**System call 종류 (운영체제 Homework #2)**

2018112178 임연지

**❑ Process & Job Manipulation (or Control)**

**1. load, execute, end, abort**

**abort()**

void abort ( void );

exit() 기능과 달리 abort()는 열려 있는 파일을 닫을 수 없다. 또한 임시 파일을 삭제할 수 없으며 스트림 버퍼를 flush할 수 없다. 또한 atexit()에 등록된 함수도 호출하지 않는다.  
  
이 함수는 실제로 SIGABRT 신호를 상승시켜 프로세스를 종료하며, 프로그램에는 이 신호를 가로채는 handler가 포함될 수 있다.  
  
따라서 아래와 같은 프로그램은 "Geeks for Geeks"를 "tempfile.txt"에 쓰지 않을 수 있다.

Abort 예시:

#include<stdio.h>

#include<stdlib.h>

int main()

{

  FILE \*fp = fopen("C:\\myfile.txt", "w");

  if(fp == NULL)

  {

    printf("\n could not open file ");

    getchar();

    exit(1);

  }

  fprintf(fp, "%s", "Geeks for Geeks");

  /\* ....... \*/

  /\* Something went wrong so terminate here \*/

  abort();

  getchar();

  return 0;

}

**2. create & kill processes**

**fork()**

pid\_t fork(void);

프로세스가 자기 자신을 복제해서 새로운 프로세스를 만든다. 이 system call로 부모 프로세스가 자식 프로세스를 만든다.

새 자식 프로세스가 생성된 후 두 프로세스 모두 fork() system call 후 다음 명령을 실행한다. 하위 프로세스는 상위 프로세스에서 사용하는 동일한 pc(프로그램 카운터), 동일한 CPU 레지스터, 동일한 열린 파일을 사용한다.

파라미터가 없고 정수 변수를 반환한다.

\* 음수인 경우 : 자식 프로세스의 생성이 성공적이지 못했음을 뜻함

\* 0인 경우 : 새로 생성된 자식 프로세스로 반환된다.

\* 양수인 경우 : 부모나 caller로 반환된다. 변수는 새로 생성된 자식 프로세스의 프로세스 ID를 포함한다.

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

    // make two process which run same

    // program after this instruction

    fork();

    printf("Hello world!\n");

    return 0;

}

위의 코드의 결과로 Hello world!가 개행을 포함해 두 번 출력된다.

**kill()**

int kill(pid\_t pid, int sig);

kill() system call은 OS에서 프로세스를 종료하도록 촉구하는 프로세스에 종료 신호를 보내기 위해 사용된다. 그러나 kill()이 반드시 kill process를 의미하지는 않으며 다양한 의미를 가질 수 있다.

#include <sys/types.h>

#include <signal.h>

#include <stdlib.h>

int main(int argc, char \*\*argv)

{

int pid;

int sig\_num;

pid = atoi(argv[1]);

sig\_num = atoi(argv[2]);

kill(pid, sig\_num);

}

**3. get & set process attributes**

**getppid()**

pid\_t getppid(void);

호출 프로세스의 상위 프로세스 ID를 반환한다. 호출 프로세스가fork() 함수에 의해 생성되었고 getppid 함수 호출 시 상위 프로세스가 여전히 존재하는 경우, 이 함수는 상위 프로세스의 프로세스 ID를 반환한다. 그렇지 않으면, 이 함수는 1의 값을 반환하는데, 이것은 초기 프로세스의 프로세스 ID이다.

#include <iostream>

#include <unistd.h>

using namespace std;

int main()

{

    int pid;

    pid = fork();

    if (pid == 0)

    {

        cout << "\nParent Process id : "

             << getpid() << endl;

        cout << "\nChild Process with parent id : "

             << getppid() << endl;

    }

    return 0;

}

**4. wait for a certain time**

**pause()**

int pause(void)

함수로부터 시그널을 발견하게 되면 반환한다. 이 경우에 pause 는 –1과 EINTR에 의해서 세팅된 errno를 리턴한다. EINTR은 신호를 발견하고 신호 발견 함수가 반환된다.

#include <stdio.h>

#include <signal.h>

void sigint\_handler( int signo)

{

printf( "Ctrl-C \n");

}

int main( void)

{

signal( SIGINT, sigint\_handler);

pause();

}

**5. wait & signal events**

**wait()**

pid\_t wait(int \*status)

일부 시스템에서는 프로세스가 실행을 완료하기 위해 다른 프로세스를 기다려야 한다. 이러한 유형의 상황은 부모 프로세스가 자식 프로세스를 생성할 때 발생하며, 부모 프로세스의 실행은 자식 프로세스가 실행될 때까지 보류된 상태로 유지된다.  
부모 프로세스의 중단은 wait() 함께 자동으로 발생한다. 자식 프로세스가 실행을 종료하면 컨트롤은 다시 부모 프로세스로 이동한다.

#include<stdio.h>

#include<stdlib.h>

#include<sys/wait.h>

#include<unistd.h>

int main()

{

    pid\_t cpid;

    if (fork()== 0)

        exit(0);

    else

        cpid = wait(NULL);

    printf("Parent pid = %d\n", getpid());

    printf("Child pid = %d\n", cpid);

    return 0;

}

**6. allocate & free(deallocate) memory**

**fallocate()**

int fallocate(int fd, int mode, off\_t offset, off\_t len)

메모리 블록을 예약한다. 파일 시스템 단위에서 지원되어야 사용할 수 있으며 현재 Ext4와 XFS에서 지원 하고 있다고 한다. 향후 다양한 플랫폼 및 파일시스템에 이식 되어 널리 사용될 것이라고 한다.

fallocate()을 사용하면 호출자가 오프셋에서 시작하여 len bytes범위에 대해 fd로 참조되는 파일에 할당된 디스크 공간을 직접 조작할 수 있다.

mode인수는 주어진 범위에서 수행 될 작업을 결정한다. 현재 mode에는 하나의 플래그만 지원된다.

**pkey\_free()**

int pkey\_free(int pkey)

보호키를 해제하고 나중에 할당 할 수 있게 한다. 보호키를 해제한 후에는 더 이상 보호키 관련 작업에 사용할 수 없다. 응용 프로그램은 사용 중인 모든 보호키에서 pkey\_free()를 호출해서는 안 된다. 이 경우의 동작은 정의되어 있지 않으며, 오류가 발생할 수 있다. pkey는 key의 포인터에 대한 값이다.

**❑ File Manipulation**

**7. create & delete file**

**creat()**

int creat(const char \*pathname, mode\_t mode)

기존의 파일 또는 새로운 파일을 쓰기 전용으로 open한다.

pathname : open할 파일명에 대한 fullpath 또는 상대 path.

mode : 파일에 대한 access 권한을 설정합니다.

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int fd;

fd = creat(file, 0644);

**8. open & close file**

**open()**

int open(const char \*pathname, int flags, mode\_t mode)

파일 또는 장치로의 접근 경로를 만든다. 반환 값, file discriptor는 프로세스 독립적으로 고유하다.

pathname : 사용할 파일의 위치

flags : 사용할 방식

|  |
| --- |
| #include<stdio.h>  #include<fcntl.h>  #include<errno.h>  extern int errno;  int main()  {      int fd = open("foo.txt", O\_RDONLY | O\_CREAT);      printf("fd = %d/n", fd);        if (fd ==-1)      {          printf("Error Number % d\n", errno);          perror("Program");      }      return 0;  } |

**9. read, write & reposition**

**read()**

ssize\_t read(int fd, void \*buf, size\_t len)

fd로 표시된 파일에서 buf로 표시된 메모리 영역에 대한 len 바이트의 입력을 읽는다.

성공시 반환 값 : 읽은 바이트 수

파일 끝인 경우 반환 값 : 0

오류시 반환 값 : -1

1. fd 가 참조하는 파일의 현재 파일 오프셋에서 len 바이트 만큼 buf로 읽어 들인다.

2. 성공하면 buf에 쓴 바이트 숫자를 반환하고, 실패하면 -1을 반환한다.

3. 파일의 위치는 fd 에서 읽은 바이트 숫자만큼 전진한다.

#include<stdio.h>

#include <fcntl.h>

int main()

{

  int fd, sz;

  char \*c = (char \*) calloc(100, sizeof(char));

  fd = open("foo.txt", O\_RDONLY);

  sz = read(fd, c, 10);

  c[sz] = '\0';

}

**10. get & set file attributes**

**sendfile()**

ssize\_t sendfile(int out\_fd, int in\_fd, off\_t \*offset, size\_t count)

데이터는 가능한 경우 DMA를 통해 OS캐시로 바로 읽혀지고, TLB캐시는 손상되지 않는다. 메모리에 직접 접근하지 않고, 오버헤드를 줄임으로, sendfile()을 사용하는 응용프로그램의 성능은 높다.

In\_fd : 읽기전용으로 열린 file 디스크립터, out\_fd : 쓰기 전용으로 열린 디스크립터

**❑ Device Manipulation**

**11. request & release device**

**scull\_release()**

int scull\_release(struct inode \*inode, struct file \*filp)

open()의 반대 역할을 수행한다. 시스템 닫기 호출은 파일 구조에 대한 카운터가 0으로 떨어질 때만 release()를 실행한다. 이는 구조가 파괴될 때 발생한다. 릴리스 방법과 시스템 호출 사이의 이러한 관계는 모듈의 사용횟수가 일관되도록 한다.

int scull\_release(struct inode \*inode, struct file \*filp) {

MOD\_DEC\_USE\_COUNT;

return 0; }

**12. read, write & reposition**

**write()**

size\_t write( int fildes, const void \*buf, size\_t nbytes)

파일을 쓸 수도 있지만 디바이스에 출력에 값을 쓸 수도 있다. fildes의 값에 1이 들어가면 모니터로 값을 출력한다.

fildes – 디스크립터 번호.  
\*buf – 파일에 쓸 내용을 담은 버퍼.  
nbytes – 쓰기할 바이트 수.

성공 시 반환 값 : 성공 시 기록된 바이트 수

오류 시 반환 값 : -1

1. fd 가 참조하는 파일의 현재 위치의 시작 지점인 buf의 내용을 count 바이트 숫자만큼 기록한다.

2. 성공하면 쓰여진 바이트 숫자를 반환하고, 파일 위치도 같은 크기만큼 전진한다. 오류가 나면 -1 을 반환한다.

3. 파일이 O\_APPEND 등으로 파일의 쓰기 위치를 바꾸지 않는다면, 항상 처음 위치에 쓰여진다.

#include <unistd.h>

int main() {

write(1,”Helloworld!,11);

return 0;

}

**13. get & set device attributes**

**getcpu()**

int getcpu(unsigned \*cpu, unsigned \*node, struct getcpu\_cache \*tcache)

호출 중인 스레드 또는 프로세스가 현재 실행중인 프로세서 및 노드를 식별하고 이를 cpu 및 node인수가 가리키는 정수에 쓴다. 프로세서는 cpu를 식별하는 고유한 작은 정수이다. CPU 또는 노드가 NULL이면 해당 포인터에 아무것도 기록되지 않는다.

#include <linux/getcpu.h>  
int getcpu(unsigned \*cpu, unsigned \*node, struct getcpu\_cache \*tcache);

**14. logically attach & detach devices**

**Ioctl()**

int ioctl(int fc, int request)

ioctl()을 장치 파일에 적용시키면 장치 파일에 연결된 장치 드라이버의 파일 오퍼레이션 구조체의 ioctl 필드에 선언된 함수가 호출된다. ioctl()시스템 콜은 장치 파일 이외에는 사용할 수 없는 장치 파일 전용 시스템 콜이므로 각 장치마다 고유하게 선언하 여 사용한다.

파일 디스크립터 fd에 해당하는 디바이스 파일에 read()와 write() 시스템 콜로 처리하기 힘든 입출력 처리에 사용한다.

fd : open()시스템 콜 실행 결과로 반환된 파일 디스크립터

request – 장치 파일에 연동된 장치 드라이버에서 취해야 할 명령을 정의한다.

실행이 성공적이면 0을 반환하고 실패하면 –1을 반환한다. -1을 반환했을 때는 error값을 참 조하면 실제 장치 드라이버에서 반환한 값을 확인할 수 있다.

#include <sys/ioctl.h>  
int ioctl(int fc, int request);

**❑ Information Maintenance**

**15. get & set time & date**

**get time()**

int clock\_gettime(clockid\_t clk\_id, struct timespec \*tp)

지정된 clk\_id의 시간을 검색하고 설정한다.

clk\_id : 작동할 특정 시계의 식별자이다. 클럭은 시스템 전체에 적용될 수 있으므로 모든 프로세스에서 볼 수 있거나 단일 프로세스 내에서만 시간을 측정하는 경우 프로세스별로 볼 수 있다.

#include <time.h>  
int clock\_gettime(clockid\_t clk\_id, struct timespec \*tp);

**16. get & set system data**

**fsync()**

int fsync(int fd)

fd 와 관련된 데이터를 디스크에 쓰기 위해서는 fsync() 와 fdatasync() 함수를 사용하는 것이 좋다. fsync는 fd 로 지정된 파일과 관련된 모든 변경 자료(데이터 + 메타데이터)를 디스크와 동기화한다.

성공 시 반환 값 : 0

오류 시 반환 값 : -1

fd 는 반드시 쓰기 상태로 열려있어야 하며, sync() 함수와 달리 하드디스크가 데이터와 메타데이터를 모두 쓸 때까지 기다렸다가 반환된다.

#include <unistd.h>

int fsync (int fd);

**17. get & set process, file & device attributes**

**getpid()**

pid\_t getpid(void);

호출 프로세스의 프로세스 ID를 반환한다. 에러를 반환하지 않으므로 항상 성공적으로 실행된다. 종종 독특한 임시 파일 이름을 생성하는 루틴에 의해 사용된다.

#include <iostream>

#include <unistd.h>

using namespace std;

// Driver Code

int main()

{

    int pid = fork();

    if (pid == 0)

        cout << "\nCurrent process id of Process : "

             << getpid() << endl;

    return 0;

}

**❑ Communications**

**18. create & delete communication connection**

**getuid()**

uid\_t getuid(void);

호출할 프로세스의 유저 ID를 반환한다.

#include <unistd.h>

#include <sys/types.h>

uid\_t getuid(void);

**19. send & receive messages**

**pipe()**

pip()는 한 프로세스에서 다른 프로세스로 정보를 전달하는 데 사용되며, 단방향이기 때문에 프로세스 간 양방향 통신을 위해 각 방향으로 하나씩 두 개의 pipe()를 설정할 수 있다.

int pipe(int filedes[2]); // 성공 시 0, 실패 시 -1 반환

filedes[0]; *//-> for using read end*

filedes[1]; *//-> for using write end*

filedes에는 인덱스마다 파일 디스크립터가 저장된다.

파일 디스크립터는 파이프로부터 데이터를 수신하는데 사용된다.

부모(상위) 프로세스는 입구 또는 출구에 해당하는 파일 디스크립터 중 하나를 자식 프로세스에게 전달해야한다. -> 이는 fork()로 가능

자식은 입력 경로에만, 부모는 출력 경로에만 접근

상위 프로세스 내부 : 먼저 첫 번째 pipe(fd1[0])의 판독 끝을 닫고 pipe(fd1[1])의 끝을 쓰면서 문자열을 작성한다. 이제 부모들은 자식 과정이 끝날 때까지 기다릴 것이다. 자식 과정이 끝나면 부모는 두 번째 pipe(fd2[1])의 쓰기 끝을 닫고 pipe(fd2[0])의 읽기 끝을 통해 문자열을 읽는다.  
  
자식 프로세스 내부 : pipe(fd1[1])의 쓰기 끝을 닫아 부모 프로세스에서 보낸 첫 번째 문자열을 읽고 두 문자열을 연결한 후 fd2 pipe를 통해 부모 프로세스에 문자열을 전달하고 종료한다.

**20. transfer status information**

**connect()**

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

파일 설명자 sockfd가 참조하는 소켓을 addr이 지정한 주소에 연결한다. 서버의 주소와 포트는 addr에 지정되어 있다. 반환값이 -1 이면 접속실패를 뜻한다.

**#include <sys/types.h>   
#include <sys/socket.h>**

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

**21. attach or detach remote devices**

**uselib()**

int uselib(const char \**library*)

시스템 호출 uselib()는 호출 프로세스에서 사용할 공유 라이브러리를 로드하는 역할을 한다. 그것은 경로명이 주어진다. 로드할 주소는 도서관 자체에서 찾을 수 있다. 라이브러리는 인식된 이진 형식을 가질 수 있다.

#include <unistd.h>

int uselib(const char \**library*);