CS-118-02 Week 15

Varun Narravula

2022-05-09

Introduction to Interrupts

Up to this point, you've let the OS handle quite a lot of the inner workings of how to run things. However, what if you need to do a task immediately? Enter interrupts. An interrupt is a way for code to request that the CPU pause what it is currently doing, run a specified task, then return control when it is done. In some ways, it is similar to threading, but it is more low-level than the higher-level POSIX threads that you have used with the `pthread` library.

Categorizing interrupts is based on it is asynchronous or synchronous. An asynchronous interrupt is an interrupt that can happen at any time immediately. These are almost always generated by other hardware devices than the CPU or by code you can write, such as drivers. A good example of this is keyboard input. Anytime you hit a key on your keyboard, an interrupt is generated at that instant, and it asks the CPU to pause and interpret the key that was typed and do things with it (like sending the typed key to the screen, or maybe something else like for game controls). In the Linux kernel, such interrupts are referred to as IRQs, or **I**nterrupt **R**e**Q**uests. A synchronous interrupt, on the other hand, is different; it is an interrupt that can only happen after terminating another instruction; these are not generated by the OS, but rather generated by the CPU's control unit strictly after terminating instructions. The naming is a bit misleading because even though an asynchronous interrupt can happen at any time, it still needs to respect the CPU's clock signals; otherwise havoc ensues. To stop this naming confusion, asynchronous interrupts are often called "software interrupts", while synchronous interrupts are called "hardware interrupts".

Interrupts can also either be maskable or non-maskable. If an interrupt is maskable, that means the CPU can ignore the interrupt and continue execution. These are often used for I/O interrupts such as interacting with the keyboard. Non-maskable interrupts are interrupts that require immediate execution; think of Ctrl-Alt-Delete to reboot your system when the BIOS is starting up; the CPU cannot ignore this interrupt and will force a restart and POST.

To generate a software interrupt directly from assembly, you can use the `int` instruction. However, interestingly enough, you have already used interrupts in the form of the `syscall` instruction. `syscall` calls a function defined by

the Linux kernel with a number stored in `rax` that corresponds to it; under the hood, this is mapped to specific interrupts that are then called. These mappings, along with all other interrupts that can happen, are stored in a table called the Interrupt Vector Table (or IVT). On the x86 architecture, the IVT uses a slightly different name and calls it an Interrupt Descriptor Table (or IDT). The IDT contains a mapping of the addresses of interrupt handlers (which are essentially functions to run, similar to the way you pass a pointer using `pthread_create`; these are often also called interrupt service routines) to IRQs along with a few other things. When the CPU receives an interrupt, it will look up the corresponding interrupt handler to the IRQ using the IDT and then transfer control to it. The IDT itself is created during boot, without the OS, and is one of the very first things to be created at boot in memory so that other initialization routines can run using it.

CPUs often have a dedicated Programmable Interrupt Controller (or PIC) microchip that allows for interrupts to be prioritized and queued in a similar manner to threads. This will handle scheduling and will also allow for more flexibility in defining and executing interrupts.

Interrupts have two halves; a lower half, and an upper half. The upper half is what is happening in the interrupt itself, while the bottom half is the kernel reacting to the upper half of an interrupt. You would generally want to keep the upper half as tiny as possible, because interrupts are extremely computationally expensive. If enough are generated, this can overwhelm a CPU and decrease performance drastically. However, if an interrupt does need to run a lot of code, you can likely run it on a separate thread to decrease the amount of time needed for the OS to switch contexts between interrupts and programs.

Exceptions

Generating a hardware interrupt is much more difficult to do in the way that software interrupts are generated. In fact, you cannot generate these directly; only the CPU's control unit can. Despite this, you've already encountered the result of a hardware interrupt in the form of a segfault. You may have heard before that a segfault is a type of general protection fault (or GPF); these GPFs are actually specially generated synchronous interrupts generated that the OS then reacts to by sending a signal to the affected program. On Linux and almost all, if not all, Unix-like systems, segfaults and GPFs result in a `SIGSEGV` (segfault) or a `SIGTERM` (termination) signal being sent, which terminates the program immediately and sometimes dumps the core depending on the `sysctl` setting `kernel.core_pattern`.

A GPF is a type of exception; exceptions are a category of interrupts that only occur in abnormal situations. This is similar to Java and C++ exceptions in name only. There are three types of exceptions: faults, traps, and aborts. Faults are when a program has errored in some way (i.e. divide by zero, GPFs, page faults, segfault), but the error is recoverable. Traps are a type of exception that are

immediately reported after being executed; these are not normally used in normal programming, but rather for debugging; for example, the breakpoint interrupt can be inserted into a program by `gdb`, and this allows for stopping execution of the program for inspection at a certain point, and allow for it to continue after manual user intervention, or step-by-step if the programmer wants. The last type of exception, aborts, is extremely uncommon and is always unrecoverable. An abort exception will certainly result in a kernel panic (equivalent of the blue screen of death, or BSOD, on Windows systems) in the best case scenario or even the CPU halting until it is reset, either by a hard reboot or by a power cycle. Aborts can happen if a fault exception is unhandled (this is called a double fault), or this ensuing double fault is not handled. If the latter happens, then the CPU halts completely. You will likely never run into this unless you do kernel programming.

Virtual Memory

When the CPU accesses memory, it does not retrieve single bytes; this would be horribly slow. Instead, it retrieves sections of memory; these sections are called pages; think of turning the pages in a book, except the pages are fixed-size blocks of memory and the book is the computer's RAM. On earlier Intel x86 processors, this used to be configurable with a Linux kernel compile-time setting, but on x86_64 processors, this is no longer true, and is hardcoded in the CPU architecture as being 4 kilobytes (or 4096 bytes).

Important note: pages are sections of virtual memory. To refer to the corresponding section in physical memory or swap space on persistent storage, the term page frame is used. space Despite the fact that memory can be accessed more efficiently using pages, how do you deal with running out of memory? Enter virtual memory. Virtual memory is a way to use persistent storage (i.e. your hard drive, SSD, etc.) as a cache for memory when it is not being used. This is referred to as a pagefile on Windows, and as swap on Linux. Imagine that you have a dinner table, a pantry, and lots of food. It's impossible to fit all the food you have on the dinner table, so you instead put it in the pantry when you are not eating it to keep your dinner table free of extra food not being eaten. Virtual memory works in exactly the same way; the kernel watches programs that are not being used, and moves them out to swap space when they are not being used and back when they are needed again. Keep in mind that since persistent storage is not as fast as RAM, if this happens too much or your virtual memory size is too big, then a phenomenon known as thrashing occurs, and the kernel attempts to use this as memory much more than it needs to. This shortens the life of the persistent storage, and also makes your system much slower; RAM is many magnitudes faster than storage, and it shows when your system starts to thrash.

How does the kernel do this? It uses a type of fault called a page fault to know when the memory requested by a program is not at a specified location. A confession: the addresses you think are actual memory addresses when you

inspect pointers don't actually point to physical memory addresses. You've been lied to this entire time! Since Linux uses virtual memory, there are virtual memory addresses generated by the kernel that are mapped to physical addresses; this is defined in yet another mapping called a virtual translation table (or VTT). This VTT has a few columns: the virtual address assigned by the kernel, the real physical memory location it maps to, and a few other flags, the most important one being a flag that tells you whether or not the specified location is currently in physical memory or not.

When a memory address is requested, the kernel will look at the VTT and see if the assigned memory is in that physical memory location. If it is, then no page fault is generated, and the program proceeds. However, if the memory is not present, but is marked as being present in swap, then a minor (or soft) page fault is raised. The kernel will then look for this memory in swap and load it, and then the program can proceed normally. However, if a program decides that it needs more memory to accomplish its tasks, then it will raise a major (or hard) page fault. The kernel will then look for free pages of memory, and it will allocate free page(s) it finds to the program as needed, or it will attempt to free pages by moving unused programs to swap. If it is not able to do any of this, then two things can occur: if the out-of-memory (OOM) killer is enabled in the kernel, then it will attempt to forcibly stop any programs that it can to free up space; if it can't or if the OOM killer is not enabled, it will panic. A third type of page fault is an invalid page fault; this can sometimes be generated when invalid or protected memory is accessed, and can often be reported as a segfault as well; Windows uses page faults to report segfaults, but Linux does not.

Quock note: the kernel is not using normal standard library functions to do any of this. In fact, it cannot use the standard library, because it needs to be statically compiled so it can be loaded into memory independently by a bootloader without depending on a program interpreter that cannot be loaded because the kernel does not exist yet (yay, circular dependencies!). Using the standard library would also dramatically increase the kernel's physical size and space in memory. Instead, the kernel has handlers for any functions it might need as self-contained functions. To use outside functionality, kernel modules are used, but these cannot use the standard library or any external code either for the same reason.

Interrupts Example

Here is some code for an Arduino that uses hardware interrupts generated by touching a 5V power source to a digital pin input:

```
const byte ledPin = 10;
const byte interruptPin1 = 3;
const byte interruptPin2 = 2;

volatile byte state1 = 0;
```

```
volatile byte state2 = 0;

void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin1, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin1), isr_handler1, RISING);
  pinMode(interruptPin2, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin2), isr_handler2, RISING);
}

void loop() {
  if (state1 == 1) {
    noInterrupts();
    digitalWrite(ledPin, HIGH);
    Serial.print("Hello");
    digitalWrite(ledPin, LOW);
    state1 = 0;
    interrupts();
  }

  if (state2 == 1) {
    noInterrupts();
    digitalWrite(ledPin, HIGH);
    Serial.print("Bye");
    digitalWrite(ledPin, LOW);
    state2 = 0;
    interrupts();
  }

  delay(100);
}

void isr_handler1() {
  state1 = 1;
  state2 = 0;
}

void isr_handler2() {
  state1 = 0;
  state2 = 1;
}
```

This is code that is meant to run on an Arduino, so you can't run it on a normal Linux system. Since you might be familiar with Arduino code, here are few definitions on what some maybe-unfamiliar Arduino constructs are:

- `volatile` :: force memory allocation for variable, not on CPU cache/registers
- `setup()` :: function that runs before loop() once to initialize serial port and pins with their options
- `loop()` :: infinite loop that works as a `main` function after `setup()`
- `pinMode(ledPin, OUTPUT)` :: sets LED pin as output so it can receive power
- `pinMode(interruptPin1, INPUT_PULLUP)` :: sets input pin 1 as a pin that takes input with some resistance so that it can only be triggered by inserting a pin;
- `attachInterrupt(digitalPinToInterrupt(interruptPin1), isr_handler1, RISING)` :: creates an interrupt and handler to run after the set interrupt pin is triggered when the current rises
- `interrupts()` :: allows interrupts
- `noInterrupts()` :: disallows interrupts
- `Serial.print()` :: writes bytes to the Arduino's external serial port that you can inspect
- `digitalWrite(ledPin, HIGH/LOW)` :: toggles LED on/off

First, the `setup` function initializes the serial port so that you can print to it, and also sets up the LED pins so they can light up. There are a few global variables to store state (this is the only real way to have shared state): they are the pins that represent the LED and the pins to insert to trigger the specified interrupt service routine, as well as two variables that hold integers that are forcibly assigned using memory instead of on the stack or inside registers. Then, inside the loop, it prints to the serial port depending on the value of `state1` or `state2`. Third, it resets the state if was set, so that it doesn't run again. Every time you insert a pin into a corresponding input, the interrupt service routine is triggered for it, and the state is set; the loop then sees this, and then it resets the state after printing to the console and making the LED blink. Notce how interrupts are disabled in the lower half of the interrupt inside of the `loop`; this is to prevent an interrupt from disrupting state again and so that an appropriate number of interrupts can correspond with the results (in simpler language this means that the number of triggered interrupts correspond to the number of prints).