CS 33 – Discussion Week 7

Agenda:

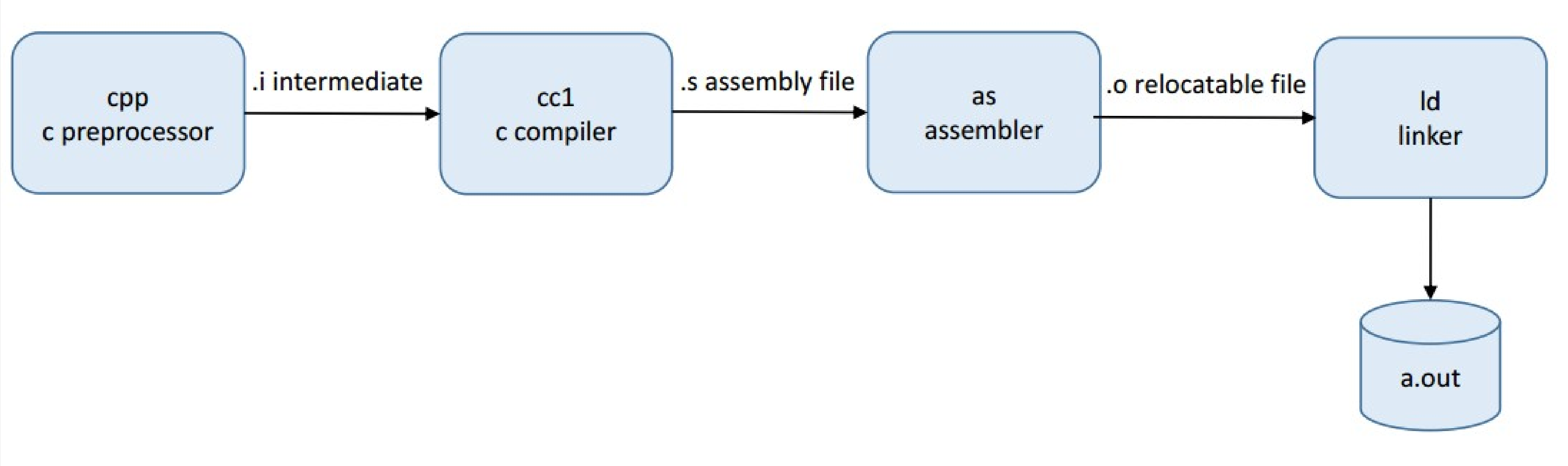
- Midterm
- Lab 3
- Linking
- Exception Control Flow

Lab 3

- Write a cache simulator
- Keep track of read and write hits/misses.
- See how the hit rate changes when we perform a matrix transposition in several different ways.
  - Row major order
  - Column major order
  - Blocking
- LRU and write-back
- Need to use C for this lab
- Compile with `gcc cachelab.c -lm`
- From now on, avoid using extra compiler flags

Linking

- Linking is the last step in compiling.



- Linker inputs: Relocatable object files (e.g. `main.o`, `swap.o`) that are compiled separately.
- Linker output: Fully linked executable object file (e.g. `a.out`) ready to be copied into memory and begin execution.

Linking

- Two tasks performed by linker:
  - Symbol Resolution – associate symbols with exactly one symbol definition. What exactly are symbols?
  - Relocation – Merges code and data into single sections. Relocates symbols from their locations in each “.o” to their final memory locations
- We will go into more detail about these 2 tasks.

What are Relocatable Object Files?

- `.o` files (e.g. `swap.o`)
- Piece of compiled code and data.
- Not yet executable without linking.
- What does it look like?
- Follows ELF: Unix **E**xecutable and **L**inkable **F**ormat
- pp.658-59 of textbook go into detail about each of these sections

Typical ELF Relocatable Object File

ELF Header
.text: machine code
.rodata: read only data
.data: initialized globals
.bss: unitialized globals (description)
.symtab: symbol table (globals and external function info)
.rel.text: relocation information for externals
.rel.data: relocation information for cross referenced data
.debug: -g symbols for gdb
.line: -g line numbers for gdb
.strtab: descriptive strings for .symtab
Section header table: which sections are in the table

Symbols and Symbol Tables

- Every Relocatable Object File has its own symbol table, with information about its own symbols. This lives in `.symtab`
- 3 types of symbols:
  - Global – All *nonstatic* C functions and *nonstatic* global variables defined in this module.
  - External – C functions and variables referenced in current module but defined in other modules.
  - Local – only used within this object file. e.g. global variables with `static` attribute. NOTE: These are **not** the same as local variables!

What symbols do we have here?

```c
// Code for main.c
int buf[2] = {1, 2};

int main()
{
 swap();
 return 0;
}
```

Symbols:

- Global: `buf, main()`
- Extern: `swap()`

```c
// Code for swap.c
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;
void swap()
{
 int temp;
 bufp1 = &buf[1];
 temp = *bufp0;
 *bufp0 = *bufp1;
 *bufp1 = temp;
}
```

Symbols:

- Global: `bufp0, swap()`
- Extern: `buf[]`
- Local: `bufp1`

Linker – Symbol Resolution

- Match up identically named external (global) symbols. Some defined within the module, some outside.
- How is this done? Program symbols are put into two categories:
  - Strong – functions and initialized globals
  - Weak – uninitialized globals

What symbols do we have here?

```c
// Code for main.c
int buf[2] = {1, 2};

int main()
{
 swap();
 return 0;
}
```

```c
// Code for swap.c
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;
void swap()
{
 int temp;
 bufp1 = &buf[1];
 temp = *bufp0;
 *bufp0 = *bufp1;
 *bufp1 = temp;
}
```

Symbols:

S S
- Global: `buf`, `main()`
- Extern: `swap()`

W

Symbols:

S S
- Global: `bufp0`, `swap()`
- Extern: `buf[]` W
- Local: `bufp1` W

Symbol Resolution

- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol
- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with `gcc -fno-common`

Source: CMU cs 213 slides

Examples

```c
// foo1.c
int main() {
    return 0;
}
```

```c
// foo2.c
int main() {
    return 5;
}
```

What happens when I try to compile this?

```
$ gcc foo1.c foo2.c
```

Taken from CSPP textbook p. 664

Examples

```c
// foo3.c
#include <stdio.h>
void f(void);

int x = 15;

int main() {
    f();
    printf("x = %d\n", x);
    return 0;
}
```

```c
// bar3.c
int x;

void f() {
    x = 25;
}
```

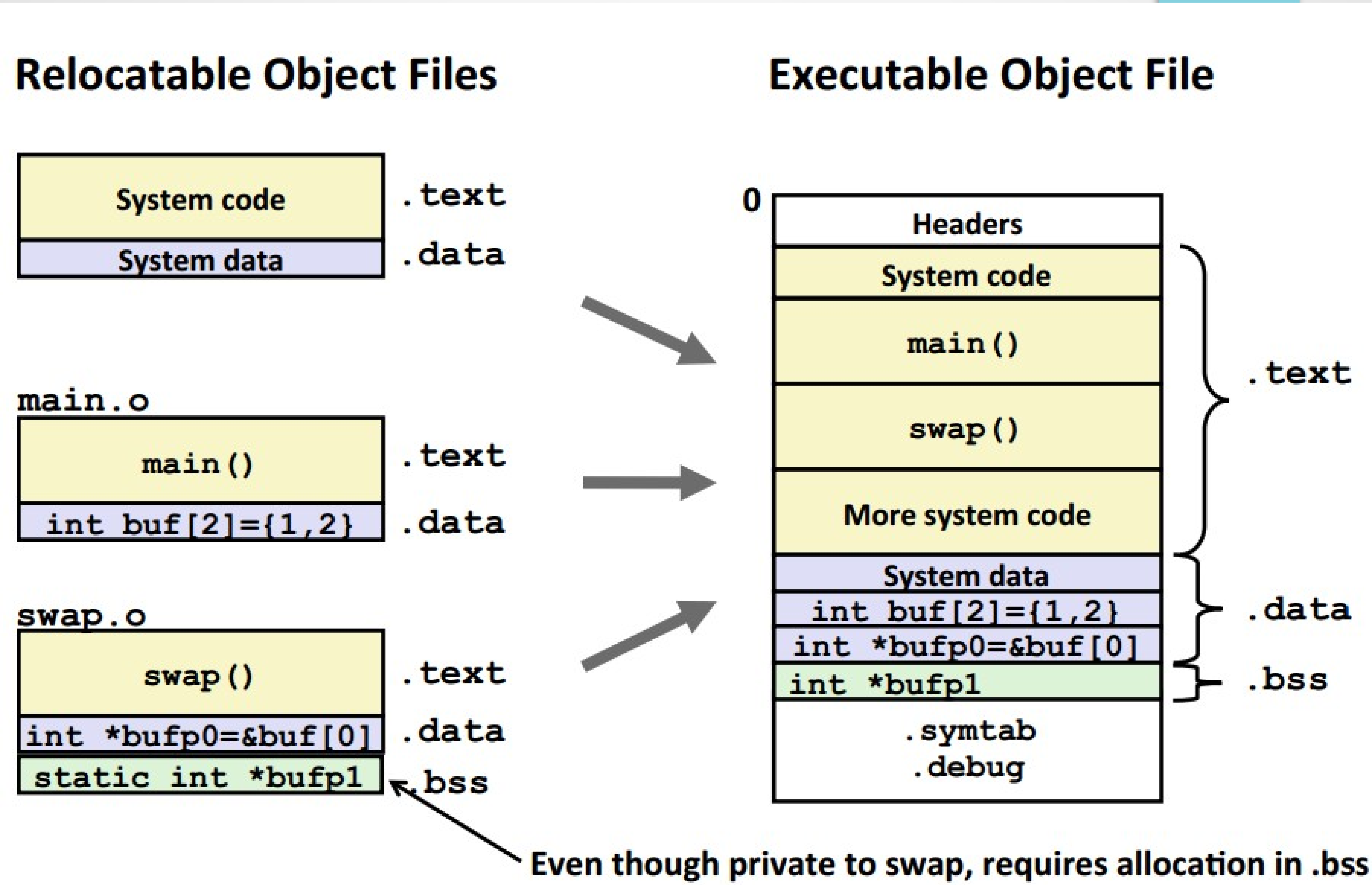
What happens when I try to compile this?
`$ gcc foo3.c bar3.c`
What happens when I run it?

Taken from CSPP textbook p. 665

Notice

- These previous slides should illustrate why using global variables can be very dangerous, and is generally avoided in good coding practice.

Linking - Relocation



Relocation

- Aggregates sections of the same type from each of the different relocatable object files.
- Changes any symbol reference to point to the correct run-time address.

Example

main.c

```
int buf[2] =
   {1,2};

int main()
{
   swap();
   return 0;
}
```

main.o

```
00000000 <main>:
   0:    8d 4c 24 04         lea     0x4(%esp),%ecx
   4:    83 e4 f0            and     $0xfffffff0,%esp
   7:    ff 71 fc            pushl   0xfffffffc(%ecx)
   a:    55                  push    %ebp
   b:    89 e5               mov     %esp,%ebp
   d:    51                  push    %ecx
   e:    83 ec 04            sub     $0x4,%esp
  11:    e8 fc ff ff ff      call    12 <main+0x12>
                        12: R_386_PC32   swap
  16:    b8 00 00 00 00      mov     $0x0,%eax
  1b:    83 c4 04            add     $0x4,%esp
  1e:    59                  pop     %ecx
  1f:    5d                  pop     %ebp
  20:    8d 61 fc            lea     0xfffffffc(%ecx),%esp
  23:    c3                  ret
```

-4

Source: objdump -r -d main.o

```
Disassembly of section .data:

 00000000 <buf>:
   0:    01 00 00 00 02 00 00 00
```

Source: objdump -j .data -d main.o

Source: CMU cs 213 slides

Continued

```
00000000 <main>:
...
   e:    83 ec 04               sub    $0x4,%esp
  11:    e8 fc ff ff ff         call   12 <main+0x12>
                             12: R_386_PC32   swap
  16:    b8 00 00 00 00         mov    $0x0,%eax
 ...
```

Link time :

```
0x8048398 + (-4)
- 0x8048386 = 0xe
```

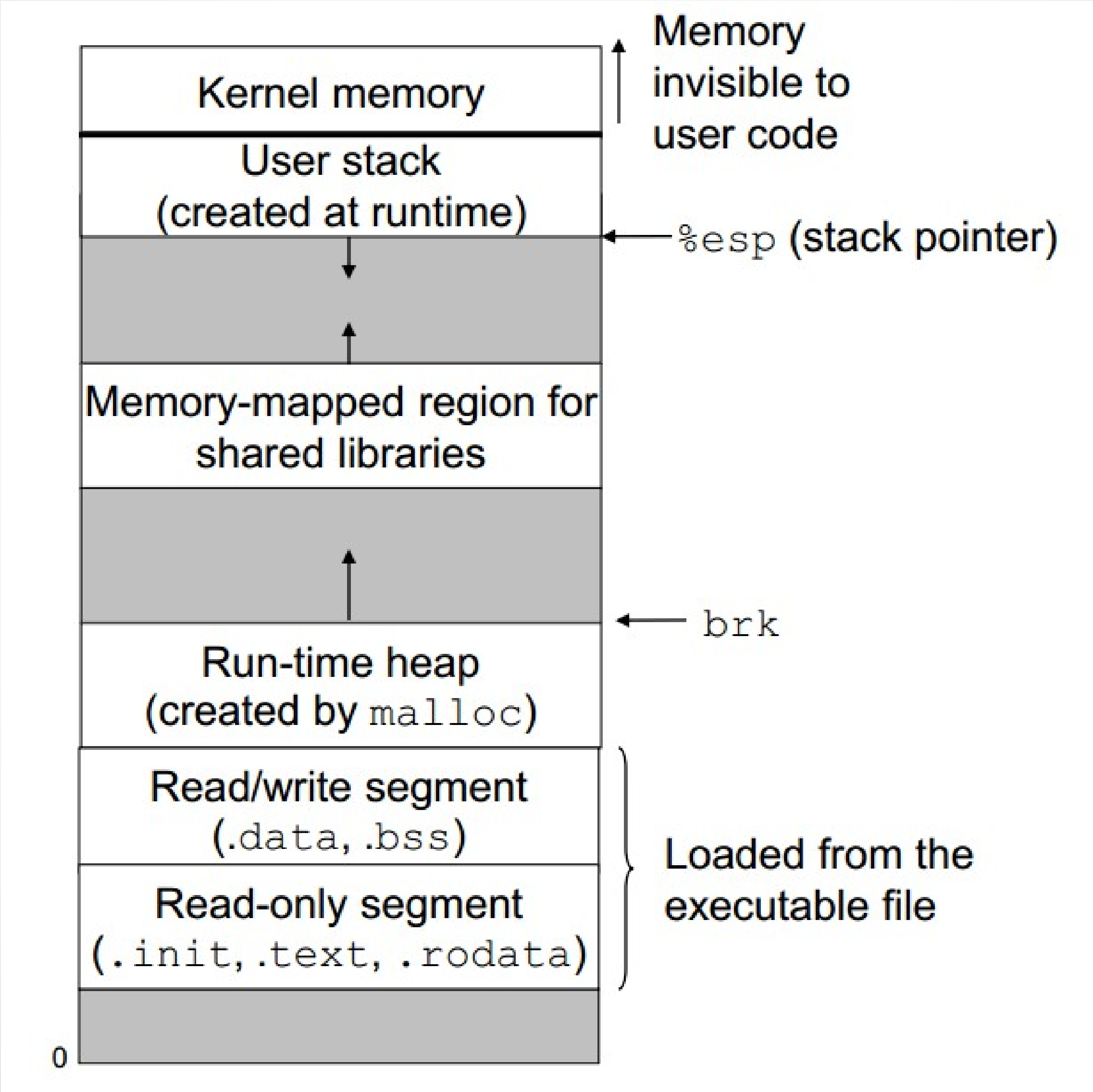
Runtime:

```
0x804838a + 0xe
= 0x8048398
```

```
08048374 <main>:
 8048374:       8d 4c 24 04            lea    0x4(%esp),%ecx
 8048378:       83 e4 f0               and    $0xfffffff0,%esp
 804837b:       ff 71 fc               pushl  0xfffffffc(%ecx)
 804837e:       55                     push   %ebp
 804837f:       89 e5                  mov    %esp,%ebp
 8048381:       51                     push   %ecx
 8048382:       83 ec 04               sub    $0x4,%esp
 8048385:       e8 0e 00 00 00         call   8048398 <swap>
 804838a:       b8 00 00 00 00         mov    $0x0,%eax
 804838f:       83 c4 04               add    $0x4,%esp
 8048392:       59                     pop    %ecx
 8048393:       5d                     pop    %ebp
 8048394:       8d 61 fc               lea    0xfffffffc(%ecx),%esp
 8048397:       c3                     ret
```

Source: CMU cs 213 slides

How is the executable loaded in memory?



How do we see Program Execution?

- When we execute a program, %rip (or %eip) points to the first instruction.
- Each instruction in the program is executed step by step (or jumping as the case may be), using the CPU, RAM, and the CPU registers.
- Life is good.
- But let's look at the bigger picture for a moment.

The Bigger Picture

- Programs and applications are run on top of the operating system right.
- For one thing, what happens the program finishes? Presumably, %rip points to the next program to execute?
- But wait, how can we run multiple programs at once right? There's only one %rip per CPU right? Do they also share the stack?
- Can we go back to not thinking about this?

Nope

- For now, focus on one concern:
  - What happens when something unusual (one could even say... exceptional) occurs.
- What do I mean?
  - Divide by zero
  - Invalid operation
  - OS needs to interrupt or halt program execution.

Exceptions

- “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

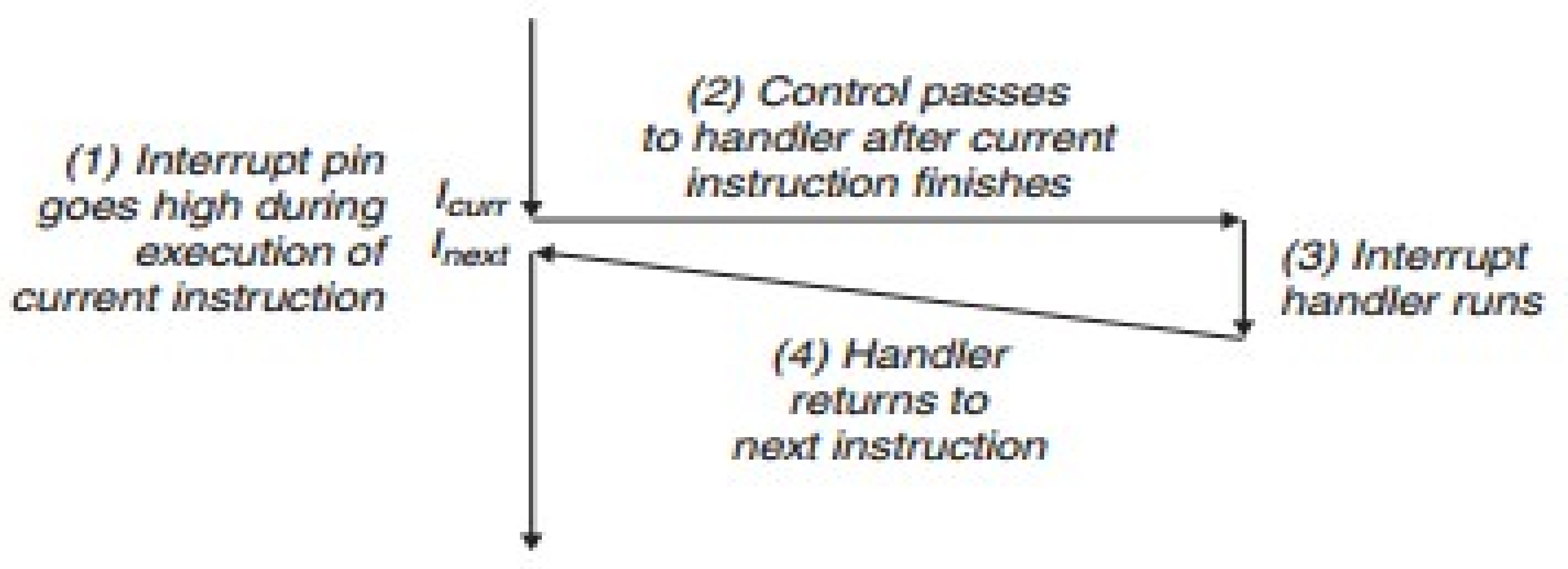
- Most commonly signals from I/O devices.
  - Keyboard key presses.
  - Mouse movement
  - Network adapter activity
  - Etc.
- Asynchronous
  - Occurs independently of currently executing program

Interrupt Handling

- I/O device triggers the “interrupt pin”
- After current instruction, stop executing current program and “control switches to interrupt handler”.
  - What does “control flow” and “passing control” mean?
  - High level: control flow is the execution of a single program and switching control means to allow another program to use the CPU resources to execute.
  - But more on that later

Interrupt Handling

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

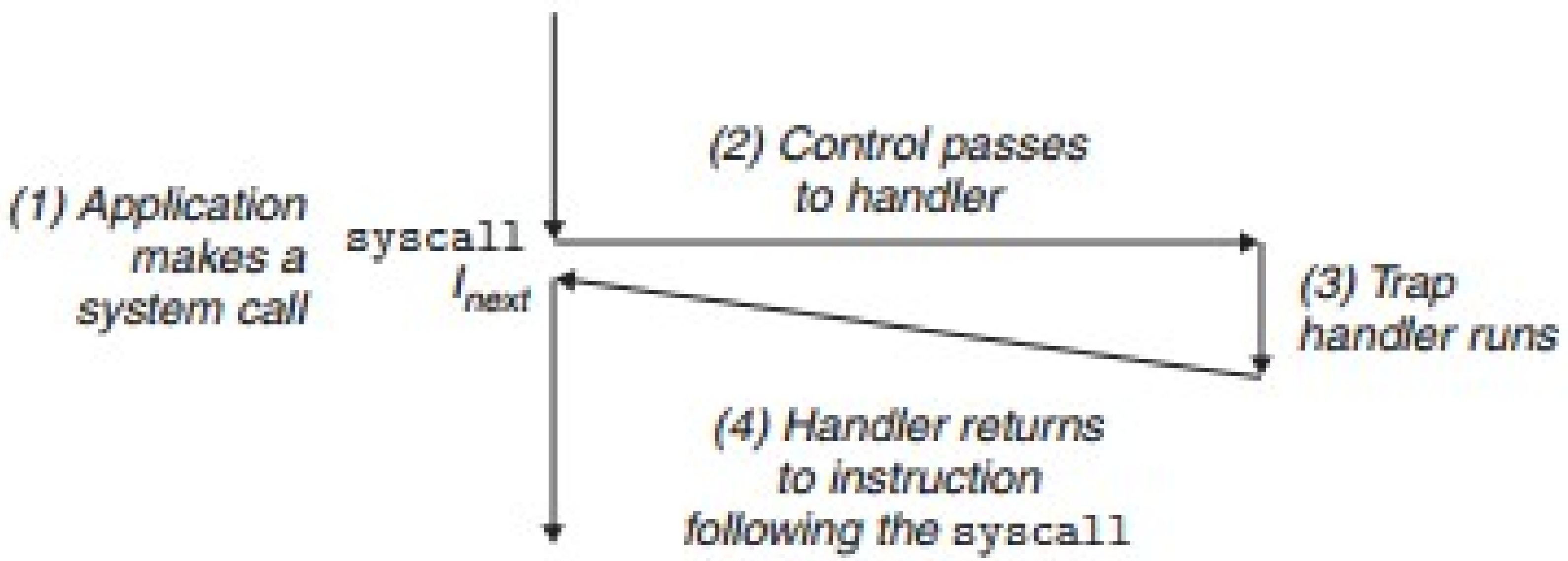


Trap

- 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.
  - Read a file
  - Create a new process
  - Load a new program
- Synchronous: occurs as a result of program instruction.

Trap handling

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

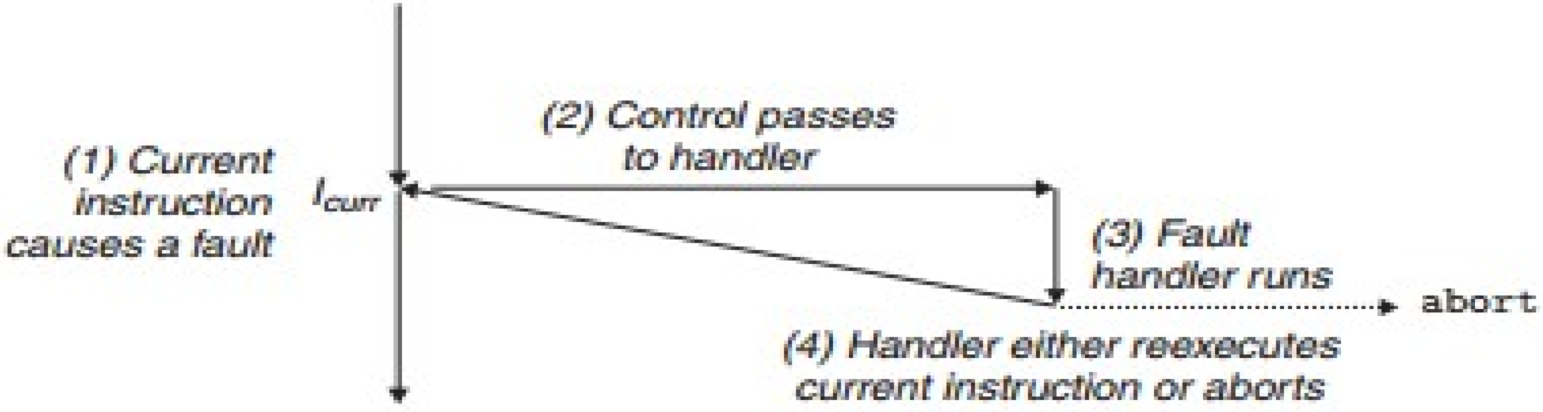


Fault

- 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)
    - Like cache misses but oh so much worse.
    - But more on that later.
- 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.

Trap: syscall

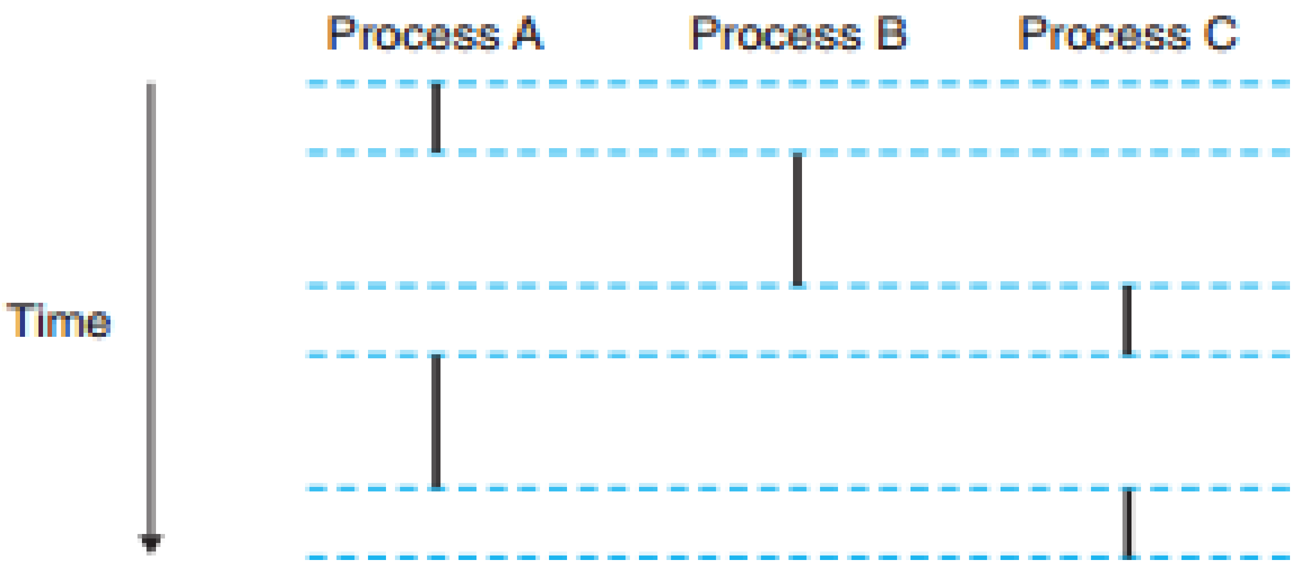
- Linux provides “system calls” which provides services from the OS to an executing program.
- In C, can use syscall function, but can more simply use wrapper functions.
  - read, write, open, close, execve, exit, fork, etc.
- These syscalls will cause a trap.

But hold on...

- How exactly is a program “paused” so that an exception handler can execute?
- For that matter, how can multiple programs run at the same time? There's only one %rip, %rbp, etc.

The dark secret

- Programs on a single CPU do not truly run simultaneously.
- A program generally corresponds to a single process and processes share...



Processes

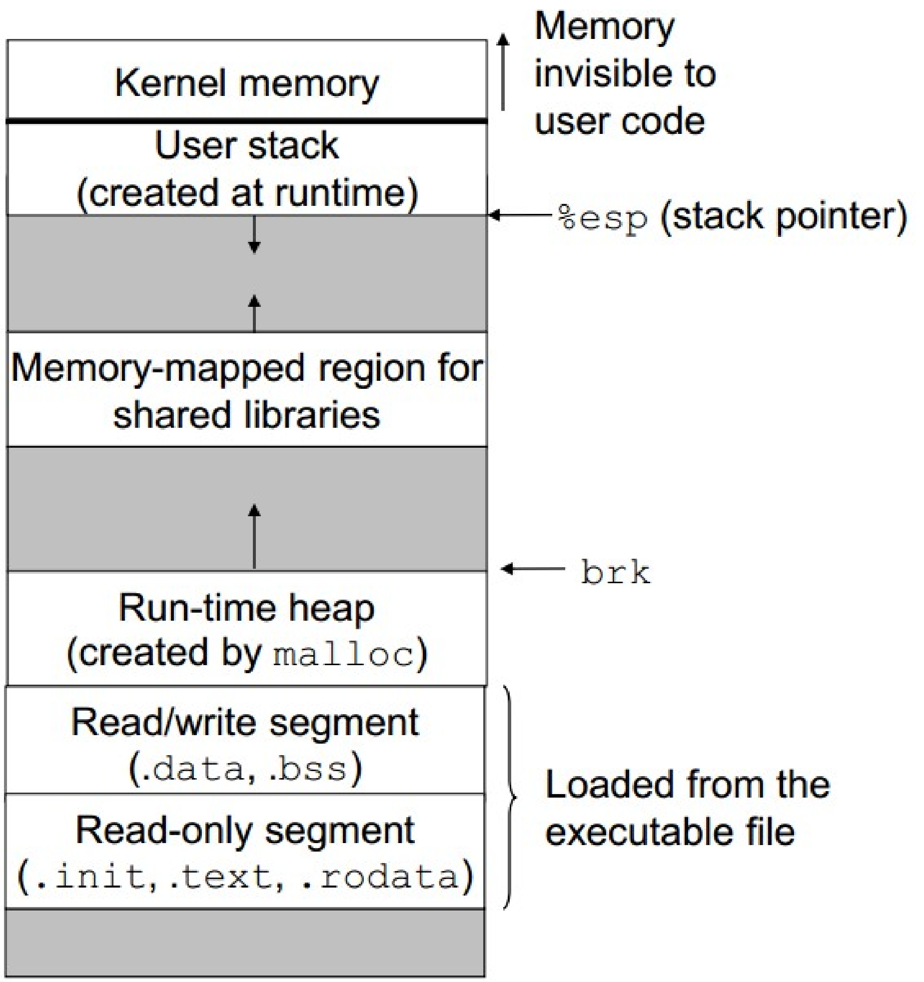
- Programs run atop a process, which appears to provide:
  - Control flow, or exclusive use of the processor to execute instructions
  - Its own memory.
- Every process is special... just like every other process.
- In reality, multiple processes take turn using the processor.

Context Switching

- When the CPU needs to switch to another process to execute, the current process' state (registers, memory) must be stored.
- The state of the next process to execute is restored and the next process runs.
- The previous process is none the wiser
  - This is context switching.
- This is what happens when switching to exception handlers.

What about memory?

- Do we need to save the entire addressable space?
- This is only what the process thinks it has.
- But more on this later (Virtual Memory).



Processes in C

- A program usually corresponds to a single process.
- But, you can actually refer to and create a process from within a program.
- Processes are referred to by a number id or in C, the data type “pid_t”.
- The syscall (wrapper) fork() will create a child process.

Processes in C

```
#include "csapp.h"
int main()
{
  pid_t pid;
  int x = 1;
  pid = fork();

  if (pid == 0) { /* Child */
  printf("child : x=%d\n", ++x);
  exit(0);
  }

  /* Parent */
  printf("parent: x=%d\n", --x);
  exit(0);
}
```

- pid = fork()
- As soon as fork is called a child process with an identical duplicate of the parents memory is made, which one exception.
- pid (parent process) = child process' id
- pid (child process) = 0
- fork() returns 0 for the child
  .

Processes in C Major

```
#include "csapp.h"
int main()
{
  pid_t pid;
  int x = 1;
  pid = fork();

  if (pid == 0) { /* Child */
  printf("child : x=%d\n", ++x);
  exit(0);
  }

  /* Parent */
  printf("parent: x=%d\n", --x);
  exit(0);
}
```

- Both child and process will run the same code in parallel, but now the difference in pid will yield different behavior.