EEE6032 - Autumn 2013/14 - Solutions

1.

a) The microkernel philosophy is to include in the kernel only the minimal set of functionality. The minimal set comprises:

i) Process/thread scheduling

ii) A means of managing address spaces (although not necessarily a full memory manager

iii) An interrupt dispatcher (although the handlers may run in user space

iv) Interprocess communications

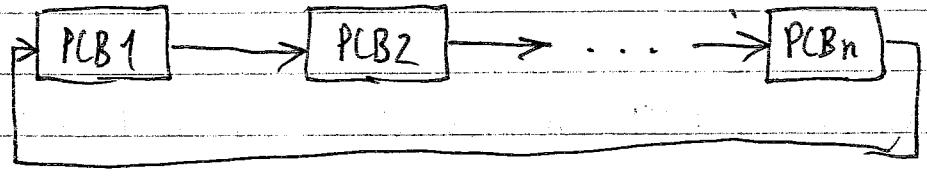
Components i), iii) and iv) require unfettered access to the whole memory space and therefore have to run in kernel space. Since interrupts are central to the operation of the OS, placing this functionality in user space could compromise the system - it has, therefore, to be part of the kernel.

A microkernel may be a good architecture for a real-time

OS because embedded systems usually have modest resources. With a μkernel, it is possible to 'mix-and-match' only the OS components which are needed.

b) To implement pre-emptive multitasking there has to be some means of allowing a process to run for a maximum time. A suitable addition is a countdown timer which is loaded at the end of a context switch, and then counts down independently of the CPU. If the countdown timer reaches zero it triggers an exception which invokes the scheduler to swap the running processes.

c) Typically, the information about a process is held in a process control block (PCB). These are arranged in a circular linked list:



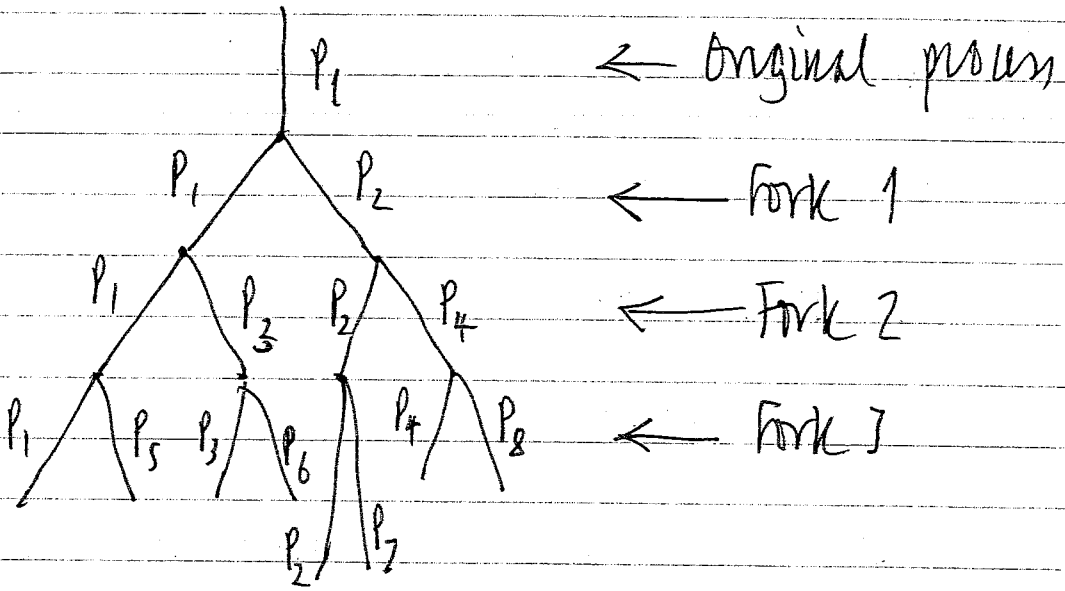
such that the scheduler repeatedly traverses the list, giving each process a timeslice quantum.

d) Many interrupt event give rise to lengthy periods of processing, for example I/O operations. If the single handler is allowed to run to completion, the latency of the interrupt will be poor. If the top half of the handler spawns a bottom half to carry out deferred processing, the top half can immediately return to the interrupted process. Interrupt processing time is effectively reduced.

Since the bottom half is processing the bulk of the interrupt operation, at some later time, care must be taken to lock any resources to which bottom half requires exclusive access to prevent data corruption.

2.

a) The execution of this fragment of code can best be illustrated diagrammatically:



Consequently, a total of eight processes, including the original process, are created.

The clone function gives greater control over process creation than fork. Whereas fork produces a child process with all resources shared with the parent, and an identical copy of the parent's address space, clone allows each commonality to be specified. For example, clone can specify that the child shares memory thereby creating a thread.

Clone thus creates a unified and highly flexible means of creating a concurrent path of execution.

b) A critical section comprises an:

i) Entry section

ii) Critical code

iii) Exit section.

The entry section acts as a 'gatekeeper' which allows only a single process to execute its critical code at any one time; any other process trying to execute its critical code will be blocked.

The principal use of a critical section is to assure mutually-exclusive access by a process to a section of code.

A critical section would not, however, be appropriate for ensuring synchronisation between processes.

c) The completely fair scheduling (CFS) method uses an accounting window in time. If there are n processes currently opened, each process is allocated a timeslice of 1/n of the accounting window. This has two advantages:

i) The timeslice duration is dynamic; processes do not repeatedly run for a fixed timeslice, get interrupted and get immediately scheduled to run again

ii) Interactive processes which wake on user input are given a high priority because they have consumed no time in the current accounting period. The system is thus responsive to interactive processes.

As the number of processes tend to infinity, the duration of the dynamically-allocated timeslice tends to zero. In practice, this is dealt with by specifying a minimum timeslice duration to prevent to large a fraction of CPU time being spent solely on context switching.

3.

a) In a statically-linked program, all the code, including libraries, is combined into a single, standalone executable. A dynamically-linked program contains only references to a shared library; many programs can simultaneously access this/these shared libraries.

The advantages of static linking are that the program is self-contained and cannot be 'broken' if, say, a shared library is updated. The size of the executable is larger, however, so the program occupies more memory.

A dynamically-linked program is smaller, on the other hand, because part of the code exists as a shared library. (This advantage may not amount to anything, however, if the shared library is not being used by any other programs.) Additionally, ~~there~~ dynamically-linked programs are susceptible to being 'broken' by updates of shared ~~lib~~ libraries.

b) Typically, an executable is assembled from a number compiled source files and libraries, a process which is carried out by the linker. The symbol resolution phase checks that all symbols used are correctly defined and of the appropriate type. For example, that a function has the correct prototype and that the body of the function exists.

When a code module is compiled, the resulting (relocatable) object file is based (arbitrarily) at address zero. During relocation, the necessary code and global variables are allocated unique addresses in the final executable and all symbolic references replaced with the final addresses.

c) GUIs handle user input asynchronously. The underlying OS is modified to send (short) messages to a application which have to be interpreted and induce the requested

action ~~was~~ by calling an event handler. The message loop comprises an infinite loop:

```
while (true)
{
    GetMessage (&msg);
    switch (msg)
    {
    case WM_PAINT:
        OnPaint (msg);
        break;
        :
    }
```

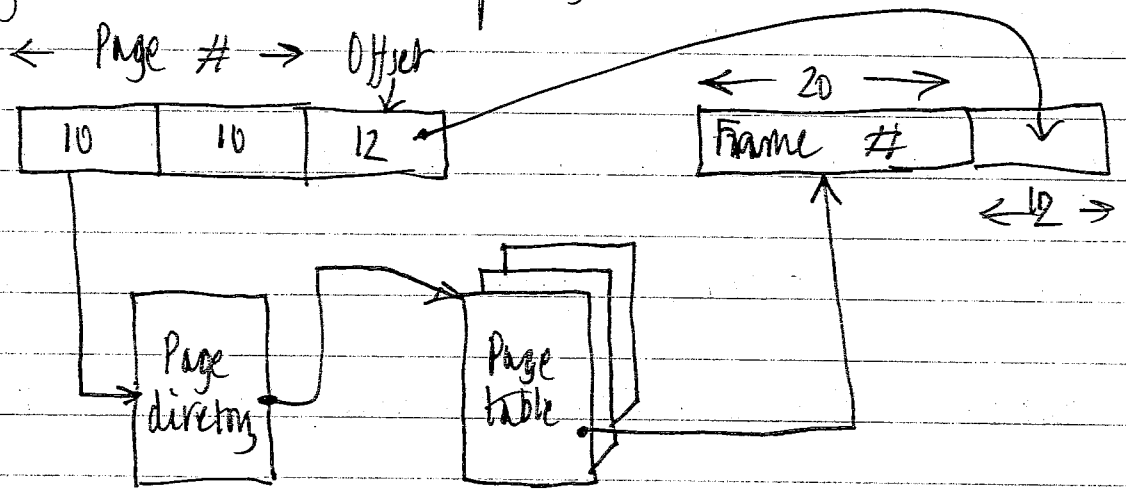
which first calls the blocking function GetMessage() which reads the first item from the application's message buffer. The code then branches depending on the message. For example, if the message is the pre-defined integer constant WM_PAINT, the OnPaint() function is called (which would repaint the window).

In this way, asynchronous messages actioned by the user are handled by the program.

d) Mapping the kernel memory into the upper half of every process's address space allows user processes to execute kernel functions (via the trap mechanism) without switching context. Since the kernel functions are 'visible' to the user process, they can be executed by a simple call instruction (provided the CPU is in kernel mode).

4.

a) Simplistically, if a process has a single-level page table this must be loaded into memory. However most of such a page table will be empty thereby wasting large amounts of memory. A multi-level page table interprets the page number as two parts:



The topmost 10 bits are the index into a page directory comprising $2^{10} = 1024$ entries which point to page tables. The next 10 bits of the logical address are an index into the appropriate page table which yields the 20-bit frame number. The advantage is that only as many 1024-entry page tables need be allocated resulting in a

large saving of memory.

The disadvantage of multi-level page tables is that two levels of indirection are now necessary to retrieve the frame number making the arrangement slow. This can be mitigated by caching recently used frames in a translation lookaside buffer (TLB).

b) Allowing a pre-call to see if making a system call would block makes it possible to prevent the whole process (and every thread within it) from blocking. Thus the speed benefits of user-level threading can be fully utilized.

It is necessary for user-level threads to periodically execute a thread_yield because, since the OS does not support threads, there is no other way to achieve multi-tasking. Pre-emptive multi-tasking, which relies on hardware support, cannot be implemented and co-operative multi-tasking between threads ~~may~~ must be used.

c) The main advantage of multi-threading an application is to generate ~~user-interface~~ user-interaction threads and worker threads. The user-interaction threads can give fast response while worker threads can continue processing in the background.

The other reason for multi-threading is to exploit multiple CPU cores.

One complication is that synchronisation between threads may be necessary if they are accessing shared data structures.

d) A disk is divided into fixed size sectors which are chained into a file. It is thus necessary to record and access the sequence of sectors which comprise the file. This can conveniently be done in a file allocation table (FAT) where the initial entry gives the ~~first~~ index of the second

entry, and so on.

The advantage of a PAT is that, because the whole table is in memory, access, particularly random access, is very fast. The disadvantage is that, since the whole table must be maintained in memory, the approach does not scale well for large disks; the PAT requires on entry per sector on the disk.