COS Questions – Lecture 1

Operating System Concepts (Tenth Edition)

Operating System Structures

What is a kernel and what are its objectives?

A kernel is the core component of an operating system. Using inter-process communication and system calls, it acts as a bridge between applications (user space) and the data processing performed at the hardware level (kernel space).

2.1 What is the purpose of system calls? Give some examples of system calls.

System calls allow user-level processes to request services of the operating system.

A system call is a way for programs to interact with the operating system. A system call is used to request system-level services from the kernel, such as creating a process, opening a file, allocating memory etc.

A better way to think about it is that system calls define an API (application programming interface) to kernel code.

2.2 What is the purpose of the command interpreter? Why is it usually separate from the kernel?

The command interpreter is used to interact with system commands. On UNIX system, a command is used to launch a system program. It is usually not part of the kernel because the command interpreter is subject to changes, also allowing users to alter or exchange the CLI.

2.3 What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?

A process is created using the `fork()` command, which creates a child process.

2.4 What is the purpose of system programs?

System programs are programs associated with the operating system, but are not necessarily part of the kernel. They are used to aid managing the system while it's running. They can leverage system calls to provide a functionality, and be chained with other system programs for a pipeline of operations.

In some sense, they are bundles of useful system calls.

2.5 What is the main advantage of the layered approach to system design? What are the disadvantages of the layered approach?

The layered approach provides a modular structure, which is easier for developers to implement, maintain and debug. E.g., the first layer can be debugged without any concern for the rest of the system.

Pure layered approaches are rarely used in OS design. It is difficult to appropriately define the functionality of each layer. The overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service.

2.6 List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.

Program execution. The operating system loads the contents (or sections) of a file into memory and begins its execution. A user-level program could not be trusted to properly allocate CPU time.

I/O operations. Disks, tapes, serial lines, and other devices must be communicated with at a very low level. The user need only specify the device and the operation to perform on it, while the system converts that request into device- or controller-specific commands. User-level programs cannot be trusted to access only devices they should have access to and to access them only when they are otherwise unused.

File-system manipulation. There are many details in file creation, deletion, allocation, and naming that users should not have to perform. Blocks of disk space are used by files and must be tracked. Deleting a file requires removing the name file information and freeing the allocated blocks. Protections must also be checked to assure proper file access. User programs could neither ensure adherence to protection methods nor be trusted to allocate only free blocks and deallocate blocks on file deletion.

Communications. Message passing between systems requires messages to be turned into packets of information, sent to the network controller, transmitted across a communications medium, and re-assembled by the destination system. Packet ordering and data correction must take place. Again, user programs might not coordinate access to the network device, or they might receive packets destined for other processes.

2.10 Describe three general methods for passing parameters to the operating system.

1. Pass parameters in registers
2. Registers pass starting addresses of blocks (tables) of parameters
3. Parameters can be placed, or pushed, onto the stack by the program, and popped off the stack by the operating system.

2.12 What are the advantages and disadvantages of using the same system call interface for manipulating both files and devices?

Each device can be accessed as if it was a file in the system. It provides a more generic interface; user program code would be the same used to access both files and devices. A device can be swapped out without the need of changing code (since the API is the same).

It may, however, be difficult to capture the functionality of some devices within the context of file access API, resulting in a loss of functionality or loss of performance.

A good use case example might to be working on some software that requires a simulated device, such as a credit card reader. In Linux, one could implement a simulated device as a file (e.g. `/dev/card_reader`) that returns a string when reading from that file using the `open()` and `read()` system calls. Later on, this file could be replaced with the actual device, requiring no changes in the software.

2.13 Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?

Yes, it is possible to develop a new command interpreter using the system-call interface on those OS's where the interpreter is not tightly integrated into the system (i.e., most operating systems other than Windows.)

The Unix family of operating systems provides multiple command-line interpreters (sh, bash, csh, etc.) The "sh" in each of these programs stands for and is pronounced as "shell", because the command-line interpreter is thought to serve as a protective shell around the system programs.

2.14 Describe why Android uses ahead-of-time (AOT) rather than just-in-time (JIT) compilation.

Whereas many Java virtual machines perform just-in-time (JIT) compilation to improve application efficiency, ART performs ahead-of-time (AOT) compilation. Here, `.dex` files are compiled into native machine code when they are installed on a device, from which they can execute on the ART.

Just in Time (JIT)	**Ahead of Time (AOT)**
Each time when the app is run, it dynamically translates a part of the bytecode into machine code.	During the app's installation phase, it statically translates the bytecode into machine code and stores in the device's storage.

(source)

Since JIT compiles only a part of the code, it has a smaller memory footprint and uses less physical space on the device. However, AOT compilation allows more efficient application execution as well as reduced power consumption, features that are crucial for mobile systems.

2.15 What are the two models of inter-process communication? What are the strengths and weaknesses of the two approaches?

Message passing model and the shared-memory model.

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. The developers need to worry about synchronization.

Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. This requires synchronization.

2.16 Contrast and compare an application programming interface (API) and an application binary interface (ABI).

The API is the "human-level" application interface, such as function name, return type, parameters etc. In C/C++ this is what you expose in the header files that you ship with the application.

The ABI is what the compiler uses "under-the-hood" to build an application. It defines, amongst other, how parameters are passed to functions (registers/stack), who cleans parameters from the stack (caller/callee), where the return value is placed for return, how exceptions propagate etc.

2.17 Why is the separation of mechanism and policy desirable?

Mechanisms determine how to do something; policies determine what will be done. The separation of policy and mechanism is important for flexibility. Mechanism and policy must be separate to ensure that systems are easy to modify.

2.19 What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

Microkernels ease the extension of the operating system; new services are added to user space and do not require modification of the kernel. The microkernel also provides more security and reliability, since most services are running as user processes and interact with the kernel via message passing – if a service crashes, the whole system will not necessarily fail. Unfortunately, microkernels can suffer from performance decreases due to increased system function overhead.

2.21 How are iOS and Android similar? How are they different?

Both are based on UNIX, Android uses the Linux (monolithic) kernel, whereas iOS uses a hybrid kernel. Both use middleware that supports databases, multimedia, and graphics (to name only a few).to augment the functionality of the OS.

Process Concept

3.0.1 What is the difference between a process and a program?

A process is an instance of a program, i.e., a program is essentially code waiting to be executed.

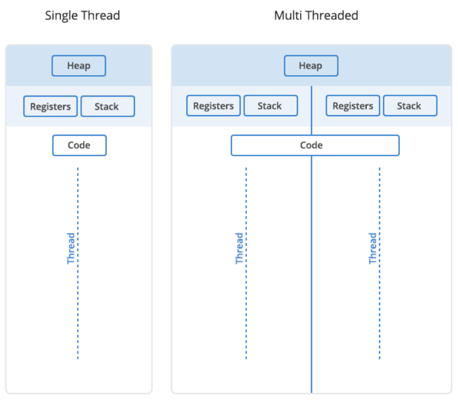
3.0.2 What process states are there?

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (e.g. I/O completion or signal).
- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

3.0.3 What is the difference between a thread and process?

A process is a program that is running with its own PCB and allocated memory, and can own multiple threads.

A thread is a subset of the process, and can be seen as a "lightweight process". All the threads running within a process share the same address space, but threads typically get their own stack.



3.0.4 What is a Process Control Block (PCB) and what is its purpose? Can users access the information stored in one?

The process control stores many data items that are needed for efficient process management, such as the process state, process id, program counter, registers etc. The process control block is kept in a memory area that is protected from the normal user access.

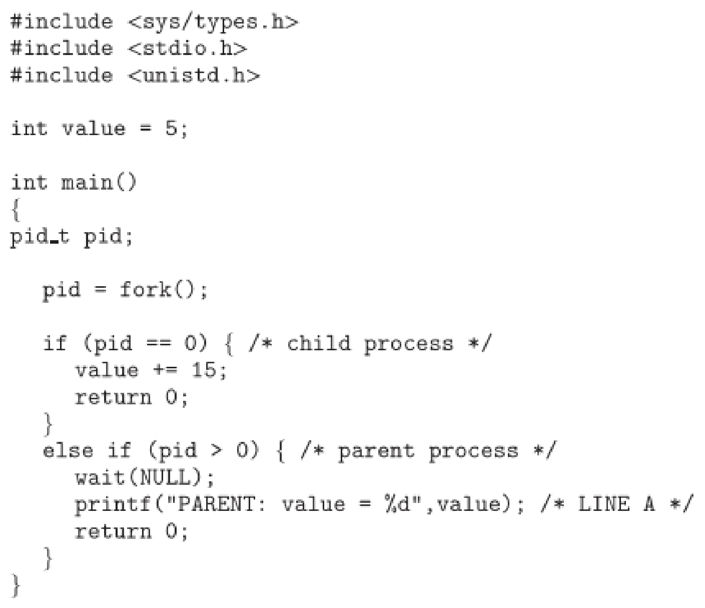
3.0.5 What is the difference between heap and stack storage?

Both are allocated memory to store data such as variables.

The stack has a fixed size and is used for local variables, return values etc. When a function executes, it is added onto the call stack in a FIFO order as a stack frame with allocated memory for the local variables of that function. If a stack runs out of space, a *stack overflow* occurs and the program crashes.

The heap is used for dynamic memory allocation and can grow in size if necessary. It may experience problems of fragmentation and is generally a bit slower due to allocation.

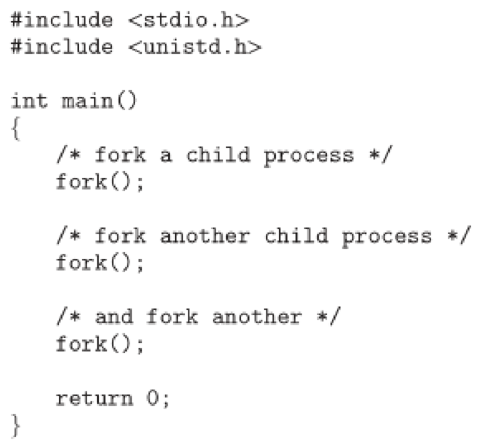
3.1 Using the program shown in below, explain what the output will be at LINE A.



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE A */
      return 0;
   }
}
```

Since a fork creates a new process with its own global variables, the `value` variable of the parent process is unaffected by the child process. The output value will thus be 5.

3.2 Including the initial parent process, how many processes are created by the following program?



```
#include <stdio.h>
#include <unistd.h>

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

After each fork, the number of processes doubles, making the total number of processes 8.

3.3 Original versions of Apple's mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

- **Scheduling**
  The CPU scheduler must be aware of the different concurrent processes and must choose an appropriate algorithm that schedules the concurrent processes.
- **Inter-process communication**
  Concurrent processes may need to communicate with one another, and the operating system must therefore develop one or more methods for providing inter-process communication.
- **Memory management**
  Because mobile devices often have limited memory, a process that manages memory poorly will have an overall negative impact on other concurrent processes. The operating system must therefore manage memory to support multiple concurrent processes.

3.4 Some computer systems provide multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

The CPU current-register-set pointer is changed to point to the set containing the new context, which takes very little time. If the context is in memory, one of the contexts in a register set must be chosen and be moved to memory, and the new context must be loaded from memory into the set.

3.5 When a process creates a new process using the `fork()` operation, which of the following states is shared between the parent process and the child process?

a. Stack
b. Heap
c. Shared memory segments

Only the shared memory segments are shared between the parent process and the newly forked child process. Copies of the stack and the heap are made for the newly created process.

3.8 Describe the actions taken by a kernel to context-switch between processes.

1. All context switches are initiated by an interrupt.
2. The kernel performs a context save to save the context (CPU registers, state etc.) of the currently executing process into the PCB.
3. The context of the new process is loaded.

This pipeline may also include the use of a scheduler to decide which process is to be switched in.