CHUNGNAM NATIONAL UNIVERSITY

시스템 프로그래밍

강의 9. 프로세스 II

교재 8.4

http://eslab.cnu.ac.kr

좀비의 교훈

- 프로세스가 종료하는 경우 커널은 종료된 프로세스를 시스템에서 즉시 제거하지 않는다
- 종료된 프로세스는 자신의 부모가 제거할 때까지 종료된 상태로 남아있게 된다
  - 즉, 부모가 죽지 않으면서, 자식의 좀비를 제거 안하는 경우가 문제
  - 이런 경우는 언제일까?
- 부모가 정상적으로 자식을 청소하지 않고 종료되면, 커널은 init 프로세스(PID = 1) 로 하여금 대신 청소하도록 해 준다(간접적으로 자식을 제거)
  - 부모가 종료된다면, 종료된 자식프로세스는 init이 제거하므로 문제가 안된다
- 프로세스를 직접(explicit) 제거하는 함수
  - wait 함수를 이용해서 자식을 청소할 수 있다

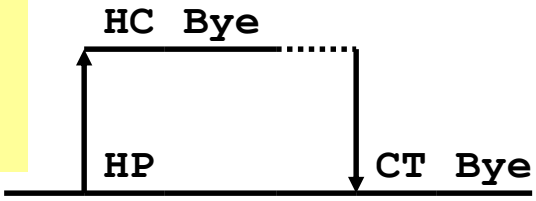
wait: 자식 프로세스와 동기하기

- `int wait(int *child_status)`
  - 현재 프로세스를 자신의 자식프로세스들 중의 하나가 종료될 때까지 정지시킨다
  - 리턴 값은 종료한 자식 프로세스의 **pid** 가 된다
  - 만일 **child_status != NULL**, 인 경우는, 자식 프로세스의 종료 이유를 나타내는 상태 정보를 갖는다.
  - 그리고 나서, 커널은 자식 프로세스를 제거한다
  - 이와 같이 부모는 자식 프로세스를 명시적으로 삭제한다.

wait: 자식과 동기하기(즉, 좀비 제거하기)

```
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();
}
```



Wait() Example

- 만일 여러개의 자식들이 종료되었다면, 무순으로 자식을 제거한다
- exit 상태 정보를 얻기 위해서 **WIFEXITED** 와**WEXITSTATUS** 등의 매크로를 사용한다

```
void fork10()
{
    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 = 0; i < N; i++) {
        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);
    }
}
```

정상적인 종료(exit, 또는 리턴)라면 참을 리턴

정상종료라면 exit 상태정보를 리턴

Waitpid(): 특정 pid를 청소한다

- **waitpid(pid, &status, options)**
  - Pid : 특정 프로세스 아이디를 기다린다. -1이면 wait( )과 동일.
  - Options : 0(종료된 자식을 기다린다), WNOHANG(==1 한번만 체크), WUNTRACED(==2, 정지되거나 종료된 자식을 기다린다)

```
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 = 0; i < N; 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 terminated abnormally\n", wpid);
    }
```

Wait/Waitpid 실행예

Using wait (fork10)

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

Using waitpid (fork11)

```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminated with exit status 103
Child 3572 terminated with exit status 104
```

연습문제 1. waitpid

- 아래 프로그램의 가능한 출력을 모두 쓰시오.

code/ecf/waitprob0.c

```
1   int main()
2   {
3       if (Fork() == 0) {
4           printf("a");
5       }
6       else {
7           printf("b");
8           waitpid(-1, NULL, 0);
9       }
10      printf("c");
11      exit(0);
12  }
```

code/ecf/waitprob0.c

주) waitpid시에 pid=-1 이면 부모의 모든 자식프로세스를 기다린다

연습문제 2. waitpid

- 다음 프로그램을 실행하면 몇 줄을 출력하게되는가?

code/ecf/waitprob1.c

```
1   int main()
2   {
3       int status;
4       pid_t pid;
5
6       printf("Hello\n");
7       pid = Fork();
8       printf("%d\n", !pid);
9       if (pid != 0) {
10          if (waitpid(-1, &status, 0) > 0) {
11              if (WIFEXITED(status) != 0)
12                  printf("%d\n", WEXITSTATUS(status));
13          }
14      }
15      printf("Bye\n");
16      exit(2);
17  }
```

code/ecf/waitprob1.c

프로세스를 sleep 시키기

- `unsigned int sleep(unsigned int secs)`
  - 자기 자신을 **secs** 초 동안 정지**(suspend)** 시킨다
  - 정상적으로 깨어날 때는 **0**을 리턴하고, 그 외의 경우에는 잔여 시간을 초로 리턴한다
- `int pause(void)`
  - 호출하는 프로세스를 시그널**(signal)**을 받을 때까지 잠재운다(정지)

연습문제 3. sleep

- sleep 함수를 이용하여 다음과 같은 snooze( ) 함수를 완성하시오.

```
unsigned int snooze(unsigned int secs);
```

- 이 함수는 현재까지 잠을 잔 시간을 초로 다음과 같이 출력하는 것 외에는 sleep과 동일하게 동작한다

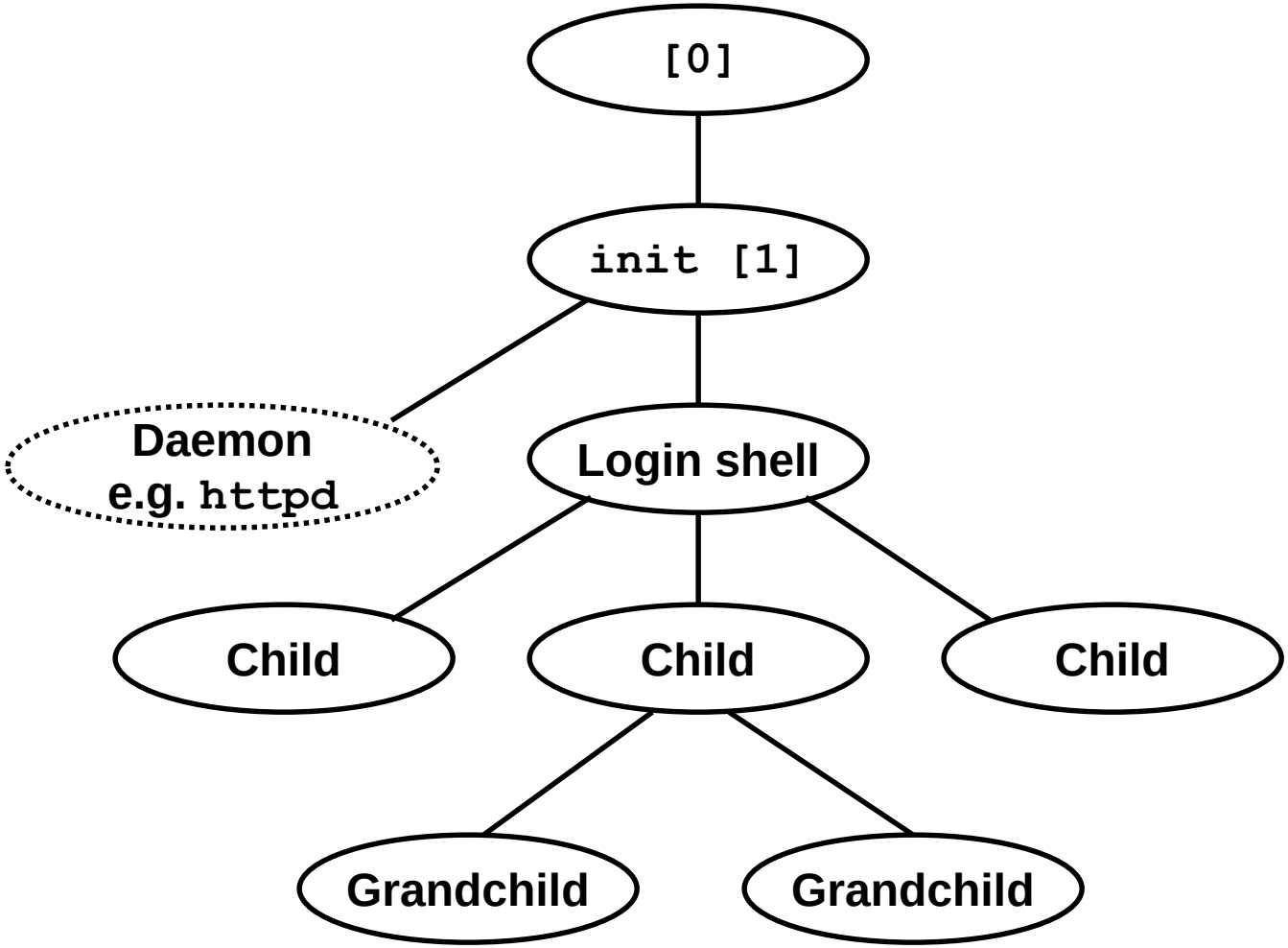
```
Slept for 4 of 5 secs.
```

새 프로그램의 로딩과 실행

- `int execve(char *filename, char *argv[], char *envp[])`
  - 실행파일 filename 을 현재 프로세스의 환경변수를 이용하면서, argv로 현재의 code, data, stack을 덮어 씌움
  - argv, envp: 널 문자로 끝나는 포인터 배열
  - 한번 호출되고, 리턴하지 않음
    - 에러가 있으면 리턴됨

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

fork와 exec의 사용예 : 리눅스 프로세스 체계



쉘 프로그램 예제

```
linux> ./shellex
> /bin/ls -l csapp.c
-rw-r--r-- 1 bryant users 23053 Jun 15  2015 csapp.c
> /bin/ps
  PID TTY          TIME CMD
31542 pts/2    00:00:01 tcsh
32017 pts/2    00:00:00 shellex
32019 pts/2    00:00:00 ps
> /bin/sleep 10 &
32031 /bin/sleep 10 &
> /bin/ps
 PID TTY          TIME CMD
31542 pts/2    00:00:01 tcsh
32024 pts/2    00:00:00 emacs
32030 pts/2    00:00:00 shellex
32031 pts/2    00:00:00 sleep
32033 pts/2    00:00:00 ps
> quit
```

Must give full pathnames for programs

Run program in background

Sleep is running in background

쉘 프로그램의 구현

- 쉘 *shell* 은 사용자의 명령을 처리해 주는 응용 프로그램이다
  - `sh` – Original Unix Bourne Shell
  - `csh` – BSD Unix C Shell
  - `tcsh` – Enhanced C Shell
  - `bash` –Bourne-Again Shell

```
int main()
{
    char cmdline[MAXLINE];

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

        /* evaluate */
        eval(cmdline);
    }
}
```

- 메인 루프는 사용자 명령입력 및 명령처리로 구성된다

쉘 처리 명령어 : Utility와 Built-in 명령어

- Shell의 built-in 명령어는 utility program과의 차이는 shell에 내장, search path에서 찾기 전에 실행
- Q. cd ? ls ?

백그라운드 처리 Background processing

- Job
  - **Shell 이 사용하는 추상화로 한 개의 명령어줄을 실행해서 생성된 프로세스들을 말함**
  - 어느 한 순간에는 최대 한 개의 foreground job과 0 또는 하나 이상의 background job으로 구성됨
  - 쉘은 각 job 마다 별도의 그룹을 할당
- 리눅스 Job 제어
  - foreground job 을 정지 시키려면 Ctrl+z(SIGSTOP 시그널을 보냄)
  - fg => 정지된 작업을 포그라운드 작업으로 실행, bg=> 백그라운드 작업으로 실행(SIGCONT 시그널을 보냄)
- 백그라운드 작업
  - 수행시키려면 명령줄 뒤에 & 추가
  - 자식셸로 생성되어 부모 셸과 같이 수행되나 키보드를 제어하지 않음
  - 예:

    $find . –name b.c –print &

    cf. $find . –name b.c –print
  - $date & pwd & # 두개의 백그라운드 프로세스 생성

백그라운드 작업 Background Job

- 유저는 대개 한 번에 한 개의 명령을 실행한다
  - 명령을 치고, 결과를 읽고, 다음 명령을 입력
- 어떤 프로그램은 실행시간이 길다
  - 예 : 이 파일을 두 시간 후에 지워라
  - % sleep 7200; rm/tmp/junk # shell stuck for 2 hours
- "백그라운드 작업 background" job 은 기다리는 것을 원치 않는 프로세스다
  - % (sleep 7200 ; rm/tmp/junk) & <= 이문자를 입력하면 백그라운드
  - [1] 907
  - % # ready for next command

간단한 쉘의 eval 함수

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    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);
            }
        }

        if (!bg) {   /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else         /* otherwise, don't wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
```

간단한 쉘 구현의 문제점

- 쉘은 포그라운드 작업들을 정확히 기다리고, 제거해준다
- 그러나, 백그라운드 작업들은 어떻게 처리하는가 ?
  - 이들은 종료되면 좀비가 된다
  - 이들은 쉘이 제거하지 않는데, 그 이유는 쉘은 대개 종료되지 않기 때문이다
  - 이 경우, 메모리 누수가 발생하며, 그 결과 커널 메모리 부족현상이 발생한다
  - 최신 **Unix:** 프로세스 메모리 할당량이 있어서 이를 초과하면 새로운 명령을 실행하지 못하게 된다. **fork()**는 **-1**을 리턴
- 해결책
  - 백그라운드 작업을 제거하기 위해서 ***signal*** 이라고 하는 방법을 사용한다