Process

* Some materials adapted from CS:APP, Prof. Bryant, O'Hallaron, Doeppner, Galvin, etc.

Processes

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

Obtaining Process IDs

- `pid_t getpid(void)`
  - Returns PID of current process
- `pid_t getppid(void)`
  - Returns PID of parent process

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 */
}
```

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);
}
```

`fork` Example: Two consecutive `forks`

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

forks.c

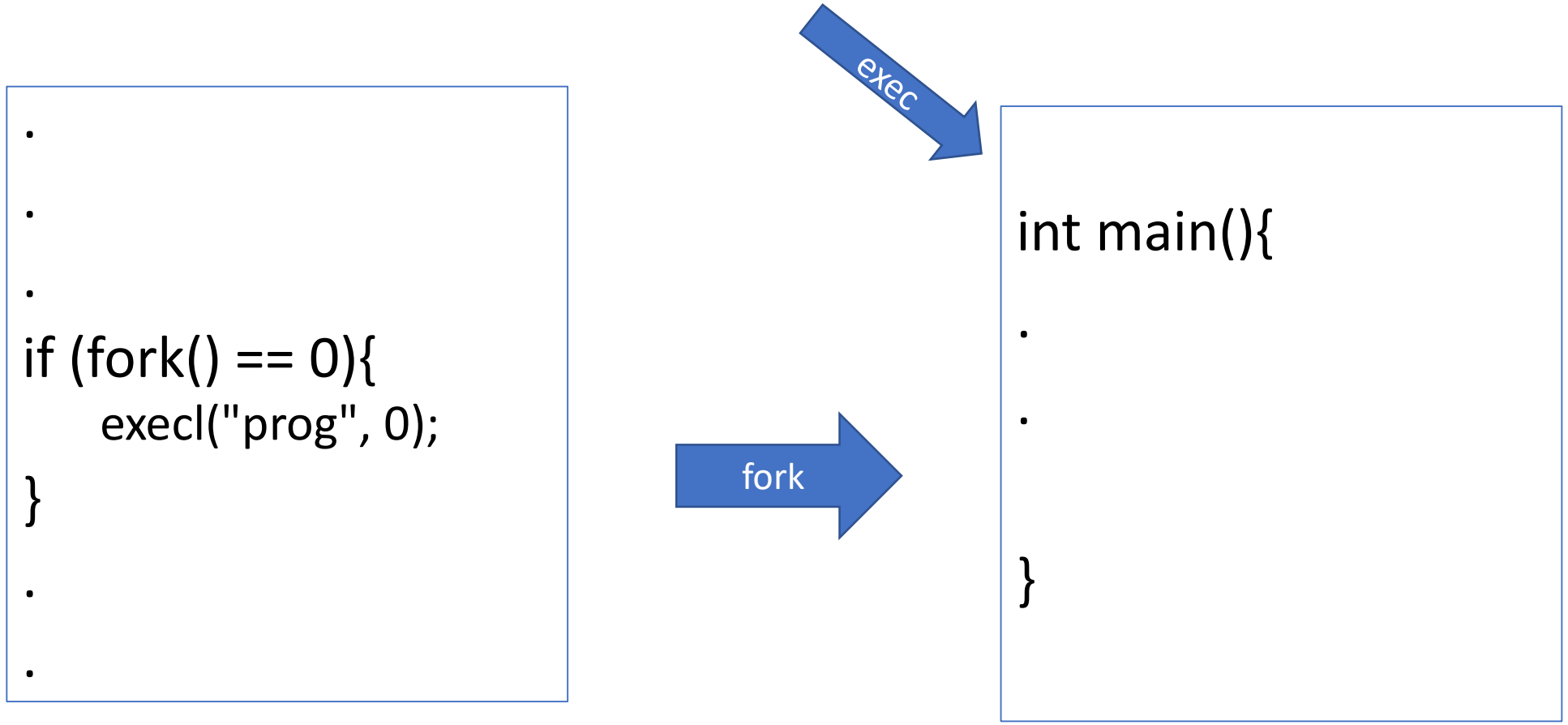
Feasible output:

```
L0
L1
Bye
Bye
L1
Bye
Bye
```

Infeasible output:

```
L0
Bye
L1
Bye
L1
Bye
Bye
```

Putting Programs into Processes

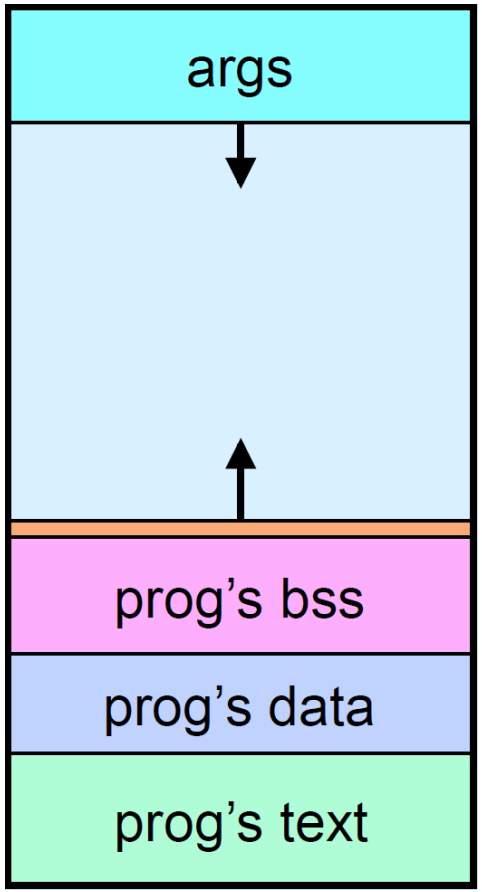


Execv

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

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

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**

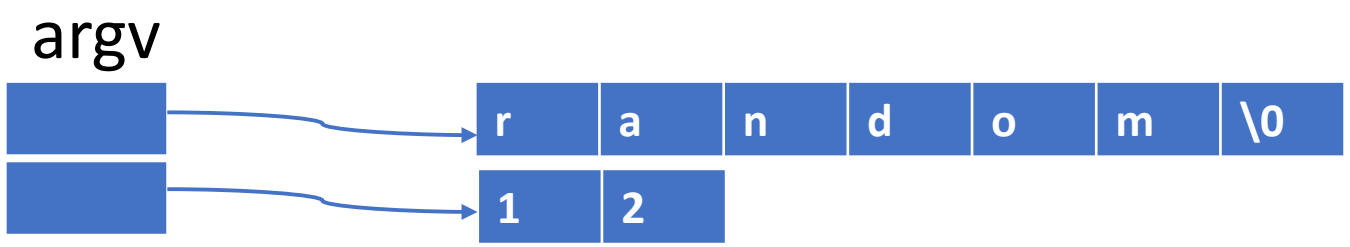
```
if (fork() == 0) {
    char *argv[] = {”./random", "12", (void *)0};
    execv("./random", argv);
}
```

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

Question: Will the *printf* statement be executed, after random completes?

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;
}
```



Shell

- **How does the shell invoke your program?**

```
if (fork() == 0) {
    char *argv = {”./random", "12", (void *)0};
    execv("./random", argv);
}
/* what does the shell 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);
```

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);
}
```

Zombie Example

```c
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./fork7 &
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- `ps` shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by `init`

Reaping Child Processes

- Idea
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a “zombie”
    - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Non-terminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

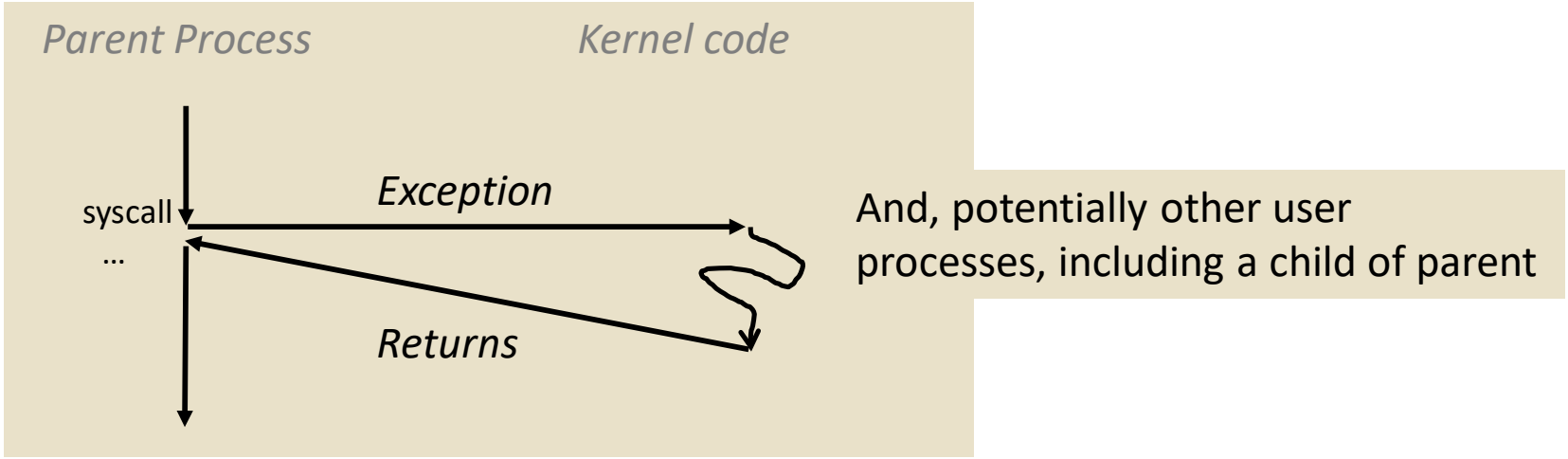
```
linux> ./forks8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill child explicitly, or else will keep running indefinitely

`wait`: Synchronizing with Children

- Parent reaps a child by calling the `wait` function

- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates



`wait`: Synchronizing with Children

- Parent reaps a child by calling the `wait` function

- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`

`wait`: Synchronizing with Children

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

forks.c

Feasible:

HC	HP
HP	HC
CT	CT
Bye	Bye

Infeasible:

HP
CT
Bye
HC

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);
/* WEXITSTATUS(status) extracts it */
```

Summary

- Exceptions
  - Events that require nonstandard control flow
  - Generated externally (interrupts) or internally (traps and faults)

- Processes
  - At any given time, system has multiple active processes
  - Only one can execute at a time on any single core
  - Each process appears to have total control of processor + private memory space

Summary (cont.)

- Spawning processes
  - Call `fork`
  - One call, two returns
- Process completion
  - Call `exit`
  - One call, no return
- Reaping and waiting for processes
  - Call `wait` or `waitpid`
- Loading and running programs
  - Call `execve` (or variant)
  - One call, (normally) no return