Operating Systems 2024/2025

TP Class 05 – Threads and synchronization (2/2)

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Some slides based on previous versions from Bruno Cabral, Paulo Marques and Luis Silva.

operating system
noun
the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

Abbreviation: OS

Source: Dictionary.com

1 2 9 0
Departamento de Engenharia Informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Updated on
28 February 2025

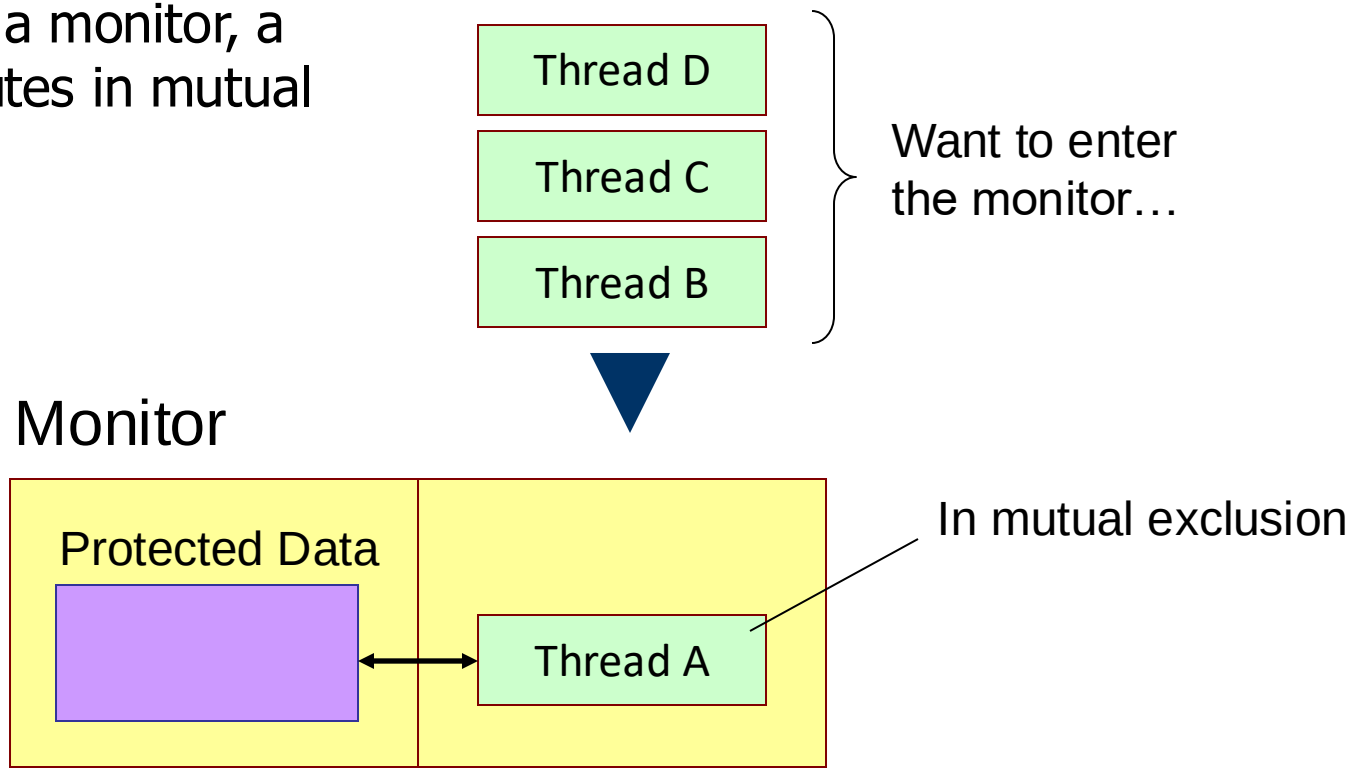
CONDITION VARIABLES

Monitors & Semaphores

- Semaphores provide an effective mechanism for synchronization, if corrected used;
- Using semaphores incorrectly can lead to errors difficult to detect as they depend on specific execution sequences;
- Wait and Signal operation may be scattered throughout a program, with a combined result difficult to preview, or incorrectly used by some programmers ;
- A monitor is a high-level programming-language synchronization construct that provides similar functionality while easier to control.

Monitors

- A monitor is an abstraction where only one thread or process can be executing at a time.
  - Normally, it has associated data
  - When inside a monitor, a thread executes in mutual exclusion



A Monitor in Java

```
import java.util.*;

public class Buffer
{
    //---------------------------------------
    private final static int MAX_SIZE = 10;

    private LinkedList<Integer> elements;
    private int totalElements;
    //---------------------------------------

    public Buffer() {
        elements = new LinkedList<Integer>();
        totalElements = 0;
    }

    //---------------------------------------

    public synchronized void putValue(int e)
    {
        // ...
    }

    public synchronized int getValue()
    {
        // ...
    }
}
```

Protected Data

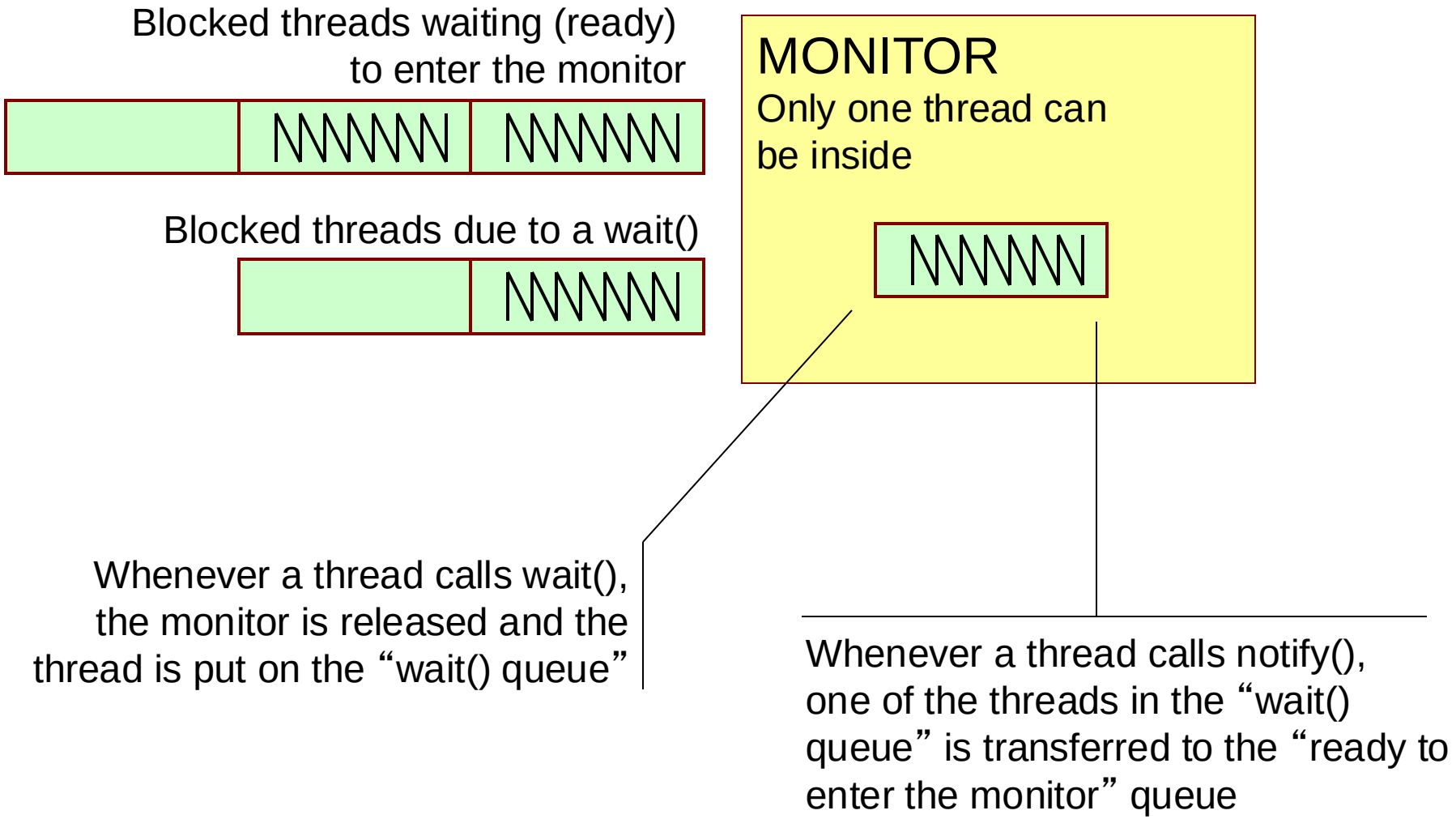
Only one thread can be inside these methods at a time; the others wait outside

Monitors (2)

- So far, monitors looked a lot like a simple MUTEX…
- Two more primitives are provided with a monitor:
  - **wait()** → Suspends the execution of the current thread, immediately relinquishing the monitor. The thread is put on a blocked threads list, waiting to be notified that "something" has changed.
  - **notify()** → Informs one of the threads that is waiting for something to change that "something" has changed. Thus, one of the awaiting threads is put on the "ready to execute" list. Note that only after the thread that has called notify exits the monitor, will the thread that was awaken be allowed to check what has changed and enter the monitor!
    [**notifyAll()** → Notifies all waiting threads to check the condition]

- Important!
  - wait() and notify() are not like wait() and post() in semaphores. If notify is called when there is no awaiting thread, it is lost. There is no associated counter, only the means to notify threads that things changed.

(A possible) Structure of a monitor



Monitors in Unix?

- In "C" you do not have monitors, but you have **CONDITION VARIABLES**
- Condition variables are somewhat like monitors. They allow the programmer to suspend a thread until a certain condition is satisfied or to notify a thread that a certain condition has changed.
  - The condition can be anything you like!
- Counting semaphores are a special type of condition variables → The condition is "**counter==0**"

Monitors in Unix?

- Locks and condition variables can be used together to create a monitor:
  - A collection of procedures manipulating a shared data structure.
  - One lock that must be held whenever accessing the shared data (typically each procedure acquires the lock at the very beginning and releases the lock before returning).
  - One or more condition variables used for waiting.

POSIX Condition Variables

Initialization

```
#include <pthread.h>

// Creates a new initialized condition variable
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// Explicitly initializes a condition variable
int pthread_cond_init(pthread_cond_t  *cond,
                      pthread_condattr_t *cond_attr);

// Signals a condition variable -- only one thread is notified
int pthread_cond_signal(pthread_cond_t *cond);

// Signals a condition variable -- all waiting threads are notified
int pthread_cond_broadcast(pthread_cond_t *cond);

// Waits on a condition variable. Mutex is release while waiting
// and automatically reaquired when a thread in unblocked
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

// Same  as pthread_cond_wait() but allows for a timeout
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);

// Release a condition variable
int pthread_cond_destroy(pthread_cond_t *cond);
```

Static initialization using a MACRO. Uses default attributes.

If NULL the attributes used are the default

Have no effect if no threads are blocked in the cond var

Synchronization – Condition Variables (2)

- Important rule:
  - A condition variable always has an associated **mutex**.
  - Always check the condition variable in mutual exclusion. The **mutex** must be locked.
- How does it work?

```
// Thread A
pthread_mutex_lock(&mutex);
while (condition() != true)
{
  pthread_cond_wait(&cond_var, &mutex);
}

// After this step we know that
// the condition is true and we
// are in mutual exclusion

// ...

pthread_mutex_unlock(&mutex);
```

Makes thread A check its condition again

```
// Thread B
pthread_mutex_lock(&mutex);

// Do something that may make condition()
// change: notify "Thread A" to re-check it
pthread_cond_signal(&cond_var);

pthread_mutex_unlock(&mutex);
```

Synchronization – Condition Variables (3)

- The thread tests a condition in mutual exclusion. If the condition is false, pthread_cond_wait() **atomically releases de mutex** AND **waits until someone signals that the condition should be tested again**.
- When the **condition is signaled** AND **the mutex is available**, pthread_cond_wait() **atomically reacquires the mutex** AND **releases the thread**.

```
// Thread A
pthread_mutex_lock(&mutex);
while (condition() != true)
{
  pthread_cond_wait(&cond_var, &mutex);
}

// After this step we know that
// the condition is true and we
// are in mutual exclusion

// ...

pthread_mutex_unlock(&mutex);
```

- pthread_cond_signal() indicates that **exactly one blocked thread should test the condition again**. Note that this is not a semaphore. If there is no thread blocked, the "signal" is lost.
- If all threads should re-check the condition, use pthread_cond_broadcast(). Since a mutex is involved, each one will test it one at a time, in mutual exclusion.

```
// Thread B
pthread_mutex_lock(&mutex);

// Do something that may make condition()
// change: notify "Thread A" to re-check it
pthread_cond_signal(&cond_var);

pthread_mutex_unlock(&mutex);
```

Synchronization – Condition Variables (4)

- The condition **must always** be tested with a while loop, **never an if**! Being unlocked out of a condition variable only means that the condition must be re-checked, not that it has become true!
- The condition must always be **checked** and **signaled** inside a locked mutex.

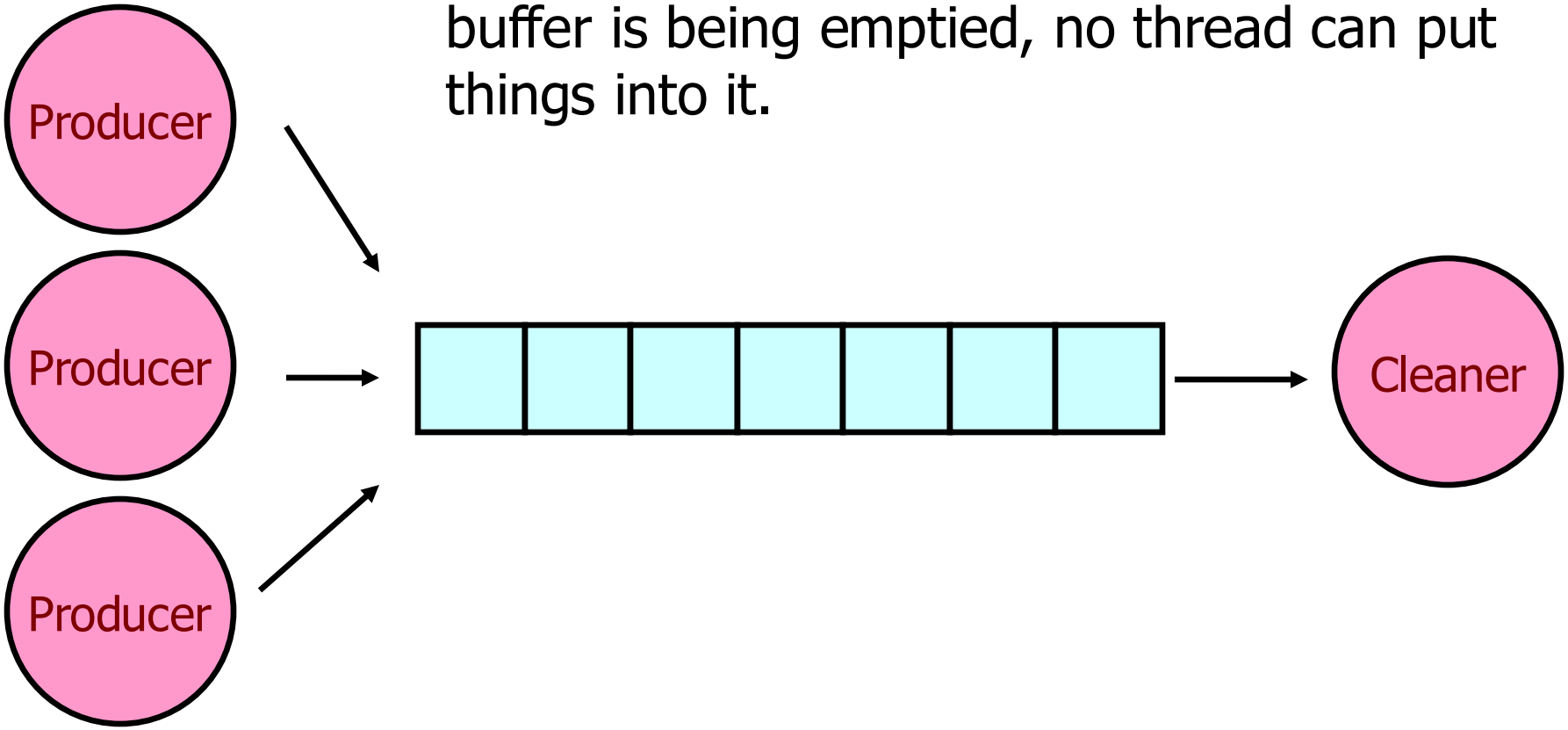
While condition() **may be** true while Thread B is executing, something may happen between the time that the condition is signaled and Thread A is unblock (e.g. another thread may change the condition)

```
// Thread A
pthread_mutex_lock(&mutex);

if (condition() != true)
  pthread_cond_wait(&cond, &mutex);

// Critical zone
// ...

pthread_mutex_unlock(&mutex);
```

```
// Thread B
pthread_mutex_lock(&mutex);

// ... something that may make condition() to change
pthread_cond_signal(&cond);

pthread_mutex_unlock(&mutex);
```

Example - Buffer Cleaner

- Suppose a buffer that can hold a maximum of N elements. When it is full, it should immediately be emptied. While the buffer is being emptied, no thread can put things into it.



Example

buffer_cleaner_with_threads.c

```
void* producer(void *id) {
  int my_id = *((int*) id);

  while (1) {
    pthread_mutex_lock(&mutex);

    // if buffer is full, notify CLEANER
    while (write_pos == N) {
      pthread_cond_signal(&is_full);
      pthread_cond_wait(&go_on,&mutex);
      }

    printf("[PRODUCER %3d] Writing %d into the buffer\n", my_id, my_id);
    buf[write_pos] = my_id;
    write_pos++;

    pthread_mutex_unlock(&mutex);
    sleep(1);
  }
}
```

Example
buffer_cleaner_with_threads.c (2)

```
void* cleaner(void *arg) {
  while (1) {
    pthread_mutex_lock(&mutex);

    // Wait until it is full
    while (write_pos != N) {
      pthread_cond_wait(&is_full,&mutex);
      }

    printf("[CLEANER] Cleaning buffer: ");
    for(int i=0;i<N;i++) printf("[%d]",buf[i]);
    printf("\n");
    write_pos=0;

    //notify everyone that is waiting in the condition variable
    pthread_cond_broadcast(&go_on);
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}
```

Example

Result with 5 producers and one cleaner...

```
user@user-virtualbox:~/Desktop/Aulas/so/tp05$ gcc -Wall -pthread buffer_cleaner_with_threads.c -o bufferc
user@user-virtualbox:~/Desktop/Aulas/so/tp05$ ./bufferc
[PRODUCER   0] Writing 0 into the buffer
[PRODUCER   1] Writing 1 into the buffer
[PRODUCER   2] Writing 2 into the buffer
[PRODUCER   3] Writing 3 into the buffer
[PRODUCER   4] Writing 4 into the buffer
[CLEANER] Cleaning buffer: [0][1][2][3][4]
[PRODUCER   1] Writing 1 into the buffer
[PRODUCER   0] Writing 0 into the buffer
[PRODUCER   2] Writing 2 into the buffer
[PRODUCER   3] Writing 3 into the buffer
[PRODUCER   4] Writing 4 into the buffer
[CLEANER] Cleaning buffer: [1][0][2][3][4]
[PRODUCER   2] Writing 2 into the buffer
[PRODUCER   3] Writing 3 into the buffer
[PRODUCER   1] Writing 1 into the buffer
[PRODUCER   0] Writing 0 into the buffer
[PRODUCER   4] Writing 4 into the buffer
[CLEANER] Cleaning buffer: [2][3][1][0][4]
[PRODUCER   3] Writing 3 into the buffer
[PRODUCER   1] Writing 1 into the buffer
[PRODUCER   2] Writing 2 into the buffer
[PRODUCER   0] Writing 0 into the buffer
[PRODUCER   4] Writing 4 into the buffer
^C
user@user-virtualbox:~/Desktop/Aulas/so/tp05$
```

Condition Variables between processes

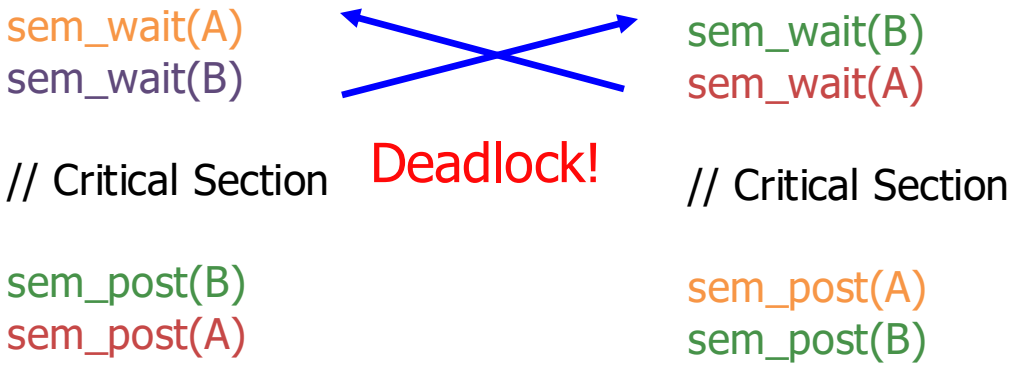
- Condition variables can be used between different processes.
- In POSIX condition variables it implies the modification of the default attributes and using a condition variable in shared memory.
- **Note:** the same happens with pthread mutexes (already seen before)

See class demos!

BASIC RULES FOR SYNCHRONIZATION

Basic rules for synchronization

- Never Interlock waits!
  - Locks should always be taken in the same order in all processes
  - Locks should be released in the reverse order they have been taken



  - One way to assure that you always take locks in the same order is to create a lock hierarchy. I.e. associate a number to each lock using a table and always lock in increasing order using that table as reference (index).

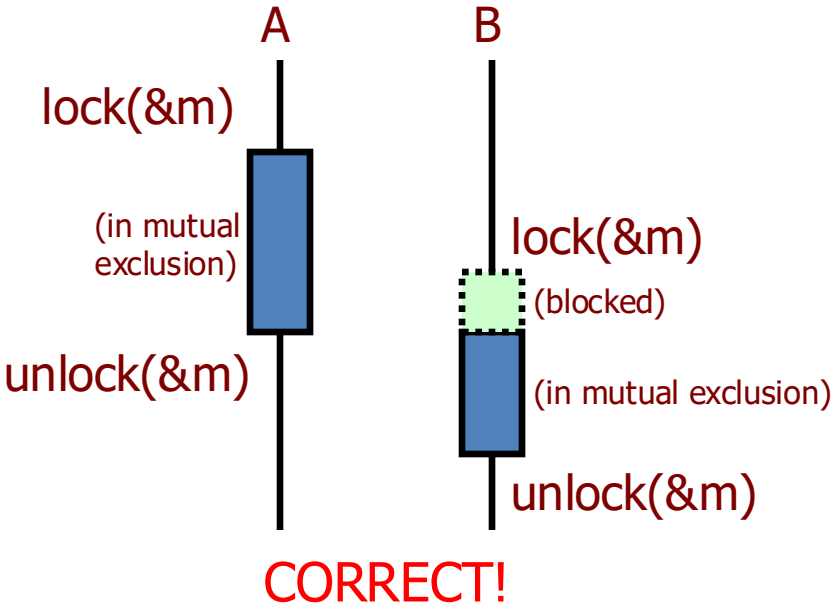
Basic rules for synchronization (2)

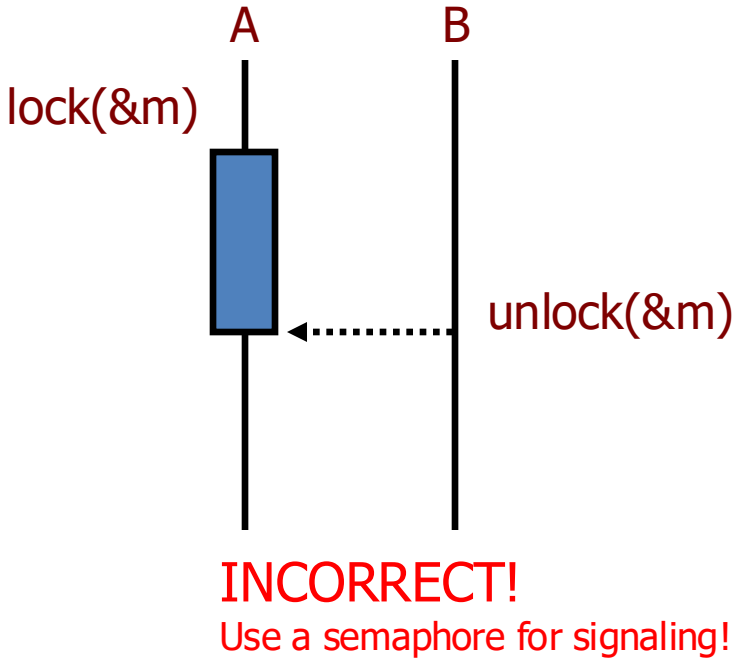
- Sometimes it is not possible to know what order to take when locking (or using semaphores)
  - Example: you are using two resources owned by the operating system. They are controlled by locks. You cannot be sure if another application is not using exactly the same resources and locking in reverse order.
- In that case, use `pthread_mutex_trylock()` or `sem_trywait()` and back off if you are unsuccessful.
  - Allow the system to make progress and not deadlock!

```
// Try to acquire both resources
while (true)
{
  // Acquire the first resource
  pthread_mutex_lock(&lockA);

  // Try to acquire the second one
  if (pthread_mutex_trylock(&lockB) != 0)
  {
    // Failed, back off
    pthread_mutex_unlock(&lockA);
    usleep(BACKOFF_DELAY);
  }
  else
    break;
}

// In mutual exclusion
// ...

// Release the resources
pthread_mutex_unlock(&lockB);
pthread_mutex_unlock(&lockA);
```

Basic rules for synchronization (3)

- Mutexes (pthread_mutex) are used for implementing mutual exclusion, not for signaling across threads!!!
  - Only the thread that has locked a mutex can unlock it. Not doing so will probably result in a core dump!
- To signal across threads use semaphores!





Some important concepts to remember!

- **Deadlock**
  When two or more processes are unable to make progress being blocked waiting for each other. All processes are in a waiting state.

- **Livelock**
  When two or more processes are alive and working but are unable to make progress.

- **Starvation**
  When a process is not being able to access resources that its needs to make progress

Class demos included

- Demo01 – Buffer cleaner with threads and condition variables

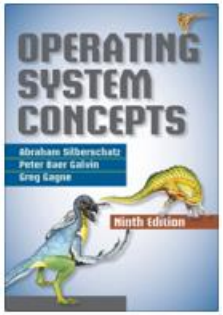
  `buffer_cleaner_with_threads.c`
- Demo02 - Buffer cleaner with processes, condition variables and mutex

  `buffer_cleaner_condvar_between_procs.c`
- Demo03 – An example with condition variables

  `condvar_ex1.c`
- Demo04 – Another example with condition variables

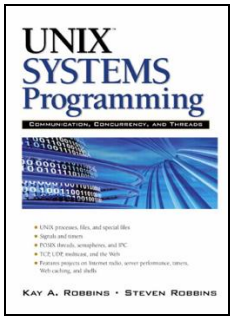
  `condvar_ex2.c`

References



- [Silberschatz13]

  Chapter 4: Threads
  - All chapter 4



- [Robbins03]

  Chapter 12: POSIX Threads

  Chapter 13: Thread Synchronization

  Chapter 14: Critical Sections and Semaphores

INTRODUCTION TO ASSIGNMENT 06 – "THREADS AND SYNCHRONIZATION II"

Thank you! Questions?

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.
—Rudyard Kipling