Operating Systems (02340123)
Summary - Spring 2025

Razi & Yara

June 7, 2025

Contents

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

Event Synchronousation 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.
- **Process ID (PID):** A unique identifier assigned to each process by the OS. The PID is used by the OS to manage the process and is used in system calls to refer to the process.

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 = <Number of Last Syscall Error>;`

As noted above, each process must be `wait()`ed for by its parent process to be able to release its PCB and resources. This is done by the `wait()` syscall, which suspends the calling process until one of its children terminates. In case the parent process terminates before the child, the child process becomes an orphan process and is adopted by the init process (PID 1), which will then wait for it to terminate and release its resources.

Process Management The OS offers various system calls to manage processes, including: (More details in the functions reference)

- `fork()`: Creates a new process by duplicating the calling process. The new process is called the child process, and the calling process is called the parent process.
- `exec()`: Replaces the current process image with a new process image, effectively running a different program in the same process.
- `wait()`: Suspends the calling process until one of its children terminates.
- `exit()`: Terminates the calling process and releases its resources.
- `getpid()`: Returns the process ID of the calling process.
- `getppid()`: Returns the process ID of the parent process.
- `kill()`: Sends a signal to a process, which can be used to terminate or suspend the process.

Parent Vs. Real Parent Process The real parent process is the one that created the current process using `fork()`, or the one that adopted it in case the real parent terminated before the child.

The parent process is the one *tracing* the current process, e.g. using `ptrace()`. The parent process is the one that will receive signals from the current process, e.g. `SIGCHLD` when the current process terminates.

In most cases, the parent process is the real parent process, but it can be different in some cases, e.g. when a process is being traced by a debugger.

Definition (**Daemon Process**). *A daemon process is a background process not controlled by the user. To run a process as a daemon use* `nohup <command> &`.

Daemon names usually end with the letter "d", e.g. `sshd` *(SSH daemon),* `httpd` *(HTTP daemon), etc.*

Signals

Definition (Signal). *Signals are "notifications" sent to a process to asynchronously notify it that some event has occurred.*

** Receiving a signal only occurs then returning from kernel mode, which in turn invokes the corresponding signal handler.*

*** Default signal handling actions: Either die or ignore*

**** In case of several signals from different types, they will be handled by the order of their definition in the signals register.*

Each signal has a name, a number, and a default action. All but 3 signals can be blocked, i.e. ignored until the process is ready to handle them. The 3 signals that cannot be blocked are:

- `SIGKILL`: Used to forcefully terminate a process. (Process becomes a zombie)
- `SIGSTOP`: Used to suspend the receiving process. (Make it sleep) The signals is sent when the user presses Ctrl+Z in the terminal. Note: In truth Ctrl+Z sends the `SIGTSTP` signal, however, we don't learn about the differences between the two signals in this course.
- `SIGCONT`: Used to resume a suspended process, usually sent after a `SIGSTOP` signal. The handler for this signal can be customized but it will always resume the process.

`SIGSTOP` and `SIGCONT` are useful for debugging purposes, allowing the user to pause and resume the execution of a process.

Signal Handling A process can define a custom signal handler for a specific signal using the `signal()` or `sigaction()` preferred system calls. To ignore a signal, the process can set its handler to `SIG_IGN`. To restore the default action for a signal, the process can set its handler to `SIG_DFLT`.

Signal Masking A process can block signals using the `sigprocmask()` system call, which allows the process to specify a set of signals to block. This is allows the process to overcome *Race Conditions* resulted from the asynchronous nature of signals.

This is achieved by maintaining a set of currently blocked signals & a set of masked signals which is saved in the PCB.
`Blocked Signals` = mask array.
`Pending Signals` = signals that were sent to the process while it was blocked, and will be handled when the process unblocks them.

Common Signals the following are some of the most common signals:

1. `SIGSEGV, SIGBUS, SIGIILL, SIGFPE`: These are driven by the associated (HW) interrupts - The OS gets the associated interrupt, then the OS interrupt handler sees to it that the misbehaving process gets the associated signal, lastly the signaly handler is invoked.

   - `SIGSEGV`: Segmentation violation (illegal memory reference, e.g., outside an array).
   - `SIGBUS`: Dereference invalid address (null/misaligned, assume it's like SEGV).
   - `SIGIILL`: Illegal instruction (trying to invoke privileged instruction).
   - `SIGFPE`: Floating-point exception (despite the name, *all* arithmetic errors, not just floating point. e.g., division by zero).

2. `SIGCHLD`: Parent (not real parent) get it whenver `fork()`ed child terminates or is `SIGSTOP`-ed.

3. `SIGALRM`: Get a signal after some specified time, can be set using the `alarm()` & `setitimer()` system calls.

4. `SIGTRAP`: When debuggin/single-stepping a process, the debugger can set a breakpoint in the code, which will cause the process to receive a `SIGTRAP` signal when it reaches that point.

5. `SIGUSR1, SIGUSR2`: User-defined signals, user can decide the meaning of these signals and their handlers.

6. `SIGPIPE`: Write to pipe with no readers.

7. `SIGINT`: Sent when the user presses Ctrl+C in the terminal. The default action is to terminate the process, but it can be customized.

8. `SIGXCPU`: Delivered when a process used up more CPU than its soft-limit allows: soft-/hard limits are set using the `setrlimit()` system call. Soft-limits warn the process its about to exceed the hard-limit, Exceeding the hard-limit will cause `SIGKILL` to be sent to the process.

9. `SIGIO`: Can configure file descriptors such that a signal will be delivered whenever some I/O is ready.

   Typically makes sense when also configuring the file descriptor to be *non-blocking*, e.g., when `read()`ing from a non-blocking file descriptor, the system call immediately returns to user if there's currently nothing to read. In this case, `errno` will be set to `EAGAIN=EWOULDBLOCK`.

Signals Vs. Interrupts

	interrupts	**signals**
Who triggers them? Who defines their meaning?	Hardware: CPU cores (sync) & other devices (async)	Software (OS), HW is unaware
Who handles them? Who (un)blocks them?	OS	processes
When do they occur?	Both synchronously & asynchronously	Likewise, but, technically, invoked when returning from kernel to user

Lecture 3: IPC - Inter-Process Communication

Part II

Overall Summary

Part III

Highlights and Notes