Operating Systems (02340123)
Summary - Spring 2025

Razi & Yara

June 12, 2025

Contents

Part I

Lectures & Tutorials

Topic 1: Introduction

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.

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.

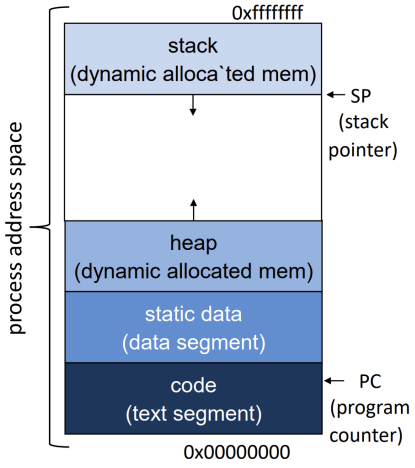
Notable services provided by an OS:

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.

Topic 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 (kernel space), i.e. needs privilege level 0 (kernel mode) to access it.

  Each PCB contains: `real_parent`, `parent`, `children`, `siblings`,...
  Each process has a `task_struct current` pointer to its PCB.
- **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.

Process States 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.

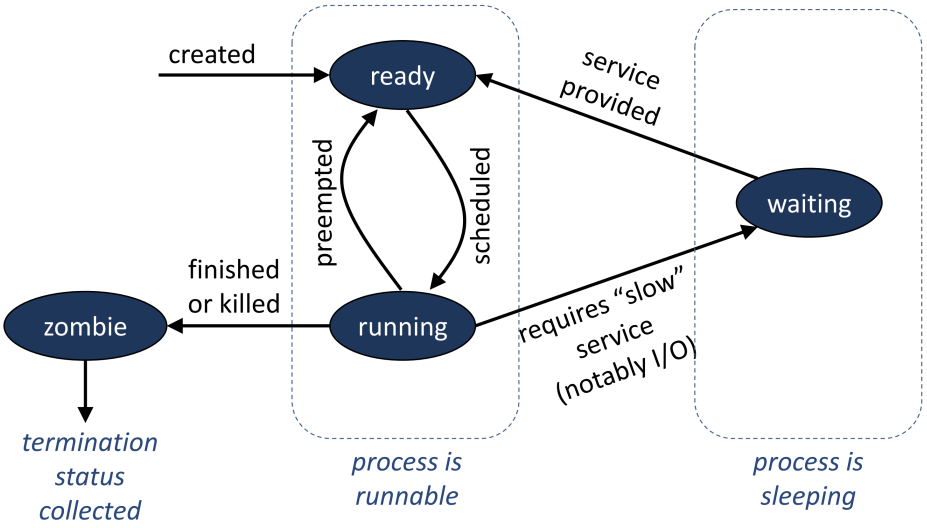


Figure 1: Process States

- **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.

Daemon Processes 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

Signals 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`.

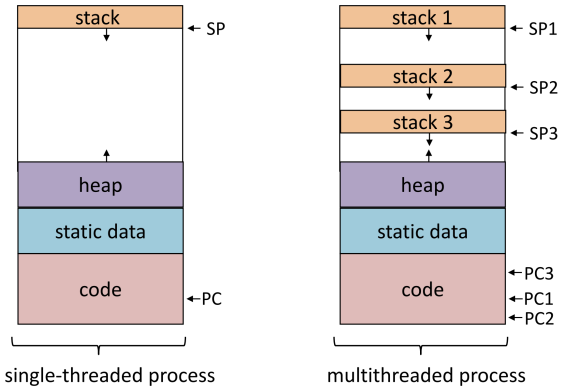
Signals Vs. Interrupts

	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

Topic 3: IPC - Inter-Process Communication

Threads & IPC

Multiprocessing Using several CPU cores (=processors) for running a single job to solve a single "problem".



Threads A process can have multiple threads, which share (nearly) everything, including the process's memory space, file descriptors, heap, static data, code segment and more.

However, each thread has its own stack, registers, and program counter. But they can still access each other's stack since they share the same memory space.

Note that *Global/Static Variables & Dynamically Allocated Memory* are shared between threads since they are stored in the Static Data Segment and the Heap, respectively, which are shared between all threads of a process. However, *Local Variables* are not shared between threads since they are stored in the stack, which is unique to each thread.

	Unique to Process	Unique to pthread
Registers (notably PC)	Y	Y
Execution stack	Y	Y
Memory address space	Y	N
Open files	Y	N
Per-open-file position (=offset)	Y	N
Working directory	Y	N
User/group credentials	Y	N
Signal handling	Y	N

OpenMP Open Multi-Processing (OpenMP) consists of a set of compiler *'pragma'* directives that allows the compiler to generate multi-threaded code for parallel execution.

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
        arr[i] = 2*i;
}
```

Pthreads POSIX threads (pthreads) is a standard for multi-threading in C/C++. It provides a set of functions to create and manage threads, as well as to synchronize them. (More details in the functions reference)

File Descriptors A non-negative integer representing an I/O "channel" on some device. File descriptors are saved in the process's `PCB` inside the `FDT` (File Descriptor Table), which is an array of pointers to *file objects*, each representing an open file or device. So in fact, an FD is an index to a kernel array of channels.

Processes don't share `PCB` nor `FDT` but they can share file descriptors, i.e. two processes can have the same file descriptor pointing to the same file object, which allows them to share the same open file or device. Upon forking, the child process inherits a copy of the parent's `FDT`. Note: Threads do share the same `FDT`.

Pipes A pipe is a unidirectional communication channel between two processes, allowing one process to send data to another. Pipes are a pair of two file descriptors `int pipe_fd[2]`. Each integer is a handle to a kernel communication object, which is a buffer that holds the data being sent between the two processes.

- `pipe_fd[0]` = read side of the communication channel.
- `pipe_fd[1]` = write side of the communication channel.
- Everything written via `pipe_fd[1]` can be read via `pipe_fd[0]`.
- Blocking: If the read side is empty, the read operation will block until data is written to the write side. If the write side is full, the write operation will block until space is available.
- `SIGPIPE`: Writing to a pipe whose read end is `close()`d will result in a `SIGPIPE` signal.

Multi-tasking	Multi-programming	Multi-processing
Having multiple processes *time slice* on the same CPU core.	Having *multiple jobs* in the system (either on the same core or on different cores). i.e. the existence of multiple processes in the system, regardless of whether they are running on the same core or not.	Using *multiple processors (CPU cores) for the same job* in parallel. i.e. creating multiple threads to run on different cores for the same process.

Intro. Context Switching & Caching

Context Switching is the process of saving the state of a currently running process and restoring the state of another process to allow it to run. This is done by the OS kernel and is necessary for multitasking, allowing multiple processes to share the CPU.

- **Context** = the state of a process, including its registers, program counter, stack pointer, and memory management information.
- **Context Switch** = the process of saving the context of the currently running process and restoring the context of another process.

Context Switch Overhead consists of two components:

- **Direct Overhead:** The measurable time it takes to perform the context switch, which includes saving the current process's state and restoring the next process's state.
- **Indirect Overhead:** The time it takes for the CPU to cache the new process's data, which can be significant if the new process's data is not already in the CPU cache.

Cashing The CPU cache is a small, fast memory that stores frequently accessed data to speed up access times. The Cache allows the CPU access to a fast memory that is closer to the CPU than the main memory (DRAM), and a larger one than the CPU registers. The cache works because of the *Principle of Locality*:

- **Temporal Locality:** If at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again soon.
- **Spatial Locality:** If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced soon.

Copy-on-Write (COW) The `fork()` system call creates a copy of the address space of the parent, but:

- It only creates a *logical* copy.
- There is no physical duplication of the memory pages, until we really need to write to them (from either process). And even then, only the page that is being written to is duplicated. (More details in the virutal memory lectures)

User Level Threads (ULTs) are threads that are managed by the user-level library, rather than the OS kernel. ULTs are not visible to the OS, which means that the OS does not know about them and does not schedule them. This allows for faster context switching between ULTs, but it also means that the OS cannot take advantage of multiple CPU cores to run ULTs in parallel.

The usage of ULTs is mainly for concurrent programming, where we want multiple multiple tasks to progress in parallel without the overhead of kernel-level threads.

Topic 4: Scheduling

Part II

Overall Summary

Part III

Highlights and Notes