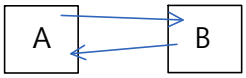
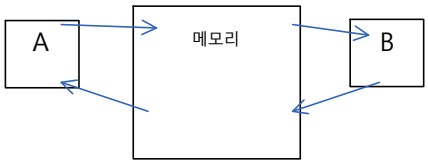
Interprocess Communication (IPC)

2020년 6월 2일 화요일 오전 11:02



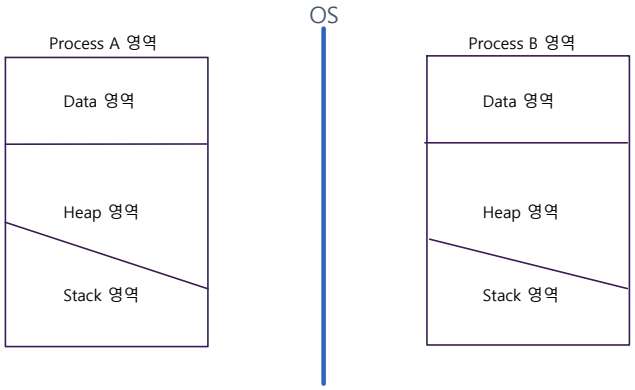
의미 : 완전히 독립된 두 프로세스가 있을 때, A가 B에게 B가 A에게 데이터를 전달하는 것을 IPC라 한다.

근데 여기서 단순히 보내고 받는다는 것만 생각하기 쉬운데, IPC는 "메모리 공유" 기법을 통해 send, read가 이루어짐을 알아야 함.



위의 그림처럼 메모리 공유 관점에서 A가 data를 가져다 놓고, B가 가져가는 통신(그 반대도 동일)임을 알아야 함.

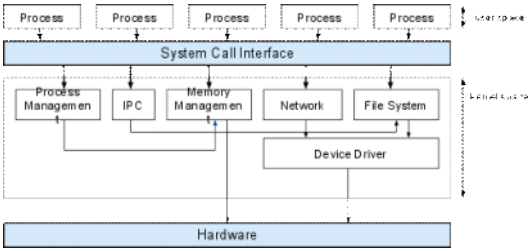
그런데 우리가 IPC를 어렵게 느끼는 이유는,



OS (또는 kernel)가 각각의 process 영역을 분리시켜서, 영역 외의 영역엔 접근이 불가능하기 때문이다.

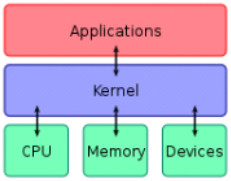
단순하게 메모리 공유 방법을 생각해보면 process A의 data영역에 process B의 접근을 어용하는 영역을 만든 다음, 그 영역에 A가 data를 보내면 B가 받아가서 쓰는걸 생각할 수 있지만, OS에서 제한하고 있기 때문에 IPC가 어렵다는 것.

그럼 OS가 이를 제한하고 있기 때문에, 이들 간의 메모리 공유 관련 해결책도 OS가 제공하면 됨.
=> 이것이 IPC 기법



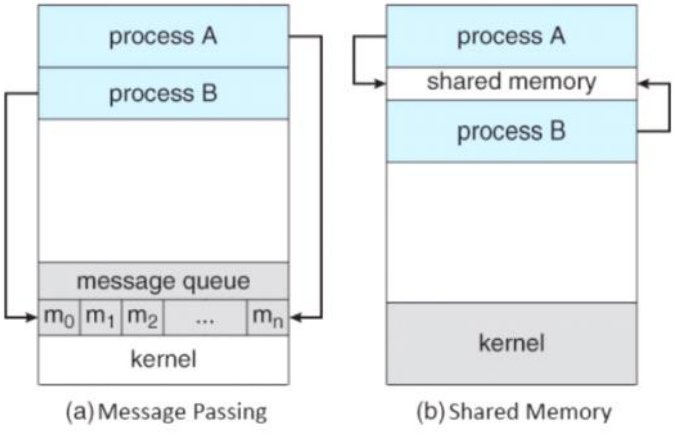
프로세스가 독립적임을 리눅스 커널 구조 그림을 통해 알 수 있다.

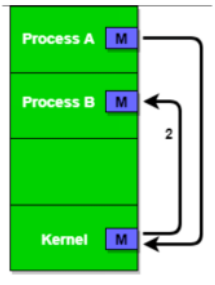
커널 : 컴퓨터의 운영 체제의 핵심이 되는 컴퓨터 프로그램의 하나로, 시스템의 모든 것을 완전히 통제한다. 운영 체제의 다른 부분 및 응용 프로그램 수행에 필요한 여러 가지 서비스를 제공.



커널이 응용 소프트웨어를 컴퓨터 하드웨어에 연결함을 알 수 있다.

IPC모델은 message passing과 shared memory 두 가지 모델이 있다.





Message passing은 위의 그림처럼 kernel을 이용해 교환한다.

Message passing은 kernel이 memory protection을 위해 대리 전달해 주는 것을 말한다. 따라서 안전하

고 동기화 문제가 없다. 하지만 성능이 떨어지는 단점을 가지고 있다. (message queue처럼 한 번 어딘가를 거쳤다가 전달해야 하는 오버헤드가 있기 때문.)
Shared memory는 두 프로세스간 공유된 메모리를 생성한 후 이용하는 것을 말한다. 성능은 좋지만 동기화 문제가 발생한다. (동기화 문제란 프로세스 A가 쓴걸 프로세스 B가 읽는 상황인데 B가 조금 일찍 읽게 되는 상황이 발생한다면 A가 작성을 다 한걸 읽는 건지 작성 덜한걸 읽는 건지 알 수가 없음. 이런 경우 다 썼는지 안 썼는지 확인하는 것이 동기화.)

Shared Memory에 대한 추가 설명.

정의 : 여러장치(주로 CPU)가 공동으로 사용하는 메모리, 혹은 여러 개의 프로세스가 공통으로 사용하는 메모리를 의미한다.

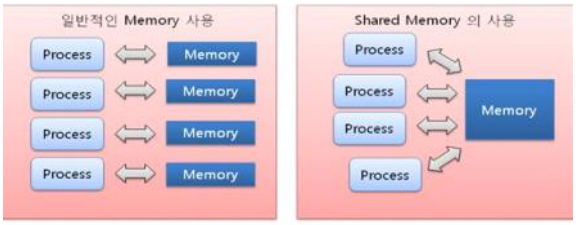


그림 2 일반적인 메모리 사용과 공유메모리 사용의 차이

일반적인 메모리와 공유메모리가 어떻게 다른지 그림을 통해 확인할 수 있다.

공유메모리가 사용되는 영역은 크게 하드웨어적인 부분과 소프트웨어적인 부분으로 나눌 수 있다.

지금 IPC를 다루고 있기 때문에, 소프트웨어에서의 공유메모리에 대해서 살펴보고 넘어가자.
IPC에서 shared memory는 위의 그림처럼 메모리를 공유할 때도 쓰지만, 다른 하나는 메모리를 아끼기 위한 보존관리 방법으로 사용이 된다. 이는 아래의 그림을 통해 쉽게 이해될 수 있다.

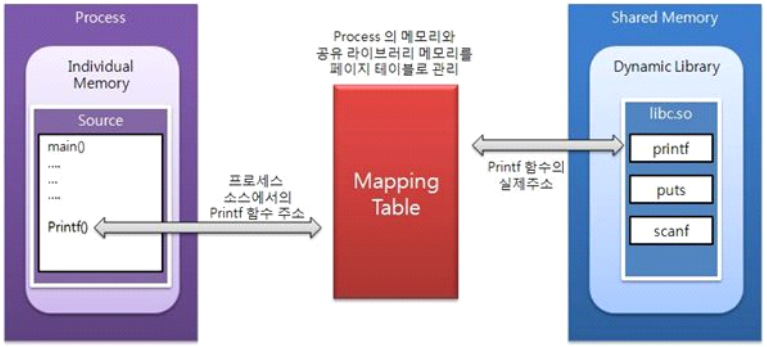


그림 6 동적 라이브러리의 주소 Mapping

프로그래머들은 프로세스가 자주 사용하는 특정한 코드가 있음을 발견했다. 그러고는 그 코드들을 따로 분리하여 라이브러리(library)를 만든 후, 라이브러리를 공유메모리 부분에 적재시켜, 메모리 공간을 아꼈다. (가끔 linux에서 쓰는 함수들과 헤더에 대해서 구글링 하다보면 그 함수가 어느 폴더에 저장되어 있는지 적혀있는 경우가 있었는데, 그런것들이 모두 shared memory구나 생각함.) OS는 프로세스의 메모리 영역에 해당 라이브러리 부분을 mapping해줬다.

Shared memory의 특징
: IPC 기법 중 처리속도가 가장 빠르다. 다른 기법들과는 다르게 메모리 자체를 공유하므로 위에서 봤던 pipe처럼 데이터 복사와 같은 불필요한 과정이 발생하지 않기 때문.
단점으로는 반드시 같은 Ram(기계)에서만 사용가능 하다는 것이다. => 근데 네트워크 사용이 불가능하다는 단점이 워낙 치명적이라 다른 IPC 기법보다 덜 강력하다.

** IPC의 종류??

IPC의 종류는 여러 가지가 있겠지만, 대표적으로

1. 시그널 (Signal)
2. 파이프 (Pipe)
3. 메세지 큐 (Message Queue)
4. 공유 메모리 (Shared Meory)
5. 메일박스 (Mailbox)

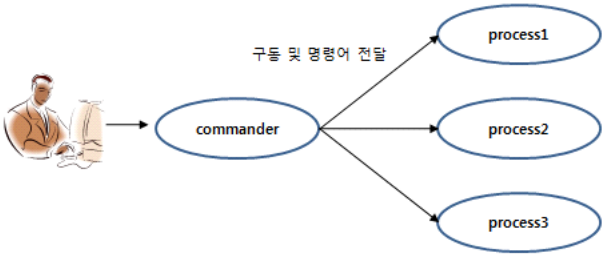
1. Message queues

웹에서 많은 데이터 처리로 인해 대기 요청이 쌓이게 되면 이러한 것들이 서버의 성능을 저하시키고 최악의 경우 서버를 다운 시킨다. 이러한 이유로 웹 서버를 구성할 때, 성능에 대한 부분도 고려를 해야 함.
이러한 문제점을 해결하기 위해 다양한 방법들이 있지만 좀 더 저렴하고 쉽게 구현할 수 있는 방법이 바로 메시지 처리 방식이다.

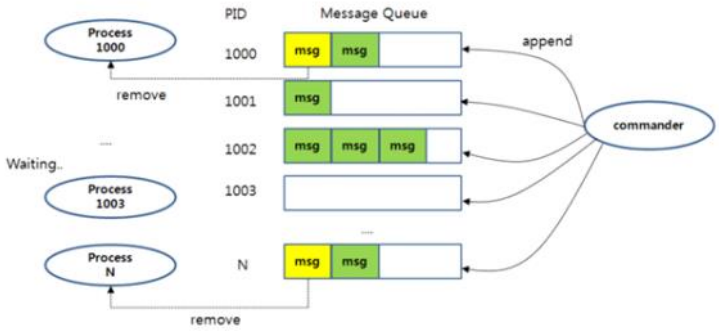
기본적인 원리는 producer가 message를 queue에 넣어두면 consumer가 message를 가져가 처리하는 방식이다. (메시지를 queue 데이터 구조 형태로 관리하는 것.)
메시지 큐는 kernul에서 전역적으로 관리되며, 모든 프로세스에서 접근가능 하도록 되어있으므로, 하나의 메시지 큐 서버가 커널에 요청해서 메시지 큐를 작성하게 되면, 메시지 큐의 식별자를 아는 모든 프로세서는 동일한 메시지 큐에 접근함으로써 데이터를 공유할 수 있다.
message queue의 장점은 다른 IPC 공유방식에 비해서 사용방법이 매우 직관적이고 간단하다. 그리고 메시지의 type에 따라 여러 종류의 메시지를 효과적으로 다룰 수 있다.

https://www.joinc.co.kr/w/Site/system_programing/IPC/MessageQueue : 메시지 큐 구현 방법에 대해 잘 나와 있음.

메시지 큐 예시)

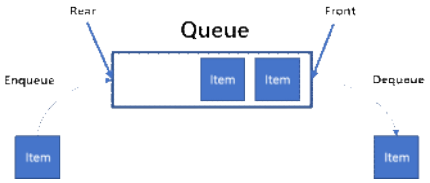


사용자가 commander라는 프로그램으로 명령어를 입력하면, commander가 process A를 실행시키고, 이후 입력되는 메시지들을 process A에 전달하는 구조.

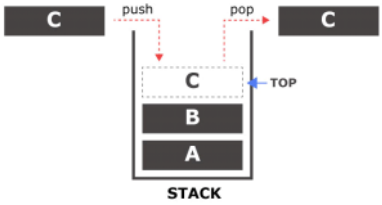


위의 그림처럼 commander는 입력이 들어오면 입력을 처리할 process를 선택하고 그 곳에 메시지를 넣어준다. 그러면 process는 수신된 메시지를 꺼내서 일을 처리한다.

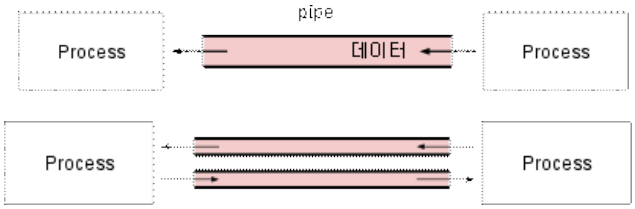
- Queue

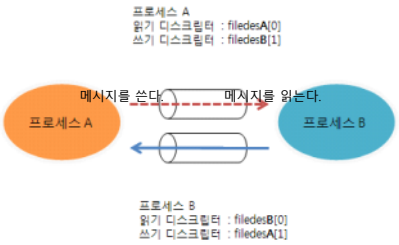


- stack



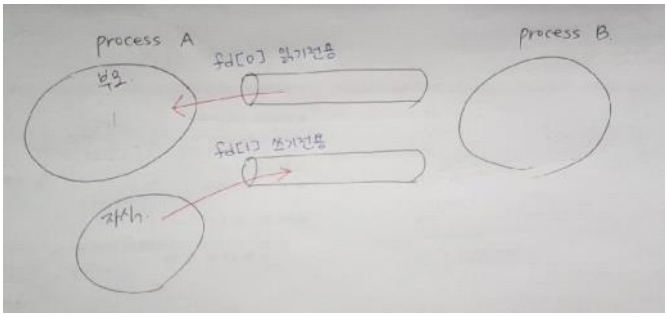
2. Pipe





파이프는 그냥 우리가 생각하는 그 파이프와 비슷하게 작동한다고 생각하면 쉽다.

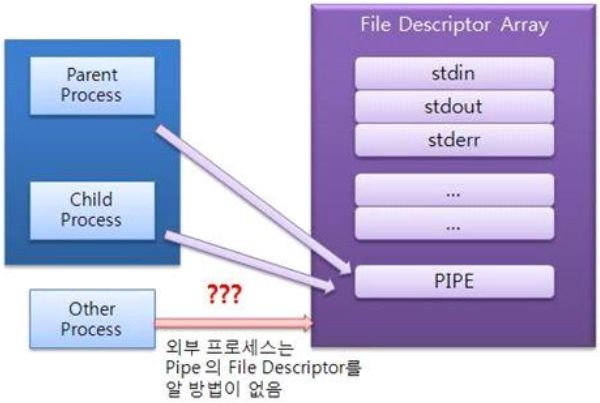
파이프의 특징: 파이프는 두 개의 프로세스를 연결하는데 파이프는 입구와 출구의 개념이 없는 단순한 통로이다. 방향성이 없기 때문에 자신이 쓴 메시지를 자신이 읽을 수도 있기에 두 개의 프로세스가 통신할 때는 읽기전용(read) 파이프와 쓰기전용(write) 파이프 두 개의 파이프를 쓰게 된다. 이때 읽기전용 파이프와 쓰기전용 파이프를 만드는 방법은 두 개의 파이프를 만든 후 한쪽 파이프를 일부러 닫아버리면 된다.
또한, 파이프의 중요한 특징이자 유의할 것은 파이프는 fork()함수에 의해 자식프로세스가 부모프로세스를 그대로 복사할 수 없다는 것이다. 파이프의 경우에는 복사되는 것이 아니라 자식 프로세스와 부모 프로세스가 그냥 같은 파이프를 가리키게 된다. (뒤에서 예제를 보면 와 닿을 것 같다.)



여기서 파이프는 여러 개의 프로세스가 데이터를 주고받기 위해 사용하는 임시공간이라고 했는데, 이때 이 임시 공간은 실제로 파일 시스템에 생성되는 임시 파일이다. (이 말이 와닿지 않아도, 파이프도 파일임을 유의할 것.)

파이프는 이름없는 파이프와 이름을 가진 named pipe 두 가지가 있다.

이름없는 파이프는 부모 자식프로세스 사이에서만 파이프 사용이 가능함. 다른 외부 프로세스는 pipe의 주소를 알 길이 없기 때문. (아래의 그림 참고)



이름있는 파이프 (named pipe)의 경우 이름이 있기 때문에, 지정한 이름으로 파일 디스크립터를 열수 있으므로, 통신범위엔 제한이 없다.

Pipe의 장점 : 매우 간단하게 사용할 수 있다. 한쪽 프로세스를 읽기전용으로 만들고, 나머지 프로세스를 쓰기전용으로만 만든다면 데이터의 흐름을 고민 없이 pipe를 사용하면 된다. 큰 데이터들을 주고 받을 수 있다.

단점은 위에서도 계속 말했듯이, 읽기와 쓰기를 모두 하기 위해서는 pipe를 두 개 만들어야 하는데, 구현이 복잡해 질 수 있다. 예를 들어 read와 wirte가 기본적으로 block(한쪽을 닫아서)으로 작동하기 때문에 read가 대기 중이면 read가 끝나기 전까진 write할 수 없다.

상황 봐가면서 코드 한번 보고 넘어가기.

```
1  #include <unistd.h>
2
3  int pipe(int filedes[2]);
```

함수의 인자로 int형 배열이 들어감. 배열이 파이프의 지시번호를 리턴하기 때문.

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
	int n, fd[2];
	char buf[255];
	 int pid;

	 if (pipe(fd) < 0)
	{
		perror("pipe error : ");
		exit(0);
	}

	// 파이프를 생성한다.
	if ((pid = fork()) < 0)
	{
		perror("fork error : ");
		exit(0);
	}

	// 만약 자식프로세스라면 파이프에 자신의 PID 정보를 쓴다.
	else if (pid == 0)
	{
		close(fd[0]);
		while(1)
		{
			memset(buf, 0x00, 255);
			sprintf(buf, "Hello Mother Process. My name is %d₩n", getpid());
			write(fd[1], buf, strlen(buf));
			sleep(1);
			}
		}

		// 만약 부모프로세스라면 파이프에서 데이타를 읽어들인다.
		else
		{
			close(fd[1]);
			while(1)
			{
				memset(buf, 0x00, 255);
				n = read(fd[0], buf, 255);
				fprintf(stderr, "%s", buf);
			}
		}
}
```

getpid 함수는 부모프로세스의 ID를 리턴함.

read( )함수 / 헤더: unistd.h
: open( )함수로 열기를 한 파일의 내용을 읽기를 한다.
ssize_t read(int fd, void *buf, size_t nbytes)
fd : 파일 디스크립터
void *buf : 파일을 읽어 들일 버퍼
size_t nbytes : 버퍼의 크기

ssize_t : 정상적으로 실행되었다면 읽어들인 바이트 수를, 실패했다면 -1을 반환.

표준 출력함수 fgets( )에는 버퍼가 크더라도 파일의 첫 번째 행만 읽어서 반환하지만 read( )는 버퍼의 크기만큼 읽을 수 있다면 모두 읽어 들인다. 그러므로 read( )에서 사용할 버퍼의 크기가 파일 보다 크다면 파일의 모든 내용을 읽어 들이게 된다.

· write 파일 쓰기 / 헤더: unistd.h
: open( ) 함수로 열기를 한 파일에 쓰기를 한다. open( )함수는 fctl.h에 정의 되어 있지만 write( ), read( ), close( )는 unistd.h에 정의되어 있다.
ssize_t write(int fd, const void *buf, size_t n)
fd : 파일 디스크립터
void *buf : 파일에 쓰기를 할 내용을 담은 버퍼
size_t n : 쓰기할 바이트 개수

ssize_t : 정상적 쓰기를 했다면 쓰기를 한 바이트 개수를, 실패했다면 -1을 반환.

3. Mailbox (mailslot)

원리 : 앞에서 말했듯이 프로세스는 자신에게 할당된 메모리 공간 이외에는 접근할 수가 없다. 따라서 mail slot(우체통)을 이용하여 데이터를 송,수신 할 수 있다. 데이터를 수신하고자 하는 process A가 mail slot을 생성한다. 그리고 데이터를 송신하고자 하는 process B가 process A의 mail slot의 주소로 데이터를 송신한다. 그 후 process A가 자신의 mail slot을 통해 데이터를 얻게 된다.

특징:

1. 메일슬롯은 할당된 주소를 기반으로 통신하기 때문에 관계없는 프로세스 사이에서도 서로 통신이 가능하다. (예를들어, 네트워크 연결되어 통신하는 프로세스들, 부모자식 관계가 아닌 프로세스들 ...)
2. 한쪽방향으로만 메시지를 전달할 수 있다. (서로 데이터를 주고 받기 위해서는 각각의 메일슬롯을 생성해야 한다.)
3. 메시지를 보내는 사람은 한번 메시지를 전송할 때, 여러 receiver에게 동일한 메시지를 전송할 수 있다.