Announcements

- Graded Lab 3
    - December 3-4
    - Covers Assignments 4-7
    - You will be asked to write:
        - One IA-32 assembly program and one C driver program (like Assignment 6)
        - One C program using dynamic memory (like Assignment 7)
- Assignment 9
    - Writing your own shell
    - Due 11:59 p.m., Monday, December 8

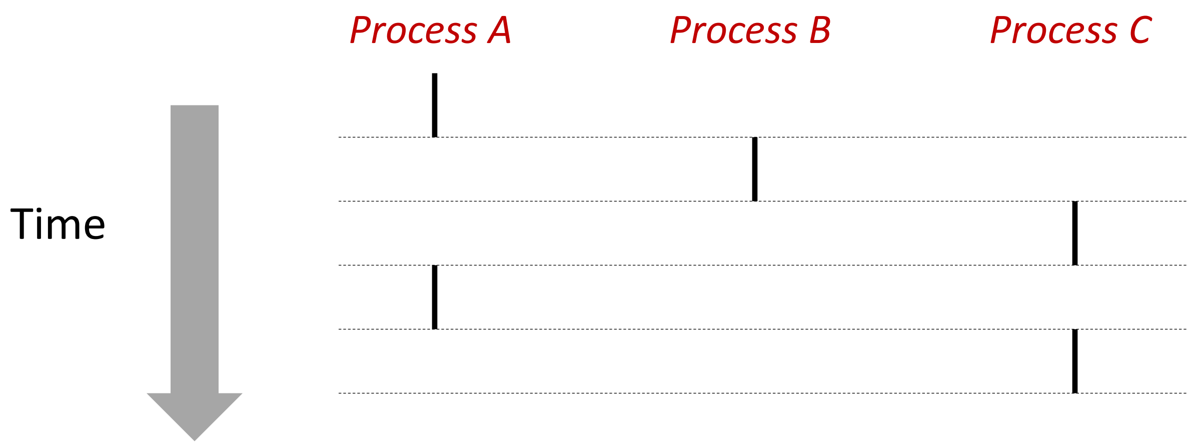
Lecture 33

More on Processes

CPSC 275
Introduction to Computer Systems

Processes

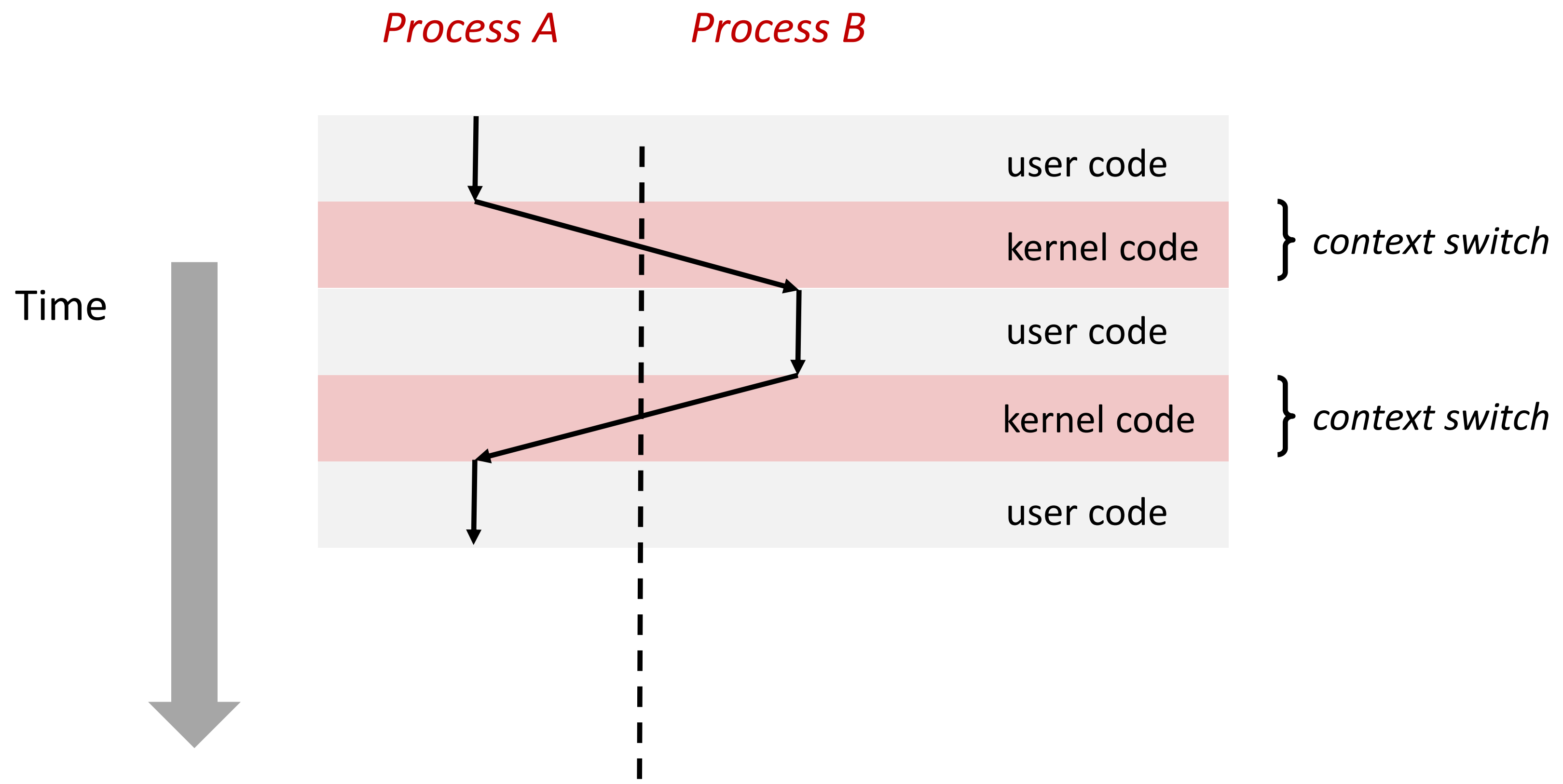
- A *process* is an instance of a running program.
- Two processes *run concurrently* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):



  – Concurrent: A & B, A & C
  – Sequential: B & C

Context Switching

- Control flow passes from one process to another via a *context switch*



Creating new processes

`int fork(void)`

- creates a new process (*child* process) that is identical to the calling process (*parent* process)

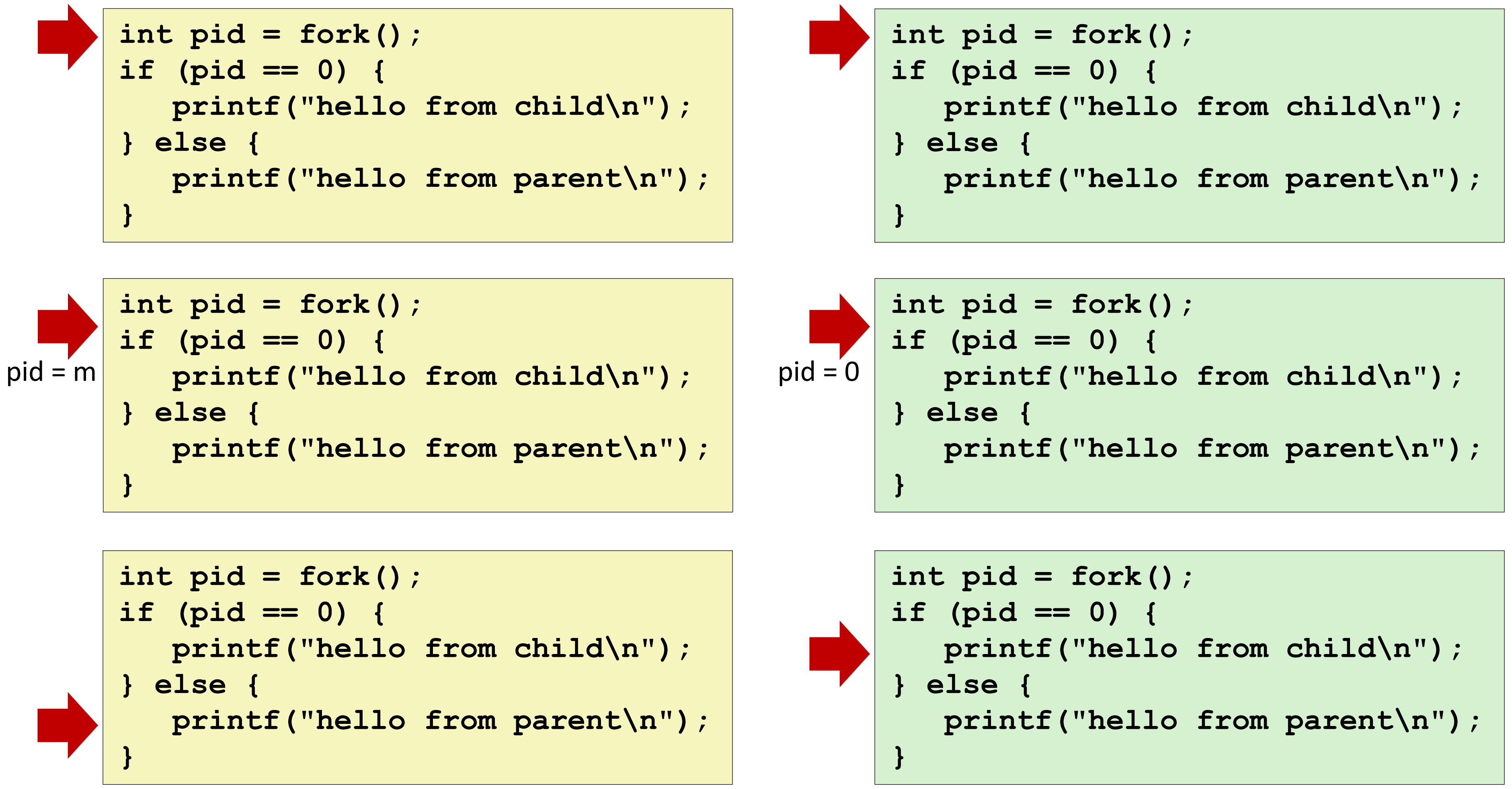
```
int pid = fork();
if (pid == 0)
   printf("hello from child\n");
else
   printf("hello from parent\n");
```

- called *once* but returns *twice*
  - returns 0 to the child process
  - returns child's **`pid`** to the parent process

Understanding `fork()`

Process n

Child Process m



```
int pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
int pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
int pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
int pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
int pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
int pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent` *Which one is first?* **`hello from child`**

`fork` Example #1

- Parent and child both run same code
  - Distinguish parent from child by return value from **`fork`**
- Start with same state, but each has private copy
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```
void fork1() {
   int x = 1;
   int pid = fork();
   if (pid == 0)
      printf("Child has x = %d\n", ++x);
   else
      printf("Parent has x = %d\n", --x);
   printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Ending a process

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

- exits a process
- normally return with status 0

Synchronizing with child processes

- The parent process must wait for all child processes, or it will create *zombie* processes.
- *Reaping*
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process

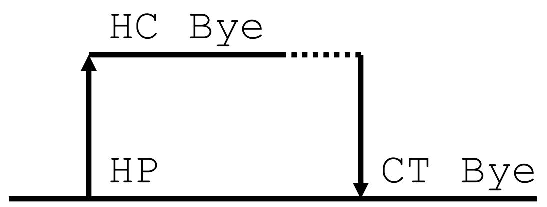
```
int wait(int *child_status)
```

- 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 object it points to will be set to a status indicating why the child process terminated.

Example: Synchronizing with child processes

```
void fork9() {
    int child_status;

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



Checking exit status of children

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    int pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* child */
    for (i = 0; i < N; i++) {
        int 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);
    }
}
```

Waiting for a specific process

`waitpid(pid, &status, options)`

- suspends current process until specific process terminates
- various options (see manpage of waitpid())

```
void fork11()
{
    int 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--) {
        int 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 terminated abnormally\n", wpid);
    }
}
```

Other useful system functions on processes

`sleep(n)`

– suspends current process for `n` seconds.

`exec()`

– Family of functions to load and run a new program in the context of the current process.

Assignment 9

- Writing your own shell (`tsh.c`)
  - Interactive
  - Process concept
  - Using `fork`(), `exec`(), and `wait`()
  - No need for support
    - input/output redirection
    - background jobs
    - pipes
    - …

Assignment 9

```
$ ./tsh
tsh> echo hello
hello
tsh> ls
file1 file2
tsh> ls -l
-rwxrwxrwx 0 pyoon pyoon   5600 Sep  7 15:18 file1
-rwxrwxrwx 0 pyoon pyoon   5825 Sep  7 15:19 file2
tsh> quit
$
```

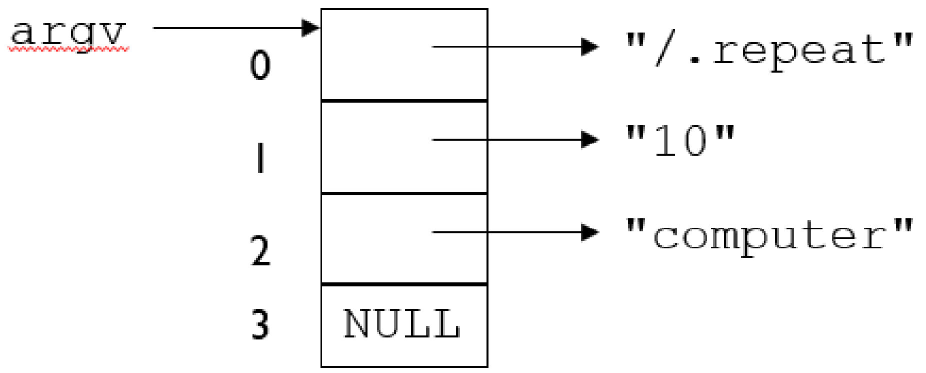
Assignment 9: Command-Line Arguments

- If the user enters the command line

```
$ ./repeat 10 computer
```

then `argc` will be 3 (why?),

and `argv` will contain:



Assignment 9

- Creating an array of strings (statically)

```
char *strarr[] = {“Trinity”, “College”};
printf(“%s”, strarr[1]);
```

- Creating an array of strings (dynamically)

```
char *strarr[2]; // static allocation
strarr[0] = (char *)malloc(10); // more than we need
strarr[1] = (char *)malloc(10);
strcpy(strarr[0], “Trinity”);
strcpy(strarr[1], “College”);
printf(“%s”, strarr[1]);
```

But what if we don’t know how many strings and their lengths?

Assignment 9

- We must create an array of `char` pointers dynamically AND allocate enough space for each string dynamically.
- Creating an array of two `char` pointers dynamically

```
char **strarr;
strarr = (char **)malloc(2*sizeof(char *));
strarr[0] = “Trinity”;
strarr[1] = “College”;
```

Assignment 9

- Allocate enough space for each string (dynamically)

```
    strarr[0] = (char *) malloc(10);
    strarr[1] = (char *) malloc(10);
    strcpy(strarr[0], “Trinity”);
    strcpy(strarr[1], “College”);
```

or

```
    char **p = strarr;
    *p = (char *) malloc(10);
    strcpy(p*, “Trinity”);
    *++p = (char *) malloc(10);
    strcpy(p*, “College”);
```