Name: _____ Student ID# _____ Total: [ ]

This test is made of 5 questions and is worth at most 100 marks.

March 2, 2009

1 Question worth 25%

Consider the following code:

```
global: int x = 0 ;

... ... ...

void handler( int S )
{
   x = x + S ;    // Not atomic
   printf( "%d\n" , x ) ;
   x = x + 100 ;  // Not atomic
   return ;
}
```

This is the code of a signal handler that was installed by a suitable set of **sigaction** calls for signals: **SIGSEGV** (11), **SIGFPE** (8) and **SIGALRM** (14). The handler is reentrant (new signals are handled while the handler is executing) and is installed permanently.

It will print something every time it is executed. Mark with a $\mathcal{Y}$ the boxes corresponding to results that are not impossible as part of output (there may be more than one line of output from this handler) and with a $\mathcal{N}$ those that cannot possibly occur. If you are not sure, leave the box blank. No explanation is needed; you will receive 1–5 marks for a correct answer, 0 marks for a blank answer and -1 for an incorrect answer.

All the answers are possible.

Example: 8 + 8 + 14 = 30 (the first two interrupted right before the **printf**).

2 Question worth 25%

An application requires 2 distinct classes of operations, **READ** and **SEEK**. Many processes can execute **READ** or **SEEK** operations concurrently, but executing a **READ** operation concurrently with a **SEEK** operation must not be allowed (this is a variation on the readers–writers problem).

A programmer proposes a standard **entry** and **exit** code that must be placed around every critical section in which **READ** and/or **SEEK** operations are performed.

This code was used in the following two simple processes (but it must work for any processes performing **READ** or **SEEK**).

Shared variables accessible to all processes:

```
semaphore mutex = 1 , lock = 1 ;
global int Acount = 0 , Bcount = 0 ;
```

```
ProcessA() {
A.0:  while( Bcount > 0 ) ; // spin
A.1:  P(mutex) ;
A.2:   Acount++ ;
A.3:   if( Acount == 1 ) P(lock) ;
A.4:  V(mutex) ;
A.5:  READ filename arguments
A.6:  P(mutex) ;
A.7:   Acount– – ;
A.8:   if( Acount == 0 ) V(lock) ;
A.9:  V(mutex) ;
}
```

```
ProcessB() {
B.0:  while( Acount > 0 ) ; // spin
B.1:  P(mutex) ;
B.2:   Bcount++ ;
B.3:   if( Bcount == 1 ) P(lock) ;
B.4:  V(mutex) ;
B.5:  SEEK filename arguments
B.6:  P(mutex) ;
B.7:   Bcount– – ;
B.8:  if( Bcount == 0 ) V(lock) ;
B.9:  V(mutex) ;
}
```

The **READ** and **SEEK** operations were tested thoroughly; they work correctly when protected by mutual exclusion. **Note: do not ask what they do; it makes no difference.**

Your objective is to assess the quality of the synchronisation scheme. It must allow an arbitrary number of processes ProcessA and ProcessB to execute concurrently. No other processes have access to the shared variables.

1. Is deadlock possible? Explain your answer.
   Deadlock occurs when the next statements to be executed are: A.3 and B.6.

2. Is mutual exclusion assured? Explain your answer.
   It is assured by the semaphores: if SEEK processes are in, the first READ process will block in A.3; all the subsequent ones will block in A.1.
   Alas, all the SEEK processes will also block at this point, which is not desired.

3. Is starvation possible? Explain your answer.
   Starvation is possible because the condition in B.0 may remain true forever (with new READ processes arriving often enough) Obviously, deadlock will prevent starvation (as formally defined).

4. Rewrite the code of **ProcessA** to make it perfect.
   See readers–writers in any textbook.

3 Question worth 30%

In one or two sentences indicate whether the statements below are true or false and why.

1. I/O devices indicate the completion of an I/O operation by interrupting the process that requested that operation.
   Nonsense: a process cannot be interrupted; moreover, I/O devices never heard of processes (they deal with the OS only).

2. **fork()** can fail and no child process is created.
   It can (Process Table full, etc.). Returns -1 (NOT 0!!!!).

3. An orphaned **child process** becomes a zombie by terminating.
   This is true, but does not last long (see init). (A process always has a parent, so "orphaned" must mean "the original parent is gone").

4. A process can guarantee mutual exclusion by **disabling interrupts**.
   One a single–processor system: YES (if they let you). On a multi–processor system: NO.

5. It is possible that an interrupt sent by an I/O device will cause a change in the state of the **running** process from running to blocked.
   The correct answer is YES (with an explanation). NO with an explanation is also correct.
   An example of a reasonable NO answer:

   > A process can be blocked only if a required resource (physical or logical) is not available to this process. But an I/O device does not know about processes, so its iterrupt cannot affect the process directly. It might happen that the interrupt will make the OS decide that a process should be blocked but that would happen at a later moment.

6. The scheduler takes the CPU away from a process by sending it the **SIGSTOP** signal.
   Nonsense! The Scheduler does not take away, it gives. Moreover, SIGSTOP blocks a process, clearly not a desired outcome.

7. A process that just became running can tell how long it was in the ready state provided that it released the CPU voluntarily by issuing a *sleep(0)* system call (it is assumed that the process saved the current time just before issuing the call).
   As long as we do not want a value that is very exact, the answer is YES. If an exact value is needed, the answer is NO, atleast because the process may lose the CPU between the moment it recorded the current time and the moment it issued the *sleep(0)* call.

8. A ready process can get the CPU by sending a wakeup signal to itself.
   Firstly, a ready process cannot send anything. Moreover, it would just make the process even more ready and nothing more.

9. A blocked process may remain blocked forever even though no other process is waiting for it.
   Obviously, a "D" state process falls in this category.

10. A running process can lose the CPU when another process issues an I/O request.
    How can the other process do it without having the CPU? It could on old machines like the IBM/360 (etc.).

4 Question worth 15%

Explain what can cause the following transitions of the state of a process:

1. running→ready
   Scheduler did it!

2. running→running
   A short interrupt jumped in. The scheduler did not get invoked or did not see fit to exercise its powers.

3. blocked→running
   Cannot be: must become ready first.

4. blocked→terminated
   A SIGKILL (not SIGTERM) will do it.

5. ready→blocked
   A SIGSTOP signal will do it. A process cannot do it to itself, though.

5 Question worth up to ∞ marks

```
// Global variable shared by all three processes
int x = 0 ;
```

```
void ProcessA()
{
    if( x == 0 )
        x = 10 ;
    else
        x = 0 ;
}
```

```
void ProcessB()
{
int i ;
    i = x ;
    i = x + i + 2 ;
    x = i ;
}
```

```
void ProcessC()
{
int i ;
    i = x + 1 ;
    x = i ;
    printf( "   %d\n" , x ) ;
}
```

Three processes are started at the same time in an unknown order: one copy of ProcessA, one copy of ProcessB , and one copy of ProcessC. **No other activity modifying x ever takes place. List all the possible outputs coming from this group of processes.** No explanation is necessary; you will be awarded -1 for each incorrect value and positive marks for each correct value (the marks will be: 1 for each of the first 3 values you list, 2 for each of the next 3 values you list, 3 for each of ... , etc.).

0, 1, 2, 3 , 4, 10 , 11, 12, 13, 22, 23, 24