EEE412 / EEE6032 – January 2010

Question 1

Describe the spawning of a child process using the fork system call under the UNIX operating system. [7 marks]

In the UNIX operating system, a child process is created by calling the fork system call which clones the calling process. Thus on completion, the fork call returns in two places: in the original calling process – to which it returns the process ID of the child process – and in the child process, to which it returns a value of zero. The parent and child can thus distinguish their identities. Typically, the parent process continues execution of the original program while the child overlays itself with a new executable using one of the exec*() family of system calls.

How many processes in total will be produced by running the code in Figure 1? Explain your reasoning.

```
int main()
{
fork();
fork();
fork();
return 0;
}
```

Figure 1

[5 marks]

Labelling each of the fork calls:

```
int main()
{
fork();   // (a)
fork();   // (b)
fork();   // (c)
return 0;
}
```

(a) will produce two processes, a child and a parent. Each of these will produce two processes, each of which will produce a child. Therefore a total of eight processes will be produced, including the original process. (This answer would be well-answered by a tree diagram.)

A multiprogrammed operating system can run several instances of the same program. How can the operating system differentiate between them?

[2 marks]

When a process is created it is allocated a unique process identifier (PID). This allows the OS to identify the different instance of the same program.

Why do many operating systems use spinlocking for process synchronisation on multiprocessor systems but not on single processor systems?

[6 marks]

If a process needs to block awaiting the completion of some other process, this second process clearly has to run in order for the first process to continue. On multiprocessor systems, the second process could run on another processor. Therefore it makes sense for the first (blocked) process to enter a loop in which it continually tests for permission to continue (*i.e.* spinlock), rather than context switching. In single processor systems, a process which spinlocks is using the only processor therefore the second process cannot run and so fulfil the conditions for the first process to be unblocked.

Question 2

What is the purpose of the command interpreter? Why is the command interpreter normally separate from the operating system kernel?

[7 marks]

The command interpreter is usually a text-based interface which allows the user to interact with the OS. Commands are invariably interpreted (*i.e.* translated to lower-level actions) at the time of entry rather than being pre-compiled. Command interpreters usually provide simple looping/decision making constructs as well as scripting facilities to allow more complex operations to be automated.

Command interpreters can be built into the OS kernel but are more usefully run as separate, user-space process for a number of reasons: Firstly, this allows the customisation of the command interpreter (*cf.* the range of shells available under the UNIX OS). Second, the command interpreter only needs to run (and therefore occupy memory) when a user is logged in; running the command interpreter in response to a login and destroying it when the user logs out conserves system memory.

Nowadays, what are the principal advantages of a virtual machine?

[7 marks]

Although the original motivations for a virtual machine were to give users the illusion of being the sole users of a computer, this has long since ceased to be a valid objective.

a) The principal advantage is to ensure compatibility, for example backwards compatibility with software which only runs on another operating system. Virtual machines like the Java virtual machine allow code compatibility across different hardware platforms.
b) Encapsulation/security. Different virtual machines running on the same computer are strongly isolated from each other thus ensuring that OS malfunctions in one VM do not affect other VMs. Alternatively, individual VMs exhibit a high degree of immunity from security breaches due to this isolation.

Virtual machines (VMs) are often categorised into: hardware VMs, system-level VMs and application-level VMs. Explain what each of these terms mean and give an example of each kind of virtual machine.

[6 marks]

Hardware VMs are VMs which export a virtualisation of the underlying hardware. The OS which runs on top of such a VM makes extensive use of the real hardware and only calls the virtual machine monitor fairly rarely to virtualise physical devices. An example would be IBM's original VM/CMS system.

A system-level VM exports an system-call API to programs which run on top of it. Such a VM would be used for running a guest OS on top of a host OS. Examples would be VMWare or VirtualBox.

An application-level VM allows the running of an individual application (or family of applications) on a host system and therefore exports library function implementations. An example would be the Java virtual machine which allows any Java program to run on any hardware.

Question 3

Describe the actions taken by the kernel during a context switch between processes.

[2 marks]

When a context switch is initiated, either by a running process executing an I/O instruction or exhausting its timeslice, an interrupt will transfer control to the OS dispatcher. The dispatcher will save the context of the currently running process, writing register values, *etc*. to the process's PCB (process control block). The dispatcher will then restore the context of the next process to run. (In a pre-emptive system, the dispatcher will then load the watchdog countdown timer with a count corresponding to the incoming process's timeslice) and then transfer control to the incoming user process.

A number of processor architectures have a multiple register sets. Why is this useful to an operating system?

[4 marks]

During a context switch, register contents have to be saved to kernel-maintained data structures which are stored in memory. Multiple register can speed-up context switching because the dispatcher only has to switch to another set of registers, rather than write then read from memory.

Is there any speed advantage to splitting a process in to multiple threads of execution? What factors determine whether there is any advantage to multi-threading?

[6 marks]

The answer to this question is involved. Some of the factors to consider are:

a) Splitting a task into multiple threads incurs a context switching overhead. This may or may not be compensated by the sub-division of the task.
b) If thread scheduling is determined at user-level, any thread which executes an I/O operation may context switch the whole process.
c) Multiple threads may be scheduled to run on multiple cores, if the processor has such a feature, but often multiple cores share (the often quite small) L1 cache. If each thread needs to access more memory than can be accommodated in (its 'share' of) the cache, there may be continual cache misses and no speed improvement at all.

d) Interacting threads may need to be synchronised. If the run-time is dominated by one thread for which all the others have to wait then there will be little speed advantage to multi-threading.

In respect of counting semaphores, how are the wait and signal operations defined? Show that unless these two operations are performed atomically mutual exclusion cannot be guaranteed between two processes both waiting on the same semaphore.

[8 marks]

The wait operation on a counting semaphore with internal count variable s is defined as:

```
while(s >= 0)
        donothing;
s = s – 1;
```

similarly, the signal operation is defined by:

```
s = s + 1;
```

If two wait operations execute on a semaphore when the value of s is 1, unless the operations are performed atomically then both waits may decrement the value of s and both may (incorrectly) unblock, thereby violating mutual exclusion.

Question 4

What is the cause of disk thrashing in a paged virtual memory system? How can the operating system detect page thrashing and, once detected, what steps can the operating system take to eliminate it?

[8 marks]

When a system becomes very busy, all pages will be allocated so when a process page faults it will remove a page from another process. When this second process runs, it will immediately page fault. Under adverse circumstances, the situation can develop where the first operation of almost every process is to page fault to try to obtain the page it needs, causing performance to drop dramatically. (In practice, the number of disk accesses to the page file soars causing the disk to be in permanent in operation, hence 'thrashing'.)

The system can detect this situation by monitoring the frequency of page faults. If this rises above some pre-defined limit then the system can reduce the number of concurrent processes by suspending some (maybe low-priority processes) until the system is less busy.

Explain the operation of the open() and close() file operations in a multi-programmed operating system. How can this mechanism be extended to prevent two (or more) processes from both gaining write-access to the same file.

[6 marks]

The open() call will be invoked with the textual name of the file to access. The directory system will be searched and the file descriptor will be added to the process's open-file table; each table will

have its own open-file table. The open() call will return an index (or pointer) into the open-file table which will subsequently be used for file access.

The close() call removes the entry in the open-file table, making the file inaccessible to the process.

To prevent two or more processes from simultaneously having write-access to the same file, the operating system will maintain a system-wide open-file table. When a process makes a request to open a file with write access, this system-wide table is searched. If the corresponding entry is found and another process already holds write-access, the current open() call returns an error condition.

If the file is not currently being used by another process, the file descriptor is added to the system-wide table and a reference to entry in this system-wide table added to the process's local open-file table.

If a process terminates abnormally (*i.e.* with an error), how does the operating system ensure that any files which the process has opened are closed and thus available to future processes?

[6 marks]

The process will maintain its own table of open files which will be referenced via the process control block (PCB). If the process terminates abnormally, the OS will invoke the kernel process to terminate the process which will examine, among other things, the open file table and close (release) all open files as part of terminating the process.