Contents

Chapter 1

Kernel Abstraction

A central role of an operating system is protection.

Reliablity: prevent bugs from another program to interfer.

Security: protection against malicious software/programs.

Privacy: prevent untrusted code from accessing unauthorized data.

Efficiency: allocate resources effectively to ensure fairness.

The kernel is the lowest level software with full access to all the capabilites of hardware. It is necessarily trusted.

A process is an abstraction for protection provided by operating system kernel: The execution of an application program with restriced rights. Needs premission to read and write.

1.1 The process abstraction

A code is compiled by the compiler which returns the program's *executable image.* Note that, the compiler also includes any static data the program needs in the executable image. Operating system copies the executable image into the memory and sets aside *stack* for local variable and procedure calls and *heap* for dynamically allocated data structures. Lastly, operating systems sets stack pointed to the first instruction.

A process is an instance of a program executing with restriced rights. The operating system keeps track of varius processes using a structure called *process control unit* (PCB). PCB includes the following information

- memory address of the process.

- executable image on the disc.

- the user that initiated the process.

- the privileges the process has.

- etc. :)

1.2 Dual-mode operation

Implements checks in hardware whether to run an instruction. *Dual-mode operation* is a single bit in the processor status registes to signify in which mode the processor is currently running.

- In user mode, it checks each instruction before executing,
- In kernel mode, it executes without checking.

This hardware must support at least three things:

Privileged instructions: All potentially unsafe instructions are prohibited in user mode.

Memory protection: All memory access outside the processor's valid memory regions are prohibited in user mode.

Timer interrupts: The kernel must be able to tkae control periodically.

In addition, hardware must provide safe transfer control from user to kernel and back.

1.2.1 Privileged instructions

A process should not be able change its privilege level directly. Although, it can transfer control into the kernel. Hence, it can indirectly changes its privilege level by executing *system calls.*

Other instructions are also limited to use by kernel code. For example, an application shoud not be able to change the set of memory locations it can access. And disabling and enabling interrupts is another example of *privileged instructions* –can be run only in kernel mode. An attempt to execute a privileged instruction results in a *process exception.*

1.2.2 Memory protection

Both kernel and the user process must be in memory in order to run an application. Therefore, kernel must provide a method of memory sharing that ensures privacy and security.

A simple method to implement such protection is to use two extra registers called *base* and *bound.* Theb base register specifies the starting memory location and bound specifies the ending memory location.

- These registers can only be changed by privileged instructions.
- Processor proceeds running an instruction that wants to access memory if the address is between base and bound. Otherwise, the hardware raises an exception and control is transferred back to the kernel.
- Some disadvantages of this method are
  - fixed size heap and stack.
  - no sharing between processes.
  - set the global adddress each time the program is loaded.

  - impossible to relocate the process. After some time, memory become fragmented and even though there is enough space in aggregate, there is no large enough contiguous space.

To fix these issues, processors use *virtual addresses*. Basically, each process' virtual address starts at zero and then it is mapped by hardware into a physical memory location; e.g. by adding the base register.

add the C code on page 67.

1.2.3 Timer interrupts

Operating system must periodically regain control of the processor. Almost all computer systems include a device called *hardware timer*. It is set to interrupt the processor after a speicified delay. Resetting the timer is a privileged operation hence, only kernel is able to set it. When the timer interrupt occurs, the hardware transfers control from the user process to the kernel running in the kernel mode, similar to other types of hardware interrupts.

1.3 Types of mode transfer

1.3.1 User to kernel mode

There are three ways to transfer from user to kernel mode; *interrupts, exceptions, and system calls*. *Trap* refers to any synchronous transfer control from user to kernel. Exceptions and system calls are traps :).

Interrupts: an asynchronous signal to the processor that some external event has occurred. Each interrupt has its own handler. Some interrupts include timer interrupt, I/O interrupt, interprocessor Interrupts – the kernel uses these interrupts to coordinate actions across the multiprocessor.

Processor exceptions: a hardware event caused by user program bahevior. Some actions that cause exceptions include

- attempts to perform a privileged action.
- access memory outside its region.
- divide by zero.
- access non-aligned addresses.
- reaching a breakpoint. The kernel replaces machine instruction in memory and sets up a trap. upon reaching the breakpoint, kernel restores the old instruction and transfers control to the debugger.

System calls: voluntary requests to switch. A process triggers a system call be executing an instruction with a specific invalid opcode. In each system call, processor mode switches to kernel and starts executing at a pre-defined handler – this prevents jump to arbitrary places.

1.3.2 Kernel to user mode

There are several types of transition from kernel to user.

New process: starts a new process by copying the program into memory and setting the PC and SP, and switching to user mode.

Resume process: restoring the PC and SP, and changing mode back to user mode.

Different porcess: saves PC, SP, and the whole state of current process then, loads a different process.

User-level upcalls: Operating system provide user programs with the ability to receive asynchronous notification of events. ?!

1.4 Implementing Safe transfer mode

for safe mode switch the follwoing must be done at minimum.

Limited entry into the kernel: hardware must ensure that the entry point into kernel is set by kernel itself; i.e. user programs cannot be allowed to jump to arbitrary locations in the kernel.

Atomic changes to processor state: transitioning between the two mode atomic – the mode, PC, SP, and memory protection are changed at the same time.

Transparent, restartable execution: saving and restoring the user process should be invisible to user – except mayb temporarily slowing down

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

1.4.1 Interrupt vector table

The processor has a special register that points to an area of a kernel memory called the *interrup vector table*. Each entry of interrupt vector table points to the first instruction of a different *interrupt handler* in kernel. The special register can only be set in kernel mode.

1.4.2 Interrupt stack

Interrupt stack is the place where interrupted processes are saved. A processor register points to the interrupt stack. Upon receiving a context change, hardware sets the register to the base of the interrupt stack. Then, hardware saves some of the interrupted process' registers by pushing them onto the interrupt stack before calling the kernel's handler. When kernel's handler runs, it can push any remaining registers onto the stack. When returning from an interrupt first, the registers pushed by kernels' handler are popped and then, the registers pushed by hardware.

1.4.3 Two stacks per process

Most operating system kernels allocate a kernel interrupt stack for every user-level process. When the process is running the hardware interrupt stack points to that process's kernel stack. A pointer to the kernel stack is stored in PCB.

1.4.4 Interrupt Masking

Delivering an interrupt inside interrupt handler causes the stack of the original process to be lost. Therefore, we need to defer interrupt – disable interrupts or mask – until it is safe to do – unmask interrupt. The instruction to mask and unmask interrupt must be privileged.

Generally, the hardware only buffer one interrupt for each type. Hence, some interrupts may be lost.

1.4.5 Hardware support for saving and restoring registers

In x86:

- the x86 pushes the interrupted process' stack pointer onto the kernel's interrupt stack and switches to the kernel stack.
- x86 pushed the interrupted process' PC.
- x86 pushed the *x86 process status word*.
- processor jumps to the interrupt vector table and then interrupt handler.
- interrupt handler pushes the remaining registers by 'pushad' instruction onto the stack (??).
- floating point registers are not usually saved unless the kernel switches to a different process.
- To restore state, we 'popad' the registers off the stack and 'iret' which loads a stack pointer, instruction pointer, and processor status word off of the stack into appropriate processor registers.

1.5 x86 mode transfer

The section is a detailed description of concepts described earlier. x86 is segmented; so the current instruction is a combination of code segment (cs) plus the instruction pointer (eip). Similarly, for the stack we have stack segment (ss) and stack pointer (esp). The current privilege level is stored as the low-order bits of the cs register rather than the processor status word (eflags).

1. Mask interrupts; to prevent interrupts in the middle of switching from user to kernel mode.
2. Save three key values; saves stack pointers (ss and esp), execution flags (eflags), instruction pointers (cs and eip) to internal and temporary hardware registers.

3. Switch onto the kernel interrupt stack; switches stack pointers (ss and esp) to the base of the kernel interrupt stack, as specified in a special hardware register.

4. Push the three key values onto the stack; pushes the saved values Step 2) onto the kernel stack.

5. Optionally save an error code; hardware pushes an error code to provide more information about the event or just pushes a dummy value onto the stack so it is the same format.

6. Invoke the interrupt handler; changes the instructions pointers to the address of the interrupt handler, as specified by a special register which contains the location of the interrup vector table.

After handler is called, it may pushes the rest of the values onto the stack if it wishes to change them.

When the handler completes, it pops the registers and the error code of the stack. Finally, it executes iret instruction to restore processors' state before the interrupt. It may also be needed to change the instruction pointers to point to the next instruction before iret to avoid infinite loop; e.g. when emulating hardware. For a system call trap, the x86 hardware does the increment when it saves the user-level state.

A *trapframe* is the data stored by hardware and interrupt handler at the base of the interrupt stack, as explained above. Typically, a pointer the trapframe is passed as an argument to the handler; e.g. to allow system calls to access arguments passed in registers.

1.6 Implementing secure system calls

Implementing system calls requires the operating system to define a *calling conventions.* System calls use the same mechanism for switching mode. In fact, the x86 instruction to trap into the kernel on a system call is called int.

A *pair of stubs* is a pair of procedures that mediate between two modes to provide a safe system call; e.g. againts malicious use of the interface.

- The usre calls the stub in the normal way.
- The user stub fills in the code for the system call and executes the trap instruction.
- Hardware transfers control to the kernel. The handler acts as a stub on the kernel side, copying and checking arguments and then calling the kernel implementation of system call. In detail kernel stub has four tasks.

  Locate system call arguments. The call arguments are stored in user memory, typically on the user stack which may be corrupted. Even if it is valid, it is virtual, not a physical address. Other pointer arguments must be verified to be a legal address and then converted to physical address.

  Validate Parameter. The kernel must protect itself against malicious or accidental errors in the format or the content of its arguments; zero-terminate strings, corrupted files, pointing to memory outside the application region, or extending beyond it, not having permission to access the file, file may not exists, and so forth. If error is detected kernel returns it to the user program.

Copy before check. to prevent application changing its arguments, they must be copied before being checked– *time of check vs. time of use (TOCTOU) attack*.

Copy back any results. copy back results to user memory but check the user address before doing so :).

- After the system call complete, it returns to the handler.
- The handler returns to user level at the next instruction in the stub.
- The stub returns to the caller.

1.7 Starting a new process

To start running at the user level, the kernel must

- Allocate and initialize the PCB.
- Allocate memory for the process.
- Copy the program from disk into newly allocated memory.
- Allocate a user-level stack for user-level execution.
- Allocate a kernel-level stack for handling system calls, interrupts and processor exceptions.

  to start running the program
- Copy the arguments into user memory.
- Transfer control to user mode.

There is another level indirection, inserted by the compiler. To allow main to return (instead of exit) complier wraps it with a start method and then exits.

1.8 Implementing upcalls

Virtualized interrupts and exceptions are called *upcalls* – *signals* in UNIX. There are several uses for immediate event delivery with upcalls.

Preemptive user-level thread. timer upcall to switch tasks and share the processor more evenly, or to stop a runaway task.

Asynchronous I/O notification. A notification can be sent once I/O completes which will allow user to return immediately after a system call requests to do other things.

Interprocess communication. when an event requires the instant attention of another process; e.g. for logout, to notify applications that they should save file data and cleanly terminate.

User-level exception handling. operating system needs to inform the application when it receives a processor exception so the application runtime, rather than the kernel, handles the event.

User-level resource allocation. the operating system must inform the process when its allocation changes; e.g. because some other process needs more or less memory.

Similarities of signals and interrupts:

Types of signals In place of hardware-defined interrupts and exceptions, the kernel defines a limited number of signal types that a process can receive.

Handlers Each process defines its own handler for each signal type or it can resort to default handler instead.

Signal stack Application have the option to run UNIX signal handlers on the process' normal execution stack or on a special signal stack allocated by the user process in user memory.

Signal masking UNIX defers signals for events the occur while the signal handler for those types of events is in progress. UNIX also provides a system call for applications to mask signals as needed.

Processor state Normally, when the signal handler returns, the kernel reloads the saved state into the processor to resume program execution. The signal handler can also modify the saved state; e.g., so that the kernel resumes a different user-level task when the handler returns.

Chapter 2

Slides

2.1 Lecture 1

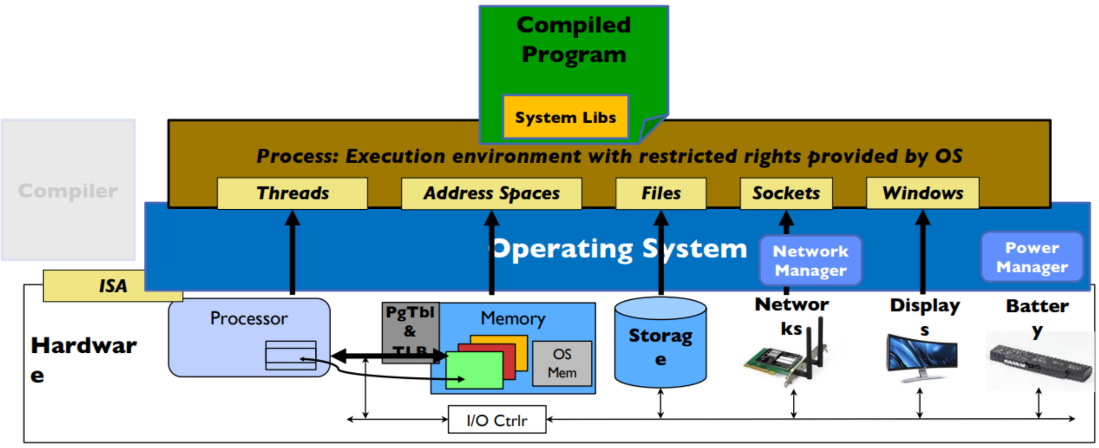
The key building blocks of OS

- Process
- Threads, Concurrency, Scheduling, Coordination
- Address space
- Protection, Isolation, Sharing , Security
- Communication, Protocols
- Persistent storage, Transaction, Consistency, Resilience
- Interfaces all devices

2.2 Lecture 2

2.2.1 Thread

A single unit execution context



- It has program counter (PC), registers, execution flags, stack, and memory state
- A thread executes on a processor when it is resident in the processor's register.
- A thread is suspended when its state is not load into processor.
- When a thread is not running it's saved in the memory called thread control unit (TCB).
- TCB is saved in kernel.

2.2.2 Address space

The set of accessible address and the state associated with them.

2.2.3

An operating system must

- protect itself from user programs.
- be reliable.
- be secure.
- ensure privacy.
- ensure fairness.
- protect programs from one another.
- prevent threads owned by one program to impact other

2.2.4 Simple protection: Base and Bound (B&B)

sets lower and upperbound for a program to access.

2.2.5 Process

execution enviroment with restricted access.

- (Protected) address space with one or more threads
- owns memory
- has file descriptors, file system context
- encapsulate one or more therad sharing process resources.

2.2.6 Dual mode

1. Kernel mode (superivor)
2. User mode