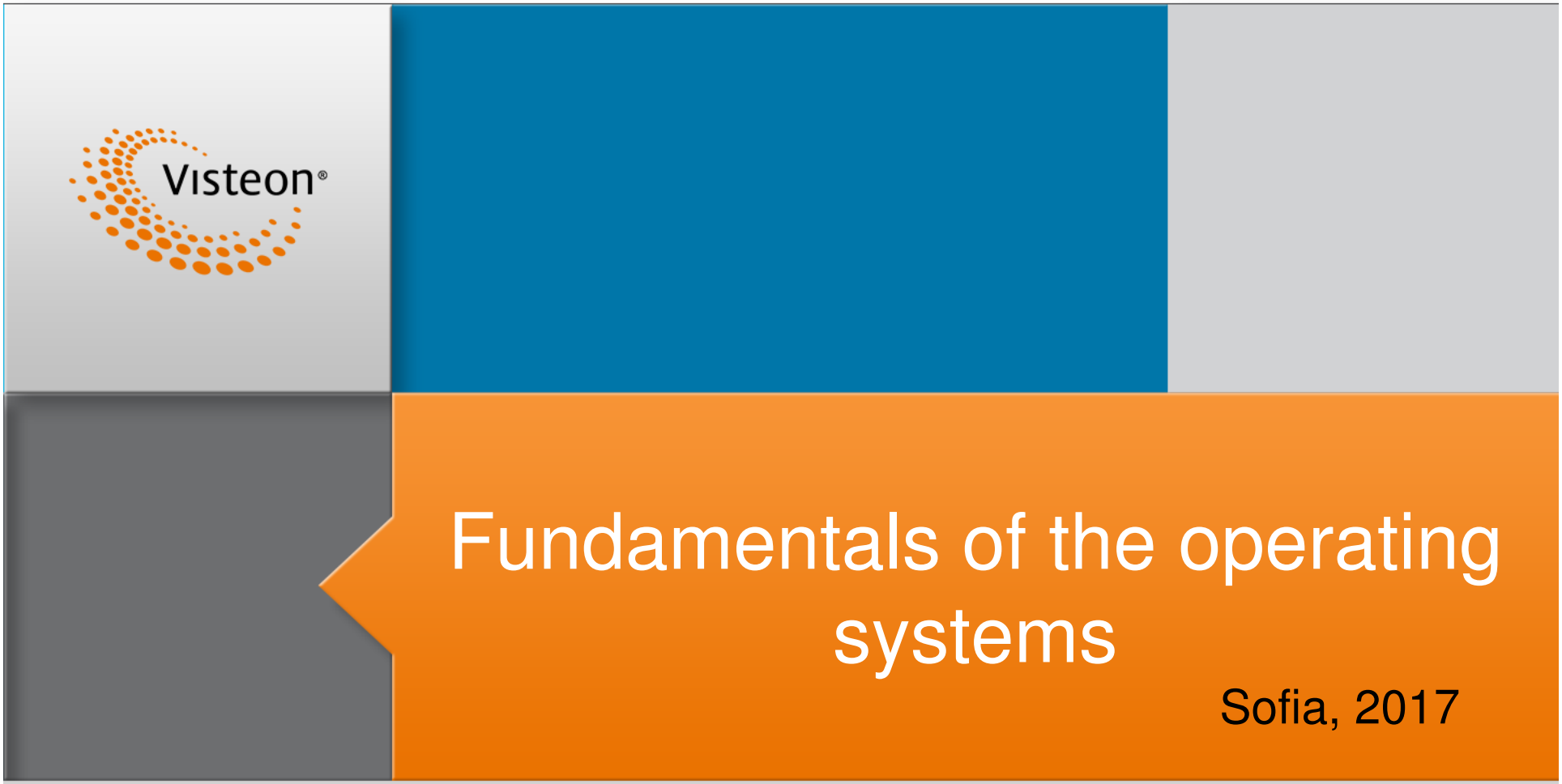



# Fundamentals of the operating systems

Sofia, 2017

# Contents

1. What is an operating system?
2. Kernel
3. System call
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations

# Contents

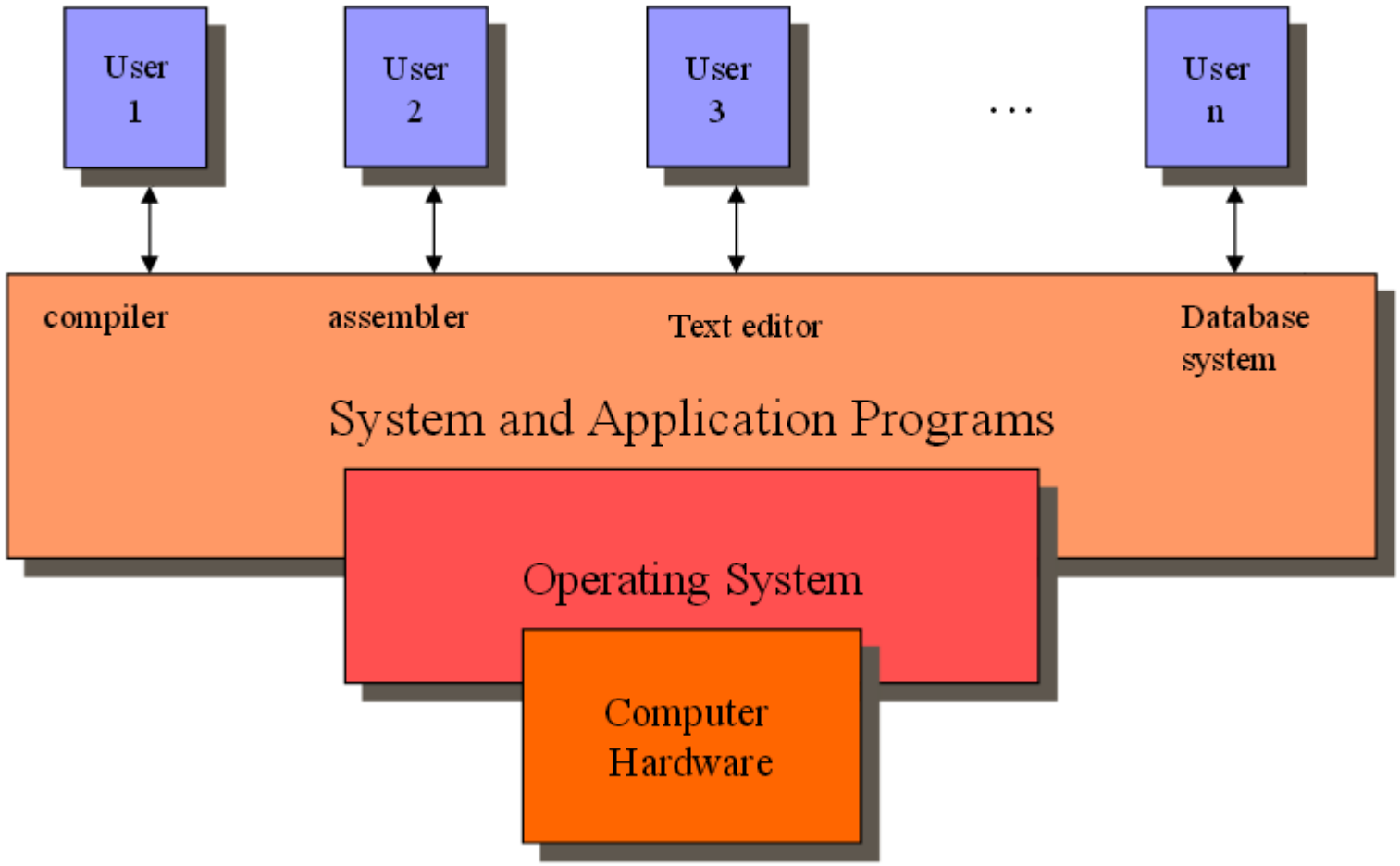
1. **What is an operating system?**
2. Kernel
3. System call
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations

# What is an operating system?

A system software/collection of procedures that:

- manage all the system's hardware resources
- control user actions to prevent errors and improper system usage
- provide the users the environment in which they can:
  - use the system resources
  - run their own applications

# What is an operating system?



# Operating system: Definitions

- Resource allocator
  - to allocate resources (software and hardware) of the computer system and manage them efficiently.
- Control program
  - controls the execution of user programs and operations of I/O devices.
- Kernel
  - the one program running at all time. Everything else is either a system program (ships with the operating system) or an application program.

# Contents

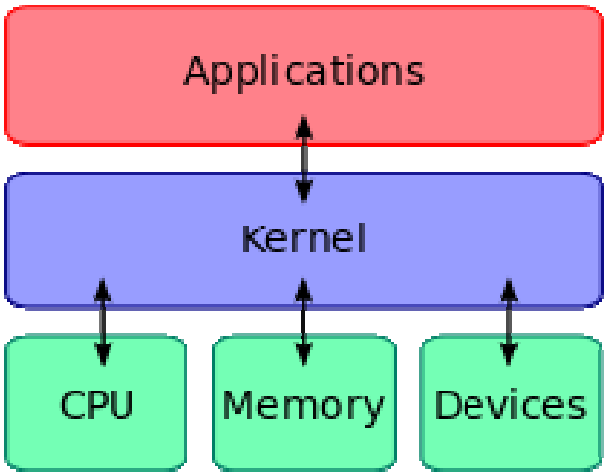
1. What is an operating system?
2. **Kernel**
3. System call
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations

# Kernel

## What is a kernel?

- The kernel is a fundamental part of any operating system.

- The kernel manages I/O requests from software, and translates them into data processing instructions for the central processing unit and other electronic components of a computer.

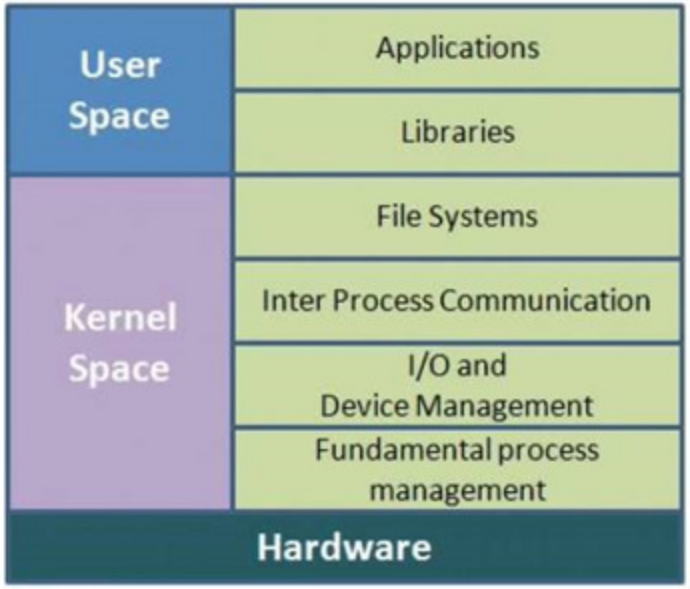


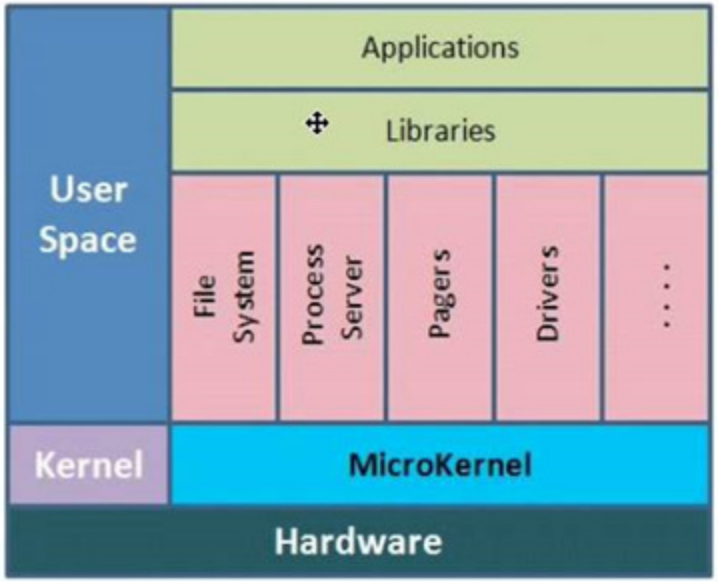
# Kernel types

- Micro-kernels
  - the kernel modules run in user mode → protection against bugs
  - adaptability to use in distributed systems
  - forces the programmers to adopt a modularize approach
  - easily ported to other architectures
  - better use of RAM than monolithic kernels
- Monolithic kernels
  - monolithic OS faster than micro-kernel OS
  - modularized approach
  - platform independence
  - frugal main memory usage
  - no performance penalty

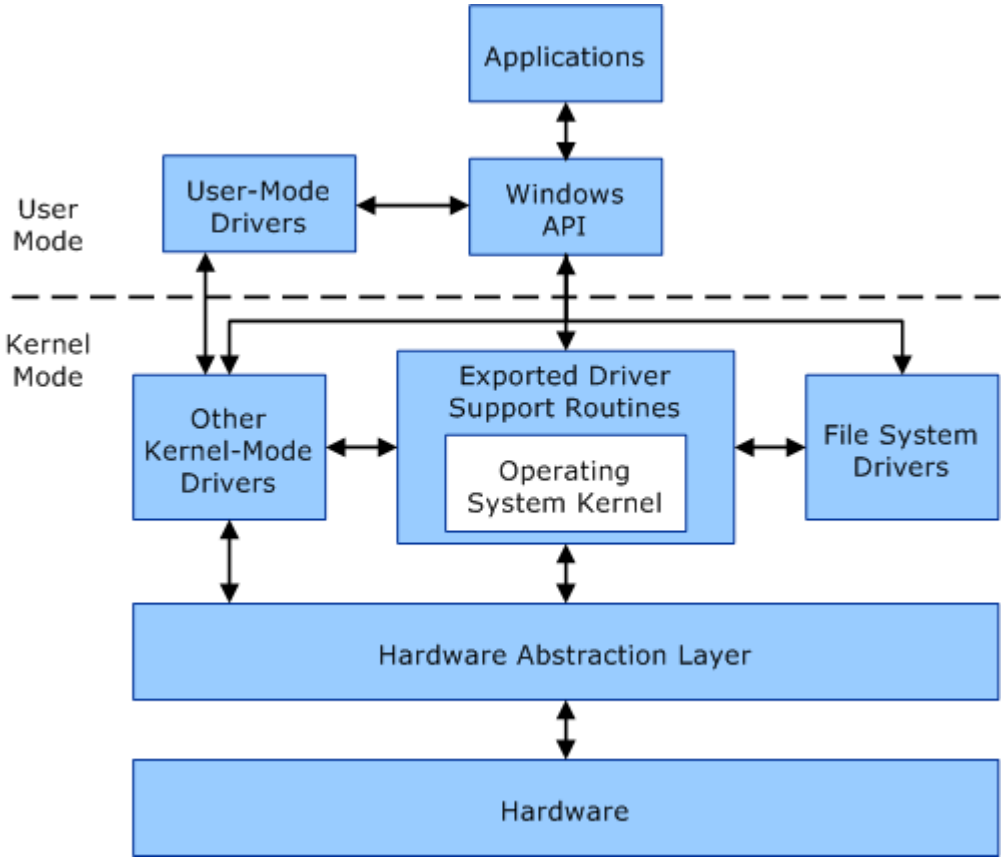
# Kernel types

## Monolithic Kernel vs Microkernel



# Kernel mode vs User mode



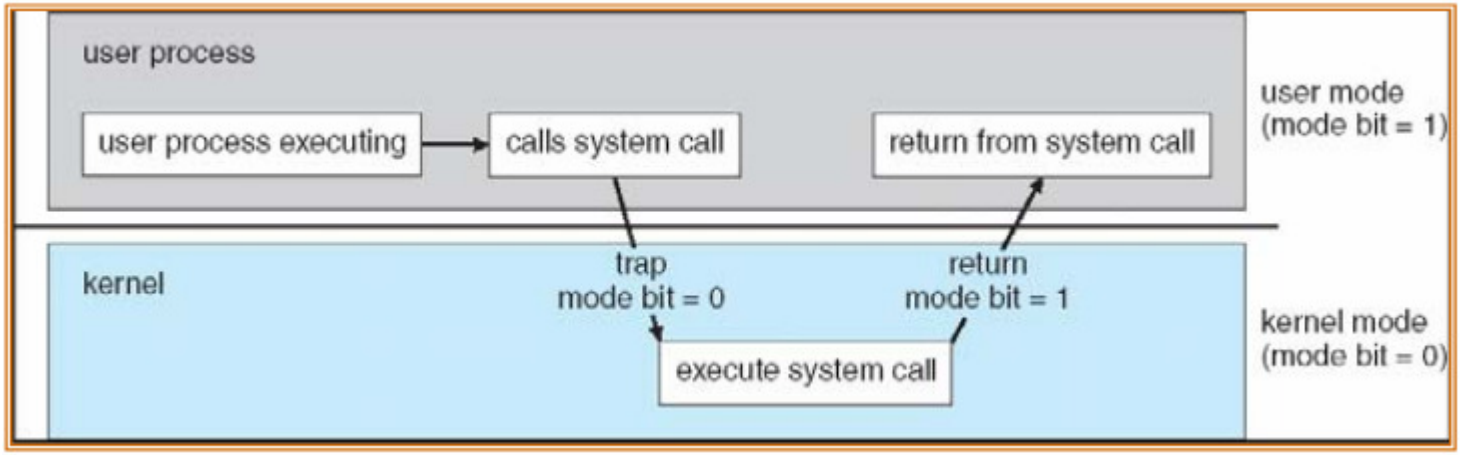

- Main reason for separation of kernel and user space is **security!**

# Contents

1. What is an operating system?
2. Kernel
3. **System call**
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations

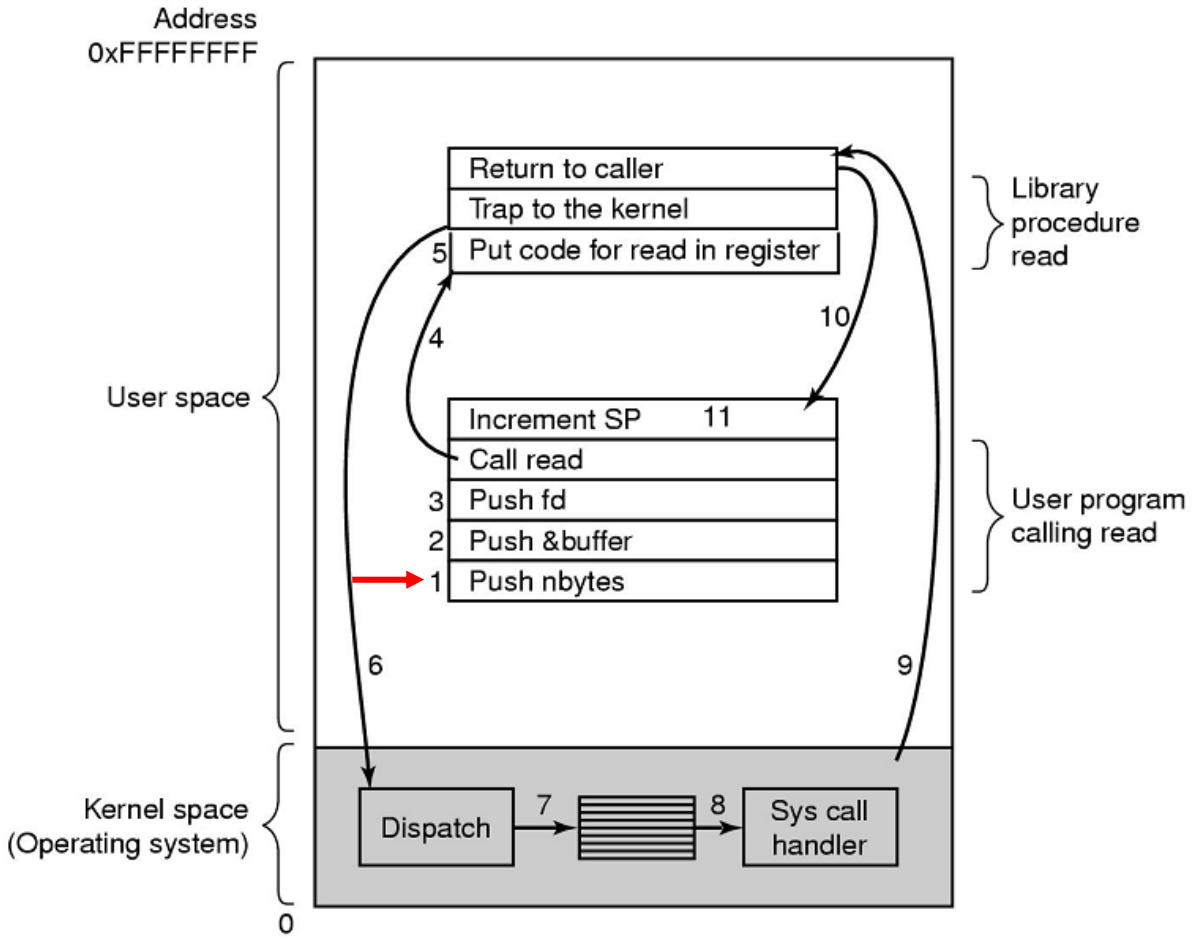
# System calls

- Definition
  - a call to an OS service
  - a trap into the OS code
- Examples of system calls
  - File manipulation: open(), read(), write(), lseek(), close() …
  - File system management: mkdir(), mount(), link(), chown() …
  - Process management: fork(), exec(), wait(), exit() …



# System functions – libc.so

- Steps in making a system call




**There are 11 steps executing the system call:**
**read (fd, buffer, nbytes)**

# Contents

1. What is an operating system?
2. Kernel
3. System call
4. **Concurrency – issues and solutions**
5. Multi process programming
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations

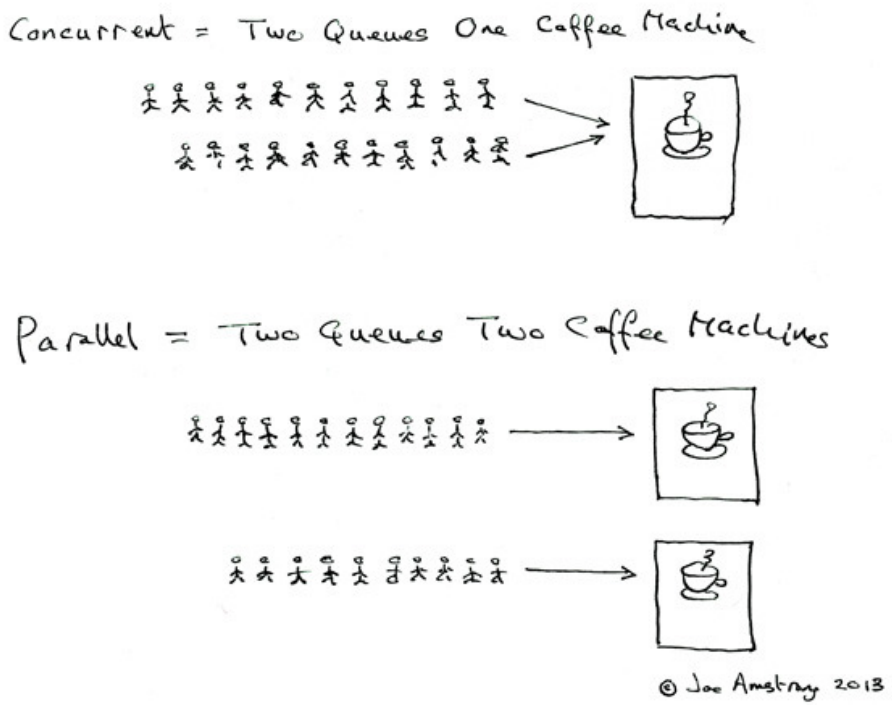
# Basic concepts of OS: Concurrency

The operating system provides the ability to have more than one independent executions at a given time.

Usually there are no limitations on the number of independent executions at a given time, however, if the separate parallel executions, have to access a single resource, a concurrency arises.

# Basic concepts of OS: Concurrency

Concurrency is the interleaving of processes in time to give the appearance of simultaneous execution. Thus it differs from parallelism, which offers genuine simultaneous execution.

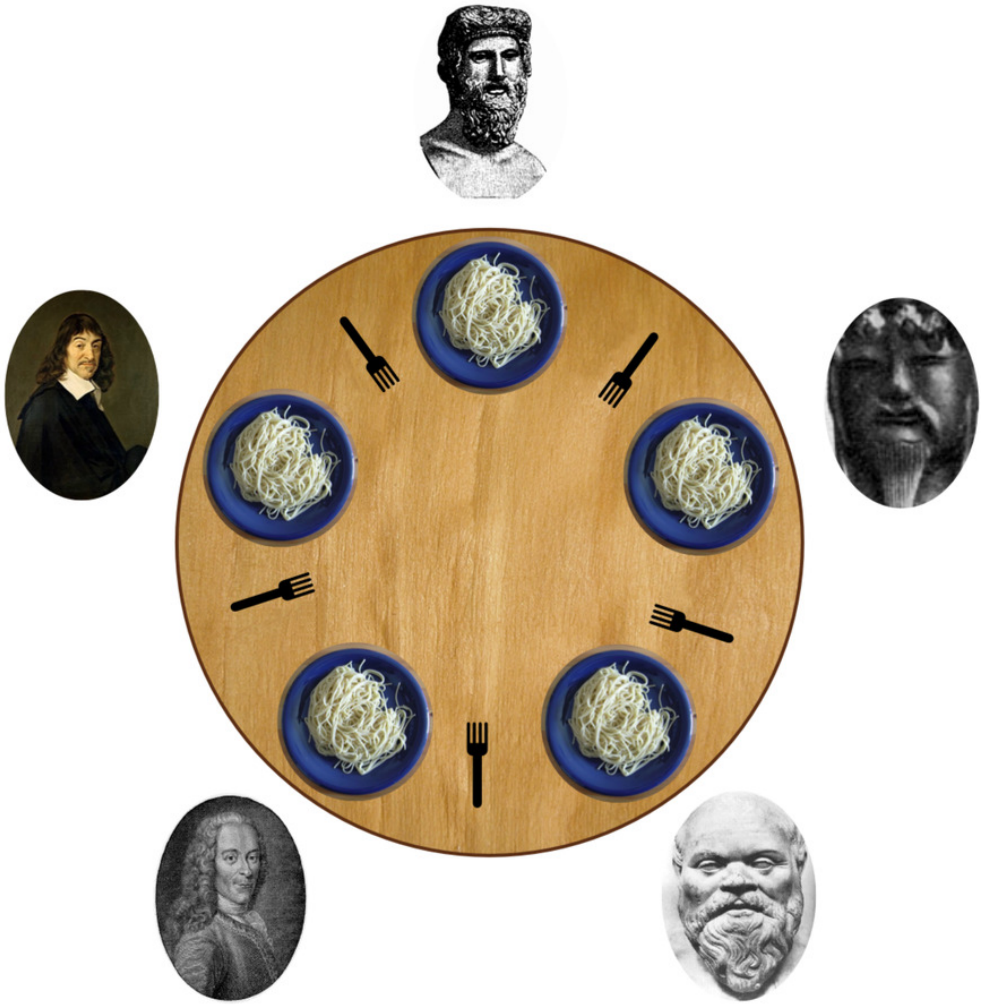


# Basic concepts of OS: Concurrency

**Concurrency vs. parallelism**

- Concurrency is about dealing with lots of things at once.
- Parallelism is about doing lots of things at once.
- Not the same, but related.
- Concurrency is about structure, parallelism is about execution.
- Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.

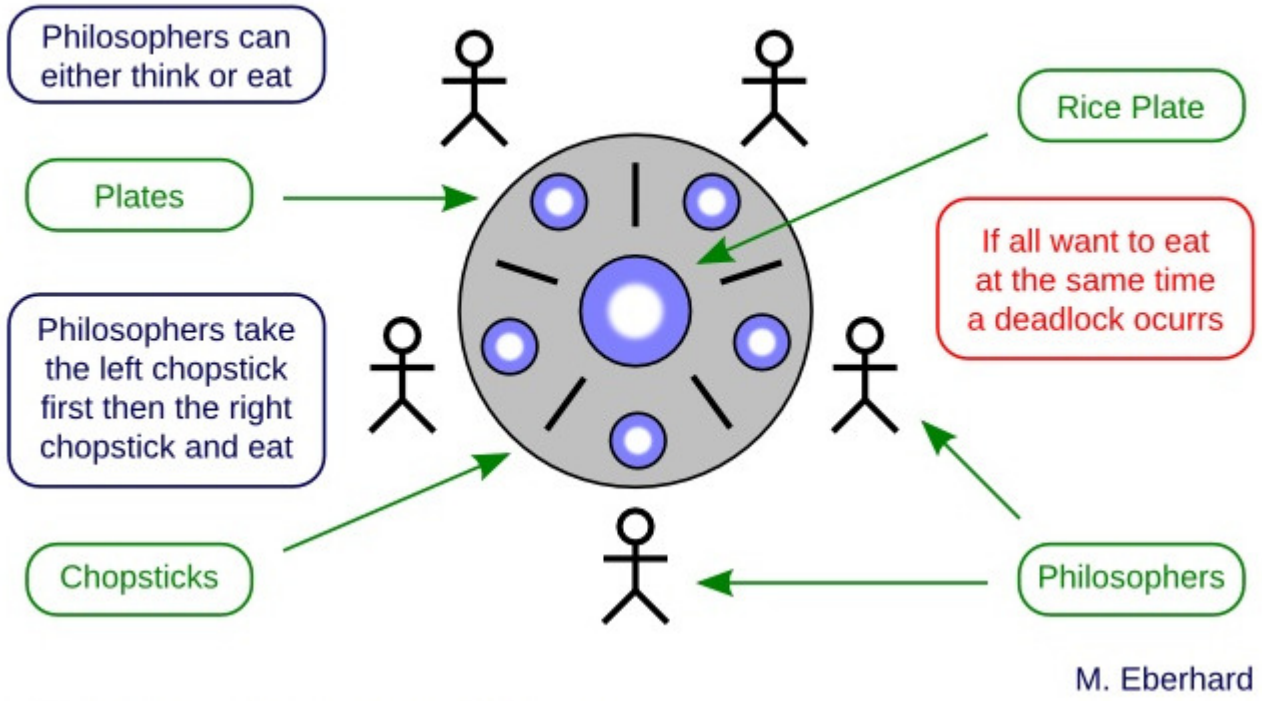
# Basic concepts of OS: Concurrency



"Dining philosophers" by Benjamin D. Esham / Wikimedia Commons. Licensed under CC BY-SA 3.0

# Basic concepts of OS: Concurrency

- **The Dining Philosophers Problem (Deadlock)**



# Basic concepts of OS: Concurrency

## Concurrency cannot be avoided because:

- Users are concurrent - a person can handle several tasks at once (have you every listened to music while doing other work and heard the phone ring?) and expects the same from a computer.
- Multiprocessors are becoming more prevalent. The Internet is perhaps a huge multiprocessor.
- A distributed system (client/server system) is naturally concurrent.
- A windowing system is naturally concurrent.
- I/O is often slow because it involves slow devices such as disks, printers; many network operations are essentially (slow) I/O operations. When doing I/O it is helpful to handle the I/O concurrently with other work.

# Basic concepts of OS: Concurrency

## Concurrency issues:

- Separable operation (non-Atomic)

An operation is atomic if the steps are done as a unit. Operations that are not atomic, but interruptible and done by multiple processes can cause problems.

- Race condition

- A race condition occurs if the outcome depends on which of several processes gets to a point first.

- Situations like this where processes access the same data concurrently and the outcome of execution depends on the particular order in which the access takes place

For example, fork( ) can generate a race condition if the result depends on whether the parent or the child process runs first. Other race conditions can occur if two processes are updating a global variable.

- Blocking and starvation

While neither of these problems is unique to concurrent processes, their effects must be carefully considered. Processes can block waiting for resources. A process could be blocked for a long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable. Starvation occurs when a process does not obtain sufficient CPU time to make meaniful progress.

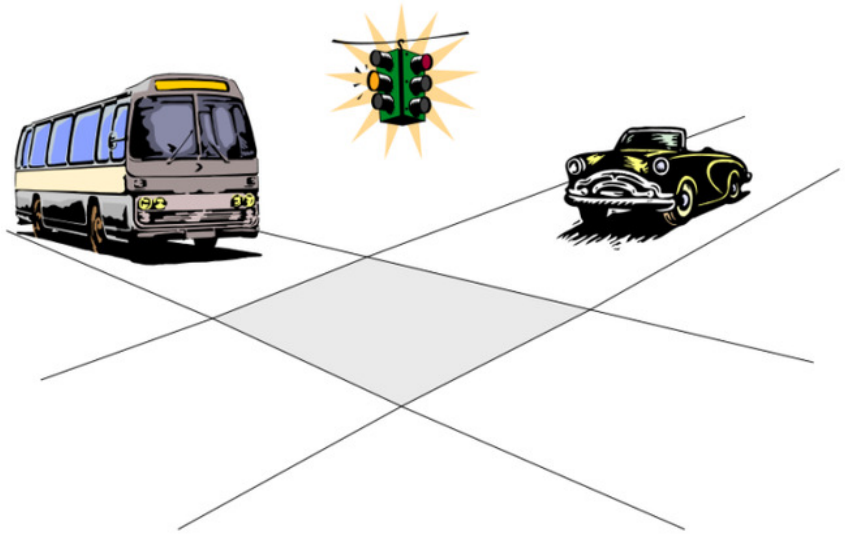
- Deadlock

Deadlock occurs when two processes are blocked in such a way that neither can proceed. The typical occurrence is where two processes need two non-shareable resources to proceed but one process has acquired one resource and the other has acquired the other resource. Acquiring resources in a specific order can resolve some deadlocks.

# Basic concepts of OS: Concurrency

## Critical section & Mutual exclusion

- When a process executes code that manipulates shared data (or resource), we say that the process is in it's critical section (for that shared data)
- The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple CPUs)



# Basic concepts of OS: Concurrency

## Critical section & Mutual exclusion

- Entry of critical section
  - Each process must request the permission to enter it's critical section (CS)
  - Once entered, each process must lock the critical section, so other cannot be permitted to enter

- Exit of critical section
  - Each process must unlock the critical section so other could be permitted to enter

```
repeat
   entry section
    critical section
   exit section
    remainder section
forever
```

# Basic concepts of OS: Concurrency

## Semaphore

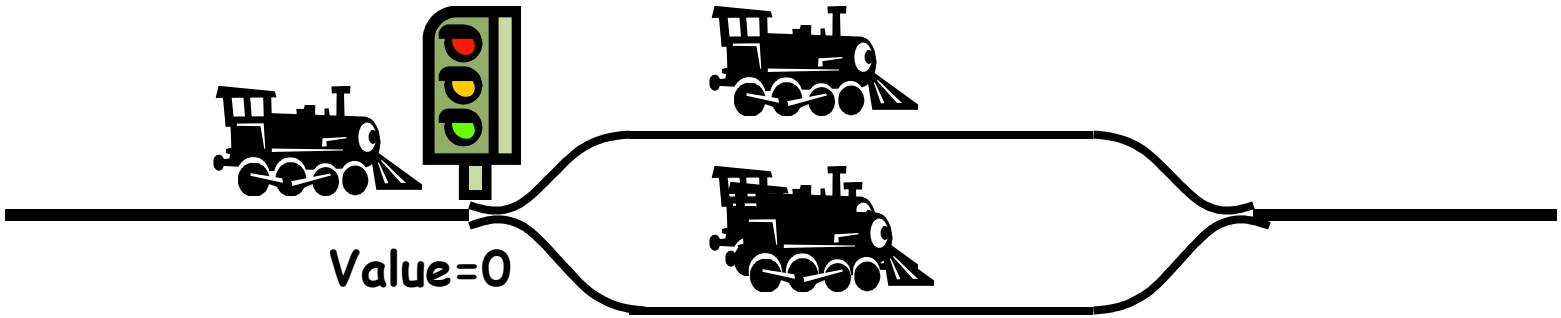
- Semaphores are a kind of generalized lock
  - First defined by the Dutch Edsger Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX

- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - This of this as the signal() operation

- Semaphores were initially introduced to solve the producer – consumer paradigm

# Basic concepts of OS: Concurrency

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - Two P's together can't decrement value below zero
    - Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



# Two uses of semaphores

- Mutual Exclusion (initial value = 1)
  - Also called “Binary Semaphore”.
  - Can be used for mutual exclusion:

```
semaphore.P(); // Lock the access
// Critical section goes here
semaphore.V(); // Unlock the access
```

- Scheduling Constraints (initial value = 0)
  - Locks are fine for mutual exclusion, but what if you want a thread to wait for something?
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminiate:

```
Initial value of semaphore = 0
ThreadJoin {
   semaphore.P();
}
ThreadFinish {
   semaphore.V();
}
```

# Basic concepts of OS: Concurrency

Instead of using semaphores for mutual exclusion we can use *mutexes.*

- Usage of a mutex:

  1. When a program is started, a mutex is created with a unique name

  2. After this stage, any thread(process) that needs the resource must lock the mutex from other threads while it is using the resource

  3. The mutex is set to unlock when the data is no longer needed or the routine is finished

# Basic concepts of OS: Concurrency

Instead of using semaphores for scheduling constraints we can use *condition variables ( a.k.a*. CondVars).

- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep.
  - In other words, the lock is released, while one process is waiting on the CondVar, so another process can acquire the lock and signal the first process to continue its operation
  - Contrast to semaphores: Can't wait inside critical section

# Basic concepts of OS: Concurrency

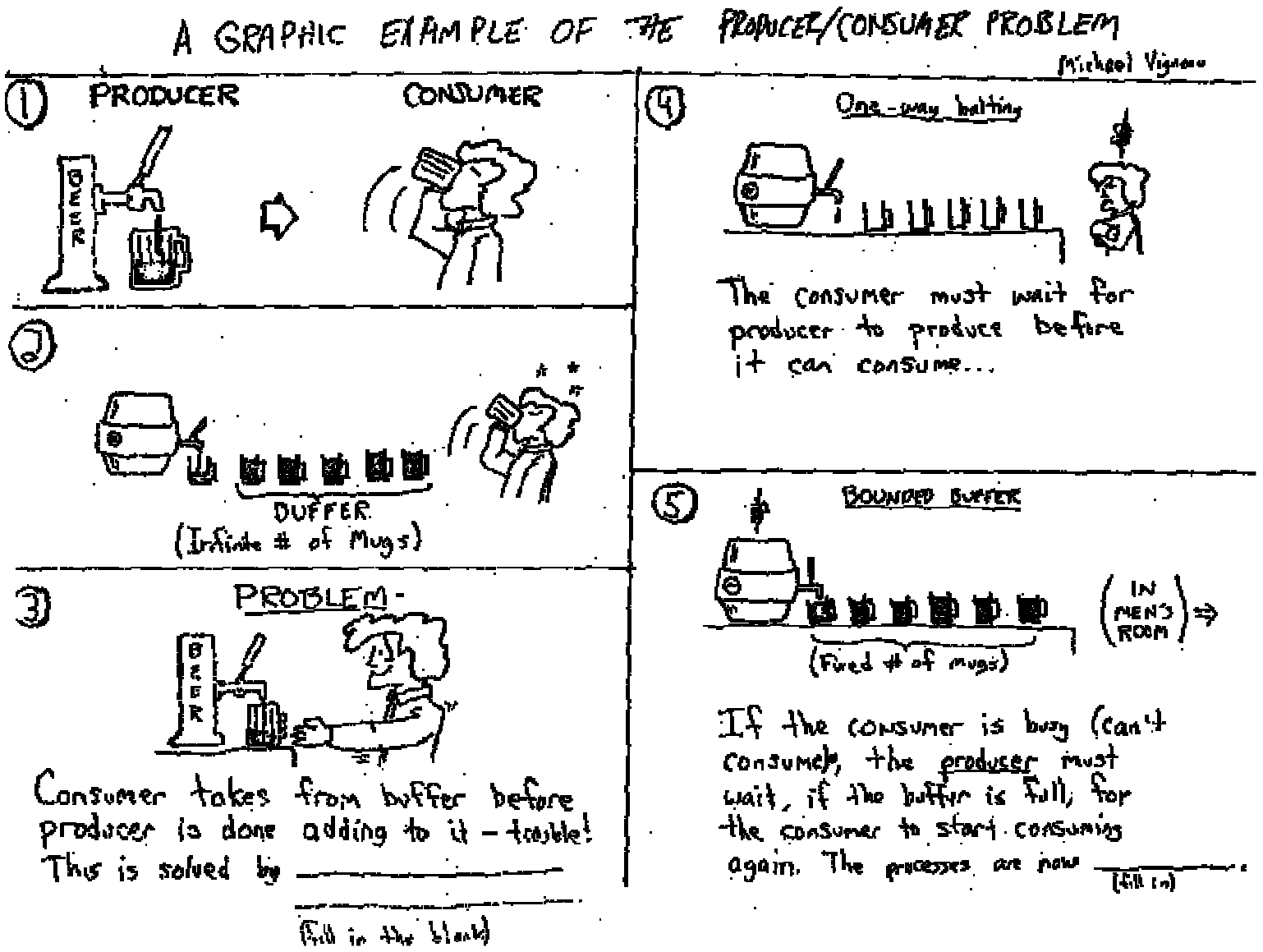
A synchronized queue example

```
Mutex lock;
CondVar dataready;
Queue queue;


AddToQueue(item) {
   lock.Acquire();                  // Get Lock
   queue.enqueue(item);             // Add item
   dataready.signal();              // Signal any waiters
   lock.Release();                  // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();                  // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();          // Get next item
   lock.Release();                  // Release Lock
   return(item);
}
```

# Basic concepts of OS: Concurrency example



# Contents

1. What is an operating system?
2. Kernel
3. System call
4. Concurrency – issues and solutions
5. **Multi process programming**
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations

# Multi process programming

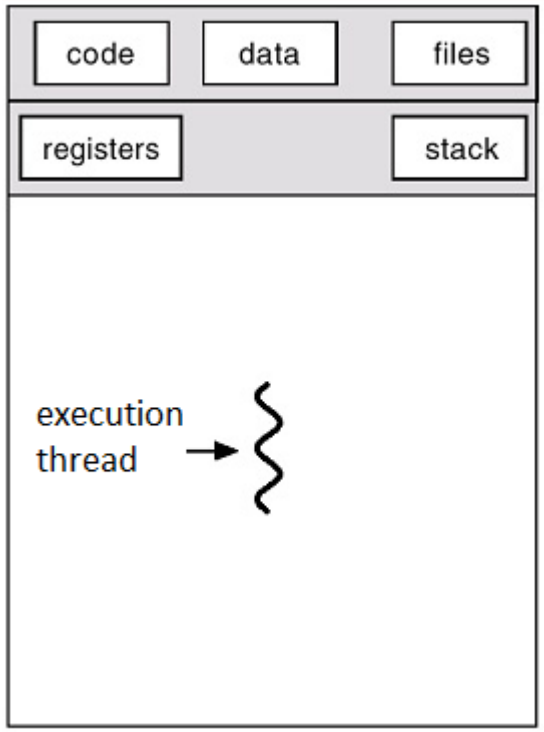
- What Is A Process
- Process Creation
  - The fork() System Call
- Child Process Termination
  - The wait() System Call

# Multi process programming

## What Is A Process?

UNIX definition:

- *An entity that executes a given piece of code, has its own execution stack, its own set of memory pages, its own file descriptors table, and a unique process ID.*



## Slide 34

**TLT(1** Tsirov, Lyubomir Todorov (L.); 30.9.2015 г.

# Multi process programming

## What Is A Process?

**A process is not a program! Several processes may be executing the same computer program at the same time, for the same user or for several different users.**

# Multi process programming

## What Is A Process?

- It might be that many different processes will try to execute the same piece of code at the same time, perhaps trying to utilize the same resources, and we should be ready to accommodate such situations. This leads us to the concept of 'Re-entrancy'.

- Re-entrancy
  - The ability to have the same function (or part of a code) being in some phase of execution, more than once at the same time.

- This re-entrancy might mean that two or more processes try to execute this piece of code at the same time.

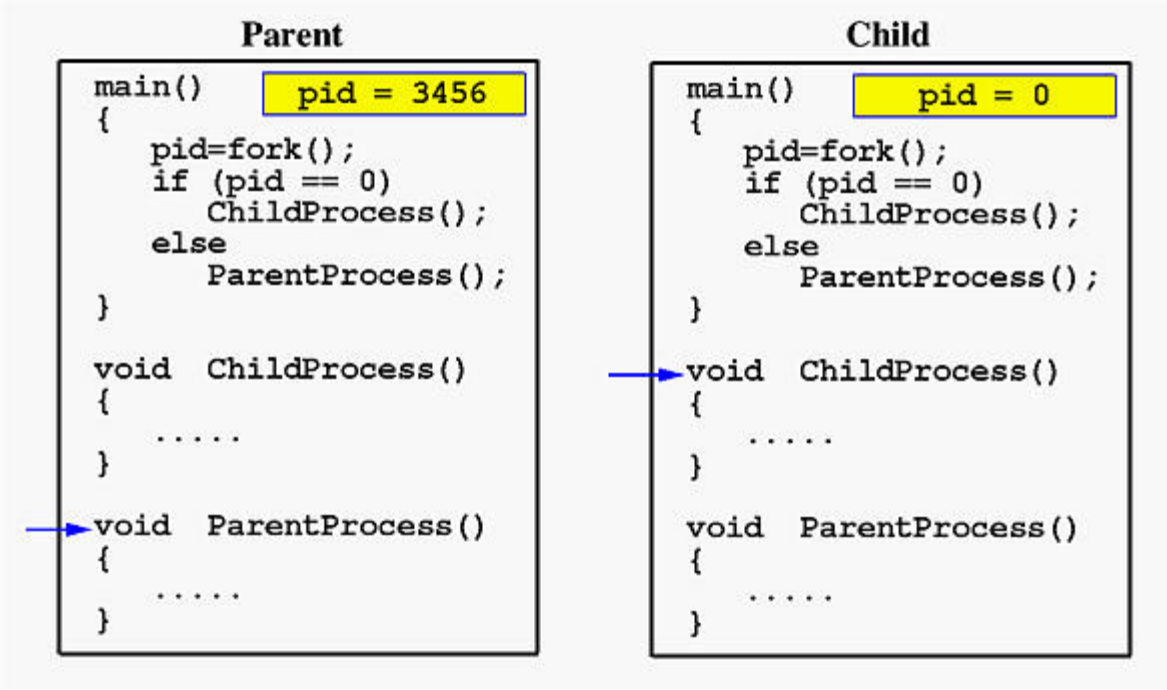
# Multi process programming

## Process Creation

- The fork() system call is the basic way to create a new process. fork() is used to produce child shell.
- Returns twice(!!!!)
- fork() causes the current process to be split into two processes
  - a parent process
  - a child process.
- All of the memory pages used by the original process get duplicated during the fork() call, so both parent and child process see the exact same memory image.

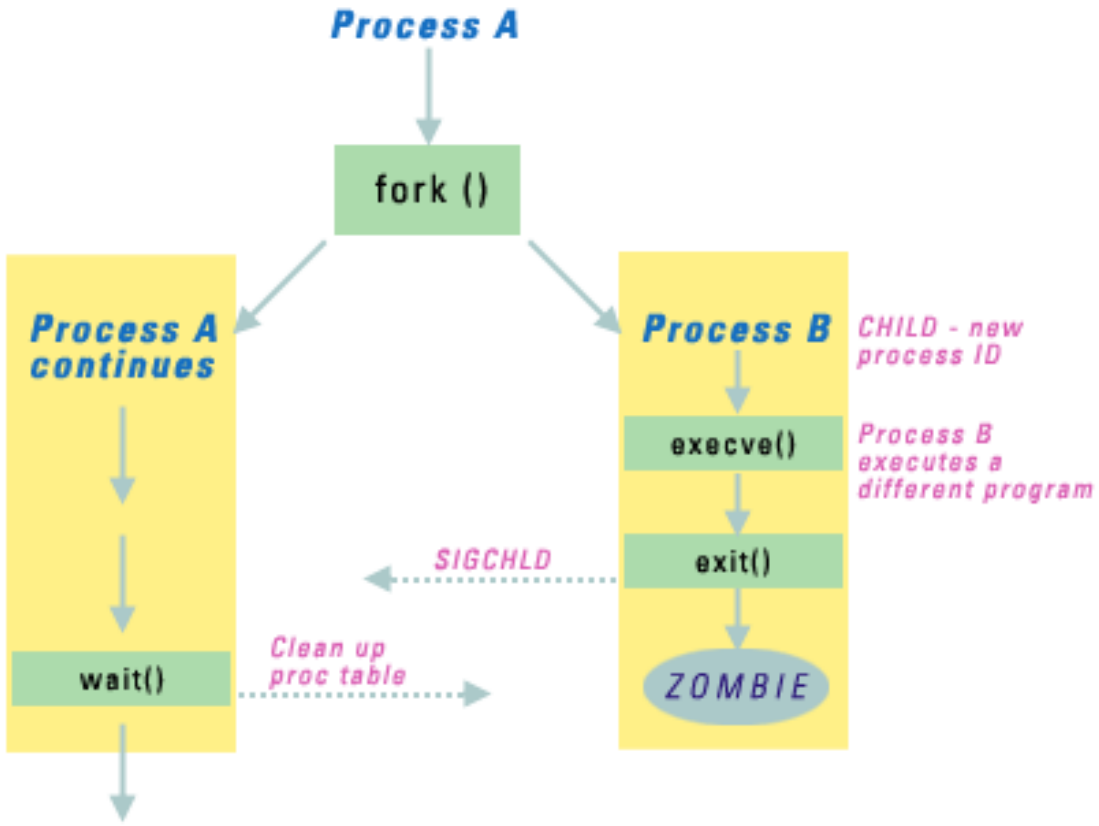
# Multi process programming

## Process Creation



# Multi process programming

## Process Creation



## Child Process Termination

Once we have created a child process, there are two possibilities.

- When a child process exits, it is not immediately cleared off the process table. Instead, a signal is sent to its parent process, which needs to acknowledge it's child's death, and only then the child process is completely removed from the system. In the duration before the parent's acknowledgment and after the child's exit, the child process is in a state called "**zombie**".
- When a process exits (terminates), if it had any child processes, they become orphans. An orphan process is automatically inherited by the 'init' process (process number 1 on normal Unix systems), and becomes a child of this 'init' process. This is done to ensure that when the process terminates, it does not turn into a zombie, because 'init' is written to properly acknowledge the death of its child processes.
- When the parent process is not properly coded, the child remains in the zombie state forever.

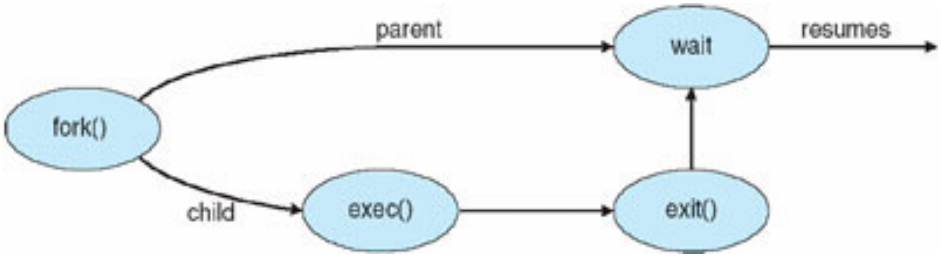
## Child Process Termination

Once we have created a child process, there are two possibilities.

- When a child process exits, it is not immediately cleared off the process table. Instead, a signal is sent to its parent process, which needs to acknowledge it's child's death, and only then the child process is completely removed from the system. In the duration before the parent's acknowledgment and after the child's exit, the child process is in a state called "**zombie**".
- When a process exits (terminates), if it had any child processes, they become orphans. An orphan process is automatically inherited by the 'init' process (process number 1 on normal Unix systems), and becomes a child of this 'init' process. This is done to ensure that when the process terminates, it does not turn into a zombie, because 'init' is written to properly acknowledge the death of its child processes.
- When the parent process is not properly coded, the child remains in the zombie state forever.

## The wait() System Call

- The simple way of a process to acknowledge the death of a child process is by using the wait() system call.
- When wait() is called, the process is suspended until one of its child processes exits, and then the call returns with the exit status of the child process.
- If it has a zombie child process, the call returns immediately, with the exit status of that process.

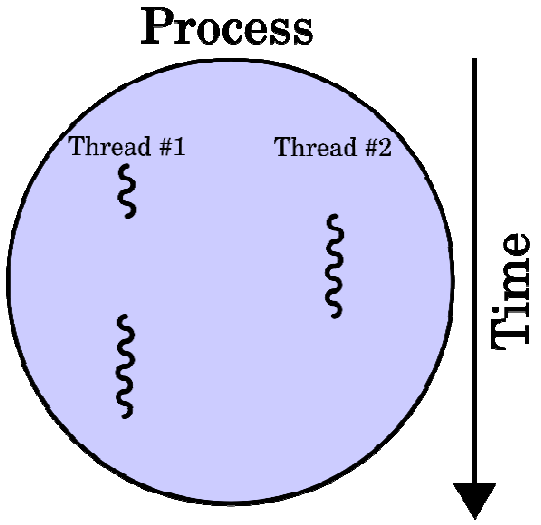


# Contents

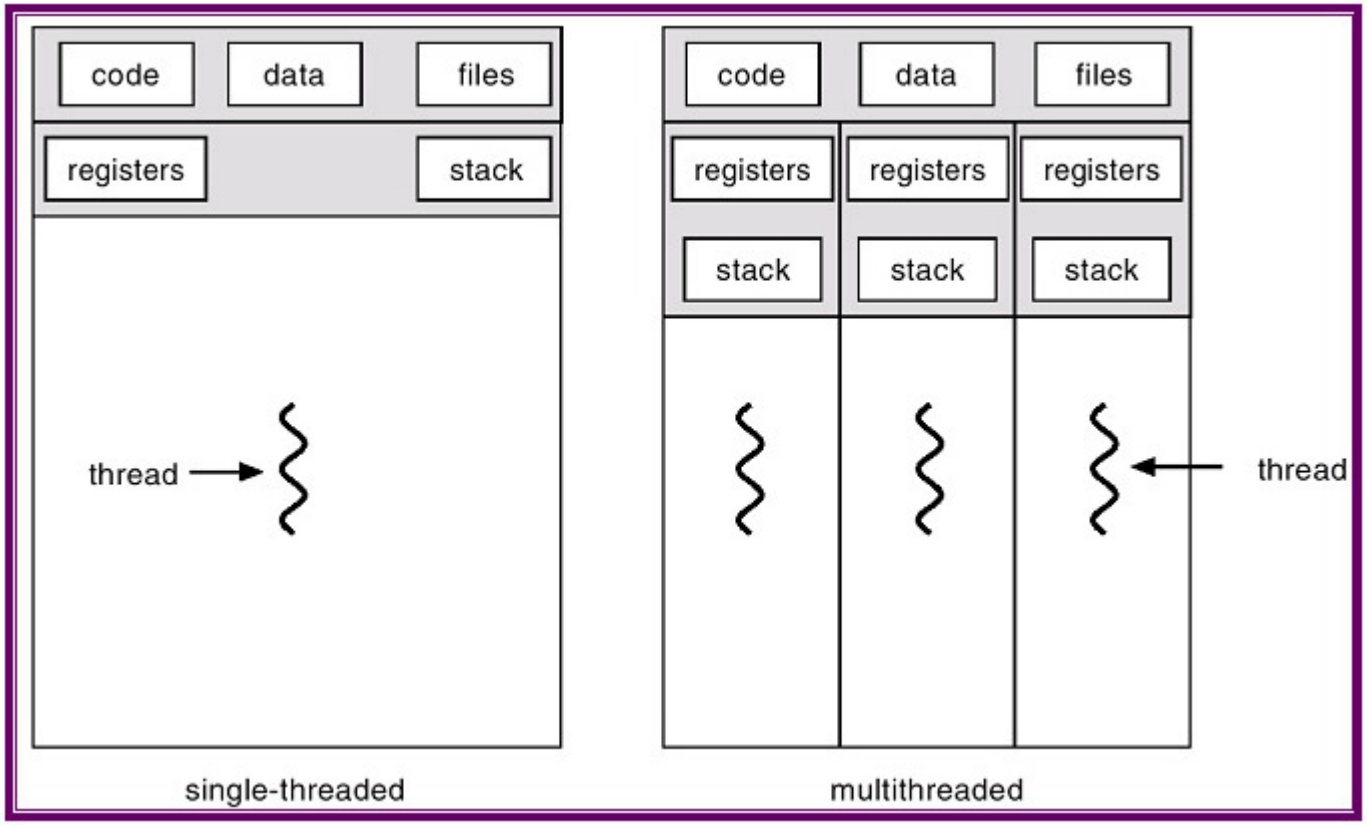
1. What is an operating system?
2. Kernel
3. System call
4. Concurrency – issues and solutions
5. Multi process programming
6. Multi thread programming
7. Advanced OS concepts
   - Scheduling(time sharing/time slicing)
   - Memory management & Virtual memory(address space)
   - I/O access
   - Interrupts
   - Real time operations

# Multi thread programming

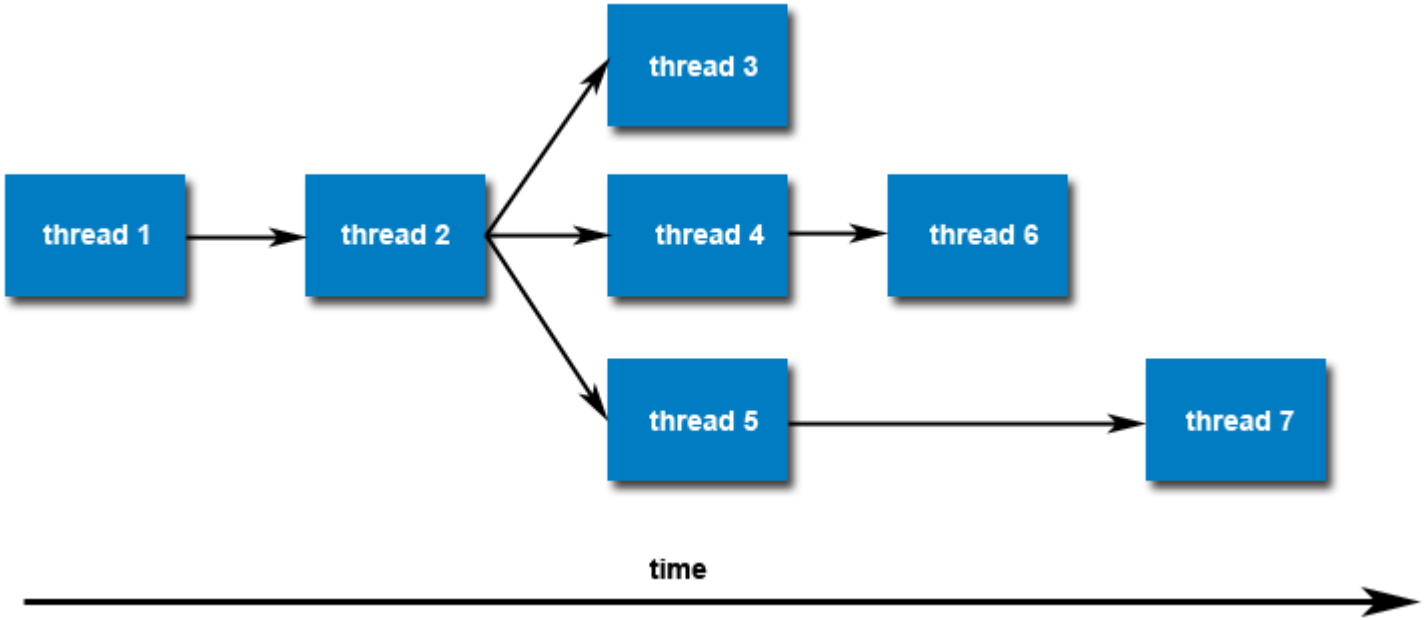
- What is a thread?
- *Definition:*
  - *a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.*
- Processes vs. threads



# Multi thread programming



# Multi thread programming



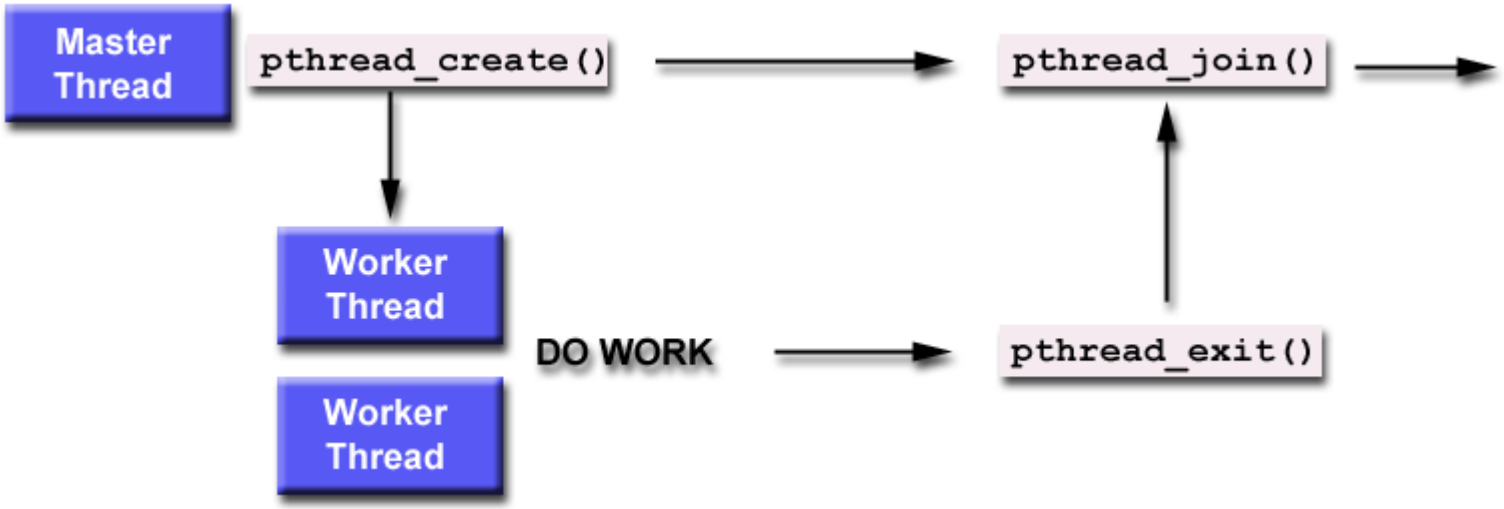
# Multi thread programming

- POSIX threads
  - The pthread library is encapsulating all of the thread functionality in the POSIX based OS( e.g. Linux, QNX…)
- Thread manipulation
  - pthread_create (thread,attr,start_routine,arg)

# Multi thread programming

- Thread manipulation
  - pthread_exit (status)
  - pthread_join (threadid,status)

# Multi thread programming



# Multi thread programming

- Thread manipulation
  - pthread_cancel (thread)

# Multi thread programming

3 main thread synchronization primitives in pthreads library:

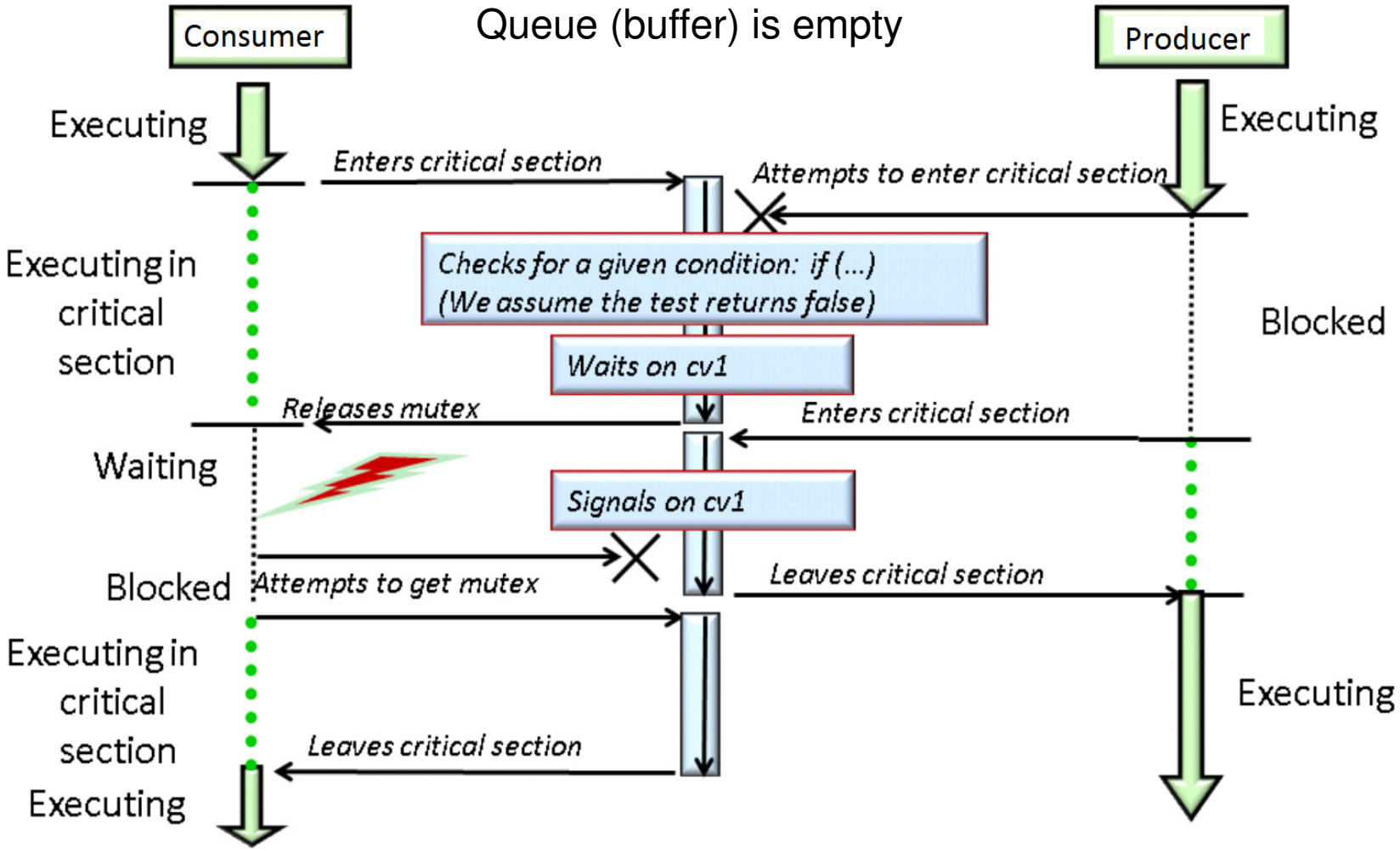
- semaphore
- mutex
- condvar

- Mutex manipulation
  - pthread_mutex_init (mutex,attr)
  - pthread_mutex_destroy (mutex)
  - pthread_mutex_lock (mutex)
  - pthread_mutex_unlock (mutex)

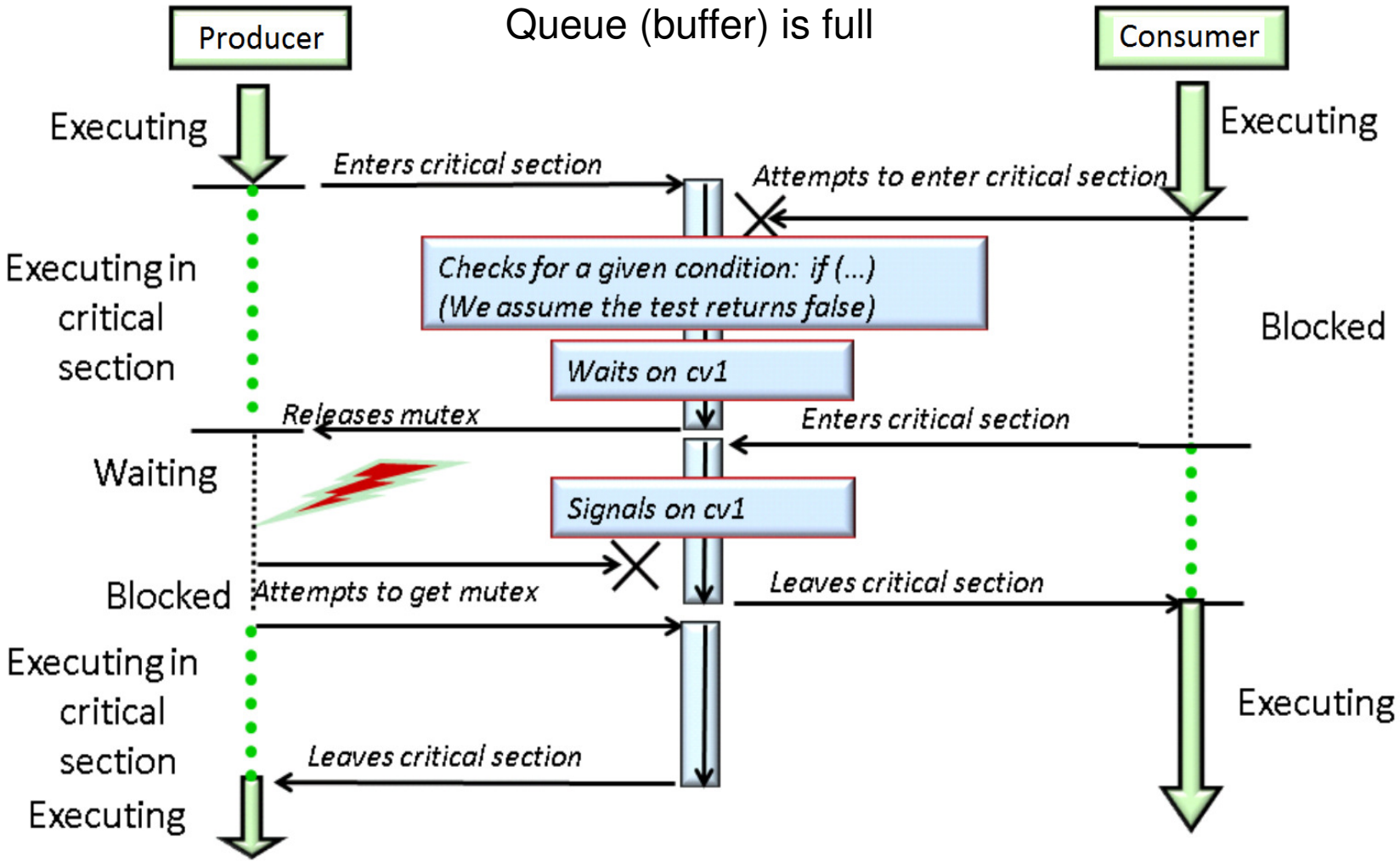
# Multi thread programming

- Condvar manipulation
  - pthread_cond_init (condition,attr)
  - pthread_cond_destroy (condition)
  - pthread_cond_wait (condition,mutex)
  - pthread_cond_signal (condition)
  - pthread_cond_broadcast (condition)

# Multi thread programming



# Multi thread programming



# End of part 1