시스템 프로그래밍

강의 8 : 8.2 프로세스

http://eslab.cnu.ac.kr

* Some slides are from Original slides of RBE

프로세스Processes

정의 : 프로세스는 실행하고 있는 프로그램의 한 실행 예이다

- 컴퓨터과학 분야에서 가장 심오한 개념중의 하나
- 프로세스와 프로세서를 혼돈하지 마라

프로세스는 프로그램에 두 개의 중요한 추상화를 제공한다:

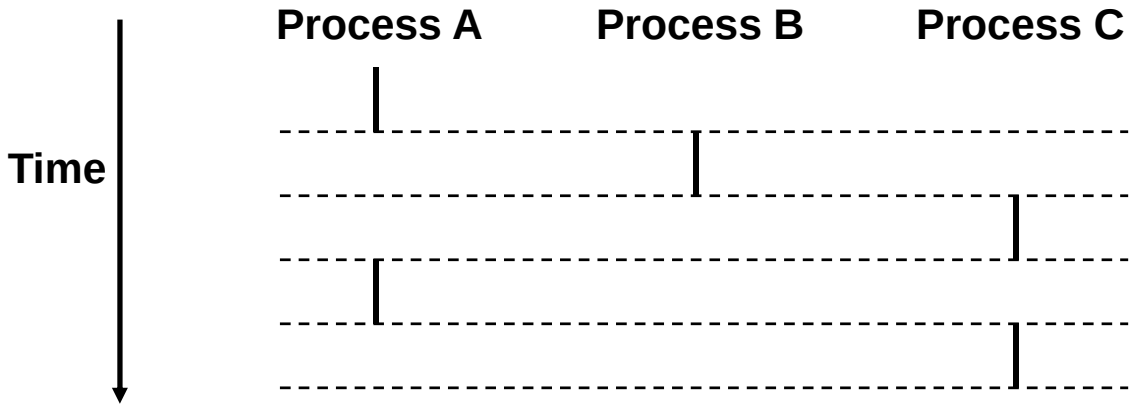
- 논리적인 제어흐름
  - 각 프로그램이 CPU를 독점하는 것처럼 보이도록 한다.
- 사적인 주소공간
  - 각 프로그램이 주 메모리를 독점하는 것처럼 보이도록 한다.

어떻게 이러한 착시가 가능한가?

- 프로세스의 실행이 서로 교대로 실행된다(interleaved , multitasking)
- 주소공간은 가상메모리 시스템에 의해 관리된다

논리적 제어흐름

각 프로세스는 자신만의 논리적인 제어흐름을 갖는다



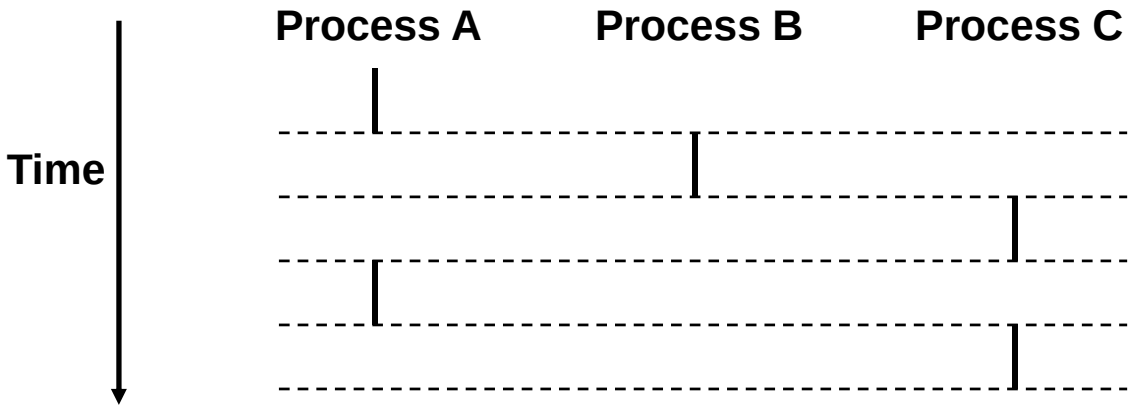
동시성 프로세스

두 프로세스는 그들의 실행 시간이 서로 중첩되면, 동시에 실행된다고 부른다.(*are concurrent)*

그렇지 않다면, 순차적으로 실행된다고 정의한다(*sequential.*)

Examples:

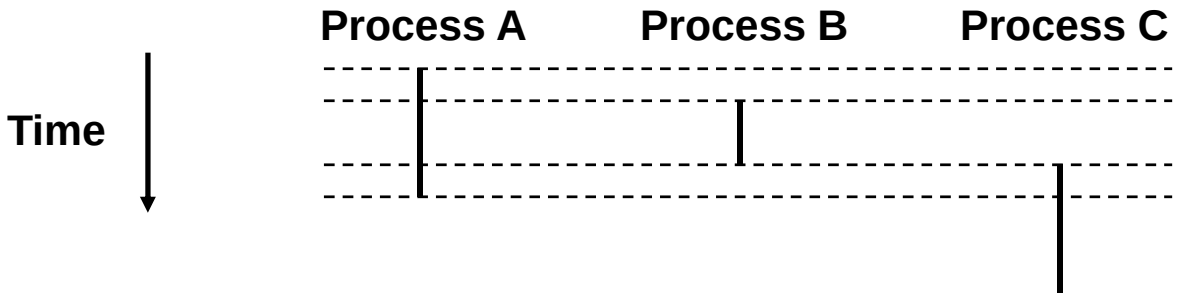
- 동시실행: A & B, A & C
- 순차실행: B & C



동시프로세스의 사용자 관점

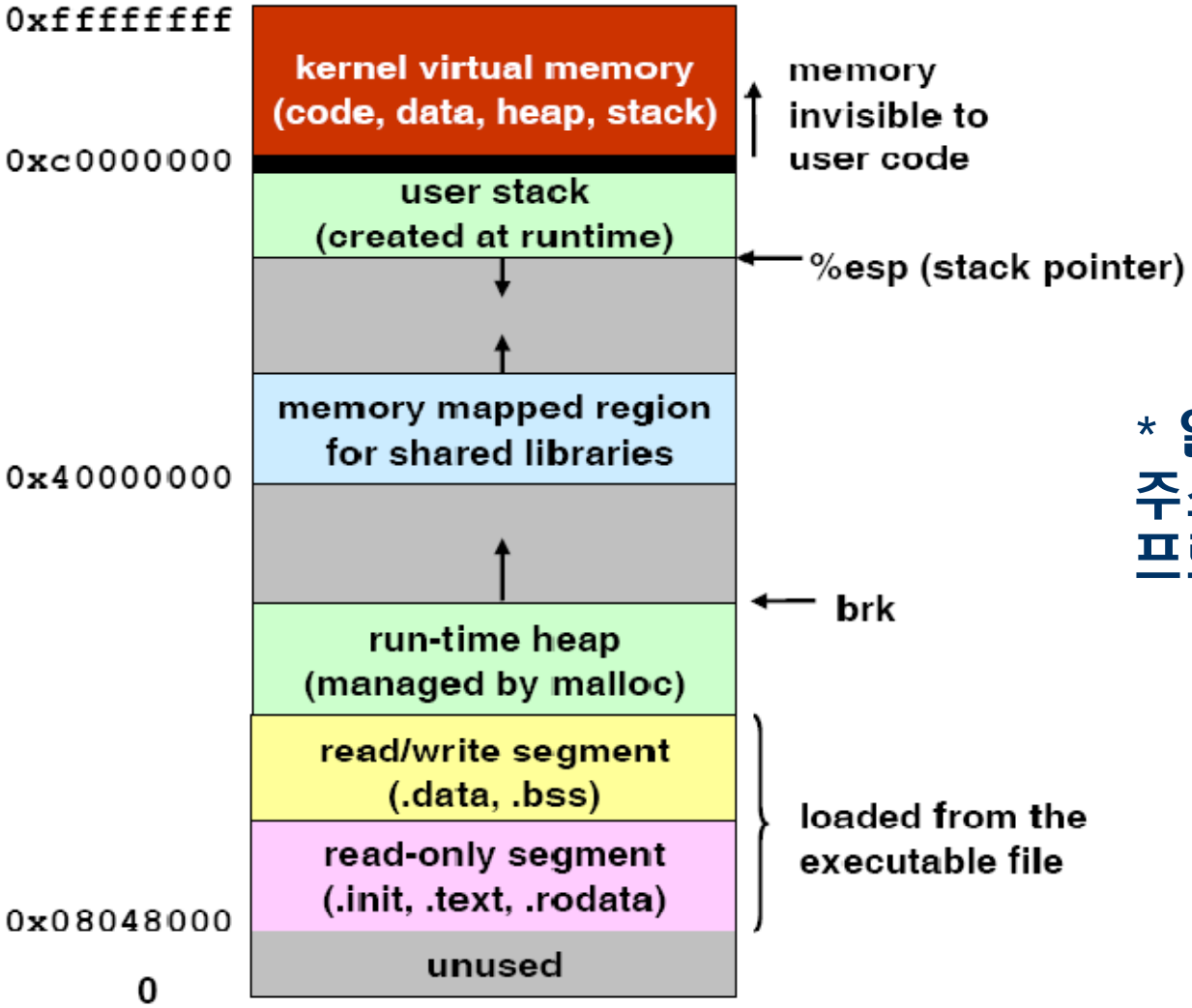
동시 프로세스들을 위한 제어흐름은 시간상으로는 물리적으로 분리된다.

그러나, 동시프로세스들이 서로 병렬로 실행된다고 생각할 수 있다.



* 멀티 태스킹 또는 타임 슬라이싱이라고 부름

리눅스 프로세스에서의 사적(Private)주소공간



* 시스템의 주소공간을 독자적으로 이용하는 착시현상

* 일반적으로 주소상의 특정 주소를 갖는 메모리는 다른 프로세스가 접근 할 수 없다.

유저모드와 커널모드(1)

프로세스의 모드를 구분

- 응용프로그램이 실행할 수 있는 명령어를 제한하고, 접근할 수 있는 주소공간을 제한하는 매커니즘

구현

- 최신 프로세서들은 모드비트를 컨트롤 레지스터에 제공한다

커널모드

- 모드비트가 세트 되면, 프로세스는 커널모드(또는 슈퍼바이저 모드)로 실행된다
- 커널모드로 실행되는 프로세스는 모든 명령어를 실행 할 수 있으며, 모든 메모리 영역을 접근 할 수 있다

유저모드

- 모드비트가 설정되지 않고, 프로세스가 특수 명령어를 실행할 수 없다.(프로세서 halt 명령, 모드비트 변경 명령)
- 커널 영역의 코드와 데이터를 접근 할 수 없다

유저모드와 커널모드(2)

모드의 전환

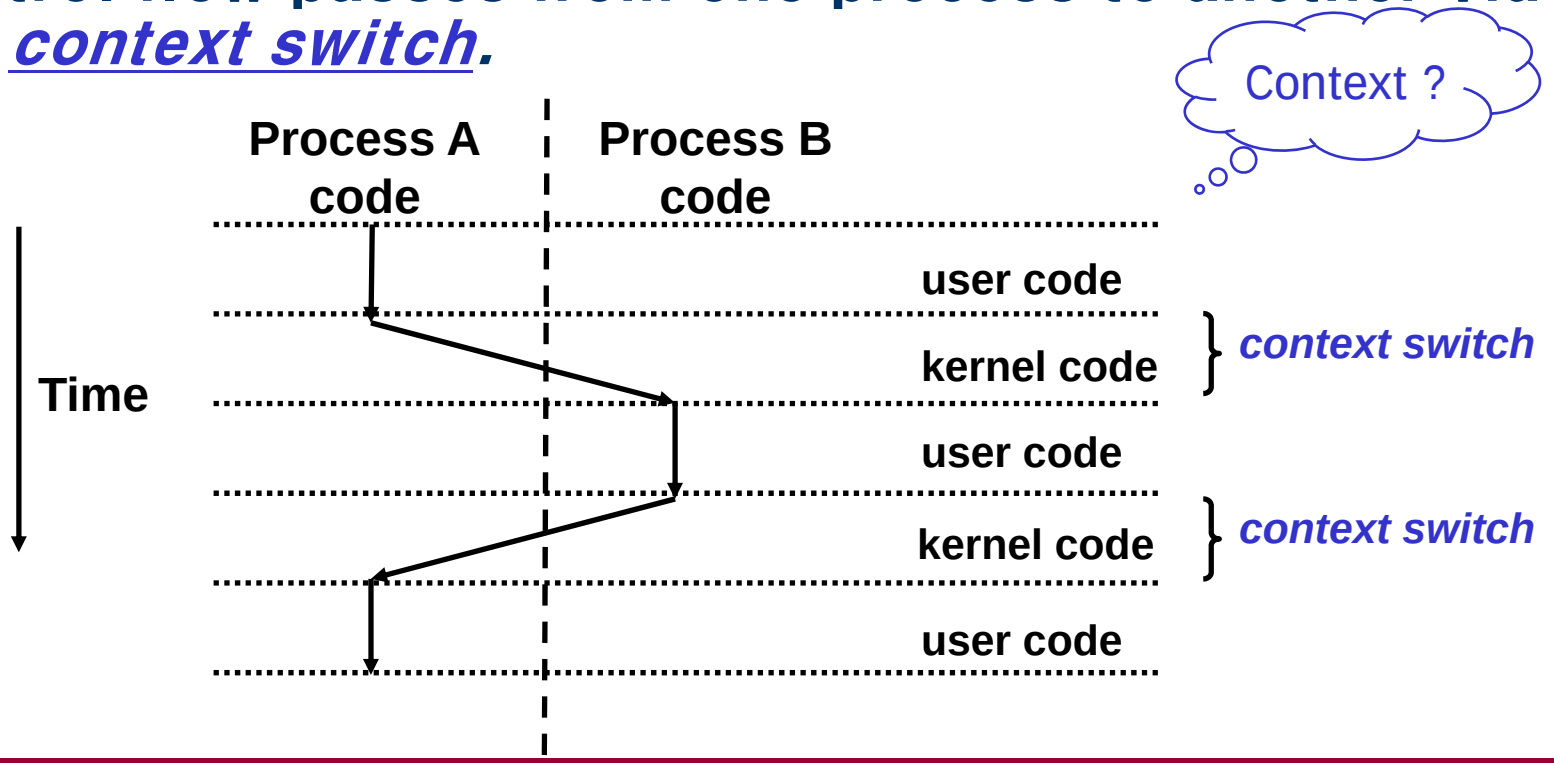
- A process running application code is in user mode
- Changing to kernel mode via an exception such as …. ?
- When the exception occurs, and control passes to the exception handler, the processor changes the mode from user mode to kernel mode.
- The handler runs in kernel mode.
- When it returns to the application code, the processor changes the mode from kernel mode back to user mode

커널 모드의 데이터 접근하기

- `/proc` filesystem exports the contents of many kernel data structures as a hierarchy of ASCII files
- `/proc/cpuinfo, /proc/<process id>/maps`

문맥전환(Context Switch)

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via a *context switch*.



시스템 콜(System Calls)

Unix systems provide a set of system calls

- that application programmers use when they want to request services from the kernel
- eg. reading a file or creating a new process
- In Linux, about 160 system calls
- `man syscalls`

Invoking System calls

- In C, we can directly invoke systems calls using `_syscall` macro
- How the system calls implemented ?
- Calling system calls directly is not desirable => Why ?
- Standard C library provides a set of wrapper functions

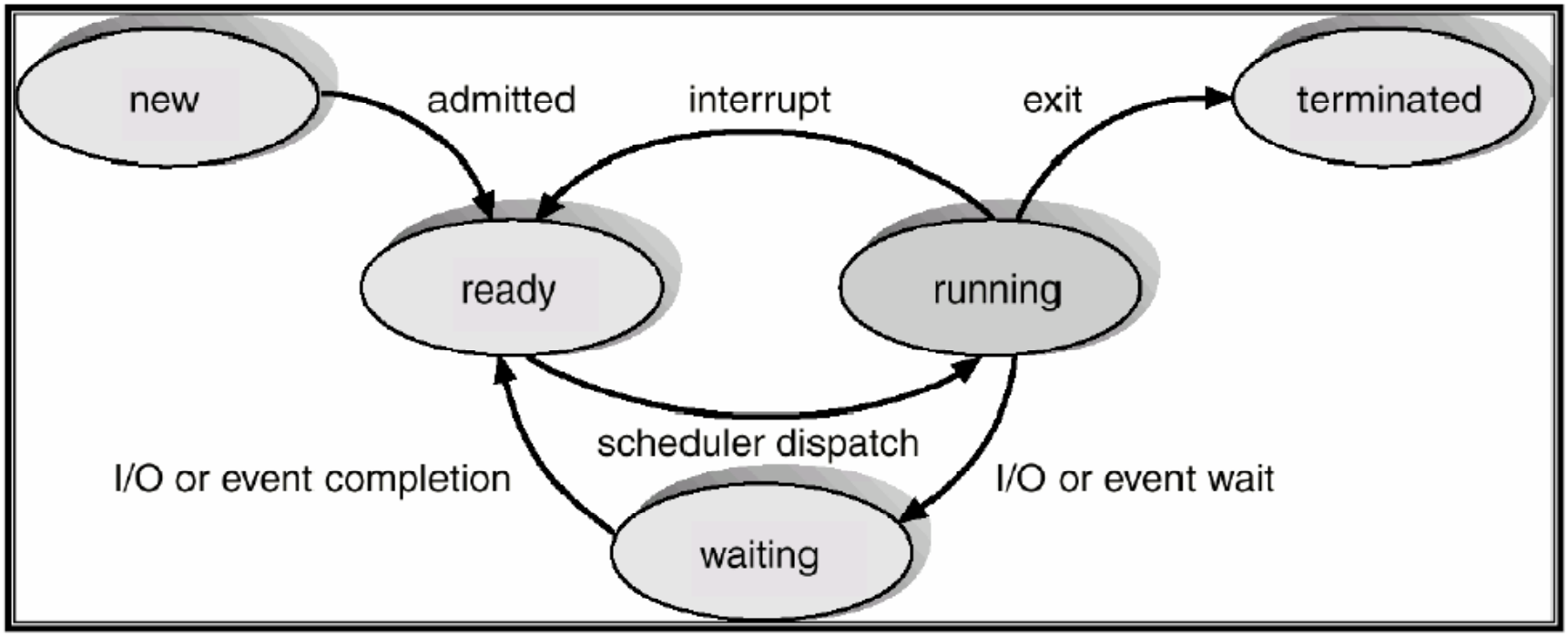
오류처리(Error Handling)

- When Unix system-level functions encounter an error, they return -1 and set the global integer variable `errno` to indicate the error
- Programmers should always check the errors, but many skip
- Because, checking errors will increase code size and make it harder to read
- `strerror` function returns a text string related to the `errno`

```
if ((pid = fork())<0) {
   fprintf(stderr, “fork error: %s\n”, strerror(errno));
   exit(0);
}
```

8.4 Process Control

Process States



- **Stopped or waiting**
  - A process stops as a result of receiving SIGSTOP, SIGSTP, SIGTTIN or SIGTTOU and it remains stopped until it receives a SIGCONT signal(signal is a SW interrupt)
- **Terminated when**
  - receives a signal whose default action is terminate
  - Returning from the main routine
  - Calling the exit function

Process Control

Unix provides a number of system calls for manipulating processes from C programs

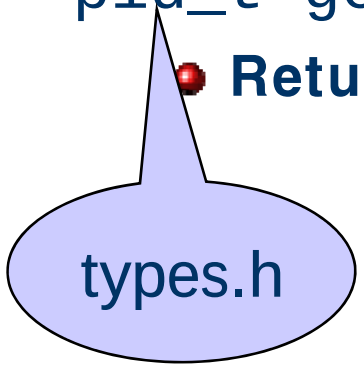
- Obtaining process ID
- Creating and Terminating Processes
- Reaping child processes
- Loading and running programs

Obtaining Process ID's

Each process has a unique positive process ID

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

- **Returns PID of either the caller or the parent**



fork: 프로세스 만들기

`int fork(void)`

- creates a new process (*child process*) that is identical to the calling process (*parent process*)
- returns 0 to the child process
- returns child's `pid` to the parent process

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

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Fork Example #1

Key Points

- 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

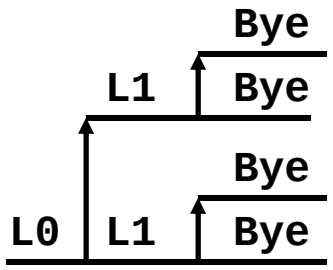
```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
       printf("Child has x = %d\n", ++x);
    } else {
       printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

Key Points

- Both parent and child can continue forking

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



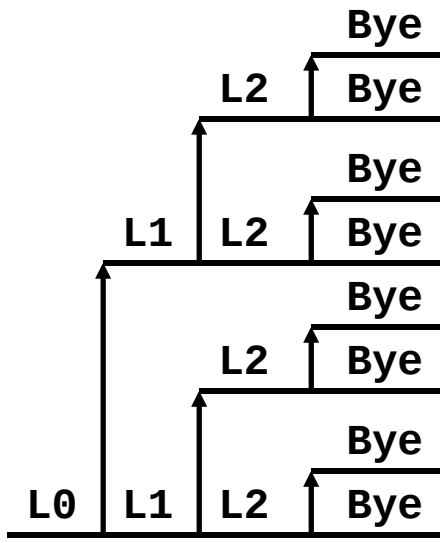
Process graph is useful

Fork Example #3

Key Points

- Both parent and child can continue forking

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



exit: 프로세스 종료하기

`void exit(int status)`

- **Terminates the process with an exit status of** `status`
  - **Normally return with status 0**
- `atexit()` **registers functions to be executed upon exit**

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

좀비(Zombies)

Ide

Rea

Wh

ed

Why ?

좀비(Zombies)

Idea

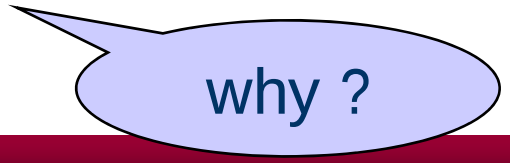
- When process terminates, still consumes system resources
  - Various tables maintained by OS
- Called a "zombie" : process terminated but not reaped
  - Living corpse, half alive and half dead

Reaping procedure

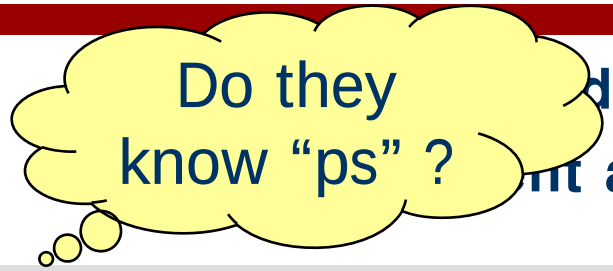
- Performed by parent on terminated child
- Parent is given exit status information
- Kernel discards the terminated process

What if Parent Doesn't Reap?

- If any parent terminates without reaping a child, then child will be reaped by `init` process
- long-running processes should reap their zombie children
  - E.g., shells and servers



Zombie Examp




```
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> ./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
```

Zombie Example 2

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

```
linux> ./forks 8
Terminating Parent, PID = 6
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcs
 6676 ttyp9    00:00:06 for
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcs
 6678 ttyp9    00:00:00 ps
```

```
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);
    }
}
```