exec 함수군 활용

exec 함수군

- exec로 시작하는 함수들로, 명령이나 실행 파일을 실행할 수 있다.
- exec 함수가 실행되면 프로세스의 메모리 이미지는 실행파일로 바뀐다.

exec 함수군의 형태 6가지

끝에 l: 일일이 argn 나열. 끝에 v: argv 배열 사용. 끝에 e: 환경 변수 사용.

끝에 p: path 대신에 file

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ..., const char *argn,
(char *)0);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ..., const char *argn,
(char *)0, char *const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ..., const char *argn,
(char *)0);
int execvp(const char *file, char *const argv[]);
```

- path : 명령의 경로 지정
- file : 실행 파일명 지정
- arg#, argv : main 함수에 전달할 인자 지정
- envp : main 함수에 전달할 환경변수 지정
- 함수의 형태에 따라 NULL 값 지정에 주의해야 한다.

exec 함수군 활용

```
printf(…);
execl("/bin/ls",…);
printf("ERROR₩n");
```

화살표 위까지는 정상 작동.

exec 함수를 만나는 순간, 이 프로그램이 완전히 ls 코드로 바뀜.

위 exec 함수가 정상 작동했다면, 출력되지 않음.

```
/* ls code */
```

[예제 6-4] execlp 함수 사용하기(test1.c)

```
01  #include <unistd.h>
02  #include <stdlib.h>
03  #include <stdio.h>
04
05  int main(void) {
06      printf("--> Before exec function\n");
07
08      if (execlp("ls", "ls", "-a", (char *)NULL) == -1) {
09          perror("execlp");
10          exit(1);
11      }
12
13      printf("--> After exec function\n");
14
15      return 0;
16  }
```

인자의 끝을 표시하는 NULL 포인터

첫 인자는 관례적으로 실행파일명 지정

메모리 이미지가 'ls' 명령으로 바뀌어 13행은 실행안됨

```
# ex6_4.out
--> Before exec function
.       ex6_1.c      ex6_3.c      ex6_4.out
..      ex6_2.c      ex6_4.c      han.txt
```

[예제 6-5] execv 함수 사용하기(test2.c)

```
01  #include <unistd.h>
02  #include <stdlib.h>
03  #include <stdio.h>
04
05  int main(void) {
06      char *argv[3];
07
08      printf("Before exec function\n");
09
10      argv[0] = "ls";
11      argv[1] = "-a";
12      argv[2] = NULL;
13      if (execv("/usr/bin/ls", argv) == -1) {
14          perror("execv");
15          exit(1);
16      }
17
18      printf("After exec function\n");
19
20      return 0;
21  }
```

첫 인자는 관례적으로 실행파일명 지정

인자의 끝을 표시하는 NULL 포인터

경로로 명령 지정

역시 실행안 됨

```
# ex6_5.out
--> Before exec function
.    ex6_1.c   ex6_3.c   ex6_5.c     han.txt
..   ex6_2.c   ex6_4.c   ex6_5.out
```

[예제 6-6] execve 함수 사용하기(test3.c)

```
...
05  int main(void) {
06      char *argv[3];
07      char *envp[2];
08
09      printf("Before exec function\n");
10
11      argv[0] = "arg.out";
12      argv[1] = "100";
13      argv[2] = NULL;
14
15      envp[0] = "MYENV=hanbit";
16      envp[1] = NULL;
17
18      if (execve("./arg.out", argv, envp) == -1) {
19          perror("execve");
20          exit(1);
21      }
22
23      printf("After exec function\n");
24
25      return 0;
26  }
```

실행파일명 지정

인자의 끝을 표시하는 NULL 포인터

환경변수 설정

ex6_6_arg.c를 컴파일하여 생성

```
# ex6_6.out
--> Before exec function
--> In ex6_6_arg.c Main
argc = 2
argv[0] = arg.out
argv[1] = 100
MYENV=hanbit
```

[예제 6-6] (2) arg.c 파일 (arg.out)

```
01  #include <stdio.h>
02
03  int main(int argc, char **argv, char **envp) {
04      int n;
05      char **env;
06
07      printf("\n--> In ex6_6_arg.c Main\n");
08      printf("argc = %d\n", argc);
09      for (n = 0; n < argc; n++)
10          printf("argv[%d] = %s\n", n, argv[n]);
11
12      env = envp;
13      while (*env) {
14          printf("%s\n", *env);
15          env++;
16      }
17
18      return 0;
19  }
```

환경변수 또한 받아야 함.

인자 값 출력

환경변수 출력

exec 함수군과 fork 함수

Compiler directive 사용

시간이 얼마나 걸리는지 측정하고 싶다면 test5.c 참고

```
#ifdef T
        start = dclock();
#endif
```

원래는 되야 하나 구버전에서는 작동하지 않으므로, gettimeofday() 로 바꿔서 써야 함.

의 사이에 시간측정 관련된 코드를 작성.

시간측정부분 시간을 측정하고 싶은 구간

```
#ifdef T
        end = dclock() - start;
        printf(end);
#endif
```

gettimeofday()

작성하지 않으면, '#ifdef T' ~ '#endif' 사이의 코드는 실행 안됨.

즉, 시간 측정을 원치 않으면, gcc -o sample sample.c 로 하면 됨.

```
gcc -DT -o sample sample.c
```

컴파일 할 때, 입력해야 할 명령어

TIMES == 377321 377321 O
Compiler directive 에 의해 출력된 부분
O 은 실행속도가 너무 빨라 측정이 안됨.

fork로 생성한 자식 프로세스에서 exec 함수군을 호출

- 자식 프로세스의 메모리 이미지가 부모 프로세스 이미지에서 exec 함수로 호출한 새로운 명령으로 대체
- 자식 프로세스는 부모 프로세스와 다른 프로그램 실행 가능
- 부모 프로세스와 자식 프로세스가 각기 다른 작업을 수행해야 할 때 fork와 exec 함수를 함께 사용

[예제 6-7] fork와 exec 함수 사용하기

```
...
06  int main(void) {
07      pid_t pid;
08
09      switch (pid = fork()) {
10          case -1 :  /* fork failed */
11              perror("fork");
12              exit(1);
13              break;
14          case 0 :   /* child process */
15              printf("--> Child Process\n");
16              if (execlp("ls", "ls", "-a", (char *)NULL) == -1) {
17                  perror("execlp");
18                  exit(1);
19      }
20              exit(0);
21              break;
22          default :   /* parent process */
23              printf("--> Parent process - My PID:%d\n",(int)getpid());
24              break;
25  }
27      return 0;
28   }
```

자식 프로세스가 실행하는 부분

자식프로세스에서 execlp 함수 실행

부모프로세스는 이 부분 실행

```
# ex6_7.out
--> Child Process
ex6_1.c  ex6_3.c  ex6_5.c  ex6_6_arg.c  ex6_7.out
ex6_2.c  ex6_4.c  ex6_6.c  ex6_7.c      han.txt
--> Parent process - My PID:10535
```

실습

- 부모 프로세스에서 개방된 파일은 자식 프로세스에서도 개방되어 있고, 자식은 각 파일과 연관하여 자신의 파일 기술자의 복제를 유지하고 있다. 이와 관련된 사항을 확인하는 코드를 다음과 같은 차례로 작성하라
  - 데이터 파일을 open
  - 10bytes을 읽고, 현재 파일 위치를 출력
  - fork를 호출 offset 이 0 에 위치한 것이 아니라 10 bytes 에 위치.
  - 자식은 10bytes를 읽고, 현재 파일 위치를 출력 → 실행 후, 20 bytes 에 위치하게 됨.
  - 부모는 wait((int *)0)로 자식이 종료하기를 기다린 후, 현재 파일 위치를 출력
    20 bytes 를 출력.

시그널의 개념

시그널

- 소프트웨어 인터럽트 프로세스에 뭔가 발생했음을 알리는 간단한 메시지를 비동기적으로 보내는 것

발생사유

- 0으로 나누기처럼 프로그램에서 예외적인 상황이 일어나는 경우
- kill 함수처럼 시그널을 보낼 수 있는 함수를 사용해서 다른 프로세스에 시그널을 보내는 경우 시그널 발생
- 사용자가 Ctrl+C와 같이 인터럽트 키를 입력한 경우

시그널 처리방법

- 각 시그널에 지정된 기본 동작 수행. 대부분의 기본 동작은 프로세스 종료
- 시그널을 무시
- 시그널 처리를 위한 함수(시그널 핸들러)를 지정해놓고 시그널을 받으면 해당 함수 호출
- 시그널이 발생하지 않도록 블록처리

인터럽트를 처리하는 것은 인터럽트 핸들러

시그널 핸들러에 지정된 내용을 수행할 수 있음.

일단은 블록 처리하고 시그널이 해제가 되면,
발생된 시그널을 쭉 처리.

들어오는 시그널들을 큐에 넣어서 순서대로 처리해야 함.

시그널의 종류

이런 것이 있다 정도만 알아두면 됨.

시그널	번호	기본 처리	발생 요건
SIGHUP	1	종료	행업으로 터미널과 연결이 끊어졌을 때 발생
SIGINT	2	종료	인터럽트로 사용자가 Ctrl+C를 입력하면 발생
SIGQUIT	3	코어 덤프	종료 신호로 사용자가 Ctrl+\를 입력하면 발생
SIGILL	4	코어 덤프	잘못된 명령 사용
SIGTRAP	5	코어 덤프	추적(trace)이나 중단점(breakpoint)에서 트랩 발생
SIGABRT	6	코어 덤프	abort 함수에 의해 발생
SIGEMT	7	코어 덤프	에뮬레이터 트랩으로 하드웨어에 문제가 있을 경우 발생
SIGFPE	8	코어 덤프	산술 연산 오류로 발생
SIGKILL	9	종료	강제 종료로 발생
SIGBUS	10	코어 덤프	버스 오류로 발생
SIGSEGV	11	코어 덤프	세그먼테이션 폴트로 발생
SIGSYS	12	코어 덤프	잘못된 시스템 호출로 발생
SIGPIPE	13	종료	잘못된 파이프 처리로 발생
SIGALRM	14	종료	알람에 의해 발생
SIGTERM	15	종료	소프트웨어적 종료로 발생
SIGUSR1	16	종료	사용자 정의 시그널 1

이외에도 시그널의 종류는 다양함 (표 7-7 참조)

시그널 보내기[1]

kill 명령

- 프로세스에 시그널을 보내는 명령
- 예 : 3255번 프로세스에 9번 시그널(SIGKILL) 보내기 -> 프로세스 강제 종료

```
# kill -9 3255 명령어
```

시그널 보내기: kill(2) 함수 버전

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

int kill(pid_t pid, int sig);
```

- pid가 0보단 큰 수 : pid로 지정한 프로세스에 시그널 발송
- pid가 -1이 아닌 음수 : 프로세스 그룹ID가 pid의 절대값인 프로세스 그룹에 속하고 시그널을 보낼 권한을 가지고 있는 모든 프로세스에 시그널 발송 받을 수 있는 모든 프로세스에게
- pid가 0 : 특별한 프로세스를 제외하고 프로세스 그룹ID가 시그널을 보내는 프로세스의 프로세스 그룹ID와 같은 모든 프로세스에게 시그널 발송 현재 이 함수를 실행하고 있는 프로세스와 같은 그룹의 프로세스들
- pid가 -1 : 시그널을 보내는 프로세스의 유효 사용자ID가 root가 아니면, 특별한 프로세스를 제외하고 프로세스의 실제 사용자ID가 시그널을 보내는 프로세스의 유효 사용자 ID와 같은 모든 프로세스에 시그널 발송

[예제 7-1] kill 함수 사용하기(test6.c)

ex7_1.c

```
01  #include <sys/types.h>
02  #include <unistd.h>
03  #include <signal.h>
04  #include <stdio.h>
05
06  int main(void) {
07      printf("Before SIGCONT Signal to parent.\n");
08
09      kill(getppid(), SIGCONT);
10
11      printf("Before SIGQUIT Signal to me.\n");
12
13      kill(getpid(), SIGQUIT);
14
15      printf("After SIGQUIT Signal.\n");
16
17      return 0;
18  }
```

종료했기 때문에

SIGQUIT의 기본동작은 코어덤프

출력하지 못함.
(코어덤프)

```
# ex7_1.out
Before SIGCONT Signal to parent.
Before SIGQUIT Signal to me.
끝(Quit)(코어 덤프)
```

시그널 보내기[2]

시그널 보내기: raise(2)

```
#include <signal.h>

int raise(int sig);
```

- 함수를 호출한 프로세스에 시그널 발송
- 만약 시그널 핸들러가 호출되면 시그널 핸들러의 수행이 끝날 때까지 리턴하지 않는다.
- 성공하면 0, 실패하면 -1 return

시그널 보내기: abort(3)

```
#include <stdlib.h>

void abort(void);
```

- 함수를 호출한 프로세스에 SIGABRT시그널 발송
- SIGABRT 시그널은 프로세스를 비정상적으로 종료시키고 코어덤프 생성

시그널 핸들러 함수[1]

시그널 핸들러

- 시그널을 받았을 때 이를 처리하기 위해 지정된 함수
- 프로세스를 종료하기 전에 처리할 것이 있거나, 특정 시그널에 대해 종료하고 싶지 않을 경우 지정

시그널 핸들러 지정: signal(3)

```
#include <signal.h>

void (*signal(int sig, void (*disp)(int)))(int);
```

- disp : sig로 지정한 시그널을 받았을 때 처리할 방법
  - 시그널 핸들러 함수명
  - SIG_IGN : 시그널을 무시하도록 지정
  - SIG_DFL : 기본 처리 방법으로 처리하도록 지정
- signal함수는 시그널이 들어올 때마다 시그널 핸들러를 호출하려면 매번 시그널 핸들러를 재지정해야함.

이것은 SUN OS 의 얘기임.
리눅스의 경우에는 일단, 시그널 핸들러를
지정을 해놓으면 시그널이 들어올 때 마다
지정된 시그널 핸들러로 호출이 됨.
SUN OS 의 경우, 한 번 지정한 핸들러로
처리한 이후에 두 번째 시그널이 들어오면
종료 프로세스를 종료하는 Defalut
handler 로 수행됨.

[예제 7-2] signal 함수 사용하기(test7.c)

ex7_2.c

```
...   시그널이 발생할 때 마다 이 함수를 실행.
07  void handler(int signo) {
08      printf("Signal Handler Signal Number : %d\n", signo); 시그널 넘버 출력
09      psignal(signo, "Received Signal"); 시그널 메시지를 출력하는 함수
10  }
11
12  int main(void) {
13      void (*hand)(int); 시그널 핸들러의 포인터
14
15      hand = signal(SIGINT, handler);
16      if (hand == SIG_ERR) {
17          perror("signal");
18          exit(1);
19      }
20
21      printf("Wait 1st Ctrl+C... : SIGINT\n");
22      pause(); 원래는 단순히 기다리는 함수지만, 여기서는 컨트롤 + C 를 눌러 시그널 발생.
23      printf("After 1st Signal Handler\n");
24      printf("Wait 2nd Ctrl+C... : SIGINT\n");
25      pause();
26      printf("After 2nd Signal Handler\n");27
28      return 0;
29  }
```

```
# ex7_2.out
Wait 1st Ctrl+C... : SIGINT
^CSignal Handler Signal Number : 2
Received Signal: Interrupt
After 1st Signal Handler
Wait 2nd Ctrl+C... : SIGINT
^C# SUN OS 에서는 Default 로 돌아가서 그런 것.
```

두번째 Ctrl+C는 처리못함

리눅스에서는 그렇지 않음.

리눅스에서는 실행됨.

[예제 7-3] 시그널 핸들러 재지정하기(test8.c)

ex7_3.c

```
... 핸들러 내에서 핸들러를 재지정     이것을 리눅스에서는 굳이 해줄 필요는 없는 것 같음. (SUN OS 전용)
07  void handler(int signo) {
08      void (*hand)(int);
09      hand = signal(SIGINT, handler);
10      if (hand == SIG_ERR) {
11          perror("signal");
12          exit(1);
13      }
14
15      printf("Signal Handler Signal Number: %d\n", signo);
16      psignal(signo, "Received Signal");
17  }
...
```

시그널 핸들러 재지정

두번째 Ctrl+C도 처리

```
# ex7_3.out
Wait 1st Ctrl+C... : SIGINT
^CSignal Handler Signal Number: 2
Received Signal: Interrupt
After 1st Signal Handler
Wait 2nd Ctrl+C... : SIGINT
^CSignal Handler Signal Number: 2
Received Signal: Interrupt
After 2nd Signal Handler
```