Introduction to Computer Organization

DIS 1A – Week 10

Slides modified from Uen-Tao Wang

Agenda

- Virtual Memory
- Linking
- Exceptional Control Flow

Sample Question

- Suppose we change the Core i7’s virtual addresses to use "huge" pages, i.e., pages of size 1 GiB. We alter the rest of the implementation as little as possible: for example, we don’t change the page table size.
- 4a (10 minutes). What should virtual addresses look like with under this regime? In other words, what components would virtual addresses be broken up into, compared to the components normally used in the Core i7?

Sample Question

- 4b (10 minutes). Give two performance advantages that come from having huge pages, as opposed to the usual 4 KiB pages. One should be a time advantage, the other a space advantage. Briefly explain.
- 4c (10 minutes). Give two performance disadvantages, again one in time and one in space. Briefly explain.

Sample Question

- Addresses were originally 48-bits long.
- Page Size is usually 4 KiB or 2^12 bytes.
- Now, pages are 1 GiB or 2^30 bytes.
- How many bits are required for the virtual page offset?
  -
- How many bits are required for the virtual page number?
  -

Sample Question

- Addresses were originally 48-bits long.
- Page Size is usually 4 KiB or 2^12 bytes.
- Now, pages are 1 GiB or 2^30 bytes.
- How many bits are required for the virtual page offset?
  - 30
- How many bits are required for the virtual page number?
  - 18

Sample Question

- Time Advantage?
  - Applications that access large sequential data structures will benefit from not needing to constantly swap in new pages for that data structure.
  - Say a single array spans 1 GiB. To iterate through this using the normal 4 KiB page size (2^12 bytes per page), you'd need to page fault 2^18 times to pull it all into memory.
  - With the 1GiB huge page, it would take only page fault.
  - Disk accesses are incredibly slow so this could be a win.

Sample Question

- Space Advantage?
  - Smaller page tables.
  - We will now only need 2^18 entries. Previously since the VPO was 12 bits, we needed 2^36 entries. Each entry governs more data.
  - Additionally, consider 32-bit physical addresses. In the 2^12 byte page size case, the physical page number would require 20-bits. However, if the VPO requires 30-bits, then so does the PPO. Thus, the physical page number in the new case would be 2 bits. Each entry is also smaller.

Sample Question

- Time Disadvantage?
  - Because each page is huge, actually copying a single page from disk to RAM will take much longer than in the case of a conservatively sized page.

Sample Question

- Space Disadvantage?
  - Say we have 4 MiB of data that we're interested in, spread across 4 different pages (ie, the text segment has 1 MiB we want to use, the data segment has 1 MiB, etc.)
  - To pull this all into RAM, we need to use of 4 GiB of physical memory even though we were only interested in 4 MiB total.

Virtual Memory

- A comment regarding the TLB.
- Recall that the TLB is an additional cache that allows us to store page table entries.
- Unlike the page table, the TLB is a physical component.
- Is there going to be a problem with that?

Virtual Memory

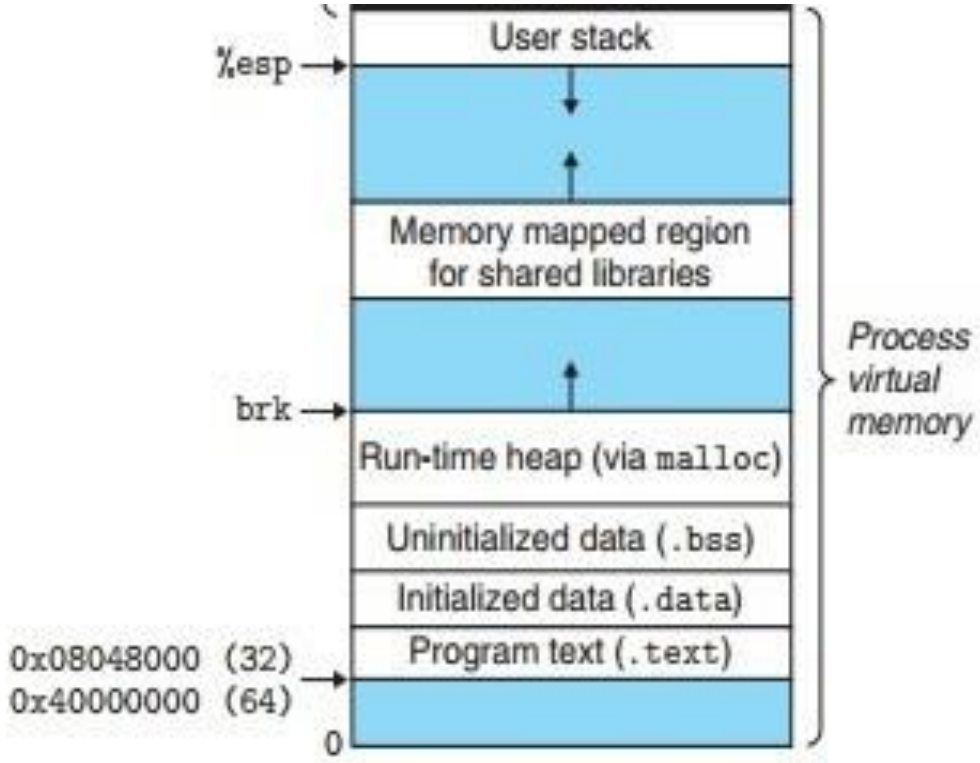
- We will have one TLB shared among all of the processes.
- You index into a TLB using the virtual address.
- Each process shares the virtual address space.
- This means that a TLB entry that is valid for one process may *appear* to be valid for another process if they happen to issue the same virtual address

Virtual Memory

- If left unchecked, this could be very bad. We could get false positive hits in the TLB and just keep going with the wrong addresses.
- The simple solution (as unfortunate as it is) is to flush the TLB after every context switch.
- An alternative would be to store information in the TLB as to which process an entry corresponds to.

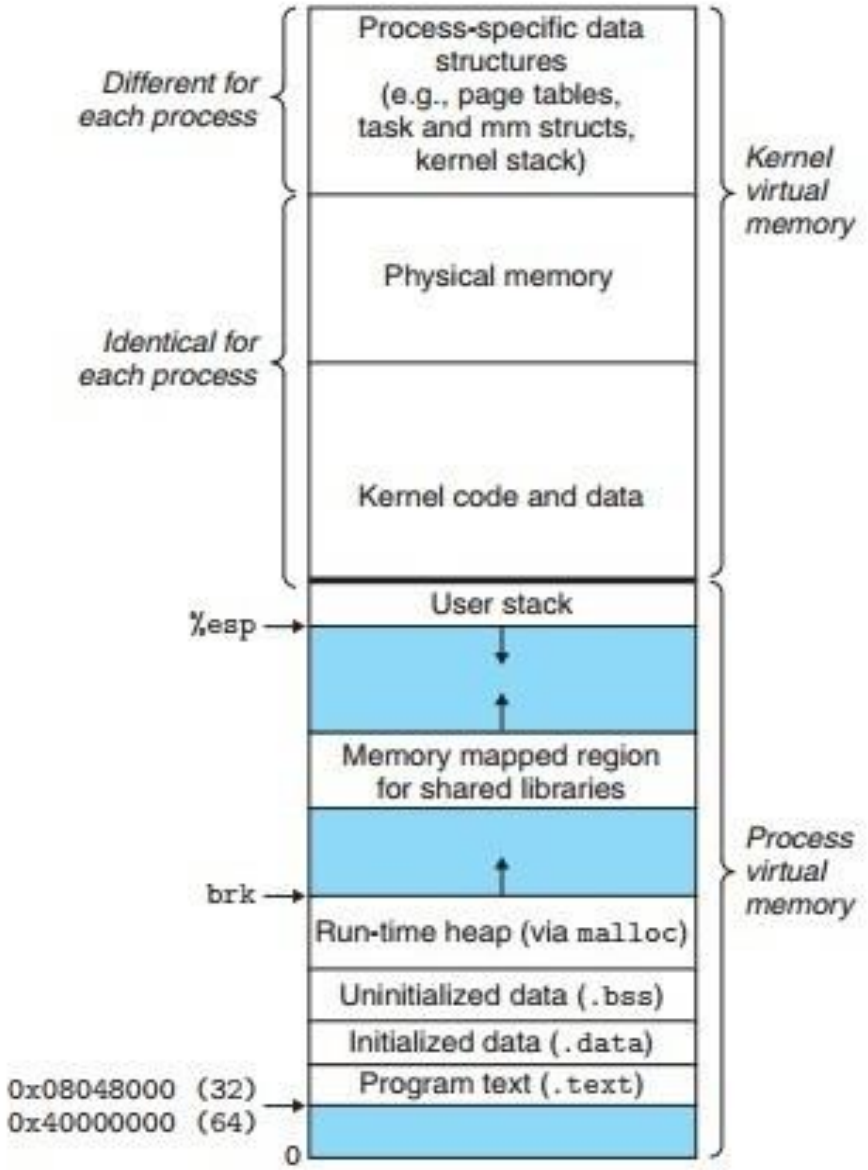
Virtual Memory

- The forbidden secret:
- We like to think of the virtual memory of a process as looking like this:



Virtual Memory

- It's actually more like this:
- The virtual memory space is split up into “user memory” which is the memory that is relevant to the running process.
- The “kernel memory” is memory that the OS/kernel requires.



Virtual Memory

- The kernel is the main component of an operating system that essentially *does* all of the work of the OS.
  - Schedules the thread/process that is run by the CPU
  - Handles the virtual memory assignments of each process.
  - Deals with I/O

- The kernel is just another program

Virtual Memory

- A chunk of the virtual address space is reserved for the kernel's use (ex. storing the kernel's code)
- In some instances, for example, half of the space could belongs to the kernel.
- User space: 0x0000000000000000 – 0x7FFFFFFFFFFFFFFF
- Kernel space: 0x8000000000000000 – 0xFFFFFFFFFFFFFFFF

Virtual Memory

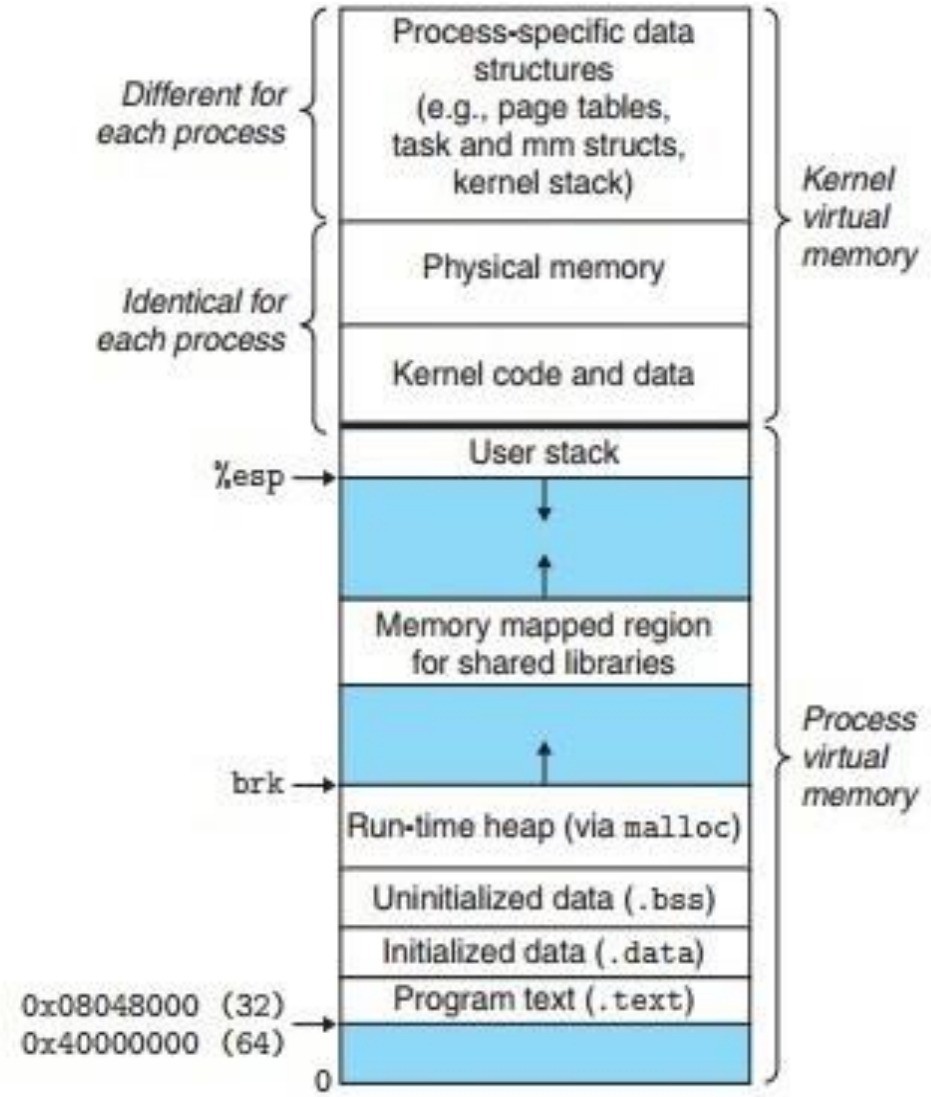
- This particular division gives the process about 2^{63} bytes of virtual memory space and the kernel 2^{63} bytes of virtual memory space.
- This is not enforced as some natural law or property. It's adjustable.
- In reality, because 2^{63} is 8 exabytes, there is no need to have so much memory.
- As a result we use 48-bit addresses.

Virtual Memory

- The address is technically 64-bits but we can effectively implement 48-bit addresses by taking a 48-bit address and sign extending to 64-bits.
- Thus:
  - 0x7FFFFFFFFFFF becomes 0x00007FFFFFFFFFFF
  - 0x800000000000 becomes 0xFFFF800000000000
  - etc.
- In modern systems, user space is addresses 0 to 0x7F...FF and the kernel space is 0xFFF8000...000 to 0xFFF...FFF.
- Let's consider the book diagram again.

Virtual Memory

Figure 9.26
The virtual memory of a Linux process.



Virtual Memory: User Space

- Unused buffer:
  - Addresses: 0x00000000-0x40000000
  - (Very generous) protection from dereferencing null or bad pointers.
  - You don't need to worry the next time you accidentally dereference 134000000, as we all tend to do.
- .text:
  - The source code of the loaded program.

Virtual Memory: User Space

- .data: initialized global variables, ex:

```
int a = 10; ←
int main()
{
  ...
}
```

- .bss: uninitialized global variables, ex:

```
int b; ←
int main()
{
  ...
}
```

Virtual Memory: User Space

- Why the distinction between bss and data?

Virtual Memory: User Space

- Why the distinction between bss and data?
  - When copying and loading the program from disk to be run, initialized global data must be copied from the disk to memory.
  - Uninitialized globals are assumed to be undefined, therefore we only need to allocate the necessary amount of space instead of having to read the file from disk.

Virtual Memory: User Space

- Heap:
  - For memory dynamically allocated via malloc
- Shared libraries:
  - The libraries included in dynamic linking (we'll get to that later)
- Stack
  - I really hope you know what the stack is at this point.

Virtual Memory: Kernel Space

- Kernel Code and Data
  - As mentioned above, the kernel is simply a program and therefore needs the same memory reservations to function correctly (.text, .data, etc).
  - This section is identical for each process (each process is running the same kernel)
- Here's where it's gets a bit weird. Yes. Here.

Virtual Memory: Kernel Space

- Process specific data
  - Resides at the uppermost sections of the memory.
  - The unique data and structures needed to maintain the correct execution of this process.
  - The book says this includes “page tables, mm structs, and the kernel stack”
  - The kernel stack: the kernel will have the same code for each process, but will be running different operations for each process, therefore, it needs its own stack
  - Makes sense

Virtual Memory: Kernel Space

- Process specific data
  - Page tables.
  - Wait, pages tables?
    - “Other regions of kernel virtual memory contain data that differs for each process. Examples include pages tables ...” (pg. 830, 3rd ed.)
  - So the page tables are actually stored in the kernel virtual memory?
    - “...and a data structure stored in physical memory known as the *page table* that...” (pg. 807 3rd ed.)
  - The page table is actually stored in main memory, but the process specific data of the kernel memory presumably maps to the page table in main memory.

Linking

- Yup, it's this picture again:

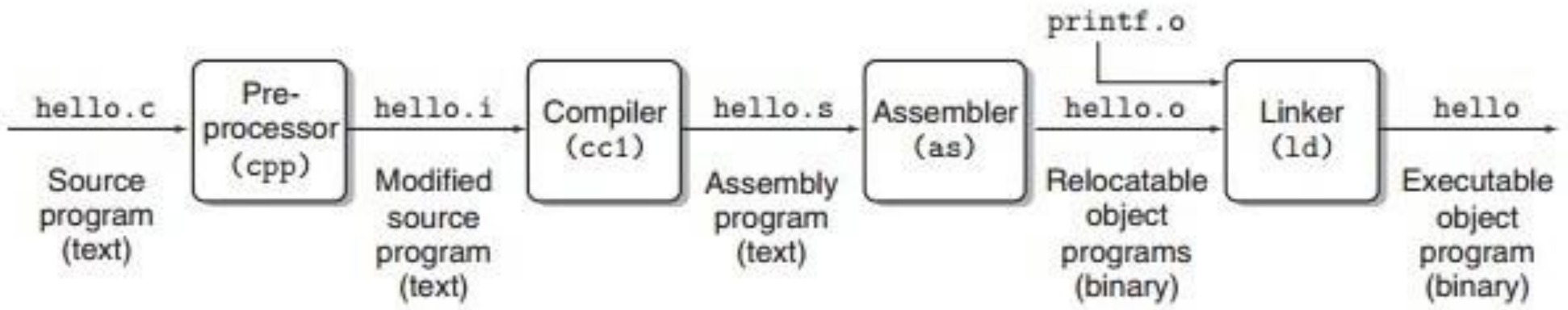


Figure 1.3 **The compilation system.**

Linking

- Except, we are here:

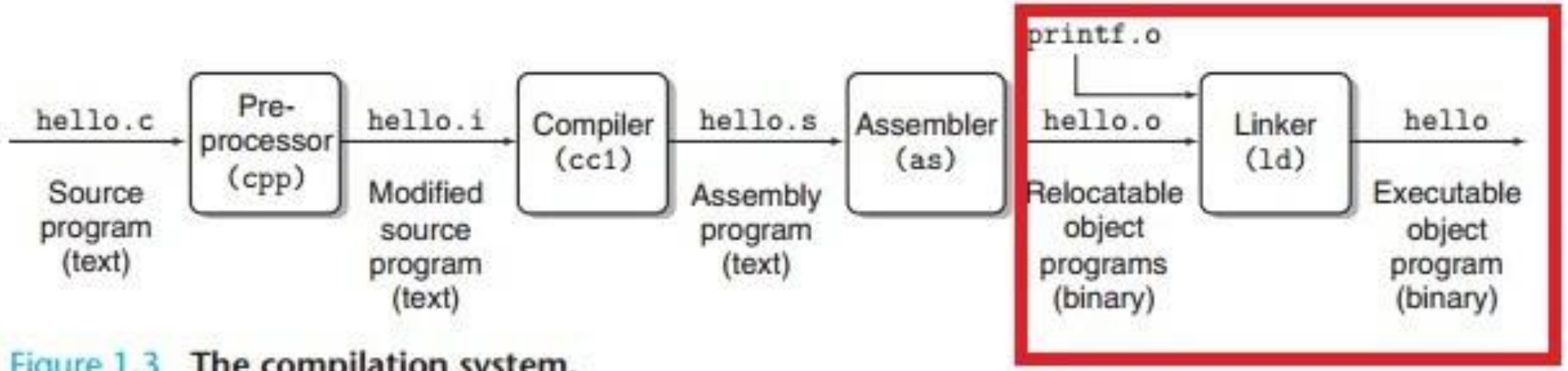


Figure 1.3 **The compilation system.**

Linking

1. Pre-processor: respond to and replace the compiler directives (lines marked with #) with the appropriate modifications
2. Compiler: Compile the C code into readable (so to speak) assembly.
3. Assembler: Compile the assembly into byte code/object files.

- Isn't that it? What else needs to be done?

Linking

- When something like this is done:
  - gcc main.c swap.c -o out
- ...the files for main.c and swap.c are compiled separately.
- Suppose we have the following:
- main.c:

```
int main()
{
  …
  swap();
}
```

swap.c:

```
void swap()
{
   ...
}
```

Linking

- main.c calls the function “swap” that is defined in swap.c
- main.c is compiled independently of swap.c, how does it know what swap is?
  - We need to do symbol resolution.
- Function calls look like “call 0x400688”. How do we know the address of swap? What is even the address of swap at this point?
  - We need relocation.

Static Linking

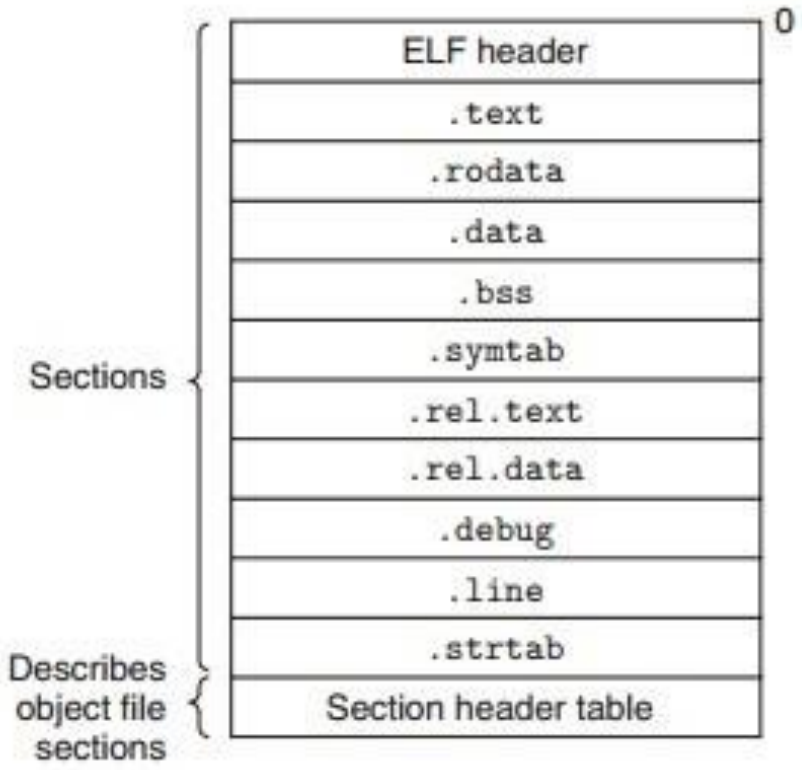
- Because of the issues described on the previous page, it's clear that there's a little more to compilation than simply translating C to assembly.
- Eventually, the entirety of the (statically linked) code will be in the .text section where all of the code will exist in the same place.
- However, because files are assembled separately, there is no prior knowledge of their position in memory.

Static Linking

- The reality is that the assembler step produces “relocatable object files”.
- These relocatable object files cannot be executed. Instead, they are primed so that they can be included as part of a project later on.
- When the relocatable object files are linked together, they form an “executable object file”, the final executable.

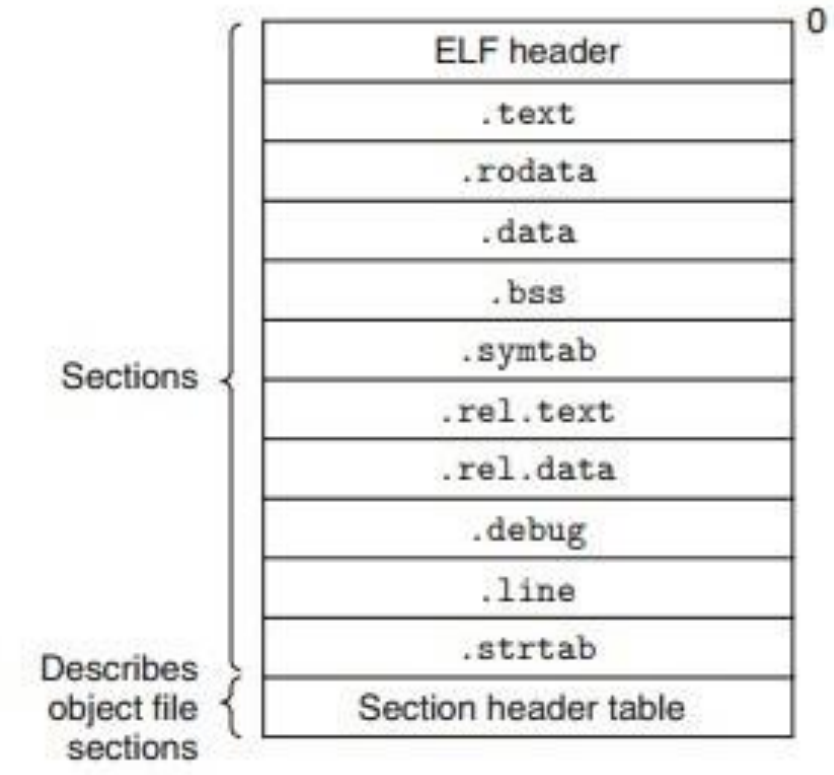
Static Linking

- Relocatable object file format:
- ELF header
  - File info (word size, byte ordering)
- .data, .bss
  - Same as in full object file



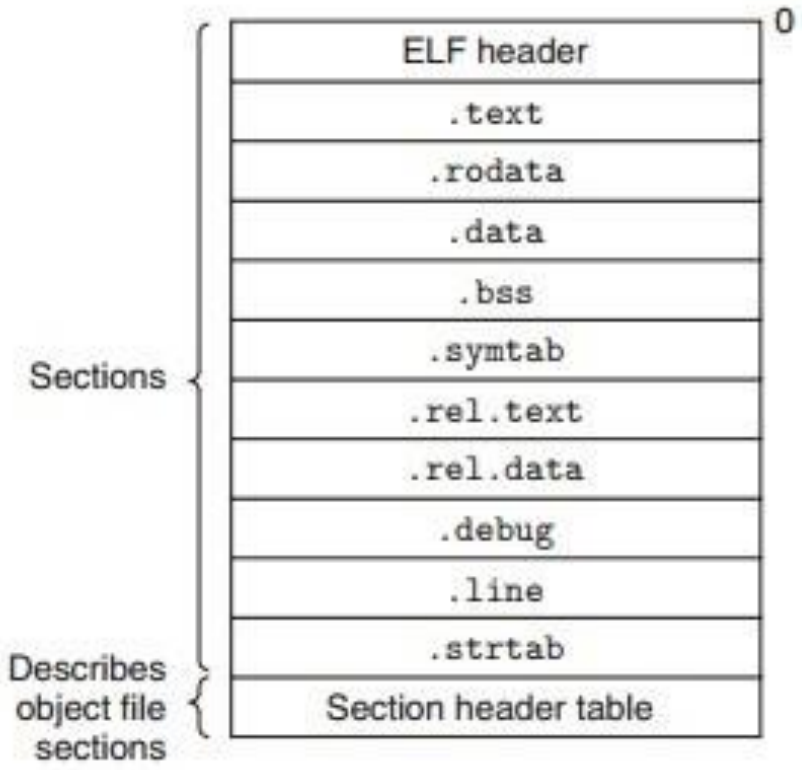
Static Linking

- .text:
  - The code of the compiled program
  - Incomplete (symbols are not resolved, relocation is not done)
  - In place of actual addresses for functions/variables that must be relocated, placeholder stubs are present.
- .rodata
  - “Read only data”
  - Things like jump tables, printf strings.



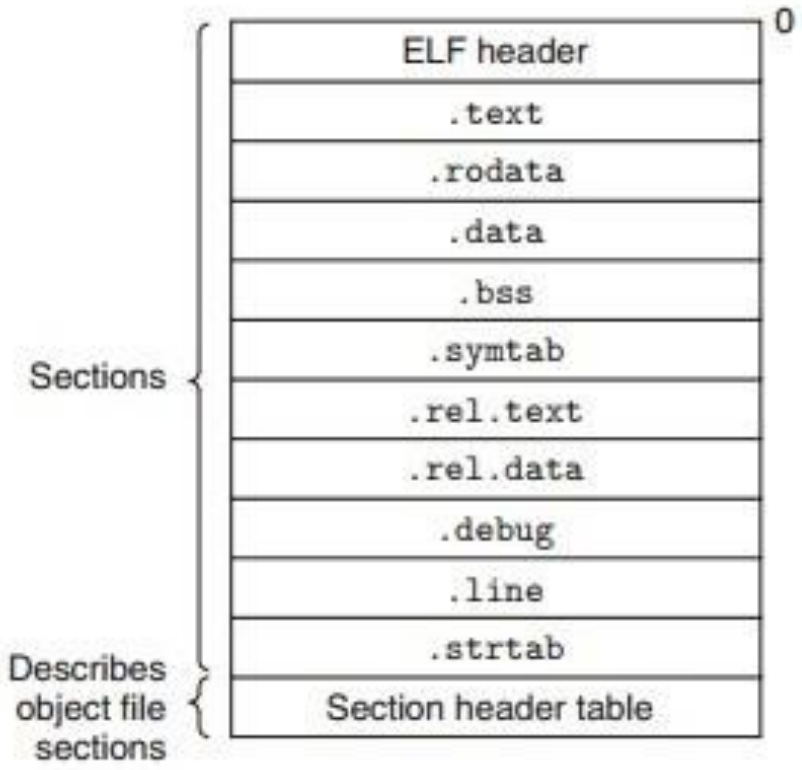
Static Linking

- .rel.text:
  - “relocation text”
  - List of locations in the .text section that will need to be modified to contain the actual addresses of functions/data.



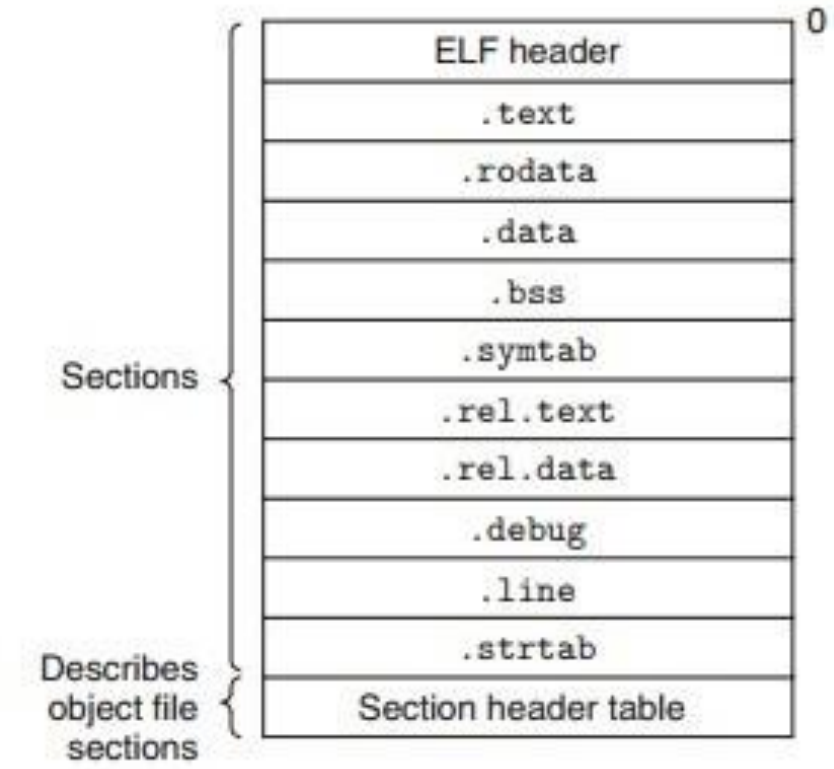
Static Linking

- .rel.data:
  - “relocation data”
  - Information about global variable referenced or defined in this module.



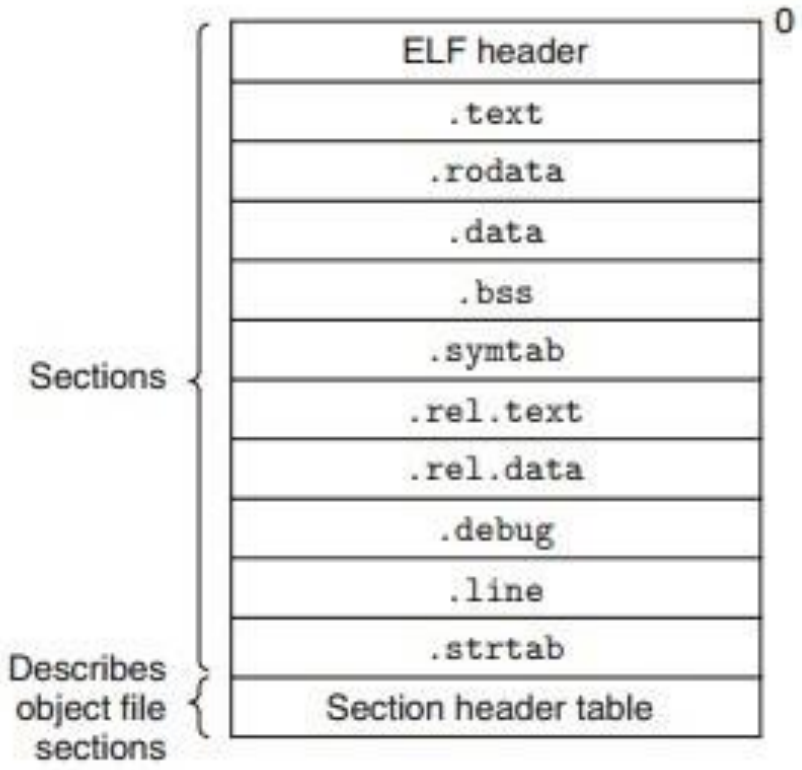
Static Linking

- .symtab:
  - “symbol table”
  - List of symbols that are defined and referenced in this module
  - Ex. names of functions, global/static variables, etc.
  - Does NOT contain symbols of local variables



Static Linking

- .strtab:
  - A string table that contains the actual strings of the symbols referred to in the symbol table.



Symbols

- The three types of symbols:
- 1. Global symbols defined by module m that can be referenced by other modules:
  - non-static global variables and functions

Symbols

- The three types of symbols:
- 1. Global symbols defined by module m that can be referenced by other modules:

foo.c:

```
extern int z;
int fun() ←
{
  int y = 10;
  z = 10;
}
```

bar.c:

```
static int v;
int z;
static void blah()
{
  v = 0;
  fun();
}
```

Symbols

- The three types of symbols:
- 2. Global symbols referenced by the module but defined by another module.
  - Ex: externals

Symbols

- The three types of symbols:
- 2. Global symbols referenced by the module but defined by another module.

foo.c

```
extern int z; ←
int fun()
{
  static int x = 0;
  int y = 10;
  z = 10; ←}
```

bar.c

```
static int v;
int z;
static void blah()
{
  v = 0;
  fun();
}
```

Symbols

- The three types of symbols:
- 3. Local symbols that are defined but used exclusively in this module.
  - Functions and global variables that are defined by static.
  - Recall, when static is applied to globals/functions, that means this variable/function is only visible to THIS module.
  - These “local” symbols do NOT refer to local variables. Non-static local variables DON'T need representation in the symbol table.

Symbols

- The three types of symbols:
- 3. Local symbols that are defined but used exclusively in this module.

foo.c:

```
extern int z;
int fun()
{
  static int x = 0;
  int y = 10;
  z = 10;}
```

bar.c:

```
static int v; ←
int z;
static void blah() ←
{
  v = 0;
}
```

Symbol Resolution

- When it comes time to link, each unknown symbol will be resolved to one entry of one of the symbol tables of the input relocatable object files.
- Local symbols are easy to resolve since the symbols in the code are only referenced in the same module.
- Globals are trickier since there can be multiple global variables with the same name.

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.

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

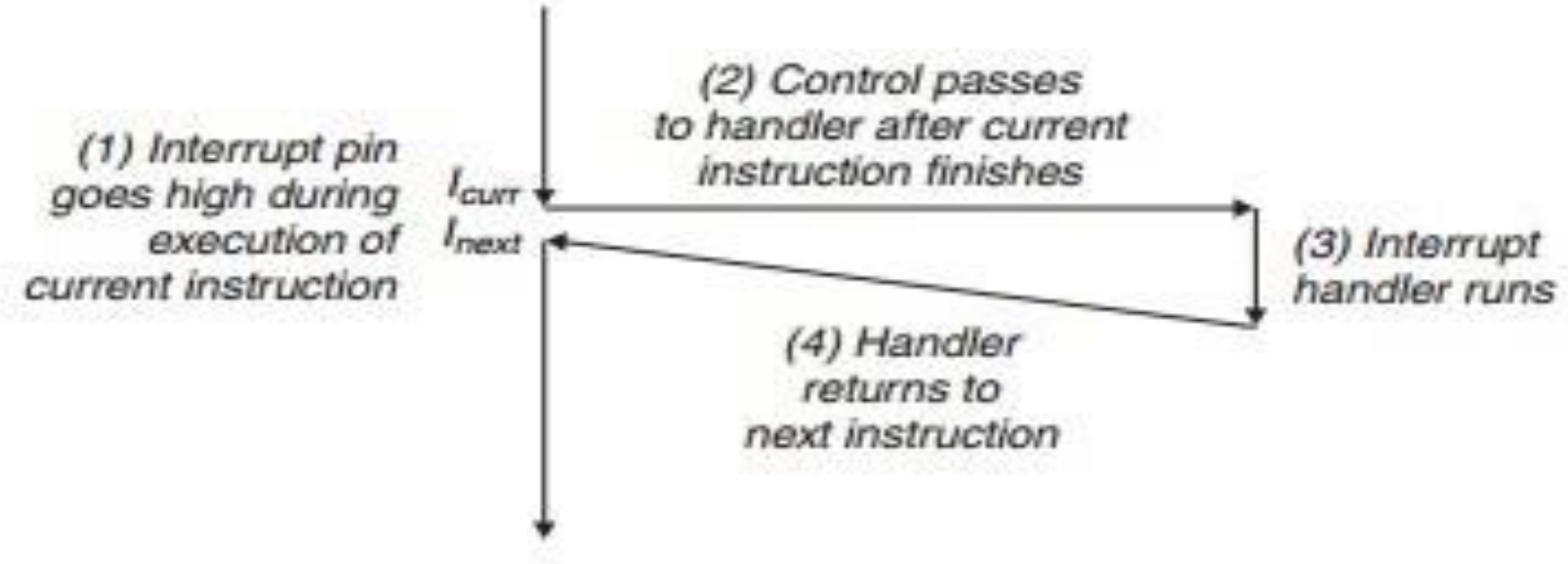
- Most commonly signals from I/O devices.
  - Keyboard key presses.
  - Mouse movement
  - Network adapter activity
  - Etc.
- Asynchronous
  - Occurs independently of currently executing program

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
```

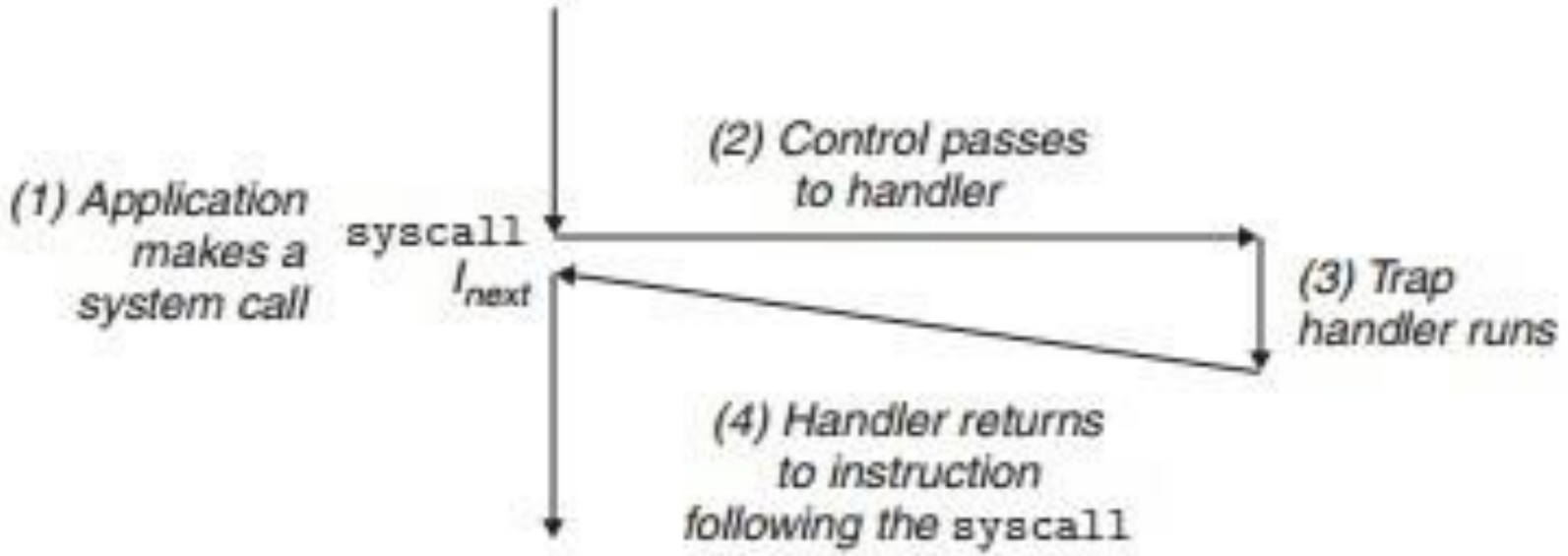
- The value in the %rax register determines which system call to perform.
- The values in %rdi, %rsi, etc determine what arguments are passed into the system call.

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.



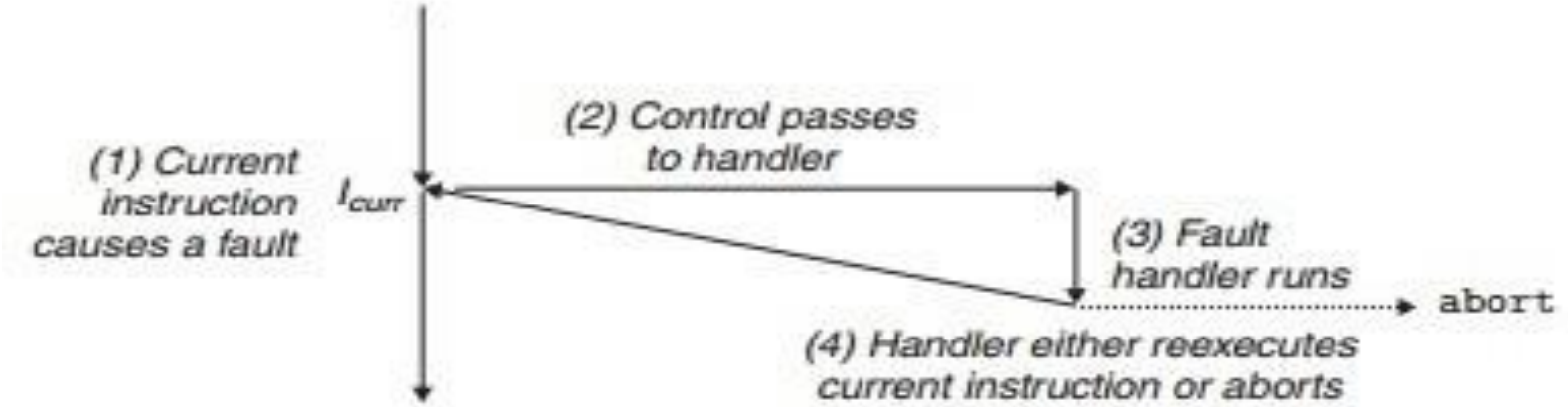
Fault

- 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.



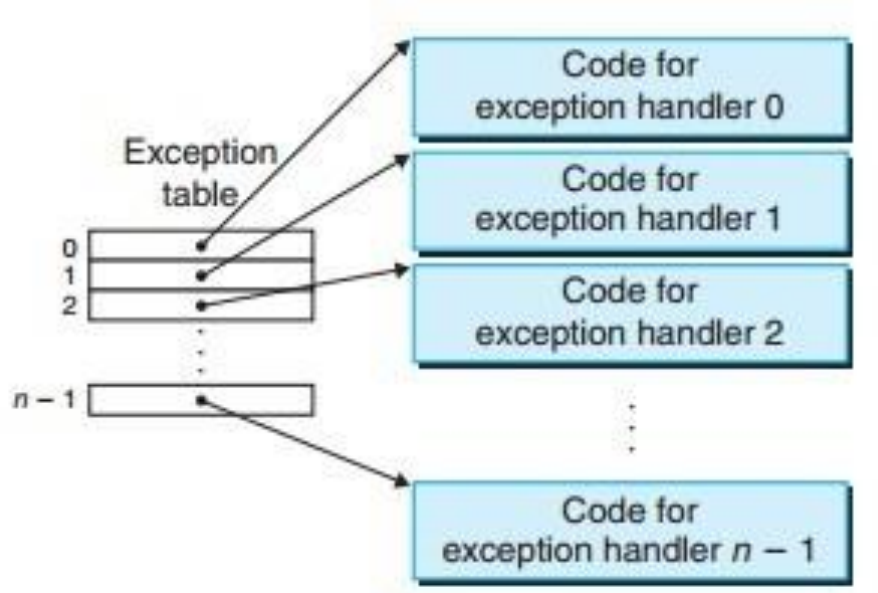
Abort

- 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.