Name:
Login (UPI):

COMPSCI340SC & SOFTENG370SC 2007
Operating Systems Test

Tuesday 21st August, 6:30pm – 7:30pm

- Answer all questions in the spaces provided.
- The test is out of 70 marks. Please allocate your time accordingly.
- Make sure your name is on every piece of paper that you hand in.
- When you are asked to explain something or to give reasons for something you can give your answers as a series of points. Be brief.
- The last page of the booklet may be removed and used for working.

Name:
Login (UPI):

For markers only:

Question 1	*/15*	
Question 2	*/12*	
Question 3	*/8*	
Question 4	*/6*	
Question 5	*/11*	
Question 6	*/18*	
	Total	

Name:
Login (UPI):

Question 1 – History and development of Operating Systems (15 marks)

a) SPOOLing stood for Simultaneous Peripheral Operation On-Line. Describe how this was different from off-line peripheral operation.

In off-line operation there were separate computers to do the slow IO. The slow devices were not connected to the main computer – hence off-line. With spooling the devices were attached to the main computer and it dealt with them using interrupts, transferring data to or from the slow devices and the disk which acted as a buffer. This had the advantage of not requiring extra computers to deal with the slow devices and yet did not noticeably slow the computations of the main computer as it did not poll for IO. Then when a program needed data from a slow device it was already buffered on the disk.

4 marks

b) Computer systems now provide memory protection. What could go wrong if there was no memory protection of the operating system code?

The operating system code could be changed by any running process. This would give all processes complete power over the system. Of course it also allows a process to crash the operating system by overwriting parts of it.

2 marks

c) Computers provide a way for a process to switch from running in user mode to kernel mode and to return from kernel mode to user mode. Do both of these operations have be privileged? Explain why or why not.

Not necessarily. Changing from user to kernel mode must be executable by any process, this is sometimes seen as privileged because we need the instruction to cause a jump to an address the user cannot modify.

Returning from kernel to user mode doesn't have to be privileged if all it does is change the mode to user and jump to an address on the stack. However if the instruction gets the mode from an operand (or the stack) then it must be privileged to stop the user changing to kernel mode with the same instruction.

3 marks

Name:
Login (UPI):

d) Early microcomputer or personal computer operating systems did not require user authentication and security was non-existent. Explain why and then describe the changes that occurred in the use of these systems which required security to be taken seriously.

They were single user systems. The only information that could be tampered with belonged to the user.

One change was implementing multi-user systems on PCs; another was networking. As soon as the information for several people could be accessed by the system, authentication had to be provided and used as the basis for security.

This was made even more essential with anonymous networks, such as the Internet.

3 marks

e) Virtual Machines must have the characteristics of fidelity, performance and safety. Briefly describe each of these.

Fidelity – software should run identically (except for speed) on the Virtual Machine as on the real machine.

Performance – most instructions in a VM must run directly on the hardware (therefore at the same speed as on the real machine).

Safety – the Virtual Machine Monitor (VMM) must be safe from any actions of the VM. Also one VM must be safe from any of the actions of another VM.

3 marks

Question 2 – Processes (12 marks)

a) The following Ruby program is run on my computer:

```
3.times do
  if fork.nil?
    puts "child id: #{Process.pid}, my parent: #{Process.ppid}"
  else
    puts "parent id: #{Process.pid}, my parent: #{Process.ppid}"
  end
end
```

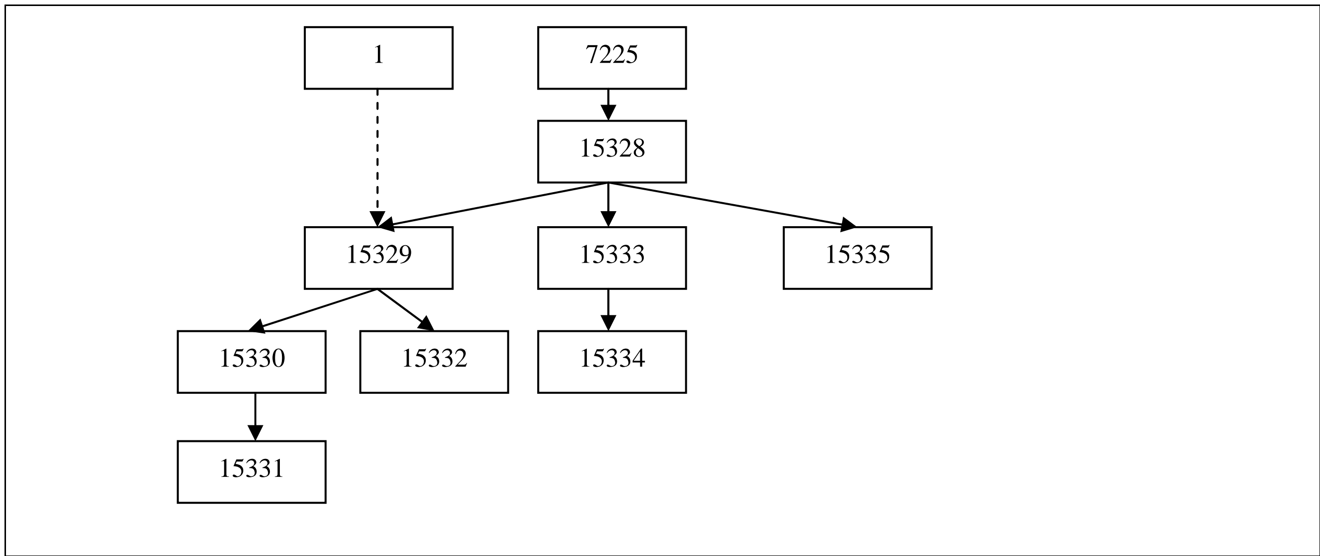
And it produces this output:

```
child id: 15329, my parent: 15328
child id: 15330, my parent: 15329
child id: 15331, my parent: 15330
parent id: 15330, my parent: 15329
parent id: 15329, my parent: 15328
parent id: 15328, my parent: 7225
child id: 15333, my parent: 15328
child id: 15334, my parent: 15333
```

Name:
Login (UPI):

```
parent id: 15333, my parent: 15328
parent id: 15328, my parent: 7225
child id: 15335, my parent: 15328
parent id: 15328, my parent: 7225
robert@roberts:~/Desktop$ child id: 15332, my parent: 15329
parent id: 15329, my parent: 1
```

Draw a graph showing the relationships between these processes. Clearly label each node with its process id.



5 marks

b) In part a) why does process 15329 have two different parents?

Its real parent was process 15328 which is the original process. When this process has completed process 15329 is still running. When a process completes its child processes get given init (process 1) as a replacement parent process.

2 marks

c) In part a) what program is running in process 7225?

The shell (bash actually).

2 marks

d) In Unix what are zombie processes and how are they removed?

A zombie process is a process which has finished and its parent has not waited for its result. To remove a zombie the parent processes can wait for the process, or finish before them. Then init waits for them immediately and they go away.

3 marks

Name:
Login (UPI):

Question 3 – Scheduling (8 marks)

a) Given the following real-time processes calculate a cyclic schedule using Earliest Deadline First. If the deadlines are the same, do NOT unnecessarily pre-empt the running process. Show the schedule as a Gantt chart. The three numbers are Compute time, Period, and Deadline.

Process A (3, 8, 8) Process B (3, 6, 6) Process C(3, 24, 9)

Also use this space for working.

B B B A A A C C C B B B A A A B B B B B B A A A

3 marks

b) Here are the arrival and burst times for a number of processes:

Process	Arrival time	Burst time
A	0	4
B	1	4
C	3	1
D	7	4
E	9	2

From this table draw a Gantt chart showing a Shortest Job First schedule and calculate the average waiting time. If the times are the same do NOT pre-empt the running process.

Also use this space for working.

A A A A C B B B B E E D D D D

Average waiting time = (0 + 4 + 1 + 4 + 0)/5 = 1.8

5 marks

Name:
Login (UPI):

Question 4 – Deadlock (6 marks)

a) There are three copies of a resource R in a system. There are two processes P and Q with the following maximum resource requirements for completion:

P needs 2 Rs

Q needs 3 Rs

Using the Banker's algorithm explain why it is not safe to allocate one R to P after allocating two Rs to Q as in the following table:

command	**state (allocation of Rs)**	**safe?**
Q requests R	P0, Q1	safe
Q requests R	P0, Q2	safe
P requests R	P1, Q2	not safe

P may not be able to complete because it may need a second R which is currently not available. Similarly Q may not be able to complete because it may need a third R. Therefore we can not guarantee that eventually all requests will be met and all processes can finish.

3 marks

b) Given the system in part a) but with one R allocated to P and two Rs allocated to Q (P1, Q2), list a series of requests and releases of Rs that prove it is possible for both processes to complete without deadlock.

Q releases R

P requests R

P finishes (releasing both Rs)

Q requests R

Q requests R

Q finishes (releasing all Rs)

3 marks

Name:
Login (UPI):

Question 5 – Concurrency (11 marks)

a) What is wrong with the following lock and unlock operations (give as many reasons as you can think of)? `locked` is the lock variable.

```
lock:
     while locked
     end
     locked = true

unlock:
     locked = false
```

It doesn't work. There is a gap between testing the locked variable and setting it.

It is a busy wait, unnecessarily consuming CPU cycles.

It is not fair; effectively random selection of waiting processes.

3 marks

b) In the current version of Ruby a lock could be implemented like this:

```
def lock
     loop do
          Thread.critical = true
          if !@locked
               break # out of the loop now
          end
          Thread.critical = false
     end
     @locked = true
     Thread.critical = false
end

def unlocked
     @locked = false
end
```

Name:
Login (UPI):

Explain the reasons for the two calls to `Thread.critical = false` in the lock method.

The first one allows other threads to be scheduled while this thread is waiting for the lock. In particular the thread which currently holds the lock. If this thread is not allowed to run the waiting thread will busy wait forever.

The second one allows other threads to be scheduled after this thread has claimed the lock.

4 marks

c) Complete the try_lock method below. The method should safely set the @locked variable and return true if the lock was granted. It returns false without waiting if the lock was not granted.

```
def try_lock
    Thread.critical = true
    result = !@locked
    @locked = true
    Thread.critical = false
    return result
end
```

4 marks

Name:
Login (UPI):

Question 6 – Assignment 1 (18 marks)

Here is some code from a possible `rsh.rb` program from assignment 1 that can handle single and pipeline commands. The line numbers are purely for reference.

```
 1    # Executes non-builtin and compound commands.
 2    # "commands" is a list of lists,
 3    # e.g. [["ls", "-l"],["grep", "robert"],["&"]]
 4    def handle_compound_or_non_builtin_command(commands)
 5       if commands[-1] == ["&"] # the last element
 6           background = true
 7           commands.pop # removes the last element
 8       end
 9       if (current_pid = fork).nil?
10           while commands.size > 1
11               command = commands.shift # remove the first element
12               rd, wr = IO.pipe
13               if fork.nil?
14                   rd.close
15                   $stdout.reopen(wr)
16                   exec(*command)
17               end
18               wr.close
19               $stdin.reopen(rd)
20           end
21           command = commands.shift
22           exec(*command)
23       end
24       if background
25           # add the current job to the jobs list
26       else
27           Process.wait(current_pid)
28       end
29   rescue Errno::ENOENT
30       $stderr.puts "#{command}: command not found"
31       exit
32   ensure
33       @current_pid = nil
34   end
```

a) Explain what line 12 (`rd, wr = IO.pipe`) does. In particular why is it necessary?

It creates a pipe. The pipe returns two values. An IO object to read from and an IO object to write to. This pipe is going to convey the output from one command in a pipeline into the following command in the pipeline.

4 marks

Name:
Login (UPI):

b) Why is there a call to `wr.close` on line 18? In particular what would happen if the call was not there when executing the following pipeline:

```
ls -l | grep robert
```

To close the write end of the pipe in the parent. The parent will only use the pipe to read from. If the parent keeps the write end open then it will never receive an EOF indicator.

In the example the grep command will never complete as it is still waiting for data from the pipe. The OS doesn't know that no more data will be written to the pipe as it is still open for writing in the grep process.

4 marks

c) The `yes` Unix command sends a constant stream of "y" characters to its standard output. The `head` command extracts the first `n` number of lines from its standard input and then stops. Here is some output from a properly functioning `rsh.rb` program.

```
rsh>yes | head -n 3
y
y
y
rsh>ps
  PID TTY          TIME CMD
 7225 pts/0    00:00:00 bash
13161 pts/0    00:00:00 ruby
13168 pts/0    00:00:00 ps
rsh>
```

If the call to `rd.close` on line 14 is removed the program produces this output:

```
rsh>yes | head -n 3
y
y
y
rsh>ps
  PID TTY          TIME CMD
 7225 pts/0    00:00:00 bash
13054 pts/0    00:00:00 ruby
13062 pts/0    00:00:00 yes
13067 pts/0    00:00:00 ps
rsh>
```

Explain how the `yes` process is stopped in the correct version of `rsh.rb` (with the call to `rd.close`) and why it is still running in the second version.

Name:
Login (UPI):

The yes process gets stopped in the correct version because there is no longer any process with the pipe open for reading. The OS signals a process if it tries to write to a pipe that no one can read from. This signal causes the yes process to stop. In the second version the signal is not sent because the yes process itself could read from the pipe thus the yes process is still running.

4 marks

d) The following lines show the shell handling a command which does not exist:

```
rsh>bad
bad: command not found
```

Explain in detail (with references to line numbers) how this output is produced from the code above.

At the second exec(*command) line 22 an exception is thrown because the command is not available in the user's PATH. The exception is an Errno::ENOENT error which is caught by the rescue statement at line 29.

3 marks

e) Explain why there is a call to `exit` at line 31 in the rescue block. Hint: Think about child processes.

This code is called when an error occurs at the exec. If the exec fails the child process is still running its copy of the rsh.rb program. If exit is not called then both parent and child will continue. The parent will probably be blocked at the call to wait in line 27, and the child will carry on to get the next line of input from the command line. Over time an increasing number of parent processes will be left hanging around, while their children take over running the shell.

3 marks

Name:
Login (UPI):

Overflow space for answers.

Name:
Login (UPI):

Overflow space for answers.

This page may be used for working.