Arizona State University
UGTA Lisonbee (4), UGTA Karaliova (4), Lecturer Acuña (1)

Unit 4 Sample Problems - Operating-System Structures

In this sample problem set, we will practice concepts of operating system structures.

- Length: 50 minutes with discussion.
- Questions: Q1-Q3, Q6-Q7 (optional: Q4-Q5, Q8)

1 Operating-System Services

2 User and Operating System Interface

3 System Calls

1. [Karaliova] Consider the following pieces of functionality:

   (a) Finding duplicate nodes in a linked list
   (b) Providing a report on system health that details all processes currently running on a system
   (c) Compressing images

   Which of these would need to be implemented as a system call?

   Ans: [Karaliova]

   A system call is a call that invokes some low-level subroutine that likely requires a high level of privilege such that the OS alone offers its service. Finding duplicate nodes in a linked list (a) does not require an invocation of system calls as it can be completed within a runtime environment. Providing a report on system health that details all processes currently running on a system (b) would need to be implemented as a system call in order to leverage encapsulation of reusable low-level subroutine that is executed by the operating system and provide a secure way of accessing information about processes running on a system. Compression for images (c) does not need to be a system call. However, compressing functionality requires access to file system with an ability to manipulate files, so it is likely to utilize system calls within the functionality itself.

2. [Karaliova] Consider the following function from C stdio.h library that handles renaming of a file:

   ```
   int rename (const char *oldname, const char *newname)
   ```

   What conditions/events does the underlying system call need to check for in order to rename a file? Name at least three. [Hint: think in terms of exceptions that can be thrown while performing this operation].

   Ans: [Karaliova]

   The system call has to provide a guard against unauthorized file access (that is, file permissions or read-only directory), disk space restrictions (e.g.: the directory that would contain 'newname' has no room for another entry, and there is no space left in the file system to expand it), file with the name 'newname' already exists (the directory newname isn't empty), file with the name 'oldname' doesn't exists, file with the name 'oldname' is already in use by the system (locked for writing), file name 'newname' is invalid (e.g.: contains illegal characters).

3. [Karaliova] For most programming languages, the run-time support library provides a system call interface that translates function calls into corresponding system calls. Why do we need an intermediate between the two instead of invoking system calls from a program directly? Explain.

   Ans: [Karaliova]

   System call interface provides a layer of abstraction that separates user mode from kernel mode by hiding the details of the operating system interface and what specifically is taking place during the invocation and execution of a subroutine/service. Hiding these specifics, along with that fact that an invocation from a user application allows operation system to manage user access priviledges, facilitate a more secure way of handing sensitive (and potentially harmful if used maliciously) system calls. Further, it enables maintainability and portability.

1 Types of Systems Calls

4. [Karaliova] One of Unix process control type system calls is fork(). Fork() creates a new process by duplicating the calling process. The new process is referred to as the child process, the calling process is referred to as the parent process. The child process and the parent process run in separate memory spaces. Using fork() or any other example of system call of process control type (Unix or Windows), explain why it is important for programming APIs to have access to system calls.
   Ans: [Karaliova]

   System calls of process control type within a programming API provide a programmer an ability to interact with processes running on the system. Higher level languages provide abstracted way of manipulating processes via libraries while lower level languages like C typically allow more direct control with less overhead. Fork() specifically is utilized for implementing concurrency within the program. Further, malloc() and free() in C are examples of process control systems calls implemented within a programming API that allow a programmer to perform manual memory management within a program.

2 Operating-System Design and Implementation

5. [Lisonbee] Why are operating systems generally implemented using lower level languages? For instance, Windows, MacOS, and Linux are all implemented in mostly C++, C, and some assembly. **Explain** why these types of languages are chosen over higher level ones such as Java or Python.

   Ans: [Lisonbee]

   Generally speaking, lower level languages produce lower overhead and allow greater control of the hardware. An operating system must sit very close to the hardware (relatively speaking), and as such use languages that can better manipulate said hardware will generally produce more favorable results in terms of efficiency and resource management. More abstract languages may lack the ability to do low level manipulation of hardware and memory, making the creation of an operating system outright impossible.

3 Operating-System Structure

6. [Acuña] Say that you are designing an operating system for an intersellar probe that is meant to run for hundreds of years. What structure (simple, layered, microkernel, or modular) should you choose for its kernel? Explain.

   Ans: [Acuña]

For interstellar probes we require an OS that is reliable androbust as it is expected to run correctly for hundreds of years. Keeping the kernel as small as possible would be the better option as it would be more efficient without any significant overhead and easier to debug/maintain. One good choice could be a microkernel, since we can focus on debugging the core system, making sure it has the ability to isolate components, and recover from errors. Other choices mean the system becomes more tightly dependent, which can lead to cascading errors.

7. [Lisonbee] Compare and contrast the organic and microkernel operating system structures. **Explain** how the efficiency (speed), security, and maintainability of each compare, and why.

   Ans: [Lisonbee]

   An organic approach will generally provide the greatest speed when compared to the microkernel structure as there is very little encapsulation and abstraction between different layers. Thus application programs can directly communicate to lower level software that in a microkernel approach it would otherwise have to be routed through the appropriate kernel modules. Both the security and maintainability of the microkernel structure are better than that of organic as the kernel is clearly defined, and should properly act as an encapsulating layer between application programs and the hardware. Furthermore, separate modules can be developed and debugged independently of one another, and therefore makes it easier to maintain.

8. [Lisonbee] Different operating system structures offer both benefits and drawbacks over one another, and it's important to understand what kind of structure should be used for different use-cases.

   Consider the following situation. You are tasked with creating a very secure piece of software for an embedded system that should have a relatively small amount of overhead. Given the nature of embedded systems, once you deploy the system you won't be able to update the software, thus maintainability isn't a large factor.

   What structure (simple, layered, microkernel, or modular) should you choose for its kernel? **Explain**.

   Ans: [Lisonbee]

   A layered kernel design would be best suited for this application as it provides the greatest amount of encapsulation for low level data, and keeps the end user far away from the hardware. A layered approach will be more difficult to design and change, though that isn't an important factor for this use case.