1 Introduction

Definition 1.0.1 ▶ Von Neumann architecture

The ***Von Neumann architecture*** describes a computer architecture that fetches, decodes, and executes instructions from memory.

In the Von Neumann architecture, a program is composed of many instructions. The central computing unit (CPU) executes each instruction one after the other until the program completes.

Definition 1.0.2 ▶ Operating system (OS), virtualization

An ***operating system*** is some software that manages a system's resources and makes it easy to run programs. Generally, it creates abstractions of physical resources such as the processor or memory—a technique called ***virtualization***.

Virtualization allows an operating system to efficiently and easily manage resources as well as providing concurrency to processes. Concurrency is one of the most difficult things to get right, especially with the advent of multicore CPUs.

2 Virtualization

2.1 Processes

Definition 2.1.1 ▶ Process, process state

A ***process*** is an OS abstraction of a running program. A process can be described by its ***state***, which includes:

- memory or ***address space*** which contains the program's instructions and data that the program reads from and writes to;
- registers which many instructions must read from or write to, as well as specialized registers such as the program counter, stack pointer, and frame pointer[a]; and
- I/O information that specifies access to persistent storage devices.

[a] The program counter specifies the very next instruction for the program to execute. A stack pointer and frame pointer are used to manage the process' stack memory.

Nowadays, our computers are often running tens if not hundreds of processes concurrently, despite lacking tens or hundreds of CPU cores. The OS presents the illusion of many CPUs through virtualization.

Definition 2.1.2 ▶ Time sharing, space sharing

Time sharing is a basic technique used by an OS to share a resource. It involves rapidly switching between executing different concurrent processes, allowing for the illusion of concurrency at the cost of performance as more processes are being run. ***Space sharing*** simply divides a resource among the processes that use it (e.g. disk space).

The OS does not arbitrarily choose which process it will start and stop running; it has a ***scheduling policy*** which enforces time sharing decisions, which we discuss later.

Process API The operating system makes available an application programming interface (API) to manage processes. The ***process API*** includes functionality for

- creating new processes,

- destroying running processes,
- waiting for a process to finish running,
- suspending and resuming processes,
- obtaining status information about a process, and
- other miscellaneous control features.

We discuss this in more detail in Section 2.2.

Process Creation To create a process, an OS:

1. loads its code and any static data (e.g. initialized variables) into memory as the address space for the process[1];
2. allocates memory for the process' ***stack***, used for local variables, function parameters, and return addresses, as well as arguments such as the `main()` function's `argc` and `argv`;
3. may allocate memory for the process' ***heap***;
4. does I/O setup, such as opening file descriptors for stdin, stdout, and stderr; and finally
5. starts the program execution at the entry point `main()`, transfering control of the CPU to the newly-created process.

Process States A process can be in the following states:

1. ***Running:*** The processor is executing instructions of the program.
2. ***Ready:*** A process is ready to run, but the OS has not chosen to run it yet.
3. ***Blocked:*** A process has performed some operation that requires it to wait until another thing happens first.

Moving a process from ready to running is ***scheduling*** a process; moving from running to ready is ***descheduling*** a process. When a process is blocked, the OS will wait for the desired event to occur, then unblock it by moving it to the ready state.

Deciding whether to move a process to running or ready and vice versa is dictated by an OS scheduler, which will be discussed later.

Some other niche states include:

- ***Initial/Embryo:*** The process is being created by the OS.

[1] Older operating systems tend to load programs eagerly, meaning loading all its code at once. Newer operating systems tend to load code lazily, meaning loading code as needed during execution.

- ***Final/Zombie:*** The process exited but has not yet been cleaned up; useful for a parent process to see if the child process executed successfully. The parent calls `wait()` to wait for the child process to finish and to indicate to the OS that it can clean up the child process.

Data Structures To keep track of processes, the OS maintains a list of processes (called a ***process list***) as well as information to keep track of which processes are running and bookkeeping for blocked processes.

The OS stores a couple pieces of important information for each process in what's called a ***process control block (PCB)***. This is composed of:

- a ***register context*** which holds the contents of a register right before the process is moved from running to ready,
- information about the process memory, such as where it starts and the size of the memory,
- process state and a unique identifier (PID),
- a pointer to the PCB of the parent (if applicable),
- a list of open files for the process,
- the current working directory of the process, and
- much much more (depending on the operating system).

More things from the proc struct?

2.2 Process API

Fork, exec, wait system calls

API Design Motivations The separation of `fork()` and `exec()` is essential to UNIX shells, enabling the shell to run code after forking and before execing. For example, in

2.3 Limited Direct Execution

The OS gives users the illusion of concurrency by switching execution time between active processes, a kind of virtualization called ***time-sharing***. With this technique comes some pertinent

design challenges:

- **Performance:** how can we implement virtualization without adding excessive overhead to the system?
- **Control:** how can we run processes efficiently while retaining control over the CPU?

Limited Direct Execution The simplest approach is simply running processes natively on hardware CPU. In short, when we want to start a process, the OS goes through the following steps:

1. Create entry for process list.
2. Allocate memory for program.
3. Load program into memory.
4. Set up stack with argc/argv.
5. Clear registers.
6. Execute call main(), in which the program takes over control and runs main until it returns.
7. After the program returns, free memory of process.
8. Remove from process list.

This approach is straightforward and has the advantage of being fast. However, it poses two problems:

1. The OS gives full trust to the program to run successfully without doing anything we don't want it to do.
2. The OS cannot suspend the process while it's running, so we cannot implement any kind of time-sharing technique.

Restricted Operations and System Calls To prevent processes from doing things we don't want them to do, we introduce two modes of execution: ***user mode*** and ***kernel mode***.

Definition 2.3.1 ▶ User mode, kernel mode

User mode and ***kernel mode*** describe modes of operation for processes.

- Code that runs in ***user mode*** has restricted capabilities. For example, user mode processes cannot issue I/O requests.
- Code that runs in ***kernel mode*** has unrestricted capabilities, allowed to do anything.

When a user process wants to perform a restricted operation, the kernel offers a set of ***system calls*** to user processes.

- These carefully expose certain key pieces of functionality to user processes, such as I/O operations, creating/killing/communicating with processes, allocating memory, and much more.
- All system calls start with a ***trap*** instruction which transfers control to the kernel and raises privilege level to kernel mode. Once done, the kernel issues a ***return-from-trap*** instruction which returns into the calling program and deescalate privilege to user mode.
- Before executing the trap instruction, the processor saves a copy of the sensitive registers onto a per-process ***kernel stack***[2]. When the system call returns, the original registers get restored.
- Each system call has a corresponding number, which is handled by the OS in some functionality called the ***trap handler*** to jump into the right portion of kernel code. This protects the kernel code from being arbitrarily jumped into.
- This set of correct system call numbers is established when the OS boots and is only possible in kernel mode.

There are two distinct phases in limited direct execution:

1. At boot time, the kernel initializes the trap table which gets saved by the CPU for future use. This is done by an instruction only available in kernel mode.
2. When running a process, sets up some bookkeeping info for a process and issues a return-from-trap to start running the process. If the process issues a system call, it "traps" back to the OS which handles it and returns via a return-from-trap instruction. The process ends by returning from `main()`, which usually returns into some end code such as a trap back into the OS. From here, the OS cleans up the dead process.

Figure 6.2 Limited Direct Execution Protocol

Switching Between Processes With all this, we have a way for user processes to invoke kernel functionality. However, how can the kernel voluntarily suspend a process to switch to another? There are a few approaches.

[2]The kernel stack is used exclusively when invoking kernel-mode instructions, like trap. The user stack is responsible for local variables, function parameters, return addresses, etc. The user stack is generally just called "stack".

Cooperative Approach If the OS is willing to trust processes to behave reasonably, then we could just hope that processes voluntarily relinquish control, whether it be through system calls or trying to invoke an illegal operation (e.g. divide by zero, which generates a trap instruction). From there, the OS can decide to switch to another process. In this approach, we can offer processes a ***yield*** system call. All it does is relinquish control of the current process, letting the OS decide what to do from there.

Obviously, this is a flawed approach. A process can simply invoke an infinite loop, never giving up control of the processor.

Non-cooperative Approach The OS can reclaim control at its own will by interrupting the running process. This obviously cannot be done in software alone, so we introduce a hardware timer that interrupts the processor at a fixed interval. When this interrupt happens, the running process is halted, and the OS runs a pre-configured interrupt handler. From here, the OS can decide to switch to another process. This interrupt handler is configured at boot time, a privileged operation similar to establishing the trap table.

Context Switch In practice, the OS regains control through both cooperative and non-cooperative approaches. Once it does so, a scheduler decides whether to keep running the same process or switch to another. If it does choose to switch, it performs what is called a ***context switch***.

Definition 2.3.2 ▶ Context switch

A ***context switch*** is a low-level piece of code run by the OS when it switches which process it runs. In it, the OS saves important register values for the previous process (e.g. on its kernel stack) and restore a few register values for the next process (from its kernel stack).

The context switch is crucial in creating smooth transitions when switching between processes. Some registers that get saved include general purpose registers, program counter (PC), and stack pointers of the current process.

There are two kinds of context switches, depending on how they're invoked:

A. When a timer interrupt happens, the hardware implicitly saves the user registers of the running process into the kernel stack of the running process.

B. When the OS decides to switch to another process, the OS explicitly saves the kernel registers into the process structure of the previous process.

Concurrency It's possible that an interrupt can happen while the OS is already handling an interrupt. A simple fix is to just disable interrupts during interrupt processing. A downside is that disabling interrupts for too long can lead to lost interrupts. There are other solutions, such as some locking mechanisms to protect concurrent access. We discuss this more later.

2.4 Scheduling

Definition 2.4.1 ▶ Preemptive, non-preemptive

A scheduling policy is ***preemptive*** if it can stop the currently running process in favor of starting/resuming another process (called a context switch). A scheduling policy is ***non-preemptive*** if it always executes the currently running process to completion, never interrupting it.

Nearly all modern operating systems use a preemptive scheduling policy. Non-preemptive scheduling policies are prone to ***starvation***, where a process can wait an arbitrary amount of time before starting.

Workload Assumptions The operating system must enforce a scheduling policy to determine which processes it should execute at any given time. These scheduling policies almost always make choices based on the current active processes or ***jobs***, referred to collectively as the ***workload***.

To simplify our approach, we will consider a workload in which:

1. Each job runs for the same duration.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU and perform no I/O.
5. The run-time of each job is deterministic.

It will be unrealistic to assume that practical workloads follows these assumptions.

Scheduling Metrics We compare different scheduling policies by ***scheduling metrics***. Some common ones include:

- ***Turnaround time:*** the time between when the job first arrives and when the job finishes.

- ***Response time:*** the time between when the job first arrives and when the job first starts running.
- ***Fairness:*** ensuring every job gets a good chance to run.

We call turnaround time and response time ***performance*** metrics.

First In, First Out (FIFO) The most basic scheduling policy is ***First In, First Out (FIFO)***. As the name suggests, the very first job gets fully executed, and any jobs that arrive must wait in a queue to get executed. As such, it is a non-preemptive scheduling policy.

Pros	**Cons**
Incredibly simple to understand and implement.	***Convoy Effect:*** Long-duration jobs will force long waits for future jobs, yielding bad turnaround times and bad response times.

Shortest Job First (SJF) This non-preemptive scheduling policy solves the convoy effect present in FIFO. It runs the shortest job first, then the next shortest, and so on. This works fine if all the jobs come at the same time, but can go bad if a big job arrives before a small job.

Pros	**Cons**
Good turnaround times, so long as jobs all arrive at the same time.	Bad response times, and bad turnaround times if big jobs arrive first.

Shortest Time-to-Completion First (STCF) ***Shortest Time-to-Completion First*** adds preemption to SJF, solving the issues in SJF. That is, if we allow the operating system to interrupt a currently running job, then we can solve the problem in SJF. Whenever a new job arrives, the OS can decide to suspend the currently running job in favor for the new job if the new job will complete faster than the current job.

Pros	**Cons**
Always good turnaround times.	Bad response times.

Round Robin ***Round-Robin (RR)*** scheduling is optimized for response time. Instead of running jobs to completion, RR runs a job for a set ***time slice***, then switches to the next job in the queue. RR performs context switches only on timer-interrupt periods, so the duration of each time slice must be a multiple of the timer-interrupt period. Performing context switches often

can be costly, so the time slices are often larger multiples of the timer-interrupt period (at the cost of reduced response time).

mention amortization in here

Pros	Cons
Absolute best response time.	Terrible turnaround times.

Considering I/O When a process access I/O resources, it will be blocked waiting for the I/O operation to finish. As such, it's best for the scheduler to switch to another job while the current process is waiting for I/O to finish.

In the case of preemptive sheduling, an I/O operation can incur a context switch. Read and write can both cause a process to be blocked. Meanwhile, the OS may schedule another process to run while the I/O job finishes.

Check this fact

MORE!!! overlapping stuff

Shortest CPU Burst First In practice, we can't accurately guess the runtime of jobs. We instead think of a process as a series of ***CPU bursts***, which are periods of execution without any internal events. A CPU burst usually ends when a context switch is necessary, such as an I/O operation.

Shortest CPU Burst First is the best way to approximate SJF. Although we cannot estimate the runtime of an entire job, we can reasonably estimate the runtime of a single CPU burst. We estimate a CPU burst by how long the current CPU burst lasted. The longer it runs, the more likely

finish this

figure out shit

Process Priority:

- Called "niceness" in Linux
- A single number associated with a process
- Simple order ready queue as a priority queue
- Calculate effective priority from base priority and wait time in queue (since last time random) called ***aging***

Underlying theme: minimality; shaped infrastructure for years to come

What is a process? There are many ways to interpret this question:

1. It can be natively executed alongside the operating system

2. It can be interpreted code, where only an interpreter is natively executed (e.g. the Java virtual machine)
3.

The pieces of a process are:

- Memory image (contents of RAM)
- Register contents (both user-visible and OS managed)
- Process control block (an OS controlled data structure in kernel mode)

Inside the process control block (PCB) are:

- A unique integer identifier for the process called a ***process ID***
- A ***state*** which denotes the current state of the process, such as running
- A pointer to the parent process

Process Creation involves:

- Allocation of resources within the OS
- User identity (UID), a field of the process control block
- Group identity (GID), which identifies a set of UIDS
- Limits on resources and billing

In the POSIX standard, a process creates another process by the `fork()` system call. This creates a child process that replicates the parent process, allocating a PCB and memory for the child. It populates the child PCB with copies of parent's fields except for identity (PID) and data structures that are allocated and copied. It also copies memory image and register values from parent to child.

Both the child and parent processes commence execution by returning from the `fork()` system call. In the parent process, `fork()` returns the PID of the child. In the child process, `fork()` returns `0`. This is the definitive way to differentiate between child and parent process, and it should be used to control the flow of logic in your program.

`fork()` allows for asynchronous computation, perhaps letting the child process execute a subroutine call. To synchronize the two processes after the child has done its task, the parent may call `wait()`, which waits for the child process to die in order to continue execution. It will

return the PID of the child process. If there are more than one dead child processes, `wait()` returns the PID of one random child process.

The `init` process is the initial process that creates all other processes. There is a file called an administrative file that gives the init process a list of processes to fork off and start. Init has some special properties:

- `pid` set to 0,
- `uid` set to 0 (which is the root user's `uid`)

Every terminal device attached to the machine has an associated `login` process, which outputs "login" to the terminal, and waits for user input to determine the identity of the user. After this, it changes UID to the respective user.

Memory management:

- control what a process can see and change
-

It's an allocation of resources—a data structure within the operating system. The operating system devotes resources to each process. This is a power that was deemed too sensitive for the user to have control over. Moreover, the creation of processes was often a dangerous task. Because of this, this is something that the operating system takes care of.

The operating system is considered to be out of the hands of the end user. There are several layers that make up the operating system, including:

- Hardware
- Boot loader, kernel, and OS
- Utilities
- User programs

Anything other than user programs was out of the user's control and are deemed "protected" from the user.

In many operating systems before UNIX, there was a supervisory mode of operation which existed between the user and kernel. Nowadays, the POSIX standard has since remove the supervisory mode, giving us user and kernel modes.

OS mechanisms:

- Native execution, available to the bootloader and operating system without memory management.
- Multiple registers contexts in hardware
- Base and limit registers for memory management
-

> Virtual memory is completely implemented in hardware and thus has zero overhead for executing processes.

User mode is a sandbox. Going from kernel to user mode is completely controlled by the kernel. However, to go from user to kernel, we need a special mechanism. There are generally four user-to-kernel transitions. These are to allow the kernel to regain control from the user:

1. Clock ticks: save PC and PCR values to memory, load the PC for a kernel entry point, and load a PCR value for kernel execution. The PC and PCR values are called the ***kernel entry vector***. Then, we save user-accessible registers to the process control block (the `proc` data structure).
2. Hardware interrupts: these are handled just like clock ticks, but are generated from hardware other than the CPU. These can be disabled separated.
3. System calls: an instruction `TRAP` where we load the PC and PCR from entry vector. There is sometimes an argument (syscall number) that indicates the purpose of the system call. The syscall number may index into a list of entry vectors, each corresponding to a different piece of OS code. For example, `fork()` and `read()` have a unique system call number as the argument to `TRAP` which correspond to a specific functionality. It looks like a subroutine call. A system call's arguments are stored on the stack, and the result of the system call is stored in a register.
4.

There are four kernel entry events to transition from executing a user program to executing the OS:

1. Exceptions
2. Clock tick
3. System calls (through a trap instruction)
4. Hardware interrupts

Note that clock ticks and hardware interrupts are not part of the software, rather controlled by the hardware itself. On the other hand, exceptions and system calls are invoked directly by the software itself.

Sandbox mode includes the following controls:

- Create sandbox with memory management and kernel/user mode
- Kernel mode is all-powerful
- We want to minimize resources and privileges of processes (weak, simple, generic, resource-limited)
- Superuser (i.e. root) identity has UID 0. For example, the `init` process (the process creating during system bootstrap) runs as root. The `login` process also runs as root.
-

> Running persistent processes as root is prone to security vulnerabilities (e.g. finger daemon, Cornell worm).

In UNIX, the `exec()` system call takes a string filename, reads that file, puts the corresponding portions into the currently running process' memory, and starts executing the new file. It reuses the same PCB, with the code and data portions being overwritten by whatever is in the new file. Thus, when a process calls `exec()`, it "becomes" a new process, and `exec()` never returns.

The fork-exec sequence seems to be wasteful. (TODO: more). This used to be optimized in the following way. If a process calls `fork()`, it creates a copy of the memory image for the child. We can avoid this copying by just letting the child use the same memory as the parent. This is done through the `vfork()` call, where the parent is immediately blocked, and the child uses the parent's same memory image without copying. This must only be used when the child does not interfere with the parent's memory image, such as when it performs `exec()` immediately afterwards. Nowadays, virtual memory takes care of this overhead in a much safer way.

Pipes are a shared storage resource in kernel (OS) memory. It's used as a buffer between a writing and reading process, shared between those two processes. It is created by only a single running process.

(Page table stuff)

The cache is a small, fast piece of memory. It holds a subset of the Page Table Entires (PTE).

Initially, it is empty. Each PTE used is inserted into cache. On subsequenct accesses to memory, it looks into cache simultaneously with address translation. If found, abort translation (lookaside buffer). Process switch: clear cache.