.

CS 162: Operating Systems and Systems Programming

Lecture Notes

Mokhalad Aljuboori

Fall 2024 — Ion Stoica

Contents

08/29/24

Lecture 1

Introduction

1.1 What is an Operating System?

An operating system is the layer of software that manages a computer's resources for its users and their applications. In other words, the OS is a specail layer of software that enables the applications to run and share the hardware of a system.

OS provides:

1. Convenient abstraction of complex hardware devices.
2. Protected access to shared resources
3. Security and authentication
4. Communication amongst logical entities

How can an operating system run multiple applications? For this, operating systems need to play three roles:

1. **Referee:** Since the applications share physical resources, the operating system needs to decide which application gets what resources and when. This role is somewhat akin to that of a patient kindergarten teacher. It balances needs, separates conflicts, and facilitates sharing.
2. **Illusionist:** OS's needs to provide an abstraction of physical hardware to simplify application design. They need to provide the illusion of nearly infinite memory. Also provide the illusion that each program has the computer's processors entirely to itself. These illusions let you write applications independently of the amount of physical memory on the system or the physical number of processors.
3. **Glue:** OS's needs to provide a set of common services that facilitate sharing among applications. Many OS's provide provide a common user interface routines so applications can have the same "look and feel". Most importantly, the OS provides a layer separating applications from hardware I/O devices like monitors, keyboards, and mouses so applications can be written independently of the specific hardware.

Definition 1.1: Process

A process is an execution enviornment provided by the OS to run a program. A process has its own memory space, its own file descriptors, and its own CPU time. The OS can run multiple processes at the same time, switching between them to give the illusion that they are all running concurrently.

A process consists of:

- Address space
- One or more threads of control executing in that address space.
- Additional system states.

The OS provides access protection between processes. Process 2 cannot access the resources and memory used by Process 1 and vice verse.

Note that a processo**r** can only run one process at a time. So how could it possible run multiple process? The OS uses a technique called context switching, switching between executing different process very fast.

1.2 Operating System Evaluation

Having defined what an operating system does, how should we choose among alternative designs? We discuss several desireable criteria for operating systems:

1. **Reliability and Availability:** Does the operating system do what you want?
2. **Security:** Can the operating system be corrupted by an attacker?
3. **Portability:** Is the operating system easy to move to new hardware platforms?
4. **Performance:** Is the user interface responsive, or does the operating system impose too much overhead?
5. **Adoption:** How many other users are there for this operating system?

In many cases, tradeoffs between these criteria are inevitable — improving a system along one dimension may hurt it along another.

09/03/24

Lecture 2

Four Fundamental OS Concepts

Definition 2.1: Kernel

Implementing protection is the job of the *operating system kernel*. The kernel is the lowest level of software running on the system, and has full access to all of the machine hardware. Therefore the kernel is necessarily *trusted* to do anything with the hardware. Everything else — that is, the untrusted software running on the system — is run in a restricted environment with less than complete access to the full power of the hardware.

2.1 The Process Abstraction

Definition 2.2: Process

You can think of a process as an instance of a program, same way that an object is an instance of a class in OOP. Each process has it's own memory space for program's data, stack, and heap. The process is the abstraction for protected execution provided by the operating system kernel.

A process needs permission from the kernel before accessing the memory of any other process, before reading or writing to the disk, before changing hardware settings, and so forth.

Applications/programs need to run on the processor with all potentially dangerous operations **disabled**. To make this work, hardware needs to support **dual-mode operation**. See next section.

The OS keeps track of the various process on the computer using a data structure called the ***process control block***.

Definition 2.3: Process Control Block (PCB)

The PCB is a data structure that stores all the information the OS needs about a particular process, including:

- Status (Running, ready, blocked, ...)
- Where its stored in memory,
- Where it's executable image resides on disk,
- Which user asked it to start executing,
- What privileges the process has,
- and so forth.

2.2 Dual-Mode Operation

How does the operating system kernel prevent a process from harming other processes or the operating system itself? What prevents a process from overwriting another process's data structures, or even overwriting the operating system image stored on disk?

It's done in hardware, and it's called ***dual-mode operation***. Most instructions are perfectly safe, such as adding two registers together and storing the result in a third register. But some instructions are dangerous, such as reading from or writing to disk, or changing the memory protection settings. The hardware must be able to distinguish between these two types of instructions.

Definition 2.4: Dual-Mode Operation

A single bit in the processor status register is used to signify the current mode of the processor. The mode bit is modified by some instructions. When set to 1, the processor is in **kernel mode**. When set to 0, the processor is in **user mode**.

In **user mode**, the processor checks each instruction before executing it to verify that it is permitted to be performed by that process. We'll describe the specific checks soon.

In **kernel mode**, the OS executes the instruction with protection checks disabled.

The current privilege level is stored as the low-order bits of the cs register in x86 processors rather than in the processor status word (eflags registers).

What hardware is needed to le the operating system kernel protect applications and users from one another, yet also let user code run directly on the processor? At a minimum, the hardware must support the following three things:

1. **Privileged Instructions.** All potentially unsafe instructions are prohibited when executing in user mode.
2. **Memory Protection.** All memory accesses outside of a process's valid memory region are prohibited when executing in user mode.
3. **Timer Interrupts.** Regardless of what the process does, the kernel must have a way to periodically regain control from the current process.

In addition, the hardware must also provide a way to safely transfer control from user mode to kernel mode and back. We'll discuss how this is done in a later section.

2.2.1 Privileged Instructions

Instructions available only in kernel mode are called ***privileged instructions***. The OS kernel must be able to execute these instruction to do its work. Thus, while application programs can use only a subset of the full instruction set, the OS executes in kernel mode with the full power of the hardware.

If an application were to attempt to access restricted memory or attempts to change its privilege level, such actions would cause a *processor exception* to occur. Unlike exceptions in a high-level language, a processor exception causes the processor to transfer control to an exception handler in the OS kernel. Usually, the kernel simply halts the process after a privilege violation.

2.2.2 Memory Protection

To run an application process, both the operating system and the application must be resident in memory at the same time. Further, other application processes may also be stored in memory. They share the same physical memory.

To make memory sharing safe, the OS must be able to configure the hardware so that each application process **can read and write only its own memory**, not the memory of the OS or any other application. Otherwise, an application could modify the operating system kernel's code or data to take control of the system or corrupt other applications.

While it may seem that read-only access to memory is harmless, recall that the OS needs to provide both **security and privacy**. Kernel data structures may contain private user data, and kernel memory may contain sensitive information such as passwords while they are being verified.

2.2.3 Timer Interrupts

Process isolation also requires to provide a way for the operating system kernel to periodically regain control of the processor. This is done using a **timer interrupt**. If the application enters an infinite loop, or if the user simply becomes impatient and wants the system to stop the application, then the operating system must be able to regain control.

The operating system also needs to regain control of the processor in normal operation. To respond to user input in a timely manner while also running other applications, the OS must be able to regain control to switch to a new task. A ***hardware timer*** device is used to generate periodic interrupts after a specified delay (either in time or after some number of instructions have been executed).

When the timer interrupt occurs, the processor stops executing the current process and transfers control from the user process to the kernel running in kernel mode. The kernel can then decide what to do next, such as switching to a new process or continuing to run the current process.

Resetting the timer is a privileged operation, so only the kernel can reset the timer. This prevents an application from disabling the timer and running forever. A timer or other interrupt does not imply that the program has an error.

2.3 Types of Mode Transfers

The question is now, how to safely transition between executing a user process to executing the kernel and vice versa. This mechanism must be both fast and safe, as it is used frequently (a high performance web server could switch between user and kernel thousands of times per second), leaving no room for error.

We'll discuss first transfers from user to kernel mode, as transitioning in the other direction works by "undo"-ing the transition from user process into the kernel.

2.3.1 User to Kernel Mode Transfer

There are three reasons for a user process to transfer control to the kernel. Interrupts, processor exceptions, and system calls.

1. **Interrupts:** An interrupt is an ***asynchronous*** signal to the processor that some external event has occurred that may require the processor's attention. As the processor is executing instructions, it checks for whether an interrupt has arrived. If so, it completes or stalls any instructions in progress, and instead of fetching the next instruction, the processor hardware saves the current execution state and starts executing at a specially designated interrupt handler in the kernel. Each different type of interrupt requires its own handler and there are many types of interrupts.
2. **Processor Exceptions:**
3. **System Calls:** User processes can transition into the operating system kernel voluntarily to request that the kernel perform an operation on the user's behalf. A ***system call*** is any procedure provided by the kernel that can be called from user level.

2.3.2 Kernel to User Mode Transfer

Similar to the user to kernel mode transfer, the kernel to user mode transfer has different types of transitions, namely 4 types:

1. **New Process**: When starting a new process, the following steps are performed:
   (a) Kernel copies the program into memory.
   (b) Sets the program counter to the first instruction of the process
   (c) Sets the stack pointer to the base of the user stack
   (d) And switch to user mode.
2. **Resume after an interrupt, processor exception, or system call**: When the kernel finishes handling the request, it resumes execution of the interrupted process by doing the following:
   (a) Restore the program counter (PC) (in the case of a syscall, the instruction after the trap),
   (b) Restoring its registers,

   (c) Changing the mode bit to user mode.

3. **Switch to a different process**: When the kernel decides to switch to a different process, as in the case of a timer interrupt, it does the following:
   (a) Since the kernel will eventually resume the old process, the kernel needs to save the process state — its PC, registers, and so forth — in the **PCB**.
   (b) The kernel then loads the state of the new process from the PCB.
   (c) and switch to user mode.

4. **User-level upcall**: This is a mechanism where the kernel notifies a user-level process of an event, such as the arrival of a signal or the completion of an asynchronous I/O operation. The steps involved are:
   (a) The kernel sets up the user-level stack with the appropriate context for the upcall.
   (b) The kernel sets the program counter to the user-level upcall handler.
   (c) The kernel switches to user mode, allowing the user-level process to handle the event.

2.4 Implementing Safe Mode Transfer

The context switch code must be carefully crafted to ensure that a buggy or malicious user program cannot corrupt the kernel, and it does this by relying on hardware support.

The common sequences of instructions that the operating system provides for entering the kernel from user mode and returning to user mode, at the minimum must provide:

- **Limited Entry into the Kernel**: When transferring control from user mode to kernel mode, the kernel must ensure that the entry point into the kernel is one *set up by the kernel*. The user programs should not be able to jump to arbitrary locations in the kernel.
- **Atomic Changes to Processor State**: Transitions between user and kernel mode must be **atomic**. That is, the mode, program counter, stack, and memory protection are all changed at the same time, in a way that cannot be interrupted by another process.
- **Transparent, Restartable Execution**: An event may interrupt a user-level process at any point, between any instruction and the next one. The OS must be able to restore the state of the user program exactly as it was before it was interrupted, so that to the user process, the interrupt is “invisible” /transparent and continues executing as if nothing happened.

Lets now describe the hardware/software mechanism for handling an interrupt, processor exception, or system call.

Today, the four Fundamental OS Concepts are

- Thread: Execution context
  - Fully descrie program state
- Address Space (with or w/o translation)
- Process: an instance of a running program
- Dual Mode operation / Protection

The buttom line of an OS is **to run programs**. Here is how they are run:

1. Write and compile them
2. Load instruction and data segments of executable file into memory
3. Create stack and heap
4. "Transfer control to program"
5. Provide services to program

09/05/24

Lecture 3

Abstractions 1: Threads and Processess A quick, programmer's viewpoint

3.1 The Thread Abstraction

Definition 3.1: Thread

A thread is a *single execution sequence* that represents a *separately schedulable task.*

Let us be a explain what these words mean:

- **Single Execution Sequence:** Each thread executes a sequence of instructions — assignments, conditionals, loops, procedures, and so on — just as in the familiar sequential programming model.
- **Separately Schedulable Task:** The operating system can run, susped, or resume a thread at any time.

Threads let us define a set of tasks that run **concurrently** while the code for each task is sequential. Threads are mechanism for concurrency.

Each thread behaves as if it has its own dedicated processor (it's a virtual processor). In reality ofcourse, a machine only has a finite number of professors and it is the OS job to transparently **multiplex** threads onto the actual processors.

The thread abstraction lets the programmer create as many threads as needed without worrying about the exact number of physical processors, or exactly which processor is doing what at each instant.

The code being executed within each thread is unaware when the OS switches between threads. This abstractions makes each thread appear to be a single stream of execution; this means the programmer can pay attention to the sequence of instructions within a thread and not whether or when that sequence may be suspended to let another thread run.

Threads thus provide an execution model in which *each thread runs on a dedicated virtual processor with unpredictable and variable speed.* It is as if the thread were running on a processor that sometimes becomes very slow.

3.1.1 Running, Suspending, and Resuming Threads

Threads provide the illusion of an infinite number of processors. How does the OS implement this illusion? It must execute instructions from each thread so that each thread makes progress, but the underlying hardware has only a limited number of processors.

To map an arbitrary set of threads to a fixed set of processors, OS include a ***thread scheduler*** that can switch between threads that are running and those that are ready but not running.

Example 3.2: Threads Vs. Processes

As we described in previous lectures, a process is an execution of a program with restricted rights. A thread is an independent **sequence of instructions** running within a program.

3.2 Simple Thread API

Figure 3.1 shows a simple API for using threads and is based on the POSIX standard pthreads API.

A good way to understand the simple threads API is that it provides a way to invoke an *asynchronous procedure call.* A normal procedure call passes a set of args to a function, runs the function immediately on the caller's stack, and when the function is completed, returns control back to the caller with the result. An asynchronous procedure call separates the call from the return, allowing the caller to continue running while the function runs in

`void thread_create(thread, func, arg)`	Create a new thread, storing information about it in `thread`. Concurrently with the calling thread, `thread` executes the function `func` with the argument `arg`.
`void thread_yield()`	The calling thread voluntarily gives up the processor to let some other thread(s) run. The scheduler can resume running the calling thread whenever it chooses to do so.
`int thread_join(thread)`	Wait for `thread` to finish if it has not already done so; then return the value passed to `thread_exit` by that thread. Note that `thread_join` may be called only once for each thread.
`void thread_exit(ret)`	Finish the current thread. Store the value `ret` in the current thread's data structure. If another thread is already waiting in a call to `thread_join`, resume it.

Figure 3.1: Simplified API for using threads.

the background. With `thread_create`, the caller starts the function, but unlike a normal procedure call, the caller continues execution concurrently with the called function. Later, the caller can wait for the function to complete with `thread_join`.

Definition 3.3: Asynchronous Procedure Call

An *Asynchronous Procedure Call (APC)* is a mechanism by which a function or task is executed asynchronously, meaning it runs separately from the main thread of execution. This allows the main program to continue running while the procedure runs in the background, without blocking or waiting for the procedure to complete. APCs are typically used in multitasking environments to improve responsiveness and efficiency by allowing multiple tasks to execute concurrently.

LEFT AT 4.4 TO READ ON SYNCHRONIZATION

09/10/24

Lecture 4

Abstractions 2: Processes, Files and I/O

There are 3 types of Kernel Mode Transfers.

1. Syscall.
2. Interrupts
3. Trap or Exception, eg. Seg fault.

4.0.1 Creating Processes

09/12/24

Lecture 5

Abstractions 3: Files and I/O (cont'd), Sockets and IPC

09/17/24

Lecture 6

Abstractions 4: Sockets, I/O, and IPC

09/19/24

Lecture 7

Concurrency, Mutal Exclusion, and Atoc Operations

Most multi-threaded programs have both *per-thread state* (e.g., a thread's stack and registers) and a *shared state* (e.g., global variables, heap memory, and file descriptors). ***Cooperating threads*** read and write shared state.

7.1 Challenges

The core challenge of multi-threaded programming is that the program's execution depends on the interleavings of different threads' access to shared memory, which can make it difficult to reason about or debug these problems because there is a combinatorial explosion in the number of possible interleavings as the program grows.

7.1.1 Race Conditions

In particular, cooperating thread's execution may be affected by ***race conditions***.

Definition 7.1: Race Condition

A race condition occurs when the behavior of a program depends on the interleaving of operations of different threads. In effect, the threads run a race between their operations, and the results of the program execution depends on who wins the race.

Example 7.2: a race condition

Suppose that initially $x = 0$ and we run a program with two threads that do the following:

Thread A	**Thread B**
$x = x + 1;$	$x = x + 2;$

Question: What are the possible final values of x?

Answer: Obviously, **one possible outcome is** $x = 3$. For example, Thread A runs to completion and then Thread B starts and runs to completion. However, **we can also get** $x = 2$ **or** $x = 1$. In particular, when we write a single statement like $x = x + 1$, compilers on many processors produce multiple instructions, such as:

1. Load memory location x into a register.
2. Add 1 to that register.
3. Store the result to memory location x.

If we disassemble the above program into simple pseudo-assembly code, we can see some of the possibilities.

Final Value: x == 3		**Final Value: x == 2**		**Final Value: x ==**	
Thread A	**Thread B**	**Thread A**	**Thread B**	**Thread A**	**Thread B**
load r1, x		load r1, x		load r1, x	
add r2, r1, 1			load r1, x		load r1, x
store x, r2		add r2, r1, 1		add r2, r1, 1	
	load r1, x		add r2, r1, 2		add r2, r1, 2
	add r2, r1, 2	store x, r2			store x, r2
	store x, r2		store x, r2	store x, r2	

Not only would one have to reason about all possible interleavings of statements, but one would also have to disassemble the program and reason about all possible interleavings of assembly instructions (And if the compiler and hardware can reorder instructions, then there are even more possibilities to consider).

7.1.2 Atomic Operations

Definition 7.3: Atomic Operations

An atomic operation is an indivisible operation that cannot be interleaved with or split by other operations. In other words, an atomic operation is a single operation that appears to occur instantaneously from the perspective of other threads.

7.2 Structured Shared Objects

Definition 7.4: Shared Objects

Shared objects are objects that can be accessed safely by multiple threads. All shared state in a program — including variables allocated on the heap and static, global variables — should be encapsulated in one or more shared objects.