Intelligent Systems Engineering Department
The National School of Artificial Intelligence
2nd Year, Semester 2 — 2024/2025

Operating Systems

Tutorial Worksheet 03

Description. This tutorial worksheet covers the topic of process synchronization.

1. **Concepts Understanding**

   a) What is a critical section in a program?

   b) Explain how old uniprocessor operating systems (e.g., earlier version of Unix) handled the critical section problem.

   c) Explain the two styles of using semaphores.

   d) Can a process be interrupted (context-switched) during the execution of its critical section? Explain your answer.

   e) Why disabling interrupt system during critical section execution is not a good idea in multiprocessor systems?

   f) What are the three requirements for the critical section problem. Explain them.

   g) What are the two main advantages of implementing the primitives `acquire()` and `release()` using queues instead of using a busy-waiting statement in `acquire()` and an incrementation statement in `release()`.

   h) Semaphores are mutual-exclusive accessed shared variables used to set up synchronization among cooperative processes. Explain why semaphores cannot be used when cooperative processes are executing on different computers?

2. **Race condition and non-determinism**

   Let x and y be two shared variables stored in the main memory. Assuming that the two variables are initialized to 10, what can be the possible outcomes for x and y if the following two processes are executed concurrently?

   Process A : `x = x + y`   Process B : `y = x * x`

3. **Critical section**

Consider the following program in which two codes C_1 and C_2 are executed concurrently. These two codes share certain variables and use private variables as well.

Begin
  Integer: `x, y, t, p, u, r, h, a, b, c;`

  CoBegin C_1; C_2; **CoEnd**
End

Code C_1:	Code C_2:
Begin	**Begin**
`b = r + 4;`	`y = t - p;`
`c = x * b;`	`x = x + 3;`
`a = a % b;`	`t = u * x;`
`a = y - 55;`	`h= x * r + y;`
End	**End**

a) Identify critical sections in each code. Just surround the statements that compose a critical section in the given code and name each critical section (e.g., CS_1, CS_2, ...). Note that multiple critical sections may exit in one single code.

b) On the above code, use semaphores to implement the entry and exit section for each identified critical section (i.e., make use of `P(sem)` to acquire and `V(sem)` to release a semaphore `sem`). Your implementation should not lead to a deadlock or to mutual exclusion violation. Also, for each semaphore that you use, you need to declare it and initialized it

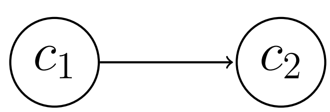
4. **Process precedence graphs**

Draw a precedence graph (a.k.a., dependency graph) that reflects the parallelism in the following code. Recall a precedence graph is a *directed graph* in which the nodes (vertices) represent processes and the edges represent the causal relation between two processes. If an edge exists from process node P_i to P_j then the process P_j must start executing its code c_j only when process P_i finishes executing its code c_i. Also we use the notation `ParBegin` and `ParEnd` to indicate a block in which program codes (or instructions) must run in parallel. Another alternative notation that can be found in the literature is `CoBegin` and `CoEnd` for concurrent begin/end.

```
Begin
  c_0
  ParBegin
    c_1
    Begin
      c_2; c_3; c_4
    End
    c_5
    c_6
    Begin
      c_7; c_8; c_9
    End
  ParEnd
  c_10
End
```

5. **Using semaphores in waiting-style**

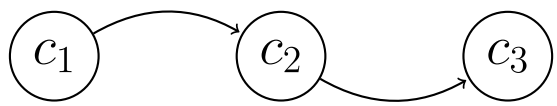
   a) Complete the following program code to reflect the process precedence graph shown below (where c_i refers to the code-i). This code basically consists of creating two processes in parallel: the first process executes program P_1 and the second process executes program P_2. You should use one semaphore for that.



ParBegin P1, P2 **ParEnd**

Program P1:	**Program P2:**
Begin	**Begin**
code$_1$	**code**$_2$
End	**End**

b) Complete the following program code to reflect the process precedence graph shown below (where c_i refers to the code-i). This code basically consists of creating two processes in parallel: the first process executes program P_1, and the second process executes program P_2. You should use semaphores for that.



ParBegin P1, P2 **ParEnd**

Program P1:	**Program P2:**
Begin	**Begin**
code$_1$; **code**$_3$**;**	**code**$_2$
End	**End**

c) Let Process P_1 and P_2 be two processes that execute concurrently using a shared semaphore s initialized to 10. Each process increments, in an infinite loop, an integer i (process P_1) or j (process P_2) both initialized to 0. After executing for a while, the values of i and j will change considerably. In this case, what will be the relation between the two variables?

Begin
Semaphore M = 10;
Integer i = 0;
Integer j = 0;
ParBegin P_1, P_2 **ParEnd**
End

P1	**P2**
Begin	Begin
while(true) { P(M); i++; }	for(int x=10; x>=0; x- -) { j++; V(M); }
End	End

a) $i = j$ b) $j = i + 10$ c) $i \leq j$ d) $i = j + 10$ e) $i \geq j + 10$

d) Give the precedence graph for the following code and check whether a deadlock may occur or not. If there is a deadlock, then modify the codes to fix that:

Begin
Semaphore M = 0;
Semaphore N = 0;
ParBegin P_1, P_2 **ParEnd**
End

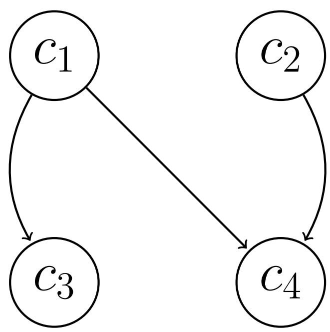
```
P1            P2
Begin         Begin
    P(N)          P(M)
    Code 1        Code 2
    V(M)          V(N)
End           End

ParBegin P1, P2 ParEnd
```

e) Modify the code of the previous question so that the two programs P_1 and P_2 execute alternatively (i.e., $\ldots, P_1, P_2, P_1, P_2, P_1, \ldots$ with P_1 being the first to start execution.

f) Modify the previous program (using two semaphores) so that the execution order becomes $\ldots, P_1, P_2, P_2, P_1, P_2, P_2, P_1, \ldots$) with P_2 being the first to start execution.

6. **CoBegin/CoEnd with PPGs**

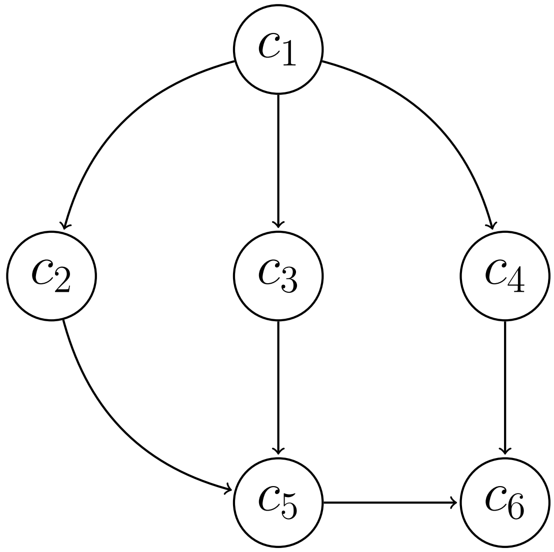
Consider the following notation: let Bc_1 (for Begin c_1) expresses the starting of the code c_1 and Ec_1 (for End c_1) the ending of the code c_1, and let the precedence graph shown below reflects the execution order of program codes c_1, c_2, c_3, and c_4. Then, do the execution sequences shown below respect the precedence graph constraints or not?



a) $Bc_1Ec_1Bc_2Ec_2Ec_3Bc_3Ec_4Bc_4$
b) $Bc_1Bc_2Ec_2Ec_1Bc_3Bc_4Ec_3Ec_4$
c) $Bc_1Bc_2Ec_1Ec_2Bc_3Bc_4Ec_3Ec_4$
d) $Bc_1Bc_2Ec_2Ec_1Bc_3Ec_3Bc_4Ec_4$
e) $Bc_1Bc_2Ec_2Bc_3Ec_1Bc_4Ec_4Ec_3$
f) $Bc_1Bc_2Ec_2Bc_3Ec_3Ec_1Bc_4Ec_4$
g) $Bc_1Bc_2Ec_1Bc_3Ec_2Ec_3Bc_4Ec_4$
h) $Bc_2Ec_2Bc_1Ec_1Bc_4Ec_4Bc_3Ec_3$

7. **Writing synchronization code for PPG**

Write the program code that reflects the following precedence graph using six processes $P_1, \ldots, P_6$ each executing its dedicated code $c_1, \ldots, c_6$. You are required to use three semaphores.



8. **Peterson's algorithm**

Consider the Peterson's algorithm for the critical section problem with two processes.

Process 0:

```
Begin
 while(true)
 {
    (1) Flag[0]=true;
    (2) turn=1;
    (3) while(flag[1] && turn==1);
    (4) ...  Critical Section ...;
    (5) flag[0] = false;
 }
End
```

Process 1:

```
Begin
 while(true)
 {
    (1) Flag[1]=true;
    (2) turn=0;
    (3) while(flag[0] && turn==0);
    (4) ...  Critical Section ...;
    (5) flag[1] = false;
 }
End
```

a) If the two statements (1) and (2) are swapped in both processes, state whether the altered code still satisfies the three requirements of the critical section problem. Explain which requirement has been violated.

b) If we change the logical operator in the condition of the `while` statement (Line 3) from `and` to `or` (i.e., `while(flag[1-i] && turn==1-i)` becomes `while(flag[1-i] || turn==1-i)`), state whether the altered code still satisfies the three requirements of the critical section problem. Explain which requirement has been violated.

9. **Atomic instructions: Test-&-Swap**

Hardware solutions to the critical section problem use read-modify-write instructions: these instructions are indivisible because the CPU gets to hold onto the memory bus for two cycles, to both read and then update the shared memory location. As an example, here is code for implementing mutual exclusion using a Swap () machine-code instruction; that instruction has the opcode EXCH in the Pentium instruction set. Each process P_i executes the same code to access its critical section, where `key` is a boolean local variable and `lock` is a shared memory location among processes P_i that contains a boolean value, initially set to `false`.

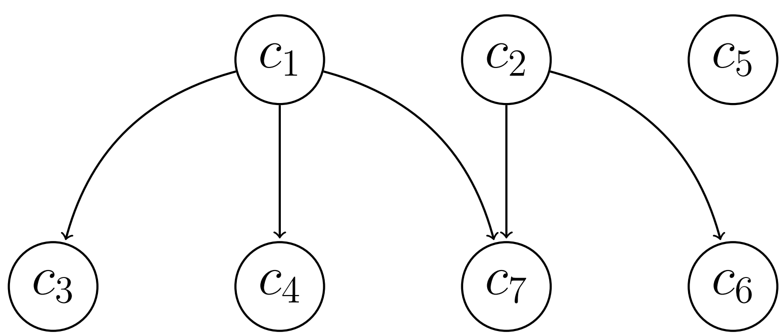
Process P_i:

```
do {
        key = true;
         while (key == true) Swap(&lock, &key);
         ... Critical section of Pi ...
         lock = false
      } while(true);
```

Describe what happens if three processes try to enter their critical section at about the same time. How does the given code ensure that only one process gets entry?

10. **Relaxing precedence constraints in PPGs**

Write CoBegin/CoEnd code for the following precedence graph. Make your program express as much parallelism as possible within the limitations of CoBegin/CoEnd, while being sure to enforce all the constraints that are in the precedence graph. (The best solutions introduce two or three extra precedence edges. Try to find one of them.)



11. **Classical synchronization problem (The Guarantita Problem)**

Imagine a bustling food stall selling the popular snack, guarantita. There's one skilled cook preparing these delights, with a few stools for eager customers to wait their turn. Customers arrive randomly, hoping to get their hands on this delicious treat. If a stool is free, they sit down and wait patiently. But if all stools are occupied, they sadly move on to find another snack. Once the cook finishes preparing a guarantita, they glance at the waiting area. If someone's patiently waiting, the cook whips up their order next. However, if no one's there, the cook wouldn't want to stand around idly. This is where your creative problem-solving comes in! Can you design a system using semaphores to guarantee fairness and avoid chaos? Ensure neither the cook waits forever nor hungry customers miss their chance to savor a tasty guarantita.