IV. Concurrent Processes - Semaphores, Classic Problems

Note: As for the previous "concurrency" material, some (much?) of the material presented below will be revision of last year's work. Numbering follows the text book.

7.4 SEMAPHORES

Because our solutions (in previous sections) are not easily to generalise to more complex problems, we need some other "synchronization" tools. ***Semaphores*** are such tools and are described as follows:

- A semaphore "S" is an integer variable that can be accessed in just 3 ways
- Way 1: "S" can be initialized to some value.
- Way 2: "S" can be accessed by the standard atomic operation ***wait***, defined as

```
wait(S) {
        while (S ≤ 0)
                ; /* no op */
        S--;
}
```

- Way 3: "S" can be accessed by the standard atomic operation ***signal***, [*sometimes **send** is used*] defined as

```
signal(S) {
        S++;
}
```

N.B.: Modifications to "S" must be executed indivisibly (atomically) - no other process can modify "S" at the same time. Similarly, the test " S ≤ 0" in wait(S) must be executed without modification of "S" by any other process.

Solution of the n-process critical-section problem using semaphores:

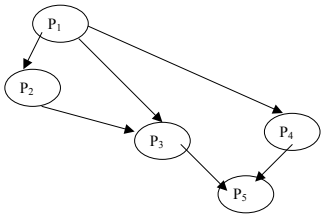
Let "mutex" be a semaphore that is shared by the "n" processes and that is ***initialized to 1***. Then the solution of the problem simply requires organising each P_i as follows:

do{	
wait(mutex);	P_i can enter its critical section if "mutex > 0"
critical section	
signal(mutex);	
remainder section	
}while(1);	

Here is a simple illustration of this algorithm:

C = critical region = stretch of rail track that can have one train at a time
Time-line of events:
set semaphore: mutex=1
Train T_1 arrives: wait(mutex) executed => mutex=0 and T_1 enters C
Train T_2 arrives: wait(mutex) executed => T_2 is blocked => Q(ueue) = T_2
Train T_3 arrives: wait(mutex) executed => T_3 is blocked => Q = T_3T_2
Train T_1 leaves C: signal(mutex) executed => T_2 (*say*) is freed & enters C => Q = T_3
Train T_2 leaves C: signal(mutex) executed => T_3 is freed & enters C => Q = empty
Train T_4 arrives: wait(mutex) executed => T_4 is blocked => Q = T_4
Train T_3 leaves C: signal(mutex) executed => T_4 is freed => Q = empty
Train T_4 leaves C: signal(mutex) executed => mutex=1

Application of semaphores to implementation of precedence graphs:
Suppose we have a precedence graph such as that below and we want a suitable structure to implement the precedence relationships within it.



This can solved rather simply through the use of semaphores.

For this type of problem, it is convenient to introduce a slightly different notation for process structure than the "entry section - critical section - exit section - remainder section" used for our other examples. Our task is to extend the given processes by adding suitable "wait" and "signal" instructions. If P_j is a given process then we let P_j' denote P_j after it has been extended by suitable "waits" and "signals".

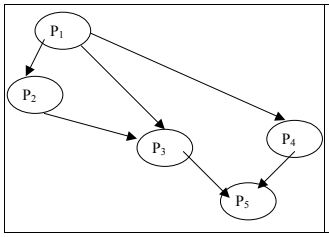
It turns out that there are some simple rules that guide us to the required solution. These rules are:

(1) Introduce a semaphore s_j corresponding to each non-initial process P_j.

(2) Let the initial value of s_j be defined by $-(d_j - 1)$ where d_j is the number of immediate predecessors of P_j

(3) To form P_j' prefix non-initial P_j by wait(s_j) and suffix each non-terminal P_j with a signal(s_k) for each immediate successor P_k of P_j

Applying these rules to the precedence graph above, we have:



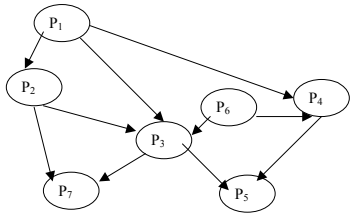
P_1': P_1, signal(s_2), signal(s_3), signal(s_4)
P_2': wait(s_2), P_2, signal(s_3)
P_3': wait(s_3), P_3, signal(s_5)
P_4': wait(s_4), P_4, signal(s_5)
P_5': wait(s_5), P_5

with initial semaphore values s_2=0, s_3=-1, s_4=0, s_5=-1.

Then, a typical execution sequence, tracking semaphore values, is

Initial	After P_1'	After P_2'	After P_3'	After P_4'	After P_5'
s_2=0	1	0	0	0	0
s_3=-1	0	1	0	0	0
s_4=0	1	1	1	0	0
s_5=-1	-1	-1	0	1	0

Question 1: What other execution sequences are possible?
Question 2: Apply the same method to implement the following precedence graph and illustrate your result with typical execution sequence(s):



[7.4.2] Aspects of Implementation of semaphores (very briefly!)

Busy waiting: The solutions so far presented, and our semaphore definition, all involve ***busy waiting***. For example,

Critical section problem (2 processes): "***while (flag[j] && turn == j);***"

Critical section problem (TestAndSet): " ***while (waiting[i] && key) key = TestAndSet(lock));***"

Semaphore "wait(S)" pseudo-code definition: "***while (S ≤ 0) ; /* no op */***"

The problem is that while a process is in its critical section, any other process that tries to enter its own critical section must loop continuously in the entry code.

- Thus, ***busy waiting*** "wastes" CPU cycles that some other process could perhaps use productively.
- On the other hand, this type of semaphore (*called a* ***spinlock*** *as it spins while awaiting a lock*) has the advantage that no context switch - which could take considerable time - is required. So, spinlocks can be useful where locks are expected to be held for just a short time.

Notes:
(1) The semaphore definition can be modified to avoid (most of the) busy waiting. The definition is more complex, involving association of a waiting queue with the semaphore.
(2) Deadlocks can occur when a semaphore is implemented with a waiting queue.
We will ***not*** go into these points in this module.

7.5 SOME CLASSIC PROBLEMS OF SYNCHRONISATION (via SEMAPHORES)

Two general guidelines on using semaphores:
(1) In order to **protect** a critical section of code, **matching waits and signals** are placed around it. For example, we had

```
do{
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
}while(1);
```

(2) When **communicating** between processes, **the waits and signals are placed in opposite** processes. (The following ("bounded-buffer" problem) illustrates this guideline).

7.5.1 THE BOUNDED-BUFFER PROBLEM:
A more general statement of our "producer-consumer" problem. We have

- A pool of "n" buffers (storage locations) each of which can store 1 item
- Semaphore ***mutex*** to provide mutually exclusive access to the buffer pool. Its initial value is "1".
- Semaphores ***empty*** (initialised to n) and ***full*** (initialised to 0) to count the number of empty and full buffers, respectively.

```
do{...                          do{...
    <Produce an item in "nextp">    wait(full);
    ...                             wait(mutex);
    wait(empty);                    ...
    wait(mutex);                    <remove an item to "nextc">
    ...                             ...
    <add "nextp" to buffer>         signal(mutex);
    ...                             signal(empty);
    signal(mutex);                  ...
    signal(full);                   <consume the item in "nextc">
}while(1);                          ...
                                }while(1);
```

Producer - *producing full buffers for the consumer*	Consumer - *producing empty buffers for the producer*

7.5.2 THE READERS-WRITERS PROBLEM:
(1) Several concurrent processes wish to access a common file (or similar data object)
(2) Some wish to read; some wish to write
(3) Shared read accesses are required but exclusive access is required for writers

We identify two cases here:

(A) ***First readers-writers problem***: **Readers have priority**
"No reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish just because a writer is waiting".
POTENTIAL PROBLEM: Writers may "starve".

(B) ***Second readers-writers problem***: **Writers have priority**
"Once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new reader may start reading".
POTENTIAL PROBLEM: Readers may "starve".

NOTE: Because of the potential "starvation" problems noted, other variants have been produced. We focus on the first problem only.

"Readers have priority" problem: We introduce a shared variable ***readcount*** to keep track of how many processes are currently reading the object. Essentially, we need 2 shared semaphores, one (***mutex***) to protect ***readcount*** and one (***wrt***) to give exclusive access to writers. We have

```
wait(mutex);                [
readcount++;                Protecting
if(readcount == 1)          readcount
    wait(wrt);
signal(mutex);              ]
...
<reading is performed>
...
wait(mutex);                [
readcount--;                Protecting
if(readcount == 0)          readcount
    signal(wrt);
signal(mutex);              ]
```

Structure of a Reader process

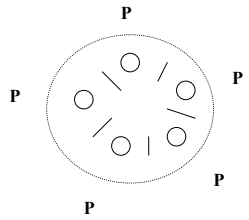
```
wait(wrt);
    ...
    <writing is performed>
    ...
signal(wrt);
```

Structure of a Writer process

(i) If a writer is in critical section & n readers are waiting, then one reader is queued on ***wrt*** and the rest on ***mutex***.
(ii) When a writer executes signal(wrt) then it is up to the CPU scheduler to resume running all waiting readers or 1 waiting writer.

Note: ***mutex*** and ***wrt*** are both initialised to "1"; ***readcount*** to "0".

7.5.3 THE DINING-PHILOSOPHERS PROBLEM:
(1) Five philosophers sitting at table.
(2) In front of each philosopher there is a plate of rice (5 plates)
(3) Between each pair of plates there is a chopstick that can be used by either corresponding philosopher. (5 chopsticks)
(4) A philosopher requires two chopsticks to eat.



(5) While thinking, a philosopher does not interact with the others.
(6) If hungry, a philosopher tries to pick up the 2 closest chopsticks (left & right). This is not possible if one or both neighbours is already holding a chopstick.
(7) If successful in picking up both chopsticks, a hungry philosopher commences eating and continues eating until finished. When finished, and not before, the philosopher puts down (releases) both chopsticks, and starts thinking again.

This problem is representative of concurrency-control problems where several resources must be allocated among several processes in such a way that neither ***deadlock*** nor ***starvation*** will occur.

We present one solution using semaphores (which, however, is not satisfactory in that it leads to deadlock). In the text book (p219) a solution is presented using a higher-level mechanism called ***monitors***. However, we will not have time to study this in any detail.

Solution attempt: **Structure for philosopher i**

do{	
wait(chopstick[i]);	Philosopher i tries to get chopstick i
wait(chopstick[(i+1)%5];	Philosopher i tries to get chopstick "i+1"
...	
<eat>	
...	
signal(chopstick[i]);	Philosopher i releases chopstick i
signal(chopstick[(i+1)%5];	Philosopher i releases chopstick "i+1"
...	
<think>	
...	
}while(1);	

Notes:
(a) Shared data are ***semaphore chopstick[5]*** all of whose elements are initialised to 1.
(b) This solution guarantees that neighbouring philosophers are not eating simultaneously.
(c) ***Deadlock*** can occur as follows: all 5 philosophers become hungry at the same time and all pick up their left chopstick. Thus all the elements of ***chopstick*** are now zero. Hence, all 5 philosophers will wait indefinitely for their right chopstick to become free.

There are remedies to this deadlock problem (including allowing a philosopher to pick up a chopstick only if both required chopsticks are free). It must be ensured, however, that ***starvation*** won't occur that is that a hungry philosopher is caused to wait indefinitely. Also, it is desirable that a solution be ***efficient*** - with 5 chopsticks it should be possible to have two philosophers eating at the same time.