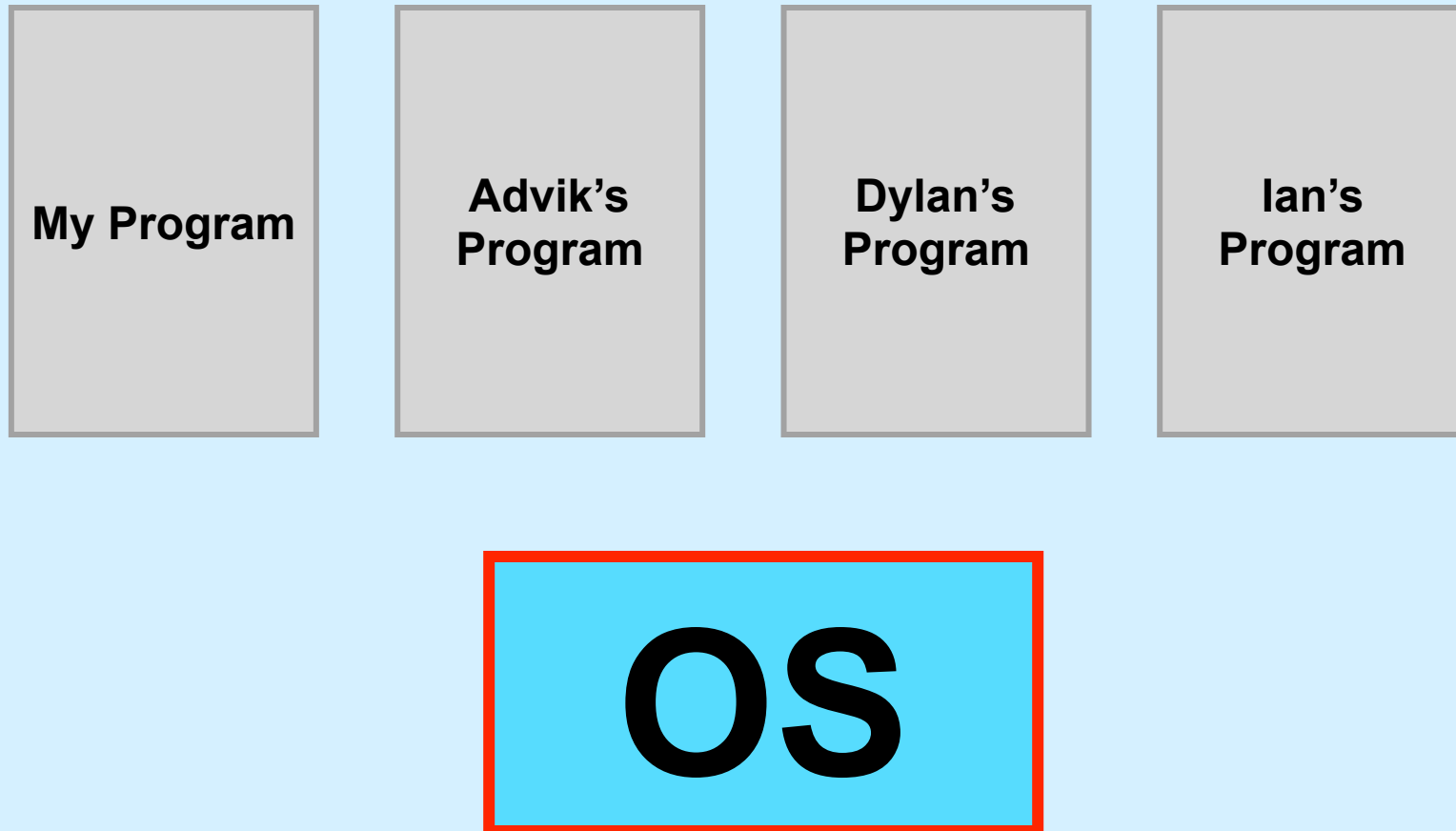


# CS 33

## Architecture and the OS

# The Operating System



# Processes

- **Containers for programs**
  - **virtual memory**
    - » **address space**
  - **scheduling**
    - » **one or more threads of control**
  - **file references**
    - » **open files**
  - **and lots more!**

# Idiot Proof ...

```
int main( ) {  
    int i;  
    int A[1];  
  
    for (i=0; ; i++)  
        A[rand()] = i;  
}
```

Can I clobber  
Advik's  
program?

**Advik's  
Program**

# Fair Share

```
int runforever( ){  
    while(1)  
        ;  
}  
  
int main( ) {  
    runforever();  
}
```

Can I  
monopolize  
the  
processor?

**Dylan's  
Program**

# Architectural Support for the OS

- **Not all instructions are created equal ...**
  - non-privileged instructions
    - » can affect only current program
  - privileged instructions
    - » may affect entire system
- **Processor mode**
  - user mode
    - » can execute only non-privileged instructions
  - privileged mode
    - » can execute all instructions

# Which Instructions Should Be Privileged?

- I/O instructions
- Those that affect how memory is mapped
- Halt instruction
- Some others ...

# Who Is Privileged?

- **You're not**
  - and neither is anyone else
- **The operating-system kernel runs in privileged mode**
  - nothing else does
  - not even super user on Unix or administrator on Windows



# Entering Privileged Mode

- **How is OS invoked?**
  - very carefully ...
  - strictly in response to interrupts and exceptions
  - (booting is a special case)

# Interrupts and Exceptions

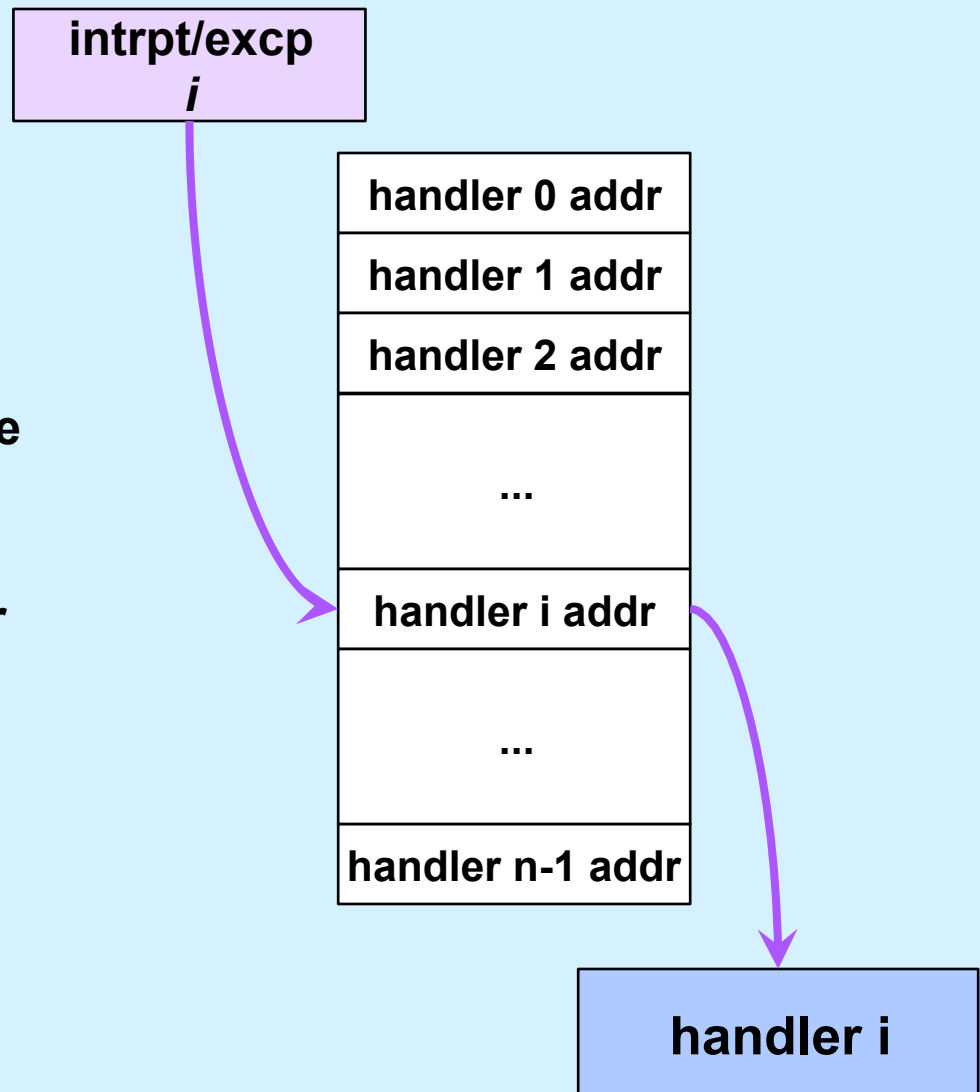
- **Things don't always go smoothly ...**
  - I/O devices demand attention
  - timers expire
  - programs demand OS services
  - programs demand storage be made accessible
  - programs have problems
- **Interrupts**
  - demand for attention by external sources
- **Exceptions**
  - executing program requires attention

# Exceptions

- **Traps**
  - “intentional” exceptions
    - » execution of special instruction to invoke OS
  - after servicing, execution resumes with next instruction
- **Faults**
  - a problem condition that is normally corrected
  - after servicing, instruction is re-tried
- **Aborts**
  - something went dreadfully wrong ...
  - not possible to re-try instruction, nor to go on to next instruction

# Interrupt and Exception Handling

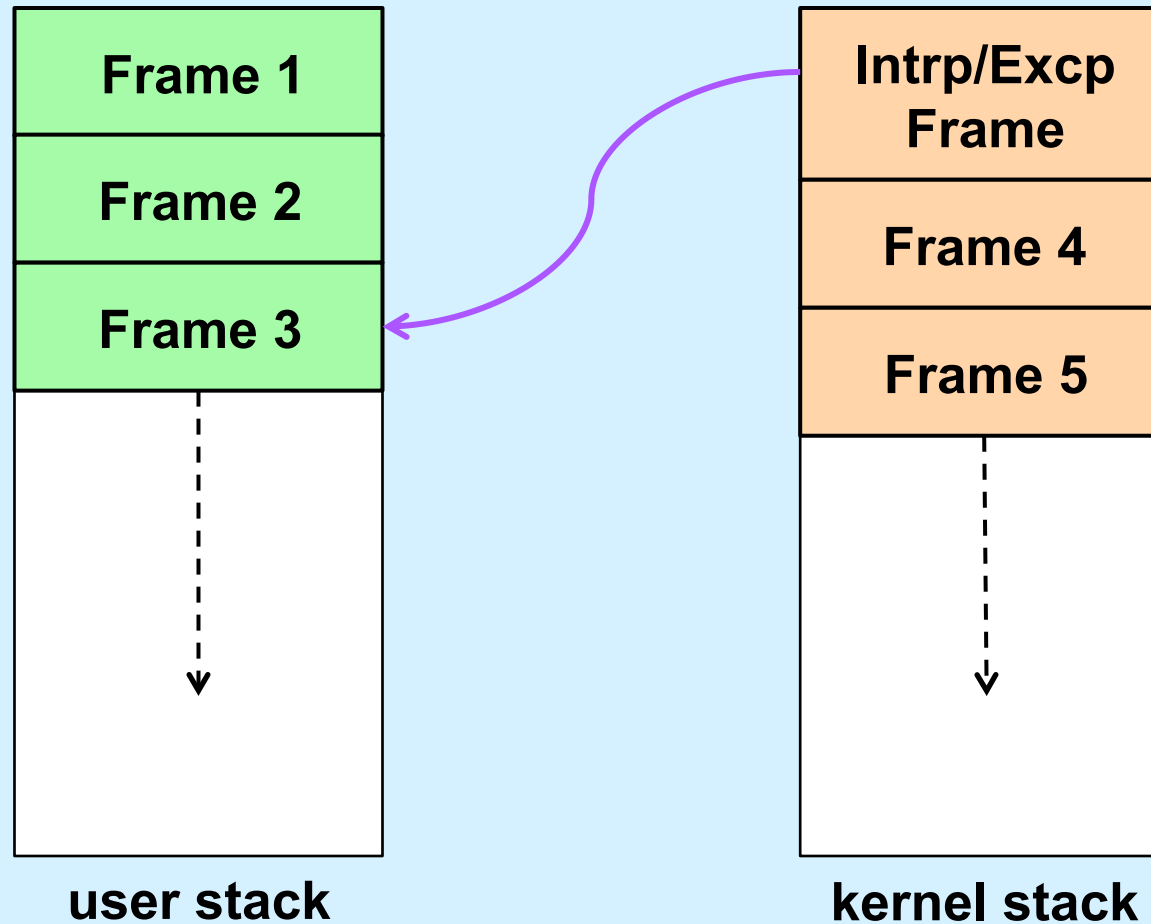
- **Interrupt or exception invokes handler (in OS)**
  - via interrupt and exception vector
    - » one entry for each possible interrupt/exception
      - contains
        - address of handler
  - code executed in privileged mode
    - » but code is part of the OS



# Entering and Exiting

- **Entering/exiting interrupt/exception handler more involved than entering/exiting a procedure**
  - **must deal with processor mode**
    - » **switch to privileged mode on entry**
    - » **switch back to previous mode on exit**
  - **stack in kernel must be different from stack in user program**
    - » **why?**

# One Stack Per Mode



# Quiz 1

**If an interrupt occurs, which general-purpose registers must be pushed onto the kernel stack?**

- a) none
- b) callee-save registers
- c) caller-save registers
- d) all

# Back to the x86 ...

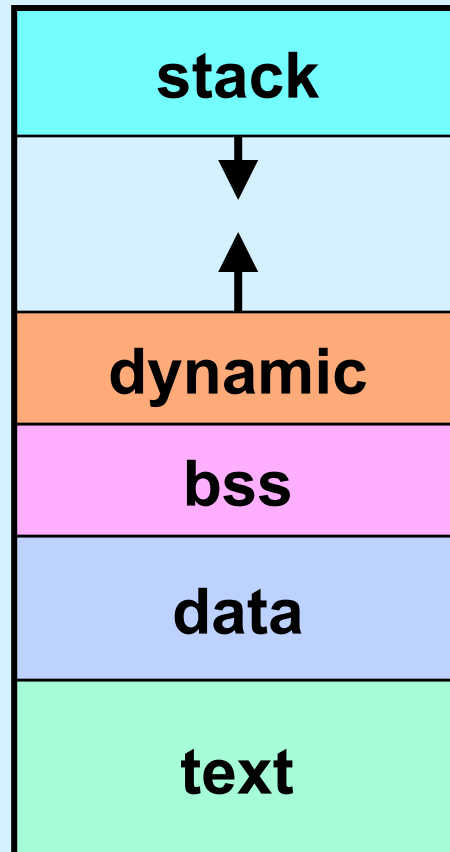
- **It's complicated**
  - more than it should be, but for historical reasons ...
- **Not just privileged and non-privileged modes, but four “privilege levels”**
  - **level 0**
    - » most privileged, used by OS kernel
  - **level 1**
    - » not normally used
  - **level 2**
    - » not normally used
  - **level 3**
    - » least privileged, used by application code



# Processes and Address Spaces

- The *process*
  - OS entity representing a computation
    - » sort of a virtual computer
    - » encapsulates
      - processors (threads)
      - storage
        - containing code and data
          - address space
          - open files
      - identity

# The Unix Address Space

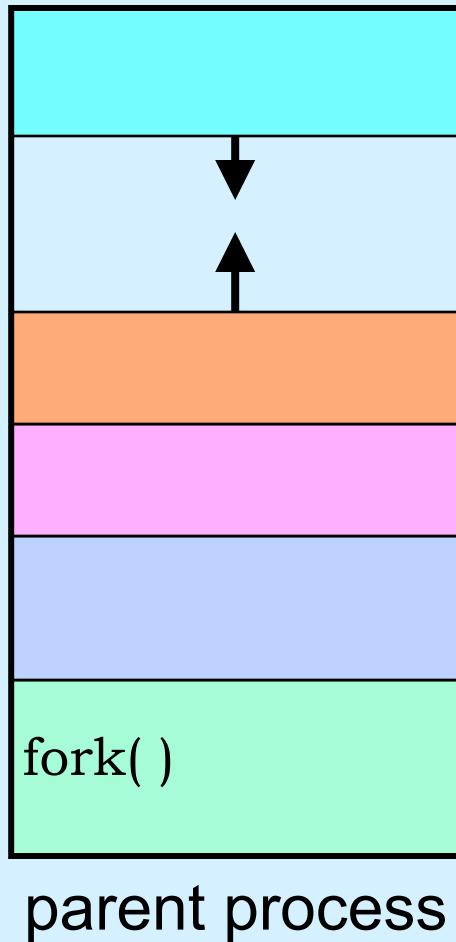


# Creating Your Own Processes

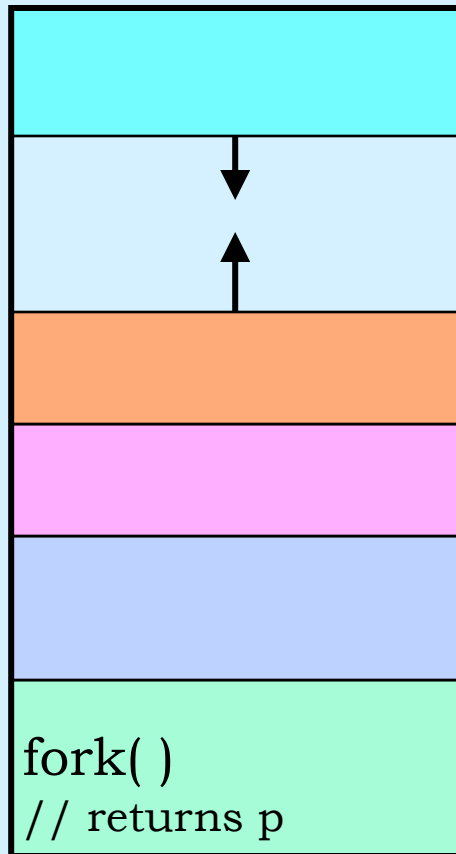


```
#include <unistd.h>
int main( ) {
    pid_t pid;
    if ((pid = fork()) == 0) {
        /* new process starts
           running here */
    }
    /* old process continues
       here */
}
```

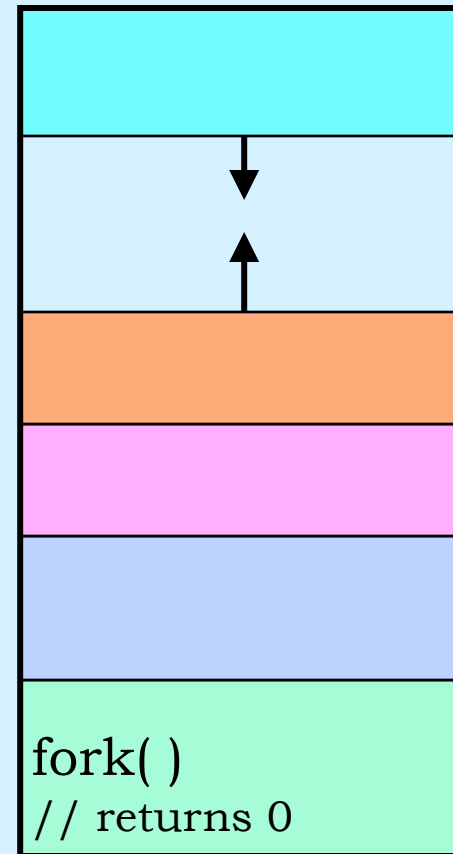
# Creating a Process: Before



# Creating a Process: After



parent process



child process  
(pid = p)

# Quiz 2

The following program

- a) terminates quickly
- b) runs forever

```
int flag;
int main() {
    while (flag == 0) {
        if (fork() == 0) {
            flag = 1;
            exit(0); // causes process to terminate
        }
    }
}
```

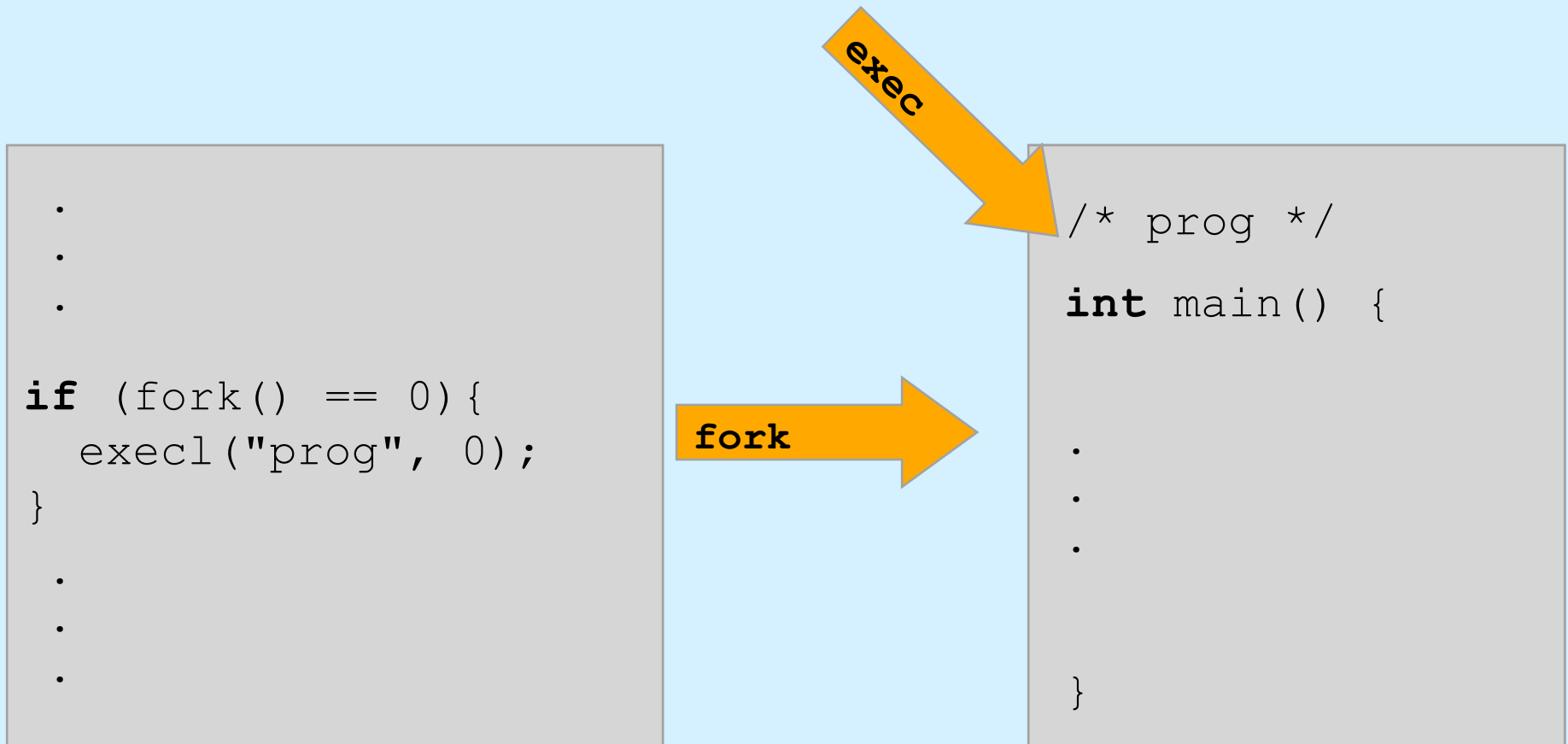
# Process IDs

```
int main( ) {  
    pid_t pid;  
    pid_t ParentPid = getpid();  
  
    if ((pid = fork()) == 0) {  
        printf("%d, %d, %d\n",  
                pid, ParentPid, getpid());  
        return 0;  
    }  
    printf("%d, %d, %d\n",  
            pid, ParentPid, getpid());  
    return 0;  
}
```

**parent prints:**  
27355, 27342, 27342

**child prints:**  
0, 27342, 27355

# Putting Programs into Processes





# Exec

- **Family of related routines**
  - we concentrate on one:
    - » **execv(program, argv)**

```
char *argv[] = {"MyProg", "12", (void *)0};  
if (fork() == 0) {  
    execv("./MyProg", argv);  
}
```

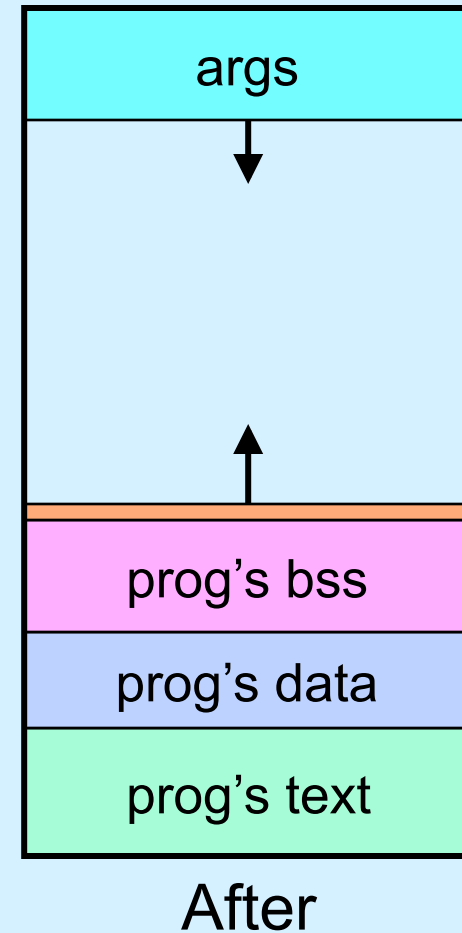
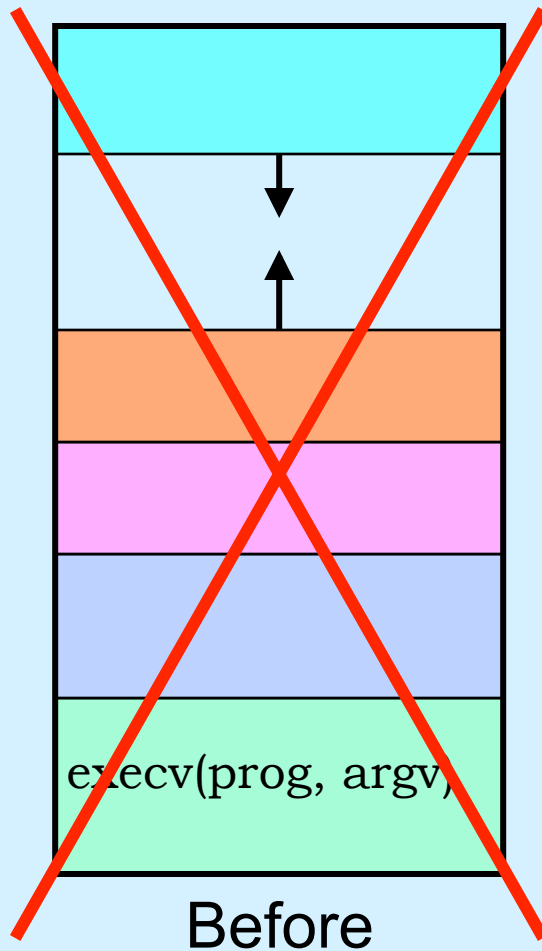
First “real”  
argument

End of  
list

Name of the file that  
contains the program

argv[0] is the name  
of the program

# Loading a New Image



# A Random Program ...

```
int main(int argc, char *argv[]) {  
    int i;  
    int stop = atoi(argv[1]);  
    for (i = 0; i < stop; i++)  
        printf("%d\n", rand());  
    return 0;  
}
```

# Passing It Arguments

- **From the shell**

```
$ random 12
```

- **From a C program**

```
if (fork() == 0) {  
    char *argv[] = {"random", "12", (void *)0};  
    execv("./random", argv);  
}
```

## Quiz 3

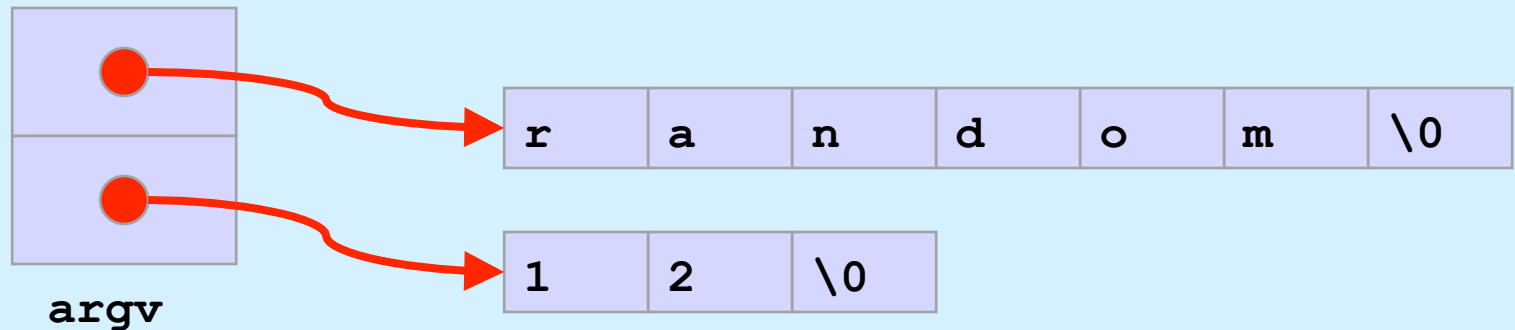
```
if (fork() == 0) {  
    char *argv[] = {"random", "12", (void *)0};  
    execl(".", "random", argv);  
    printf("random done\n");  
}
```

The *printf* statement will  
always be executed, after  
random completes.

- a) yes
- b) no

# Receiving Arguments

```
int main(int argc, char *argv[]) {  
    int i;  
    int stop = atoi(argv[1]);  
  
    for (i = 0; i < stop; i++)  
        printf("%d\n", rand());  
  
    return 0;  
}
```



# Not So Fast ...

- How does the shell invoke your program?

```
if (fork() == 0) {  
    char *argv = {"random", "12", (void *)0};  
    execv("./random", argv);  
}  
/* what does it do here??? */
```

# Wait

```
#include <unistd.h>
#include <sys/wait.h>
...
pid_t pid;
int status;
...
if ((pid = fork()) == 0) {
    char *argv[] = "random", "12", (void *)0};
    execv("./random", argv);
}
waitpid(pid, &status, 0);
```

---



# Exit

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main( ) {
    pid_t pid;
    int status;
    if ((pid = fork()) == 0) {
        if (do_work() == 1)
            exit(0); /* success! */
        else
            exit(1); /* failure ... */
    }
    waitpid(pid, &status, 0);
    /* low-order byte of status contains exit code.
       WEXITSTATUS(status) extracts it */
}
```

The diagram illustrates the flow of exit codes from the code to a box labeled "exit code". Red lines connect the exit codes 0 and 1 in the `exit()` calls to the "exit code" box. Another red line connects the `&status` argument in the `waitpid()` call to the same box, indicating that the status variable contains the exit code.

# Shell: To Wait or Not To Wait ...

```
$ who
```

```
    if ((pid = fork()) == 0) {  
        char *argv[] = {"who", 0};  
        execv("who", argv);  
    }  
    waitpid(pid, &status, 0);  
    ...
```

```
$ who &
```

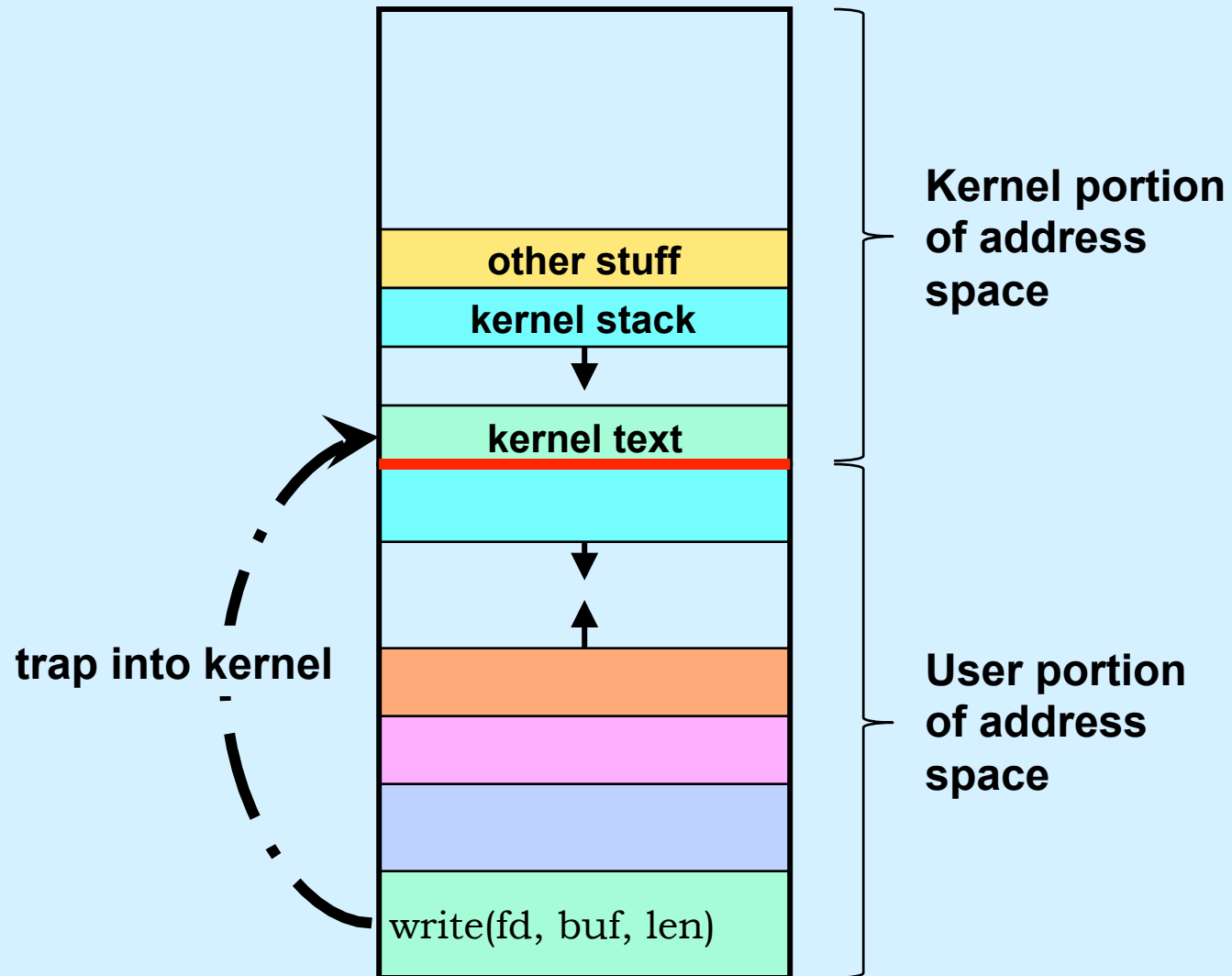
```
    if ((pid = fork()) == 0) {  
        char *argv[] = {"who", 0};  
        execv("who", argv);  
    }  
    ...
```

# System Calls

- Sole direct interface between user and kernel
- Implemented as library routines that execute *trap* instructions to enter kernel
- Errors indicated by returns of  $-1$ ; error code is in global variable *errno*

```
if (write(fd, buffer, bufsize) == -1) {  
    // error!  
    printf("error %d\n", errno);  
    // see perror  
}
```

# System Calls



# Multiple Processes

