Solutions for Tutorial Session N°2: Events and Locks

Objective: Learn how to control and coordinate multiple threads using Events and Locks, so threads can communicate safely and avoid conflicts when accessing shared resources.

Exercise 1

Two processes, P1 and P2, where P1 has higher priority than P2, share a critical resource. The scheduling rules ensure that P1 is executed as soon as it becomes ready (higher priority first).

While P1 is performing I/O, process P2 enters the critical section. Then, process P1 becomes ready while P2 is still in the critical section. Process P2 is then suspended in favor of process P1. P1 requests to enter the critical section.

1. What happens if we use the TAS instruction?
2. Give a solution using events.

SOLUTION

1- What happens if we use the TAS instruction:

- P2 enters its critical section.

-While P2 is still inside, P1 finishes its I/O and becomes ready.

- The scheduler suspends P2 (because P1 has higher priority) and runs P1.

- P1 tries to enter the critical section — but the lock is held by P2.

- If we use TAS, P1 stay in a busy-waiting while P2 is preempted — so P2 can't ever release the lock.

➞ **Deadlock / priority inversion occurs**.

2- Solution using events

Let E be a binary event:

- ✓ E = signaled (1) → the resource is free.
- ✓ E = non-signaled (0) → the resource is taken.

Operations:

- ✓ wait(E) → if E is 0, the process is blocked/ If E is 1, it consumes the signal and continues (sets E to 0).
- ✓ signal(E) → sets E to 1 and wakes one waiting process (if any).

Initial state: E = 1 → resource is free.

```
Process Pi;
begin
    wait(E);      // blocks if E == 0, continues if E == 1
    CRITICAL SECTION;   // exclusive access to shared resource
    signal(E);    // signals that resource is now free
end;
```

Execution scenario:

1. Initially: E = 1 (free)
2. P2 runs first, executes wait(E) → consumes the event (E = 1), enters critical section.
3. P1 becomes ready → scheduler preempts P2, P1 starts running.
4. P1 calls wait(E) → since E = 0, P1 is blocked by the OS.
1. The scheduler now chooses another ready process — possibly P2.
5. P2 resumes, finishes its critical section, and calls signal(E) → this sets E = 1 and wakes P1.
6. P1 becomes ready, the scheduler runs it → P1 continues past wait(E) and enters the critical section.

Result:

- Only one process is in the critical section at a time.
- No busy waiting.
- No priority inversion: P1 is blocked, so P2 can finish and release the resource.

Exercise 2:

The most common situation of cooperation between processes occurs when data is transmitted from one process to another. Let's consider the simplest case: A keyboard process reads a character typed on a terminal keyboard, and transmits it to a screen process, which displays it on the same terminal.

```
var c : char;   /* shared variable */
```

Process keyboard:	Process screen

begin
char cr;
while (1) {
  read a character from keyboard into cr;
  c = cr; /* instruction 1 */
}
end;
	begin
	char cw;
	while (1) {
	  cw = c; /* instruction 2 */
	  write the character cw to the screen;
	}
	end;

The screen process must not execute instruction 2 before the keyboard process has executed instruction 1 (i.e., before a new character is available). Conversely, the keyboard process must not execute instruction 1 again (produce a new character) before the screen has finished executing instruction 2 for the previous character.

1. Using events, synchronize the two processes
2. Repeat the first question with Locks synchronization mechanism
3. If we have one keyboard and 3 screens, the keyboard process reads a character, and transmits it to the 3 screen processes, which display it at the same time. In this case what is the suitable synchronization mechanism to apply events or locks? Justify your answer.

SOLUTION

1- Solution using events

We use two **binary events**:

Event name	Meaning	Initial state
ScreenReady	The screen is ready to receive a new character (buffer empty)	**signaled (1)**
CharReady	A character has been produced and is ready to display (buffer full)	**non-signaled (0)**

```
process Keyboard
var cl : char;
while (true) do
begin
   wait(ScreenReady);
   read character into cr;
   c := cr;
   signal(CharReady);
end
```

```
process Screen
var ce : char;
while (true) do
begin
   wait(CharReady);
   cw := c;
   write cw on the screen;
   signal(ScreenReady);
end
```

- The **keyboard** waits on ScreenReady, then reads and writes a character, then signals CharReady. So, the keyboard **never overwrites** an unread character.
- The **screen** waits on CharReady, reads the shared variable, displays the character, then signals ScreenReady. So, the screen **never reads** an undefined character.

1- Solution using Locks

We'll use **two locks**:

```
shared var c : char
shared var lock_kb : lock := 0 (unlocked)     /* keyboard starts first */
shared var lock_scr : lock := 1 (locked )    /* screen waits for first character */
```

```
process Keyboard
begin
   var cr : char;
   repeat
      read a character from keyboard into cr;
      /* wait until previous character displayed */
      Lock(lock_kb) ;
      c := cr;          /* produce new character */
```

```
process Screen
begin
   var cw : char;
   repeat
      /* wait for a new character */
      Lock(lock_scr);
      cw := c          /* consume the character */
```

/* allow screen to read and display */ **Unlock(lock_scr);** Until false; End;	write cw to the screen; /* allow keyboard to produce next character */ **Unlock(lock_kb);** Until false; End;

3- events or locks?

Option 1: Locks

- ✓ Locks work well for mutual exclusion between two processes.
- ✓ But with multiple consumers, locks alone are not convenient:
- ✓ Each screen would need its own lock, or we'd need a shared counter.
- ✓ Managing which screens have finished displaying becomes complex.
- ✓ Locks only ensure exclusive access, but they don't notify multiple waiting processes simultaneously.

Option 2: Events

- ✓ Events can broadcast a signal to several waiting processes (the three screens).
- ✓ When the keyboard produces a new character, it can signal (or broadcast) that event once, and all screens waiting on that event will wake up to read the character.
- ✓ This is much simpler and more efficient than coordinating multiple locks.

So ,the suitable synchronization mechanism for this case is events.

Exercise 3

Two processes **P1** and **P2** share an integer variable **x**, which is initially equal to **zero (0)**. Each process increments the variable **x** **five (5) times**. The two processes execute their instructions **concurrently without any synchronization mechanisms**.

1. Explain how it is possible that, even though there are ten increment operations in total, the final value of **x** could be **3**.
2. What are all the possible final values of **x** after both processes have finished executing?
3. Give a solution to this problem when the variable **x** is protected using a **lock synchronization mechanism**.

SOLUTION

1- How to obtain a final value of x = 3

Each process increments **x** five times. Each incrementation consists of three processor instructions: {Load x; add 1; store x;}. A clock interrupt may occur between any two processor instructions. The two processes execute concurrently (in pseudo-parallel), so several execution sequences are possible:

- Process **P** executes the following sequence five times: { (p0): Load x; (p1): add 1; (p2): store x; } We denote the 5 sequences executed by P as: $[(p0)(p1)(p2)]^5$
- Process **Q** executes the following sequence five times: { (q0): Load x; (q1): add 1; (q2): store x; } We denote the 5 sequences executed by Q as: $[(q0)(q1)(q2)]^5$

- Therefore, the sequence $[(p0)(p1)(p2)]^5[(q0)(q1)(q2)]^5$ represents the sequential execution of the two processes, and the final value of variable **x** is 10.

To obtain a final value of **x = 3**, it is enough for clock interrupts to occur at specific points, producing the following sequence: $(p0)(p1)(p2)(p0)[(q0)(q1)(q2)]^4(p1)(p2)(q0)[(p0)(p1)(p2)]^3(q1)(q2)$

2- The possible values of x after both processes have finished are: $x \in \{2,3,4,5,6,7,8,9,10\}$

The different possible values of **x** can be deduced using the same reasoning as in the previous answer.

3- Solution using a Lock Synchronization Mechanism

<table>
<tr><td colspan="2">Shared data:
shared var x : integer := 0;
shared var L : lock := 0 (unlocked) ;</td></tr>
<tr><td>process P1
begin
  for i = 1 to 5 do
  Lock(L); /* enter critical section */
  x := x + 1; /* increment the shared variable */
  Unlock(L); /* exit critical section */
  end for
end</td><td>process P2
begin
  for i = 1 to 5 do
  Lock(L); /* enter critical section */
  x := x + 1; /* increment the shared variable */
  Unlock(L); /* exit critical section */
  end for
end</td></tr>
</table>

The lock guarantees mutual exclusion:

- ✓ only one process can access x at a time.
- ✓ This eliminates interleaving of read/write operations.
- ✓ As a result, all ten increments are correctly executed.
- ✓ Final value of x is always = 10