Operating Systems (02340123)
Summary - Spring 2025

Razi & Yara

June 6, 2025

Contents

I Lectures & Tutorials **3**

Lecture 1: Introduction **4**

Lecture 2: Processes & Signals **6**

II Overall Summary **8**

III Highlights and Notes **9**

Part I

Lectures & Tutorials

Lecture 1: Introduction

Definition (Operating System (OS))**.** *An Operating System's job is:*

- *Coordinate the execution of all SW, mainly user apps.*
- *Provide various common services needed by users & apps.*
- *An OS of a physical server controls its physical devices, e.g. CPU, memory, disks, etc.*
- *An OS of a virtual server only believes it does. There's another OS underneath, called* ***hypervisor*** *which fakes it.*

Using an OS allows us to take advantage of ***"virtualization"***:

- **Server Consolidation:** Run multiple servers on one physical server. This is allows for better resource utilization, smaller spaces, and less power consumption.
- **Disentangling SW from HW:** allows for backing up/restoring, live migration, and HW upgrade. This give us the advantage of easier provisioning of new (virtual) servers = "virtual machines", and easier OS-level development and testing.

Most importantly, an OS is **reactive**, "event-driven" system, which means it waits for events to happen and then reacts to them. This is in contrast to typical programs which run from start to end without waiting for external events to occur to invoke them.

	Typical Programs	**OS**
what does it typically do?	get some input, do some processing, produce output, terminate	waits & reacts to "events"
structure	has a `main` function, which is (more or less) the entry point	no `main`; multiple entry points, one per event
termination	end of main	power shutdown
typical goal	$\sim$ finish as soon as possible	handle events as quickly as possible $\Rightarrow$ more time for apps to run

OS events can be classified into two:

- **Asynchronous interrupts:** keyboard, mouse, network, disk, etc. These are events that can happen at any time and the OS must be ready to handle them.

- **Synchronous:** system calls, divide by zero, page faults, etc. These are events that happen as a result of the program's execution and the OS must handle them immediately.

Definition (**Multiplexing**)**.** *Multiplexing is the ability of an OS to share a single resource (e.g. CPU, memory, disk) among multiple processes or threads. This allows for better utilization of resources and enables multiple applications to run concurrently. *Multiprogramming means multiplexing the CPU recourse.*

Notable services provided by an OS:

1. **Isolation:** Allow multiple processes to coexist using the same resources without stepping on each other's toes. Usually achieved by multiplexing the CPU, memory, and other resource done by the OS. However, some physical resources know how to multiplex themselves, e.g. network cards, sometimes called *"self-virtualizing devices"*.
2. **Abstraction:** Provides convenience & portability by:
   - offering more meaningful, higher-level interfaces
   - hiding HW details, making interaction wiht HW easier.

Lecture 2: Processes & Signals

Processes

Each process is an instance of a program in execution, which includes:

- **Program code:** The actual code of the program.
- **Process state:** The current state of the process, including the program counter, registers, and memory management information.
- **Process control block (PCB):** A data structure used by the OS to manage the process, containing information such as process ID, process state, CPU registers, memory management information, and I/O status information.
  A process doesn't have direct access to its PCB, it is managed by the OS, i.e. needs privilege level 0 (kernel mode) to access it.
- **Process resources:** The resources allocated to the process, such as memory, file descriptors, and network sockets.

Definition (**Process State**). *A process can be in one of the following states:*

- ***Running:*** *The process is currently being executed by the CPU.*
- ***Ready:*** *The process is ready to be executed but is waiting for the CPU to become available.*
- ***Waiting:*** *The process is waiting for an event to occur, such as I/O completion or a signal.*
- ***Zombie:*** *The process has terminated but its PCB is still in the system, waiting for the parent process to read its exit status. In this state, the process has released almost all of its resources, but the PCB is still in the system.*

As we saw in "ATAM", each process can only access a certain set of utilites and functions, those who require privilege level 3 (user mode). So to access the OS services, a process must use **system calls** which are functions provided by the OS that allow processes to request services from the OS. System calls are typically implemented in the OS kernel and provide a controlled interface for processes to interact with the OS.

Each *syscall*, in case of an error, will change the `errno` variable to indicate the error type. The `errno` variable is a global variable that is set by system calls and some library functions in the event of an error to indicate what went wrong. It is defined in the header file `errno.h`.
Note: `errno` is not reset to 0 after a successful syscall, so it must be checked immediately after the syscall, and be reset before usage if need be (if there is not any other way to make sure there is an error indeed).
i.e. `errno=`

Listing 1: Example of using `errno` in C

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
int main() {
    FILE *file = fopen("non_existent_file.txt", "r");
    if (file == NULL) {
        // An error occurred, print the error message
        printf("Error opening file: %s\n", strerror(errno));
        return 1;
    }
    fclose(file);
    return 0;
}
```

Part II

Overall Summary

Part III

Highlights and Notes