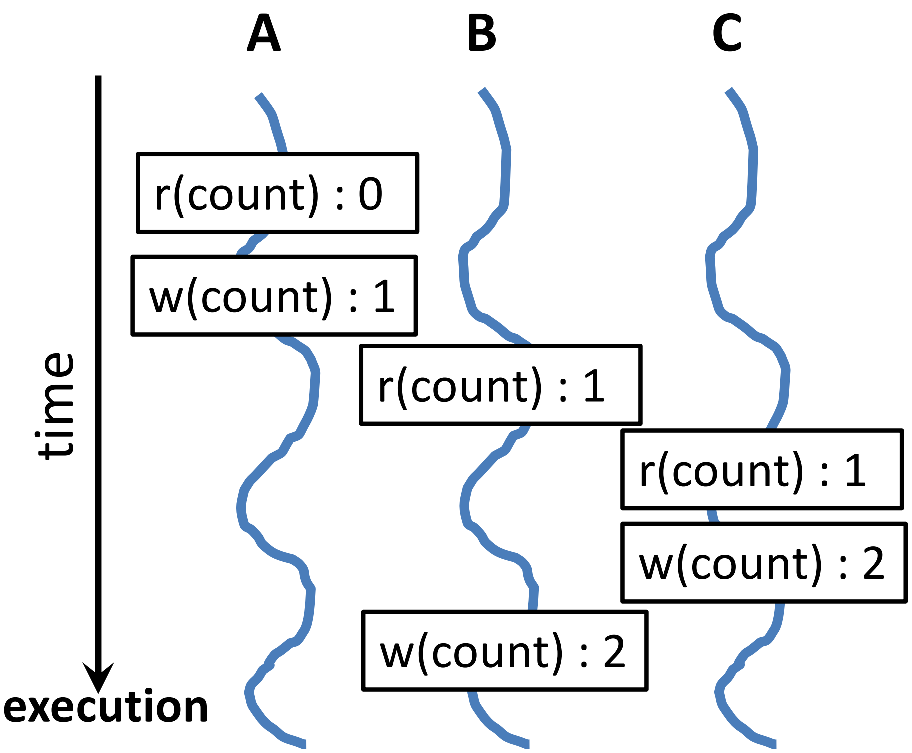
Process/thread synchronization

Why is it needed?

- Because threads share the same resources, we need synchronization
- To prevent inconsistency

Race Condition

```
int count = 0;
void increment()
{
  count = count + 1;
}
```



Result depends on order of execution

=> Synchronization needed

Example

a=10

```
P(){
    read(a);
    a = a+1;
    write(a);
}
```

#1: P1 should first execute P and then P2.

Ans = 12

#2: P1 -> reads a=10, context switch to P2

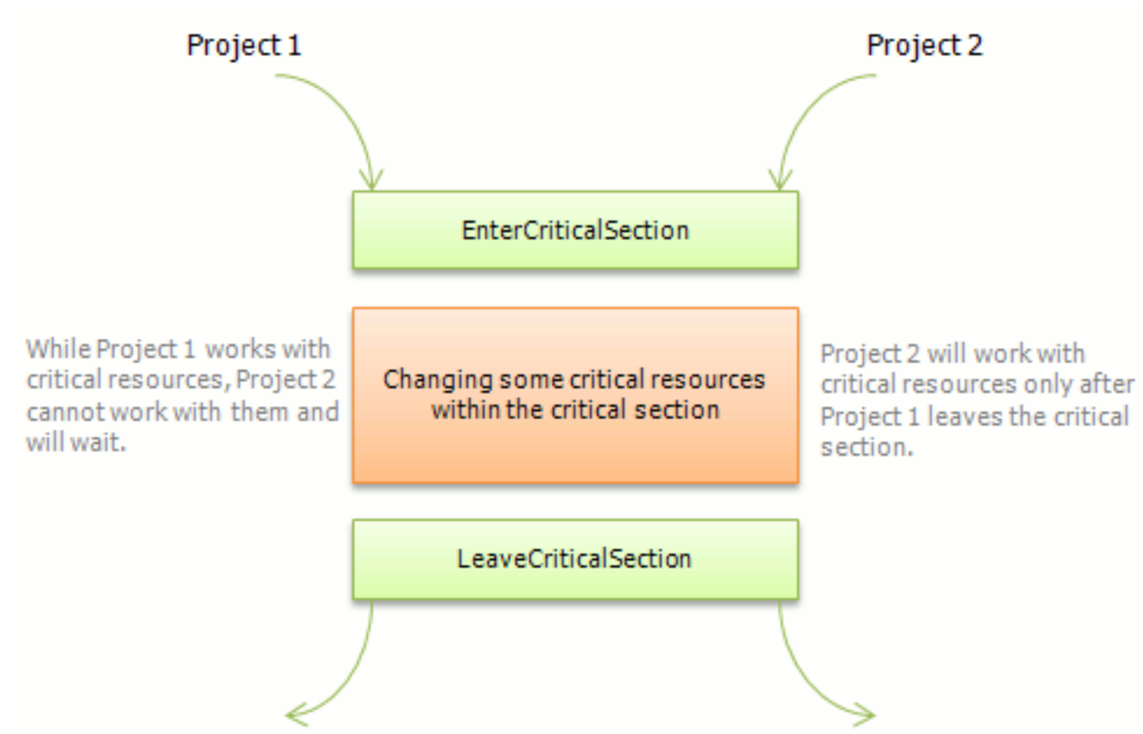
P2 -> reads a=10, adds 1 to a -> 11, switch to P1

P1 -> (has already read 10) a=11

Ans = 11

How to deal with it?

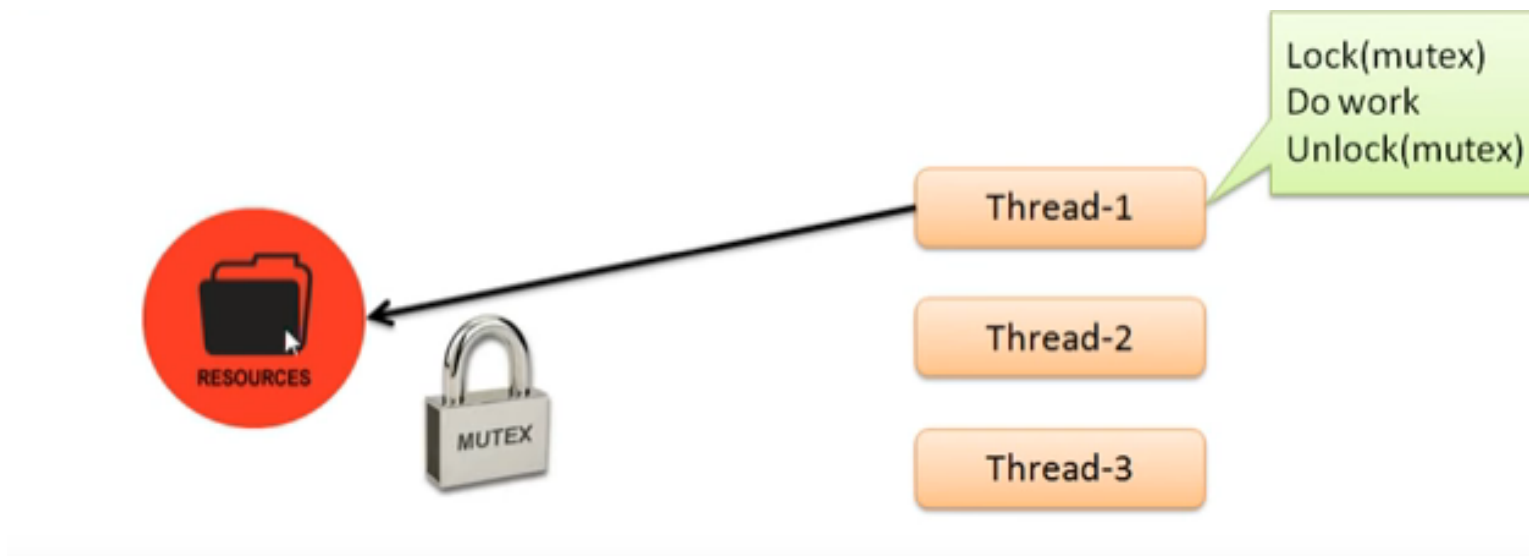
Critical section (prevents race condition)



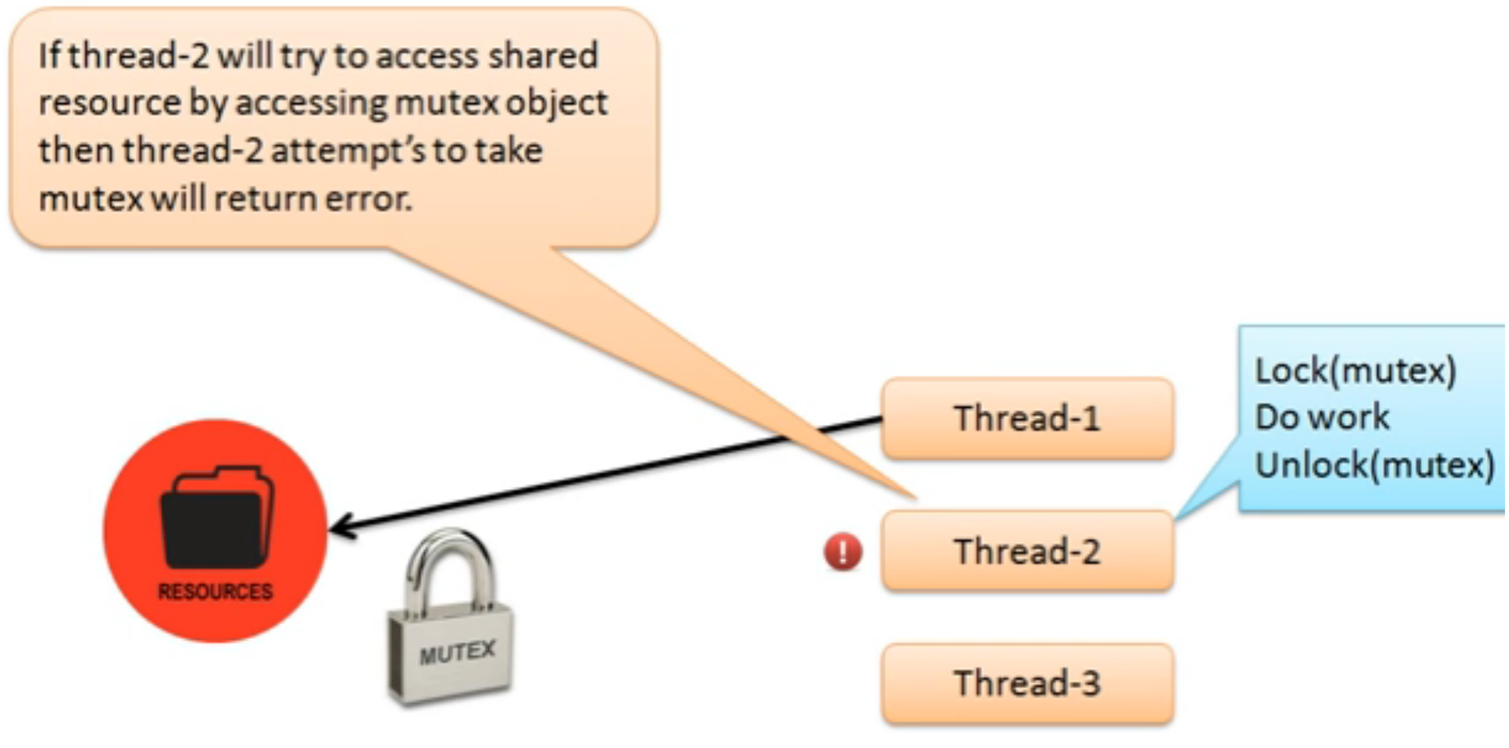
Mutex

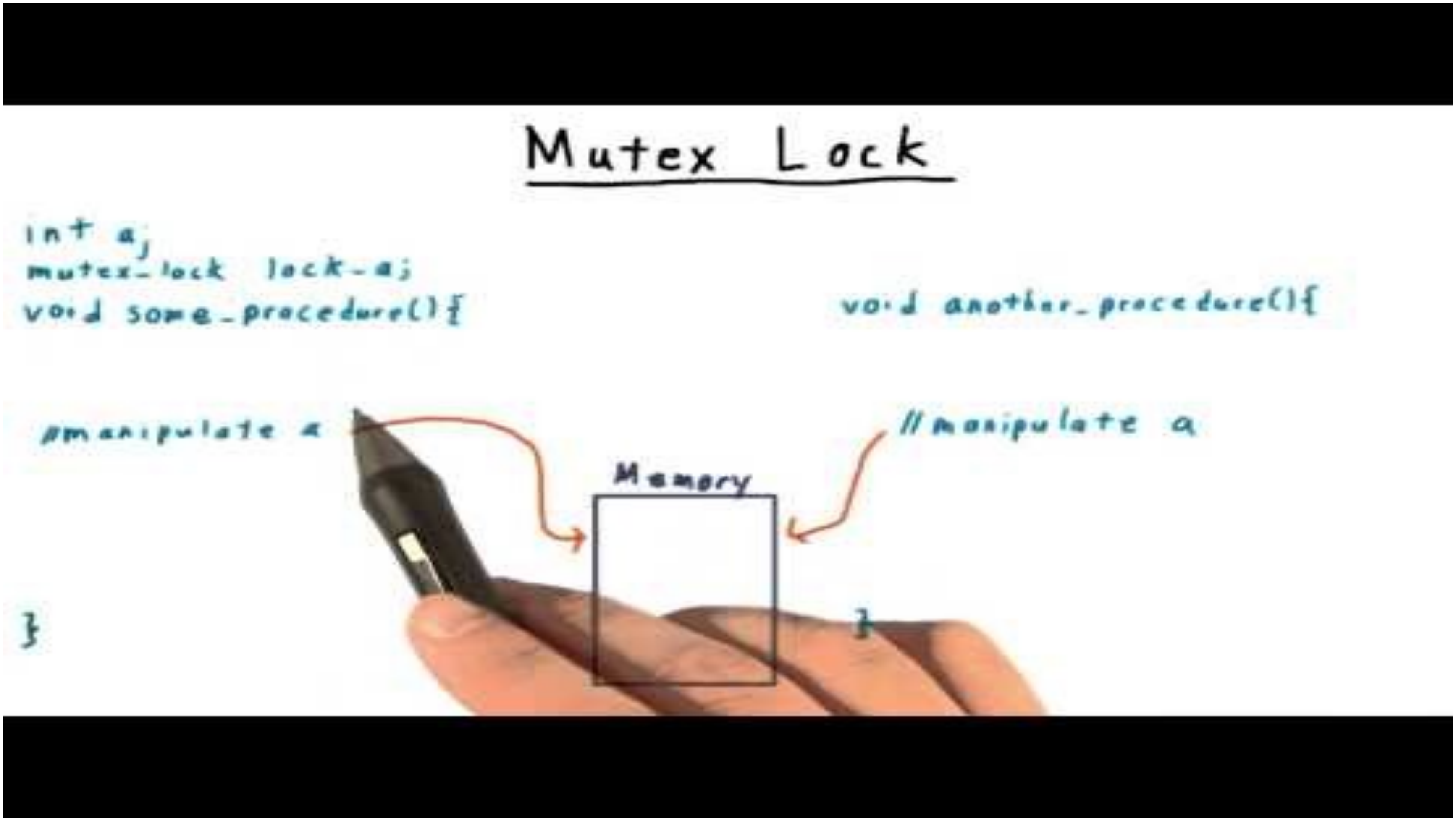
- Mutex is an object which allows only one thread into a critical section
- Mutex is owned by a thread
- It forces other threads which attempt to gain access to that section, to wait until the first thread has exited from the section
- Each resource has a mutex

RESOURCES
Thread-1
Thread-2
Thread-3
Lock(mutex)
Do work
Unlock(mutex)



Lock(mutex)
Do work
Unlock(mutex)
Thread-1
Thread-2
Thread-3
RESOURCES
MUTEX

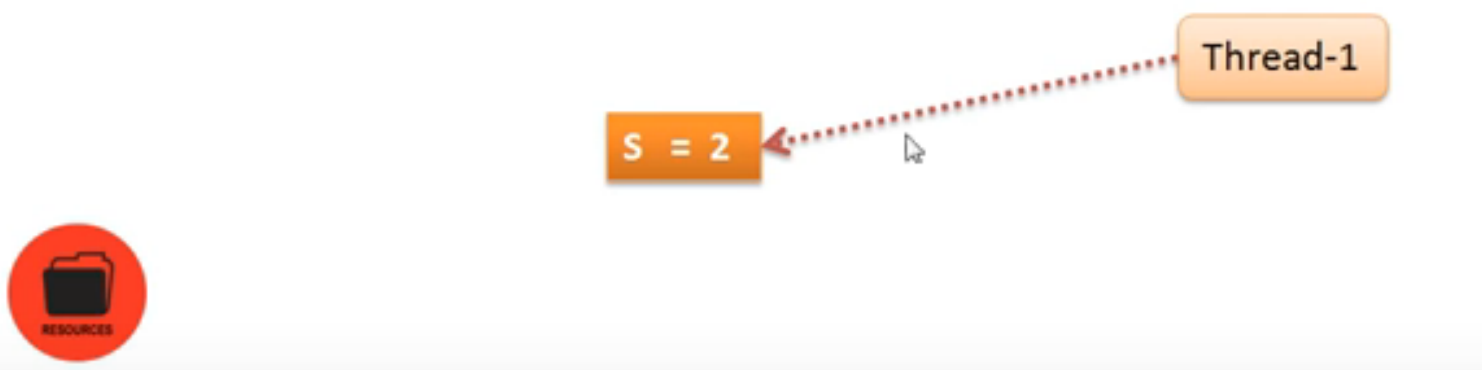

If thread-2 will try to access shared resource by accessing mutex object then thread-2 attempt's to take mutex will return error.
Lock(mutex)
Do work
Unlock(mutex)
Thread-1
Thread-2
Thread-3
RESOURCES
MUTEX



Mutex Lock
int a;
mutex_lock lock_a;
void some_procedure(){
//manipulate a
}
Memory
void another_procedure(){
//manipulate a
}

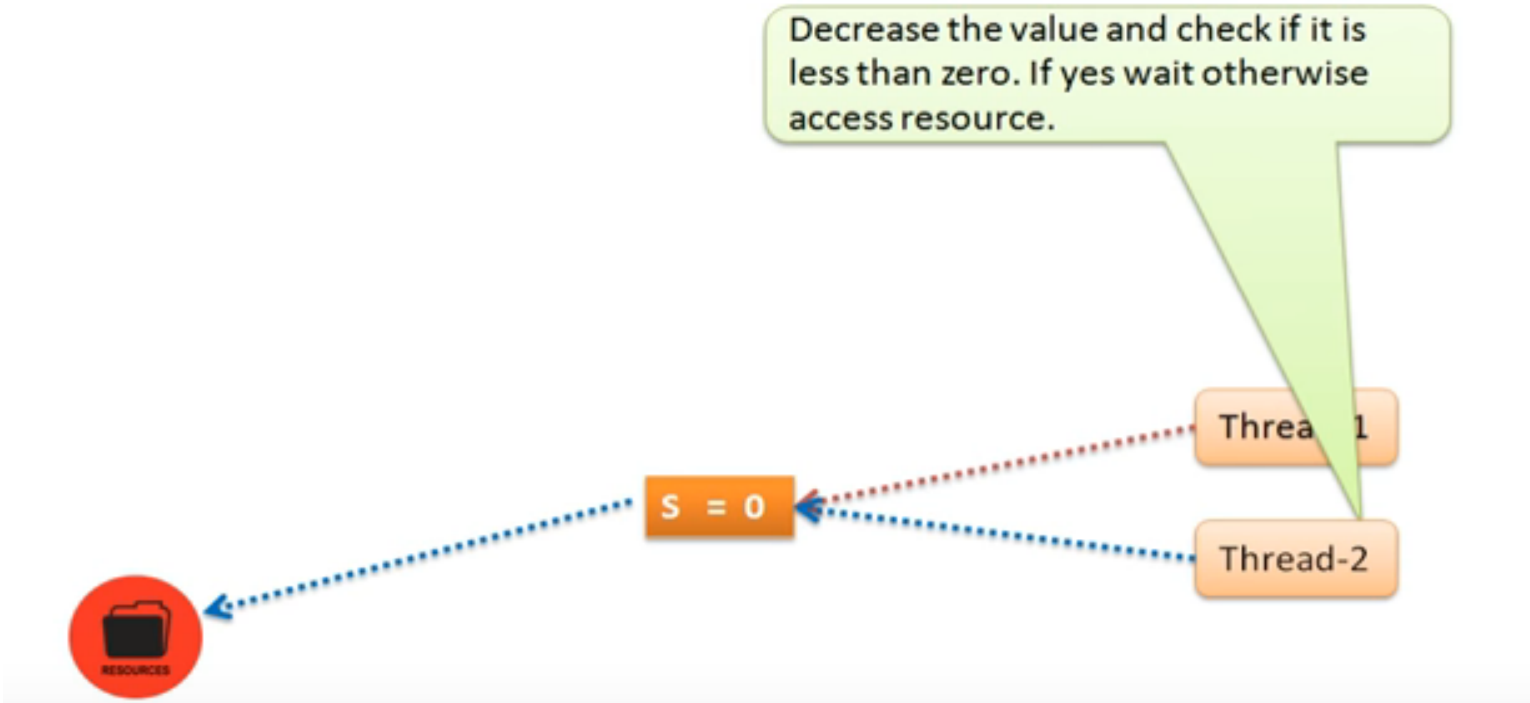
Semaphores

- A semaphore is a value in a designated place in the OS (or kernel) storage that each process can check and then change.
- signaling mechanism
- restricts/allows the number of simultaneous threads of a shared resource upto a maximum number
- threads can request access to a resource (decrements the semaphore)
- threads signal that the have finished using the resource (increments the semaphore)



S = 2
Thread-1
RESOURCES

RESOURCES
S = 1
Thread-1
Thread-2


Decrease the value and check if it is less than zero. If yes wait otherwise access resource.
Threa 1
S = 0
Thread-2
RESOURCES


signal()

Semaphores v/s mutex

- Semaphore allows multiple program threads to access the finite instance of resources.

- On the other hand, Mutex allows multiple program threads to access a single shared resource but one at a time.

Multitasking

- $ tr -cs 'A-Za-z' '[\n*]' | sort -u | comm -23 – words
  - Process 1 (tr)
  - Process 2 (sort)
  - Process 3 (comm)
- Each process has its own address space
- How do these processes communicate?
  - Pipes/System Calls

Multithreading

- Threads share all of the process's memory except for their stacks
- => Data sharing requires no extra work (no system calls, pipes, etc.)

POSIX Threads

- import the pthread library
  Example: #include<pthread.h>

- Use -pthread while compiling

- Represented by pthread_t (datatype)

Basic pthread Functions

There are 5 basic pthread functions:

1. pthread_create: creates a new thread within a process

2. pthread_join: waits for another thread to terminate

3. pthread_equal: compares thread ids to see if they refer to the same thread

4. pthread_self: returns the id of the calling thread

5. pthread_exit: terminates the currently running thread