HY-345: Operating Systems

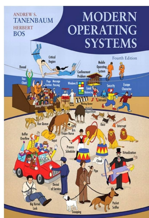
Recitation 1
Process Management - Synchronization

Papadogiannakis Manos
papamano@csd.uoc.gr

Computer Science Department
University of Crete

Overview

- **Background**
  - Processes & Threads
  - Shared Memory
  - Scheduling
- **Exercises**



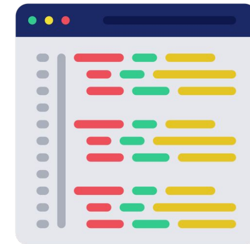
Background

Process

- A process is just an instance of a program
  - Including the current values of the program counter, registers, and variables

- Each process has an address space that contains its instructions, its data and its stack

- All the information about a process is stored in an OS table called the process table



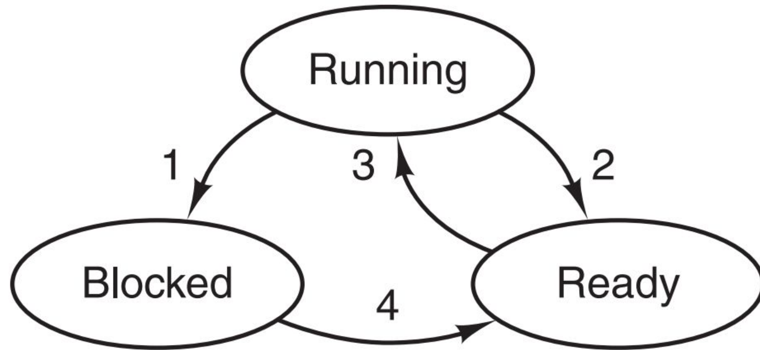
Process Details

- **Information inside a process table entry:**

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Process Details

- A process can be in one of three **states**:



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Thread

- **Threads are like lightweight processes**

- **Do we need threads?**
  - In many applications, multiple activities are going on **simultaneously**
  - Some of these may block from time to time

Thread

Why threads?

1. Lighter weight than processes, they are easier (i.e., faster) to create and destroy than processes

2. Ability for the parallel entities to share an address space and all of its data among themselves

3. Allow many activities to overlap, thus speeding up the application.

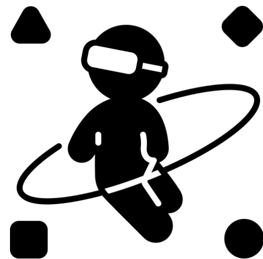
Process vs Thread

- Per-process vs Per-thread **resources**

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

User-Space Threads

- **Implement threads as a package/library entirely in user space**



- **Why?**
  - Can be implemented on operating systems that do not support threads natively

User-Space Threads

- Major problems
  - How do you implement blocking system calls?
- Suppose that a thread reads from the keyboard before any keys have been hit
- Letting the thread actually make the system call is unacceptable, since this will block all threads

Interprocess Communication

- Processes that are working together may share some common storage

- Each process can read and write some shared data

- The final result depends on who runs precisely when
  - Race Conditions

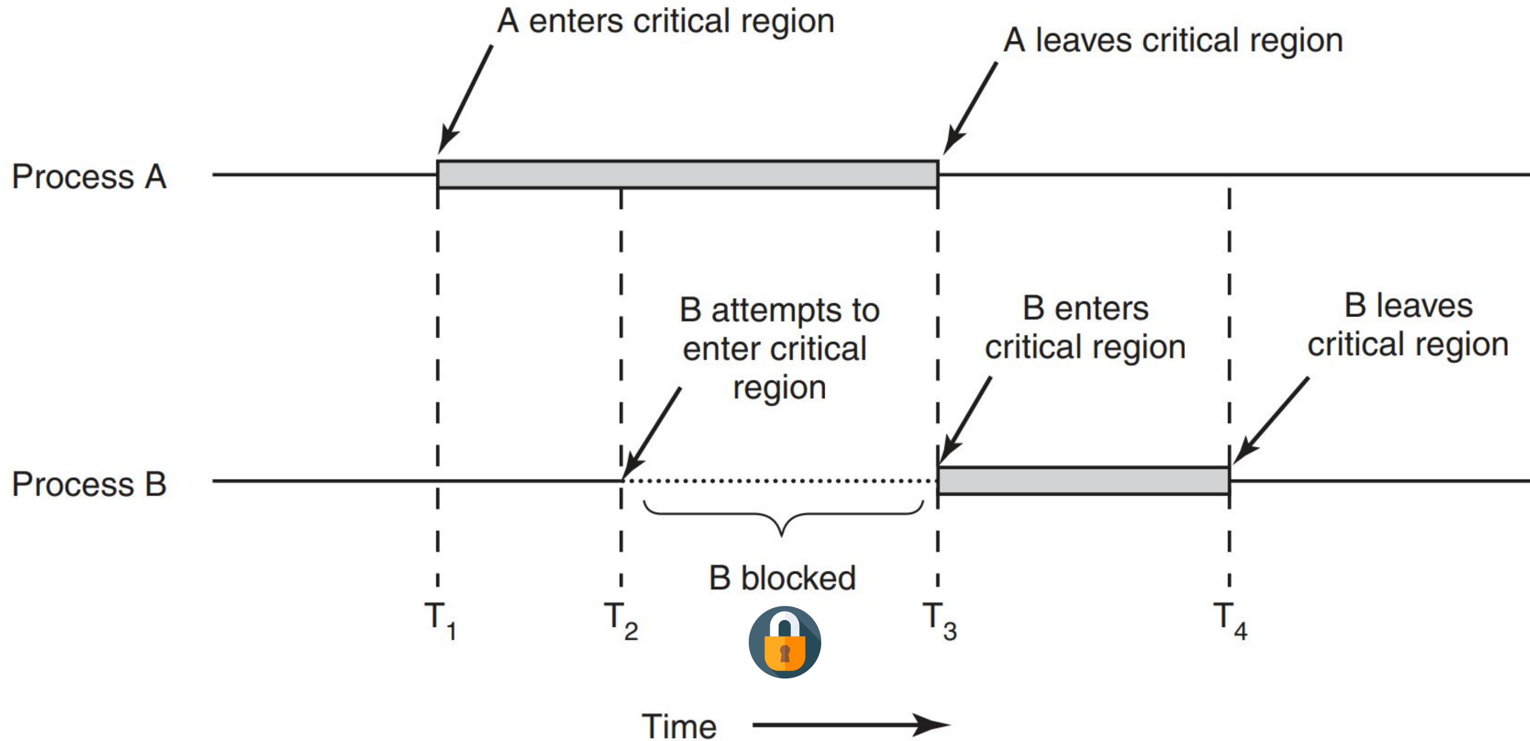
Mutual Exclusion

- **Need to prevent trouble in situations involving shared memory**

- **Mutual Exclusion**
    - Or other synchronization primitives (e.g. semaphores, locks)

- **Ensure that if one process is using a shared variable, a different process will be excluded from doing so**

Mutual Exclusion

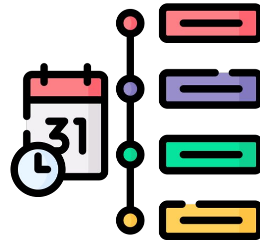


Scheduling

- A single processor may be shared among several processes
- You need to know when each process gets time on the CPU and for how long
- The scheduling algorithm determines when to stop a process and service another

Scheduler

- Coordinates how tasks share the available processor(s)
- Prevents task starvation and preserves fairness
- Algorithms:
  - First-In-First-Out
  - Shortest Job First
  - Round robin
  - …



Exercises

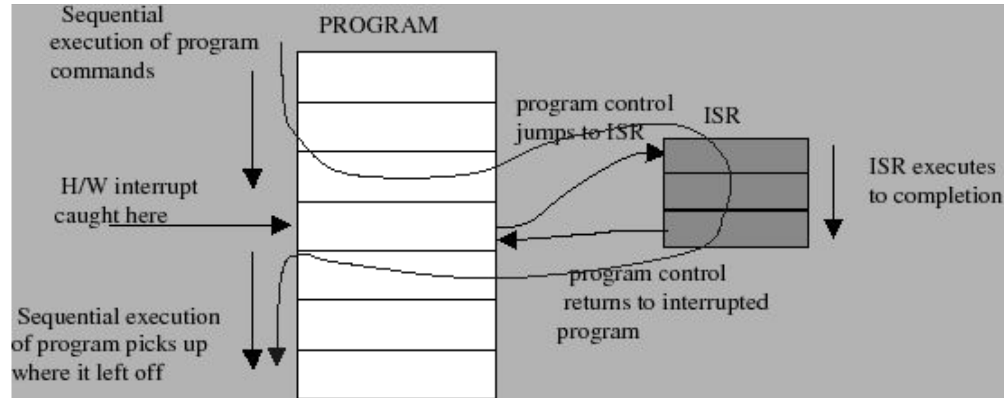
Exercise 1

- On all current computers, at least part of the interrupt handlers are written in assembly language. Why?

Exercise 1

- **On all current computers, at least part of the interrupt handlers are written in assembly language. Why?**

➢ **All interrupts start by saving the registers (e.g. program counter) to the stack and switching to kernel mode. Then, we need to set up a context and a stack for the interrupt service-procedure to run.**



Exercise 1

- **On all current computers, at least part of the interrupt handlers are written in assembly language. Why?**

➢ **Actions such as saving the registers and setting the stack pointer cannot even be expressed in high-level languages such as C, so they are performed by a small assembly-language routine.**
  - Usually the same one for all interrupts since the work of saving the registers is identical, no matter what the cause of the interrupt is.

➢ **Interrupt service routines must execute as rapidly as possible**

Exercise 2

- **When an interrupt or a system call transfers control to the operating system, a kernel stack area separate from the stack of the interrupted process is generally used. Why?**

Exercise 2

- **When an interrupt or a system call transfers control to the operating system, a kernel stack area separate from the stack of the interrupted process is generally used. Why?**

1. **First, you do not want the operating system to crash because a poorly written user program does not allow for enough stack space.**

Exercise 2

- **When an interrupt or a system call transfers control to the operating system, a kernel stack area separate from the stack of the interrupted process is generally used. Why?**

1. **First, you do not want the operating system to crash because a poorly written user program does not allow for enough stack space.**
2. **Second, if the kernel leaves stack data in a user program's memory space upon return from a system call, a malicious user might be able to use this data to find out information about other processes.**

Exercise 3

- **The register set is listed as a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.**

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

Exercise 3

- The register set is listed as a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.

➢ The thread has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables.
➢ Just like when multiprogramming processes, when a thread is stopped, it has values in registers, which must be saved

Exercise 4

- **What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?**

Exercise 4

- **What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?**

✓ **The biggest advantage is the efficiency. No traps to the kernel are needed to switch threads.**
✓ **Each process can have its own customized scheduling algorithm for its threads.**

Exercise 4

- **What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?**

✓ The biggest advantage is the efficiency. No traps to the kernel are needed to switch threads.
✓ Each process can have its own customized scheduling algorithm for its threads.
○ The biggest disadvantage is that if one thread blocks, the entire process blocks. I/O, some system calls and page faults block the entire process
○ Programmers generally want threads in applications where the threads block often (e.g. Web server). User-level threads work for CPU bound applications.

Exercise 5

- In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used?

Exercise 5

- **In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used?**

➢ **Each thread will generally call different procedures and thus have a different execution history. It must have its own stack for the local variables, return addresses, and so on. This is why each thread needs its own stack.**

Exercise 5

- **In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used?**

➢ **Each thread will generally call different procedures and thus have a different execution history. It must have its own stack for the local variables, return addresses, and so on. This is why each thread needs its own stack.**

➢ **This is equally true for user-level threads as for kernel-level threads.**

Exercise 5

- **In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used?**

➢ **Each thread will generally call different procedures and thus have a different execution history. It must have its own stack for the local variables, return addresses, and so on. This is why each thread needs its own stack.**

➢ **This is equally true for user-level threads as for kernel-level threads.**

➢ **When threads are managed in user space, each process needs its own private thread table to keep track of each thread’s program counter, stack pointer, registers, state, etc…**

Exercise 6

- **Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions.**

Exercise 6

- **Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions.**

➢ **Associated with each counting semaphore are two binary semaphores, M, used for mutual exclusion, and B, used for blocking. Also associated with each counting semaphore is a counter that holds the number of ups minus the number of downs, and a list of processes blocked on that semaphore.**

Exercise 6

- **Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions.**

➢ To implement down, a process first gains exclusive access to the semaphores, counter, and list by doing a down on M.

➢ It then decrements the counter. If it is zero or more, it just does an up on M and exits.If M is negative, the process is put on the list of blocked processes. Then an up is done on M and a down is done on B to block the process.

Exercise 6

- **Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions.**

➢ **To implement up, first M is downed to get mutual exclusion, and then the counter is incremented.**

➢ **If it is more than zero, no one was blocked, so all that needs to be done is to up M. If, however, the counter is now negative or zero, some process must be removed from the list. Finally, an up is done on B and M in that order**

Exercise 7

- Synchronization within monitors uses condition variables and two special operations, wait and signal. A more general form of synchronization would be to have a single primitive, waituntil, that had an arbitrary Boolean predicate as parameter:
  `waituntil x < 0 or y + z < n`
  The signal primitive would no longer be needed. This scheme is clearly more general but it is not used. Why not?

Exercise 7

- Synchronization within monitors uses condition variables and two special operations, wait and signal. A more general form of synchronization would be to have a single primitive, waituntil, that had an arbitrary Boolean predicate as parameter:
  `waituntil x < 0 or y + z < n`
  The signal primitive would no longer be needed. This scheme is clearly more general but it is not used. Why not?

➢ It is very expensive to implement. Each time any variable that appears in a predicate on which some process is waiting changes, the run-time system must re-evaluate the predicate to see if the process can be unblocked.

Exercise 8

- A process running on CTSS (Compatible Time Sharing System) needs 30 quanta to complete. How many times must it be swapped in, including the very first time?

Exercise 8

- **A process running on CTSS (Compatible Time Sharing System) needs 30 quanta to complete. How many times must it be swapped in, including the very first time?**

➢ **Sets up priority classes**
➢ **Always prefer processes of higher priority class**
➢ **Usually CPU-bound processes are assigned a smaller priority class than I/O-bound processes**

➢ **Each process gets 2^n quantums. If it consumes the entirety of its assigned slot, next time it will receive 2^{n+1}**

Exercise 8

- **A process running on CTSS (Compatible Time Sharing System) needs 30 quanta to complete. How many times must it be swapped in, including the very first time?**

➢ **Process gets 2^n quantums each time**

Exercise 8

- **A process running on CTSS (Compatible Time Sharing System) needs 30 quanta to complete. How many times must it be swapped in, including the very first time?**

➢ **Process gets 2^n quantums each time**

➢ **First time it get 1 quantum**
Second time, 2 quantums
Third time, 4 quantumes, etc.

Exercise 8

- **A process running on CTSS (Compatible Time Sharing System) needs 30 quanta to complete. How many times must it be swapped in, including the very first time?**

➢ **Process gets 2^n quantums each time**

➢ **First time it get 1 quantum**
Second time, 2 quantums
Third time, 4 quantumes, etc.

➢ **It will get 1 + 2 + 4 + 8 + 15**
(The last time it will actually get 16 quantumes but it needs only 15 of them)

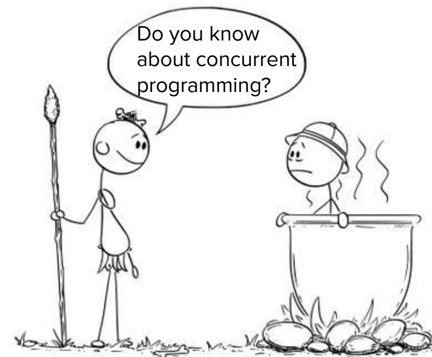
➢ **It must be swapped in 5 times.**

Exercise 9

- **A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary.**

- **When a savage wants to eat, he helps himself from the pot, unless it is empty.**

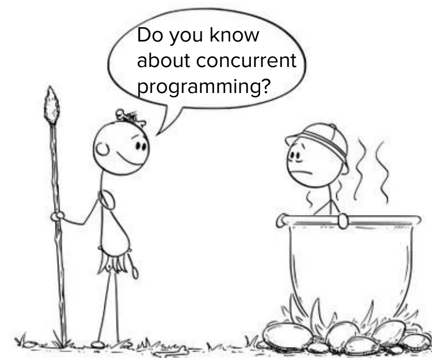
- **If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.**



Exercise 9

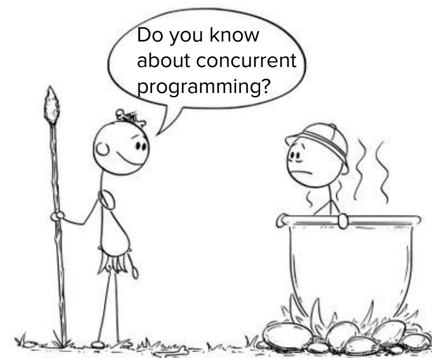
- **Constraints:**
  a. Savages cannot get a serving from the pot if the pot is empty

  b. The cook can put more servings in the pot only if the pot is empty



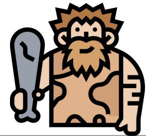
Exercise 9

- **Details:**
  - The cook and each savage are all separate threads
  - Threads compete for CPU access
  - The behavior of the system depends on the order each threads is executed



Exercise 9

- **Naive approach: Consumer - Producer**

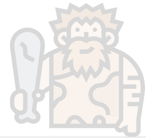


Consumer	Producer

`while True:`
`    fullServings.wait()`
`    mutex.lock()`
 	
`    eatOneServing()`
 	
`    mutex.unlock()`
`    emptyServings.post()`
	`while True:`
	`    emptyServings.wait()`
	`    mutex.lock()`
	
	`    putNewServing()`
	
	`    mutex.unlock()`
	`    fullServings.post()`

Exercise 9

- **Naive approach: Consumer - Producer**



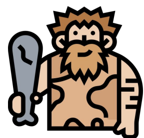
```
while True:
    fullServings.wait()
    mutex.lock()

    eatOneServing()

    mutex.unlock()
    emptyServings.post()
```

```
while True:
    emptyServings.wait()
    mutex.lock()

    putNewServing()

    mutex.unlock()
    fullServings.post()
```

"The cook can put more servings in the pot <u>only if the pot is empty</u>"

Exercise 9

- **Naive approach: Busy waiting**



`while True:`
`    fullServings.wait()`
`    mutex.lock()`
 	
`    eatOneServing()`
 	
`    mutex.unlock()`
	`while True:`
	`    mutex.lock()`
	
	`    if servings == 0:`
	`        putNewServing()`
	`        for i in 1...N:`
	`            fullServings.post()`
	
	`    mutex.unlock()`

Exercise 9

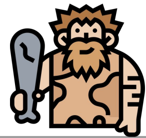
- **Naive approach: Busy waiting**

```
while True:
    fullServings.wait()
    mutex.lock()

    eatOneServing()

    mutex.unlock()
```

```
while True:
    mutex.lock()

    if servings == 0:
        putNewServing()
        for i in 1...N:
            fullServings.post()

    mutex.unlock()
```

Waste of computer resources

Exercise 9

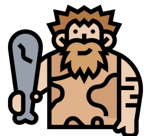


```
while True:
    mutex.lock()

    if servings == 0:
        emptyPot.post()
        fullPot.wait()

    eatOneServing()

    mutex.unlock()
```

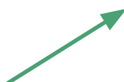
```
while True:
    emptyPot.wait()

    putServingsInPot()

    fullPot.post()
```

Note: `emptyPot` ***and*** `fullPot` ***are semaphores***

Exercise 9



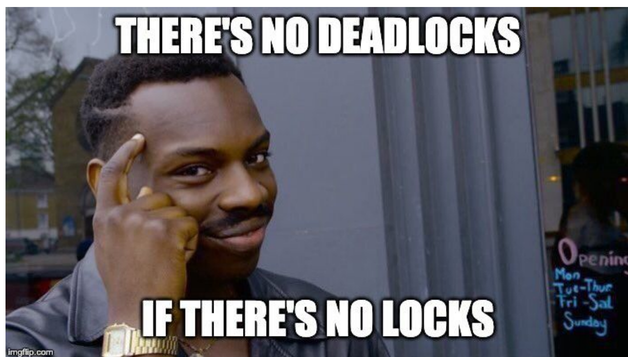
```
while True:
    mutex.lock()

    if servings == 0:
        emptyPot.post()
        fullPot.wait()

    eatOneServing()

    mutex.unlock()
```



Scoreboard Variable: Keeps track of the system's state

```
while True:
    emptyPot.wait()

    putServingsInPot()

    fullPot.post()
```

Note: *`emptyPot`* ***and*** *`fullPot`* ***are semaphores***



Credit

- Icons from FlatIcon, made by:
  - Freepik
  - Eucalyp
  - FlowIcon
  - Juicy_fish

Thank You!

papamano@csd.uoc.gr

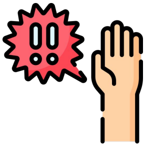
Questions?

Backup Slides

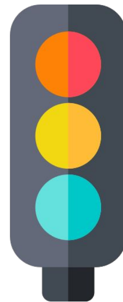
Interrupts

What happens in an OS when an interrupt occurs?

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly-language procedure saves registers.
4. Assembly-language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly-language procedure starts up new current process.



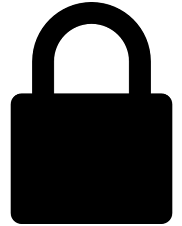
Semaphores



- **Synchronization mechanism**
  - Stores the number of remaining wakeup signals
- **Think of it as a room with specific seats**
  - You have to wait until a seat is available before you sit down
- **Both the** `down()` **and the** `up()` **operations are atomic**

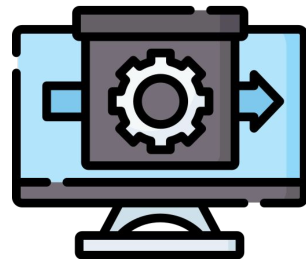
Mutexes

- **Synchronization mechanism**
  - They are a binary semaphore



- **This time the room has only one seat**
  - Only one person can enter the room at a time
- **They control access to a critical region**

Monitors



- **Synchronization mechanism**
  - Higher level than a semaphore
- **A collection of methods, variables and data structures**
  - Threads don’t have access to the monitor’s internal state
- **In order for threads to produce/consume data, they need to use the monitors methods (i.e. API)**
- **If a thread has to wait, it is placed in a queue**
  - When a thread is done with its task, it signals a thread from the queue