UNM SCHOOL *of* ENGINEERING

Department of Computer Science

Lecture #25 (ECF 3) – part 1 More operations to manipulate processes + process commands and shells

Prof. Soraya Abad-Mota, PhD

Preliminaries

- Changing the Final exam time to 5pm (Quiz)
- Two more examples of programs using fork (did them? Questions?)
  - fork4
  - fork5
- what is done by the fork function? (stop for a moment and write it down)

example fork4

```c
void fork4()
 {
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

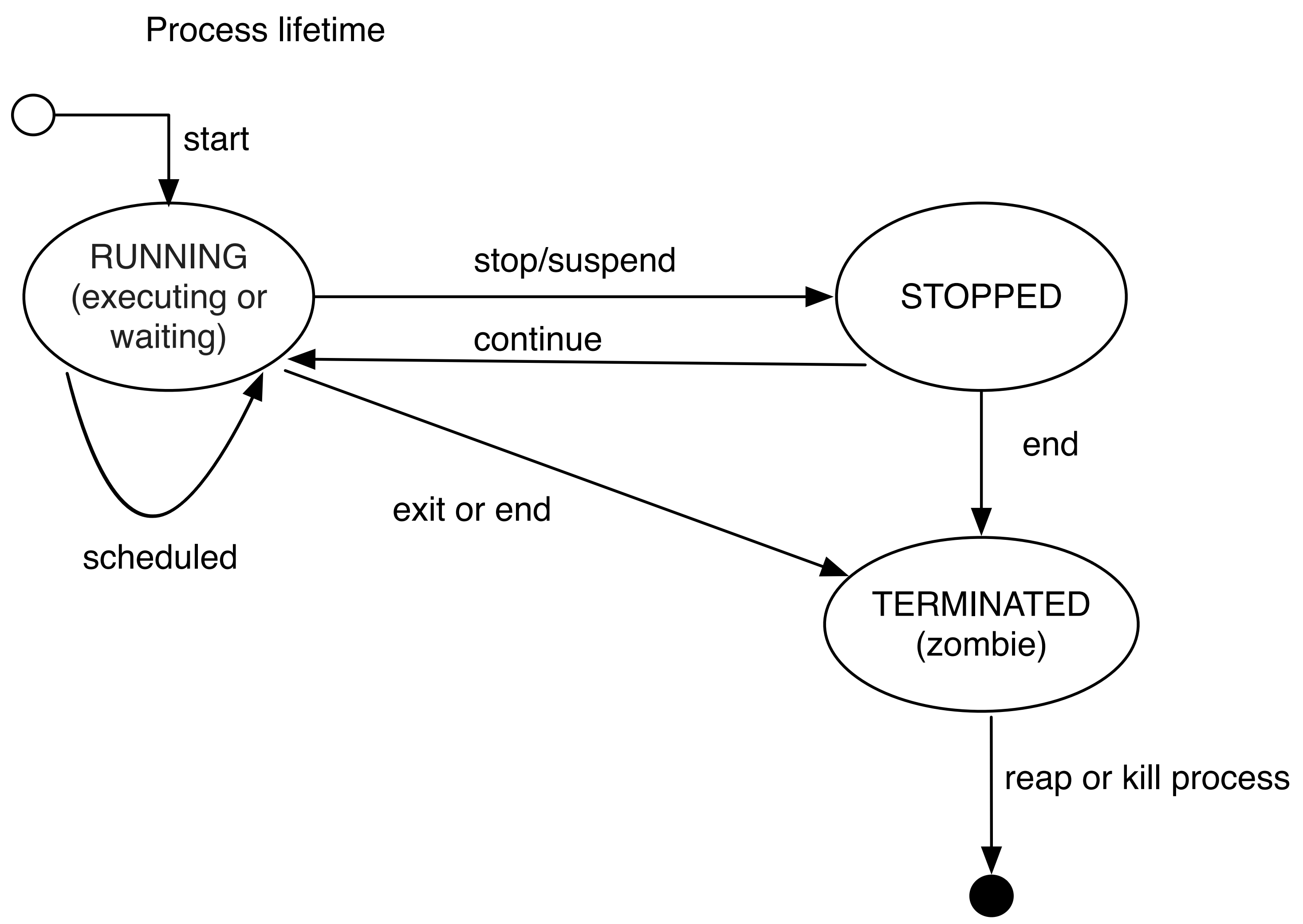
fork Example: Nested forks in children

Practice

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c

Lifetime of a process (recall)



What happens in Terminated state?

- Terminated
  - process still consumes memory resources
  - it cannot be scheduled again but has memory allocated
- To free all its resources, the process must be reaped

Reaping Child Processes (sec. 8.4.3)

- Idea
  - Process terminates, it still consumes system (memory) 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 discards zombie child process, which ceases to exist
- 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

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 */
    }
}
```

forks.c

```
linux> ./forks 7 &
[1] 6639
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`

Non-terminating Child Example

```c
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);
    }
}
```

forks.c

```
linux> ./forks 8
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

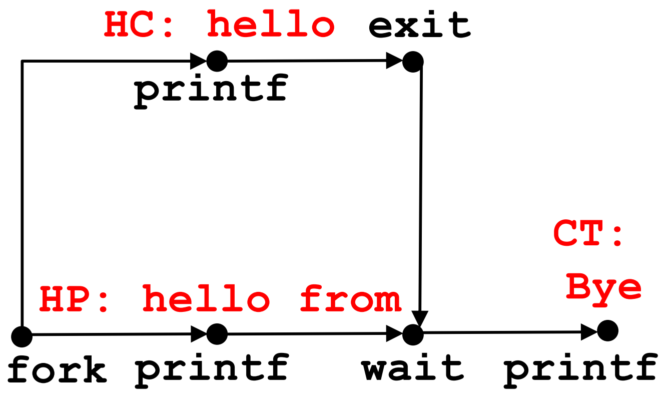
`wait`: Synchronizing with Children

- Parent reaps a child by calling the `wait` function
- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - 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`
      - Used to interpret the status argument, see textbook for details (p.745)

`wait`: Synchronizing with Children

```
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 output: (why?)

HC: hello from child
HP: hello from parent
CT: child has terminated
Bye

Infeasible output:

HP: hello from parent
CT: child has terminated
Bye
HC: hello from child

Another wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status (detailed explanation in textbook pp. 746-749)

```
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) { /* creating N children*/
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

forks.c

`waitpid`: Waiting for a Specific Process

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - Suspends current process until specific process terminates
  - Various options (see textbook)

```c
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

forks.c

Topics

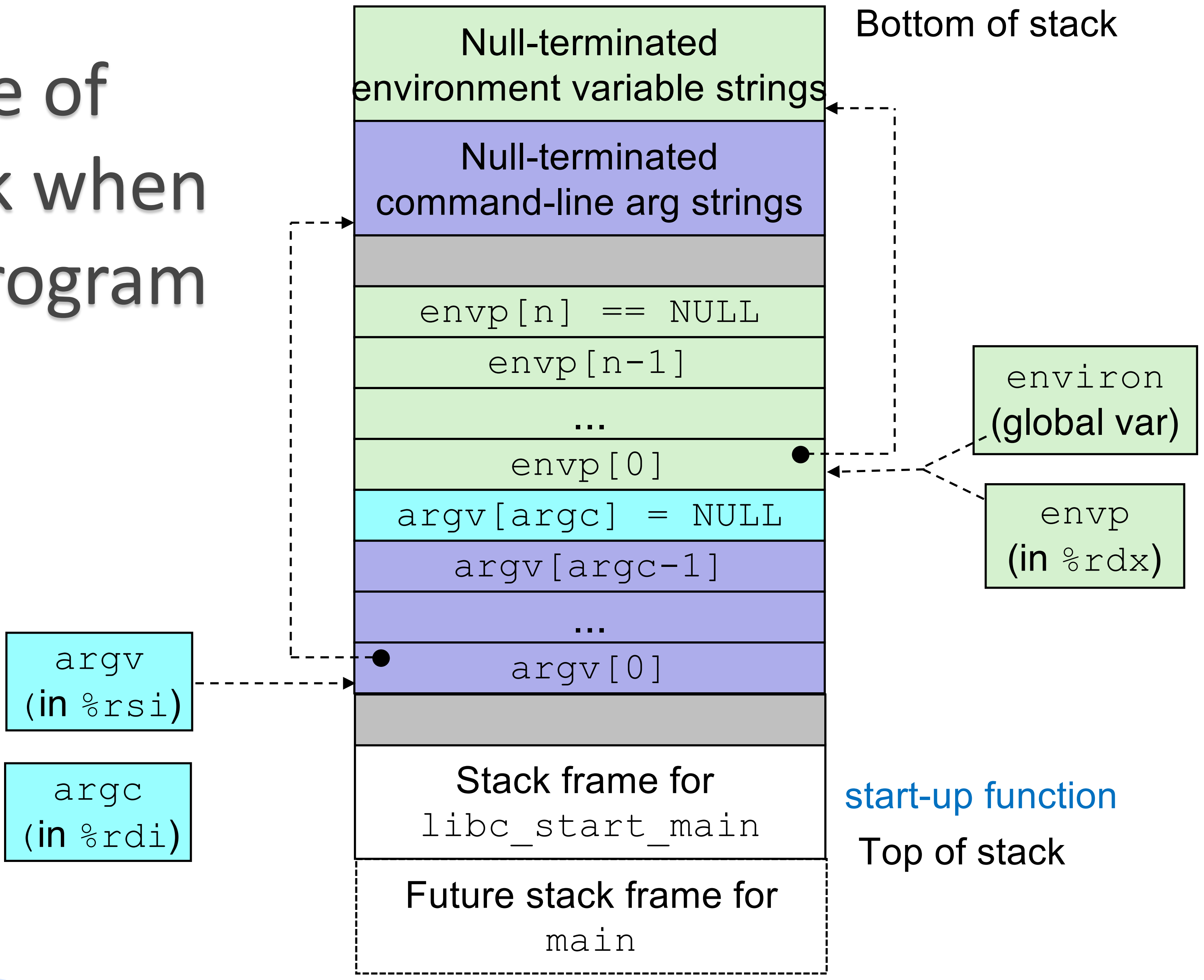
- Process Control:
  - system calls for manipulating processes:
    - two more examples on modeling for with process graphs
    - destruction and reaping
    - loading and running programs (execve in sec. 8.4.5)
    - process commands and shells

students read 8.4.4 on their own
(Putting Processes to sleep: sleep and pause functions)

`execve`: Loading and Running Programs

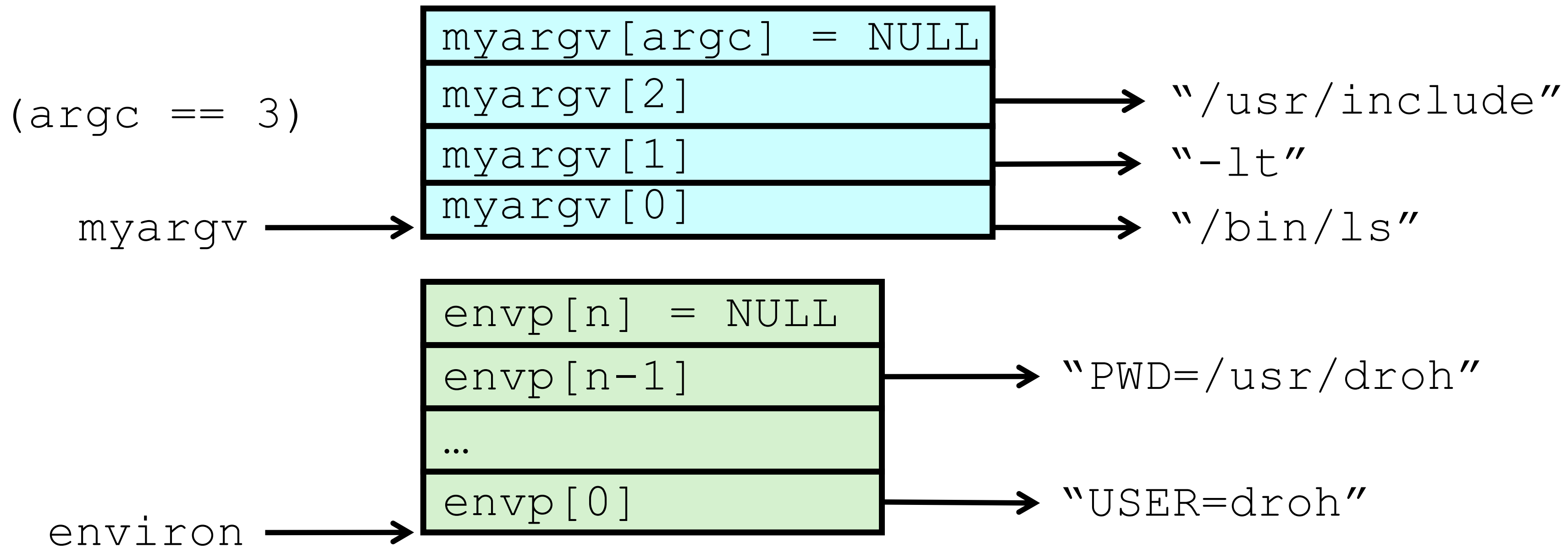
- `int execve(char *filename, char *argv[], char *envp[])`
- Loads and runs in the context of current process:
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`** (of the file to be executed)
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
- Called once and never returns
  - …except if there is an error

Structure of the stack when a new program starts



`execve` Example

- **Executes** `"/bin/ls -lt /usr/include"` **in child process using current environment:**



```
if ((pid = Fork()) == 0) {   /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Summary

- Exceptions (in general)
  - 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 a single core, though
  - Each process appears to have total control of processor + private memory space

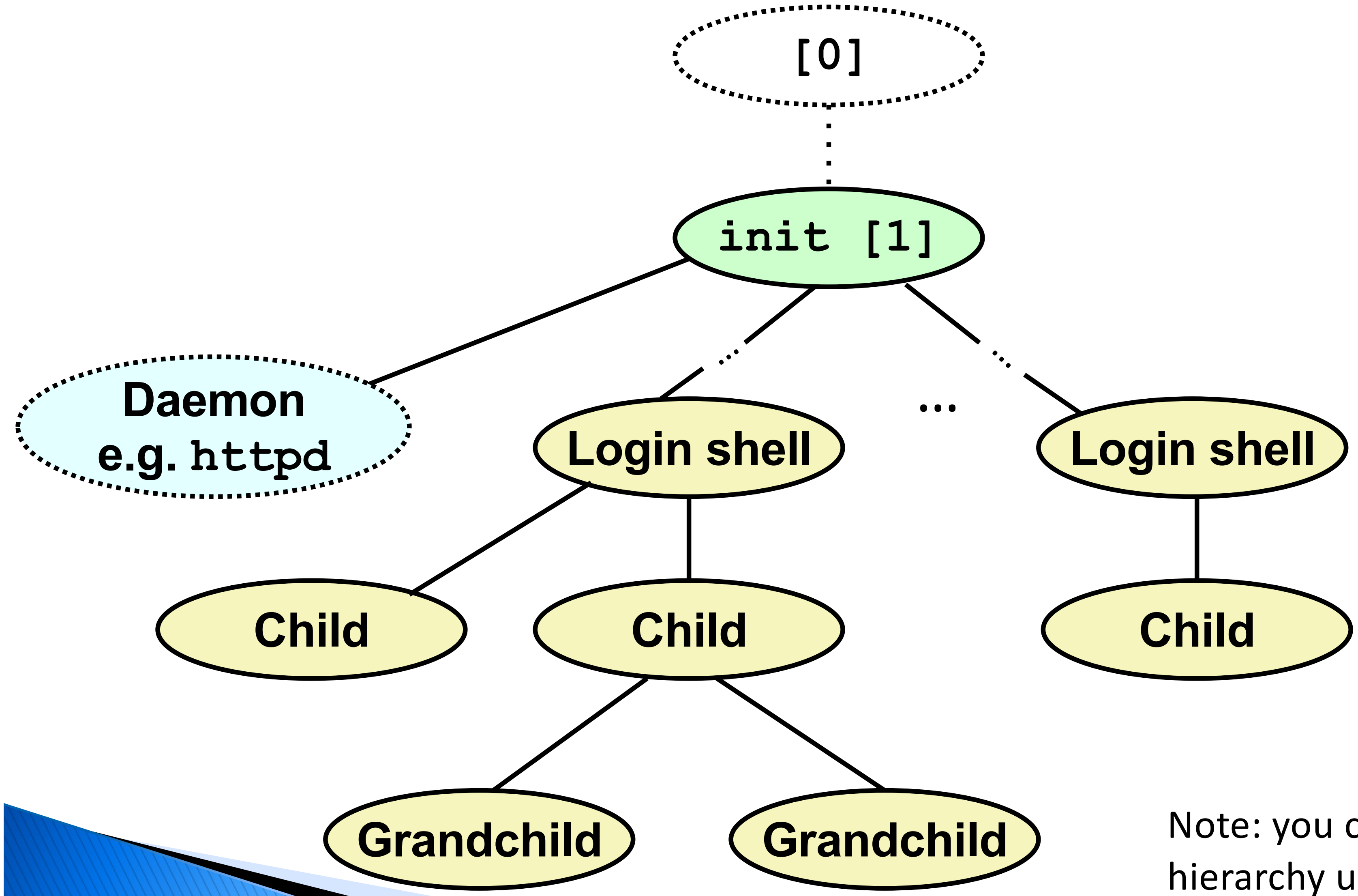
Summary (cont.)

- 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

Topics

- Process Control:
  - system calls for manipulating processes:
    - more examples on process creation and process graphs
    - destruction and reaping (sec 8.4.3)
    - loading and running programs
    - process commands and shells (sec. 8.4.6)

Linux Process Hierarchy



Note: you can view the hierarchy using the Linux `pstree` command

Process Command and Control

- ***Command shell:*** program responsible for creating and controlling processes for a user
- UNIX Command Shells
  - /bin/bash
  - /bin/tcsh
  - Graphical file browsers
  - Etc.
- Turns user input into calls to fork() and execve() to control program execution
- Web servers also use fork() and execve()

Shell Programs

- A *shell* is an application program that runs programs on behalf of the user.

sh	Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
csh/tcsh	BSD Unix C shell
bash	"Bourne-Again" Shell (default Linux shell)

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

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

        /* evaluate */
        eval(cmdline);
    }
}
```
shellex.c

Execution is a sequence of read/evaluate steps

tcsh, bash, zsh in MacOS

Simple Shell `eval` Function

```
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);
            }
        }

        /* 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;
}
```

shellex.c

What is a builtin in Linux?

- “In computing, a shell builtin is a command or a function, called from a shell, that is executed directly in the shell itself, instead of an external executable program which the shell would load and execute. Shell builtins work significantly faster than external programs, because there is no program loading overhead.”

From: https://linuxhandbook.com/shell-builtin-commands/

Problem with Simple Shell Example

- Our example shell correctly waits for and reaps foreground jobs

- But what about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory

ECF to the Rescue!

- Solution: Exceptional control flow
  - The kernel will interrupt regular processing to alert us when a background process completes
  - In Unix, the alert mechanism is called a ***signal***

NOTE: continue with slides for signals