Operating Systems

Synchronization, Part 2

Semaphores
Semaphores

- Used to control access to a common resource by multiple processes in a concurrent system
  - A record of how many units of a particular resource are available
  - Two operations (up/down) to adjust that record safely

- Semaphores which allow an *arbitrary* resource count are called *counting semaphores*

- *Binary semaphores* (mutex) are restricted to the values 0 and 1
Counting semaphore

**Init(i)**

S = i

**down(S)** [the ‘p’ operation]

If (S ≤ 0)

the process is blocked.
It will resume execution only after it is woken-up

Else

S--

**up(S)** [the ‘v’ operation]

If (there are blocked processes)

wake-up one of them

Else

S++

- Semaphore’s interface doesn’t enforce the implementation of **starvation freedom**.
- All operations are ATOMIC! That is, up(s) is more than just s:=s+1
- There is no way to access the semaphore’s internal value. Any attempt to access it is a mistake.
- Always remember to initialize the semaphore
Negative-valued semaphore

\textbf{down(S)}
\begin{align*}
S & \text{--} \\
\textbf{If} & (S < 0) \\
& \text{the process is blocked.} \\
& \text{It will resume execution only} \\
& \text{after it is woken-up}
\end{align*}

\textbf{up(S)} [the \textbackslash \texttt{`v'} operation]
\begin{align*}
S & \text{++} \\
\textbf{If} & \text{(there are blocked processes)} \\
& \text{wake-up one of them}
\end{align*}

\textbf{NOTE:}
\begin{align*}
\text{If } S \text{ is negative, there are } |S| \text{ blocked processes}
\end{align*}
Counting Semaphores (Barz)

Create a counting semaphore, $S$, by using binary semaphores

Variables?

\[\text{down(S)}\]
\[
\begin{align*}
\text{down(S2);}
\text{down(S1);}
\text{S.value--;}
\text{if (S.value>0) then}
\text{ up(S2);}
\text{ up(S1);}
\end{align*}
\]

\[\text{up(S):}\]
\[
\begin{align*}
\text{down(S1);}
\text{S.value++;}
\text{if (S.value == 1) then}
\text{ up(S2);}
\text{ up(S1);}
\end{align*}
\]
Question 1

Consider the following code snippets run by two processes, P and Q:

Shared memory
n=5;
sqr=0;

Process P
loopP:
  if (n==0)
    goto endP;
  n=n-1;
  goto loopP;
endP:
  print(sqr)

Process Q
loopQ:
  sqr = sqr + 2*n +1;
  goto loopQ;

Add the minimal number of semaphores, initialize them and place them in the code snippets, so the result of the calculation will be correct (5²=25).
Question 1

Note that we can break 25 in more than one way:

• $25 = 11 + 9 + 5 = [(2 \times 5 + 1) + (2 \times 4 + 1) + (2 \times 2 + 1)]$

• $25 = 9 + 7 + 5 + 3 + 1 = [(2 \times 4 + 1) + (2 \times 3 + 1) + (2 \times 2 + 1) + (2 \times 1 + 1) + (0 + 1)]$

**Shared memory**
- \( n = 5 \);
- \( \text{sqr} = 0 \);
- \( \text{semA} = 1 \);
- \( \text{semB} = 0 \);

**Process P**

```plaintext
loopP:
  if (n == 0)
    goto endP;
  down(semA);
  n = n - 1;
  up(semB);
  goto loopP;
endP:
  down(semA);
  print(sqr)
```

**Process Q**

```plaintext
loopQ:
  down(semB);
  sqr = sqr + 2 * n + 1;
  up(semA);
  goto loopQ;
```
Question 2 (2007 Moed A)

The following constraint graph is a DAG that defines a partial order over code lines. Each vertex is associated with a single line of code in a possible program. A directed edge $e(u,v)$ is used to represent the precedence constraint: code line $u$ must be completed prior to the beginning of code line $v$.

For example, code line $S_1$ should be completed before $S_2$, $S_5$ or $S_8$ are executed, while $S_6$ can only be executed after both $S_2$ and $S_5$ were completed.
Question 2a

The following code is executed by two processes, A and B.

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>S5</td>
<td>S3</td>
</tr>
<tr>
<td>S8</td>
<td>S6</td>
</tr>
<tr>
<td>S9</td>
<td>S4</td>
</tr>
<tr>
<td>S7</td>
<td></td>
</tr>
</tbody>
</table>
Question 2a

Use two counting semaphores with properly initialized values so that in every execution of A and B the execution order of code lines will be consistent with the constraint graph defined above. That is, add up and down operations within A and B’s code lines so that no precedence constraint is violated. You may assume that in every execution both A and B run their code only once.

Note: Partial scoring will be given to solutions using more than two semaphores.
Question 2a

Semaphores: semA=0, semB=0

Process A
S1
SemB.up
S5
SemB.up
S8
S9
SemA.down
S7
SemB.up

Process B
SemB.down
S2
S3
SemB.down
S6
SemA.up
SemB.down
S4
Question 2b

Give a concise but accurate explanation why a single semaphore is insufficient for maintaining consistency of the above constraint graph and the two processes.
Question 2b

A single semaphore is insufficient because there are constraints which require that A waits until B completes one of its lines and vice versa. In such a state, having A signaling B and immediately waiting for it on the same semaphore will result in either a deadlock or a breaking of one of its constraints.
Question 3 – Producer Consumer Problem

```c
#define N 100        /* Buffer size */
semaphore mutex = 1; /* access control to critical section */
semaphore empty = N; /* counts empty buffer slots */
semaphore full = 0;  /* full slots */

void producer(void) {
    int item;
    while(TRUE) {
        produce_item(&item);        /* generate something... */
        down(&empty);                /* decrement count of empty */
        down(&mutex);                /* enter critical section */
        enter_item(item);            /* insert into buffer */
        up(&mutex);                  /* leave critical section */
        up(&full);                   /* increment count of full slots */
    }
}
```
Question 3 – Producer Consumer Problem

```c
void consumer(void){
    int item;

    while(TRUE){
        down(&full);       /* decrement count of full */
        down(&mutex);      /* enter critical section */
        remove_item(&item); /* take item from buffer */
        up(&mutex);        /* leave critical section */
        up(&empty);        /* update count of empty */
        consume_item(item); /* do something... */
    }
}
```
Question 3

1. The red lines of the following code were swapped. How will this affect the algorithm?

```c
void producer(void) {
    int item;
    while(TRUE) {
        produce_item(&item); /* generate something... */
        down(&empty); /* decrement count of empty */
        down(&mutex); /* enter critical section */
        up(&mutex); /* leave critical section */
        enter_item(item); /* insert into buffer */
        up(&full); /* increment count of full slots */
    }
}
```

No mutual exclusion!
2. What will happen now?

```c
void producer(void) {
    int item;
    while(TRUE) {
        produce_item(&item);  /* generate something... */
        down(&empty);         /* decrement count of empty */
        down(&mutex);         /* enter critical section */
        enter_item(item);     /* insert into buffer */
        up(&full);            /* increment count of full slots */
        up(&mutex);           /* leave critical section */
    }
}
```

No problems...
Question 3

3. And now?

```c
void consumer(void){
    int item;

    while(TRUE){
        down(&mutex); /* enter critical section */
        down(&full); /* decrement count of full */
        remove_item(&item); /* take item from buffer */
        up(&mutex); /* leave critical section */
        up(&empty); /* update count of empty */
        consume_item(item); /* do something... */
    }
}
```

*Deadlock!*
An *unfair semaphore* is a semaphore which does not guarantee that the wakeup order of processes is similar to their falling asleep order. It does, however, provide the following simple guarantee: if there are sleeping processes on the semaphore while an *up* operation is invoked, then one of these processes will be woken up (not necessarily the first amongst the waiting processes to do a *down*).
Question 4

Now you are required to implement a starvation free mutual exclusion algorithm for three processes using 3 unfair counting semaphores: R, S and T, initialized to 1.
You are not allowed to use any other variable but these semaphores.
Add your code and complete the entry and exit section of each process.

Briefly explain why your code satisfies both mutual exclusion and starvation freedom.
Question 4

**unfair counting semaphores**: R, S and T initialized to 1

<table>
<thead>
<tr>
<th>P1’s code:</th>
<th>P2’s code:</th>
<th>P3’s code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry:</td>
<td>entry:</td>
<td>entry:</td>
</tr>
<tr>
<td>Critical_section()</td>
<td>Critical_section()</td>
<td>Critical_section()</td>
</tr>
<tr>
<td>exit:</td>
<td>exit:</td>
<td>exit:</td>
</tr>
</tbody>
</table>
**Question 4**

**unfair counting semaphores: R, S and T initialized to 1**

<table>
<thead>
<tr>
<th>Code</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
</table>
| **P1’s code:** | \(\text{down (S)}\)
\(\text{down (R)}\)

\(\text{Critical \_ section()}\)

\(\text{up(R)}\)
\(\text{up(S)}\)

<table>
<thead>
<tr>
<th>Code</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
</table>
| **P2’s code:** | \(\text{down (R)}\)
\(\text{down (T)}\)

\(\text{Critical \_ section()}\)

\(\text{up(T)}\)
\(\text{up(R)}\)

<table>
<thead>
<tr>
<th>Code</th>
<th>Entry</th>
<th>Exit</th>
</tr>
</thead>
</table>
| **P3’s code:** | \(\text{down (S)}\)
\(\text{down (T)}\)

\(\text{Critical \_ section()}\)

\(\text{up(T)}\)
\(\text{up(S)}\)
Question 4

**Mutual exclusion:**

Any process wishing to enter its critical section must successfully complete two ‘down’ operations on two distinct semaphores. Since any process competes over one different “successful down” with each of the other processes, only a single process may successfully enter the critical section at any given moment.
Question 4

**Starvation freedom:**

We first note that there is no starvation problem when using an unfair semaphore with 2 processes (convince yourselves!). Since entrance to the critical section requires passing semaphores which are only shared in pairs no starvation problems will occur.
Will this solution work?

**Deadlock!**
Question 5 (Midterm 2009)

In this question, you are required to implement an event counter using binary semaphores and registers (variables that support reads and writes only). As you learned,

the event counter \( E \) contains a whole number and starts at 0. It supports three atomic operations:

- **Advance** \((E)\): advances the value of \( E \) by 1 from \( v \) to \( v+1 \), where \( v \) is the value of \( E \) before the operation.

- **Await** \((E, v)\): the reader process waits until the value of \( E \) reaches \( v \). If the value of \( E \) is greater than or equal to \( v \) at the time of the operation, the reader process continues running.

- **Read** \((E)\): returns the current value of \( E \).

The event counter \( E \) is implemented using binary semaphores (binary semaphores are assumed to be fair).

It is assumed that there are \( N \) processes using \( E \) and their identifiers are \( 0, 1, \ldots, N-1 \). Each process has a variable \( MyID \) that stores its identifier.
Shared variables

Advance(E)

Await(E,v)

Read(E)
Question 5b

למדתם כי אין זה פשוט לממש סמאפורי כללי (counting semaphore) משמש סמאפורי בינארי. מסתבר כי גם הכוון ההפוך לא פשוט. להלן קטע קוד המכיל מימוש של סמאפורי בינארי וסמפורי כללי (counting semaphores) ומרגסיוס.

```
Counting semaphore mutex=1
Counting semaphore b=0
register v=0, register waiting=0

procedure down( ) {
    mutex.down( )
    if (v == 1){
        v=0
        mutex.up( )}
    else {
        waiting=waiting+1
        mutex.up( )
        b.down( )
    }
}

procedure up( ){
    mutex.down( )
    if (waiting > 0){
        waiting=waiting-1
        b.up( )}
    else if (v == 0)
        v=1
    mutex.up( )
}  ```

ניתן להניח כי הסמאפורים הכלליים בהם משתמש הקוד שלהם госעים לקסם מדוע קטע הקוד שלמה מקווניםjni ממומש סמאפור בינארי בינארית נקודת.

א- הסברו במדוייק מהי הסמנטייקה של סמאפור ינארית.

1..nilطلق תוחכם, 0,1벌בד
2. פעולות.wsop: אם ערך הסמאפור הוא 0 אזי תוסמ את התהליך, אחרית משנה את ערך הסמאפור ל-0.
3. פעולות.cn: אם ישנו תהליך שמתמיין- מעיר את הסמאפור ל-1.
Process p1 does down and stops prior to b.down
Process p2 does down and stops prior to b.down
Process p3 does up and finishes the operation (b == 1)
Process p4 does up and finishes the operation (b == 2)
Process p5 does down and finishes the operation (b == 1)
Process p6 does down and finishes the operation (b == 0)