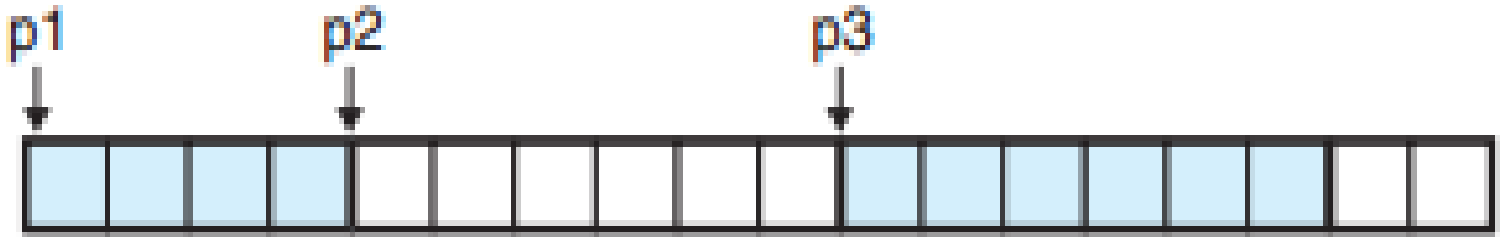
CS 33 Discussion:
Week 9

Dynamic Memory Allocation

- General Needs:
  - Robustness, able to handle arbitrary memory request and any data type.
  - Efficiency, don't waste space or leak memory and don't take too long.

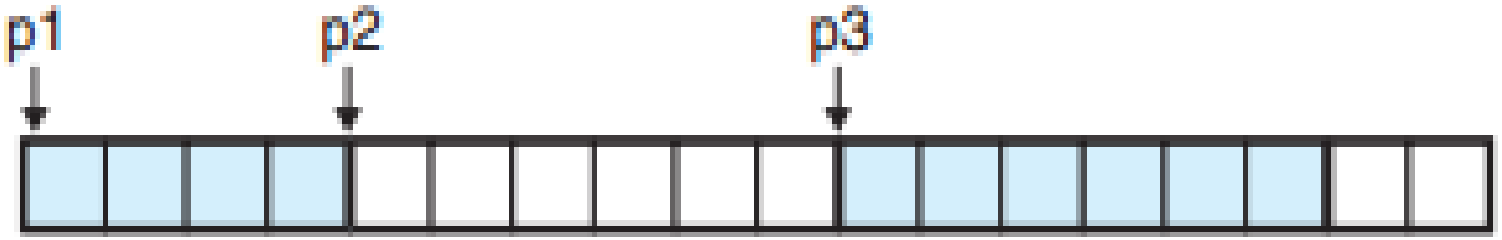
Dynamic Memory Allocation

- Blocks are allocated on the heap.
- Block sizes are always rounded up to some value (for Lab 4, they are multiples of 16 bytes).
- Blocks are aligned (for Lab 4, 16 byte boundary).
- In this book example, alignment is 8 bytes.



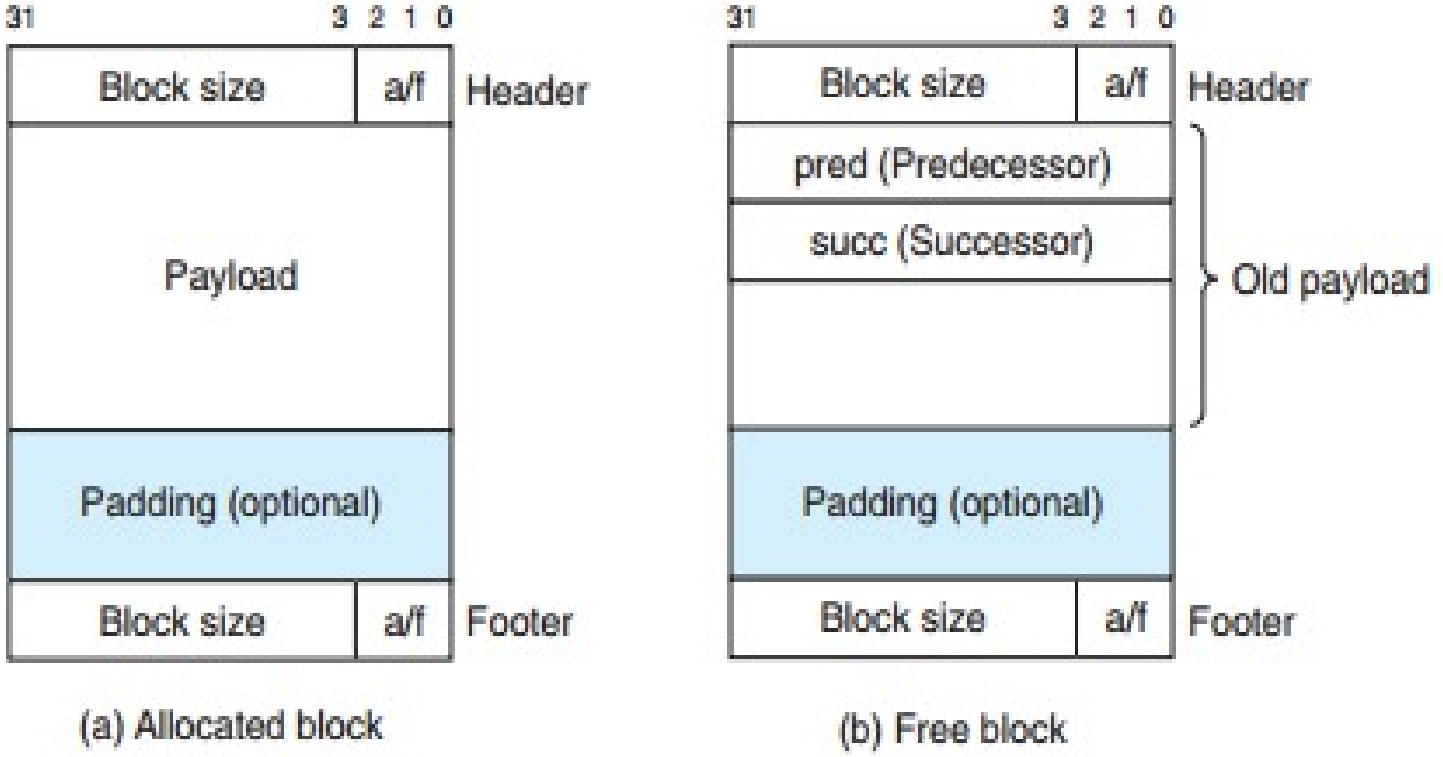
Dynamic Memory Allocation

- This example contains two allocated blocks (p1, p3) and one previously allocated, but now free block(p2).
- If we have a request for 5 bytes of memory, we'd like to use the spot for p2.
- How do we keep track of free blocks?



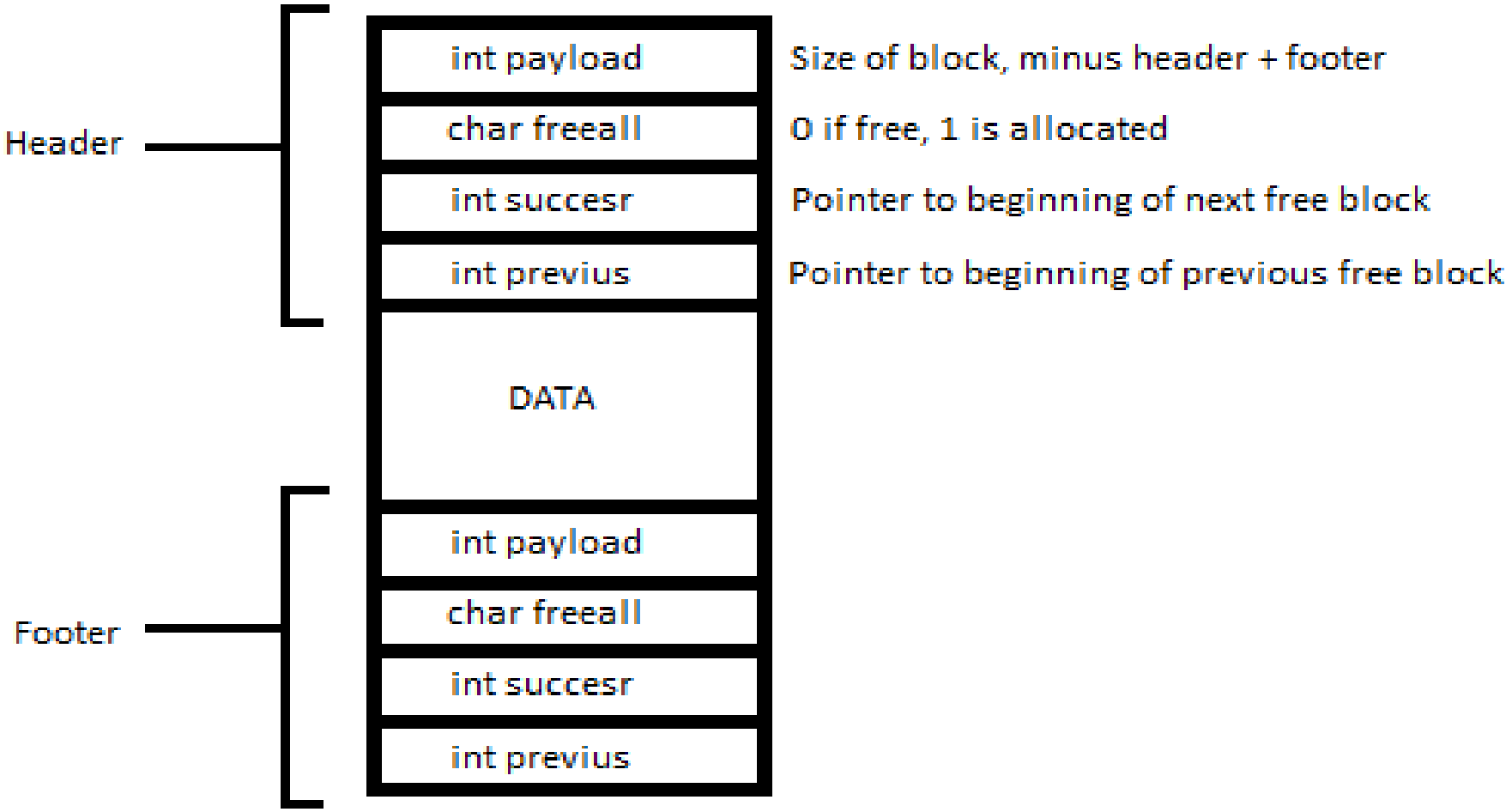
Explicit Free List

- For Lab 4, we use an explicit free list.
- Blocks have the following structure:
- Each free block has a pointer to the address of the next free block.



Explicit Free List

- However, in Lab 4, each block will look like:



How do you find a block?

- First fit
  - Start from beginning of list until you find a free block that is large enough.
  - Use this for the lab!
- Next fit
  - Same as first fit, except restart where you last searched
- Best fit
  - Search entire list to find the best block

Splitting

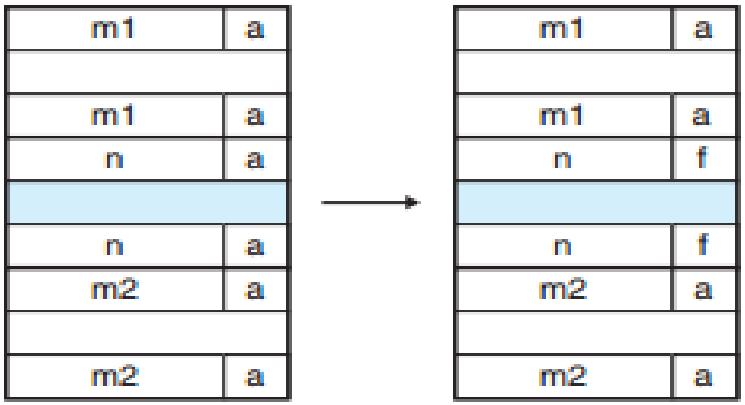
- If you request M bytes of memory and the block you find is large enough to contain M as well as a second block (ie if payload is greater than M + header + footer), split block into two.
- If not, give away the entire block.

Coalescing

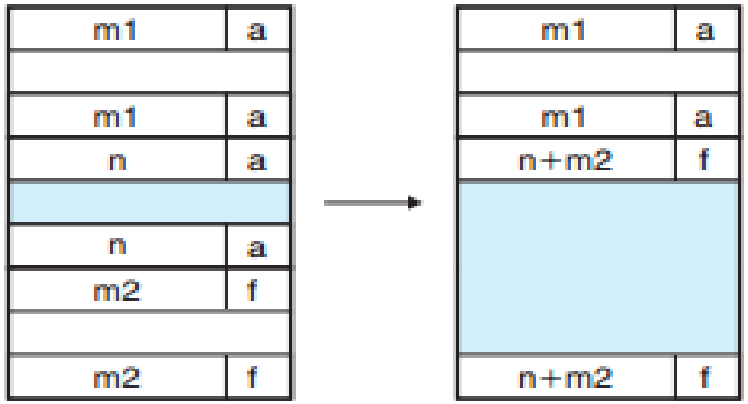
- When multiple adjacent blocks are free, they can be merged to form one large free block.

Coalescing

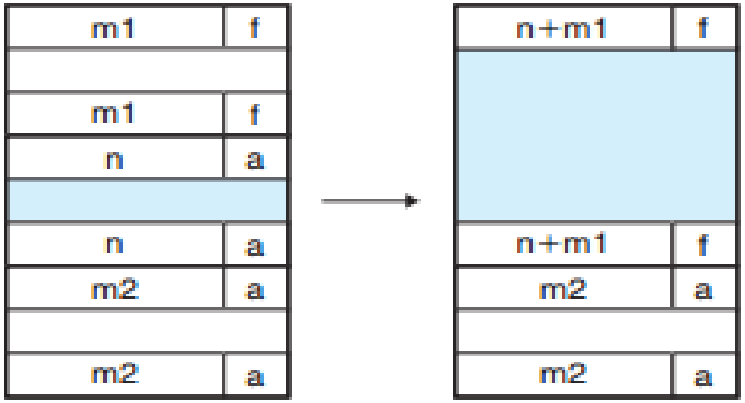
- Four cases, you must account for all of them!



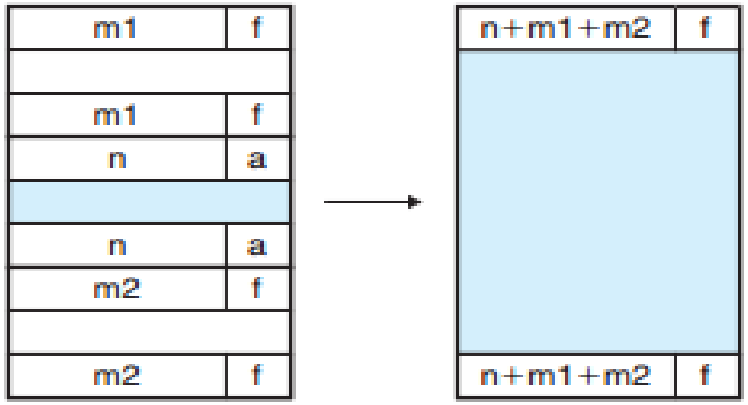
Case 1



Case 2



Case 3



Case 4

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.
- Like?
  - 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

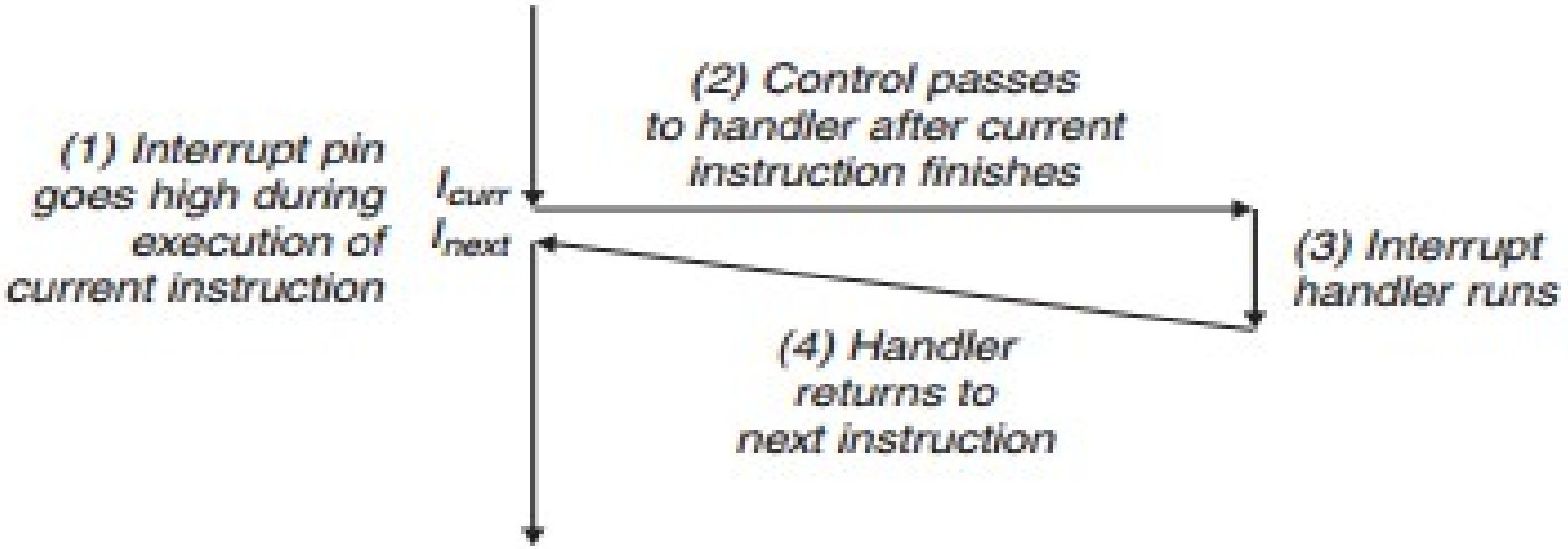
- Most commonly signals from I/O devices.
  - Keyboard key presses.
  - Mouse movement
  - Network adapter activity
  - Etc.
- Asynchronous
  - Occurs independently of currently executing program

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.

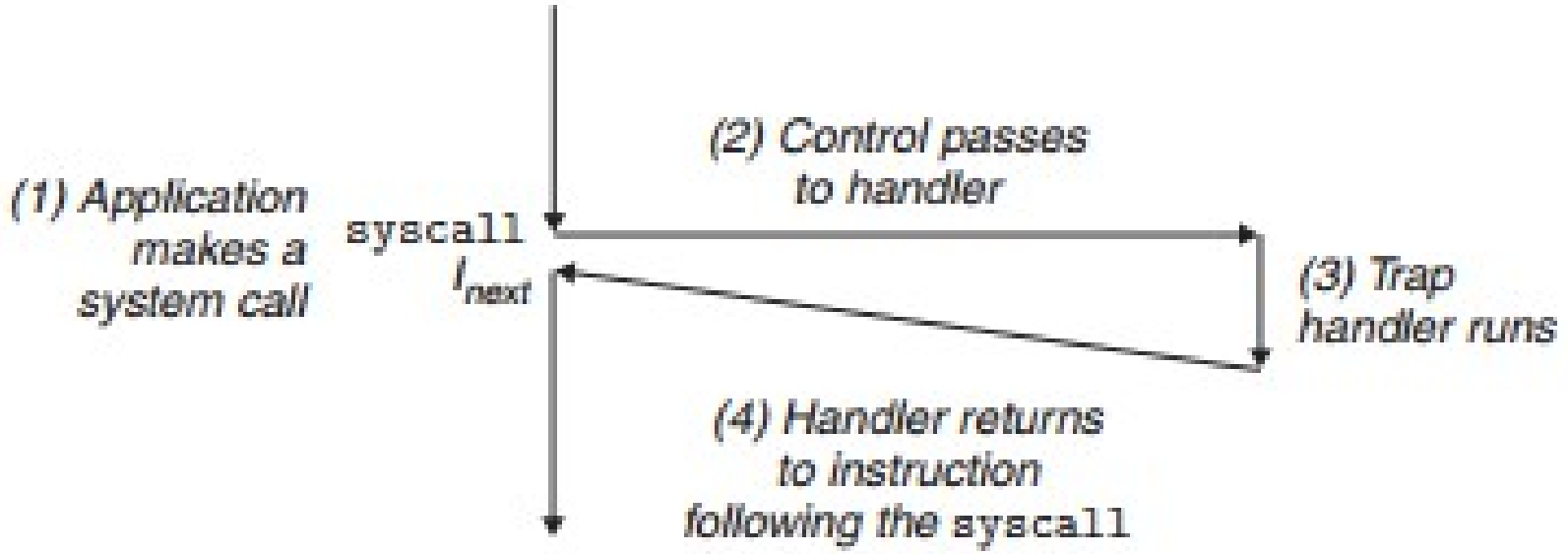


Trap

- 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.

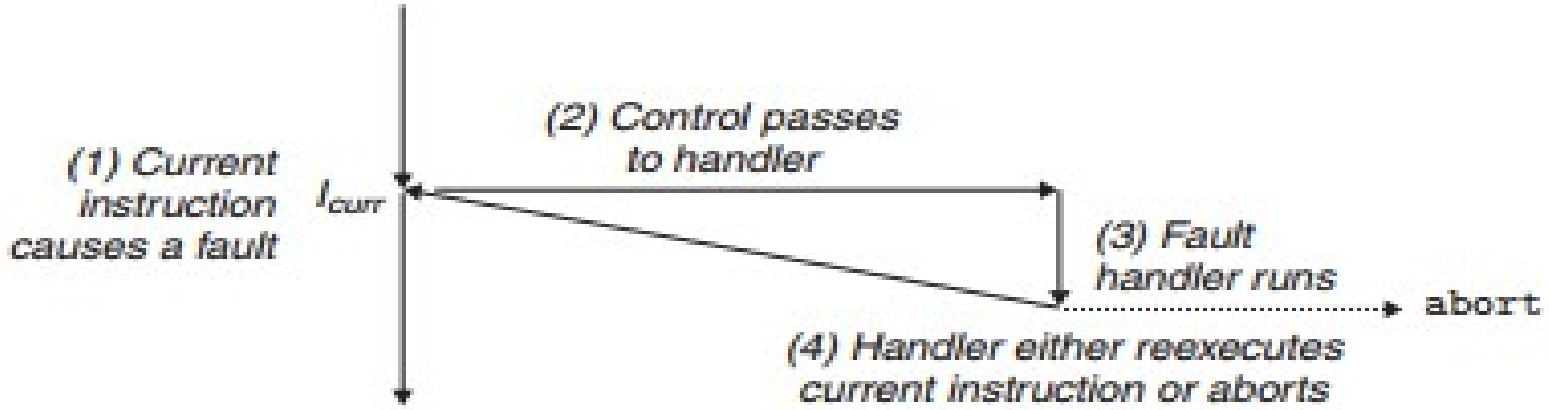


Fault

- 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

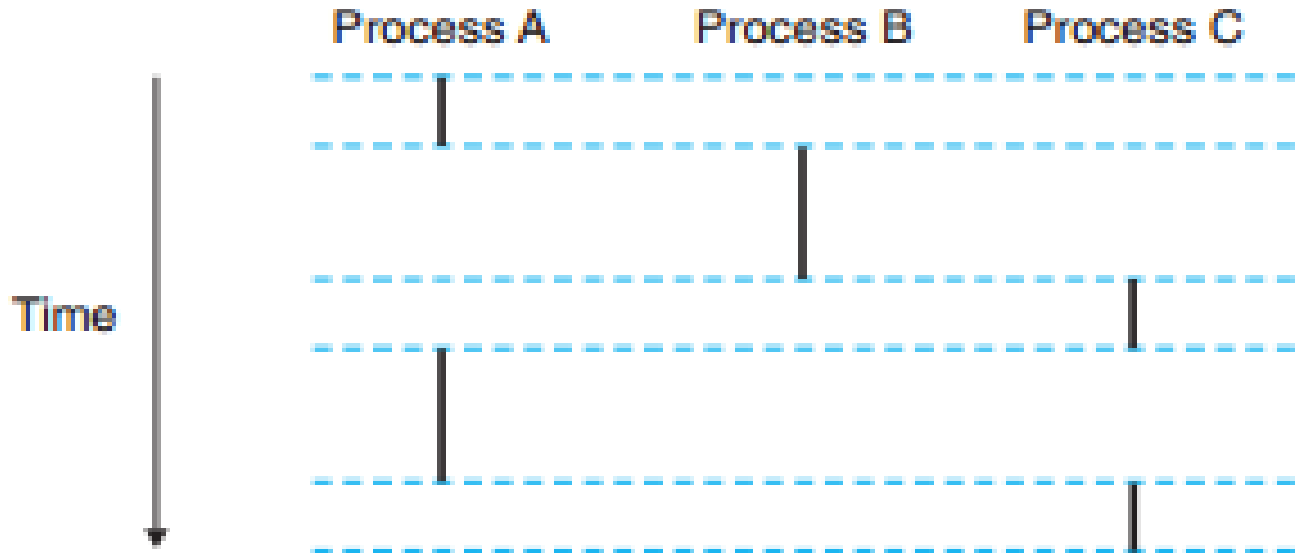
- Linux provides “system calls” which provides services from the OS to an executing program.
- In C, can use syscall function, but can more simply use wrapper functions.
  - read, write, open, close, execve, exit, fork, etc.
- These syscalls will cause a trap.

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

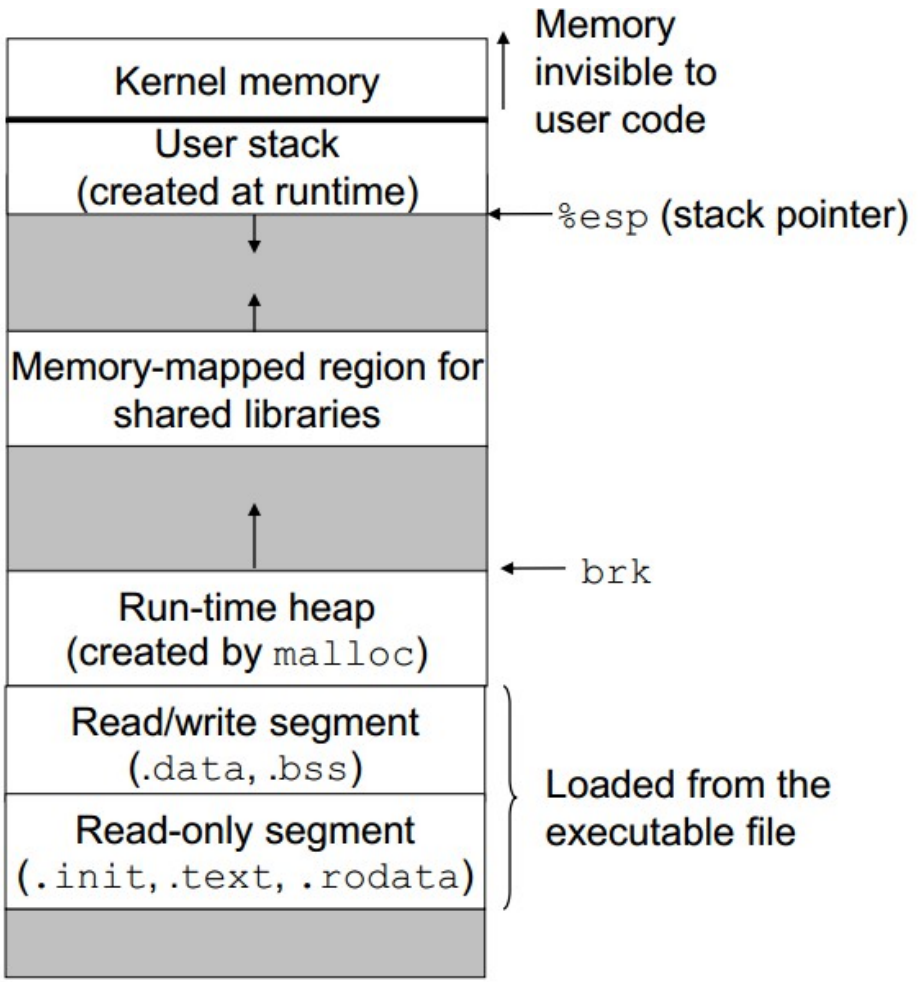
- Programs run atop a process, which appears to provide:
  - Control flow, or exclusive use of the processor to execute instructions
  - Its own memory.
- Every process is special... just like every other process.
- In reality, multiple processes take turn using the processor.

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 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

```
#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.
- What will this program print out?

Process Management

- Child processes that end are not removed from the system automatically and must be “reaped”.
- These “unreaped” processes are formally referred to as “zombies”.
- Children processes will be reaped automatically if the parent terminates.
- Parents can manually reap children processes using the “wait” functions.

Synchronization

- wait()
  - Suspends execution until a child process finishes.
- waitpid(pid_t pid, int *status, int options)
  - Suspends execution until the specified child process exits.
  - If pid = -1, waits for any child process to exit.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  if(fork() == 0)
  {
    printf("a");
  }
  else
  {
    printf("b");
    waitpid(-1, NULL, 0);
  }
  printf("c");
  exit(0);
}
```

- What are the possible outputs of this program?

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  if(fork() == 0)
  {
    printf("a");
  }
  else
  {
    printf("b");
    waitpid(-1, NULL, 0);
  }
  printf("c");
  exit(0);
}
```

- The parent will print “a” and “c”, in that order. The child will print “b” and “c”.
- The parent will not print “c” until the child has printed both “b”, and “c”.
- There is no other guarantee as to their order:
- abcc, bcac, acbc, bacc