# CSE320 System Fundamentals II

## Exceptional Control Flow

YOUNGMIN KWON / TONY MIONE

# Announcements

Exam I : Thurs 4/7

- Covers through Dynamic Memory Allocation II
- Please email me asap if you have a positive COVID test as I will have to arrange to give the same test to you online at the same time if possible

# Exceptional Control Flow

Exceptional Control Flow (ECF)

- Hardware timer goes off or a network packet arrives
- OS transfers control from one process to another
- A process sends a signal to another process

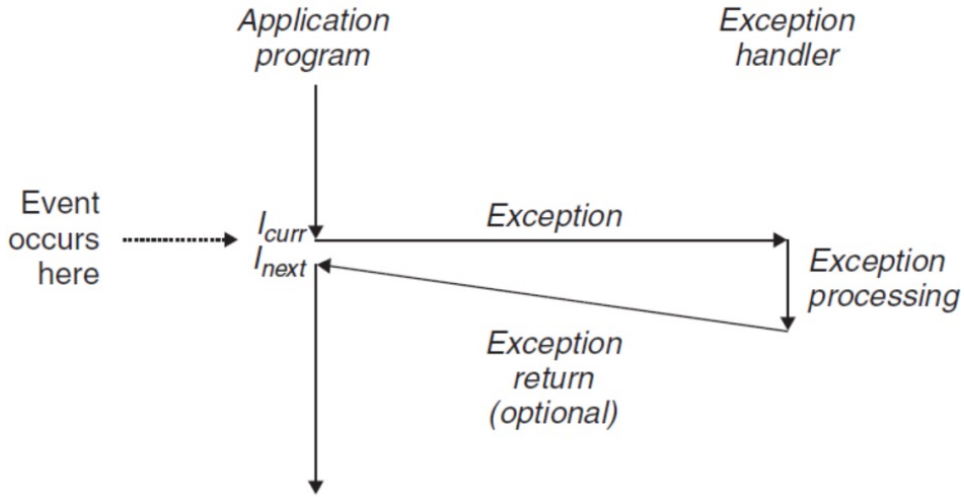
# Exceptions

Exception:

- Change in the control flow due to processor's state change (event)
- Virtual memory page fault, division by zero, I/O complete operation

After handling exceptions

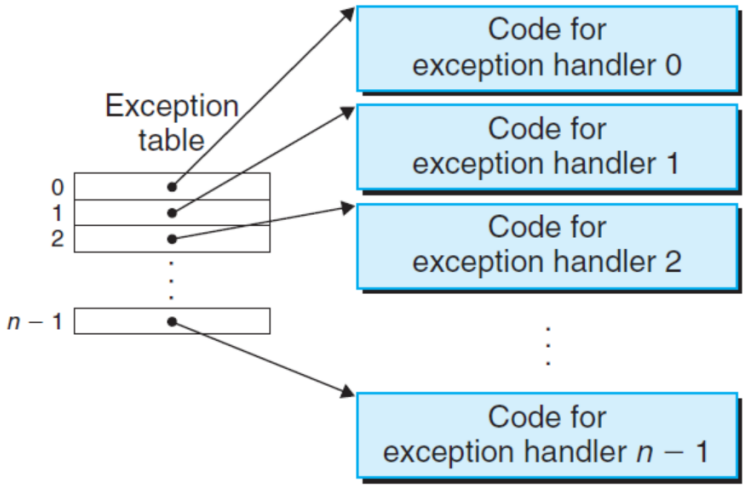
- Return to $I_{curr}$
- Return to $I_{next}$
- Abort the program



# Exception Handling

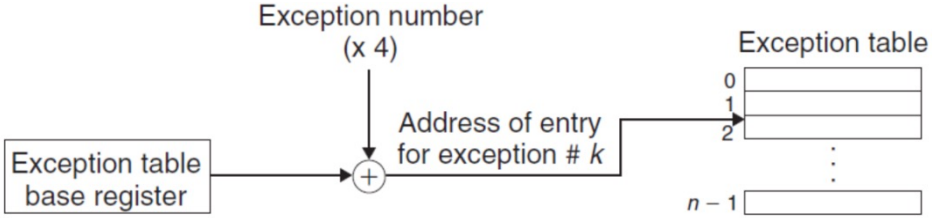
## Exception number

- Set by HW designer (division by 0)
- Set by Kernel designer (system call)




## Exception table

- When an event k occurred, the flow jumps to the $k^{th}$ entry in the exception table
- At system boot time, the OS initializes the jump table



# Exception Handling

Mode switches from user mode to kernel mode if the event occurred in the user mode.

Return address (current instruction or next instruction) and EFLAGS register are pushed onto the kernel's stack

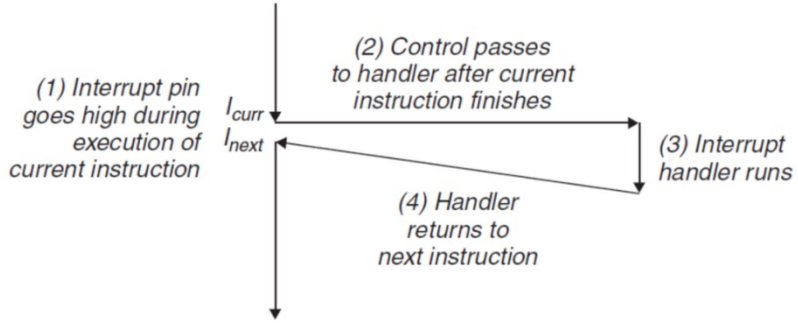
After the handler has processed the event the control returns to the interrupted program

Switch mode back to the user mode if necessary

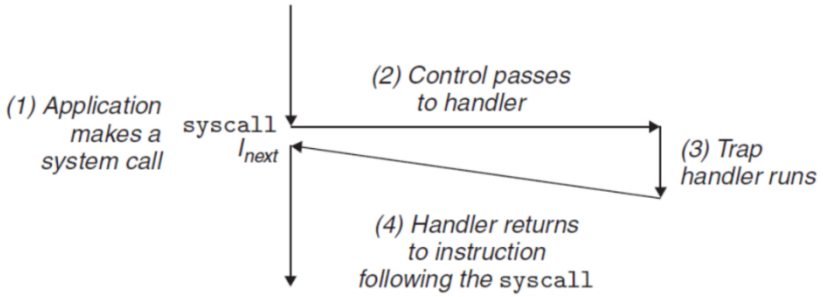
# Classes of Exceptions

| Class | Cause | Async/Sync | Return behavior |
|---|---|---|---|
| Interrupt | Signal from I/O device | Async | Always returns to next instruction |
| Trap | Intentional exception | Sync | Always returns to next instruction |
| Fault | Potentially recoverable error | Sync | Might return to current instruction |
| Abort | Nonrecoverable error | Sync | Never returns |

Interrupt:

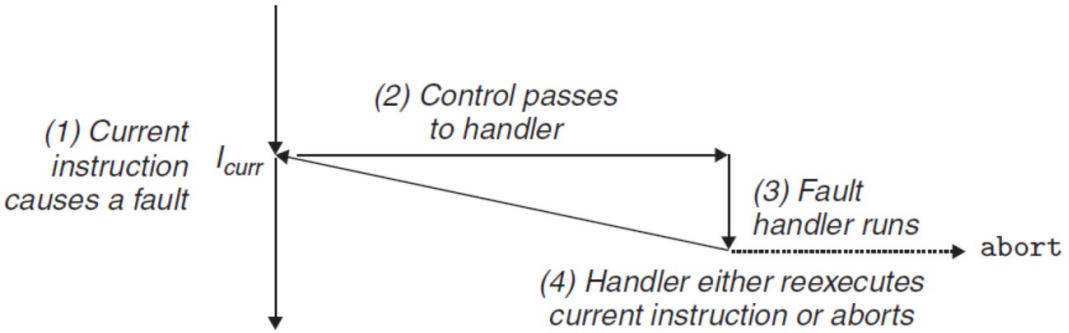



Trap:

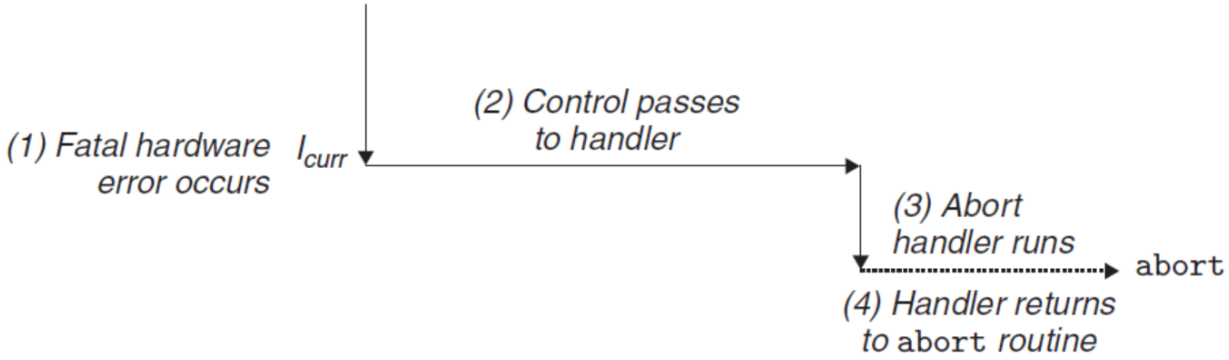


# Classes of Exceptions

Fault:




Abort:



# System Call

syscall:

- System calls are provided by a trapping instruction called syscall
- %rax has the syscall number, up to 6 arguments are %rdi, %rsi, %rdx, %r10, %r8, and %r9
- %rax has the return value, %rcx and %r11 are destroyed.

| Number | Name | Description | Number | Name | Description |
|---|---|---|---|---|---|
| 1 | exit | Terminate process | 27 | alarm | Set signal delivery alarm clock |
| 2 | fork | Create new process | 29 | pause | Suspend process until signal arrives |
| 3 | read | Read file | 37 | kill | Send signal to another process |
| 4 | write | Write file | 48 | signal | Install signal handler |
| 5 | open | Open file | 63 | dup2 | Copy file descriptor |
| 6 | close | Close file | 64 | getppid | Get parent's process ID |
| 7 | waitpid | Wait for child to terminate | 65 | getpgrp | Get process group |
| 11 | execve | Load and run program | 67 | sigaction | Install portable signal handler |
| 19 | lseek | Go to file offset | 90 | mmap | Map memory page to file |
| 20 | getpid | Get process ID | 106 | stat | Get information about file |

# System Call Example

```
1   int main()
2   {
3       write(1, "hello, world\n", 13);
4       exit(0);
5   }
```

```
1   .section .data
2   string:
3     .ascii "hello, world\n"
4   string_end:
5     .equ len, string_end - string

6   .section .text
7   .globl main
8   main:
     First, call write(1, "hello, world\n", 13)
9     movl $4, %eax          System call number 4
10    movl $1, %ebx          stdout has descriptor 1
11    movl $string, %ecx     Hello world string
12    movl $len, %edx        String length
13    int $0x80              System call code

     Next, call exit(0)
14    movl $1, %eax          System call number 0
15    movl $0, %ebx          Argument is 0
16    int $0x80              System call code
```

# Processes

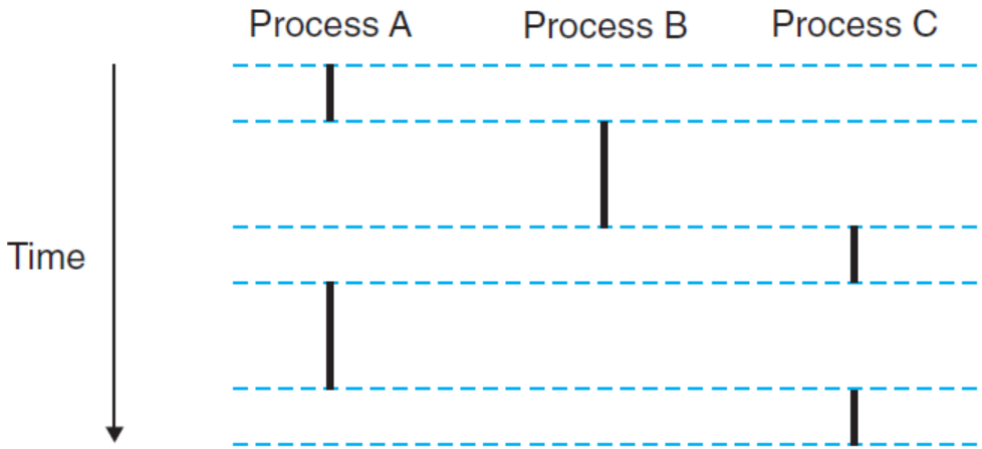
Process

- An instance of a program in execution
- Each program runs in the context of a process
- A context comprises:
  - The program's code,
  - Data in memory, stack, registers,
  - Open file descriptors…

# Logical Control Flow

A process provides each program with an illusion that it has exclusive use of the processor although many other programs are running concurrently

Each vertical bar in the figure below represents a portion of the logical control flow for a process



# Concurrent Flows

Concurrent Flow:

- A logical flow whose execution overlaps in time with other flows
- A and B run concurrently; A and C run concurrently.
- B and C do not run concurrently

Parallel flow:

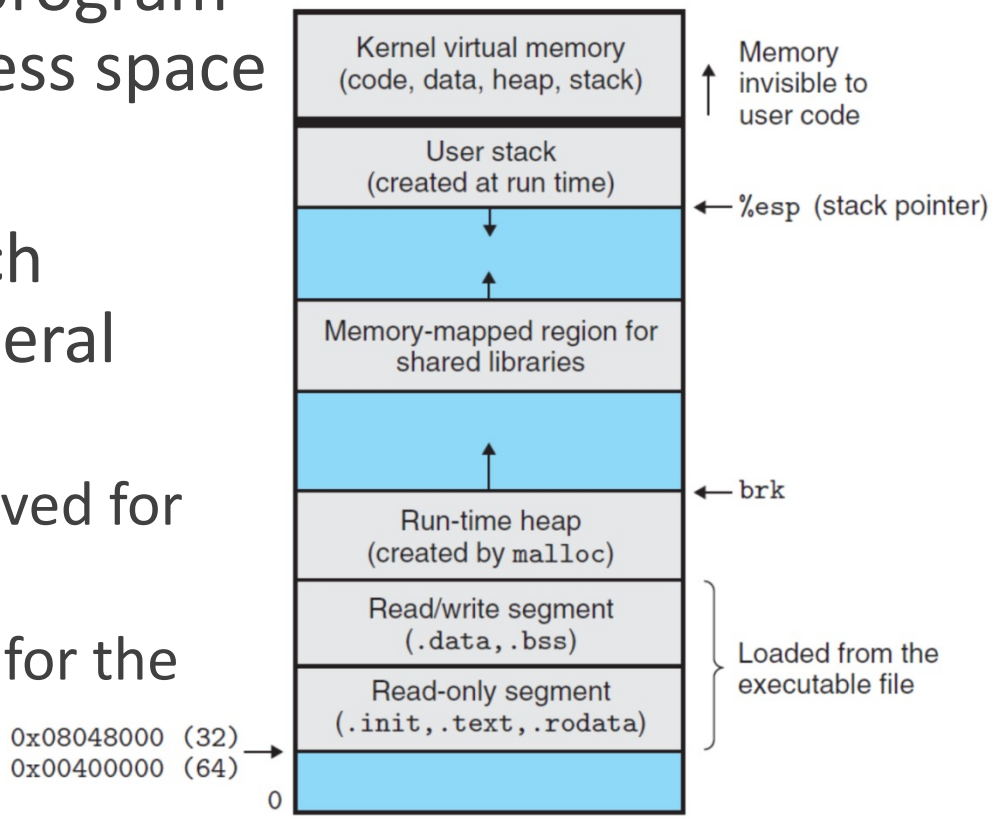
- If two flows are running concurrently on different processor cores or computers

# Private Address Space

A process provides each program with its own private address space

The address space for each process has the same general organization

- The bottom portion is reserved for the user program
- The top portion is reserved for the kernel



# User mode and Kernel mode

Processors typically provide a mode bit

Kernel mode

- When the mode bit is set
- Can execute any instructions and can access any memory location

User mode

- When the mode bit is not set
- Cannot execute privileged instructions: halt the processor, change the mode bit, I/O operation
- Cannot reference kernel code or data

# User mode and Kernel mode

To access kernel data structure from user mode

- /proc file system
- /proc/cpuinfo, /proc/process-id/maps, …

```
$ cat /proc/5558/maps
00400000-004e1000 r-xp 00000000 08:01 7209108            /bin/bash
006e0000-006e1000 r--p 000e0000 08:01 7209108            /bin/bash
006e1000-006ea000 rw-p 000e1000 08:01 7209108            /bin/bash
006ea000-006f0000 rw-p 00000000 00:00 0
00753000-00f8a000 rw-p 00000000 00:00 0                  [heap]
7f240fa46000-7f240fa51000 r-xp 00000000 08:01 7867530    /lib/x86_64-linux-gnu/libnss_files-2.15.so
7f240fa51000-7f240fc50000 ---p 0000b000 08:01 7867530    /lib/x86_64-linux-gnu/libnss_files-2.15.so
7f240fe5c000-7f240fe5d000 r--p 0000a000 08:01 7867532    /lib/x86_64-linux-gnu/libnss_nis-2.15.so
7f2410c90000-7f2410c92000 rw-p 00023000 08:01 7867529    /lib/x86_64-linux-gnu/ld-2.15.so
...
7fffcac50000-7fffcac71000 rw-p 00000000 00:00 0          [stack]
7fffcad4f000-7fffcad50000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
$
```

# Context Switches

Kernel maintains a context for each process

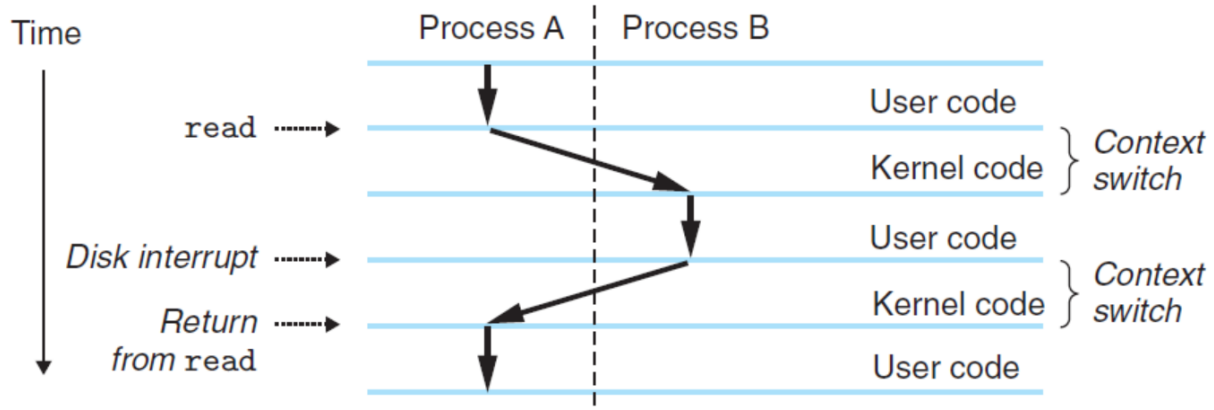
A context consists of

- registers,
- PC,
- user stack,
- kernel stack,
- kernel data structure (page table, process table, file table)

# Context Switches

Context switch

- Kernel can decide to preempt the current running process
- Scheduler picks the next process to run
- Kernel transfers the control to the new process through context switching



# Process Control

Obtaining Process IDs

```
pid_t getpid(void);
pid_t getppid(void);
```

Creating a process

```
pid_t fork(void);
```

Terminating a process

```
void exit(int status);
```

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    pid_t pid = fork();
    if(pid == 0)
        printf("child:  pid: %d, "
               "ppid: %d\n",
               getpid(), getppid());
    else
        printf("parent: pid: %d, "
               "ppid: %d, child: %d\n",
               getpid(), getppid(), pid);
    exit(0);
}
```

# Process Control

When a process terminates

- The kernel does not remove it from the system immediately
- The process is kept around in a terminated state (zombie process) until it is reaped by its parent

Waits for its children to terminate and reap it

```
pid_t waitpid(pid_t pid, int *statusp, int options)
pid_t wait(int *statusp); // wait for any children
                          // to terminate
```

# Process Control

```
pid_t waitpid(pid_t pid, int *statusp, int options)
```

- Function behaves differently depending on range of pid:
  - **pid > 0** – Wait for the single process with an id matching the value of pid
  - **pid == 0** = Wait for any child process with a process group ID equal to the process group ID of the calling process
  - **pid == -1** – Wait for any child process to terminate
- options
- **WNOHANG** – return immediately if no child has terminated (returns 0 if no child has terminated)
- **WCONTINUED** – return if a child has terminated or if a child was continued with a SIGCONT signal.

# Process Control

- **WIFEXITED(status)** – Returns true if the process returned by waitpid terminated normally
- **WIFSIGNALED(status)** – Returns true if the process returned by waitpid terminated due to an uncaught signal
- **WIFSTOPPED(status)** – Returns true if the process returned by waitpid is currently stopped
- **WIFCONTINUED(status)** – Returns true if process returned by waitpid was continued with SIGCONT
- **WEXITSTATUS(status)** – Returns the status of the normally terminated child
- **WTERMSIG(status)** – Returns number of the signal that caused process termination
- **WSTOPSIG(status)** – Returns number of the signal that caused process to stop

# Process Control

Putting process to sleep

```
//sleep for sec seconds or until signaled
unsigned int sleep(unsigned int sec);

//sleep until signaled
int pause(void);
```

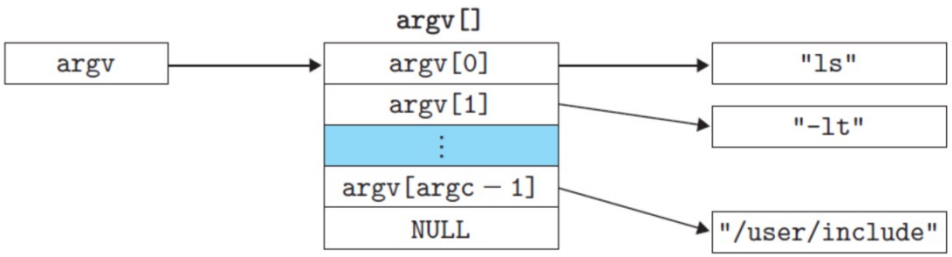
# fork and waitpid

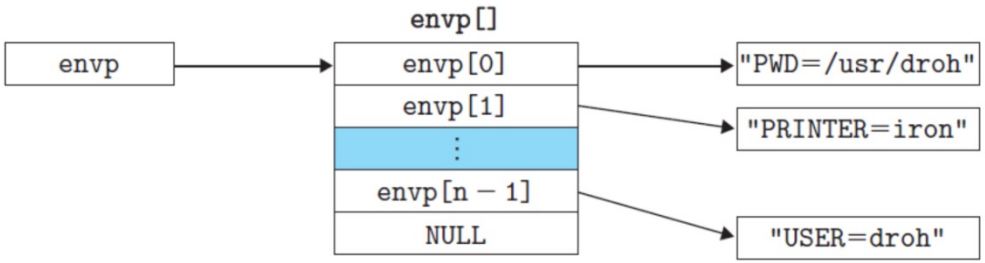
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    int status, i;
    pid_t pid;
    for(i = 0; i < 10; i++)
        if((pid = fork()) == 0)
            exit(100+i);

    while((pid = waitpid(-1, &status, 0)) > 0) {
        if(WIFEXITED(status))
            printf("pid: %d, status: %d\n", pid, WEXITSTATUS(status));
        else
            printf("pid: %d, abnormal termination\n", pid);
    }

    exit(0);
}
```

# Process Control

execve loads and runs a new program in the context of the current process

```
int execve(const char *filename,
           const char *argv[],
           const char *envp[]);
```



# A Simple Shell Program

```
#include "csapp.h"
#define MAXARGS   128

/* Function prototypes */
void eval(char *cmdline);
int parseline(char *buf, char **argv);
int builtin_command(char **argv);

int main()
{
    char cmdline[MAXLINE]; /* Command line */

    while (1) {
        /* Read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* Evaluate */
        eval(cmdline);
    }
}
```

# A Simple Shell Program

```
/* eval - Evaluate a command line */
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
```

# A Simple Shell Program

```
        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}

/* If first arg is a builtin command, run it and return true */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
        return 1;
    return 0;                     /* Not a builtin command */
}
```

# A Simple Shell Program

```
/* parseline - Parse the command line and build the argv array */
int parseline(char *buf, char **argv)
{
    char *delim;         /* Points to first space delimiter */
    int argc;            /* Number of args */
    int bg;              /* Background job? */

    buf[strlen(buf)-1] = ' ';  /* Replace trailing '\n' with space */
    while (*buf && (*buf == ' ')) /* Ignore leading spaces */
        buf++;

    /* Build the argv list */
    argc = 0;
    while ((delim = strchr(buf, ' '))) {
        argv[argc++] = buf;
        *delim = '\0';
        buf = delim + 1;
        while (*buf && (*buf == ' ')) /* Ignore spaces */
               buf++;
    }
    argv[argc] = NULL;

    if (argc == 0)  /* Ignore blank line */
        return 1;

    /* Should the job run in the background? */
    if ((bg = (*argv[argc-1] == '&')) != 0)
        argv[--argc] = NULL;

    return bg;
}
```

# Questions?