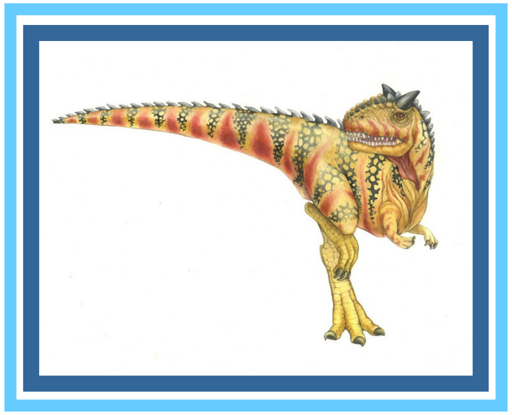
Theoretical Unit 4

(Book Chapter 6)

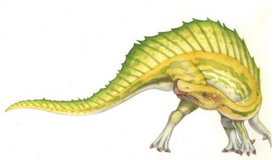
Process Synchronization

(edited & enhanced by rufino@ipb.pt, 2025/2026)



Unit 4: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Mutex Locks and Semaphores
- Classic Synchronization Problems
- Exercises



Objectives

- Present the concept of **process synchronization**
- Introduce the **critical-section problem**, whose solutions can be used to ensure the consistency of shared data
- Present software and hardware solutions of the critical-section problem
- Examine several classical process-synchronization problems
- Explore several tools that are used to solve process synchronization problems

Theoretical Unit 4

4.1 Background

Background

- Processes execute **concurrently**
  - May be interrupted at any time, **partially completing execution**
- **Concurrent access to shared data** may result in **data inconsistency**
- Maintaining data consistency requires mechanisms to ensure the **orderly execution of cooperating processes**
- **Example of the issues at stake:**
  - Two persons A and B have access to the same bank account
  - Currently, the account balance is 100€
  - Suppose A is depositing 1€, while B is withdrawing 1€
  - The final balance should still be 100€, but it may be different if the deposit by A is allowed to intermix with the withdrawal by B
    - see next slide for an explanation of how this situation unfolds

Race Condition

- Let **`balance`** be a variable in RAM representing the account balance
- **`balance++`** could be implemented as

```
register = balance
register = register + 1
balance = register
```

- **`balance--`** could be implemented as

```
register = balance
register = register - 1
balance = register
```

- Consider this execution interleaving with "**`balance = 100`**" initially:

Time	Process	Operation	Effect
T0	A	`registerA = balance`	{registerA = 100}
T1:	A	`registerA = registerA + 1`	{registerA = **101**}
T2:	B	`registerB = balance`	{registerB = 100}
T3:	B	`registerB = registerB – 1`	{register2 = **99**}
T4:	A	`balance = registerA`	{balance = **101**}
T5:	B	`balance = registerB`	{balance = **99**}

There is a **race condition**: the outcome depends on the order threads/processes access shared data.

Theoretical Unit 4

4.2 The Critical Section Problem

Critical Section Problem (CSP)

- Consider a set of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has a **critical section** segment of code
  - Process may change common variables, write common file, etc.
  - When one process enters its critical section …
  - … no other should be allowed to enter in its own critical section
- ***Critical section problem*** is how to design a “protocol” to solve this
  - Note, however, that ensuring **mutual exclusion** will not be enough !
- General structure of process ***P_i***
  - Processes must “ask permission” in an **entry section** to enter the **critical section**; when leaving it, processes relinquish their permission in an **exit section**; all the other code is represented by the **remainder section**
  - Undefined lifetime represented by infinite loop

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Solution to the Critical-Section Problem

- Requisites of a solution to the Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assumes that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Peterson's Solution for 2 Processes

- **Peterson Solution:**
  classic software-based solution for *n*=2 processes (P_0 and P_1)
- **Algorithm 1**: a 1st attempt (no yet a valid solution)
  - The processes share one integer variable (**`turn`**)
    - `int turn; // whose turn is to enter critical section`
    - `(turn == i)` ➔ P_i `may enter critical section (i=0,1)`
    - `i == 0` ➔ `j == 1; i == 1` ➔ `j == 0; initially turn = 0`

```
do {
    while (turn != i) {;}
         critical section
    turn = j;
         remainder section
 } while (true);
```

  - Complies with Mutual Exclusion and Bounded Waiting
  - Fails on Progress, because enforces strict alternation

Peterson's Solution for 2 Processes

- **Algorithm 1**: a situation where progress is not ensured

```
while (1) {
      while (turn != i) ;
      critical section
      turn = j;
      remainder section
}
```

Instant	P0 (i=0, j=1)	P1 (i=1, j=0)	turn (initially 0)
t0	while (turn != i); (False)		0
t1	critical section		0
t2		while (turn != i); (True)	0
t3	critical section		0
t4	turn = 1		1
t5	remainder section		1
t6		while (turn != i); (False)	1
t7		critical section	1
t8		turn = 0	0
t9		remainder section	0
t10		while (turn != i); (True)	0

At instant t10, P1 is the only process interested in accessing the critical section, but is unable to do so, once it passed the turn to P0, whom is still in the remainder section. Therefore, progress is being violated.

Peterson's Solution for 2 Processes

- **Algorithm 2**: a 2nd attempt (again, no yet a valid solution)
  - The processes share one array of two integers (**flag**)
    - `Boolean flag[2]; // indicates if a process is interested`
      `                 // (ready) to enter the critical section`
    - `(flag[i] == True)` ➔ P_i `ready to enter critical section`
    - `i == 0` ➔ `j == 1; i == 1` ➔ `j == 0`
    - `initially flag[2]={False,False}; // no one interested`

```
do {
   flag[i] = True;
   while (flag[j] == True) {;}

       critical section
   flag[i] = False;

       remainder section
 } while (true);
```

  - Complies with Mutual Exclusion and Bounded Waiting
  - Fails on Progress if the two processes meet at entry section

Peterson’s Solution for 2 Processes

- **Algorithm 2**: a situation where progress is not ensured

```
while (1) {
        flag[i] = True;
        while (flag[j] == True);
        critical section
        flag[i] = False;
        remainder section
}
```

Instant	P0 (i=0, j=1)	P1 (i=1, j=0)	flag: | False | False |
t0	`flag[0] = True`		| True | False |
t1		`flag[1] = True`	| True | True |
t2		`while (flag[0] == True); (True)`	| True | True |
t3	`while (flag[1] == True); (True)`		| True | True |

At instant t2, P1 is blocked at the entry section because it sees that P0 is interested in entering the critical section. At instant t3, P0 is blocked at the entry section because it sees that P1 is interested in entering the critical section. With neither process in the critical section, and both wanting to enter, neither can do so. Thus, the progress requirement is violated.

Peterson's Solution for 2 Processes

- **Algorithm 3**: an apparently working solution (*)
  - The processes share the variables of both Alg. 1 and Alg. 2
    - `initially turn = 0; // turn is P`$_0$`'s by default`
    - `initially flag[2]={False,False}; // no one interested`

```
   do {
          flag[i] = true;
          turn = j;
          while (flag[j] == True && turn == j) {;}
                 critical section
          flag[i] = false;
                 remainder section
    } while (true);
```

  - Complies with all 3 requisites
  - But it is not a reliable solution im current architectures (*)

Peterson’s Solution for 2 Processes

- **Why is Algorithm 3 unreliable in current architectures ?**
  - **Compilers** may **reorder independent instructions** to gain performance
  - **Reordering** may also happen in the **CPU during execution**, to fully use the pipeline (memory access instructions take many cycles, during which the CPU keeps the pipeline busy with other, independent instructions)
  - **What happens in Alg. 3 if the first two instructions switch order ?**

```
turn = j;
flag[i] = true;
while (flag[j] == True && turn == j) {;}
          critical section
```

  - **SMP/Multicore Systems** also introduce new problems: with 2 or more threads/processes changing the shared variables `turn` and `flag`, the new values may take some time to propagate to all CPUs/cores caches
    - This delay may introduce opportunities for new race conditions

Peterson’s Solution for 2 Processes

- **Algorithm 3’**: a situation where mutual exclusion is not ensured

```
while (1) {
        turn = j; // B
        flag[i] = True; // A
        while (flag[j] == True && turn == j);
        critical section
        flag[i] = False;
        remainder section
}
```

Instant	P0 (i=0, j=1)	P1 (i=1, j=0)	flag: | False | False |	turn: 0
t0	turn = 1		| False | False |	1
t1		turn = 0	| False | False |	0
t2		flag[1] = True	| False | True |	0
t3		while (flag[0] == True **(F)** && turn == 0 **(T)**); **(F)**	| False | True |	0
t4		critical section	| False | True |	0
t5	flag[0] = True		| True | True |	0
t6	while (flag[1] == True **(T)** && turn == 1 **(F)**); **(F)**		| True | True |	0
t7	critical section		| True | True |	0

The change in order between A and B, compared to the original algorithm, created an opportunity for a situation in which both processes are able to be in the critical section, thus violating the mutual exclusion requirement.

Theoretical Unit 4

4.3 Synchronization Hardware

Synchronization Hardware

- Software-based solutions cannot solve the CSP anymore
- In modern architectures, Hardware-assisted solutions are needed

- Race conditions can be avoided in a single-core system by turning off/not attending interrupts while executing the critical section code
  - but this option is not scalable in multi-processor/multi-core systems
  - it takes some time to tell all CPUs/cores to inhibit interrupts; that time will delay the entry in the critical section, decreasing system efficiency
  - turning off/ignoring interrupts also prevents updating the system clock

- Alternative: ISA features and instructions that help solving the CSP
  - Memory Barriers / Memory Fences
  - Atomic instructions (Test-and-Set, Compare-and-Swap, etc.)
    - atomic = indivisible (uninterruptible)

Memory Barriers

- **Memory Barriers/Fences**: instructions that force the propagation of changes in memory to the caches of all processors/cores, thus ensuring a consistent global view of the main memory contents
  - Ensure that all pending Loads and Stores are finalized
  - Prevent the reordering of the surrounding instructions
  - Have a negative performance impact (use with caution; low-level operations, typically used by kernel developers)
  - **Peterson Solution (Algorithm 3) with Memory Barriers:**

```
do {
      flag[i] = true;
      memory_barrier();
      turn = j;
      while (flag[j] == True && turn == j) {;}
                critical section
     . . .
```

test_and_set Instruction

- **TestAndSet (TAS)**: sets the parameter to the TRUE value, and returns the original value of the parameter

```
boolean test_and_set (boolean *target) {
        boolean rv = *target;
        *target = TRUE;
        return rv;
}
```

- CSP with TAS:
  - Shared boolean variable **`lock`** initialized to FALSE

```
do {
      while (test_and_set(&lock)) {;} // busy wait
            critical section
      lock = false;
             remainder section
} while (true);
```

  - Mutual Exclusion and Progress: Yes; Bounded Waiting: No

compare_and_swap Instruction

- **CompareAndSwap (CAS)**: if the 1st parameter matches the 2nd parameter (compare), it is assigned the 3rd parameter (swap); always returns the original value of the 1st parameter

```
int compare_and_swap(int *value, int expected, int new_value) {
        int temp = *value;
        if (*value == expected) *value = new_value;
        return temp;
}
```

- CSP with CAS:
  - Shared integer variable `lock` initialized to 0

```
do {
     while (compare and swap(&lock, 0, 1)) {;} // busy wait
          critical section
     lock = 0;
          remainder section
} while (true);
```

  - Mutual Exclusion and Progress: Yes; Bounded Waiting: No

Bounded-waiting with compare_and_swap

- CSP with CAS variant that satisfies all 3 PSC requirements
  - Shared variables: **boolean waiting[n]={false,…,false};**
    int lock=0;
  - Private variables: **int key, j; // j=0,1,…,n-1**

```
do {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i) lock = 0;
   else waiting[j] = false;
   /* remainder section */
} while (true);
```

Atomic Variables

- Typically, the **CompareAndSwap (CAS)** instruction is not used directly, but as building block of other tools that deal with the CSP
- **Atomic Variables** are one of such tools based on the CAS instruct.
  - can be used to ensure mutual exclusion when a single shared variable is being updated (e.g, a shared counter is modified)
  - systems that support atomic variables provide special atomic data types and functions to access and change atomic variables
  - example: `increment(&counter)` where `increment` is

```
void increment(atomic_int *v) {
       int temp;
       do {
            temp = *v;
       } while (temp != compare_and_swap(v, temp, temp+1));
 }
```

  - Atomic Variable ensure atomic updates, but leave some problems unsolved: if some processes test the same condition (busy waiting) on an atomic counter, they all may be unblocked by an atomic change in another process

Theoretical Unit 4

4.4 Mutex Locks and Semaphores

Mutex Locks: Concept

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers have built other software tools to solve the CSP
- The simplest of those high-level tools is the **mutex lock**
- Protect a critical section by
  - first **`acquire()`** a lock on entry
  - then **`release()`** the lock on exit
  - Boolean variable indicates if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Implemented via atomic instructions / atomic variables
- This solution requires **busy waiting**
  - This lock therefore is called a **spinlock**
- But this solution only ensures Mutual Exclusion and Progress
  - It does not ensure Bounded Waiting (see related exercise)

Mutex Locks: Operations and Usage

-
  ```
  void acquire(boolean *lock) {
     while (*lock == True) {;} // busy wait
     *lock = True;
  }
  ```

-
  ```
  void release(int *lock) {
     *lock = False;
  }
  ```
- Shared boolean **`L`** (lock) initialized to **`False`**
- Process P_i :

```
     do {
        acquire(&L)
          critical section
        release (&L)
          remainder section
      } while (true);
```

Semaphores with Busy Waiting

- A semaphore is a synchronization tool that provides more sophisticated ways (than Mutex Locks) for process to synchronize their activities
- A 1st version of semaphores behaves similarly to a mutex lock:
  - Represented by an integer variable (**`S`**), accessed via two atomic operations, **`wait()`** and **`signal()`**, originally called **`P()`** and **`V()`**
  - Definition of the **`wait()`** operation:

```
void wait(int *S) {
    while (*S <= 0) {;} // busy wait
    (*S)--;
}
```

  - Definition of the **`signal()`** operation:

```
void signal(int *S) {
    (*S)++;
}
```

Semaphores with Busy Waiting

- **Counting semaphore**
  - integer value can range over an unrestricted non-negative domain
  - example: initial value 3 means up to 3 processes can move into CS
- **Binary semaphore**
  - integer value can range only between 0 and 1; same as a **mutex lock**
  - may be used to implement a counting semaphore (see book for details)
- Can solve other synchronization problems, in addition to critical section
  - Consider $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ that require $\boldsymbol{Task_1}$ to happen before $\boldsymbol{Task_2}$;
    Assuming a shared semaphore `S` initialized to 0, this is solved as:

```
P1:
    Task1;
    signal(S);
P2:
    wait(S);
    Task2;
```

Semaphores with Busy Waiting

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution
- Moreover, this semaphores implementation only ensures Mutual Exclusion and Progress; again, Bounded Waiting is not ensured

Semaphores without Busy Waiting

- This is (in essence) the implementation used in the practical classes
- For each semaphore there are now two fields
  - **`value`** - this is basically a counter
  - **`list`** - queue of processes blocked on the semaphore

```
typedef struct{
   int value;
   struct process *list;
} semaphore_t;
```

- The **`wait`** and **`signal`** operations are shown in the next slide;
  they depend on two auxiliary operations:
  - block – the caller process is put in the semaphore waiting queue
  - wakeup(P) – move process P to the ready queue

Semaphores without Busy Waiting

```
void wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

void signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Semaphores without Busy Waiting

- In this implementation, the semaphore counter (**`value`**) may be <0; in that situation, the modulus is the number of processes in **`list`**
  - that is, the number of processes blocked on the semaphore
- The atomicity of **`wait`** and **`signal`** must still be ensured
  - single-processor systems: enough to inhibit interrupts during those operations (also applicable to mutex lock and active semaphores)
  - multi-processor systems: inhibiting interrupts on all CPUs takes time and harms performance; use synchronization or spinlocks to protect **`wait`** and **`signal`** (regarded as critical sections)
    - doesn't eliminate completely active waiting
    - but active waiting up to 10 machine-instructions acceptable
- Assuming that the semaphore **`list`** is FIFO-managed, Bounded-Waiting is ensured (as well as Progress and Mutual Exclusion)

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

```
          P0                         P1
      wait(S);                   wait(Q);
     wait(Q);                    wait(S);
        ...                        ...
     signal(S);                 signal(Q);
     signal(Q);                 signal(S);
```

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended (e.g., happens by unqueuing processes in LIFO order)
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Theoretical Unit 4

4.5 Classical Synchronization Problems

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffer cells, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
        ...
         /* produce an item in next_produced */
        ...
         wait(empty);
         wait(mutex);
          ...
         /* add next produced to the buffer */
          ...
         signal(mutex);
         signal(full);
 } while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
     wait(full);
     wait(mutex);
         ...
        /* remove an item from buffer to next_consumed */
         ...
     signal(mutex);
     signal(empty);
         ...
        /* consume the item in next consumed */
         ...
 } while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait(rw_mutex);
          ...
        /* writing is performed */
          ...
        signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

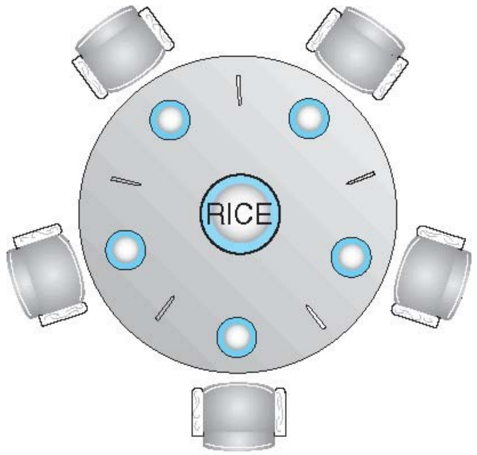
- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
             wait(rw_mutex);
        signal(mutex);
         ...
        /* reading is performed */
         ...
        wait(mutex);
        read count--;
        if (read_count == 0)
            signal(rw_mutex);
        signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i* :

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

                //  eat

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

                //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section
  - Use an asymmetric solution – an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Theoretical Unit 4

4.6 Exercises

Exercise 4.1: Critical Section Problem Requisites

Consider the next algorithm, executed concurrently in a single-core machine by processes P0 and P1, where `lock` is an integer variable (initialized to 0) shared by P0 and P1. Assume as non-atomic (divisible) the entry and exit sections.

```
while (1) {
        while (lock == 1) {;}   // A        // entry section
        lock = 1;               // B

        ...                     // C        // critical section

        lock = 0;               // D        // exit section

        ...                     // E        // remaining section
}
```

a) Based on the table below, present a situation that demonstrates that this algorithm does not respect the **Mutual Exclusion** requirement of a solution to the Critical Section Problem with 2 processes. For each instant **t+n** (n=0,1,...), fill each cell in columns **P0** and **P1** with a letter corresponding to an action in the main program (A, B, C or D), or leave it empty if the process in question does not perform any action at that moment (for each line, there can only be one action performed). Also, fill in the **lock** column with the value that this variable assumes immediately after executing the action carried out in each line. Note: the line at time **t+0** is already filled, and the table has the minimum number of lines necessary for the demonstration; however, if you find it convenient, you can add any lines you deem necessary to support your solution.

Instant	P0	P1	lock
t+0	**A**		**0**
t+1			
t+2			
t+3			
t+4			
t+5			

b) Using the same methodology, demonstrate that Bounded Waiting is also not respected (assume that the 1st line of the table is filled in the same way as in a), and that you can add more lines to support your solution).

Exercise 4.2: Semaphores and Deadlocks

Two processes, A and B, execute concurrently a sequence of 6 actions. Some actions are already defined, as well as their order: in process A, the 2nd and 3rd actions are calling functions f1() and f2(); in process B, the 3rd and 4th actions are the invocation of functions g1() and g2(). The other actions are yet to be defined, but they must be Wait and Signal operations on semaphores with passive waiting, for a semaphore S the Wait(S) operation must always precede the Signal(S) operation, and other actions may be executed between these two operations.

Process A	Process B
f1();	
f2();	g1();
	g2();

a) Assuming that processes A and B share 2 semaphores X and Y, with initial value 1, and that each process must perform Wait and Signal operations on both semaphores, complete the table below with the actions of each process over time, in order to ensure that all planned actions are carried out (i.e., there is no deadlock). Also, assume execution on a machine with a single execution core, where only one action can be executed at any given moment (either in A or in B). Additionally, fill in the last two columns with the value of the semaphores counters immediately after executing each action. Note: the line at time t+0 is already filled, and the table has the exact number of lines needed for the requested demonstration.

Instant	Process A	Process B	X Counter	Y Counter
t+0	Wait(X)		0	1
t+1				
t+2				
t+3				
t+4				
t+5				
t+6				
t+7				
t+8				
t+9				
t+10				
t+11				

b) Using the same methodology, fill in the table below in order to generate a deadlock scenario (assume that the 1st row of the table is filled in the same way as in a) and that this time you may not need to fill in all the rows in the table).

Theoretical Unit 4

References:

- "Operating System Concepts, 10th Ed.", Silberschatz & Galvin, Addison-Wesley, 2018: Chapters 6 and 7

- “The Weirdest Bug in Programming - Race Conditions” (https://www.youtube.com/watch?v=bhpzTWtee2A)

- “The 80’s Algorithm to Avoid Race Conditions (and Why It Failed)” (https://www.youtube.com/watch?v=QAzuAn3nFGo)

End of Theoretical Unit 4