프로세스 동기화 함수[1]

프로세스 동기화: wait(3)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

- stat_loc : 상태정보를 저장할 주소
- wait 함수는 자식 프로세스가 종료할 때까지 부모 프로세스를 기다리게 함

- 부모 프로세스가 wait 함수를 호출하기 전에 자식 프로세스가 종료하면 wait 함수는 즉시 리턴
- wait 함수의 리턴값은 자식 프로세스의 PID
- 리턴값이 -1이면 살아있는 자식 프로세스가 하나도 없다는 의미

[예제 6-8] wait 함수 사용하기 (test1.c)

```
...
07  int main(void) {
08      int status;
09      pid_t pid;
11      switch (pid = fork()) {
12          case -1 :  /* fork failed */
13              perror("fork");
14              exit(1);
15              break;
16          case 0 :   /* child process */
17              printf("--> Child Process\n");
18              exit(2);
19              break;
20          default :   /* parent process */
21              while (wait(&status) != pid)
22                  continue;
23              printf("--> Parent process\n");
24              printf("Status: %d, %x\n", status, status);
25              printf("Child process Exit Status:%d\n",status >> 8);
26              break;
27      }
29      return 0;
30  }
```

```
# ex6_8.out
--> Child Process
--> Parent process
Status: 512, 200
Child process Exit Status:2
```

자식 프로세스의 종료를 기다림

오른쪽으로 8비트 이동해야 종료 상태값을 알 수 있음

프로세스 동기화 함수[2]

특정 자식 프로세스와 동기화: waitpid(3)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

- **pid에 지정할 수 있는 값**
  - -1보다 작은 경우 : pid의 절댓값과 같은 프로세스 그룹ID에 속한 자식 프로세스 중 임의의 프로세스의 상태값 요청
  - -1인 경우 : wait 함수처럼 임의의 자식 프로세스의 상태값을 요청
  - 0인 경우 : 함수를 호출한 프로세스와 같은 프로세스 그룹에 속한 임의의 프로세스의 상태값 요청
  - 0보다 큰 경우 : 지정한 PID의 상태값 요청

- **options: waitpid 함수의 리턴 조건**
  - WCONTINUED: 수행중인 자식 프로세스의 상태값 리턴
  - WNOHANG: pid로 지정한 자식프로세스의 상태값을 즉시 리턴받을 수 없어도 이를 호출한 프로세스의 실행을 블록하지 않고 다른 작업을 수행토록 함
  - WNOWAIT: 상태값을 리턴한 프로세스가 대기 상태에 머물 수 있도록 함
  - WUNTRACED: 실행을 중단한 자식 프로세스의 상태값을 리턴

[예제 6-9] waitpid 함수 사용하기 (test2.c)

```
...
07  int main(void) {
08      int status;
09      pid_t pid;
10
11      if ((pid = fork()) < 0) { /* fork failed */
12          perror("fork");
13          exit(1);
14      }
15
16      if (pid == 0) { /* child process */
17          printf("--> Child process\n");
18          sleep(3);
19          exit(3);
20      }
21
22      printf("--> Parent process\n");
23
24      while (waitpid(pid, &status, WNOHANG) == 0) {
25          printf("n");
26          sleep(1);
27      }
28
29      printf("Child Exit Status : %d\n", status>>8);
30
31      return 0;
32  }
```

```
# ex6_9.out
--> Child process
--> Parent process
Parent still wait...
Parent still wait...
Parent still wait...
Child Exit Status : 3
```

WNOHANG이므로 waitpid 함수는 블록되지 않고 25~26행 반복 실행

파이프

봉쇄되지 않는 read/write (test3.c)

- if (fcntl(filedes, F_SETFL, O_NONBLOCK) == -1) perror
- filedes가 파이프에 대한 쓰기 전용 파일기술자이면, 파이프에 대한 write는 파이프가 완전히 차 있더라도 봉쇄되지 않음. 대신 write는 -1로 복귀하고 errno를 EAGAIN으로 지정
- 파이프에 대한 읽기 전용 파일기술자라면 즉시 -1로 복귀. Errno는 EAGAIN으로 지정

select (test4.c)

- 부모프로세스가 서버프로세스로 동작하고, 자신과 통신하는 클라이언트(자식)프로세스를 임의의 수만큼 가지는 경우 사용

```
include <sys/time.h>
int select (int nfds, fd_set *readfs, fd_set *writefs, fd_set *errorfs, struct timeval *timeout)
```

  - nfds : 서버가 잠재적 흥미를 가지는 파일기술자의 수
    0(stdin), 1(stdout), 2(stderr)는 default, 두개의 파일을 더 개방하면 nfds = 5
  - fd_set으로 정의된 인수들은 비트마스크. 각 비트가 하나의 파일기술자를 나타냄. 한 비트가 켜져 있으면 해당 파일 기술자에 대한 흥미를 나타냄.
    readfs: 읽을 만한 가치가 있는 것이 있는가?
    writefs: 임의의 주어진 파일기술자가 쓰기를 받아들일 준비가 되어 있는가?
    errorfs: 주어진 파일기술자중 하나라도 오류가 발생했는가?

```
/* fdset가 가리키는 마스크를 초기화 */
void FD_ZERO(fd_set *fdset);
/* fdset가 가리키는 마스크내의 비트, fd를 1로 설정 */
void FD_SET(int fd, fd_set *fdset);
/* fdset가 가리키는 마스크내의 비트, fd가 설정되어 있는가?
int FD_ISSET(int fd, fd_set *fdset);
/* fdset가 가리키는 마스크내의 비트, fd를 0으로 설정
void FD_CLR(int fd, fd_set *fdset);
```

파이프

Timeout은 struct timeval에 대한 포인터

- #include <sys/time.h>
- struct timeval {
- long tv_sec;
- long tv_usec;
- }
- 포인터가 널이면 select 는 흥미있는 일이 일어날때까지 봉쇄. 만일 timeout이 0초를 포함하는 구조를 가리키면 즉각 복귀. 0이 아닌 값을 포함하고 있으면 지정된 시간 후에 복귀

select의 복귀 값은 오류시 -1, 타임 아웃시는 0, 아니면 흥미 있는 파일기술자의 수를 나타내는 정수

파이프

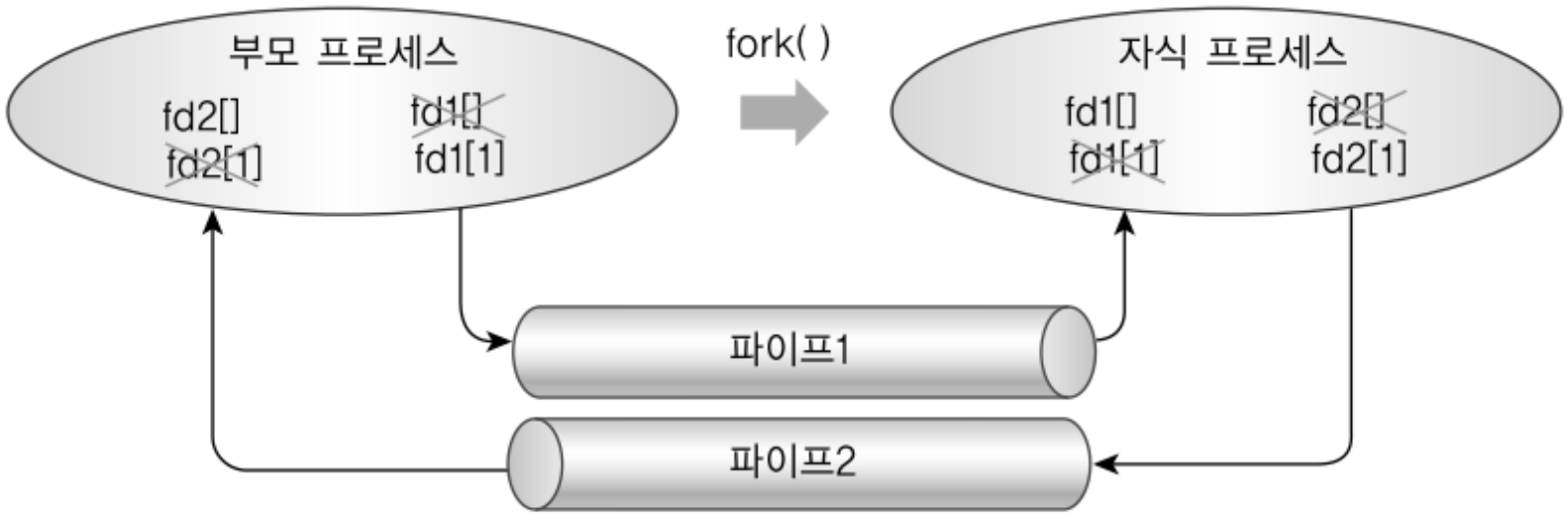
```
int fd1, fd2;
fd_set readset;

fd1 = open( “file1” , O_RDONLY);
fd2 = open( “file2” , O_RDONLY);
FD_ZERO(&readset);
FD_SET(fd1, &readset);   FD_SET(fd2, &readset);
switch(select (5, &readset, NULL, NULL, NULL))
 {

 }
```

양방향 파이프의 활용

- ps –ef | grep telnet (test5.c)
- 양방향 통신
  - 파이프는 기본적으로 단방향이므로 양방향 통신을 위해서는 파이프를 2개 생성한다.



[그림 9-4] 양방향 통신 개념도

[예제 9-5] 양방향 통신하기 (test6.c)

```
...
07  int main(void) {
08      int fd1[2], fd2[2];
09      pid_t pid;
10      char buf[257];
11      int len, status;
12
13      if (pipe(fd1) == -1) {
14          perror("pipe");
15          exit(1);
16      }
17
18      if (pipe(fd2) == -1) {
19          perror("pipe");
20          exit(1);
21      }
22
23      switch (pid = fork()) {
24          case -1 :
25              perror("fork");
26              exit(1);
27              break;
```

파이프 2개를 생성하기 위해 배열2개 선언

파이프 2개 생성

[예제 9-5] 양방향 통신하기

```
28          case 0 : /* child */
29              close(fd1[1]);
30              close(fd2[0]);
31              write(1, "Child Process:", 15);
32              len = read(fd1[0], buf, 256);
33              write(1, buf, len);
34
35              strcpy(buf, "Good\n");
36              write(fd2[1], buf, strlen(buf));
37              break;
38          default :
39              close(fd1[0]);
40              close(fd2[1]);
41              buf[0] = '\0';
42              write(fd1[1], "Hello\n", 6);
43
44              write(1, "Parent Process:", 15);
45              len = read(fd2[0], buf, 256);
46              write(1, buf, len);
47              waitpid(pid, &status, 0);
48              break;
49      }
50
51      return 0;
52  }
```

자식 프로세스
-fd1[0]으로 읽기
-fd2[1]로 쓰기

부모 프로세스
-fd1[1]로 쓰기
-fd2[0]으로 읽기

```
# ex9_5.out
Child Process:Hello
Parent Process:Good
```