COS214 Tutorial 10

Roberto Togneri, 2000

SOLUTIONS

1. The UNIX *ln* function is used to create a link to a file. Show how it can be used as an effective mechanism for mutual exclusive access to a file if there are multiple processes attempting to access a file. Are there are any problems with this solution?

Answer

Create a lock file as follows: *ln -s file file.lock*

(You can also use *touch file.lock*, but the *ln -s* allows the *file.lock* to be symbolically linked with *file*. A symbolic link is preferrable to a hard link for various reasons)

Each processes checks for and then attempts to create the lock file:

```
#!/bin/sh
…
while [ -f file.lock ]; do
   sleep 60
done
ln -s file file.lock
… {critical-section} …
rm file.lock
…
```

Problems?

1. If processes simultaneously execute the *while [-f file.lock]; do ...* test, all processes will be allowed through (race condition)! Fix? Test by attempting to run *ln -s file file.lock* and loop if an error condition that "*file already exists*" is returned.
2. If a process dies while in the critical section the file.lock will not be removed (deadlock).

2. The readers and writers problem can be formulated in several ways with regard to which category of processes can be started and when. Carefully describe three different variations of the problem, each one favoring (or not favoring) some category of processes. For each variation, specify what happens when a reader or a writer becomes ready to access the database, and what happens when a process is finished using the database. Carefully analyse the monitor solution presented in lectures, which variation does it represent?

Answer

Readers have Priority

No writer may start when a reader is active. When a new reader appears, it may start immediately unless a writer is currently active. When a writer finishes, if readers are waiting, they are all started, regardless of the presence of waiting writers. Problem? Writers may starve.

Writers have Priority

No reader may start when a writer is waiting. When the last active reader or writer process finishes, a writer is started, if there is one, otherwise, all the readers (if any) are started. Problem? Readers may starve.

Symmetric Priority

When a reader is active, new readers may start immediately. When a writer finishes, a new writer has priority, if one is waiting. In other words, once we have started reading, we keep reading until there are no readers left. Similarly, once we have started writing, all pending writers are allowed to run. Problem? Readers or writers may starve!

Monitor Solution

Implements a modified writers have priority, where when a writer finishes it allows any readers currently waiting to go through in preference to waiting writers. Problem? No starvation apparent since a current active writer defers to waiting readers and the arriving readers defer to waiting writers.

3. *The Cigarette-Smokers Problem*. Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signalling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Complete the following program fragment to synchronize the agent and the smokers:

```
semaphore a[3] = 0;        /* a[0] for tobacco, a[1] for paper, a[2] for matches */
semaphore agent = 1;

Agent(void) {
   int   i,j;
   repeat
      i = 3 * drand48( );   /* returns a random integer 0, 1 or 2 for i */
      j = 3 * drand48( );   /* returns a random integer 0, 1 or 2 for j */
     while (i != j) {       /* i and j must be different */
           the rest of it
     }
until false;
}

Smoker(int r) {             /* r indicates which ingredients this smoker has */
   repeat
      the rest of it
   until false;
}
```

Answer

The shared data structures are:

```
semaphore a[2] = 0;      /* a[0] for tobacco, a[1] for paper, a[2] for matches, all initialised to 0 */
semaphore agent = 1;     /* initialised to 1 */
```

The agent process code is as follows:

```
Agent (void) {
   int   i,j,k;
   repeat
          /* i, j  represent the two materials the agent places on the table) */
          i = 3 * drand48( );   j = 3 * drand48( );
          while (i != j) {
                 down(&agent);
                 k = 3 - (i+j);            /* the smoker with the kth ingredient is identified */
                 up(&a[k]);
          }
   until false;
}
```

Each smoker process needs the ingredient represented by the integer *r*.

```
Smoker(int r) {
   repeat
          down(&a[r]);
          smoke( );
          up(&agent);
   until false;
}
```

4. (a) Consider a system consisting of a number of processes attempting to access three identical line printers. Write a monitor that allocates line printers to these processes.

   (b) Parallel execution of list operations (e.g. max( ), min( ), sort( )) is trivially accomplished using a master-slave paradigm. Consider the case of one master and two slave processes. The master takes the list and divides it into two halves and passes one half to slave[0] and the other half to slave[1]. The slave processes then concurrently perform the list operation on their local list (e.g. find the local maximum for that half of the list) and return their result to the master (e.g. the local maximum). The master then merges the result from each slave to derive the global result (e.g. the maximum of each local maximum returned is the global maximum of the whole list). Describe the synchronisation between the master and the two slaves using message passing.

Answer

(a)

```
monitor Printers
  boolean P[0..2];          /* P[i] is true if printer i is busy */
  condition busy;

  procedure acquire (var printer-id: integer);
  begin
    if P[0] = true and P[1] = true and P[2] = true then wait(busy)
    if P[0] = false then printer-id := 0;
    else if P[1] = false then printer-id := 1;
    else printer-id := 2;
     P[printer-id]:= true;
end

  procedure release (printer-id: integer)
  begin
    P[printer-id]:= false;
    signal(busy);
  end

  P[0] := P[1] := P[2] := false;
end monitor
```

(b)

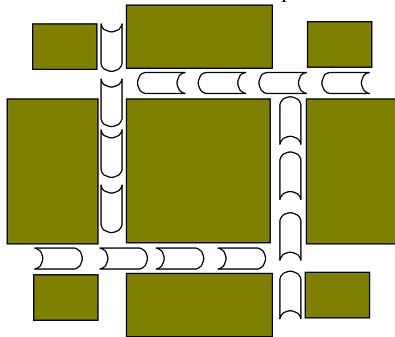
Master process:

```
…
send(slave[0], first half of list);
send(slave[1], second half of list);
for (i=0; i < 2; i++) {
   receive(any_slave, &result);
  /* merge result */
}
/* return global result */
```

Slave process *i* (for *i*=0, 1):

```
…
receive(master, half of the list);
/* derive local result */
send(master, result);
…
```

5. Consider the traffic deadlock depicted below:



(a) Show that the four necessary conditions for deadlock indeed hold in this example.
(b) State a simple rule that will avoid deadlocks in this system.

Answer

(a) Each section of the street is considered a resource.
Mutual-exclusion —only one vehicle allowed through each intersection.
Hold-and-wait — each queue of vehicles is occupying (holding) an intersection and the vehicle at the head of the queue is waiting to cross (acquire) the next intersection.
No-preemption — an intersection that is occupied by a vehicle cannot be taken away from the vehicle unless the car is able to move.
Circular-wait — each queue of vehicles is waiting to access the next intersection and the streets cross one another in a circular route.

(b) Allow a vehicle to cross an intersection only if it is assured that the vehicle will not need to wait whilst in the intersection (i.e. block the intersection). Not surprisingly, blocking an intersection is illegal even if you have the green light!

6. In an electronic funds transfer system, there are hundreds of identical processes that work as follows. Each process reads an input line specifying an amount of money, the account to be credited, and the account to be debited. Then it locks both accounts and transfers the money, releasing the locks when done. With many processes running in parallel, there is a very real danger that having locked account **x** it will be unable to lock **y** because **y** has been locked by a process waiting for **x**. Devise a scheme that avoids deadlocks. Do not release an account record until you have completed the transactions (e.g. don't lock one account and then release it immediately if the other account is found to be locked). Why is this last condition important?

Answer

To avoid a circular wait, use the account numbers as follows: After reading an input line, a process locks the lower-numbered account first, then when it gets the lock (which may entail waiting), it locks the other one. Since no process ever waits for an account lower than what it already has locked, there is never a circular wait, and hence no deadlock.
Releasing the lock on one account before the transaction is finished will lead to a race condition.