

Homework 1 Solution

1. Answering this question requires defining what is inside the operating system and what is outside. The two extremes are “the OS is the kernel” and “the OS is everything which you get on the CD”. A reasonable middle ground might be “the OS is everything which is required for running applications on the machine”, which side-steps the issue of what is in the kernel and what isn’t (many operating systems, especially microkernels, run important system services outside the kernel).

Advantages: in many OSes (e.g. Microsoft ones), applications make heavy use of shared libraries which are closely tied to the operating system (e.g. file browsing code, windowing system). Using this code, which may be in the kernel, improves the performance of the applications beyond what they could achieve if they used their own libraries. Tying important applications to the OS makes them easier to manage and use, which is also attractive from a commercial perspective.

Disadvantages: customisability is the main technical disadvantage. If you have a high-performance web server, you do not need a web browser, or a mail client, and maybe not even a windowing system. If these are tightly coupled to the OS, you pay a performance penalty (e.g. longer code paths due to “unnecessary code” in the kernel, more physical memory used by kernel services and shared libraries which you don’t need). As a user, you might want to use a different mail client, but the alternative might not be as tightly-integrated with the system, so it performs worse. There are also other commercial objections (c.f. Microsoft’s legal problems with incorporating applications into its OSes).

2. DMA is a function of advanced I/O controllers that allows the CPU to instruct those controllers to transfer large blocks of data to or from main memory, independently of the CPU. The primary advantage of handling data transfers in this way is that the CPU handles only one interrupt per block rather than one interrupt per memory word. This allows CPU and I/O operations to be performed in parallel, and is especially important for fast I/O devices. DMA is frequently used in network cards and disks.
3. Caches improve access time. They are useful in a number of places in the storage hierarchy, whenever there is sufficient access locality. For example, a memory cache can take advantage of *spatial* locality (such as accessing an array, or sequential instructions in memory) by filling an entire cache line from main memory at a time. A second major benefit occurs on writes in a disk cache, for example, where it is sometimes possible to eliminate a write, if a superceding write occurs shortly after.

Caches improve performance by reducing the latency of access to storage. This benefit comes at the price of introducing consistency problems.

The size of the cache is a function of a price-performance trade off. Since caches are far more expensive than the storage which they augment, replacing the lower level with the faster cache would greatly increase the overall cost of the system. In addition, incremental increases in the cache size provide diminishing returns in performance once the potential for access locality has been exhausted.

4.
 - (a) Sending data using fixed-size kernel mailboxes involves cutting up the data into fixed-size messages. The fixed-size messages can be sent among the processes using the appropriate system calls. The sending process would call send, there would be a context switch to the receiver, and the receiver would call receive.
 - (b) Sending data using the shared memory mechanism provided would involve two steps. The first step is to create the shared memory region. It is most efficient to share the memory actually containing the message to be sent. There is a context switch to the receiver, which then maps the memory region, reads the data from it, and finally unmaps it.
 - (c) The advantage of the memory mapped operation is that data does not get copied twice. However, it may require more system calls. For small messages, the kernel mailbox approach will be more efficient. For larger messages, the shared memory approach will win.
5. There is a faster context-switching time between threads in the same address space than between processes. Threads also consume fewer system resources than processes, since they share open files, the page table, etc, for the process they run in. The big disadvantage of threads is that there is no memory protection between threads in the same address space; one thread executing buggy code could mis-behave and trash the entire address space. Web servers are an example of applications which benefit from a multiple-threads, rather than a multiple-processes design. They have many "threads of control" which share the web server state and execute short tasks (fetching individual pages). Any application which executes potentially buggy code provided by an untrusted author would be better served by running it in a separate address space, where it cannot damage the primary process. This is the motivation for running JVMs in separate address spaces.
6. Microkernels typically implement a large part of the operating system's functions, including some system calls, outside the kernel in user-level servers. Common subsystems implemented entirely or partially outside the kernel include file system, memory management and networking. The main disadvantage of this approach is that a system call implemented by a user-level server results in four context switches: user application-to-kernel, kernel-to-server, server-to-kernel, kernel-to-application. This compares with the application-to-kernel, kernel-to-application path required in a macrokernel. Microkernel designs are therefore inherently slower than well-implemented macrokernels.
7. The principal aim in designing the system call interface to an operating system is to minimise the number of user-to-kernel transitions which an application requires while it is executing. A flexible, powerful interface improves application performance, but it can also be hard to use (for instance, system calls can have a large number of arguments and many flags and options which are rarely used). System libraries linked to applications can help mask this complexity. They can also further improve performance by caching results from system calls in the state of the process, so as to reduce the need for further system calls to retrieve the same information. For instance, there is no need to make a system call to retrieve a process's PID twice, if the operating system guarantees it will remain the same over the life of the process. Finally, a layered approach allows for lower layers to be changed, as long as the top interface remains the same. This allows for application portability across operating systems and operating system versions.