



System Programming Archive

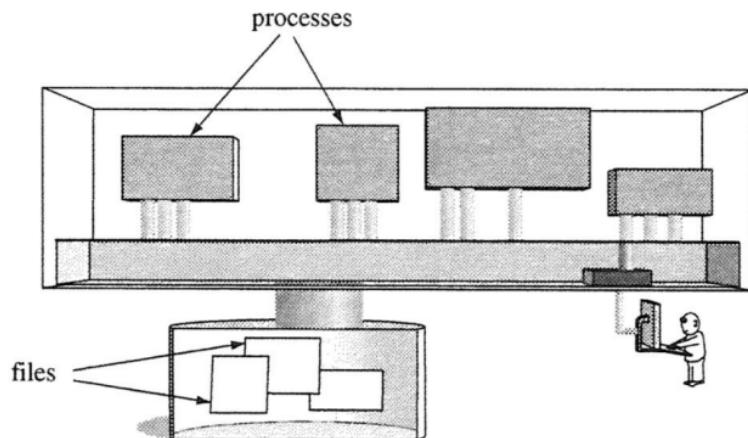
▼ Processes and Programs

- 프로세서 : CPU
- 프로그램 : 내가 실행하고자 하는 어떤 것(a.c, a.java 와 같은 바이너리 코드)
- 프로세스 : 프로그램의 실행 단위
- Linux 명령어 **ps(process status)** : 현재 실행중인 프로세스를 보여준다.

```
kimgrace@Kimgrace:~/lab8$ ps
  PID  TTY      TIME CMD
    37 pts/1    00:00:00 bash
   154 pts/1    00:00:00 ps*
```

- 시스템 프로그래밍
 - 운영체제한테 부탁하는 방법에 대해 배운다.
- 운영체제
 - 프로세스 (file ← read)
 - Memory
 - 파일(disk에 접속하는 방법)

Programs in Action, or Processes



- “Data” and “programs” stored in files on disk
- “Programs run” in processes

How can we view what processes are running?

- 프로세스란?(Processes == Programs in Action)
 - 프로세스 : 고용직 사람
 - Running a program
 - machine instructions 를 memory에서 로딩 해 오는 것
 - CPU 가 각 명령을 순서대로(또는 병렬로) 실행하도록 하는 것
 - 프로세스는 분기를 위해 존재한다.
 - ex1) chrome lab
 - ex2) vscode
 - 프로그램 :
 - file에 기계적 인스트럭션을 순차적으로 저장하는 것(소스 코드를 바이너리 코드로 바꾸는 컴파일링)
 - 어떤 주어지는 업무(PCB: process control block) == struct PCB{}, 운영체제 안에서 프로세스를 구현하기 위한 어떤 구조체의 형태 → struct PCB (주로, ID가 기록 돼 있다.)
- Executable vs Process(in Unix/Linux)
 - Executable : 기계어 명령어 및 데이터 목록.
 - Process: 메모리 공간 그리고 프로그램 runs을 유동적으로 settings 하는 것.
- Load → disk에 있는 내용인 file 을 책상 위에 옮겨놓는 것
 - ex) a라는 장난감을 조립하는 업무
- 관련 API
 1. execvp() → 새로운 장난감을 옮겨놓는 것(ex. a 장난감 조립 중 b 장난감을 책상위에 옮겨놓는 것)
 2. fork → 고용자를 뽑아오는 것(ex. b를 조립할 고용자를 뽑아)
 3. shell program(감독관처럼 어떤 일을 시키고 감시하는 것) → ex.ssh, dashshell, zshell, pcshell,
 4. wait() → 자식 프로세서의 일을 끝낼 때 까지 기다리는 것
 5. pipe() → 결과를 돌려받을 때 쓰는 것

6. Thred() → 어떤 프로그램 안에 존재하는 것으로써 다중 탭이 존재 할 때, 한 탭에 오류가 생기면 모든 탭을 종료하는게 아니라 해당 탭만 종료하는 것.

```
//Ppt 자료 psh1.c 코드 수정 (교수님 ver.)  
  
#include <stdio.h>  
#include <unistd.h>  
#include <string.h>  
  
int main ()  
{  
    char *arglist[10];  
    char buf[100];  
    int i;  
    i = 0;  
    while (i < 10) {  
        printf ("Arg[%d]? ", i);  
        gets (buf);  
        arglist[i] = (char*) malloc (strlen(buf) + 1);  
        strcpy (arglist[i], buf);  
        if (strcmp (arglist[i], "") == 0) {  
            arglist[i] = NULL;  
            break;  
        }  
        i++;  
    }  
    execvp (arglist[0], arglist);  
}
```

▼ Process vs File

| Files | Option | Processes |
|---------------------------------------|------------|--|
| data Contain | Contains | executable code Contains |
| Have attributes | attributes | Have attributes |
| Created or destroyed by the kernel | kernel | Created or killed by the kernel |
| Stored on disk | disk | Stored in memory by the kernel |
| Has an allocation list of disk blocks | memory | Has a structure to hold an allocation list of memory pages |

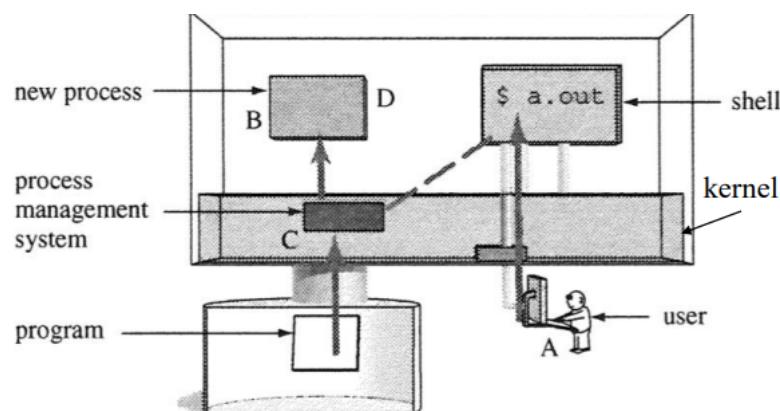


Memory management vs Disk management(How "similar" or "different")

- Kernel : Process 을 만든다.
- 매우 유사한 방식으로는 파일을 만드는 것
- The Shell
 - A Tool for Process and Program Control
- Shell
 - OS 에 접근하기 위한 user interface
 - 프로그램은 프로세스와 다른 실행 프로그램을 Manages한다.
 - **Has three main functions : Pair discussion**
 1. **Program running** : 프로그램 실행
 2. **Input/Output managing** : input args(인풋 인자)를 주고, 결과를 컨트롤한다.
 3. **Directly programmable UI** : 코드 접근이 유용하도록 만든다.
- Managing I/O
 - >, <, | input/output redirectoin symbols 이다.
 - 우분투에서 자체적으로 명령어 탭에서 우분투 컴파일 문법을 사용해서 간단한 코드를 작성 할 수 있다.

▼ Principle of Program Running in a Shell (Writing a simplified ver. of shell)

- How the Shell Runs Programs
 - 입력자가 프롬프트를 출력하고 명령을 입력하면 쉘은 명령을 실행한다. 그 다음, 쉘은 프롬프트를 인쇄한다.



1. 유저가 a.out 을 입력한다.
2. 쉘은 새로운 실행 프로그램인 프로세서를 만든다.
3. 쉘은 disk 에서부터 로드해서 프로세스한다.
4. 프로그램 유닛이 끝나면 수행 종료
 - execvp

| execvp | | |
|---------|--|-------------------------|
| PURPOSE | Execute a file, with PATH searching | |
| INCLUDE | #include <unistd.h> | |
| USAGE | result = execvp(const char *file, const char *argv[]) | |
| ARGS | file | name of file to execute |
| | argv | array of strings |
| RETURNS | -1 | if error |

- How Does a Program Run a Program
 1. 현재 실행하고 있는 프로세스가 있다.(execvp를 호출한다.)
 2. 커널은 disk에서 프로세스로 프로그램을 로드한다.
 3. 커널은 arglist 를 프로세스에 복사한다.
 4. 커널은 main(argc, argv)을 호출한다.
- exec1.c 소스 코드

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

int main(int argc, char *argv[]){
    char *arglist[3];
    arglist[0] = "ls";
    arglist[1] = "-l";
    arglist[2] = 0;

    printf("/** About to exec 'ls -l' \n");
    execvp("ls", arglist);
    printf("/** ls is done, Good bye\n");
}
```

```
*** About to exec 'ls -l'
total 120
-rw-r--r-- 1 kimgrace kimgrace 274 Nov  3 09:46 a.c
-rwxr-xr-x 1 kimgrace kimgrace 16880 Nov  3 09:46 a.out
-rw-r--r-- 1 kimgrace kimgrace 153 Nov  3 09:42 b.c
-rwxr-xr-x 1 kimgrace kimgrace 16824 Nov  3 09:43 b.out
-rwxr-xr-x 1 kimgrace kimgrace 16792 Nov  4 17:54 exec1
-rw-r--r-- 1 kimgrace kimgrace 258 Nov  4 18:01 exec1.c
-rwxr-xr-x 1 kimgrace kimgrace 16952 Nov  3 21:22 forkdemo1
-rw-r--r-- 1 kimgrace kimgrace 451 Nov  4 11:53 forkdemo1.c
-rwxr-xr-x 1 kimgrace kimgrace 17128 Nov  3 11:30 psh1
-rw-r--r-- 1 kimgrace kimgrace 548 Nov  3 11:30 psh1.c
```

- execvp()는 정체가 뭘까?
 - 어떤 일이 일어날까
 - 커널은 현재 실행중인 프로세스에서 새로운 프로그램을 로드해온다. 이 때, 새로 로드 해 온 프로그램을 새로운 코드와 데이터로 대체한다.
 - 그렇게 되면, 현재 프로그램은 프로세스로부터 삭제 당하고, 새로운 프로그램이 현재 프로세스에서 실행한다.
 - The exec system call

- execvp : 라이브러리 함수이다. execvp 시스템 콜 함수이다.
- execvp 가 호출되면, 메모리 위치가 프로그램의 수신 요구에 맞게 변경된다. 즉, 프로세스는 동일하지만, 콘텐츠는 새로운 것으로 대체된다.
- Another Ex
 - Compare process ID(or, pid) before and after execvp
 - pid_t getpid(void)
 - Returns the process ID of the current process
 - Defined in <unistd.h>
- after.c

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

int main(int argc, char *argv[]){
    pid_t pid = getpid();
    printf("After execvp(): %d\n", pid);

    return 0;
}
```

- before.c

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

void main()
{
char *arglist[2];
pid_t pid = getpid();

arglist[0] = "./after";
arglist[1] = 0;

printf("Before execvp(): %d\n", pid);

execvp(arglist[0], arglist);
return;
}
```

```
kimgrace@Kimgrace:~/lab8$ ./after
After execvp(): 210
```

```
kimgrace@Kimgrace:~/lab8$ vi before.c
kimgrace@Kimgrace:~/lab8$ ./before
Before execvp(): 221
After execvp(): 221
kimgrace@Kimgrace:~/lab8$
```

- psh1.c

- psh1 은 명령이 실행 된 후 종료된다.
- 사용자가 다른 헬을 실행하기 위해 쉘을 다시 실행하길 원한다면 어떻게 될까?

 1. 새로운 프로세스를 만든다.
 2. 현재 프로그램에서 새로운 프로세스를 만든다.

▼ Creating a New process

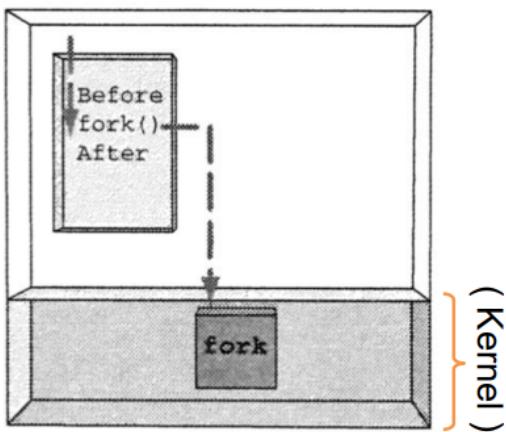
- **fork()**
- How Do We Get a New Process?
 - The answer is ...
 - Use fork

- System call: fork(); //takes no argument

| fork | |
|-------------|--|
| PURPOSE | Create a process |
| INCLUDE | #include <unistd.h> |
| USAGE | pid_t result = fork(void) |
| ARGS | none |
| | -1 : if error |
| RETURNS | 0 : when child process pid : pid of child when parent process |

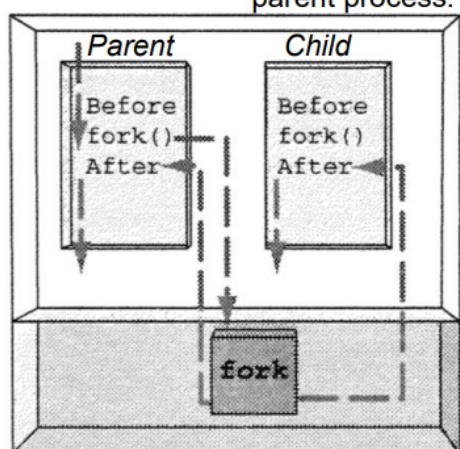
Explanation of fork: Making a Copy of a Process

Before fork:



One flow of control enters the fork kernel code.

After fork:



Two flows of control return from the fork kernel code.

- What about fork()
 - fork 가 수행되면, kernel에선 이런 일들이 발생한다.
 1. Allocate : 새로운 chunk 단위의 메모리와 데이터 스트럭처를 할당한다.
 2. Copy : 원래의 프로세스(코드와 데이터)는 새로운 프로세스로 들어간다.
 3. Add : 새로운 프로세스는 셋팅이 되어 실행된다.
 4. Return : 두 개의 프로세스를 Control한다.
- forkdemo2.c

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

int main(int argc, char * argv[]) {
    int ret_from_fork, mypid;
```

```

    mypid = getpid();
    printf("Before: my pid is %d\n", mypid);
    ret_from_fork = fork();
    sleep(1);

    printf("After: my pid is %d, fork() said %d\n", getpid(), ret_from_fork);
}

```

```

kimgrace@Kimgrace:~/lab8$ ./forkdemo2
Before: my pid is 229
After: my pid is 229, fork() said 230
After: my pid is 230, fork() said 0

```

- forkdemo3.c

```

int main(int argc, char* argv[]){
    printf("my pid is %d\n", getpid());
    fork();
    fork();
    fork();
    printf("my pid is %d\n", getpid());
}

```

```

kimgrace@Kimgrace:~/Lab8$ ./forkdemo3
My pid is 255
My pid is 256
My pid is 258
My pid is 257
My pid is 259
My pid is 260
My pid is 261
My pid is 255
My pid is 262

```

- forkdemo4.c

```

int main(int argc, char* argv[]){
    int fork_rv;
    printf("Before: my pid is %d\n", getpid());
    fork_rv = fork(); /* Create a new process */
    if ( fork_rv == -1 ) /* Check for error */
        perror("fork");
    else if ( fork_rv == 0 )
        printf("I am the child. my pid=%d\n", getpid());
    else
        printf("I am the parent. my child is %d\n", fork_rv);
}

```

```

kimgrace@Kimgrace:~/lab8$ ./forkdemo4
Before: my pid is 279
I am the parent. my child is 280
I am the child. my pid = 280

```

- Summary : Building a Shell Using ?, ?, ?, and ?

- How to create a new process
 - fork()
- How to run a program
 - execvp()

- How to tell the parent to wait until the child process finishes executing the command
 - wait(): to be discussed next
- The fourth system call will be discussed after
 - wait()

▼ Pipe

- 정렬 명령어 → sort → 종료시 ctrl+d → sort finish

```
kimgrace@Kimgrace:~/lab9$ sort
acd
bfe
acd
bfe
```

- vi a.txt → sort < a.txt (파일이 sort되도록 들어감, file input)
- 만약 sort < a.txt > b.txt → b.txt에 sort명령이 실행됨
- sort > b.txt(file output)
- ls>b.txt
- cat b.txt
- ls | sort → Pipe라고 한다. ls는 sort 명령어 2개 실행하고 2개 출력한다. → 한 프로세서에서 또 다른 프로세서한테 수행을 전달 할 때 쓰는 도구
- 1번이 standrad input이다. 메모리 스택상에서 0, 1, 2, 3, ... 있는데 0, 1, 2, 는 이미 값 있고, 우리가 쓸 때는 3번부터 채워진다. → ftest.c 참고
- 2초 후에 실행하라고 하지만, 누가 먼저 실행 될지는 모른다. fork 이후는 독립적인데, 파일은 하나인데, 두 개가 동시에 쓰니깐 write시 어떤 애가 쓰면, 그 후 기억 포인터가 다음에 써야 할 위치를 기억하고 다음 애가 쓴다. 그럼 그 다음 애가 read하고 있다가 write 를 한다.

```
#include <stdio.h>
#include <fcntl.h>

int main()
{
    int pid, fd;
    fd = open("test.txt", O_RDWR | O_CREAT, 0666);
    printf("fd = %d\n", fd);
    pid = fork();
    if (pid == 0)
    {
        //child process
        sleep(2);
        write(fd, "hello in child\n", 16);
    }
    else {
        //parent process
        sleep(2);
        write(fd, "gi in PA\n", 11);
    }
    close(fd);
}
```

- 3, 4가 찍힐텐데, 4에 hello를 쓰고, read후 write를 한다.

```
#include <stdio.h>
#include <fcntl.h>

int main()
{
    int pid;
    int fd[2];

    pipe (fd);

    char buf[100];

    printf("fd[0] = %d\n", fd[0]);
    printf("fd[1] = %d\n", fd[1]);
    write(fd[1], "hello", 6);
    read(fd[0], buf, 100);
    printf("buf = %s\n", buf);
}
```

- 부모 프로세서와 자식 프로세서가 있는데, 여기서 부모 프로세서가 메모리 스택 상에서 정보를 주고 read후, 자식 프로세서한테 값을 전달해주는 것이다.

```
#include <stdio.h>
#include <fcntl.h>

int main()
{
    int pid;
    int fd[2];
    char buf[100];
    pipe (fd);

    printf("fd[0] = %d\n", fd[0]);
    printf("fd[1] = %d\n", fd[1]);
    pid = fork();
    if (pid == 0)
    {
        close(fd[1]);
        read (fd[0], buf, 100);
        printf("child : %s\n", buf);
    }
    else
    {
        close(fd[0]);
        write(fd[1], "hello, in PA", 12);
        wait(NULL);
    }
}
```

- 만약, 부모 프로세서와 자식 프로세서가 서로 교환하고 싶게 만들고 싶다면, pipe를 한 개 더 만들면 된다.



pipe 는 first in-first out 방식인 fifo 방식이다.

- ls | sort → 내가 했던 것들을 순차적으로 정렬시켜준다.

```
kimgrace@Kimgrace:~/lab9$ ls | sort
a.txt
b.txt
c.c
c.out
ftest
ftest.c
pipetest.c
ptest
test.txt
```

- fife 생성 명령어 : mkfifo {fifo1}

```
kimgrace@Kimgrace:~/lab9$ mkfifo fifo1
kimgrace@Kimgrace:~/lab9$ ls
a.txt b.txt c.c c.out fifo1 ftest ftest.c pipetest.c pipetest2.c ptest test.txt
kimgrace@Kimgrace:~/lab9$ ls -al f*
prw-r--r-- 1 kimgrace kimgrace 0 Nov 10 10:12 fifo1
-rwxr-xr-x 1 kimgrace kimgrace 16912 Nov 10 09:35 ftest
-rw-r--r-- 1 kimgrace kimgrace 331 Nov 10 09:29 ftest.c
```

- pipe는 무조건 쌍으로 있어야 한다. 한 애는 write, 다른 한 애는 read

▼ Message

- IPC(inter process communication) : 두 개 이상의 프로그램이 서로 소통하며 실행하는 것
IPCS(IPC 동기화)
- ipcrm -q {value} -> del

 - message 기반 send ---> (message) ---> recv
 - 공유 메모리 기반 : 메모리 주소상에 X위치 상에 어떤 값을 write한다. 그럼 서로 데이터를 공유하게 된다.

▼ Semaphore

- semaphore : 정수 변수 → 컴파일 명령어 : gcc sem1.c -o {컴파일 파일 이름}-lpthread
- Math function : 수학 함수 → 컴파일 명령 : gcc {* .c} -o {compile name} -lpthread
- semaphore (동시에 접근이 불가하도록 만들어준다.)
 - signal() → 정수 변수에 접근해 신호를 줄 수 있다.
 - s++; ⇒ atomic하게 실행한다.
 - wait() → 정수 변수를 바꿀 수 있다.
 - while(s≤0) break; // 누군가 signal을 호출해 줘야 풀려난다.
 - while end → s ≥ 0 → s—;
 - 이 위의 코드가 atomic하게 실행함. (wait를 실행하거나 하지 않거나, block되어있으면, 딱 한 프로세스만 실행하도록)
- 3. init() → 정수 변수 초기화
- sem = sem_open("testsem", O_CREAT, 0666, 2);
- 저기서 2 자리에 어떤 정수 값을 넣으면 그 해당 값만큼 block되지 않고 실행 될 수 있는 것이 생성된다.
- test&set \$1, a → 메모리 스택상에 a라는 공간에 \$1 값을 넣는다.
- 세마포어 삭제 명령어 rm /dev/shm/sem.{세마포어 이름}

▼ Thread

- Semaphore에서 한 것은 독립적인 세마포를 끼리 한 것이다.
- 오늘 내용은 한 프로세서 내에 또 다른 커뮤니티를 만들 수 있음, 이를 Thread라고 한다.
- 프로세서는 서로 독립적인 프로그램이 실행되지만, Thread는 한 프로그램 내에서 독립적으로 실행이 되도록 함

- Context switch -> 현재 실행중인 프로세서의 메모리를 안전한 곳으로 옮겨놓고, 새로운 내용을 CPU에 저장하고, 내용이 없으면 MM에서 값을 가져오는 등의 역할
따라서 이 스위치를 빠르게 하는게 관건
- 독립적으로 실행하고 싶은 함수의 모양은 아래와 같이 나타낸다.

```
void *func(void *arg){}
```
- thread를 생성하면, 스택에 code, state, stak, Heap이 있는데, stak만 공유하는 것이 thread이다.
- prime_count 예제 익혀두기 $O(2^n)$ -> 소수 구하는 시간복잡도 시간이다.
어떤 n자리의 자릿수를 구하는데 걸리는 시간복잡도는 $O(10^n)$ 이다.
- thread는 각자 독립적이기 때문에, 서로의 지역변수는 따로따로이다.
- thread 동시성 충돌을 피하기 위해 사용하는 함수
 1. mutex lock
 2. semaphore
- mutex lock 사용법