

UNIX 시스템 프로그래밍

» 6장. 프로세스 생성과 실행

Process의 생성

▶ fork 시스템 호출:

```
#include <sys/types.h>
```

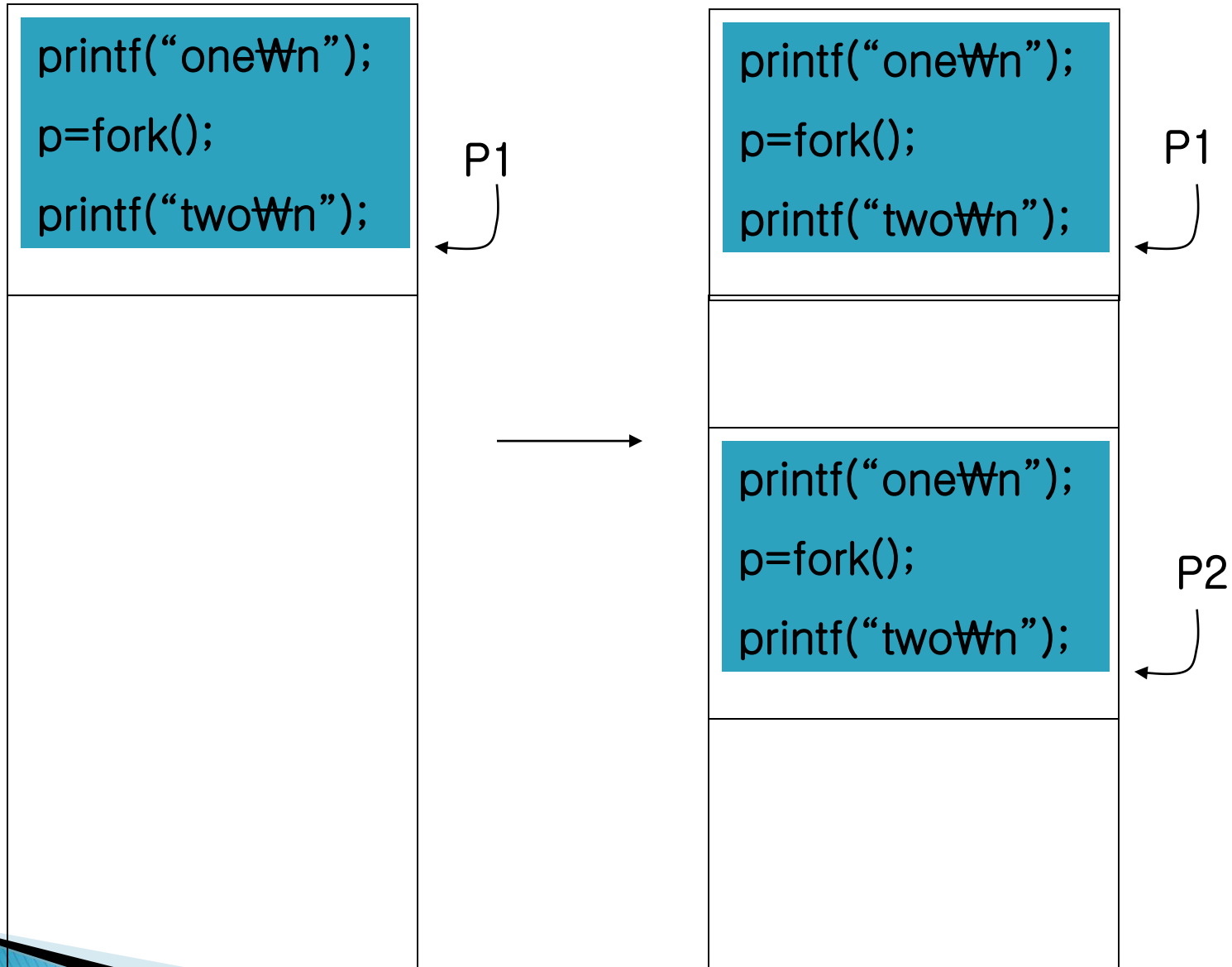
```
#include <unistd.h>
```

```
pid_t fork(void);
```

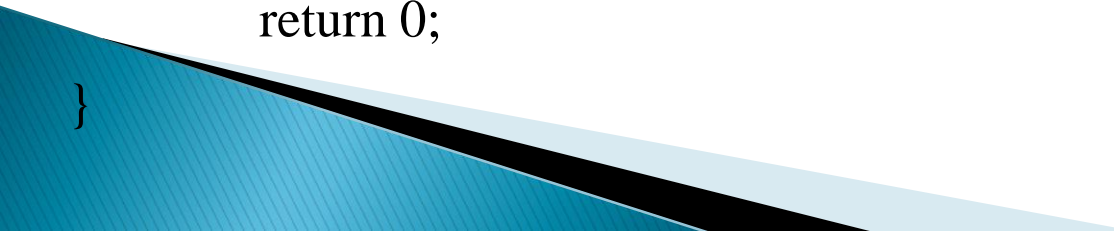
- 수행되던 process(parent)의 복사본 process(child) 생성
- fork() 바로 다음 문장부터 동시에 실행

Process의 생성 (2)

- 두 process의 차이점 :
 - pid와 ppid가 다르다
 - fork()의 return 값이 다르다.
 - parent process의 return 값은 child process의 process id;
 - child process의 return 값은 0이다;
- fork 실패 시 -1 return;
 - 실패 원인 :
 - 시스템 전체 process의 수 제한;
 - 한 process가 생성할 수 있는 process 수 제한;



```
int main(void){  
    pid_t pid;  
  
    printf("Calling fork ...\n");  
    pid=fork();  
  
    if (pid==0) {  
        printf("I am the child\n");  
    }  
    else if (pid > 0){  
        printf(" I am the parent, child has pid %d\n", pid);  
    }  
    else  
        printf("Fork returned error code\n");  
    return 0;  
}
```



fork : 파일과 자료

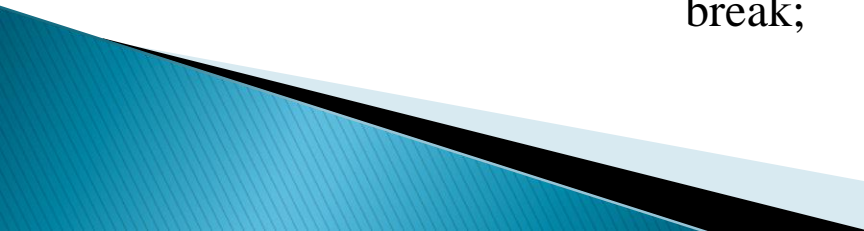
- ▶ child process는 parent process의 복제;
 - 모든 변수 값이 그대로 복제 된다.
 - fork()후에 변경된 값은 복제되지 않는다;
 - file descriptor도 복제 된다.
 - parent process가 open한 file은 child process에게도 open;
 - parent와 child가 file을 공통으로 사용 가능;

fork : 파일과 자료 (2)

```
#include<unistd.h>
#include<fcntl.h>
main(void){
    int fd;
    pid_t pid;
    char buf[10];

    fd=open("data", O_RDONLY);
    read(fd, buf, 10);
    printf("before fork: %ld\n", lseek(fd, (off_t)0, SEEK_CUR));

    switch (pid=fork()){
        case -1: perror("fork failed\n");
                exit(1);
                break;
```



fork : 파일과 자료 (3)

case 0:

```
printf("child before read: %ld\n", lseek(fd, (off_t)0, SEEK_CUR));  
read(fd, buf, 10);  
printf("child after read: %ld\n", lseek(fd, (off_t)0, SEEK_CUR));  
break;
```

default:

```
wait((int*) 0);  
printf("parent after wait: %ld\n", lseek(fd, (off_t)0, SEEK_CUR));
```

```
}
```

```
}
```

실행 결과 :

```
before fork :  
child before read :  
child after read :  
parent after wait :
```


exit 시스템 호출

▶ 사용법 :

```
#include <stdlib.h>  
void exit(int status);
```

- exit: process 정지 → open된 file 닫기 → clean-up-action
- status의 값은 프로세스 종료 후,
\$ echo \$?
명령에 의해 알아낼 수 있다.

exit 시스템 호출 (2)

- ▶ clean-up-action 지정 :

```
#include <stdlib.h>
```

```
int atexit(void (*func) (void));
```

- ▶ 지정된 순서의 역순으로 실행;

exec을 이용하여 새 프로그램을 수행

▶ 사용법 :

```
#include <unistd.h>
```

```
int execl(const char *path, const char *arg0, ..., const  
char *argn, (char *) 0);
```

```
int execlp(const char *file, const char *arg0, ..., const  
char *argn, (char *) 0);
```

```
int execv(const char *path, char *const argv[]);
```

```
int execvp(const char *file, char *const argv[]);
```

exec을 이용하여 새 프로그램을 수행 (2)

▶ 공통점 :

- 호출 프로세스의 주기억장치 공간에 새로운 프로그램을 적재;
 - ➔ 호출 프로세스는 no longer exists;
 - ➔ 새 process는 처음부터 실행;
 - ➔ 새 process는 호출 프로세스의 id로 실행;
- 실패 시 -1 return; 성공 시 return이 없다;
- fork와의 차이점 : 기존 프로세스와 병렬 수행이 아니다.

```
printf(...);  
execl("bin/ls", ...);  
printf("a");
```

P1



```
/* ls 의 첫 줄 */
```

P1



exec을 이용하여 새 프로그램을 수행 (3)

▶ 차이점 :

- path : 파일의 경로 이름 포함; vs. file : 파일 이름;
- 인수 :
 - arg0 : 프로그램 이름(경로 이름 빼고); and arg1,...,argn : 프로그램에 입력으로 사용될 인수들, 마지막엔 null pointer;
 - argv[] : 배열로 받기;

▶ file 이름을 쓰는 경우는? 환경 변수에 의해 설정된 path안의 file; (\$echo \$PATH)

exec을 이용하여 새 프로그램을 수행 (4)

▶ 예:

```
#include<unistd.h>

main(){

    printf("executing a.out \n");
    execl("./a.out", "a.out", "3", (char *) 0);
    printf("execl failed to run a.out");
    exit(1);

}
```

exec을 이용하여 새 프로그램을 수행 (5)

▶ 예 :

```
#include<unistd.h>

main(){

    char *const av[]={“a.out”, “3”, (char *) 0};

    printf(“executing a.out \n”);

    execv(“./a.out”, av);

    printf(“execv failed to run a.out”);

    exit(1);

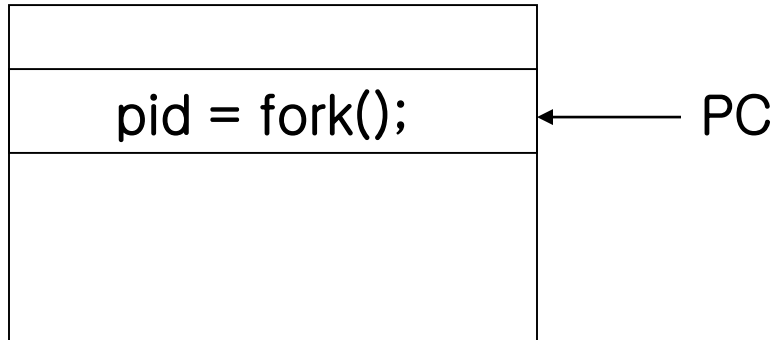
}
```


exec와 fork를 함께 사용

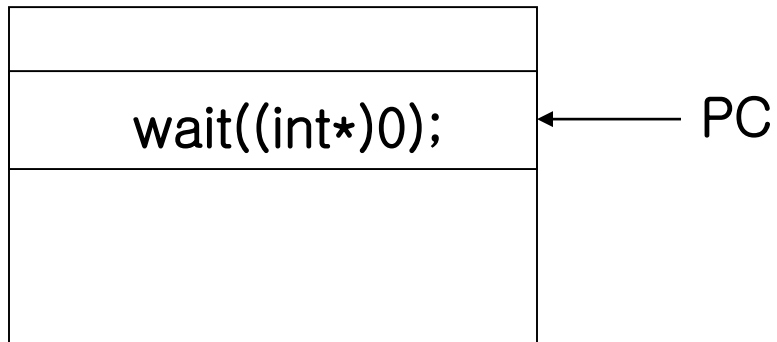
▶ 예:

```
#include <unistd.h>
main() {
    pid_t pid;
    switch (pid=fork()) {
        case -1: perror("fork failed");
                exit(1);
                break;
        case 0:  execl("/bin/ls", "ls", "-l", (char *) 0);
                perror("exec failed");
                exit(1);
                break;
        default: wait((int*) 0);
                printf("ls completed\n");
                exit(0);
    }
}
```

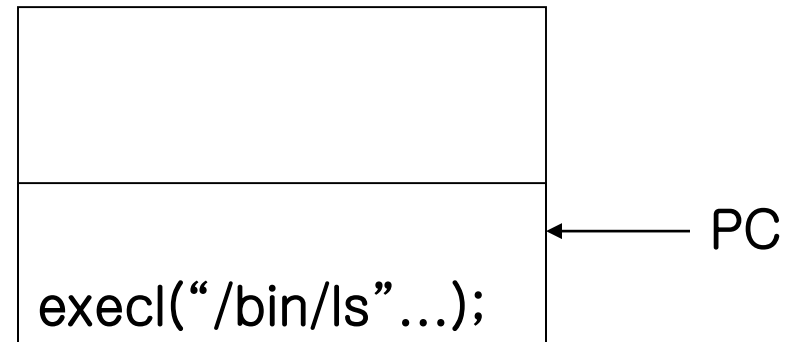
exec와 fork를 함께 사용 (2)



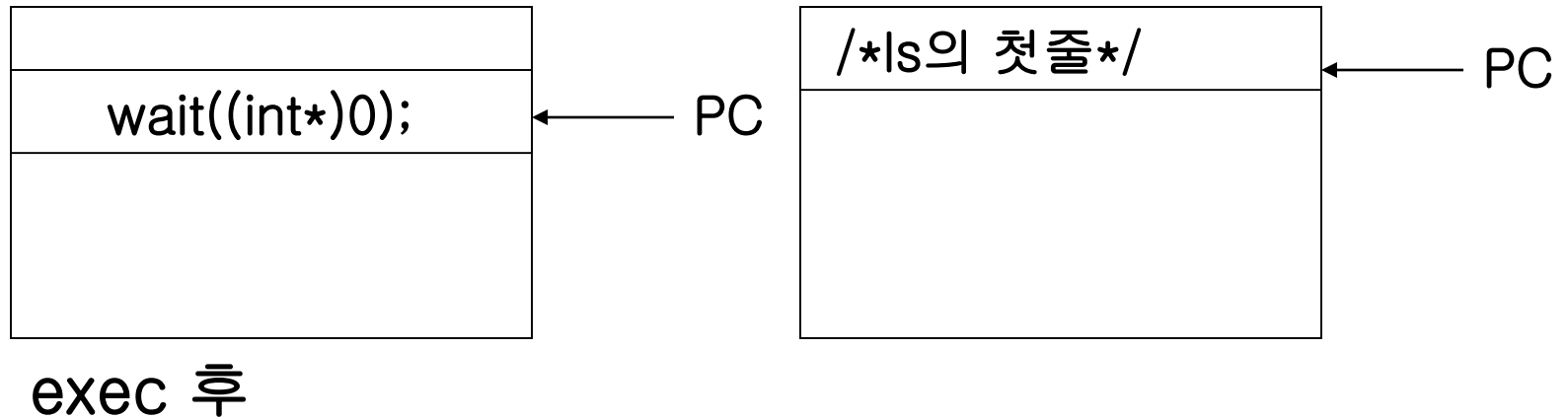
fork 전



fork 후



exec와 fork를 함께 사용 (3)



프로세스의 동기화

▶ wait 시스템 호출 :

```
#include <sys/wait.h>
```

```
#include <sys/types.h>
```

```
pid_t wait (int *status);
```

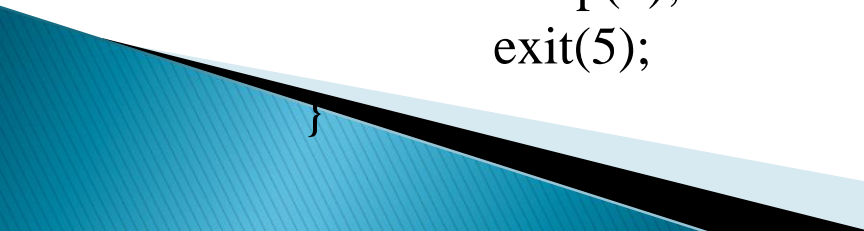
- 하나 이상의 child process 수행 시 아무나 하나가 종료 되면 return 된다;
- return 값 : 종료된 child의 id or -1 (살아있는 child process가 없는 경우)
- status : child의 종료 상태가 전달;

프로세스의 동기화 (2)

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
main() {
    pid_t pid;
    int status, exit_status;

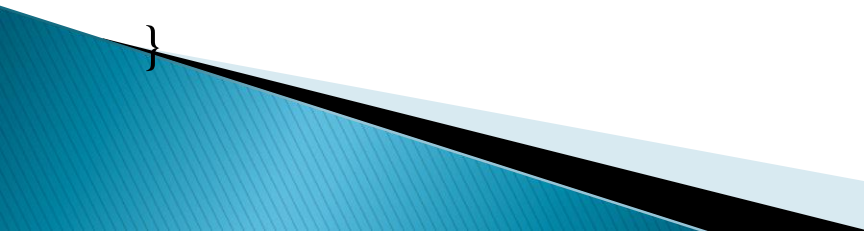
    if ((pid=fork()) < 0) {
        perror("fork failed");
        exit(1);
    }

    if (pid==0) {
        sleep(4);
        exit(5);
    }
```



프로세스의 동기화 (3)

```
if ((pid=wait(&status))==-1) {  
    perror("wait failed");  
    exit(2);  
}  
if (WIFEXITED(status)) {  
    exit_status=WEXITSTATUS(status);  
    printf("Exit status from %d was %d\n", pid, exit_status);  
}  
exit(0);  
}
```



프로세스의 동기화 (4)

- ▶ `WIFEXITED(status)` : `status`의 하위 8bit가 0인지 검사; 정상종료인지 검사;
- ▶ `WEXITSTATUS(ststus)` : `status`의 상위 비트에 저장된 값을 return;
- ▶ 왜? `child`가 `parent`에게 전달하는 값이 `status`의 상위 8bit에 저장된다.

프로세스의 동기화 (5)

▶ waitpid 시스템 호출 :

```
#include <sys/types.h>
```

```
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- pid : 기다리고 싶은 child의 id;
- status : child의 종료 상태;
- options : WNOHANG; child가 종료하지 않았으면 0을 return;

Zombie와 너무 이른 퇴장

- ▶ 부모 프로세스가 `wait`를 수행하지 않고 있는 상태에서 자식이 퇴장할 때 → `child`는 `zombie`가 된다.
- ▶ 하나 이상의 자식 프로세스가 수행되고 있는 상태에서 부모가 퇴장할 때 → `child`는 `init(pid=1)`의 `child`로 남는다.