CS 33:
Introduction to Computer Organization

Week 10, Part 2

Exceptional Control Flow

- ...or what happens when something unusual happens?
- An example of something unusual is when an error occurs.
- How the program/OS respond depends on the severity and type of the error.

Exceptional Control Flow

- Sometimes, it's enough to be told about the error.
- For example, read returns -1 upon error. Normally, read returns the bytes read. If you managed to read -1 bytes (you unread a byte? you forgot it?), something went wrong.
- waitpid(...) normally returns PID of the child waited upon. However, it returns 0 or -1 upon error.

Exceptional Control Flow

- This system works fine if you only care about when an error occurs.

```
int n = read(0, buf, sizeof(buf));
if(n < 0)
{
  printf(“An error occurred\n”);
}
```

- However, sometimes it's nice to also know exactly what happened.

Exceptional Control Flow

- One method: #include <errno.h>
- errno.h defines an integer errno. When a system call results in an error, errno is set and its value indicates what the error was.
- errno is thread local using “thread local storage”. Essentially, what looks like a global variable is actually unique to each thread.

Exceptional Control Flow

```
int n = read(0, buf, sizeof(buf));
if(errno == EINTR)
{
  printf("Read failed because it was interrupted");
}
```

Exceptional Control Flow

- errno and other methods work fine and we can condition on the value of errno and respond accordingly.
- Sometimes, however, we want to respond to errno in special, one could even say, exceptional ways.
- One example is if you're 10 functions deep and you'd like to return to the main function on error.

Exceptional Control Flow

- One way of doing this (other than literally returning 10 times) is to use a technique called non-local jumps, where you jump from one function to another function.
- This is done with setjmp() and longjmp().
- At a high level, a call to setjmp marks itself as sort of a restore point. When there's some error, jump back to setjmp.
- A call to longjmp will return to setjmp.

Exceptional Control Flow

- int setjmp(jmp_buf env);
  - jmp_buf is an integer array.
  - When setjmp is called, certain registers are stored in the jmp_buf (stack pointer, frame pointer, instruction pointer, general purpose registers).
  - Returns 0... initially.
- void longjmp(jmp_buf env, int retval)
  - Restores all of the registers backed up in env.
  - Saves “retval” to %eax. This is the return value, but this isn't the return value of longjmp, it's the return value of setjmp

Exceptional Control Flow

- Consider:

```
#include <setjmp.h>
jmp_buf env;
int main()
{
  int ret;
  if((ret = setjmp(env)) == 0)
  {
    work();
  }
  else
  {
    printf("Returned with: %d\n", ret);
  }
}
```

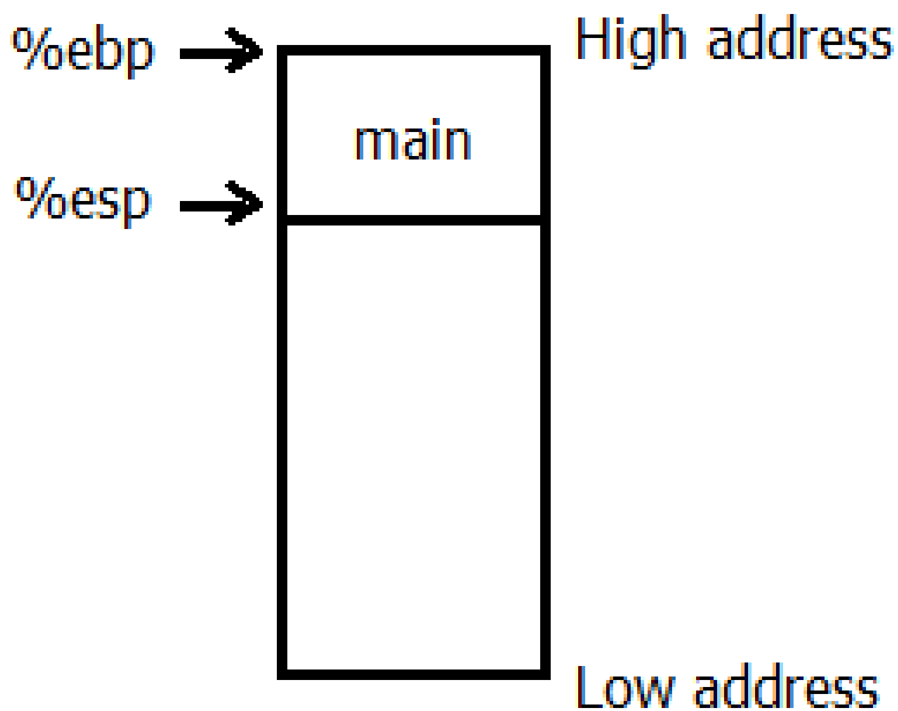
```
void work()
{
  work1();
}

void work1()
{
  work2();
}

void work2()
{
  longjmp(env, 1);
  return;
}
```

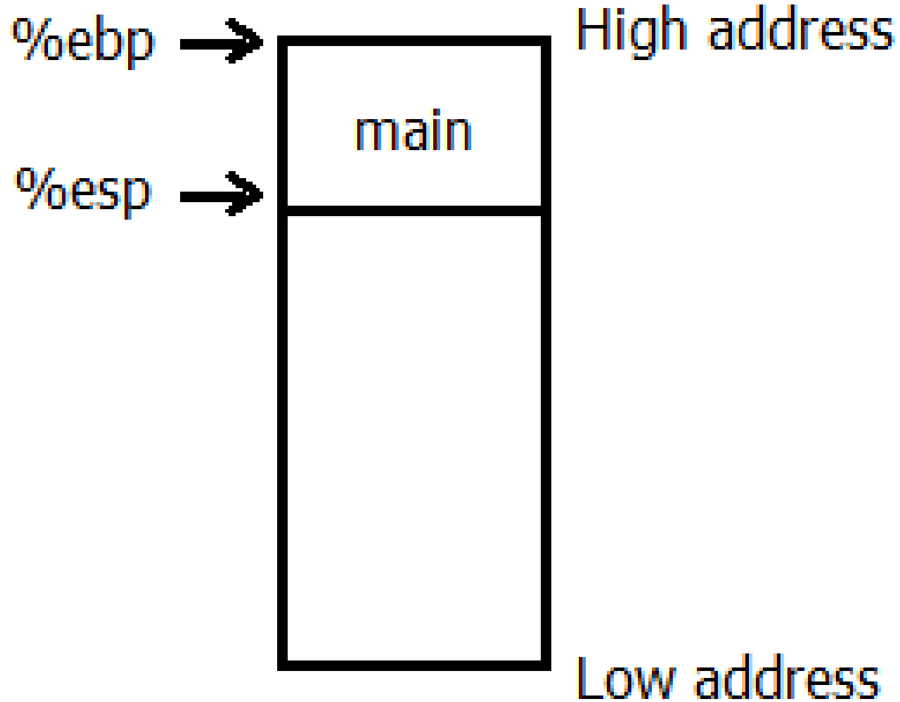
Exceptional Control Flow

- At a high level:
- Immediately before setjmp is first called, the stack looks like this:
- Once setjmp is called, the %ebp, %esp, and %eip that correspond to approximately this point in time are saved.



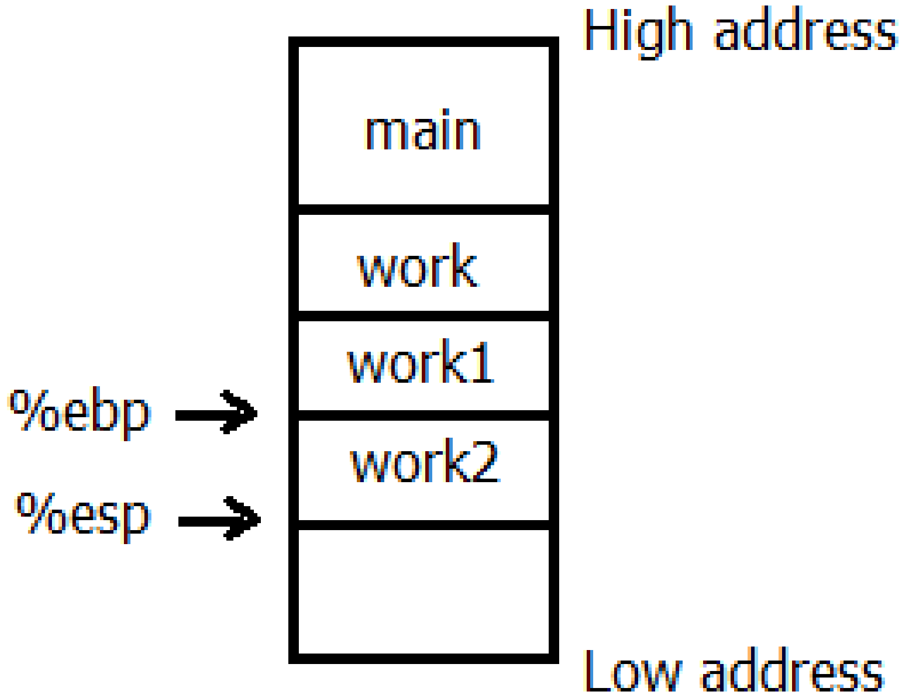
Exceptional Control Flow

- When setjmp is called, it saves the values into env and returns 0.
- Because setjmp returned 0, work() is called.
- Before longjmp is called, we'll have descended into several functions.



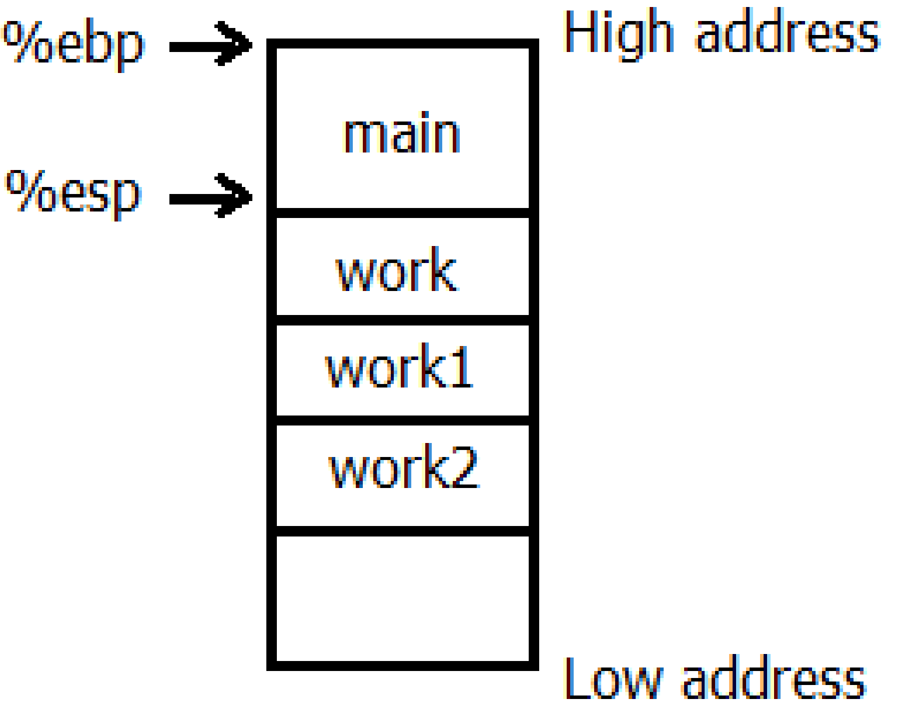
Exceptional Control Flow

- The stack will look like this:
- Then longjmp(env, 1) is called.
- %eip, %esp, %ebp are restored from the previous saved state.
- %eax is set to be 1.



Exceptional Control Flow

- The stack now looks like this.
- The %eip is restored to a point where it appears to the machine that setjmp has just returned.
- Because %eax is now 1, setjmp has effectively returned 1.



Exceptional Control Flow

- This method is useful, but beware of the following:
- The book says general purpose registers are saved, but it isn't clear which ones.
  - Say there was a local variable in main:

```
int ret;
int local_var = 10;
if((ret = setjmp(env)) == 0)
{
  work();
}
```

Exceptional Control Flow

- What if “local_var” was optimized such that it only exists in a register, say %edx.
- ...and what if %edx is not saved by setjmp? (I'm genuinely not sure, the internet is pretty tight-lipped about what exactly is saved)
- Then throughout the execution, and the work functions, %edx may have been used. When we longjmp back to main, the local_var is not restored and is completely lost.
- Use “volatile” to prevent this. Volatile makes sure that a value isn't optimized away. As a result, the variable is guaranteed to exist on the stack.

Exceptional Control Flow

- Let's say you learned your lesson. Now local_var is on the stack.

```
int ret;
volatile int local_var = 10;
if((ret = setjmp(env)) == 0)
{
  local_var = 11;
  work();
}
```

Exceptional Control Flow

- The expected state when we called setjmp was that local_var = 10.
- However, before we called work we set local_var to be 11. local_var is, for sure, on the stack now.
- When we longjmp back to setjmp, the original value of local_var was not restored; the stack isn't preserved.
- This isn't quite right either.

Exceptional Control Flow

- Additionally, because the registers were simply reverted, certain things may be left in a bad state.
  - malloc'd pointers may be lost.
  - file descriptors will not be closed
  - etc.

Exceptional Control Flow

- setjmp will save the state that of the function that called it.
- Say function foo calls setjmp. setjmp will save the state of the stack frame of foo.
- If foo returns, then suddenly the saved state (foo's stack frame) becomes invalid. It's no longer meaningful to use a longjmp.

Exceptional Control Flow

- As an aside, setjmp thus places itself in the prestigious class of, “call once, return multiple times” functions.
- What else calls once but returns more than once?

Exceptional Control Flow

- As an aside, setjmp thus places itself in the prestigious class of, “call once, return multiple times” functions.
- What else calls once but returns more than once?
  - fork()
- Meanwhile, longjmp never actually returns. It manually sets %eip to another function.

True Exceptions

- All of the previous examples represent errors/unusual behavior that we prepared for.
- True exceptions are “an abrupt change in the control flow in response to some change in the processor's state”
- Come in four flavors:
  - Interrupts
  - Traps
  - Faults
  - Aborts

Interrupts

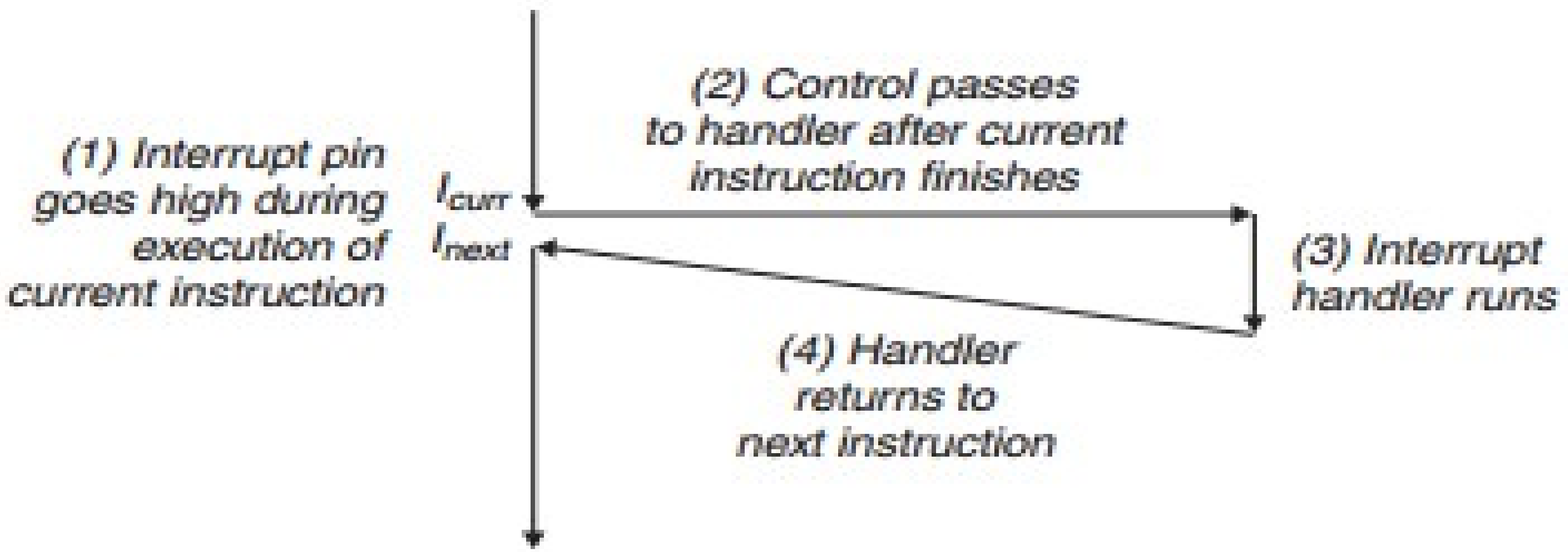
- Most commonly signals from I/O devices.
  - Keyboard key presses.
  - Mouse movement
  - Network adapter activity
  - Etc.
- Asynchronous
  - Occurs independently of currently executing program

Interrupt Handling

- I/O device triggers the “interrupt pin”
- After current instruction, stop executing current program and “control switches to interrupt handler”.
- The control is taken from the user and the interrupt handler is run by the kernel in kernel mode.
- This all occurs within the same process/thread.

Interrupt Handling

- Interrupt handler handles interrupt.
- Control is given back to previously executing program and back to the user.
- Previous program executes the next instruction.



Trap

- An intentional exception triggered by user. What for?
- Sometimes we need to do things that are not within the scope of what the program alone can do. We need the kernel's help to:
  - Read a file
  - Create a new process
  - Load a new program
- Synchronous: occurs as a result of program instruction.

Trap

- Consider the syscall assembly instruction.
- This causes a trap, which forces the kernel to take over to handle it.
- For example:

```
movq $87, %eax
syscall
```

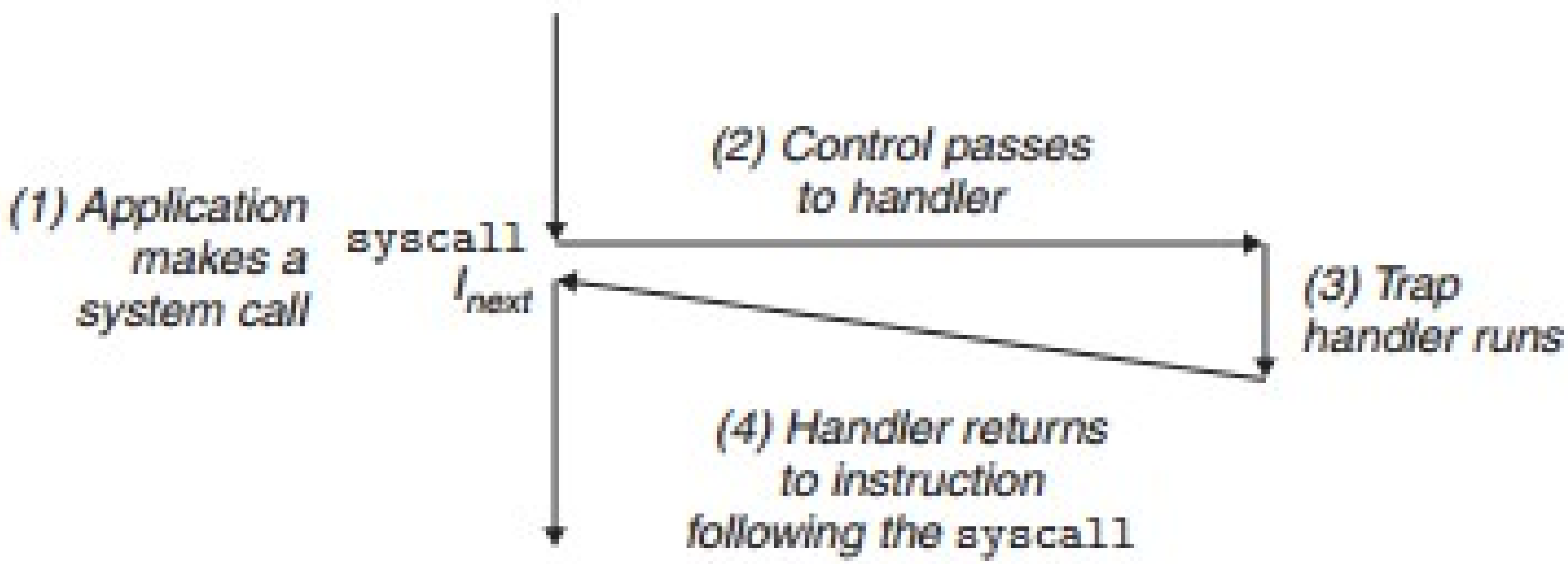
- The value in the %rax register determines which system call to perform.
- The values in %rdi, %rsi, etc determine what arguments are passed into the system call.
- This is all brand new to you I'm sure.

Trap

- Note: Turns out synchronous traps are also referred to as software interrupts, as in, interrupts caused by the software.
- Alternatively, the asynchronous interrupts from a few slides back are considered hardware interrupts.

Trap handling

- Same as interrupt handling, except caused by an explicit instruction.



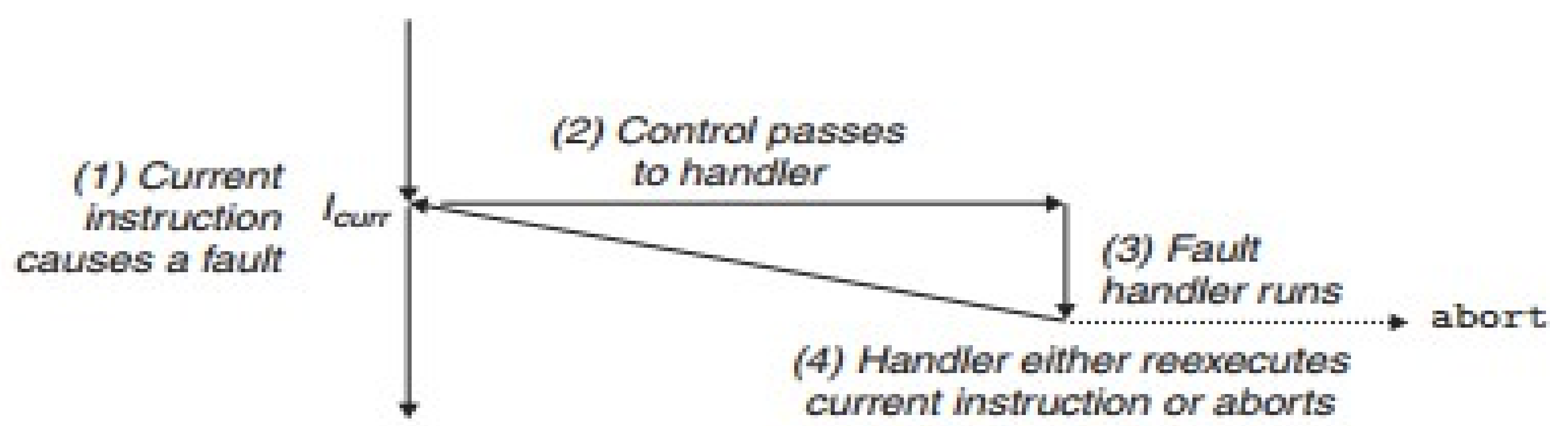
Fault

- Caused by a potentially recoverable, but unexpected error.
  - Divide by zero (in Linux, won't recover)
  - Invalid memory access (usually won't recover)
  - Page faults (must recover)

- Synchronous

Fault Handling

- Control passes to fault handler.
- Fault handler executes. If recovery is possible, return to instruction that caused fault. Else, halt.
  - Execute the instruction that caused the fault again?
  - If recoverable, whatever caused fault will be fixed and the instruction can be run without error.



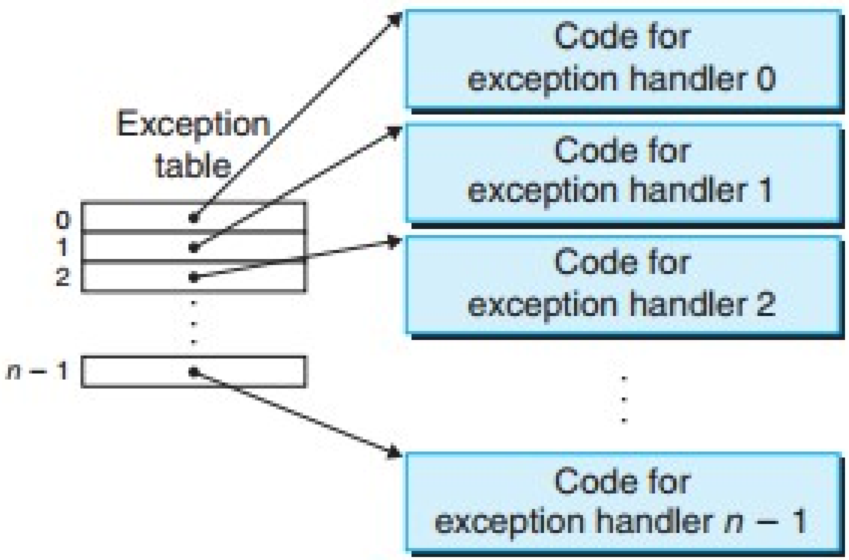
Abort

- Unrecoverable, fatal error.
  - Corrupted memory
  - Fatal hardware error
- Abort handling
  - Abort with no chance of recovery.

Exception Handling

- When an exception is received, the program responds by doing the following:
  - The current flow of execution is halted.
  - Switch from user mode to kernel mode.
  - Examine the exception and find it's exception number.
  - Use the exception number to index into the exception table.

Exception Handling



- Each entry of the exception table contains a pointer to the code that is used to handle each exception.
- Run the code
- Once exception is handled, if possible, return to the original code and switch back to user mode

Exception Handling

- Exception handling in similar to normal procedure calls except:
  - Exception handlers save more state (ex. RFLAGS) and must restore more if returning to the original code.
  - Uses the kernel stack rather than the user stack.
  - Runs in kernel mode of course.

Signals

- Exceptions are a property of the hardware.
- We introduce the concepts of signals, which are software methods of sending interrupts to running processes.
- Like interrupts, a process can receive a signal asynchronously from other processes/devices.

Signals

- Sending signals:
- int kill(pid_t pid, int sig)
  - pid is the process to send to, sig is the signal type.
- int alarm(unsigned secs)
  - Send yourself a SIGALRM signal in secs amount of seconds.
- The kernel sends these signals to processes.

Signals

- Receiving signals:
- When a process receives a signal, it switches away from its normal execution to a signal handler using the SAME thread.
- Unlike exceptions handlers however, the signal handler is run in user mode.
- Like exceptions, the type of signal is used to index into a signal handler table to determine what signal handler to run.

Signals

- Receiving signals:
- The user can define code to be run in the case of a signal using the following:
- sighander_t signal(int signum, sighandler_t handler);
  - If you receive the signal "signum", then let the function "handler" be the signal handler.

Signals

- Some nuances in handling multiple signals:
- Pending signals are blocked.
  - If we're currently running the handler for a particular signal and a new signal is received, that signal is blocked. Once the handler returns, the waiting signals are addressed
- Pending signals are not queued.
  - There can only be one pending signal of a specific type. If a signal of type k is blocked and 10 more signals of type k are received, they will be coalesced and treated as one signal of type k.

End of

The Final Week

-The Exam Remains-

Dawn of

A New Day