



These notes are designed to help you to understand how to answer the questions. A few complete answers are supplied, but the aim is to get you thinking further about the issues raised by the questions.

You need to aim to be able to answer all of these questions by the end of the course. These notes are intended to give you further guidance. There are a few answers supplied, but you need to work towards answering all the questions yourself.

1. Refer to the given background information and Figure 1 on the original [tutorial](#) handout.
 - a. How significant is it that the total cost and DRAM cost lines are almost indistinguishable? What does that mean?
DRAM cost is becoming the most significant factor in being able to build a low-cost device with nontrivial performance goals – as opposed to CPU cost, which has historically been more important.
 - b. In what year would the total device cost drop below the target cost of \$100?
Easy: Look at where the red line (or the yellow line, since it is for practical purposes the same) crosses the brown line.
 - c. In what year would the CPU cost drop below \$100?
Same as part b.
 - d. If you could halve the memory requirement, how much sooner would the target cost be met? How much sooner if you could reduce the memory required to 10% of the original design?
The second part is also pretty simple. The point where the cost = \$1000 represents when 10% of the memory originally contemplated would cost \$100. Look up the year, and the rest is simple arithmetic. Estimating the point where half the memory would be affordable is harder from the graph because of the log scale, but note that you have been told that the cost halves every 3 years – so how much sooner would the \$100 target be reached for half the memory? Not too hard when you think of it this way.
 - e. Given these trends, how would you predict the operating system for a PDA might differ in terms of its resource usage, as compared with a 1980 mainframe operating system?
The logical thing to do if designing for this market is to try to minimize memory use, rather than to focus on minimizing CPU resources. That is, given the information here. A real design has other constraints like power use. A further issue to think about: how does power use influence what you might expect the CPU to do? Would that be an issue in deciding what a suitable OS should do?
 - f. If some people have said, “Memory is so cheap, we can treat it as if it’s free – efficient memory management strategies of the past are no longer relevant.” – are they talking nonsense? Discuss this question, considering the growing market for PDAs, smart devices and embedded computers in the light of the trends in Figure 1.
Think about the previous answers, particularly the fact that the total cost line very quickly disappears into the memory trend. Moore’s Law is usually considered in terms of CPU speed improvement at a given cost. If we turn it around to focus on cost reduction for constant performance, cost reduction of DRAM turns out to be a much more significant issue than CPU cost reduction.
 - g. By only looking at CPU and DRAM cost trends, have we neglected anything important? What components of a traditional large-scale system are likely to be found in a hand-held device?
Think about which components don’t make sense in this environment, for example, facilities to support large numbers of terminals, high-bandwidth file systems, backup systems. On the other hand, do some components such as power management become more critical to the cost? How many of these issues also relate to OS facilities which will no longer be relevant on the PDA, versus others (e.g., power management) which didn’t exist on a traditional large-scale system?
2. Many portable devices have real-time requirements, such as multimedia processing, text-to-speech, handwriting recognition and speech recognition. A developer of such devices proposes to use Linux because it’s free, and the licensing costs of using a proprietary operating system would be a significant fraction of the cost.
 - a. How many of the following would you classify as *hard* versus *soft* real-time requirements? Explain your answer in each case.
The defining attribute of hard real time is that time constraints have to be met, otherwise the system is broken. In each case, then think about whether the requirements allow for some possibility that something could be missed. Then, think about whether those requirements imply that a time constraint could occasionally be violated.
 - i. Continuous speech recognition. Up to 1% of words may be incorrect, requiring user intervention.
 - ii. Handwriting recognition. Up to 5% of characters may be incorrect, requiring user intervention.
 - iii. Text-to-speech. 100% accuracy is required.
 - b. Which of the requirements do you think would be hard to achieve with an operating system like Linux? Explain in each case.
Linux has a scheduler which, despite variations like priority, does not support a specific kind of process with time guarantees. For example, only the kernel is allowed to mask interrupts. For this reason, hard realtime constraints cannot be met. What about soft realtime? If you cannot guarantee that an interrupt or other event out of your control can be prevented, can you guarantee that a soft realtime constraint can be met? Think about variations on handling missed events in the outside world in general. Taking the examples here, can you construct a situation where the soft constraints (tolerance for errors, in the examples given here) may be impossible to meet as well? Consider real-time applications you have used, like streaming audio or video off a web server, as well. What are the issues there which may soften the real-time requirements? If you have a general-purpose network like the Internet involved, can you guarantee real-time requirements of any form, or just try to aim for a target, with a fallback position if you can’t meet it?
3. Here are a few ball park numbers, relating to speeds of components of a computer system:
 - a processor completes an instruction, on average, every 0.5ns: call this 1 time unit
 - DRAM (the stuff main memory is made of) takes 50ns for an operation: this is 100 time units
 - a disk takes 5ms to complete an operation: this is 10,000,000 time units
 - a. Every time an I/O operation takes place, at least one disk access takes place. Assume that a minimum disk operation moves 512 bytes, and memory has to be accessed by the disk to store all the information. Memory is accessed in units of 4 bytes at a time, so this will be 128 memory operations. Then, to recover the information from the disk (now in DRAM), the processor also has to do 128 memory operations.
 - i. Without doing a detailed computation, think about the amount of time the processor will wait for disk and memory, if a program does a disk operation only in 1% of the instructions it executes.

A disk operation takes 10,000,000 times a CPU instruction's execution time. If these happen 1% of the time, that's still on average 100,000 times a CPU operation (averaged over all instructions), so the CPU will wait a very high fraction of the time for disk.

- ii. How can an operating system hide the time you are spending waiting for the disk?
Pretty straightforward – if you can't answer this, look for the answer in the book.
- iii. How comparable is the time you would spend waiting for DRAM? Is it a comparable problem to waiting for a disk?
DRAM operations take 100 time units. Add up all the memory operations (simple arithmetic), and multiply by 100. Compare the number you end up with against 10,000,000 – is it close?

- b. A computer has a hierarchy of memory components. The fastest, smallest, most expensive parts are closest to the processor. Given the numbers in this question, explain why the interface between DRAM and disk is usually managed in software, whereas the interface between DRAM and higher layers is usually managed in hardware.

Managing something in software allows you to be more sophisticated. In the case of managing the disk, saving one disk access through managing the interface more intelligently saves the equivalent of 10,000,000 instructions. So running a few hundred instructions when deciding what to put in memory and what to put in disk makes sense. With the given numbers, would it make sense to run a few hundred instructions when deciding what to put in DRAM, as opposed to a faster layer of memory (cache)?

4. You are planning a large-scale computer system with the following requirements:

- You don't mind if performance is reduced for up to 24 hours, as long as the system is able to continue to run if a part of it is broken
- You want to maximize performance (in the case where nothing is broken)
- Running costs are significant as a factor in your overall budget, and you are willing to buy a more expensive solution if it cuts running costs.

- a. Are you looking for a *high availability* or *high reliability* solution? Explain.

Availability implies that the system can be used (even if part of it is broken); reliability that it does not break. To answer the question is a simple matter of choosing which applies, and then explaining why the definition fits. The distinction between availability and reliability is useful because availability is what the users sees (does the system work?) whereas reliability is what the maintainer sees (is something broken?). Increasing availability in many cases may result in decreasing reliability. 4 computers, any of which can do the required work, will increase availability. But compared with only 1 computer, you have 4 times the number of things to break, so the probability of something failing is higher.

- b. Would a single-processor system be a good fit to the requirements? Explain.

What if the CPU develops a fault? Would it meet either definition, let alone the one you chose in (a)?

- c. How would a network of cheap systems compare with a single, large multiprocessor system? Discuss the trade-offs, and how you would apply the given requirements to making a decision.

Think about the cost of an army of technicians patrolling a room full of computers reinstalling images (OK, probably not that extreme – you can automate some of this) and swapping broken parts versus a single box with redundant parts. Which is going to be cheaper to look after – assuming the extra cost of the big box is not so huge as to cost more than hiring a few extra people? Otherwise, the two solutions can equally be argued for on availability grounds. Performance depends on how partitionable the workload is. Generally, the single-box solution is likely to be faster, because it doesn't have to communicate on a slow, commodity network. However, if the workload is easy to split between different computers without much communication between them, the network of cheap systems can be an option. For example, many high-traffic web sites are split over multiple servers, sometimes in different parts of the world. In this case, though, the performance goals are easier to meet by splitting the system, because alternative sites can be placed nearer the users. For example, Akamai <<http://www.akamai.com>> sells a service where you are redirected to a local site when you attempt to access a URL of one of their clients. This may be a less reliable situation than if you have one computer providing the service in the sense that if you have multiple servers, the probability of any one failing is higher than if you had one similar machine, but availability is higher, and performance is higher, because the network traffic starts from closer to the client.

5. The kernel, ideally, is the smallest part of the operating system which runs without memory protection, and other limitations of applications. The kernel provides a hardware abstraction layer, which hides the machine-specific detail and makes the rest of the operating system (ideally) relatively portable. Which of the following do you think should be part of the kernel? Explain in each case:

- a. A disk driver, which needs to know how the disk is organized internally, and its low-level command set.

Clearly, this is part of a hardware abstraction layer and hides machine-specific detail. Some would argue for drivers to be part of the kernel on these grounds and also for security. There is also however the argument that a disk is a removable device which can easily be changed, so drivers are closer to being a feature of such detachable hardware.

- b. A file system, which organizes information, without knowing how the disk or other devices drivers control each device type.

It is not so clear that the file system is part of the hardware abstraction layer, since device drivers can make different device types look the same to the file system. On security grounds, a case can be made for making the file system part of the kernel. But it's also possible to isolate aspects which are security-specific and provide only those as kernel services. The advantage of this approach is that it becomes easy to support multiple file systems.

- c. A low-level graphics driver, which knows how to address the graphics hardware, and convert programmer-level graphics primitives into hardware-specific commands.

The same issues apply here as for the disk driver, except the security issues are different. How do they differ?

- d. A window manager, which doesn't know anything about the underlying hardware, but relies on the graphics driver to provide hardware-specific functionality.

Look at the issues identified for the file system. Is the window manager at a different level in terms of abstraction? Are the security issues similar?

6. Can you think of any major attribute of traditional large-scale computers which is not today found on personal computers (excluding features not relevant to functionality, like size, and the ability to dim the lights of the nearest city)? What is the difference between the "personal" and "server" editions of mass-market systems like Windows (XP, etc.) and Mac OS X?

For the second part, look at the web sites of Apple and Microsoft, and list the differences between their personal and server editions. Think about whether these differences imply a slightly different install, or a significantly different operating system.

Let's list some features to address the first part; think about how many a typical PC can support:

- ability to support multiple users using terminals
- ability to access resources over a network, sometimes from a distant geographic location
- security including user ids and passwords, to protect users from each other
- ability to scale I/O up to dozens, even hundreds of devices
- high availability – e.g., major components such as a CPU or I/O device can be removed without halting the entire system
- support for large memory systems in hundreds of MBytes
- virtual memory capable of supporting multiple separate protected address spaces
- protected multitasking, in which processes cannot take control of the CPU to the exclusion of other processes

7. How do you think a systems programming language might differ from a language only designed for applications? Consider the following

features, and comment on whether Java meets each requirement:

Java can't do most of these directly, though it can do those which are not available in the language indirectly. At worst, Java can use the Java Native Interface <<http://java.sun.com/docs/books/tutorial/native1.1/>> to invoke code in a language which can do the required thing. Look up a Java reference if you don't know which of these Java can do easily. Would you write an operating system in Java? What other issues would you consider?

- a. Ability to access data at a specific memory location – the kernel usually runs without address translation, and some operating system details require that a programmer address a specific memory location, e.g., to check the status of a device.
- b. Ability to access code at a specific memory location – an operating system may sometimes place device drivers or interrupt handlers at specific memory locations: you need to be able to treat that memory location as a starting point for code you invoke.
- c. Ability to control memory allocation and deallocation.
- d. Ability to ensure that a particular piece of code will execute with low (ideally zero) probability of some high variation in its usual running time.
- e. Ability to pass a pointer to a specific piece of code so it can be accessed by an operating system routine.
- f. Ability to express numbers in other bases such as hexadecimal, for convenience of encoding operating system information.