<https://www.computerhope.com/jargon/s/special-file.htm>

<https://ehclub.co.kr/1318?category=600567>

1장 베이직

커널 : 운영체제의 심장, OS를 규정짓는 매우 중요한 부분. 하드웨어의 자원을 자원이 필요한 프로세스에 나눠주고 프로세스 제어, 메모리 제어, 프로그램 시스템 콜 등 수행하는 부분으로 운영체제 맨 하부에서 돌아감.

부트 로더 : 컴퓨터가 시동되었을 때 운영 체제에 필요한 조치를 취하고 하드디스크 드라이브에 기록되어 있는 운영체제를 실행시키는 프로그램

시스템 콜 : 커널이 제공하는 서비스에 대해 응용 프로그램의 요청에 따라 커널에 접근하기 위한 인터페이스

디렉토리는 디렉토리 정보를 담고있는 파일이다. (디렉토리 엔트리는 i노드와 파일 이름을 담고있다.)

유닉스 파일시스템은 디렉토리와 파일들의 계층구조이다.

모든 것은 /라는 이름을 가진 루트 디렉토리에서부터 시작된다.

하드링크 : 파일의 아이노드를 똑같이 참조함.

심볼릭링크 : 파일의 바로가기 파일 같은 느낌…

/, null 캐릭터는 파일 이름으로 못쓴다.

새로운 디렉토리가 생성이 되면 . 과 .. 이 자동으로 생성된다.

.은 현재 디렉토리이고 ..은 부모 디렉토리이다.

모든 프로세스는 워킹 디렉토리를 가진다.

로그인하면 워킹 디렉토리가 홈 디렉토리(/etc/passwd)로 바뀐다.

유닉스는 일반 파일들 뿐만 아니라 주변 장치 및 프로세스간 통신 채널도 파일의 개념을 확장한다.

프로세스간 통신 채널 : FIFO, PIPE, SOCKET

일반 파일 : 바이너리 또는 텍스트 파일(유닉스는 차이점을 몰라용 헤헤)

디렉토리 파일 : 다른 파일들의 이름과 위치를 담고있는 파일

캐릭터 스페셜, 블록 스페셜 파일 : 터미널(캐릭터 스페셜) 그리고 디스크(블록 스페셜)

블록 스페셜 파일 : 블록 단위로 I/O를 진행하는 모든 장치에 대한 직접적인 인터페이스를 제공한다. Blockdev –getbsz /dev/sda1 명령어를 통해 1블록 크기 알 수 있음.(바이트)

캐릭터 스페셜 파일 : 블록 스페셜 파일과 비슷하지만 기준이 블록이 아닌 캐릭터이다. Stdin, stdout, stderr등이 이에 속함.

FIFO(named pipe) : 프로세스간 통신에 쓰이는 파일 타입이다.

Socket : 프로세스 사이의 네트워크 통신에 사용되는 파일 타입이다.

이외에도 소켓, 심볼릭 링크, 레귤러파일, 디렉토리, FIFO(PIPE)등이 있다.

Ownership, Permission

각 파일들은 개개인의 유저의 소유이다.

소유자는 파일과 연관된 권한을 고를 수 있다.

권한

어떻게, 그리고 파일에 누가 접근할 수 있는지 조종한다.

소유권은 특정한 권한을 부여하며, 권한이라 불리는 파일의 속성을 변화시킬 수 있다.

d : directory

l : symbolic link

b : block special file

c : character special file

p : FIFO(PIPE)

-: regular file

실행중인 프로그램의 인스턴스 : ls, ps

프로세스간 통신(IPC)

Pipe, FIFO, Signals, Shared memory, Semaphore, Sockets(네트워크를 통한 협력 프로세스에 쓰임)

Ls -l ( long form)

Ls – al (hidden file and long form)

Ls ( 파일의 목록을 보는 것)

Ps (프로세스의 목록을 보는 것)

Ps -ef (시스템 내의 모든 프로세스 목록)

시스템콜 발동 시 유저 프로세스에서 커널모드로 바뀜, 트랩에 걸린 것(인터럽트) 이후 리턴하면 user mode로 되돌아감

fread를 하면 욜로 갔다가 read를 호출함

시스템콜은 read만!

시스템 콜은 두 번의 콘텍스트 스위치를 거친다. (유저에서 커널로 갔다가 다시 돌아오는 것)

시스템 콜을 여러 번 하는 것을 피해야 성능 개선에 유리하다.

모든 시스템 콜은 헤더파일에 정의되어 있다.

------------------------------------------- 2장 파일 (1) ----------------------------

데이터를 담는 콘테이너이다.

파일 : 바이트의 연속적인 시퀀스

운영체제에서 부과하는 방식이 없다.

각각의 바이트는 디스크 파일에서 주소 지정이 가능하다.

파일은 외부장치에 대해서도 균일한 인터페이스이다.

파일 시스템

컴퓨터 파일과 데이터들을 정렬하고 조직하기 위한 방법

파일들을 찾고 접근하기 쉽게 만든다.

하드디스크, CD-ROM등과 같은 데이터 저장 장치들을 사용한다.

파일 시스템 인터페이스(커널 수준)

open, create, close, read, write, lseek, unlink, remove, fcntl <- 시스템 콜들

<fcntl.h>

/usr/include

커널 수준에서 모든 오픈 파일들은 파일 디스크립터라고 언급된다.

음수가 아닌 정수로 나타내진다.

존재하는 파일을 열거나 새롭게 생성할 때, 커널은 프로세스에게 파일 디스크립터를 리턴

유닉스 셸에 의해 생성된 각각의 프로세스들은 터미널과 관련된 세가지 오픈 파일들과 함께 시작된다.

파일 디스크립터를 0, 1, 2에 이은 lowest 값에 올려놓는다. 이후 open한 파일은 파일 디스크립터저장이 되고 이를 read하여 메모리에 값을 저장한다

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

성공하면 파일 디스크립터를 리턴하고 실패하면 -1리턴

플래그에 들어가는 것은 다음과 같다.

O\_RDONLY #0 읽기 전용

O\_WRONLY #1 쓰기 전용

O\_RDWR #2 읽고 쓰기 가능

이 중 단 하나만 플래그로 사용해야함.

왜냐하면 O\_RDONLY가 0으로 설정되어 있기 때문에 O\_RDONLY || O\_WRONLY = O\_WRONLY이기 때문…

옵션으로 넣을 수 있는 친구들은 다음과 같다.

O\_APPEND 파일의 끝에 추가하기

O\_CREAT 존재하지 않는다면 파일을 생성

O\_EXCL 파일이 이미 존재하면 에러를 내뿜는다(O\_CREAT와 함께 사용)

O\_TRUNC 파일이 존재하면 비워버린다.

O\_NONBLOCK 논 블로킹 파일을 연다.

Mode는 O\_CREAT플래그와 함께 쓰인다. 권한을 나타냄. 0000꼴로 나타낸다. 맨 앞 0은 8진수라는 뜻.

파일 퍼미션은 유저, 그룹, 그 외 순이다. 각각은 rwx순이다. read, write, execute

글로 모드를 쓰자면 S\_I가 공통적으로 붙고 그 이외엔 RUSR, WUSR, XUSR, RWXU, RGRP, WGRP, XGRP, RWXG, ROTH, WOTH, XOTH, RWXO, SUID, SGID 등이 붙는다.

mote\_t fdmode = ( S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH )면 0644를 뜻한다.

create 시스템 콜

int create(const char \*pathname, mode\_t mode);

리턴값은 파일 디스크립터, 실패 시 -1

파일을 생성하기 위한 대안책

만약 파일이 이미 존재한다면 두번째 인자는 무시된다.

open과 같진 않다. creat는 파일 디스크립터를 반환하기 전에 무조건 존재했던 파일 내용을 지워버린다.

creat는 항상 쓰기 전용으로 파일을 연다.

즉 fd = create(“/tmp/newfile”, 0644) == fd = open(“/tmp/newfile”, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644); 와 같다.

System data type 은 \_t로 끝난다.

새로운 파일을 만들면(open, create) 새로운 링크가 거기에 더해지면 부모 디렉토리에서 권한을 써줘야한다.

디렉토리는 디렉토리의 파일의 리스트를 담고있는 파일이다.

그룹 및 소유자는 프로세스의 이펙티브 유저ID로부터 설정된다.

close 시스템 콜

#include <unistd.h>

int close(int filedes);

성공하면 0, 실패하면 -1 리턴

열려있는 파일을 닫는다.

모든 열린 파일들은 자동적으로 실행이 완료되면 끝난다.

read 시스템 콜

#include <unistd.h>

ssize\_t read(int filedes, void \*buffer, size\_t n);

읽은 바이트의 숫자 리턴, 파일의 끝이면 0 리턴, 실패하면 -1 리턴

파일 카피 바이트들을 현재의 파일 포지션에서 메모리까지 읽고 파일 포지션을 업데이트 한다.

파일 디스크립터, 데이터가 복사될 공간, 읽을 바이트 수

파일을 리드하면 오프셋이 읽은 바이트수만큼 변경된다.

write 시스템 콜

#include <unistd.h>

ssize\_t write(int filedes, const void \*buffer, size\_t n);

쓴 바이트만큼 리턴, -1이면 실패

메모리부터 현재 파일 포지션까지 파일에 쓴다. 그리고 파일 포지션을 업데이트한다.

파일 디스크립터, 쓰여질 파일의 데이터, 얼마나 쓸지 바이트

만약 프로그램이 쓰기 위해 이미 존재하는 파일을 연다면 오래된 데이터는 새로운 것에 의해 덮어쓰기된다.

O\_APPEND 옵션이 open될 때 활성화되면 파일의 오프셋은 EOF로 초기화된다. 즉 라이트가 끝부터 시작되는 것.

read, write 효율성

버퍼 사이즈가 클수록 성능 향상이 된다.

최고의 성능은 버퍼사이즈가 시스템 블록 사이즈의 배수일 때이다.

natural disk block : stat’s st\_blksize

시스템 콜 호출을 줄여야하는 것도 잊지말자!

write 시스템 콜이 빨라요. 왜?

write 시스템 콜에 이슈가 생기면 쓰기를 실행하지 않고 리턴한다.

데이터를 커널에 있는 버퍼 캐쉬로만 옮기고 리턴한다.(delayed writing)

하지만, 디스크 에러가 있거나 커널이 어떤 이유로라든 멈춘다면, 게임은 끝난다. 쓴 파일이 없는 것을 확인할 수 있다.

---수업중 내용---

Ssize\_t : primptive system data type

O\_CREAT가 없으면 있으면 뜨고 없으면 오류

있으면 뜨고 없으면 만든다

파일이 있으면 fail, 파일이 없을 때만 만든다.

파일이 있으면 3이 리턴되고 없으면 -1이 리턴된다.

User, group, others

R : Readable

W : Writable

X : Executable

Owner : 본 시스템 콜을 호출한 프로세스가 됨

Chmod : 권한을 변경해주는 함수, 기호 모드와 숫자 모드 두 가지로 사용 가능

Creat system call

Close

Read (int filedes, void \*buffer, size\_t n);

File position은 read write하는 위치를 나타낸다.

Read의 리턴값 : n개 만큼 읽었따! 즉 0이면 EOF라는 뜻.

프로세스가 실행이 되면 프로세스 테이블이 만들어지고 파일디스크립터 테이블도 만들어진다. 0부터 2까지는 std in, std out, std err이 된다. 파일 디스크립터번째 인덱스는 파일 테이블을 가리키고 파일 오프셋 다음 내용은 파일을 가리킨다.

시작 위치에서 거꾸로 가는 건 안돼! 마지막 위치에서 더 뒤로 가는 건 가능해…

파일을 오픈할 때마다 파일 디스크립터가 만들어진다.

i-Node : 파일에 1:1로 대응되는 노드(즉 곧 파일이다)

파일테이블에 연결된 파일 디스크립터의 개수

#include <unistd.h>

lseek 의 l은 long integer를 뜻한다.

off\_t lseek(int filedes, off\_t offset, int start\_flag);

리턴값은 성공시엔 새로운 오프셋, 에러시엔 -1이다.

열린 파일의 오프셋은 lseek을 호출함으로써 명시적으로 정해질 수 있다.

파일 오프셋은 일반 파일에서 다음의 read와 write가 일어날 장소를 나타내는 위치이다.

filedes : 파일 디스크립터

offset : start\_flag로부터 바이트를 포함한 위치

start\_flag : SEEK\_SET : 시작, SEEK\_CUR : 현재, SEEK\_END : 파일의 끝

lseek를 이용해 포지션을 SEEK\_END로 옮긴 이후 write를 시행하면 이는 O\_APPEND 옵션과 같다.

SEEK\_END를 이용하면 lseek의 반환값을 이용해 파일 크기 알아내기도 가능

모든 프로세스는 프로세스 테이블에 엔트리를 가진다. 각 프로세스 테이블 엔트리들은 열린 파일 디스크립터의 테이블이다.

커널은 모든 열린 파일들의 파일 테이블을 유지한다.

각 파일 테이블 엔트리는 다음의 내용을 담는다.

* 파일 상태 플래그
* 현재 파일 오프셋
* 파일의 v-node 테이블 엔트리에 대한 포인터

각 파일들은 v-node 구조를 가진다. 이는 파일, 파일에서 수행되는 함수들에 대한 포인터에 대한 정보를 담고있다.

태스크(프로세스)와 아이노드의 관계.

태스크 테이블(프로세스 테이블)은 fd(file descriptor)를 가리킨다. fd[0] : stdin, fd[1] : stdout, fd[2] = stderr 등이다.

vnode/vfs는 여러 타입의 파일 시스템을 지원하기 위함이다.

여러 타입의 파일 시스템을 동시에 지원해야 한다. 유닉스 파일 시스템, 비유닉스 파일 시스템을 포함한다.

다른 디스크 파티션은 다른 타입의 파일 시스템을 가질 수 있다. 그러나 일단 디스크 파티션이 마운트되면 이들 디스크 파티션은 전형적인 하나의 동질적인 파일 시스템의 모습을 보여야한다.

등등이 있다.

vnode는 커널 내의 하나의 활성화된 파일을 나타내는 추상화다. 파일에 대한 인터페이스를 정의하고 파일에 대한 모든 연산을 파일 시스템과 관련된 적당한 함수로 전달한다.

하나의 프로세스에서 같은 파일을 두 번 열었을 때, 서로 다른 파일 디스크립터를 부여하고 서로 다른 파일 오프셋을 유지.

프로세스의 파일 디스크립터 테이블에는 각각의 파일 디스크립터를 부여하고 커널에서도 각각의 파일의 상태와 현재 작업 위치를 별도로 갖는다.

하지만 이 둘은 같은 vnode를 참조

모든 프로세스는 각각의 프로세스 테이블을 가진다.

process state, process ID, user ID, group ID, program file, File descriptor table, memory mapping, saved registers, stack pointer

각각의 프로세스 테이블 엔트리는 fd flags, file pointer로 이루어짐.

파일 테이블 : file status flags, current file offset, v-node pointer

v-node table : v-node information, i-node information, current file size

같은 파일을 실행하더라도 각각의 파일 테이블은 따로 만들어져 각각의 오프셋을 가지지만 이들이 가리키는 v-node는 같다.

dup, dup2 systemp call

#include <unistd.h>

int dup(int filedes);

int dup2(int filedes, int filedes2);

새로운 파일 디스크립터 값을 리턴, 만약 에러면 -1

Dub : 파일 디스크립터 복사, 이는 곧 무슨 뜻이냐…

dub보다는 dup2를 더 많이 사용할 것입니다.

Dup2 : filedes1의 파일 디스크립터를 filedes2로 카피, 즉 filedes2의 값은 filedes1이 된다. source, destination 순서

Filedes2 를 가리키는 포인터가 사라지고 1번을 가리키게 됨.

dup : 인수로 받은 파일 디스크립터를 복제해서 새로운 파일 디스크립터 반환

파일 디스크립터를 복제하는 것이기 때문에 오프셋 등의 기본 정보들도 그대로 딸려온다.

dup2는 filedes2가 열려있다면 이를 닫고 filedes1의 내용을 filedes2에 복제한다.

dup2(fd, STDOUT\_FILENO);를 호출한 뒤에 printf를 호출하면 stdout이 fd로 복제된 상태이기 때문에 fd에 출력이 실행된다.

**Dup2가 뭐하는 친구인지 정확하게 기억하자!!**

여기서 잠깐!

UNIX에선 프로세스에 다섯 가지 번호 부여

PID(프로세스 식별자)

RUID(실제 사용자)

EUID(유효 사용자)

RGID(실제 사용자 그룹)

EGID(유효 사용자 그룹)

사용 용도 : 계정 관리에 사용 -> RUID, RGID, 접근 권한 결정에 사용 : EUID, EGID

일반적으로 리얼 넘버와 이펙티브 넘버는 동일.

권한 관리 : chmod, chown, chgrp, umask

특수 권한

UNIX 시스템은 파일에 대한 접근 권한 및 파일 종류를 나타내기 위해 16비트를 사용한다.

9비트는 유저, 그룹, 기타 권한 기술에 사용

4비트는 파일의 종류 표현에 사용

3비트는 특수 권한에 사용

파일 종류, 특수 권한, 유저, 그룹, 기타 순

특수 권한은 setuid, setgid, sticky bit로 이루어짐.

set uid 비트를 실행 파일에 적용하면 실 사용자에서 프로그램 소유자의 ID로 EUID가 변경됨.

setuid 비트가 설정된 파일은 실행순간만 그 파일의 소유자 권한으로 실행, 실행 순간만 권한을 빌려온다고 이해하면 쉬움)

매번 슈퍼 유저 root가 어떤 행위를 해주지 않아도 되고, 일반 사용자에게 root 권한을 주지 않아도 되기 때문에 setuid 비트를 적용하는 것이 시스템 운영면에서 효율적

대부분 슈퍼 유저가 소유한 소수 프로그램들에만 주어짐, 일반 사용자가 그 프로그램을 실행하면 setuid root가 되고, 슈퍼 유저의 특권들을 가지고 실행되기 때문에 일반 사용자의 접근이 금지된 파일과 디렉토리들에 접근 가능하게끔 해줌.

setgid : setuid처럼 EGID를 사용자의 실제 그룹 ID에서 파일 소유자의 그룹 ID로 변경함

setgid 비트가 디렉토리에 설정되어 있으면, 이 디렉토리에 새로 설정된 파일들은 디렉토리 그룹 소유권보다 파일 생성자의 그룹 소유권을 얻게 될 것.

일반 파일 그룹의 멤버가 파일 소유자의 그룹과 상관없이 디렉토리 내의 모든 파일에 접근이 필요한 공유 디렉토리에 유용

디렉토리 EGID 가 root면 다른 계정이 만들어도 root 소유로 뜬다.

sticky bit

8진수 1

리눅스는 파일의 스티키 비트는 무시

sticky bit는 특정 디렉토리를 누구나 자유롭게 사용할 수 있게 하기 위함.

sticky 비트가 디렉토리에 적용되면 디렉토리 소유자나 파일 소유자, 슈퍼유저가 아닌 사용자들은 파일을 삭제하거나 이름을 변경할 순 없음. 파일 또는 디렉토리 생성은 누구나 가능!

fcntl system call

#include <fcntl.h>

int fcntl(int file des, int cmd, …);

리턴값은 cmd에 따라 다르다. 실패 시 -1을 출력

열려진 파일의 속성을 가져오거나 설정할 때 fcntl 함수를 사용한다.

filedes : 파일 디스크립터

cmd : 프로그래머가 정수 cmd 패러미터를 설정함으로써 특정 기능을 선택할 수 있다.

F\_DUPFD : 존재하는 디스크립터를 복제

F\_GETFD or F\_SETFD : 파일 디스크립터 플래그를 Get/Set

F\_GETFL or F\_SETFL : 파일 스테이터스 플래그를 Get/Set

F\_GETOWN or F\_SETOWN : SIGIO와 SIGUSR 시그널을 수신하는 프로세스 ID 혹은 프로세스 그룹 ID를 Get/Set

F\_GETLK, F\_SETLK, or F\_SETLKW : 레코드 lock을 Get/Set

Redirection

입출력 재지정 : 입출력 대상으로 표준 입력, 표준 출력, 표준 오류를 사용하지 않고 다른 경로인 파일로 재지정하는 것.

꺾쇠(<, >) 를 이용하거나 dup2를 이용.

Standard I/O Library

Unix I/O (system call)

바이트의 간단한 시퀀스 형태의 데이터들만 다룬다.

프로그래머가 대부분을 담당해야한다.

효율성도 개발자의 영역이다.

Standard I/O

자동 버퍼링, 좀 더 프로그래머 친화적인 인터페이스, 라이브러리를 더 쉽게 쓰게한다. 효율성에 있어 프로그래머를 좀 더 편하게 만들어준다.

standard I/O와 UNIX I/O의 차이 : 파일을 기술하는 방법

FILE \* or file descripter

파일 열고 닫기 fopen, fclose

바이트 단위 읽고 쓰기 fread, fwrite

텍스트 라인 단위 읽고 쓰기 fgets, fputs

포맷 형식 읽고 쓰기 fscanf, fprintf

fopen

#include <stdio.h>

FILE \*fopen(const char \*restrict pathname, const char \*restrict type);

r, w, a : 읽기 열기, 첨부터 쓰기 열기, append 쓰기 열기

r+, w+, a+: + 붙으면 read, write 둘다 됨.

#include <stdio.h>

int getc(FILE \*istream); 다음 캐릭터를 리턴, EOF 파일의 끝일 때 리턴

int putc(int c, FILE \*ostream); c 리턴

버퍼링

스탠다드 I/O는 우아한 버퍼링 메커니즘으로 이 비효율을 피해간다.

버퍼는 맨 첨 I/O가 수행되는 때에 보통 malloc이라 불리는 함수 내에 포함된다.

b 붙으면 바이너리

Fopen : system call이 아님다

Restrict가 있는 변수끼리는 메모리를 공유하면 안된다.

시스템 콜이 호출될 때,

에러면 -1 리턴, 에러 없으면 0 리턴.

에러 코드는 errno에 선언이 된다.

각 상수들은 모두 E로 시작한다.

에러넘버는 어떤 시스템 콜이 성공해도 초기화되지 않는다.

어떤 시스템콜을 성공하고 에러넘버를 보는 건 아무 의미가 없다.

실패 하자마자 즉각적으로 에러넘버를 봐야 의미가 있다.

수퍼유저 맘대로

유닉스의 모든 파일은 시스템 유저에 의해 소유된다.

소유자는 보통 파일을 만든 계정이다.

소유자의 실제 아이덴티티 : user-id (uid)

uid는 특별한 유저네임과 연관되어있다.

각 유닉스 프로세스는 보통 프로세스를 시작한 유저의 uid와 연관되어 있다.

파일이 만들어질 때, 시스템은 프로세스 생성의 uid를 언급하는 것으로 소유권을 만든다.

루트 유저는 uid가 0이다

프로세스의 아이디는 리얼 유저와 이펙티브 유저 아이디가 있다.

대부분의 경우 리얼과 이펙트는 같다. 하지만 아주 특별한 경우 다르다.

ruid : 프로세스를 실행시킨 유저의 uid

euid : 특정한 액션을 수행하기 위한 과정의 권한을 평가하는데 쓰임

대부분 euid, ruid는 같다.

만약 setuid 비트를 적용하면? euid가 변하는 것!

소유권 : 파일과 연관된 권한을 고를 수 있다.

권한 : 다른 유저들이 파일에 액세스 할 수 있는지 결정한다.

슈퍼 유저는 모든 파일에 접근하거나 읽고 쓸 수 있다.

open함수가 이미 존재하는 파일을 열 때 : 시스템은 파일 권한을 체크한다

권한이 없으면 -1을 리턴, 우리가 파일을 열 때 euid와 egid에 기반을 두고 체크함.

이펙티브 유저 아이디가 실제론 더 영향을 많이 거의 끼침

S\_ISUID, S\_ISGID, S\_ISVTX

유저 이름 같고 퍼미션 있으면 열 수 있다.

유저 이름 다르면서 그룹 같고 그룹 퍼미션 있으면 열 수 있다.

유저 이름 그룹 이름 다 다르고 other퍼미션 있으면 열 수 있다.

스티키 비트는 시스템 종료시까지 시스템 스왑 영역에 남아있는다.

파일 생성 마스크

umask (뺄셈으로 생각하면 됨. 즉 000이 모든 권한)

#include <sys/stat.h>

mode\_t umask(mode\_t cmask);

리턴값은 이전 파일 모드 생성 마스크

umask 함수는 파일 모드 생성 마스크를 설정한다.

#include <unistd.h>

int access (const char \*pathname, int amode);

성공하면 0, 실패하면 -1 리턴

access는 real user 그리고 그룹 id에 기반한 경로의 접근 권한을 체크한다.

R\_OK, W\_OK, X\_OK, F\_OK : 파일 유무 테스트

#include <unistd.h>

int chmod(const char \*pathname, mode\_t newmode);

#include <unistd.h>

int chown(const char \*pathname, uid\_t owner\_id, gid\_t group\_id);

파일의 uid나 gid를 바꿀 수 있도록 해준다.

owner\_id : new owner

group\_id : new group

파일 시스템 : 저장 장치에 있는 파일들을 저장하기 위한 논리적 구조를 빌드하는 소프트웨어 컴포넌트.

마운트는 루트 트리에 덧붙인다고 생각하면 된다.

Inode 디스크 구조체에 속성 저장..

부트 블록, 슈퍼 블록, i노드, 데이터 블록

부트 블록 : 부트 코드가 들어가는 블록

슈퍼 블록 : 파일 시스템 자체의 속성과 메타데이터를 가지고 있음. 총 블록 수, inode 수, 비트맵 프리 블록 수, 바이트 블록 수, 자유 블록 수, 사용되는 블록 수 등…

inode : 모든 아이노드는 디스크에 있는 파일들과 관련되어 있다. 특별하게 파일을 분간한다.

데이터 블록 : 파일 블록들을 저장하기 위함

inode : 각각의 파일들(regular, directory, special …)은 하나의 i-node를 사용하고 무조건 최소한 하나의 디렉토리와 링크되어있어야한다.

0, 1 아이노드는 안 쓰인다. 0은 no inode를 뜻하고 1은 배드 디스크 블록을 수집하는데 쓰인다.

root가 아이노드2다.

하드링크 : 파일을 가리키는 직접 포인터이다.

링크카운트는 inode를 가리키는 디렉토리 엔트리의 개수이다.

링크가 0이 되면 삭제된다.

이 링크의 타입은 하드 링크이다.

superuser가 하드링크를 디렉토리에 만들 수 있다.

심볼릭 링크 : 파일에 대한 간접 포인터이다.

심볼릭 링크의 실제 내용은 링크된 파일이다.

파일 시스템 구조간의 한계는 없다.

#include <unistd.h>

int link(const char \*original\_path, const char \*new\_path);

hardlink 추가한다. 원본 파일, 새로운 이름

오

#include <unistd.h>

int unlink(const char \*pathname);

디렉토리 엔트리에서 삭제한다. 링크를 지우고 카운트를 하나 낮춘다.

rename 시스템 콜

#include <stdio.h>

int rename(const char \*oldname, const char \*newname);

이름 바꾸는 함수

#include <unistd.h>

int symlink (const char \*realname, const char \*symname);

심볼릭 링크 만드는 함수

#include <unistd.h>

ssize\_t readlink (const char \* sympath, char\* buffer, size\_t bufsize);

리드 링크는 다음과 같다.

sympath를 연다.

파일을 읽어 버퍼로 옮긴다.

sympath를 닫는다.

open으로 열 수는 없지만 심볼릭 링크는 readlink로 읽을 수 있다.

#include <sys/stat.h>

int stat(const char \*pathname, struct stat \*buf);

int fstat(int filedes, struct stat \*buf);

int lstat(const char \*pathname, struct stat \*buf);

stat : namedfile, fstat : open file, lstat : symbolic link

stat은 다음과 같다.

st\_mode, st\_ino… dev, rdev, nlinnk, uid, gid, size, atime, mtime, ctime, blksize, blocks

#include <stdio.h>

int remove(const char \*pathname);

unlink와 거의 같다고 보면 된다.

다른 사람의 파일 권한을 바꿀 순 없어.

스탯 아주 중요한 시스템 콜

파일과 1:1 대응이 되는 아이노드 자료구조

해당 정보를 가져오는 시스템 콜임.

디렉토리 : open이나 creat를 이용하여 만들 순 없음.

디렉토리는 inode 넘버와 캐릭터 필드로 이루어져 있다.

각 디렉토리엔 항상 .과 .. 디렉토리가 존재한다.

디렉토리 read 퍼미션은 ls를 할 수 있는거

Write 퍼미션은 파일 만들거나 지울 수 있는거

Exe 퍼미션은 (search 퍼미션)이라고도 한다. 이게 있어야 cd를 할 수 있다. 어떤 파일을 오픈하려고 하면 이 파일이 위치한 디렉토리에 퍼미션이 있어야 오픈할 수 있다.

디렉토리 퍼미션

read : ls 할 수 있는가

write : 디렉토리에 새 파일을 만들거나 삭제할 수 있는가

execute : cd를 통해 바꿀 수 있는가

스티키비트 : 셋 중 하나만 삭제할 수 있다. 파일의 소유자, 디렉토리의 소유자, 슈퍼유저

#include <dirent.h>

struct dirent{

ino\_t d\_ino;

char d\_name[NAME\_MAX + 1];

}

inode 0번은 빈 inode라는 것을 명심.

#include <sys/stat.h>

int mkdir(const char \* pathname, mode\_t mode);

mkdir은 . 과 .. 을 자동으로 넣어준다. 또한 실행 권한 비트를 넣어줘야 한다는 것을 명심!

#include <unistd.h>

int rmdir(const char \*pathname);

#include <dirent.h>

DIR \*opendir(const char \*dirname);

dir 여는 함수

#include <dirent.h>

int closedir(DIR \*dirptr);

#include <dirent.h>

struct dirent \*readdir (DIR \*dp);

int chdir (const char \*path);

Getcwd 리턴값이 문자열로 나온다.

getcwd : name에 현재 디렉토리를 넣어서 반환한다.

#include <unistd.h>

int execl, execv, execle, execve, execlp, execvpd

e : 현재 환경 대신 envp 쓴다는 뜻

fork

#include <unistd.h>

pid\_t fork(void);

자식일땐 0, 부모일땐 자식의 pid, 에러일땐 -1 리턴

Sync, fsync : 버퍼에 남아있는 친구들을 강제로 디스크에 플러쉬 해주는 친구

메이저넘버 : 디바이스 타입, 같은 드라이버 타입

마이너 넘버 : 여러 개 있으면 그 중에 몇 번째냐?

매크로란? Function 비슷한데 function은 아니고 컴파일 할 때 실제 코드로 확장되는 친구

St\_mode에 따라서 첫 글자가 다르다! Ex) -, d, c, b, p, s, 등등..

프로세스

Getpid : 프로세스 ID 리턴

Getppid : 부모 프로세스 id 리턴

부모는 포크를 실행하고 리턴하지만 차일드는 포크를 실행하지 않고 리턴한다.

Parent fork 리턴값 : 자식의 pid 값 리턴

Child fork 리턴값 : 0 리턴

자식과 부모가 실행하는 부분이 겹치면 골치하프다…

Parent와 child가 실행하는 부분은 겹치지 않도록 강력히 권고하는 바이다도시

Pathname : 경로, finlename : 프로그램 이름

P가 있으면 PATH 환경변수에서 찾는다.

정상 종료면 exit(0)를 넣는다. 1이나 2면 something wrong…

exit 핸들러 : 유저가 클린업 액션을 직접 짜줘서 등록을 해주는 것이다. 대충 atexit이다.

wait = hang = block

기말고사

프로세스ID : 0

swapper, 유저 모드가 아닌 커널에서 페이징을 담당하는 시스템 프로세스이다.

프로세스 ID : 1

Init process, 커널에 의해 부트스트랩 과정의 끝에 호출되는 프로세스

init process는 절대 죽지 않는다.

이는 노멀 유저 프로세스이다.

init 은 고아원 원장이다 , 고아 프로세스들 죄다 이쪽으로

프로세스 그룹은 보통 같은 일과 연관된 하나 이상의 프로세스들의 모음

각 프로세스 그룹은 고유한 pgid가 있다.

pgid는 pid와 비슷하다.

각 프로세스 그룹은 그룹 리더를 가진다. (pid가 pgid이면 그룹 리더이다)

#include <unistd.h>

pid\_t getpgrp (void); 호출한 프로세스의 pgid 반환

pid\_t getpgid(pid\_t pid); pgid반환, 에러면 -1 반환 ( getpgid(0) == getpgrp() )

프로세스 그룹을 바꾸려면

#include <unistd.h>

int setpgid (pid\_t pid, pid\_t pgid);

pid 프로세스의 pgid를 set하는 함수이다.

pid == pgid : 그룹 리더

pid == 0 : 호출자의 pid는 사용됨

pgid == 0 : pid에 의해 분류된 pid는 pgid로 사용됨

자신이나 자식의 pgid만 set할 수 있음.

자식이 exec 함수를 실행한 경우엔 pgid 변경 불가.

Session : 하나 이상의 프로세스 그룹들의 모음

세션에 포함된 프로세스 그룹은 단일 포그라운드 프로세스 그룹과 하나 이상의 백그라운드 프로세스 그룹으로 나눌 수 있다.

#include <unistd.h>

pid\_t getsid(pit\_t pid);

Return : 세션 리더의 pgid, or -1

pid가 0이면 호출한 프로세스의 세션 리더의 pgid를 반환한다.

데몬 프로세스 : 컨트롤 가능한 터미널이 없는 백그라운드 프로세스

#include <unistd.h>

pid\_t setsid(void);

Return : pgid, or -1

만약 호출한 프로세스가 프로세스 그룹 리더가 아니면, 새로운 세션을 생성한다.

세 가지 일이 일어난다!

1. 새로운 세션의 세션 리더가 됨.
2. 새로운 프로세스 그룹의 그룹 리더가 된다.
3. 프로세스는 컨트롤링 터미널이 없다. 만약 setsid 호출 이전에 가지고 있었다면 이는 사라진다.

current working & root directory

cwd는 프로세스를 시작한 fork 또는 exec 에 상속된다.

프로세스별 속성이다.

만약 자식 프로세스가 chdir을 호출하여 위치를 변경하면 부모 프로세스의 현재 작업 디렉토리는 변경되지 않는다.

각 프로세스는 절대 경로 이름 검색에 사용되는 루트 디렉토리와도 연관된다.

파일 시스템 계층의 시작점이다.

#include <unistd.h>

int chroot (const char \*path);

chroot가 성공하면 path는 루트로 시작하는 파일 검색의 시작점이 된다.

호출 프로세스가 하나면 시스템은 한동안 영향을 받지 않는다.

현재 날짜 시스템 개념 변경, 특정 파일 RW 권한제어는 uid, gid를 기반으로 한다.

#include <unistd.h>

int setuid(uid\_t uid);

int setgid(gid\_t gid);

ulmit

#include <ulimit.h>

long ulimit(int cmd, [long newlimit]);

각 프로세스별로 write 시스템 콜에 의해 생성될 수 있는 파일 사이즈의 한계가 있다.

cmd parameter : UL\_GETFSIZE, UL\_SETFSIZE, 슈퍼유저가 파일 사이즈 한계를 늘릴 수 있다. euid는 한계를 줄이는 것이 허용된다.

nice값을 기준으로 프로세스가 부분적으로 할당된 cpu시간의 비율을 결정한다.

0 ~ 최댓값, 숫자가 클수록 프로세스 우선 순위가 낮아진다. 슈퍼유저 프로세스는 음수값을 nice 매개변수로 사용하여 우선순위를 높일 수 있다.

시그널은 소프트웨어 인터럽트이다. 시그널은 비동기 이벤트를 처리하는 방법을 제공한다.

키보드 입력을 담당하는 커널은 인터럽트 문자를 확인.

SIGINT라는 시그널을 모든 프로세스에 전송 (포그라운드 그룹)

cc가 시그널을 수신하면 SIGINT와 관련된 기본 동작을 수행하고 종료한다.

시그널은 소프트웨어 인터럽트이다. 비동기 이벤트를 처리하는 방법을 제공한다.

프로세스는 단순히 변수를 테스트해서 시그널이 발생했는지 알 수 없다.

대신 프로세스는 커널에 시그널 핸들러를 지정해야한다.

모든 시그널은 이름을 가진다. 모두 SIG로 시작. <signal.h>에 선언되어 있다.

수많은 조건이 시그널을 생성할 수 있다.

Ctrl + C : SIGINT, divide by 0, kill, SIGURG, SIGPIPE, SIGALRM… 등등

1번부터 63번까지 아~~주많다.

시험에 나왔던 것 : SIGCONT, SIGSTOP, SIGKILL, SIGQUIT, SIGPIPE, SIGINT, SIGALRM

SIGCONT : 정지했던 프로세스가 다시 재개될 때 해당 프로세스에 보내는 시그널

디폴트 액션은 프로세스 재개나 무시

SIGSTP : Ctrl + z 에 의해서 발생, 디폴트 액션은 SIGCONT를 받을 때 까지 프로세스를 중단한다.

SIGSTOP : SIGSTP와 비슷하지만 변경할 수 없다.

SIGKILL : 프로세스를 죽임, 디폴트 액션은 종료. 변경할 수 없다.

SIGQUIT : Ctrl + \에 의해 발생

SIGPIPE : 파이프 I/O시 발생하는 시그널, 디폴트 액션은 종료. 만약 수신 프로세스가 종료했을 때 송신 프로세스가 write 하면 발생.

SIGALRM : 시스템 콜로 설정한 알람 시간이 초과했을 때 프로세스로 전달.

SIGCHLD : 자식 프로세스가 멈추거나 종료되면 부모 프로세스에게 전달된다.

시그널은 이벤트의 프로세스에 대한 소프트웨어 알림이다.

시그널은 시그널을 유발하는 이벤트 발생 시 생성된다.

시그널은 해당 시그널을 받은 프로세스가 액션을 취할 때 전달된다.

시그널 발생에서 전달까지의 기간동안 살아있다.

시그널 발생 이후 전달 이전까지는 pending

시그널을 받은 프로세스가 시그널 핸들러를 이용하여 액션을 취할 경우 : signal catch

시`그널 핸들링

프로세스가 시그널을 받으면 반드시 3가지중 하나를 취해야 한다.

Ignore, Signal handler, Default action

SIGKILL, SIGSTOP은 무시하거나 변경할 수 없다.

시그널이 발생해서 catch되면 프로세스의 실행이 시그널 핸들러에 의해 인터럽트 된다.

프로세스는 이후 진행되지만 시그널 핸들러의 명령어가 실행된다.

만약 시그널핸들러가 리턴하면 시그널이 잡혔던 지점에서 정상적으로 계속 실행된다.

하지만 시그널 핸들러에서는 어디서 시그널이 잡혔는지 모른다.

시그널이 생성되었을 때 취해지는 액션은 프로세스의 시그널 마스크와 시그널 핸들러에 따라 다르다.

시그널 마스크는 차단될 시그널 목록을 포함한다.

프로그램은 sigprocmask를 이용하여 프로세스 시그널 마스크를 바꿔 시그널을 차단한다.

프로세스는 fork, exec 이후 시그널 마스크를 상속한다.

#include <signal.h>

void (\*signal(int signo, void (\*func)(int)))(int);

시그널의 의미가 구현에 따라 다르기 때문에 sigaction을 사용하는 것이 더 좋다.

하위호환이 되지만 새로운 응용 프로그램은 안 쓰는 것이 좋다.

signo : SIGNAL 이름

\*func : SIG\_IGN, SIG\_DFL, signal handler 등

signal 함수는 2 매개변수가 필요하고 void형 포인터를 리턴한다

프로세스 시그널 마스크 : 차단될 시그널 목록.

시그널 핸들러 진입 전 시그널 마스크에 추가된다.

exec일 경우 시그널은 모두 초기화된다.(아닌 것 같은데… exec해도 ignore했던 친구들 살아있음)

fork일 경우 부모의 시그널 핸들러를 상속한다.

signal set을 선언해서 한꺼번에 다룰 수 있도록 한다.

signal sets 에서 empty와 fillset은 매크로로 되어있다.

sigaction 구조체는 4개의 멤버를 가진다.

handler, sa\_mask, sa\_flags, sigaction

sigprocmask

sigaction 함수를 사용하면 특정 시그널과 관련된 액션을 검사 혹은 수정할 수 있다.

#include <signal.h>

int sigaction(int signo, const struct sigaction \*restrice act, struct sigaction \*restrice oact);

signo : SIGNAL number

act : 우리가 지정할 핸들러

oact : 이전 액션

struct sigaction {

void (\*sa\_handler) (int);

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_sigaction) (int, siginfo\_t \*, void \*);

};

sa\_handler : 시그널 캐치 함수의 주소 (시그널 핸들러), SIG\_IGN, SIG\_DFL 상수와 반대

sa\_mask : 시그널 핸들러가 호출되기 전 프로세스 시그널 마스크에 추가될 마스크

sa\_flags : 시그널 핸들러를 위한 옵션

일반적으로 시그널 핸들러는 void handler(int signo)로 호출된다.

하지만 SA\_SIGINFO가 포함되면 void handler(int signo, siginfo\_t \*info, void \*context);로 호출된다.

시스템콜을 실행중일 때 시그널이 전달되면 이게 완료될 때까지 시그널은 작동하지 않는다.

느린 시스템 콜에서 프로세스가 블록되어 있으면 시그널에 의해 인터럽트 당한다.

느린 시스템 콜 : pipe, terminal devices, network devices, pause, wait, ioctl, ipc 등등

나머지는 빠르다.

시그널 호출 시 시스템 콜 재시작 목록 : ioctl, read, readv, write, writev, wait, waitpid 등등…

#include <setjmp.h>

int sigsetjmp(sigjmp\_buf env, int savemask);

void siglongjmp(sigjmp\_buf env, int val);

시그널 블로킹

#include <signal.h>

int sigprocmask(int how, const sigset\_t \*restrict set, sigset\_t \*restrict oset);

프로세스의 시그널 마스크는 해당 프로세스로 전달이 차단되는 시그널 세트이다.

만약 set이 NULL이면 프로세스 시그널 마스크는 안바뀌고 how는 무시된다.

how의 종류

SIG\_BLOCK : 블록될 시그널 추가

SIG\_UNBLOCK : 블록된 시그널 모음에서 찾아 삭제

SIG\_SETMASK : set을 전부 블록

#include <signal.h>

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

int raise(int signo);

kill 함수는 프로세스 또는 프로세스 그룹에 시그널을 보낸다

raise함수는 자신에게 신호를 보낼 수 있게 허용한다.

슈퍼유저는 어느 프로세스에게나 시그널을 보낼 수 있다.

발신자의 RUID EUID는 수신자의 RUID EUID와 같아야한다.

raise(signo); == kill(getpid(), signo);

void handler (int);

static struct sigaction act;

act.sa\_handler = handler;

sigemptyset(&act.sa\_mask);

sigaction(SIGINT, &act, NULL);

signal number 0 은 null signal이다.

종종 프로세스가 여전히 존재하는지 판별하는 데 사용된다.

#include <unistd.h>

unsigned int alarm (unsigned int seconds);

이전에 등록된 알람이 만료되지 않은 경우 해당 알람의 남은 시간(초)이 반환됨.

알람은 새롭게 업데이트.

PIPE : 가장 오래된 유닉스 시스템 IPC의 형태이다.

가장 간단한 프로세스간 통신 메커니즘. 특별한 파일의 형태로 표현된다.

#include <unistd.h>

int pipe(int filedes[2]);

0 if ok, -1 on error : return

filedes[0] : read

filedes[1] : write

pipe는 slow system call

두가지 제한 사항이 있다. : Half duplex, 부모와 자식 사이에서만 쓰임

프로세스가 파이프에서 읽으면?

파이프에 뭔가 있다면 바로 읽는다, 없으면 생길 때까지 기다린다.

프로세스가 파이프에 쓰면?

파이프가 다 안 찼으면 즉시 반환한다. 다 찼으면 빌 때까지 기다린다.

파이프 어느 한 쪽이 닫히면?

쓰기(1번)가 닫혔을 경우 : read는 0, 즉 EOF를 반환한다.

읽기(0번)가 닫혔을 경우 : SIGPIPE가 생성된다. 시그널을 핸들링하면 errno -1을 반환한다.

fast system call : 끝날때까지 기다림

slow면 인터럽트 걸림.

write가 닫히면 read는 곧바로 0을 리턴하고 곧바로 리턴됨.

read가 닫히면 SIGPIPE를 보냄.

두 프로세스가 동시에 파이프에서 자유롭게 읽고 쓰면 혼란이 발생할 수 있다. 따라서 필요하지 않은 파일 디스크립터를 닫는 것이 일반적이다.

블로킹/논블로킹

호출자를 영원히 블록할 수 있는 시스템콜

데이터가 특정 파일 형식으로 존재하지 않는 경우에 블록을 읽을 때.(파이프, 터미널, 네트워크 등)

파일 타입들에 의해 데이터가 즉시 수용할 수 없는 경우에 블록에 쓸 때 (파이프 꽉참, 네트워크 플로우 컨트롤 등)

FIFO 같이 뭔가 특정 조건이 발생했을 때

주어진 디스크립터를 논블로킹 할 수 있는 두가지 방법

open할 때 O\_NONBLOCK

이미 열렸으면 fcntl을 이용해서 O\_NONBLOCK flag 설정

read, write 논 블로킹

fstat:stat.st\_size

fstat(p[1], &buf);

if(buf.st\_size >= PIPE\_BUF) return error;

if(fcntl(p[1], F\_SETFL, O\_NONBLOCK) == -1)

perror(“fcntl”);

fcntl(p[1], F\_SETFL, O\_NONBLOCK);

파이프가 꽉 찼다면 write 요청이 블록되지 않고 -1을 반환함.

fcntl(p[0], F\_SETFL, O\_NONBLOCK);

파이프가 비어있다면 read 요청이 블록되지 않고 -1을 반환함.

자식이 쓰면 부모는 읽는다.

자식은 read 닫고 부모는 write 닫는다. 근데 만약 안 닫는다? 그러면 read, write 영원히 기다릴 수도 있음. deadlock…

그래서 닫아줌!

popen, pclose

#include <stdio.h>

FILE \*popen(const char \*cmdstring, const char \*type);

int pclose(FILE \*fp);

popen : 파이프 생성, 자식 포크, 안 쓰는 파이프 닫기, 명령 실행 위해 쉘 실행

pclose : 명령 종료 기다림

type이 r이면 파일 포인터는 표준 출력에 연결됨.

type이 w면 파일 포인터는 표준 입력에 연결됨.

FIFO

네임드 파이프라고도 불림.

파이프는 공통 조상이 파이프를 생성했을 경우 관련된 프로세스간에만 파이프를 이용할 수 있다.

하지만 FIFO는 관련없는 프로세스들 끼리도 데이터를 교환할 수 있다.

파이프 : 공통 조상이 파이프를 생성했을 경우에만 관련된 프로세스들끼리 파이프를 이용할 수 있다. 하지만 FIFO는 관련 없는 프로세스들 끼리도 데이터 교환이 가능하다.

FIFO에는 소유자, 크기 및 관련 액세스 권한이 있다.

다른 유닉스 파일처럼 열고 닫고 삭제할 수 있다.

읽기 전용, 또는 쓰기 전용으로 열어야한다.

FIFO는 Half Duplex라서 read/write로 열면 안된다.

파이프나 FIFO에 쓰기는 항상 데이터를 추가하고 읽기는 항상 파이프나 FIFO의 시작 위치를 반환한다.

파이프나 FIFO에 대해 lseek가 호출되면 error ESPIPE가 반환된다.

#include <sys/stat.h>

int mkfifo(const char \*pathname, mode\_t mode);

이름, 권한 마스크

논블로킹 플래그(O\_NONBLOCK)

플래그가 설정되지 않은 일반적인 케이스 :

읽기 전용으로 열기 : 다른 프로세스가 쓰기 위해 FIFO를 열 때까지 차단함.

쓰기 전용으로 열기 : 다른 프로세스가 읽기 위해 FIFO를 열 때까지 차단함.

O\_NONBLOCK이 설정된 케이스 :

읽기 전용으로 열기 : 즉시 파일 디스크립터 반환

쓰기 전용으로 열기 : 읽기 위해 FIFO를 연 프로세스가 없다면 에러

I/O 멀티플렉싱

멀티플렉싱이란 ? 하나의 통신 채널을 통해 둘 이상의 데이터를 전송하는데 사용되는 기술

물리적 장치의 효율성을 높이기 위해 최소한의 물리적인 요소만 사용해서 최대한의 데이터를 전달하기 위해 사용되는 기술

하나의 디스크립터에서 읽고 다른거에서 쓸 때, 우린 루프를 통해 블록을 걸 수 있다.

while((n = read(STDIN\_FILENO, buf, BUFSIZ)) > 0)

if( write(STDOUT\_FILENO, buf, n) != n)

err\_sys(“write error”);

두개의 디스크립터로부터 읽으려면 어떻게 해야하는가?

논블로킹 I/O모델(폴링)

CPU타임을 낭비함

멀티태스킹 시스템에서는 피해야한다.

멀티플렉싱 I/O 모델

select() or poll()

Signal-Driven I/O Model (event), Asynchronous I/O Model(event)

SIGIO

#include <sys/time.h>

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

리턴 : 준비된 디스크립터의 수, 0이면 타임아웃, -1이면 에러

커널에게 알려줌.

nfds : 우리가 관심있는 디스크립터, 만약 3, 4가 열려있다면 이는 5에 와야한다.

readfds : 주어진 디스크립터로부터 읽고 싶나?

write 동일

timeout : 얼마나 기다릴지 정해줌

fd 중 하나가 입출력이 준비될 때를 알려준다.

하나 이상의 fd가 준비될 때까지 잠든다.

어떤 fd가 준비되었는지 확인 후 준비가 되면 깨어난다.

블록 없이 모든 fd가 입출력을 준비하도록 관리한다.

1단계로 돌아가서 다시 시작한다.

타임아웃이 Null이면 : 영원히 기다림. fd중 하나가 준비되거나 시그널이 잡힐때까지 차단.

타임아웃이 0이면 : 안 기다린다. 차단 없이 fd 상ㄷ태만 확인할경우.

지정된 시간만큼 기다린다. fd가 준비되거나 시간이 만료되면 리턴한다.

select 리턴값 -1 : 디스크립터 셋은 수정되지 않음

0 : 디스크립터가 준비되지 않음. 모든 디스크립터 셋이 제로화됨.

리턴값 양수 : 준비된 디스크립터 수, read, write, except 3개의 셋에서 준비된 fd 개수 모두 합한 것. 그래서 read, write에 똑 같은 디스크립터가 준뵈되면 두번 계산됨.

세 개의 디스크립터 셋에 남은 비트는 준비된 디스크립터에 해당하는 비트

레코드락킹 생략!!

쉐어드 메모리, 메시지 큐, 세마포어 등 IPC 메커니즘

IPC 오브젝트가 생성되면 IPC 기능상태구조 또한 생성된다. 이는 euid, egid, ruid, rgid, mode 등을 포함한다.

umask 값은 적용되지 않는다.

uid, gid, mode를 수정 가능하다. msgctl, shmctl, semctl 등을 통해서(오직 작성자와 슈퍼유저만)

Key

IPC 객체의 외부 이름.

msgget, semget, shmget을 통해 IPC 구조가 생성되면 무적권 키를 지정해야한다.

key\_t <sys/types.h>.long integer

Identifier

IPC 객체의 내부 이름

음수가 아닌 정수이다.

get operation의 결과이다.

파일 디스크립터처럼 작동한다.

ftok

#include <sys/ipc.h>

key\_t ftok(const char \*path, int id);

경로와 ID를 key\_t (IPC 키 값)으로 변환한다.

path : 기존 파일

id : 하위 8비트의 id만 사용

path와 id의 조합은 IPC 객체를 고유하게 식별한다.

if((thekey = ftok(“tmp/trouble.c”, 1)) == (key\_t)-1)

perror(“Failed to derive key from /tmp/trouble.c”);

만약 path가 존재하지 않거나 프로세스 호출에 액세스 할 수 없다면 -1을 반환한다.

#include <sys/msg.h>

#include <sys/sem.h>

#include <sys/shm.h>

int msgget(key\_t key, int permflags);

int semget(key\_t key, int nsems, int permflags);

int shmget(key\_t key, size\_t size, int permflags);

IPC key값을 갖는 IPC 객체 생성 혹은 열기

키 설정법 세가지 :

시스팀이 키를 선택하도록 함. (IPC\_PRIVATE)

키를 직접 선태갛ㅁ

ftok를 통해 key를 생성.

permflags

IPC\_CREAT : O\_CREAT

IPC\_EXCL : O\_EXCL

Ctl Operatinos

#include <sys/msg.h>

#include <sys/sem.h>

#include <sys/shm.h>

int msgctl (int msgid, int command, struct msgid\_ds \*buf);

int semctl (int semid, int semnum, int command, …);

int shmctl (int shmid, int command, struct shmid\_ds \*buf);

커맨드 :

IPC\_STAT : IPC 리소스 상태 정보를 리턴한다.

IPC\_SET : 소유자, 그룹, 또는 mode를 수정한다.

IPC\_RMID : IPC 리소스 내용들을 삭제하고 시스템에서 삭제한다.

메시지 큐

파이프의 확장된 버전, 링크드리스트로 구성됨

#include <sys/msg.h>

int msgget (key\_t key, int flag);

키 매개변수와 연관된 메시지 큐 ID를 리턴한다.

flag에 들어가는 PERMS는 매크로로 정의되어 있다.

#define PERMS (S\_IRUSR | S\_IWUSR)

#include <sys/msg.h>

int msgsnd (int msqid, const void \*ptr, size\_t nbytes, int flag);

msgsnd를 사용하여 메시지를 큐에 삽입하는 프로그램

메시지는 항상 큐의 끝에 배치된다. (링크드 리스트)

ptr : 사용자 정의 버퍼를 가리킴

struct mymesg {

long mtype;

char mtext[1];

};

nbytes : mtext size

flag : 만약 IPC\_NOWAIT을 지정하지 않으면 메시지를 위한 공간이 생길때까지 블록된다.

#include <sys/msg.h>

ssize\_t msgrcv (int msqid, void \*ptr, size\_t nbytes, long type, int flag);

Argument

ptr : 사용자 정의 버퍼를 가리킴

type이 0 이면 : 큐에서 메시지를 첫번째 메시지를 읽어옴

양수면 : 타입으로 지정한 유형과 같은 첫번째 메시지를 큐에서 읽어옴

음수면 : 타입으로 지정한 값의 절댓값과 같거나 작은 첫번째 메시지를 큐에서 읽어옴.

IPC\_NOWAIT, MSG\_NOERROR

#include <sys/msg.h>

int msgctl (int msqid, int cmd, struct msqid\_ds \*buf);

msgctl을 사용하여 msqid로 식별된 메시지 큐에 대한 권한을 할당 해제하거나 변경.

세마포어 : 여러 프로세스 간 동작 순서를 지정, 상호 배제 및 동기화의 고급 관리를 포함.

공유 자원에 여러 프로세스가 동시에 쓰기를 시도하면 데이터가 손상될 수 있음

wait : down, P, lock

signal : up, V, unlock, post

p(sem) //잠금

/\* Critical Section \*/

v(sem) // 잠금 해제

p(sem) {

while sem = 0 do wait;

sem --;

}

v(sem) {

sem++;

if(queue of waiting process not empty)

restart first process in wait queue; }

세마포어는 세마포어 요소의 배열로 구성된다.

세마포어 원소는 최소한 다음 정보가 포함된다. (struct sem)

semval : 세마포어 요소의 값을 나타내는 음이 아닌 정수

sempid : 세마포어 요소를 조작하는 마지막 프로

semncnt : 세마포어 요소 값이 증가하기를 기다리는 프로세스 수

semzcnt : 세마포어 요소 값이 0이 되기를 기다리는 프로세스 수

각각의 세마포어 요소는 두개의 큐를 가진다.

값이 증가하기를 기다리는 프로세스의 큐

값이 0이 되기를 기다리는 프로세스의 큐

#include <sys/sem.h>

int semget(key\_t key, int nsems, int flag);

key와 연관된 세마포어 id를 리턴한다.

nsems : 셋에 있는 세마포어 요소의 수

nsems가 0이면 이미 존재하는 셋을 참조.

semctl

#include <sys/sem.h>

int semctl (int semid, int semnum, int cmd, … /\* union semun arg \*/);

각각의 세마포어 셋의 요소는 사용되기 전에 semctl을 통해 초기화되어야 한다.

arg : cmd의 값에 의존한다.

union semun {

int val; /\* for SETVAL \*/

struct semid\_ds \*buf; /\* for IPC\_STAT and IPC\_SET \*/

unsigned short \*array; /\* for GETALL and SETALL \*/

};

Standard IPC functions

IPC\_STAT : 세마포어 셋의 semid\_ds의 멤버를 arg.buf에 semid로 복사

IPC\_SET : arg.buf로부터 세마포어 셋의 권한을 수정

IPC\_RMID : semid에 의해 식별되는 세마포어 셋을 삭제

Single semaphore operations

GETVAL : 세마포어 요소의 특정 값을 return

SETVAL : arg.val에 세마포어 요소의 특정 값을 입력

GETPID : 요소를 마지막으로 다룬 프로세스ID를 리턴

GETNCNT : 값이 증가하기를 기다리는 프로세스의 수를 리턴

GETZCNT : 값이 0이 되기를 기다리는 프로세스의 수를 리턴

All Semaphore operations

GETALL : arg.array에 있는 세마포어 셋의 값을 리턴

SETALL : arg.array에 있는 세마포어 셋의 값을 변경

#include <sys/sem.h>

int semop (int semid, struct sembuf semoparray[], size\_t nops);

semop 함수는 원자적으로 semid의 세마포어 셋에서 사용자 정의된 세마포어 연산 모임을 수행한다.

struct sembuf {

unsigned short sem\_num; /\* member # in set (0, 1, …, nsems-1) \*/

short sem\_op; /\* operation (negative, 0, or positive) \*/

short sem\_flg; /\* IPC\_NOWAIT, SEM\_UNDO \*/

};

sem\_op > 0 이면 V( ), 즉 잠금해제, 세마포어를 증가시켜 리소스의 잠금 해제를 기록함.

< 0 이면 P( ), 즉 잠금, 세마포어를 감소시켜 semval >= abs(sem\_op), semval = semval-abs(sem\_op)가 될 때까지 리소스 블록의 발생을 기록함.

== 0이면 semval = 0일때까지 제로 블록인지 세마포어를 테스트

프로세스를 사용하지 않는 경우에도 모든 형태의 IPC 객체가 존재하므로 할당된 세마포어를 해제하지 않고 종료되는 프로그램에 대해 걱정해야한다.

Undo 기능이 이를 다뤄야한다.

exit에서 세마포어 조정

세마포어 operation에 SEM\_UNDO flag를 설정하고 p (잠금) 기능을 수행할 때마다 커널은 할당한 sem\_op의 절대값을 기억한다.

만약 프로세스에 미해결 세마포어가 있다면 프로세스가 종료되었을 경우 해당 세마포어에 조정을 적용한다.

Shared memory

공유 메모리는 프로세스가 동일한 메모리 세그먼트에서 읽고 쓸 수 있도록 한다.

클라이언트와 서버간 데이터를 복사할 필요가 없어 IPC의 가장 빠른 형식이다.

서버가 공유 메모리에 데이터를 배치할 경우 클라이언트는 서버가 완료될 때까지 데이터 액세스를 하지 않아야 한다. 세마포어는 공유 메모리 동기화에 쓰이기도 한다.

struct shmid\_ds {

struct ipc\_perm shm\_perm;

size\_t shm\_segsz;

pid\_t shm\_lpid;

pid\_t shm\_cpid;

shmatt\_t shm\_nattch;

time\_t shm\_atime;

time\_t shm\_dtime;

time\_t shm\_ctime;

};

#include <sys/shm.h>

int shmget (key\_t key, size\_t size, int flag);

size : minimum size(in bytes) of the memory segment;

#include <sys/shm.h>

void \*shmat (int shmid, const void \*addr, int flag);

return : 공유 메모리 세그먼트를 가리키는 포인터

shmat 함수는 shmid에 의해 지정된 공유 메모리 세그먼트를 호출한 프로세스의 주소 공간에 첨부하고 shmid에 대한 shm\_nattch의 값을 증가시킨다.

addr

addr 이 0일 때 : 세그먼트는 커널이 선택한 첫 번째 사용 가능한 주소에 연결된다. 권장됨.

addr이 0이 아니고 SHM\_RND가 지정되지 않았을 때 : 세그먼트는 addr에 의해 주어진 주소에 연결됨.

addr이 0이 아니고 SHM\_RND가 지정 되었을 때 : 세그먼트는 (addr – (addr % SHMLBA))에 의해 주어진 주소에 연결된다.

#include <sys/shm.h>

int shmdt (void \*addr);

addr은 shmat에 의해 반환되었던 주소이다. 성공하면 shm\_nattch의 카운트를 감소시킨다.

#include <sys/shm.h>

int shmctl (int shmid, int cmd, struct shmid\_ds \*buf);

cmd는 5개 커맨드중 하나로 지정되어 작동한다.

IPC\_STAT : shmid\_ds 를 buf로 복사한다.

IPC\_SET : buf에서 찾은 값들로부터 shmid 공유 메모리 세그먼트의 필드들의 값을 설정한다.

IPC\_RMID : shmid\_ds, shmid 공유 메모리를 삭제.

SHM\_LOCK : 메모리에서 공유 메모리 세그먼트를 잠근다.

SHM\_UNLOCK : 공유 메모리 세그먼트의 잠금을 해제한다.

lock, unlock은 오직 슈퍼유저만이 가능하다.

소켓

소켓 : 트랜스포트(TCP, uDP) 와 애플리케이션 사이에 있따.

소켓은 통신 엔드포인트의 추상화이다.

표준 유닉스 파일 디스크립터를 사용하여 다른 프로그램과 통신하는 방법.

파일 디스크립터는 단순히 열린 파일과 연관된 정수이다. 하지만 이는 네트워크 연결, FIFO, PIPE, TERMINAL, 실제 디스크 파일, 또는 기타 다른 파일 일 수 있다.

따라서 인터넷을 통해 다른 프로그램과 통신하려면 파일 디스크립터를 통해 수행해야한다.

인터넷 소켓의 두 가지 타입

데이터그램 소켓, UDP

비연결 프로토콜

흐름 및 에러 컨트롤 없음 (신뢰할 수 없음)

작은 메시지

멀티캐스트, 브로드캐스트

스트림 소켓, TCP

연결 지향 프로토콜

flow, error control 사용 (신뢰할 수 있음)

서버 : 소켓 생성, IP 주소 할당, 연결 요청 대기 상태, 연결 수락

전화기 : 폰 구입, 폰 번호 할당, usim 끼워, 전화 받아!

서버

#include <sys/socket.h>

int socket (int domain, int type, int protocol); 소켓 생성

int bind (int sockfd, const struct sockaddr \*addr, socklen\_t len); 주소 할당

int listen (int sockfd, int backlog); 연결 요청 대기 상태 진입

int accept (int sockfd, struct \*restrict addr, socklen\_t \*restrict len); 연결 요청 수락

서버의 경우 : socket -> bind -> listen -> accept 순이다.

클라이언트

socket ->

int connect (int sockfd, const struct sockaddr \*addr, socklen\_t len); 연결 요청

주소 할당 과정인 bind() 가 자동으로 됨.

서버 : socket() -> bind() -> listen() ->accept() -> recv() -> send() -> ->recv() ->close()

클라이언트 : socket() -> connect() -> send() -> recv() -> close()

UDP

server : socket() -> bind() -> recvfrom() -> sendto()

client : socket() -> bind() -> sendto() -> recvfrom()

파이프로 read, write 하는 4가지 방법 : pipe, fcntl, mkfifo, select,

msgsnd (msgid, (void\*)&msg, SIZE, flag);

두번째는 사용자 정의 구조체임

struct mymsgbuf {

long type;

char text[SIZE];

};

msgrcv (msgid, (void\*)&inmsg, SIZE, type, flag);

shmat (shmid, NULL, 0);