

# 1. Linux 및 개발도구

# 1. Linux 및 개발도구

---

## 1.1 Linux 소개

1.2 컴파일 (gcc)

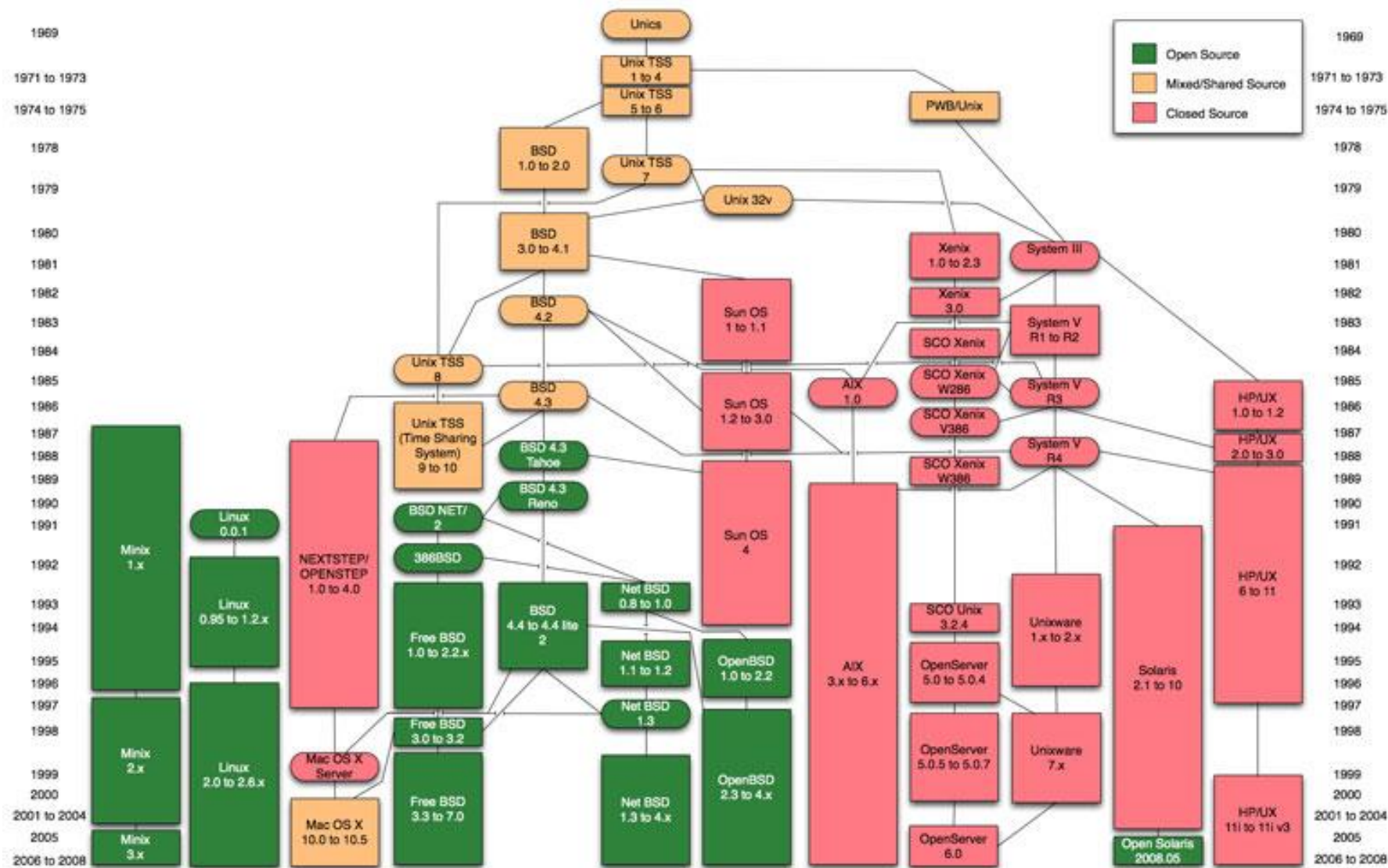
1.3 디버깅 (gdb)

1.4 빌드 (make)

---

- 유닉스 OS
  - 범용 다중 사용자 방식의 시분할 운영체제
  - 70년대 초반 AT&T Bell Lab에서 Ken Thompson, Dennis Ritchie, Douglas McIlroy등이 주축이 되어 개발
  - 초기 유닉스는 상업적으로 판매될 수 없어 소스코드와 함께 버클리 대학에 무상 제공 → BSD 계열로 발전
  - AT&T 유닉스는 SysV 계열로 발전
  - Solaris, HP-UX, AIX, SCO Unix, Ultrix, NeXTSTEP등 다양한 변종으로 발전
  - 표준화의 필요성 → POSIX

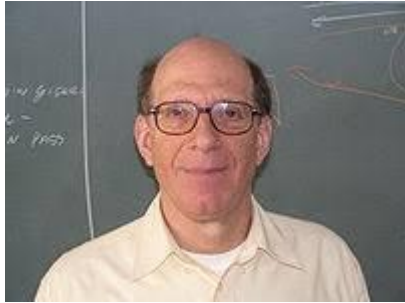
- Portable Operating System Interface
  - IEEE(Institute Of Electrical and Electronic Engineers)에서 제안
  - 소스레벨 호환성을 보장하는 유닉스
  - 대부분의 단체, 회사들이 POSIX를 지원하며 대부분의 UNIX도 POSIX compliant



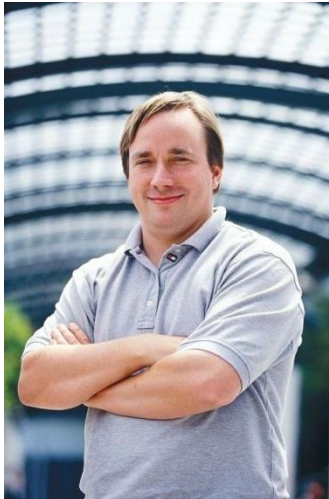


- Richard Stallman
  - Free Software Movement의 중심인물
  - GNU project와 FSF(Free Software Foundation) 설립자
  - Emacs, GDB, GCC등을 개발
  - Copyleft를 주장 – 지식과 정보는 소수에게 독점되어서는 안되며, 모든 사람에게 열려 있어야 함

- GNU (Gnu's Not Unix)
  - 80년대 초반 Richard Stallman에 의해 시작
  - GPL(General Public License)
    - GPL에 의거한 모든 소프트웨어는 무료
    - 변경 사항을 포함해 재 판매 하는 것은 허용하나 소스코드를 공개해야 함
    - 프로그래머는 자신의 소프트웨어로 인해 발생할 수 있는 어떠한 위험이나 손해에 대한 법률적 책임이 없음
  - Gcc, emacs 등을 리눅스에 포팅
  - BSD의 많은 유용한 유틸리티를 리눅스에 포함하게 하는 계기가 됨
- 리눅스도 GPL에 의거하여 배포됨



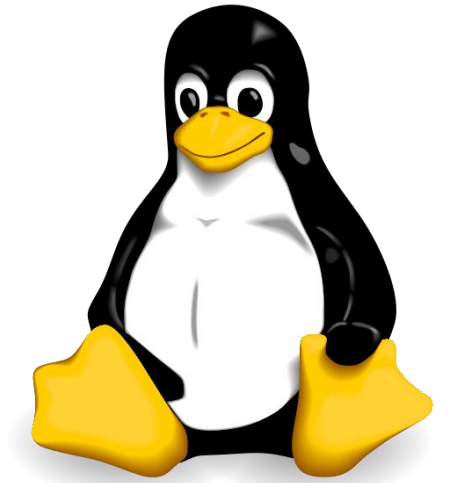
- 앤드루 타넨바움
  - 미닉스 (MIni-uNIX) 커널을 개발
  - 암스테르담 자유대학교에서 박사 과정 학생들의 연구를 지도
  - 미닉스는 리눅스 커널에 대한 리누스 토르발스의 영감의 원천



- Linus Torvalds
  - 최초 리눅스 커널을 개발
  - BDFL(en Benevolent Dictator For Life) 자비로운 종신독재자 중 한 명 – 저명한 오픈소스 개발자에게 부여되는 명예타이틀
  - 리눅스 커널 개발 최고 설계자
  - 현재 프로젝트 코디네이터로 활동



- 1991년 핀란드 헬싱키 대학의 Linus Torvalds가 개발된 오픈소스 OS
- 1991년 9월 17일 0.01 버전이 인터넷을 통해 공개
- POSIX compliant
- 뉴스그룹을 통해 수많은 개발자들이 개발에 동참 → 빠르게 버그 패치가 이루어지며 안정화
- 1994년 1.0 버전 발표
- FSF의 수많은 UNIX 유틸리티가 포팅 되어 포함됨 (gcc, emacs, ...)
- Bazaar model(시장 모델)로 개발됨
- 리눅스 기반의 다양한 distribution이 개발됨 → RedHat, Ubuntu, Slackware, Fedora 등



- 1999년 커널 버전 2.2를 거쳐 버전 2.4가 발표
  - 수많은 서버 등에서 사용되기 시작
- 버전 2.6부터 리얼타임 기능이 추가되고 안정화 되면서 수 많은 관련 산업에 적용 중
- 서버 및 임베디드 시스템 분야에서 많은 발전을 하고 있음

- 현재 널리 사용중인 오픈 소스 라이선스

종 류	내 용
GPL (General Public License)	<ul style="list-style-type: none"><li>- 대표적인 OSS license</li><li>- 오픈 소스 코드를 수정하는 경우 GPL에 의해 소스 코드를 제공해야 함</li><li>- 제한이 엄격</li></ul>
LGPL (Lesser GPL)	<ul style="list-style-type: none"><li>- 라이브러리는 공유하지만 개발된 제품에 대해서는 소스를 공개하지 않고 상용 SW로 판매가 가능</li></ul>
BSD, Apache License	<ul style="list-style-type: none"><li>- 아무런 제한 없는 license</li><li>- 소스코드 수정 후 소스 공개하지 않고 판매할 수 있음</li></ul>
MPL (Mozilla Public License)	<ul style="list-style-type: none"><li>- Netscape 브라우저의 소스코드를 공개하기 위해 개발된 license로 소스코드와 실행파일을 분리하여 취급</li></ul>

# 1. Linux 및 개발도구

---

1.1 Linux 소개

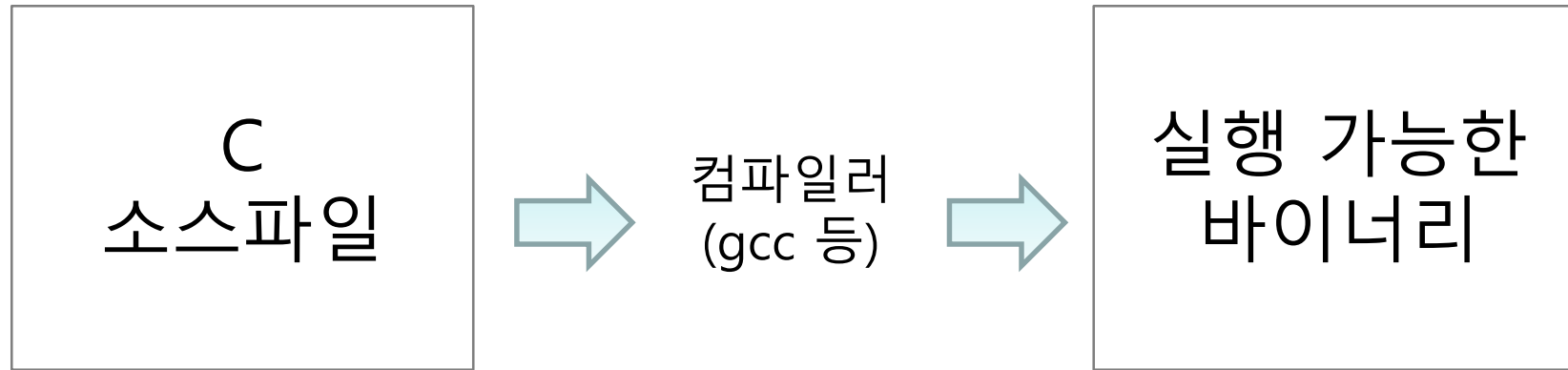
**1.2 컴파일 (gcc)**

1.3 디버깅 (gdb)

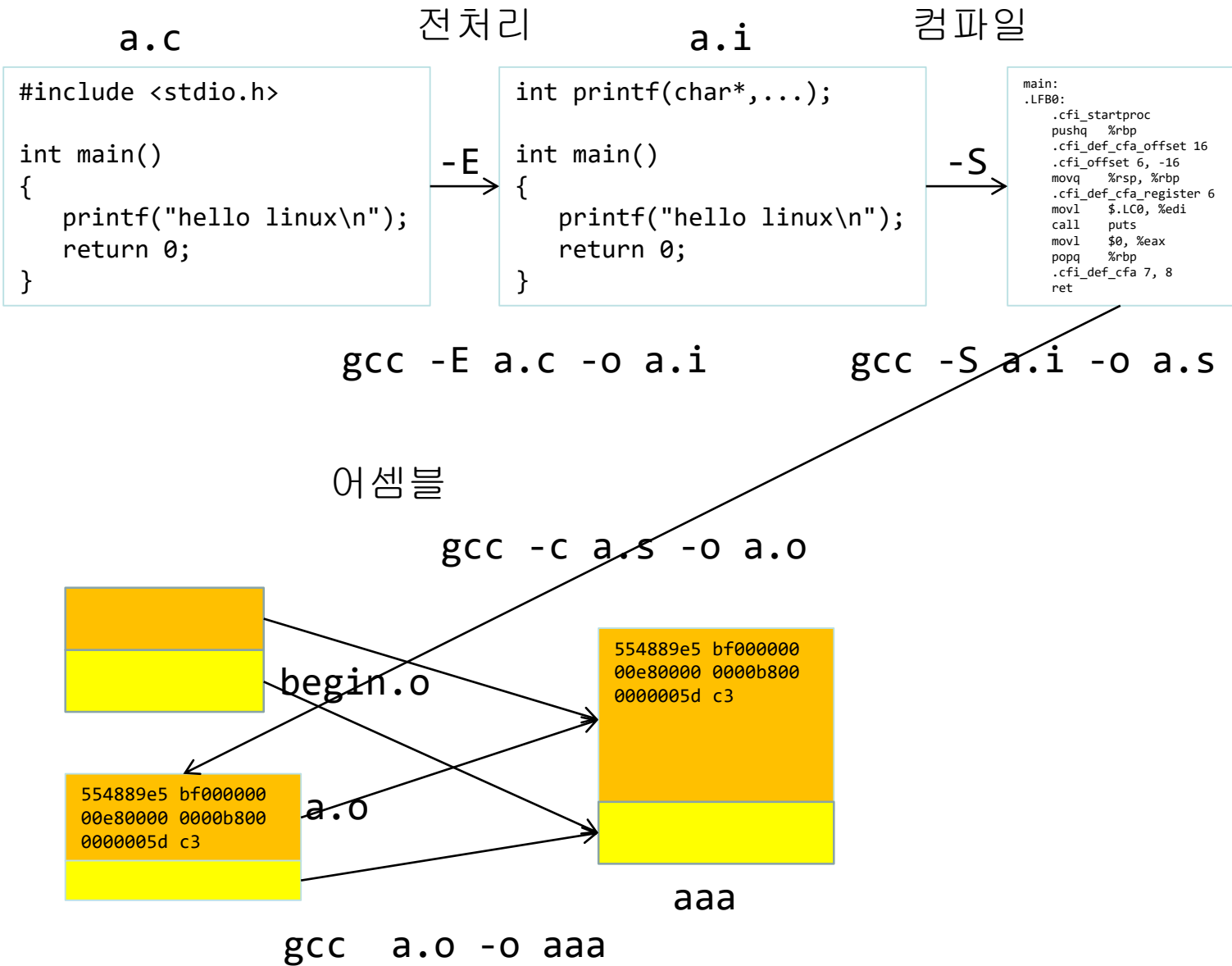
1.4 빌드 (make)

---

- 소스코드를 실행 가능한 파일로 번역하는 작업



- 일반적으로 컴파일이라고 통칭하지만 실제로는 여러 단계의 과정을 거침



- GNU에서 개발한 오픈소스 C 컴파일러
- 현재는 C 뿐 아니고 C++, Fortran, ADA, Java 등 다양한 언어의 컴파일이 가능
- Richard Stallman에 의해 개발되었고 GNU 시스템의 주요 구성요소
- 멀티 플랫폼에서 사용 가능하고 크로스 컴파일러로 사용할 수도 있음
  
- 기본 옵션
  - -g
  - -c
  - -o *{executable name}*

- UNIX/Linux 컴파일러는 관습적으로 컴파일을 하면 소스파일 이름에 관계 없이 실행파일 이름은 무조건 a.out으로 생성됨
- 출력될 실행파일 이름을 지정하고 싶으면 -o 옵션을 사용
- -o 뒤에 원하는 실행파일 이름을 지정함 ( o 옵션은 Target 파일 명을 지정하는데 사용함 )





- 실행파일을 디버깅 하려고 하면 실행파일에 디버깅 정보(심볼 이름들...)가 포함되어 있어야 함
- -g 옵션을 사용하면 실행파일에 디버깅 정보가 추가되어 디버거를 사용할 수 있음 → 단 실행 파일 크기가 커짐
- 디버깅이 끝나고 배포용 실행파일을 만들 때는 반드시 -g 옵션을 제거하고 컴파일 해야 함

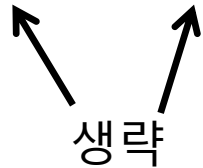
```
$ gcc -o hello hello.c
$ gcc -g -o hello1 hello.c
$ ls -l
-rwxr-xr-x 1 root root 7161 Apr  8 11:03 hello
-rw-r--r-- 1 root root  174 Apr  8 11:03 hello.c
-rwxr-xr-x 1 root root 8161 Apr  8 11:03 hello1
$
```

- 링크할 라이브러리를 지정
- 정적(static) 라이브러리 이름에서 앞쪽 부분의 'lib'와 뒤쪽 부분의 '.a'를 제외한 이름 부분만 -l 옵션 뒤에 공백 없이 써줌
- 즉 lib**m**.a (math library)를 링크해야 하는 경우 '-l**m**' 이라고 써 줌

lib**m**.a



\$ gcc -l**m** ...



생략

lib**rt**.a



\$ gcc -l**rt** ...

- Include 할 헤더파일을 찾을 디렉토리를 지정
- 소스코드에서 `#include <...>` 로 헤더파일을 추가하는 경우 `C_INCLUDE_PATH` 환경변수에 지정된 디렉토리에서 헤더파일을 찾아 추가함 (일반적으로 `/usr/include`)
- `C_INCLUDE_PATH` 환경변수에 지정되어 있지 않은 디렉토리에 헤더파일이 있는 경우 `'-I'` 옵션 뒤에 공백 없이 디렉토리 이름을 넣어주면 됨

```
#include <stdio.h>  
#include <myhdr.h>  ───────────────────> myhdr.h 가 /usr/local/myhdr 디렉토리에 있는 경우  
...
```

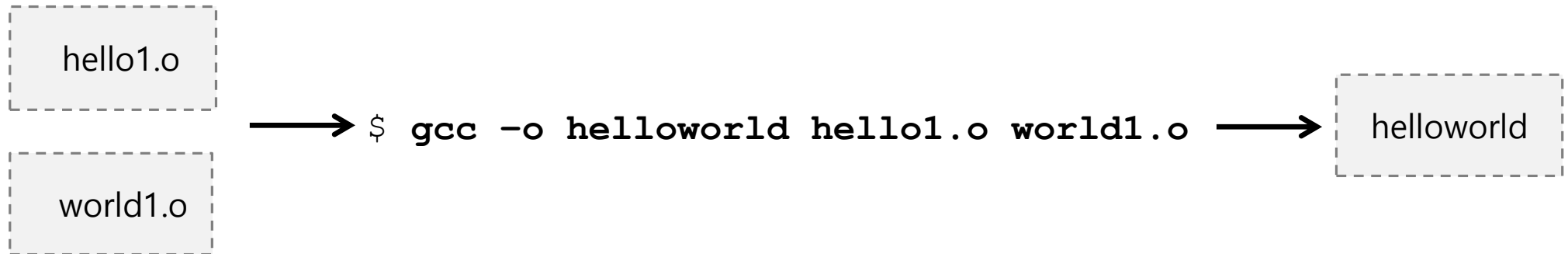
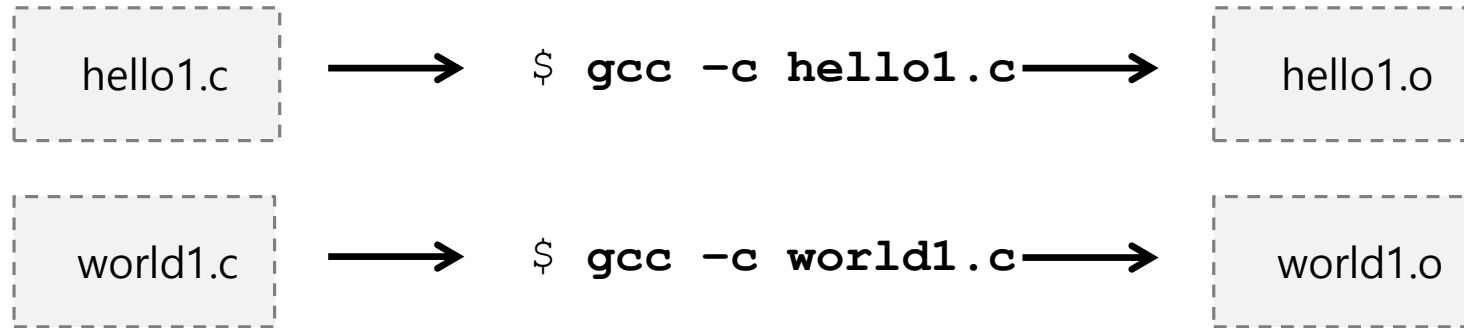
```
$ gcc -I/usr/local/myhdr ...
```

- 링크할 라이브러리 파일을 찾을 디렉토리를 지정
- 기본적으로 LIBRARY\_PATH 환경변수에 지정된 디렉토리에서 라이브러리 파일을 찾아 추가함 (일반적으로 /usr/lib)
- LIBRARY\_PATH 환경변수에 지정되어 있지 않은 디렉토리에 라이브러리 파일이 있는 경우 '-L' 옵션 뒤에 공백 없이 디렉토리 이름을 넣어주면 됨

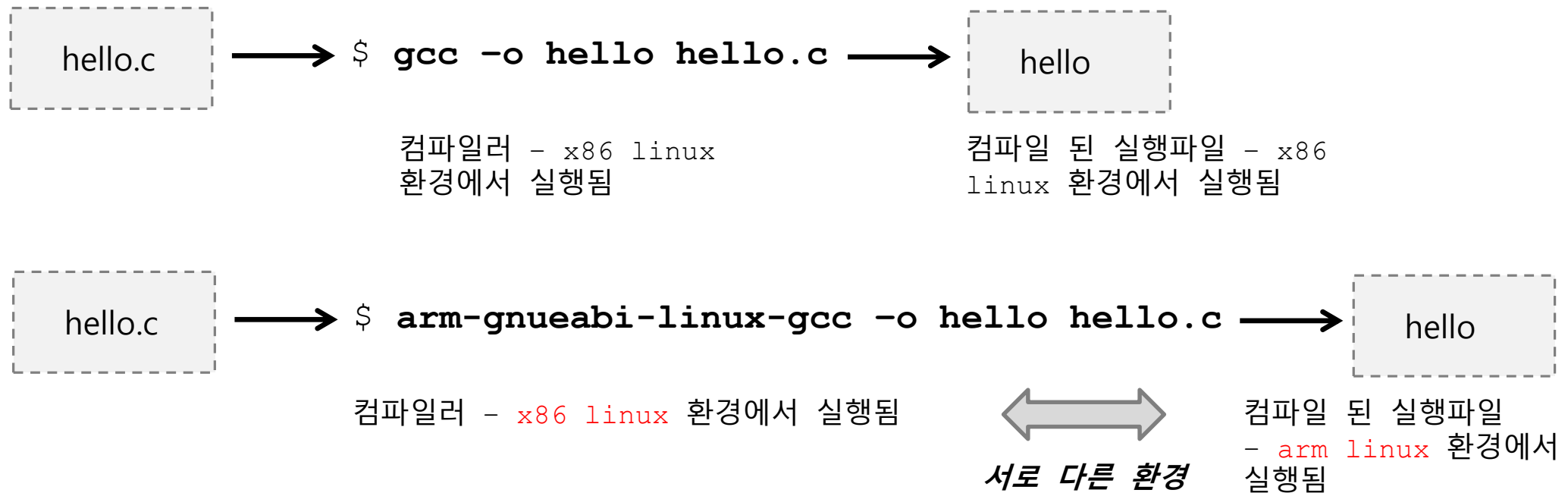
libmylib.a 가 /usr/local/mylib 디렉토리에 있는 경우 이 라이브러리를 링크하려면

```
$ gcc -L/usr/local/mylib -lmylib ...
```

- 프로그램이 여러 개의 소스파일로 구성되어 있을 때 분할컴파일을 위해 오브젝트 파일만 만들 때 사용



- 컴파일러가 실행되는 플랫폼(host)이 아닌 다른 플랫폼(target)에서 실행 가능한 코드를 생성할 수 있는 컴파일러
- Gcc는 크로스 컴파일러로 만들수도 있음



# 1. Linux 및 개발도구

---

1.1 Linux 소개

1.2 컴파일 (gcc)

**1.3 디버깅 (gdb)**

1.4 빌드 (make)

---

- GNU Debugger
- 프로그램 실행 중 특정 위치에서 어떤 동작을 하고 있는가를 감시할 수 있게 해 줌
- Segmentation fault 등의 에러도 gdb를 사용하면 쉽게 해결할 수 있음
- 온라인 메뉴얼 <http://sourceware.org/gdb/current/onlinedocs/gdb.toc>



- 프로그램 컴파일 시 '-g' 옵션 추가

```
$ cat -n hello.c
 1      #include <stdio.h>
 2
 3      int main(int argc, char **argv)
 4      {
 5          int i = 0;
 6          int j = 0;
 7
 8          i=3;
 9          j=4;
10          i=i*j+2;
11          printf("Hello, world! %d, %d\n", i, j);
12
13          return i;
14      }
$ gcc -o hello hello.c
$ ls -l
total 12
-rwxr-xr-x 1 root root 7161 Jan 15 10:12 hello
-rw-r--r-- 1 root root 144 Jan 15 10:11 hello.c
$ gcc -g -o hello hello.c
$ ls -l
total 12
-rwxr-xr-x 1 root root 8157 Jan 15 10:14 hello
-rw-r--r-- 1 root root 144 Jan 15 10:11 hello.c
$
```

- gdb [디버깅 할 실행파일 이름]

```
$ gdb hello  
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04  
.....  
Reading symbols from /root/sp/hello...done.  
(gdb)
```

OR

- gdb

```
$ gdb  
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04  
.....  
<http://bugs.launchpad.net/gdb-linaro/>.  
(gdb) file hello  
Reading symbols from /root/sp/hello...done.  
(gdb)
```

- run (r)

```
$ gdb hello
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
.....
Reading symbols from /root/sp/hello...done.
(gdb) run
Starting program: /root/sp/hello
Hello, world! 14, 4
[Inferior 1 (process 2424) exited with code 016]
(gdb)
```

프로그램 실행 결과

← 프로그램 리턴값(8진수)

- 실행시 에러 발생

```
$ cat -n seg.c
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  char *gPtr;
 5
 6  int main(int argc, char **argv)
 7  {
 8      strcpy(gPtr, "abcd");
 9      printf("%s\n", gPtr);
10
11      return 0;
12  }
```

```
$ gcc -g -o seg seg.c
```

```
$ gdb seg
```

```
.....
```

```
(gdb) r
```

```
Starting program: /root/sp/seg
```


```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x080483e9 in main (argc=1, argv=0xbffff444) at seg.c:8
```

```
8      strcpy(gPtr, "abcd");
```

```
(gdb)
```

에러가 발생한 위치  
seg.c 8번 라인



- gdb 실행할 때 넘겨줌

```
$ cat hello1.c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("no. of args = %d\n", argc);

    return argc;
}
$ gcc -g -o hello1 hello1.c
$ ./hello1 a b c
no. of args = 4
$ gdb --args hello1 a b c
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
.....
Reading symbols from /root/sp/hello1...(no debugging symbols found)...done.
(gdb) r
Starting program: /root/sp/hello1 a b c
no. of args = 4
[Inferior 1 (process 2434) exited with code 04]
(gdb)
```

- gdb 내에서 프로그램 실행할 때 넘겨줌

```
$ gdb hello1
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
.....
Reading symbols from /root/sp/hello1...(no debugging symbols found)...done.
(gdb) r a b c
Starting program: /root/sp/hello1 a b c
no. of args = 4
[Inferior 1 (process 2440) exited with code 04]
(gdb)
```

- 프로그램 실행 도중 특정 위치에서 실행을 멈춤
- 원하는 개수 만큼의 breakpoint를 설정할 수 있음

```
(gdb) break [line number]
```

```
(gdb) break [filename]:[line number]
```

```
(gdb) break [function name]
```

```
$ cat -n test1.c
 1  #include <stdio.h>
 2
 3  void swap(int, int);
 4
 5  int main(int argc, char **argv)
 6  {
 7      int i, j;
 8
 9      i=9;
10      j=17;
11      printf("Original value: i = %d, j = %d\n", i, j);
12      swap(i, j);
13      printf("Swapped value: i = %d, j = %d\n", i, j);
14
15      return 0;
16  }
17
18  void swap(int a, int b)
19  {
20      int tmp;
21
22      tmp = a;
23      a = b;
24      b = tmp;
25  }
$ gcc -g -o test1 test1.c
```



```
$ gdb test1
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
.....
Reading symbols from /root/sp/test1...done.
(gdb) break 12      ←----- 12번 라인에 breakpoint 설정
Breakpoint 1 at 0x804841a: file test1.c, line 12.
(gdb) b 13          ←----- 13번 라인에 breakpoint 설정
Breakpoint 2 at 0x804842e: file test1.c, line 13.
(gdb) info break    ←----- breakpoint 목록 확인
Num      Type             Disp Enb Address      What
1         breakpoint       keep y   0x0804841a in main at test1.c:12
2         breakpoint       keep y   0x0804842e in main at test1.c:13
(gdb) run
Original value: i = 9, j = 17

Breakpoint 1, main (argc=1, argv=0xbffff444) at test1.c:12
12          swap(i, j);
(gdb) continue    ←----- 멈춘 프로그램의 실행을 재개
Continuing.

Breakpoint 2, main (argc=1, argv=0xbffff444) at test1.c:13
13          printf("Swapped value: i = %d, j = %d\n", i, j);
(gdb) c
Continuing.
Swapped value: i = 9, j = 17
[Inferior 1 (process 2664) exited normally]
(gdb)
```

```
$ gdb test1
.....
(gdb) break 12
Breakpoint 1 at 0x804841a: file test1.c, line 12.
(gdb) run
Original value: i = 9, j = 17

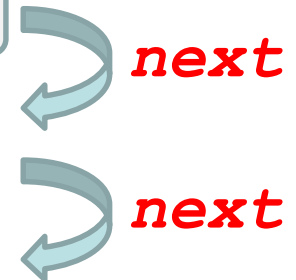
Breakpoint 1, main (argc=1, argv=0xbffff444) at test1.c:12
12             swap(i, j);
(gdb) next
13             printf("Swapped value: i = %d, j = %d\n", i, j);
(gdb)
```

```
11 printf("Original value: i = %d, j = %d\n", i, j);
```

```
12 swap(i, j);
```

여기에서 실행이 중단

```
13 printf("Swapped value: i = %d, j = %d\n", i, j);
```



```
$ gdb test1
.....
(gdb) break 12
Breakpoint 1 at 0x804841a: file test1.c, line 12.
(gdb) run
Original value: i = 9, j = 17

Breakpoint 1, main (argc=1, argv=0xbffff444) at test1.c:12
12          swap(i, j);
(gdb) step
swap (a=9, b=17) at test1.c:22
22          tmp = a;
(gdb)
```

```
12  swap(i, j);
```

여기에서 실행이 중단

```
13  printf("Swapped value: i = %d, j = %d\n", i, j);
```

```
18  void swap(int a, int b)
```

```
19  {
```

```
20      int tmp;
```

```
21
```

```
22      tmp = a;
```

```
23      a = b;
```

*step*

*step*

- 변수 또는 수식의 값 출력
- print/[format character]로 값의 출력 포맷 변경 가능

/x	hexadecimal	/o	octal
/d	decimal	/u	unsigned decimal
/t	binary	/f	float
/a	address	/i	instruction
/c	character	/s	string

- 변수값 뿐 아니라 수식을 넣을 수도 있음

```
$ cat -n test2.c
 1      #include <stdio.h>
 2
 3      int sum(int, int);
 4
 5      int main(int argc, char **argv)
 6      {
 7          int i;
 8          int j;
 9          int r;
10
11          i = atoi(argv[1]);
12          j = atoi(argv[2]);
13          printf("Add from %d to %d\n", i, j);
14          r = sum(i, j);
15          printf("Sum = %d\n", r);
16
17          return 0;
18      }
19
20      int sum(int from, int to)
21      {
22          int i;
23          int total = 0;
24
25          for (i=from; i<to; i++) {
26              total += i;
27          }
28          return total;
29      }
$ gcc -g -o test2 test2.c
```

```
$ gdb test2
```

```
.....
```

```
(gdb) b 13
```

```
Breakpoint 1 at 0x8048445: file test2.c, line 13.
```

```
(gdb) print i
```

```
No symbol "i" in current context.
```

←----- 왜 i의 값을 찍을 수 없는가?

```
(gdb) r 1 100
```

```
Starting program: /root/sp/test2 1 100
```

```
Breakpoint 1, main (argc=3, argv=0xbffff434) at test2.c:13
```

```
warning: Source file is more recent than executable.
```

```
13          printf("Add from %d to %d\n", i, j);
```

```
(gdb) print i
```

```
$1 = 1
```

```
(gdb) p j
```

```
$2 = 100
```

```
(gdb) p sum(1,10)
```

```
$3 = 45
```

```
(gdb) p sum(i,j)
```

```
$4 = 4950
```

1~100까지의 합이 4950?

(1+100)\*100/2 = **5050!!!**

```
(gdb) set j=101
```

```
(gdb) p sum(i,j)
```

```
$5 = 5050
```

```
(gdb) quit
```

```
A debugging session is active.
```

```
Inferior 1 [process 2113] will be killed.
```

```
Quit anyway? (y or n) y
```

```
$
```

- set 명령으로 변수 값 변경 가능
  - set *[변수이름] = [새 값]*

```
(gdb) set count = 10
```

- 특정 메모리 내용도 변경 가능
  - set *{[타입]}[주소] = [새 값]*

```
(gdb) set {int}0x842afa00 = 4
```

- 메모리 내용 확인 – examine (x)
- 레지스터 값 확인 – info reg

```
$ gdb test2
.....
(gdb) b 13
Breakpoint 1 at 0x8048445: file test2.c, line 13.
(gdb) r 1 4
Starting program: /root/sp/test2 1 4

Breakpoint 1, main (argc=3, argv=0xbffff434) at test2.c:13
13          printf("Add from %d to %d\n", i, j);
(gdb) x $pc
0x8048445 <main+49>: 0x0485a0b8
(gdb) x 0x8048445
0x8048445 <main+49>: 0x0485a0b8
(gdb) x/i 0x8048445
=> 0x8048445 <main+49>:          mov     $0x80485a0,%eax
(gdb) x/4x 0x8048445
0x8048445 <main+49>: 0x0485a0b8 0x24548b08 0x24548918 0x24548b08
(gdb) info reg
eax                0x4      4
ecx                0x0      0
.....
gs                 0x33     51
(gdb)
```



- 변수값이 액세스 되는 시점에 실행을 중단시킴
  - watch [변수 이름]

```
(gdb) watch count
```

- 변수 scope가 맞아야 watchpoint를 설정할 수 있음. 즉 글로벌 변수는 아무 곳에서도 watchpoint를 설정할 수 있지만 로컬 변수는 해당 변수의 scope에서만 watchpoint 설정 가능함
- watch, awatch, rwatch가 있음

```
$ gdb test2
.....
(gdb) watch i
No symbol "i" in current context.
(gdb) b 13
Breakpoint 1 at 0x8048445: file test2.c, line 13.
(gdb) r 1 4
Starting program: /root/sp/test2 1 4

Breakpoint 1, main (argc=3, argv=0xbffff434) at test2.c:13
13             printf("Add from %d to %d\n", i, j);
(gdb) n
Add from 1 to 4
14             r = sum(i, j);
(gdb) s
sum (from=1, to=4) at test2.c:23
23             int total = 0;
(gdb) watch i
Hardware watchpoint 2: i
(gdb) c
Continuing.
Hardware watchpoint 2: i

Old value = 134514080
New value = 1
0x080484a9 in sum (from=1, to=4) at test2.c:25
25             for (i=from; i<to; i++) {
```

```
(gdb) c
Continuing.
Hardware watchpoint 2: i

Old value = 1
New value = 2
0x080484b5 in sum (from=1, to=4) at test2.c:25
25             for (i=from; i<to; i++) {
(gdb) c
Continuing.
Hardware watchpoint 2: i

Old value = 2
New value = 3
0x080484b5 in sum (from=1, to=4) at test2.c:25
25             for (i=from; i<to; i++) {
(gdb) c
Continuing.
Hardware watchpoint 2: i

Old value = 3
New value = 4
0x080484b5 in sum (from=1, to=4) at test2.c:25
25             for (i=from; i<to; i++) {
(gdb) n ←-----
28             return total;
(gdb)
```

i=4일 때 total += i;를 실행하지 않고  
바로 루프를 종료함

- info breakpoint (i b)
  - 현재 설정되어 있는 breakpoint/watchpoint에 관한 정보를 보여 줌
- delete (d)
  - 설정되어 있는 breakpoint/watchpoint를 삭제
  - Breakpoint/watchpoint 번호를 지정하지 않으면 모든 breakpoint/watchpoint를 삭제함

```
$ gdb test2
(gdb) b 13
Breakpoint 1 at 0x8048445: file test2.c, line 13.
(gdb) b 15
Breakpoint 2 at 0x804847a: file test2.c, line 15.
(gdb) b 25
Breakpoint 3 at 0x80484a3: file test2.c, line 25.
(gdb) info breakpoint
Num      Type             Disp Enb Address      What
1        breakpoint       keep y   0x08048445 in main at test2.c:13
2        breakpoint       keep y   0x0804847a in main at test2.c:15
3        breakpoint       keep y   0x080484a3 in sum at test2.c:25
(gdb) delete
Delete all breakpoints? (y or n) n
(gdb) d 2
(gdb) info break
Num      Type             Disp Enb Address      What
1        breakpoint       keep y   0x08048445 in main at test2.c:13
3        breakpoint       keep y   0x080484a3 in sum at test2.c:25
(gdb)
```

- 특정 조건이 만족되었을 때에만 breakpoint가 동작
  - Breakpoint를 설정할 때 조건을 추가할 수 있음
  - `break [위치] if [조건]`

```
(gdb) b 25 if i==4
```

[illegible]

- 다음과 같은 구조체가 정의되어 있고

```
struct entry {  
    int key;  
    char *name;  
    float price;  
    long serial_number;  
};
```

- 아래 라인이 실행된 다음 실행이 중단되었을 때

```
struct entry *e1 = [something];
```



- 구조체의 메모리 주소를 볼 때

```
(gdb) print e1
```

- 특정 필드를 참조할 때

```
(gdb) print e1->key
```

```
(gdb) print e1->name
```

```
(gdb) print e1->price
```

```
(gdb) print e1->serial_number
```

- C에서처럼 '->' 대신 '\*', '.' 사용 가능

```
(gdb) print (*e1).key
```

```
(gdb) print (*e1).name
```

```
.....
```

- 포인터가 참조하는 전체 구조체 내용을 볼 때

```
(gdb) print *e1
```

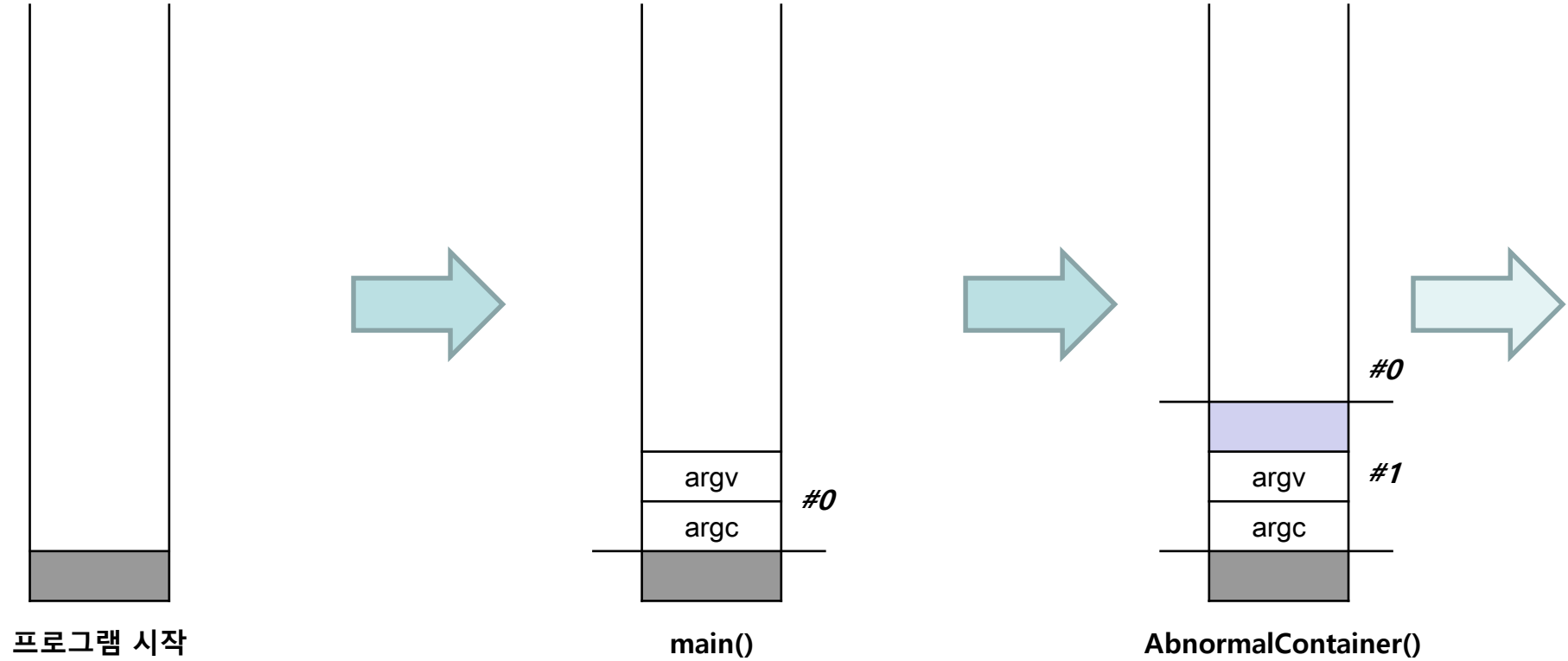
- 다음과 같이 사용할 수도 있음

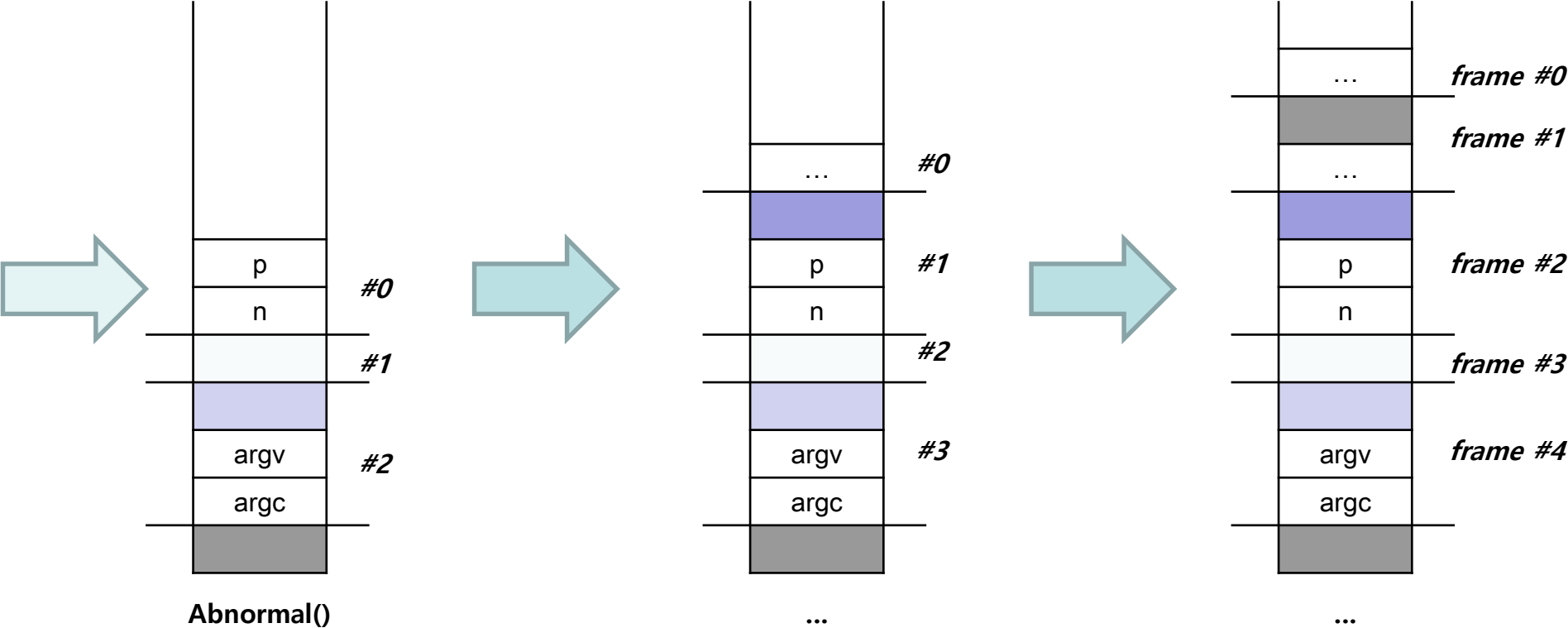
```
(gdb) print list_pre->next->next->next->data
```

- backtrace (bt)
  - 프로그램이 중단되었을 때의 스택 프레임을 출력
  - 어떤 과정을 거쳐 현재의 위치로 와 있는지를 확인할 수 있음
  - Segment fault가 발생했을 때 매우 유용

```
$ cat -n core_ex.c
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  void Abnormal()
 5  {
 6      int n = 1024;
 7      char *p = (char *)malloc(sizeof(char) * 1);
 8
 9      free(p);
10      free(p);          /* double free */
11  }
12
13  void AbnormalContainer()
14  {
15      Abnormal();
16  }
17
18  void Normal()
19  {
20      printf("normal function.\n");
21  }
22
23  int main(int argc, char **argv)
24  {
25      AbnormalContainer();
26      Normal();
27      return 0;
28  }
$ gcc -g -o core_ex core_ex.c
```

```
$ ulimit -c unlimited
$ ./core_ex
*** glibc detected *** ./core_ex: double free or corruption (fasttop): 0x09c4f008 ***
===== Backtrace: =====
.....
bfff75000-bfff96000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
$ ls -l core
-rw----- 1 root root 348160 Jan 20 13:41 core
$ gdb core_ex core
...
Core was generated by `./core_ex'.
Program terminated with signal 6, Aborted.
#0  0xb77b6424 in __kernel_vsyscall ()
(gdb) bt
#0  0xb77b6424 in __kernel_vsyscall ()
#1  0xb76251df in raise () from /lib/i386-linux-gnu/libc.so.6
#2  0xb7628825 in abort () from /lib/i386-linux-gnu/libc.so.6
#3  0xb766239a in ?? () from /lib/i386-linux-gnu/libc.so.6
#4  0xb766cee2 in ?? () from /lib/i386-linux-gnu/libc.so.6
#5  0x08048466 in Abnormal () at core_ex.c:10
#6  0x08048473 in AbnormalContainer () at core_ex.c:15
#7  0x08048494 in main (argc=1, argv=0xbff93734) at core_ex.c:25
(gdb)
```





- frame (f)
  - 기본적으로 항상 frame #0을 사용
  - 다른 frame으로 포커스를 변경할 때 사용

```
$ gdb core_ex
(gdb) b 9
Breakpoint 1 at 0x8048450: file core_ex.c, line 9.
(gdb) r
Starting program: /root/sp/core_ex

Breakpoint 1, Abnormal () at core_ex.c:9
9          free(p);
(gdb) bt
#0  Abnormal () at core_ex.c:9
#1  0x08048473 in AbnormalContainer () at core_ex.c:15
#2  0x08048494 in main (argc=1, argv=0xbffff444) at core_ex.c:25
(gdb) p argc
No symbol "argc" in current context.
(gdb) f 2
#2  0x08048494 in main (argc=1, argv=0xbffff444) at core_ex.c:25
25          AbnormalContainer();
(gdb) print argc
$1 = 1
```

argc는 main()함수의  
argument라 frame #0 에  
서는 볼 수 없음

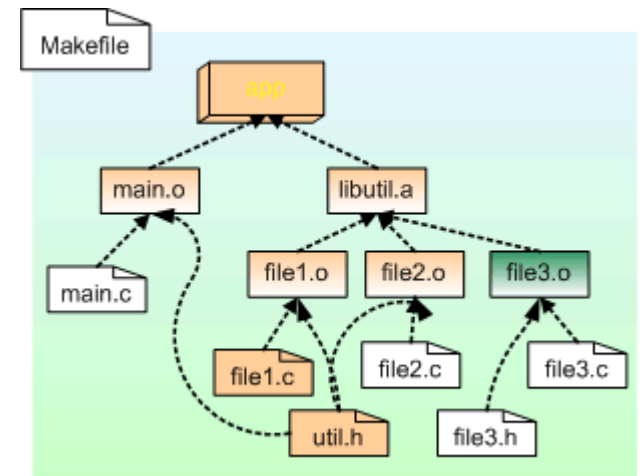
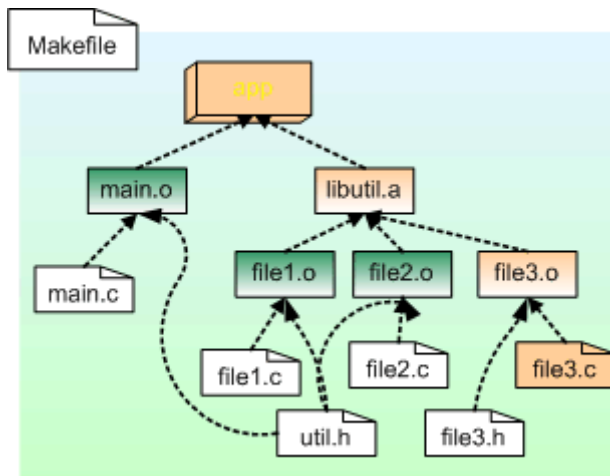
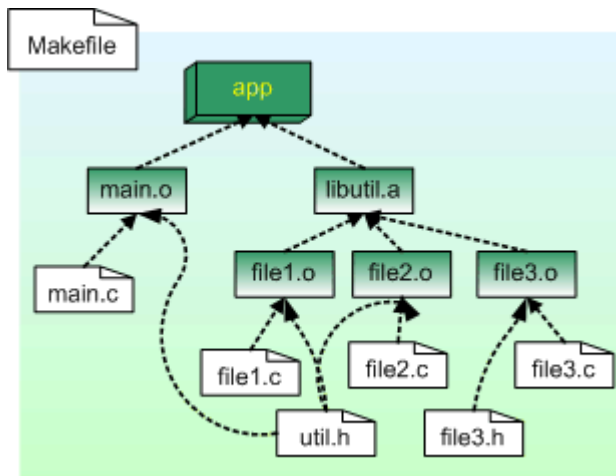


# 1. Linux 및 개발도구

---

- 1.1 Linux 소개
  - 1.2 컴파일 (gcc)
  - 1.3 디버깅 (gdb)
  - 1.4 빌드 (make)**
-

- 프로그램 빌드 자동화 소프트웨어
- 여러 파일들 간의 의존성과 각 파일을 위한 명령어를 정의한 Makefile을 해석하여 프로젝트(실행파일 또는 라이브러리)를 빌드
- 따로 옵션을 주지 않으면 default로 Makefile 또는 makefile 이라는 이름의 파일을 찾고, 다른 이름을 준 경우는 -f 옵션으로 파일 이름을 지정함



```
$ cat -n hello.c
 1      #include <stdlib.h>          /* EXIT_* */
 2      #include "hello_api.h"      /* sayhello */
 3
 4      int main(int argc, char **argv) {
 5          sayhello();
 6          return EXIT_SUCCESS;
 7      }
$ cat -n hello_func.c
 1      #include <stdio.h>          /* printf */
 2      #include "hello_api.h"      /* sayhello declaration */
 3
 4      void sayhello(void) {
 5          printf("Hello world!\n");
 6      }
$ cat -n hello_api.h
 1      #ifndef INCLUDE_HELLO_API_H
 2      #define INCLUDE_HELLO_API_H
 3
 4      /* Function to print out "Hello world!\n". */
 5      extern void sayhello(void);
 6
 7      #endif
 8      /* ifndef INCLUDE_HELLO_API_H */
$
```

- 하나의 파일에 구현

main.c

```
int add(int, int );

int main()
{
    int ret;
    ret = add(3,4);
    return 0;
}

int add(int a, int b)
{
    return a+b;
}
```

컴파일

```
# gcc main.c -o aaa
```

단점 :

- 여러 파일에 분산 구현

main.h

```
int add(int, int );
```

main.c

```
#include "main.h"
int main()
{
    int ret;
    ret = add(3,4);
    return 0;
}
```

add.c

```
#include "main.h"
int add(int a, int b)
{
    return a+b;
}
```

분할컴파일

```
# gcc -c main.c
# gcc -c add.c
# gcc main.o add.o -o aaa
```

단점 :

- 여러 파일에 분산 구현

main.h

```
int add(int, int );
```

main.c

```
#include "main.h"
int main()
{
    int ret;
    ret = add(3,4);
    return 0;
}
```

add.c

```
#include "main.h"
int add(int a, int b)
{
    return a+b;
}
```

cc.sh

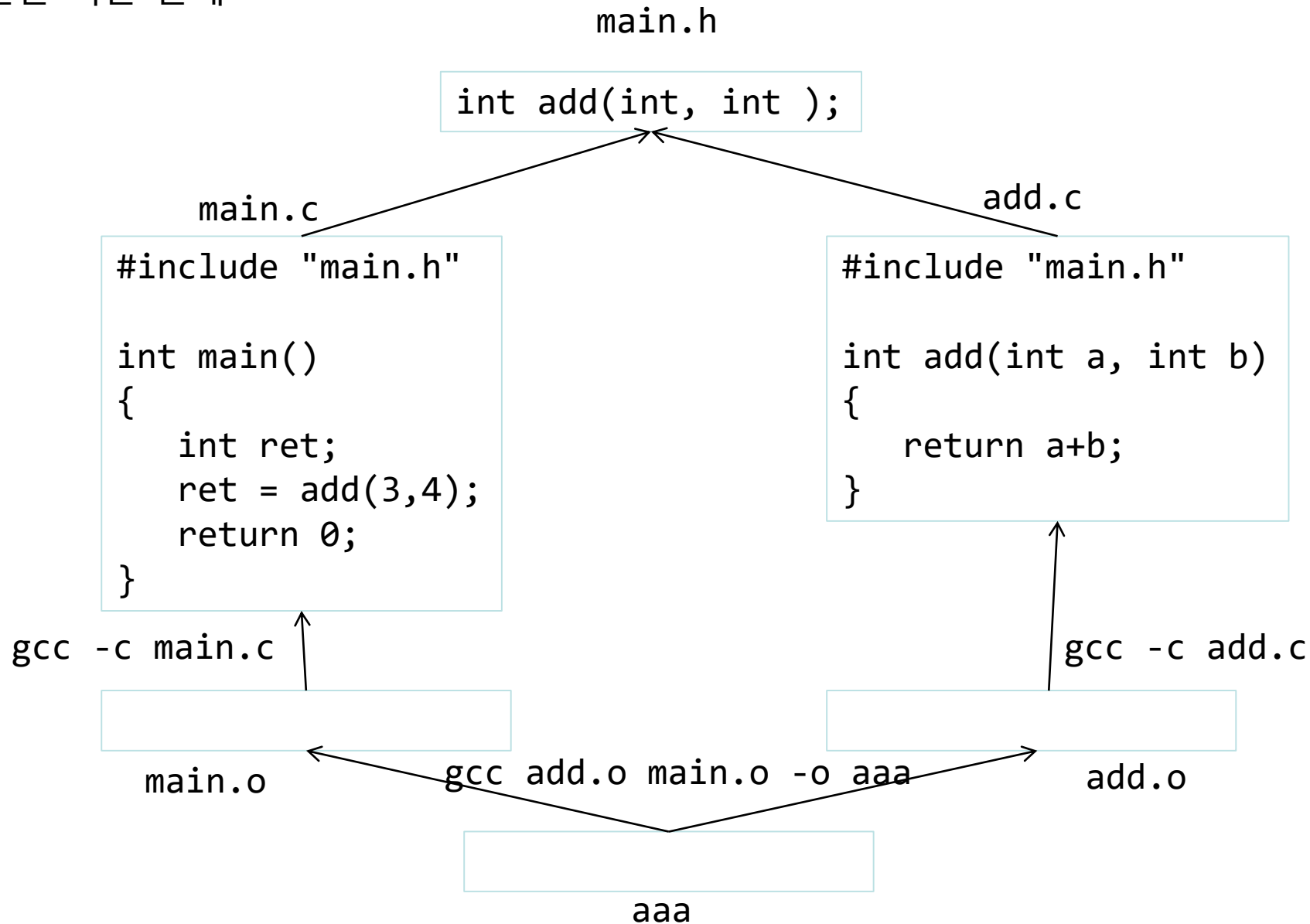
```
#!/bin/sh
set -x

gcc -c main.c
gcc -c add.c
gcc main.o add.o -o aaa
```

# sh cc.sh

단점 :

- 파일간 의존 관계



- 기본 구조

**Target: Dependencies**  
***TAB키* Commands**



※ 이 부분에 SPACE 키를 사용하면 안됨!!!

- Target – Command를 실행한 결과로 만들어 질 목적 파일
- Dependencies(Prerequisites) – Target을 만들 때 의존성(연관관계)를 규정. 이 부분에 나열된 파일이 수정되면 command를 수행해서 target을 다시 만듦. 즉 target 파일의 최종 수정 시간과 dependencies에 있는 파일들의 최종 수정 시간을 비교해서 command 수행여부를 결정
- Command(Recipe) – Target을 만들기 위한 실행해야 하는 명령. Command는 여러 줄이 될 수도 있음. (주의: Command 라인은 항상 TAB으로 시작되어야 함. 라인 앞의 공백에 SPACE 키를 사용하면 에러 발생)
- “#” 뒤에 오는 글자는 comment



- Command앞에 '@'을 붙여주면 그 명령은 화면에 출력되지 않음


```
$ cat -n Makefile
1      all:
2          echo hello
$ make
echo hello
hello
$
```



make가 실행한 명령에 의해 출력된 결과

명령 앞에 '@'이 붙어 있기 때문에 이 명령은 화면에 출력되지 않음

```
$ cat -n Makefile
1      all:
2          @echo hello
$ make
hello
$
```



make가 실행한 명령에 의해 출력된 결과

## Makefile

```
aaa : main.o add.o
<tab>gcc main.o add.o -o aaa

main.o : main.c
<tab>gcc -c main.c

add.o  : add.c
<tab>gcc -c add.c
```

# make

- LINUX 쉘 변수와 유사하게 동작
- 변수 정의는 일반 C 프로그램에서 변수값을 지정하는 것 처럼 하면 됨. 따로 변수 타입을 선언하지 않음
- 변수를 사용할 때는 '**\$(변수 이름)**' 또는 '**\${변수 이름}**'을 사용하면 됨

`PROG_NAME = myapp` ← PROG\_NAME이라는 이름의 변수를 선언하고 값(내용)을 넣어줌

`$(PROG_NAME): myapp.o util.o`

`gcc -o $(PROG_NAME) myapp.o util.o`

PROG\_NAME이라는 이름의 변수를 사용  
즉 이 부분이 변수의 내용인 'myapp'으로 치환됨 ...

- 두가지 타입의 변수가 있음
  - Simple – 변수의 값이 정의될 때 한번만 평가(evaluation)됨
  - Recursive – 변수의 값이 변수가 참조될 때마다 평가됨. 그러므로 변수를 정의할 때 그 변수가 변수값 부분에서 참조되면 안됨

```
$ cat -n Makefile
1      CC = gcc -Wall
2      CC = $(CC) -g
3      hello.o: hello.c hello_api.h
4          $(CC) -c hello.c -o hello.o
$ make
Makefile:2: *** Recursive variable `CC' references itself (eventually).  Stop.
$
```



- Simple 타입 변수를 사용

```
$ cat -n Makefile
1      CC := gcc -Wall
2      CC := $(CC) -g
3      hello.o: hello.c hello_api.h
4          $(CC) -c hello.c -o hello.o
$ make
gcc -Wall -g -c hello.c -o hello
$
```

변수를 선언할 때 '=' 대신 ':=' 를 사용하면 변수가 simple type의 변수가 됨

- 또는 '+='를 사용. 변수타입은 유지되고 내용의 뒤에 텍스트가 추가(append)됨

```
$ cat -n Makefile
1      CC = gcc -Wall
2      CC += -g
3      hello.o: hello.c hello_api.h
4          $(CC) -c hello.c -o hello.o
$ make
gcc -Wall -g -c hello.c -o hello
$
```

- 미리 정의되어 있는 매크로들이 존재 (make -p 를 입력하면 목록을 볼 수 있음)

```
$ make -p
# Copyright (C) 2006 Free Software Foundation, Inc.
...
OUTPUT_OPTION = -o $@
...
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
...
%.o: %.c
# commands to execute (built-in):
      $(COMPILE.c) $(OUTPUT_OPTION) $<
...
$
```

- 일반적으로 많이 사용되는 매크로

ASFLAGS =	← <i>as</i> 명령어의 옵션 세팅
AS = as	
CFLAGS =	← <i>gcc</i> 의 옵션 세팅
CC = cc (= gcc)	
CPPFLAGS =	← <i>g++</i> 의 옵션
CXX = g++	
LDLDFLAGS =	← <i>ld</i> 의 옵션 세팅
LD = ld	
LFLAGS =	← <i>lex</i> 의 옵션 세팅
LEX = lex	
YFLAGS =	← <i>yacc</i> 의 옵션 세팅
YACC = yacc	
MAKE_COMMAND = make	

## Makefile

```
TARGET = aaa
OBJS   = main.o add.o
CC      = gcc
CFLAGS = -c

$(TARGET) : $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)

main.o : main.c
    $(CC) $(CFLAGS) main.c

add.o : add.c
    $(CC) $(CFLAGS) add.c
```

# make

- 변수의 값을 지정하는 다른 값으로 대체하는 기능
- $\$(\text{변수이름}: a=b)$  또는  $\{ \text{변수이름}: a=b \}$
- 변수의 값에 있는 각 단어의 뒤쪽에 있는 a를 b로 교체한 값으로 대체

```
TESTSTRING = hello.c world.c
```

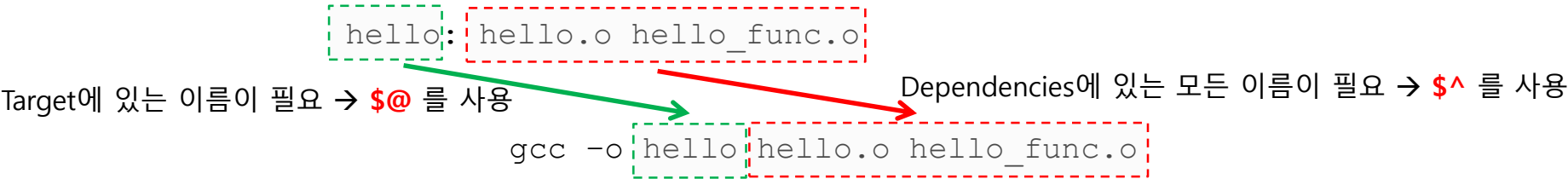
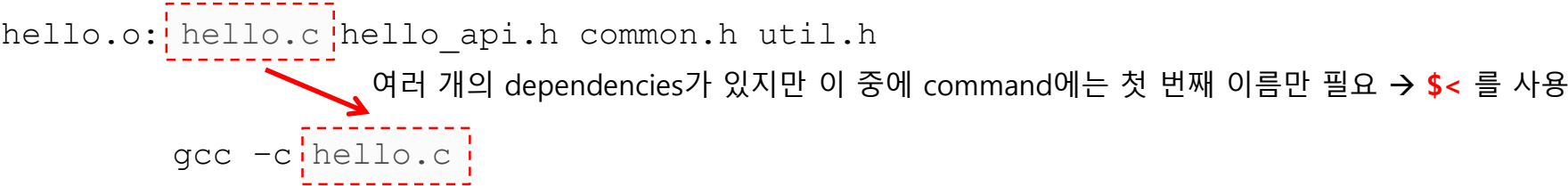
```
RESULT = $(TESTSTRING:.c=.o)
```

TESTSTRING 변수의 값(즉 hello.c world.c)에 있는 각 단어의 뒤쪽에 있는 .c 를 .o로 변경 → hello.o world.o 가 됨

'hello.c world.c'값을 가지고 있음

RESULT 변수값은 'hello.o world.o'가 됨

- \$< : Dependencies 중에 첫 번째 것으로 대체됨
- \$^ : Dependencies 전체로 대체됨
- \$@ : Target으로 대체됨
- \$\* : 확장자가 없는 target으로 대체됨
- \$? : Dependencies 중에 target보다 새로운 파일들 목록으로 대체됨





### Makefile

```
TARGET = aaa
OBJS    = main.o add.o
CC      = gcc
CFLAGS  = -c

$(TARGET) : $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)

main.o : main.c
    $(CC) $(CFLAGS) $<

add.o : add.c
    $(CC) $(CFLAGS) $<
```

# make

- 파일의 확장자에 따라 적절한 명령을 수행시키는 규칙

```
$ cat -n Makefile
1    hello: hello.o hello_func.o
2          gcc -o hello hello.o hello_func.o
3    hello.o: hello.c hello_api.h
4          gcc -c hello.c
5    hello_func.o: hello_func.c hello_api.h
6          gcc -c hello_func.c

$ make
gcc -c hello.c -o hello.o
gcc -c hello_func.c -o hello_func.o
gcc hello.o hello_func.o -o hello
$
```

```
$ cat -n Makefile
1    hello: hello.o hello_func.o
2          gcc -o hello hello.o hello_func.o
3    hello.o: hello.c hello_api.h
4
5    hello_func.o: hello_func.c hello_api.h
6
$ make
cc -c -o hello.o hello.c
cc -c -o hello_func.o hello_func.c
gcc hello.o hello_func.o -o hello
$
```

- 별도의 명령이 없어도 pre-defined macro에 정의되어 있는 suffix rule이 있기 때문에 자동으로 .c 로 끝나는 소스파일에서 .o 로 끝나는 오브젝트 파일 생성
- 두 가지 스타일로 규칙을 지정할 수 있음

```
.c.o:
    $(CC) $(CFLAGS) -c $<
```

*Old style*

```
%.o: %.c
    $(CC) $(CFLAGS) -c $<
```

*New style*

## Makefile

```
TARGET = aaa
OBJS    = main.o add.o
CC      = gcc
CFLAGS  = -c

$(TARGET) : $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)

.c.o :
    $(CC) $(CFLAGS) $<
```

## Makefile

```
TARGET = aaa
OBJS    = main.o add.o
CC      = gcc
CFLAGS  = -c

$(TARGET) : $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)

%.o: %.c
    $(CC) $(CFLAGS) $<
```

# make

## Makefile

```
TARGET = aaa
OBJS   = main.o add.o
CC      = gcc
CFLAGS = -c

$(TARGET) : $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)

.c.o :
    $(CC) $(CFLAGS) $<

clean :
    rm -f $(OBJS) $(TARGET)
```

```
# make
# make clean
```

## Makefile

```
TARGET = aaa
OBJS    = main.o add.o
CC      = gcc
CFLAGS  = -I../include -c

$(TARGET) : $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)

.c.o :
    $(CC) $(CFLAGS) $<

clean :
    rm -f $(OBJS) $(TARGET)
```

```
# ls -R
./include : main.h

./src : main.c add.c
```

```
# make
```

main.c

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

void * foo(void *data)
{
    printf("foo()\n");
    return 0;
}

int main()
{
    int ret;
    pthread_t thread;
    pthread_create( &thread , 0, foo, 0 );
    pthread_join( thread , 0 );
    return 0;
}
```

```
# make
gcc -I../include -c main.c
gcc main.o -o aaa
main.o: In function `main':
main.c:(.text+0x4b): undefined reference to `pthread_create'
main.c:(.text+0x5c): undefined reference to `pthread_join'
```

### Makefile

```
TARGET = aaa
OBJS    = main.o
CC      = gcc
CFLAGS  = -I../include -c
LFLAGS = -lpthread

$(TARGET) : $(OBJS)
    $(CC) $(OBJS) -o $(TARGET) $(LFLAGS)

.c.o :
    $(CC) $(CFLAGS) $<

clean :
    rm -f $(OBJS) $(TARGET)
```

main.c

```
#include <stdio.h>
#include <main.h>
#include <pthread.h>

void * foo(void *data)
{
    printf("foo()\n");
    return 0;
}

int main()
{
    int ret;
    pthread_t thread;
    pthread_create( &thread , 0, foo, 0 );
    pthread_join( thread , 0 );
    return 0;
}
```

```
# ar rcv ../lib/libmath.a add.o
```



## Makefile

```
TARGET = aaa
OBJS    = main.o
CC      = gcc
CFLAGS  = -I../include -c
LFLAGS  = -lpthread -L../lib -lmath

$(TARGET) : $(OBJS)
    $(CC) $(OBJS) -o $(TARGET) $(LFLAGS)

.c.o :
    $(CC) $(CFLAGS) $<

clean :
    rm -f $(OBJS) $(TARGET)
```

## 2. Linux File System

## 2. Linux File System

---

### **2.1 Linux File System**

2.2 Standard I/O Library

2.3 System Call I/O

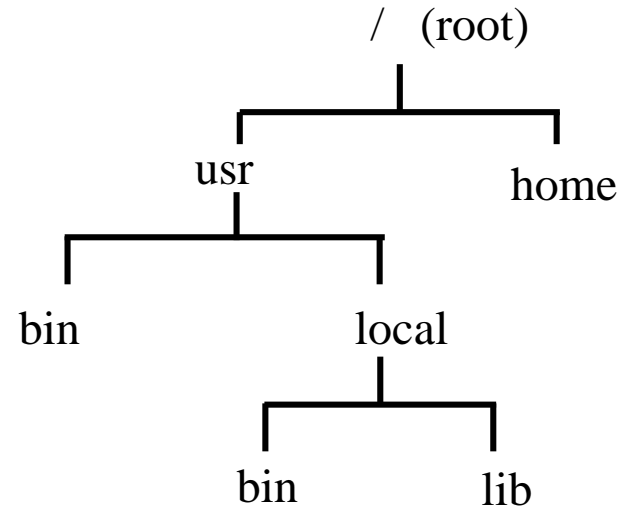
2.4 File Status & Directory

2.5 Hard Link vs Symbolic Link

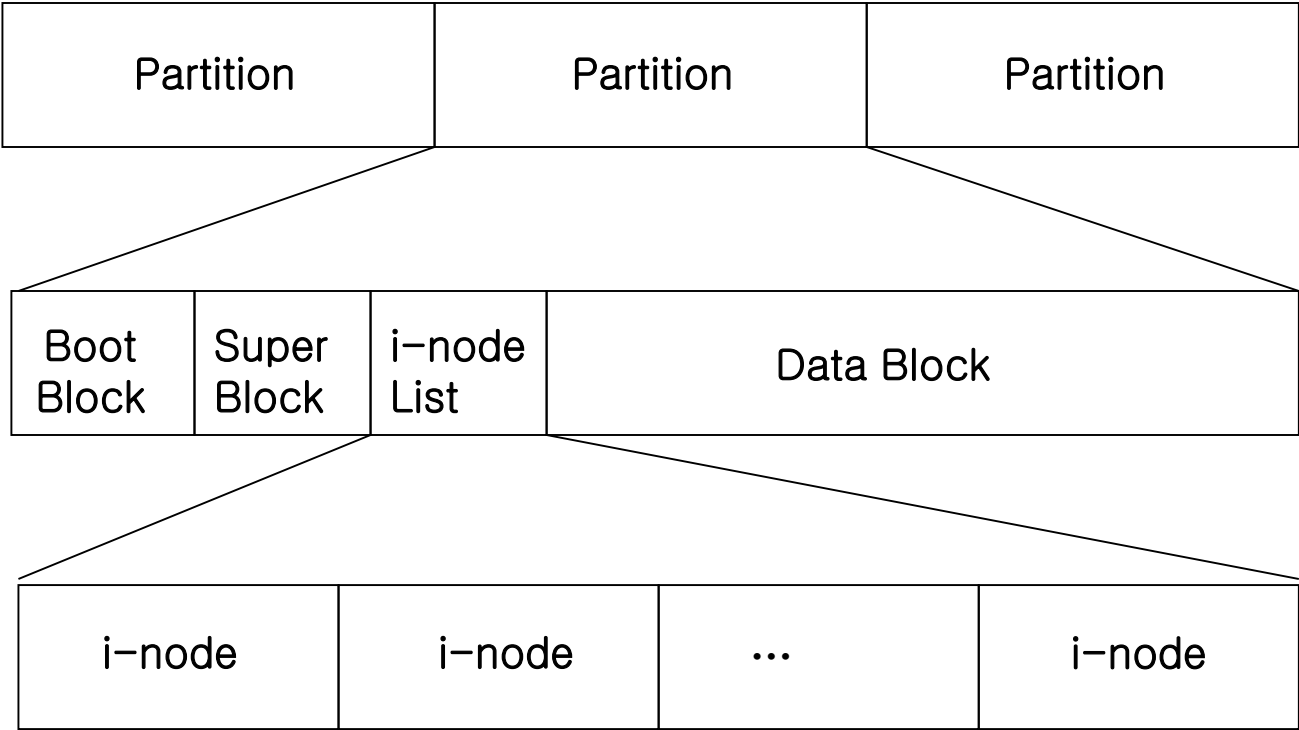
---

### ◆ File System 특징

- 트리 구조
- 터미널, 프린터, 디스크 등 모든 주변 장치들도 하나의 파일로 취급한다.
- 아스키 파일과 이진 파일을 동등하게 취급한다.
- 모든 파일은 허가모드를 갖는다.



◆ File System 구조



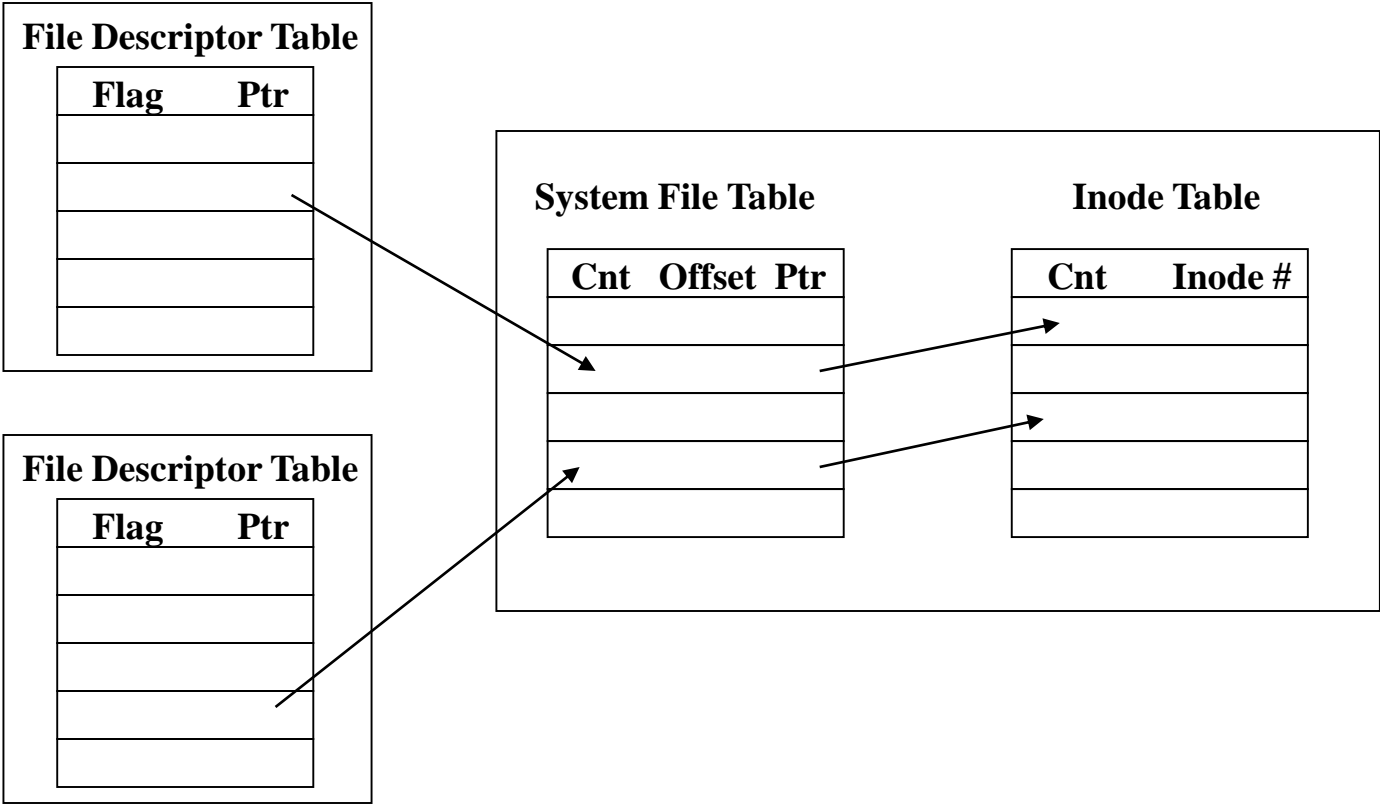
### ◆ Super Block

- 파일 시스템 크기
- 파일 시스템내의 자유 블록의 수
- 파일 시스템내에서 사용 가능한 자유 블록의 리스트
- i-node 리스트의 크기
- 파일 시스템내의 사용 가능한 i-node의 수
- 파일 시스템내의 사용 가능한 i-node의 리스트

### ◆ i-node List

- 각 항목은 하나의 파일과 대응
- 부팅시 추가 정보가 포함되어 메모리에 복사
- 각 i-node가 갖는 정보
  - 소유자 ID, 파일 유형, 파일 접근허가, 파일 접근시간
  - 링크수 . 파일 데이터의 주소 . 파일 크기
- 추가 정보
  - 참조계수, i-node번호, 파일 시스템 장치번호 등

◆ File Descriptor Table, File Table, Inode Table





## 2. Linux File System

---

2.1 Linux File System

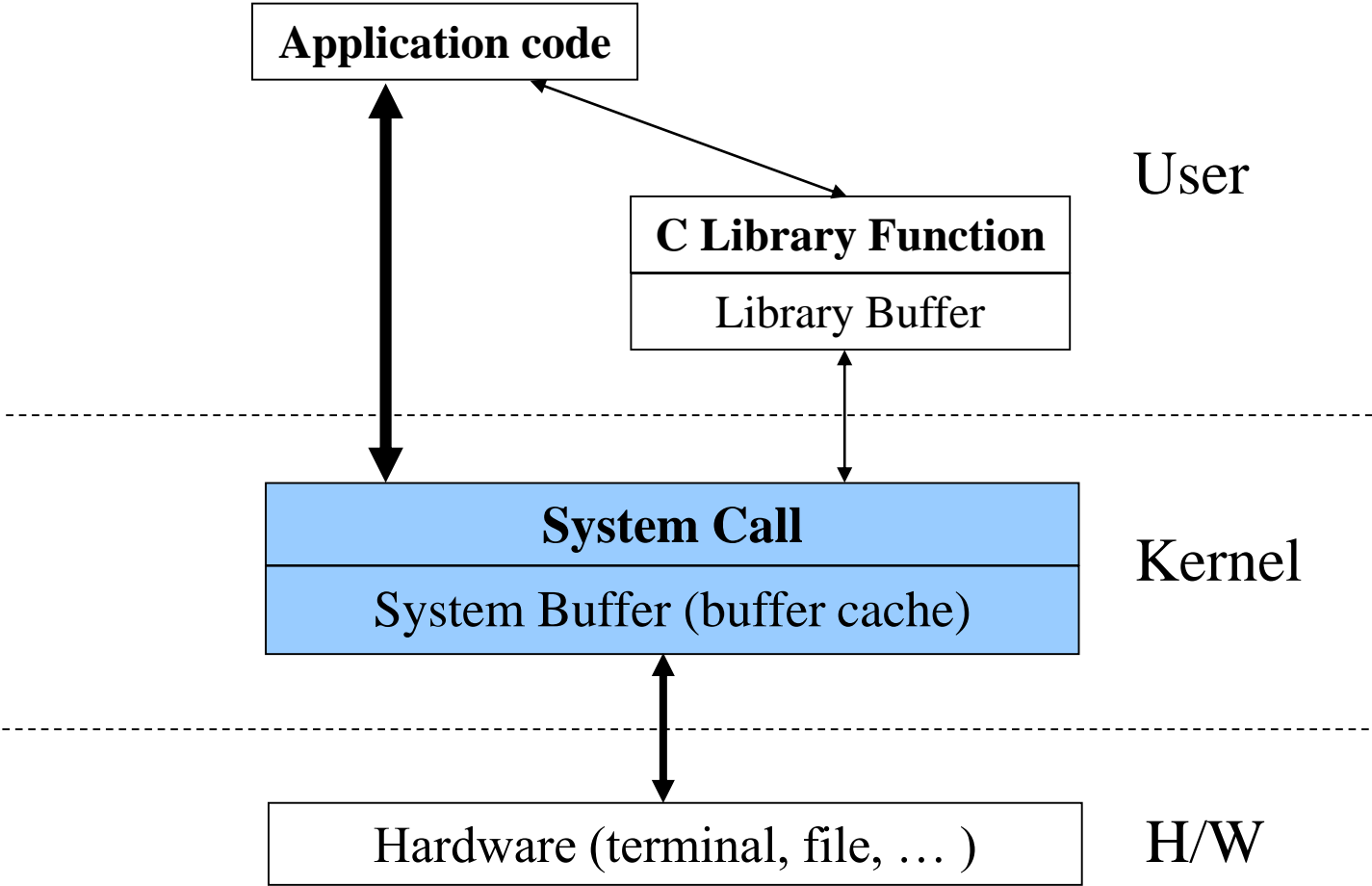
**2.2 Standard I/O Library**

2.3 System Call I/O

2.4 File Status & Directory

2.5 Hard Link vs Symbolic Link

---



file\_1.c

```
#include <stdio.h>

int main()
{
    FILE *fp;
    int ch;
    fp = fopen("file.c", "r");
    ch = fgetc( fp );
    fputc( ch, stdout );
    fclose(fp);
    return 0;
}
```

설명 :

**fopen** : 파일을 read 모드로 연다.

**fgetc** : 파일에서 1byte를 읽는다.

**fputc** : 1byte를 표준출력으로 보낸다.

**fclose** : 파일을 닫음으로 자원을 해지한다.

동작

file.c 파일을 읽기 모드로 열어서 한 바이트를 ch로 읽은 다음 화면으로 출력한다.  
연 파일을 닫는다.

## file\_2.c

```
#include <stdio.h>

int main()
{
    FILE *fp;
    int ch;
    int i;
    fp = fopen("file.c", "r");

    while((ch = fgetc( fp )) != EOF )
        fputc( ch, stdout );

    fclose(fp);
    return 0;
}
```

설명 :

EOF ( End Of File ) : 파일의 끝

#define EOF (-1)

표준 파일 입출력에서 파일의 끝은 EOF(-1)로 처리 됨으로 파일의 한바이트씩 읽어서 화면에 출력 하는데 EOF를 만나면 탈출하는 코드이다.

필수 ASCII 코드

'a' => 97

대문자 변환

'A' => 68

'a' - 32

'0' => 48

소문자 변환

'\n' => 10

'A' + 32

'\r' => 13

' ' => 32

숫자 변환

'\t' => 9

'7' - '0'

'\0' => 0

## file\_3.c

```
#include <stdio.h>
// ./a.out filename
int main( int argc, char **argv )
{
    FILE *fp;
    char ch;
    int count=0;
    fp = fopen( argv[1] , "r");

    while( (ch = fgetc( fp )) != EOF )
        count++;

    printf("count=%d\n", count );

    fclose(fp);
    return 0;
}
```

설명 :

ch를 int로 선언하는 이유는 무엇인가?

ch를 int로 선언하면 text파일이나 binary 파일 모두 파일의 사이즈가 정확히 출력된다.

하지만 ch를 char로 선언하면 컴파일 에러는 없지만 binary 파일의 사이즈가 정확히 계산되지 못하고 원래 크기보다 더 작게 출력된다.

## EOF 에 대한 고찰

EOF ( End Of File ) : 파일의 끝

EOF는 파일의 끝에 있는 구분자가 아니라  
file I/O 함수의 리턴값이다.

파일의 끝은 구분자가 따로 없다. 파일의 정보 구조체에 이미 파일의 크기가  
기록되어 있으므로 시스템은 파일을 끝까지 읽었는 지를 구분 할 수 있다.

EOF 는 4byte 전체가 1로 채워진 int 타입의 -1이다.

fgetc의 리턴값을 int로 받아야 된다.

이유 : 파일의 중간의 ff 패턴은 -1로 해석되지 않기 위해..

file\_4.c

```
int main()
{
    char buff[100];
    int ret;

    while( ret = fread( buff, 1, sizeof buff , stdin) )
        fwrite( buff, 1, ret , stdout );

    return 0;
}
```

설명 :

fread와 fwrite는 바이트 단위의 입력에 사용되는 함수 이다.

표준 입출력에서 stdin은 키보드 입력을 뜻한다.

표준 입출력에서 stdout은 화면 출력을 뜻한다.( line buffer 사용 )

표준 입출력에서 stderr은 화면 출력을 뜻한다.( non buffer )

```
#include <stdio.h>

int main()
{
    printf("hello");
    sleep(3);

    return 0;
}
```

```
#include <stdio.h>

int main()
{
    fprintf(stderr, "hello");
    sleep(3);

    return 0;
}
```

설명 :

printf의 출력에서 '\n'를 안쓰면 바로 출력되지않고  
3초뒤 프로세스가 종료할 때 출력된다. 이는 printf가 버퍼에만 출력하고  
'\n'가 출력 시점이기 때문이다 ( line buffered 방식 )

fprintf는 '\n'가 없어도 바로 출력 된다. stderr는 표준에러 스트림으로  
library버퍼를 사용하지 않음으로 바로 출력 된다.



### file\_5.c

```
#include <stdio.h>

int main()
{
    char buff[100];
    int ret;

    FILE *fp;
    fp = fopen("file.c", "r");
    ret = fread( buff, 1, 2, fp);
    fwrite( buff, 1, ret , stdout );
    getchar();

    ret = fread( buff, 1, 2, fp);
    fwrite( buff, 1, ret , stdout );
    fclose(fp);
    return 0;
}
```

설명 :

fopen으로 파일을 연후 2byte를 읽고 다시 2byte를 읽으로 파일이 연속에서 읽힌다.

당연해 보이지만 사용자는 파일의 현재 위치를 관리하지 않는다.

이는 시스템이 파일의 현재 읽는 위치를 관리 하고 있다는 말이 된다.

### file\_6.c

```
#include <stdio.h>

int main()
{
    FILE *fp;
    int ch;
    int i;
    fp = fopen("file.c", "r");

    while( (ch = fgetc( fp )) != EOF )
    {
        usleep(100000);
        fputc( ch, stdout );
        fflush(stdout);
    }

    fclose(fp);
    return 0;
}
```

설명 :

파일을 읽을때 1byte 단위로 읽으면 읽는 속도가 느리다.

이를 테스트 하기 위해 usleep을 사용하여 천천히 읽도록 했다.

또한 stdout은 '\n'가 있어야 출력 되므로 바로 바로 출력 하기 위해 fflush를 사용하여 1byte 단위로 출력 하도록 했다.

실행해 보면 화면 출력이 느리게 진행 됨을 알수 있다.

## file\_7.c

```
#include <stdio.h>

int main()
{
    FILE *fp;
    char buff[100];
    fp = fopen("file.c", "r");

    while( fgets(buff, sizeof buff, fp) )
    {
        usleep(100000);
        fputs( buff, stdout );
        fflush(stdout);
    }

    fclose(fp);
    return 0;
}
```

설명 :

fgets는 파일을 line단위로 읽어 주는 함수이고 fputs는 line 단위로 출력 해주는 함수 이다.

fgets는 파일의 끝은 만나면 0을 리턴한다.

똑같은 delay를 주었을 경우 1byte의 입/출력 보다.

1line의 평균 글자수가 20글자 라면 20배 빠른 프로그램이 된다.

## file\_8.c

```
#include <stdio.h>

int main()
{
    FILE *fp;
    char buff[100];
    int ret;
    fp = fopen("file.c", "r");

    while(ret=fread(buff,1,sizeof buff, fp ))
    {
        usleep(100000);
        fwrite( buff, 1, ret, stdout );
        fflush(stdout);
    }

    fclose(fp);
    return 0;
}
```

설명 :

fgets는 line 단위로 읽기  
때문에 buff의 크기가 100byte  
여도 한 라인의 글자수가 20  
글자라면 20 byte만 읽는다.

fread는 fully buffere를  
사용하므로 buffer 가 꽉 찰때  
까지 읽는다.

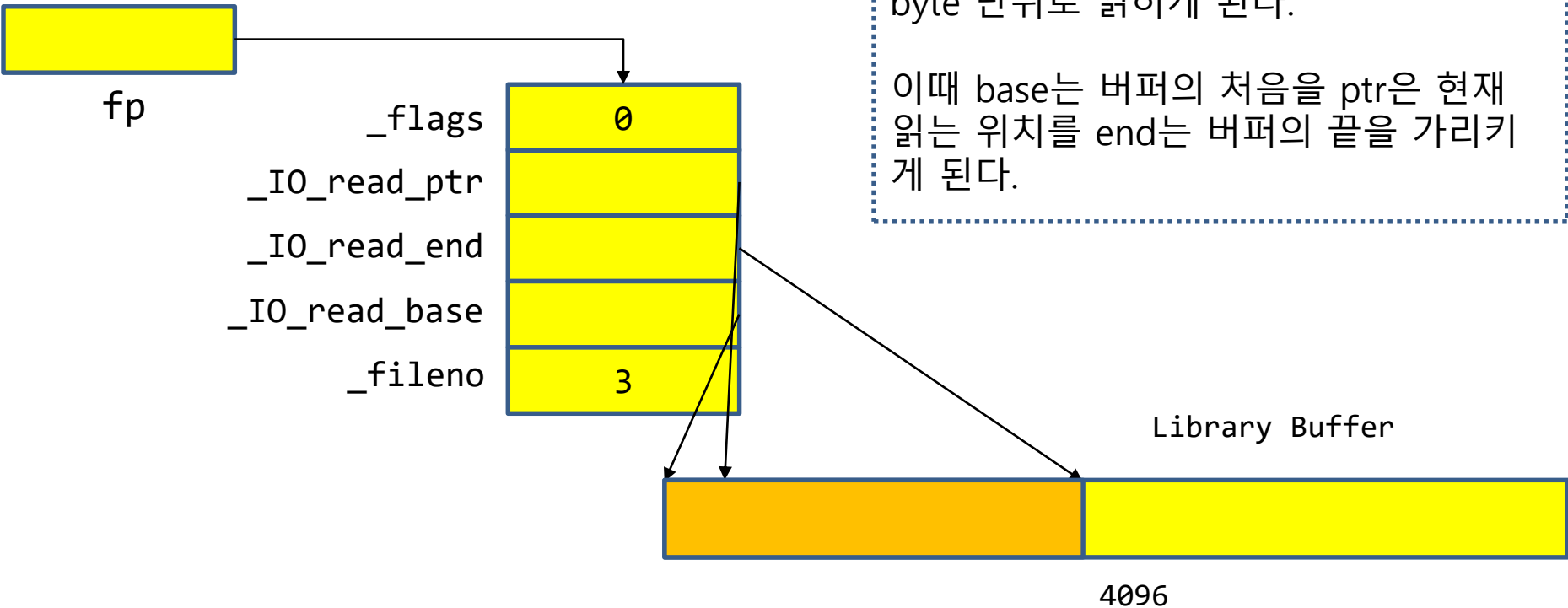
이는 1byte씩 입/출력 하는  
프로그램보다 100배 빠르다.

```
typedef struct _IO_FILE FILE;  
FILE *fp = fopen("file.c", "r");  
ch = fgetc( fp );
```

fopen 시에 FILE 구조체가 생성되며 구조체는 초기화된다.

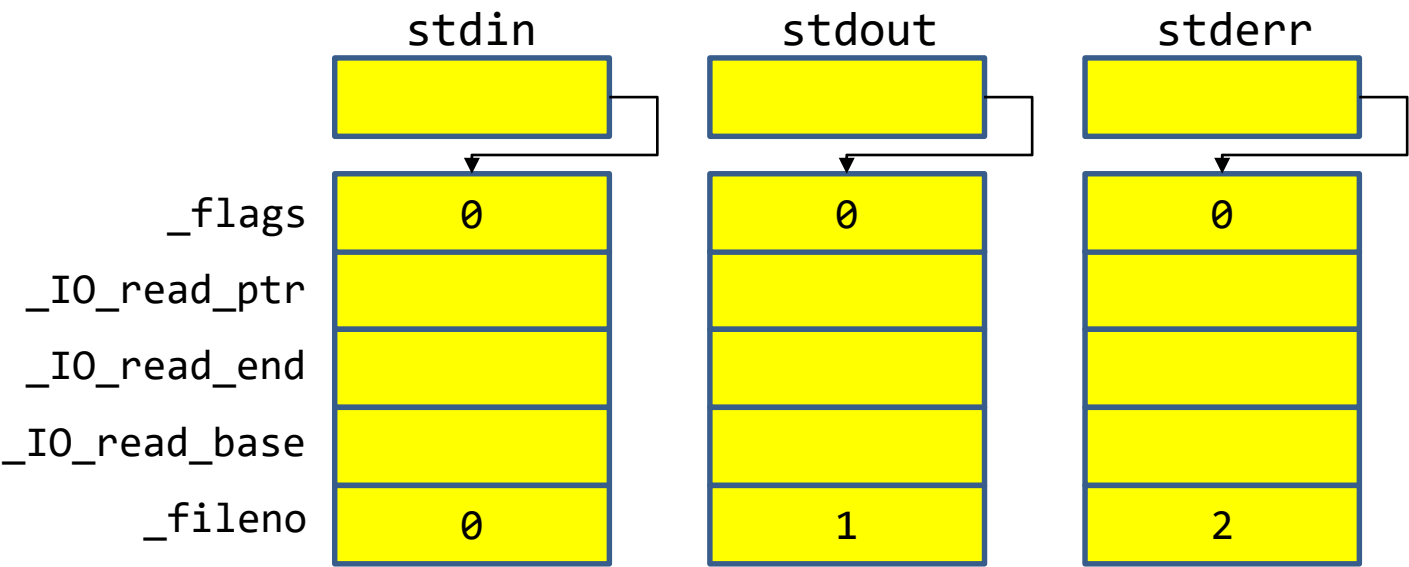
최초의 fgetc를 호출하게 되면 library 버퍼가 생성되며 파일의 데이터가 4096 byte 단위로 읽히게 된다.

이때 base는 버퍼의 처음을 ptr은 현재 읽는 위치를 end는 버퍼의 끝을 가리키게 된다.



```
scanf("%d", &data);
printf("hello\n");
fprintf(stderr, "err\n");
```

stdin : 파일의 입력을 담당하는 표준 스트림 포인터  
stdout : 파일의 출력을 담당하는 표준 스트림 포인터  
stderr : 파일의 error 출력을 담당하는 표준 스트림 포인터



a.txt

```
CameraService Mutex  
Condition PlaybackThread  
AudioFlinger      StubHandler LooperCallback  
aps_ops  
interruptible_sleep_on  
SurfaceFlinger    MPEG4Extractor  
setDataSource  
Create
```

```
# wc a.txt
```

```
8 13 170 a.txt
```

wc : 파일의 라인수, 단어수, 문자수를 세는 유틸리티이다.  
이를 구현 해 보세요..

## wc.c

```
#include <stdio.h>

// wc filename
int main( int argc, char **argv )
{
    FILE *fp;
    int ch;
    int flag=0;
    int line=0, char_cnt=0, word_cnt=0;
    int width;
    fp = fopen(argv[1], "r");

    while( (ch = fgetc( fp )) != EOF )
    {
        char_cnt++;
        if( ch == '\n' )    // 라인수는 '\n'의 개수이다.
            ++line;

        if( ch!='\n' && ch!=' ' && ch!='\t' ) // 공백 문자가 아니라면
        {
```



wc.c

```
        if( flag == 0 ) // 단어수는 단어의 첫 글자에서만 계수
        {
            word_cnt++;
            flag = 1;
        }
    }
    else
    {
        flag = 0;
    }
}
width=4;    // %*d, width는 가변적인 align을 변수를 사용하여 구현가능
printf("%*d %*d %*d %s\n",
        width, line, width, word_cnt, width, char_cnt, argv[1] );
fclose(fp);
return 0;
}
```

cp의 구현 : 파일을 복사하는 유틸리티를 구현해 보자.  
파일 복사는 원본파일에서 읽어서 목적 파일에 그대로 쓰는 것을 말한다.

```
# cp          aaa          bbb
    argv[0]    argv[1]      argv[2]
```

```
FILE *src, *dst;

src = fopen( argv[1], "r" ); // O_RDONLY
dst = fopen( argv[2], "w" );
      // O_WRONLY | O_TRUNC | O_CREAT, 0666

while( ret = fread( buff, 1, sizeof buff, src) )
    fwrite( buff, 1, ret, dst );

fclose(src);
fclose(dst);
```

merge 의 구현 : 파일을 병합하는 유틸리티를 구현해 보자.  
파일 병합은 복사의 확장형 으로 여러 원본파일에서 읽어서 목적  
파일에 읽은 순서대로 쓰는 것을 말한다.

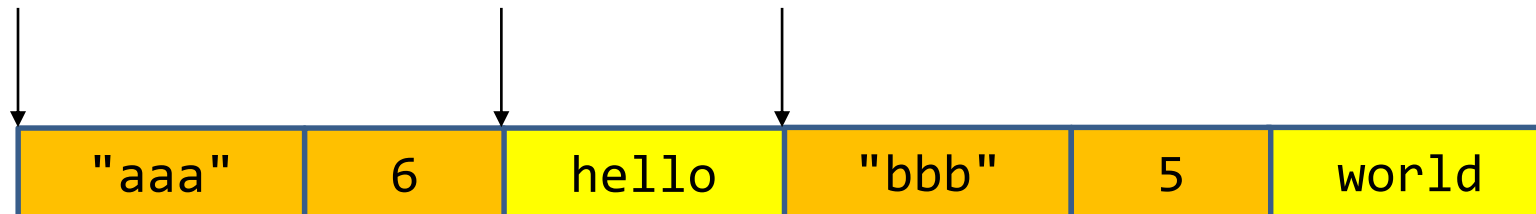
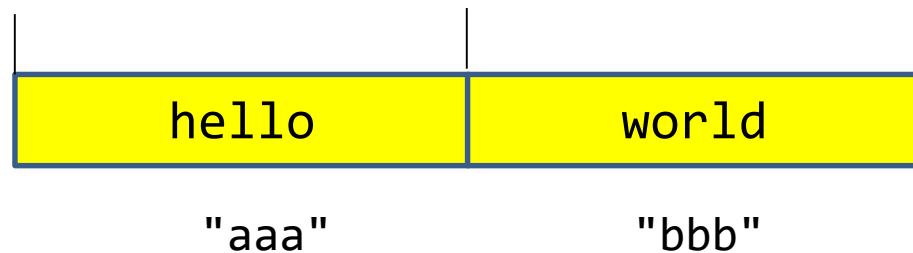
```
# merge      src1      src2      ...      target
      argv[0]  argv[1]  argv[2]          argv[argc-1]
```

```
FILE *src, *dst;

dst = fopen( argv[argc-1], "w" );

for( i=0; i<argc-2; i++)
{
    src = fopen( argv[i+1], "r" );
    while( ret = fread( buff, 1, sizeof buff, src) )
        fwrite( buff, 1, ret, dst );
    fclose(src);
}
fclose(dst);
```

merge 시의 고려 사항 : 파일을 병합 할때는 다시 파일을 분리 시킬 것을 고려해야 한다. 만일 파일의 데이터 만을 병합 하면 다시 파일을 분리 할 수 없다. 따라서 파일을 병합 할때는 필수 파일의 정보를 같이 목적 파일에 써야 한다.



```
typedef struct
{
    char fname[20];
    int  fsize;
} INFO;
```

## 파일의 크기를 계산하는 함수의 구현

```
int get_fsize( FILE *fp )
{
    int fsize, fpos;
    fpos = ftell( fp );           // 파일의 현재 위치를 백업
    fseek( fp, 0, SEEK_END );     // 파일의 offset을 파일 끝으로 옮김
    fsize = ftell( fp );          // 파일의 현재 위치를 fsize에 저장
    fseek( fp, fpos, SEEK_SET );  // 파일의 원래위치 복원
    return fsize;
}
```

### 설명 :

파일을 size를 계산 하기위해서는 세가지 방법이있다.

1. 파일을 read하고 read되는 바이트를 누적하여 계산하는 방식 ( read 이용 )
2. inode 속에있는 파일의 size를 뽑아내는 방식 ( stat 이용 )
3. fseek를 이용하여 파일의 offset을 파일 끝으로 옮기는 방식( fseek 이용 )

merge.c

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

typedef struct
{
    char fname[20];    // 파일의 이름 저장
    int  fsize;        // 파일의 크기 저장
} INFO;

int get_fsize( FILE *fp )
{
    int fsize, fpos;
    fpos = ftell( fp );
    fseek( fp, 0, SEEK_END );
    fsize = ftell( fp );
    fseek( fp, fpos, SEEK_SET );
    return fsize;
}
```

## merge.c

```
int main( int argc, char **argv ){
    INFO info;
    FILE *src, *dst;
    int ret,i;
    char buff[4096];

    dst = fopen( argv[argc-1], "w" );

    for( i=0; i<argc-2; i++ )    {
        src = fopen( argv[i+1], "r" );
        strcpy(info.fname, argv[i+1] ); // 파일의 이름 저장
        info.fsize = get_fsize( src );  // 파일의 크기 저장
        fwrite( &info, 1, sizeof info, dst ); // info 저장
        while( ret = fread( buff, 1, sizeof buff, src ) )
            fwrite( buff, 1, ret, dst );
        fclose(src);
    }
    fclose(dst);
}
```

## extract.c

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

typedef struct
{
    char fname[20];
    int  fsize;
} INFO;
// extract target
int main( int argc, char **argv )
{
    INFO info;
    FILE *src, *dst;
    int ret,i, len;
    char buff[4096];

    src = fopen( argv[1], "r" );
```



extract.c

```
while( ret = fread( &info, 1, sizeof info, src) )
{
    dst = fopen( info.fname, "w" );
    while( info.fsize > 0 ) // 파일의 크기가 소진 됐는가?
    {
        len = (sizeof buff < info.fsize )?
              sizeof buff:info.fsize;
        // 파일의 크기와 buffer의 크기중 작은것 선택
        ret = fread( buff, 1, len, src);
        fwrite( buff, 1, ret, dst );
        info.fsize -= ret; // 읽은 만큼 감소
    }
    fclose(dst);
}
fclose(src);
}
```

## 2. Linux File System

---

2.1 Linux File System

2.2 Standard I/O Library

**2.3 System Call I/O**

2.4 File Status & Directory

2.5 Hard Link vs Symbolic Link

---

## 표준 라이브러리

```
#include <stdio.h>
int main()
{
    FILE *fp;    // 파일 스트림 포인터 사용
    int ret;
    char buff[100];
    fp = fopen("a.c", "r");
    while( ret = fread( buff, 1, sizeof buff, fp ) )
        fwrite( buff, 1, ret , stdout );
    fclose(fp);
    return 0;
}
```

system programming

```
#include <fcntl.h>
int main()
{
    int fd; // 파일 디스크립터 사용
    int ret;
    char buff[100];
    fd = open("a.c", O_RDONLY);
    while( ret = read( fd, buff, sizeof buff ) )
        write( 1, buff, ret );
    close(fd);
    return 0;
}
```

### file\_read.c

```
#include <fcntl.h>
int main()
{
    int fd;
    int ret;
    char buff[100];
    fd = open("a.c", O_RDONLY);
    ret = read( fd, buff, 2 );
    write( 1, buff, ret );
    ret = read( fd, buff, 2 );
    write( 1, buff, ret );
    close(fd);
    return 0;
}
```

### file\_open.c

```
#include <fcntl.h>
int main()
{
    int fd1, fd2;
    int ret;
    char buff[100];
    fd1 = open("a.c", O_RDONLY);
    fd2 = open("a.c", O_RDONLY);
    ret = read( fd1, buff, 2 );
    write( 1, buff, ret );
    ret = read( fd2, buff, 2 );
    write( 1, buff, ret );
    close(fd1);
    close(fd2);
    return 0;
}
```

dup\_1.c

```
#include <fcntl.h>
#include <stdio.h>
// ./a.out xxx
int main(int argc, char **argv)
{
    int fd;
    fd = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0666);

    close(1); // 표준 출력을 닫는다.
    dup(fd);   // 파일 디스크립터를 표준 출력에 복제 한다.

    printf("hello\n"); // 출력이 파일로 나간다.
    close(fd);

    return 0;
}
```

dup\_2.c

```
#include <fcntl.h>
#include <stdio.h>
// ./a.out dup_2.c
int main(int argc, char **argv){
    int fd
    int ret;
    char buff[100];

    fd = open(argv[1], O_RDONLY);
    close(0);
    dup(fd);

    ret = read(0, buff, sizeof buff );
    write(1, buff, ret );

    close(fd);
    close(0);

    return 0;
}
```



cat.c

```
#include <fcntl.h>
#include <stdio.h>
// ./a.out dup.c
int main(int argc, char **argv)
{
    int fd
    int ret;
    char buff[100];
    if( argc == 2 )
    {
        fd = open(argv[1], O_RDONLY);
        close(0);
        dup(fd);
    }

    while( ret = read(0, buff, sizeof buff ) )
        write(1, buff, ret );
}
```

cat.c

```
        if( argc == 2 )
        {
                close(fd);
                close(0);
        }

        return 0;
}
```

cat 의 구현 설명 :

cat은 인자없이 실행 하면 키보드 입력을 받아서 화면에 출력 하고  
파일을 인자로 전달하면 파일을 읽어서 화면에 출력 한다.

이는 키보드 입력을 받아서 화면에 출력하는 구현을 바꾸지 않고  
표준 입력 리다이렉션을 이용하여 구현한다.

## 2. Linux File System

---

2.1 Linux File System

2.2 Standard I/O Library

2.3 System Call I/O

**2.4 File Status & Directory**

2.5 Hard Link vs Symbolic Link

---

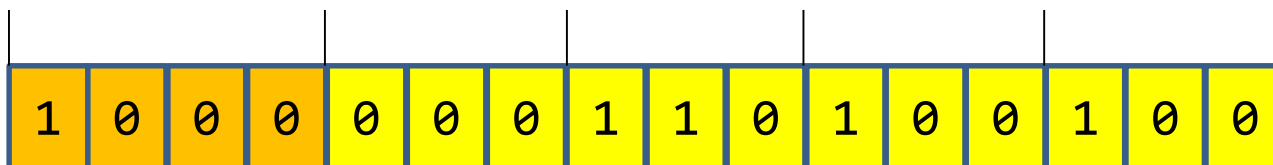
파일의 속성 추출 : stat(filename, &buf);

stat.c

```
#include <sys/stat.h>
int main(int argc, char *argv[]){
    int    i;
    struct stat    buf;
    char    *ptr;
    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        stat(argv[i], &buf);
        if      (S_ISREG(buf.st_mode))    ptr = "regular";
        else if (S_ISDIR(buf.st_mode))    ptr = "directory";
        else if (S_ISCHR(buf.st_mode))    ptr = "character special";
        else if (S_ISBLK(buf.st_mode))    ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))    ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))    ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))    ptr = "socket";
        else                                ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
}
```

st\_mode은 16bit 자료구조 이며 파일의 종류 및 퍼미션이 기록 되어 있다. 이때 상위 4bit가 파일의 종류를 의미한다.

```
buf.st_mode == 81a4
```



-

```
#define S_IFMT 0170000  
           00 1111 0000 0000 0000  
#define S_IFREG 0100000  
           00 1000 0000 0000 0000  
#define S_ISREG(m) (((m) & S_IFMT) == S_IFREG)
```

permission.c

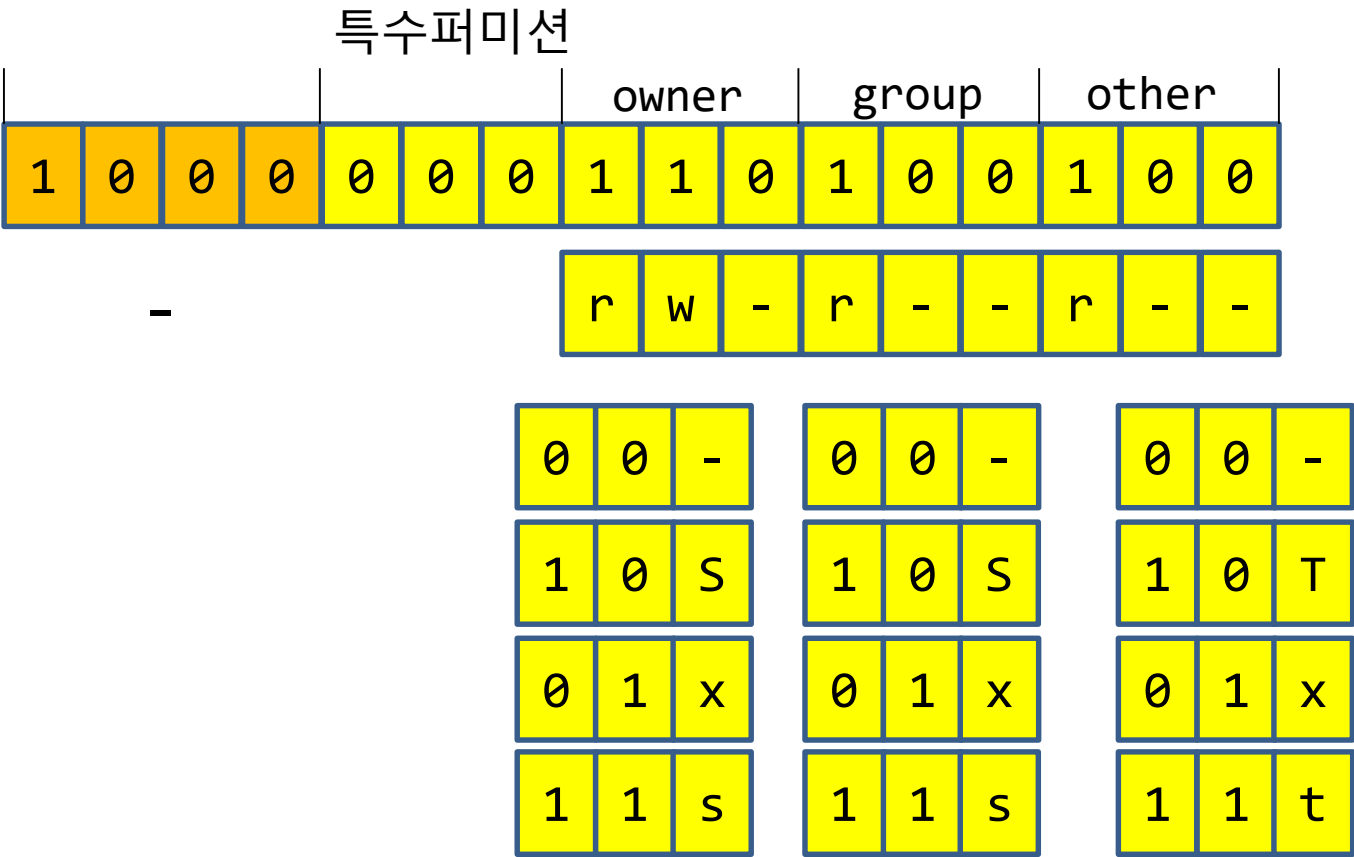
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    struct stat buf;
    char perm[11] = "-----";
    char rwx[] = "rwx";
    int i;
    stat( argv[1], &buf );

    for( i=0; i<9; i++ )
    {
        if( (buf.st_mode >> (8-i)) & 0x1 )
            perm[1+i] = rwx[i%3];
    }
    printf("%s\n", perm );
    return 0;
}
```

특수 퍼미션도  
출력 되도록 프로그램을  
수정해 보세요.

퍼미션 추출 : st\_mode의 하위 12bit는 퍼미션이다.



## 연결 계수 추출 시 자릿수 설정

```
drwxr-xr-x    2 root root  4096 Jul 19 23:47 libx32
drwx-----    2 root root 16384 Jul 19 21:20 lost+found
drwxr-xr-x    4 root root  4096 Jul 19 22:04 media
drwxr-xr-x    2 root root  4096 Apr 19  2013 mnt
drwxr-xr-x    3 root root  4096 Jul 19 21:32 opt
dr-xr-xr-x 136 root root      0 Nov 20 02:41 proc
drwx-----  48 root root  4096 Nov 21 18:44 root
drwxr-xr-x   23 root root   760 Nov 21 05:00 run
drwxr-xr-x    2 root root 12288 Jul 19 21:32 sbin
```

// 연결계수 출력시는 자릿수가 가장 큰쪽으로 정렬 하여 출력 한다.  
// %\*d 문법은 자릿수를 알수 없을 때 런타임 시점에 width를 설정 가능함

예)

```
printf("%*d", width , buf.st_nlink );
```



파일의 UID는 숫자 이므로 passwd 파일을 참조하여 string으로 바꿔야 한다.  
/etc/passwd 파일 참조

```
#include <pwd.h>

struct passwd {
    char    *pw_name;          /* username */
    char    *pw_passwd;        /* user password */
    uid_t    pw_uid;           /* user ID */
    gid_t    pw_gid;           /* group ID */
    char    *pw_gecos;         /* user information */
    char    *pw_dir;           /* home directory */
    char    *pw_shell;         /* shell program */
};

pwd = getpwuid( buf.st_uid );
printf("%-*s ", max_uid,  pwd->pw_name );
```

파일의 GID도 숫자 이므로 group 파일을 참조하여 string으로 바꿔야 한다.  
/etc/group 파일 참조

```
#include <grp.h>

struct group {
    char    *gr_name;        /* group name */
    char    *gr_passwd;      /* group password */
    gid_t   gr_gid;          /* group ID */
    char    **gr_mem;        /* group members */
};

grp = getgrgid( buf.st_gid );
printf("%-*s ", max_gid, grp->gr_name );
```

```
# ls -l stat.c
-rw-r--r-- 1 root root 938 Nov 21 18:57 stat.c
```

```
# ls -l /dev/tty
crw-rw-rw- 1 root tty 5, 0 Nov 20 02:41 /dev/tty
```

epoch : 1970년 1월 1일 자정으로 부터 현재 까지 흘러온 초 단위의 시간

```
st_mtime => 1385089351
time_t
```

st\_mtime : 파일이 마지막으로 수정된 시간

```
#include <time.h>

struct tm {
    int tm_sec;           /* seconds */
    int tm_min;           /* minutes */
    int tm_hour;          /* hours */
    int tm_mday;          /* day of the month */
    int tm_mon;           /* month */
    int tm_year;          /* year */
    int tm_wday;          /* day of the week */
    int tm_yday;          /* day in the year */
    int tm_isdst;         /* daylight saving time */
};

tmp = localtime( &buf.st_mtime );
printf("%04d-%02d-%02d %02d:%02d ", tmp->tm_year+1900,
        tmp->tm_mon+1, tmp->tm_mday,
        tmp->tm_hour, tmp->tm_min );
```

stat.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>

// -rw-r--r-- 1 root root 149 Nov 21 05:23
int main(int argc, char **argv)
{
    struct stat buf;
    struct passwd *pwd;
    struct group *grp;
    char perm[11] = "-----";
    char rwx[] = "rwx";
    int i;
    stat( argv[1], &buf );

    //printf("%x\n", buf.st_mode );
```

stat.c

```
if( S_ISDIR( buf.st_mode ) )
    perm[0] = 'd';
if( S_ISCHR( buf.st_mode ) )
    perm[0] = 'c';
if( S_ISBLK( buf.st_mode ) )
    perm[0] = 'b';
if( S_ISSOCK( buf.st_mode ) )
    perm[0] = 's';
if( S_ISLNK( buf.st_mode ) )
    perm[0] = 'l';
if( S_ISFIFO( buf.st_mode ) )
    perm[0] = 'p';

for( i=0; i<9; i++ )
{
    if( (buf.st_mode >> (8-i)) & 0x1 )
        perm[1+i] = rwx[i%3];
}

printf("%s", perm );
```

stat.c

```
printf(" %*d", 3,  buf.st_nlink );

pwd = getpwuid(buf.st_uid);
printf(" %s",  pwd->pw_name );

grp = getgrgid(buf.st_gid);
printf(" %s",  grp->gr_name );

if( perm[0] == 'c' || perm[0] == 'b' )
{
    printf(" %d, %d", (buf.st_rdev>>8)&0xff ,
               buf.st_rdev&0xff);
}
else
{
    printf(" %d", buf.st_size );
}

printf(" %s", ctime(&buf.st_mtime));
```

stat.c

```
    if( perm[0] != 'l' )
        printf(" %s\n", argv[1]);
    else
    {
        int ret;
        char temp[100];
        ret = readlink( argv[1], temp, sizeof temp);
        temp[ret] = 0;
        printf(" %s -> %s\n", argv[1], temp);
    }

    return 0;
}
```



### ◆ Directory

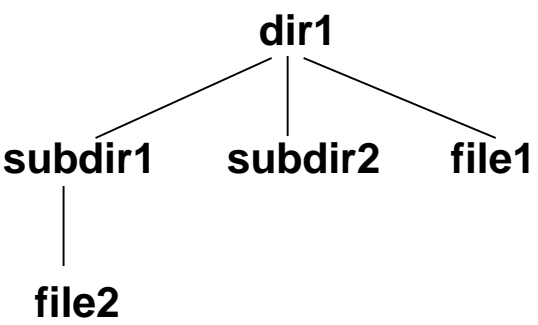
- 일반 파일 및 다른 디렉토리의 이름으로 구성

- dirent 구조체의 항목으로 구성되는 테이블 형식

```
struct dirent{
    ino_t    d_ino;           // l-node 번호
    char     d_name[NAME_MAX + 1]; // 파일 이름
}
```

- DIR 구조체
  - 개방된 디렉토리를 접근하는데 필요한 구조체
  - 표준 라이브러리 I/O에서는 FILE 구조체와 같은 역할

◆ Directory



Directory dir1	
222	.\0
122	..\0
333	subdir1\0
245	subdir2\0
432	file1\0

Directory subdir1	
333	.\0
222	..\0
433	file2\0

Directory subdir2	
245	.\0
222	..\0

- . (현재 디렉토리), .. (부모 디렉토리)

opendir() : 디렉토리 파일 스트림을 개방한다.

readdir() : dirent 구조체를 읽는다.

closedir() : 디렉토리 파일 스트림을 닫는다.

dir\_1.c

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

int main()
{
    DIR *dp;
    struct dirent *p;
    dp = opendir(".");
    while( p = readdir( dp ))
        printf("%s\n", p->d_name );
    closedir(dp);
    return 0;
}
```

dir\_2.c

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

void my_ls( char *dname )
{
    DIR *dp;
    struct dirent *p;
    chdir(dname);          // 해당 디렉토리로 이동
    dp = opendir(".");     // 현재 디렉토리 open
    while( p = readdir( dp ) )
        printf("%s\n", p->d_name );
    closedir(dp);
    chdir("..");           // 부모 디렉토리로 이동
}

int main()
{
    my_ls(".");           // 디렉토리의 순회 구현을 모듈화 한다.
    return 0;
}
```

dir\_3.c

```
void my_ls( char *dname ) {
    DIR *dp;
    struct dirent *p;
    struct stat buf;
    chdir(dname);
    dp = opendir(".");
    while( p = readdir( dp ) ) {
        lstat(p->d_name, &buf);
        printf("%s\n", p->d_name );

        if( S_ISDIR(buf.st_mode) )
        {
            if( strcmp( p->d_name, "." )
                && strcmp( p->d_name, ".." ) )
                my_ls(p->d_name);
            // 하위 디렉토리는 재귀함수로 순회한다.
        }
    }
    closedir(dp);
    chdir("..");
}
```

dir\_4.c : 화면 출력과 재귀호출을 분리 한다.

```
void my_ls( char *dname ){
    DIR *dp;
    struct dirent *p;
    struct stat buf;
    chdir(dname);
    dp = opendir(".");
    printf("%s : ", dname );
    while( p = readdir( dp ))
        printf("%s ", p->d_name );
    printf("\n");

    rewinddir(dp);
    while( p = readdir( dp )) {
        lstat(p->d_name, &buf);
        if( S_ISDIR(buf.st_mode) ) {
            if( strcmp( p->d_name, "." ) && strcmp( p->d_name, ".." ) )
                my_ls(p->d_name);
        }
    }
    closedir(dp);
    chdir("..");
}
```

## ◆ 옵션 처리

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

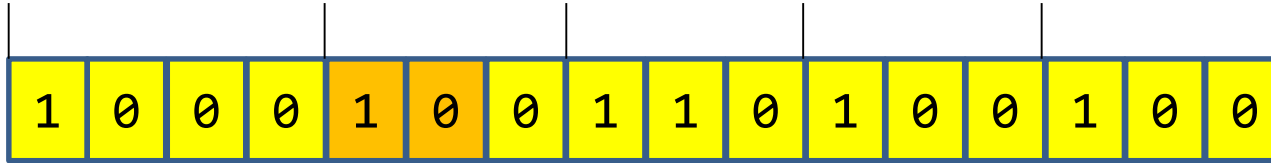
#define LIST    1
#define INODE   2
#define RECUR   4
#define ALL     8
int main( int argc, char **argv ){
    int ch;
    int flag=0;
    while( (ch = getopt(argc, argv, "liRa" ) ) != -1 )
    {
        switch(ch)
        {
            case 'l': flag |= LIST; break;
            case 'i': flag |= INODE; break;
            case 'R': flag |= RECUR; break;
            case 'a': flag |= ALL; break;
        }
    }
}
```

### ◆ 옵션 처리

```
    if( flag & LIST ) printf("l\n");  
    if( flag & INODE ) printf("i\n");  
    if( flag & RECUR ) printf("R\n");  
    if( flag & ALL ) printf("a\n");  
    return 0;  
}
```



## ◆ Set User ID 비트의 동작 방식



퍼미션의 일반적인 룰상 유저는 자신의 패스워드를 바꿀수 없다.  
하지만 Set User ID bit를 이용하면 가능하다.

```
linux123 , md5("$6", "linux123" );
```

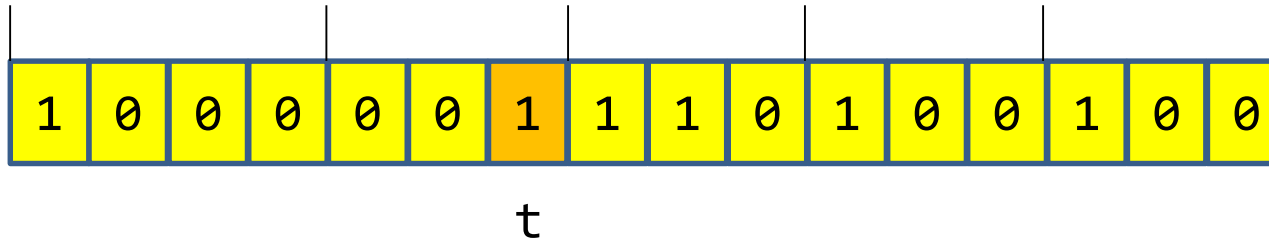
```
$6$UXqokE00$RfNOuaLC9Gohf.iuIm0XbYcq16XoviQwWpwKtx1krEgokwFvdRrGlJYRq...
```

**Real User ID => Effective User ID**

**S => Set User Id bit = 1**

**Owner User ID => Effective User ID**

### ◆ Sticky Bit



Sticky bit :

- 디렉토리에 해당 bit가 세팅 되어 있을때 소유자만이 파일을 지울 수 있다.
- stat 구조체의 st\_mode값과 S\_ISVTX 마스크를 AND 연산에 의해 속성을 알수 있다.

## 2. Linux File System

---

2.1 Linux File System

2.2 Standard I/O Library

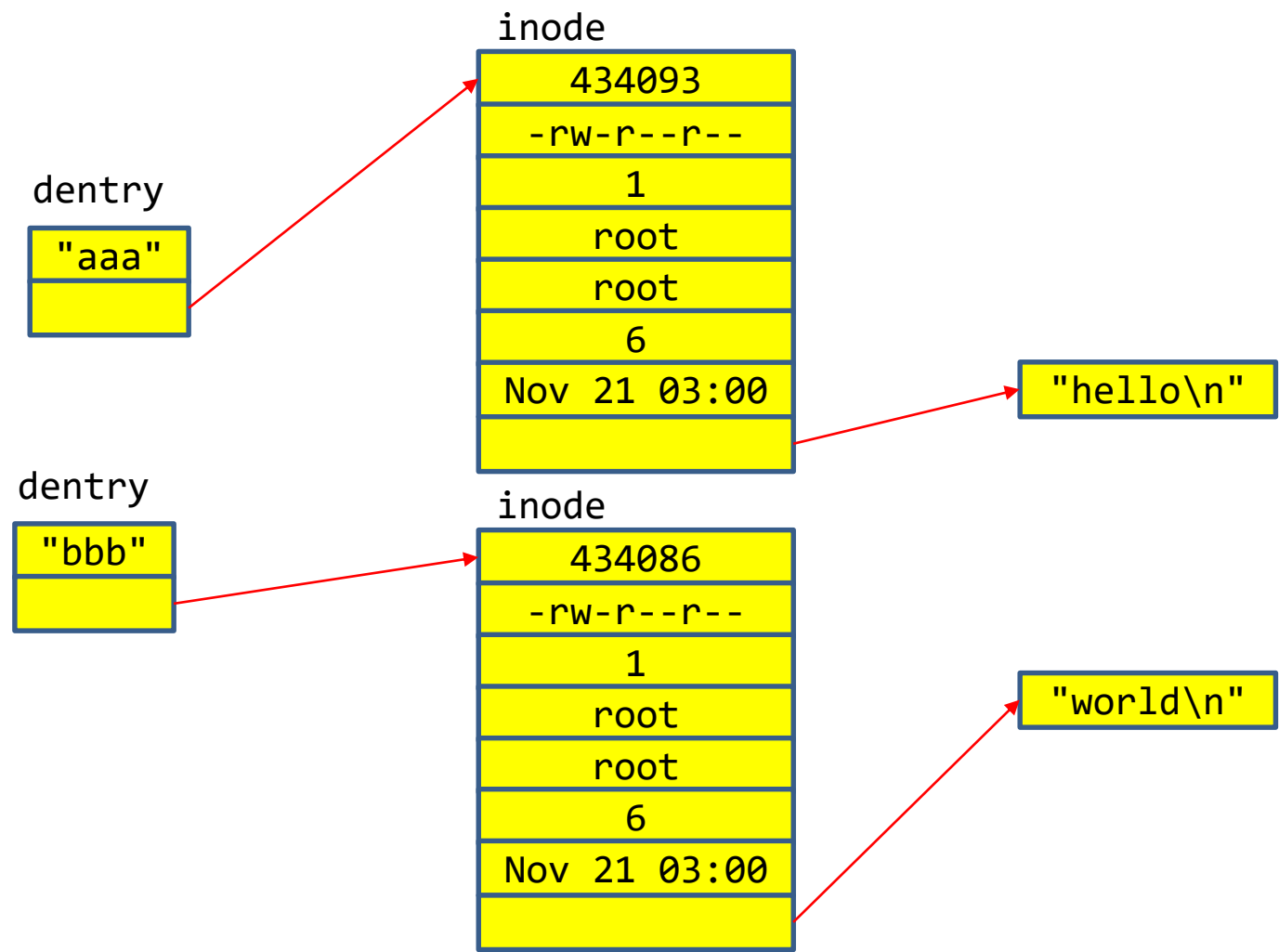
2.3 System Call I/O

2.4 File Status & Directory

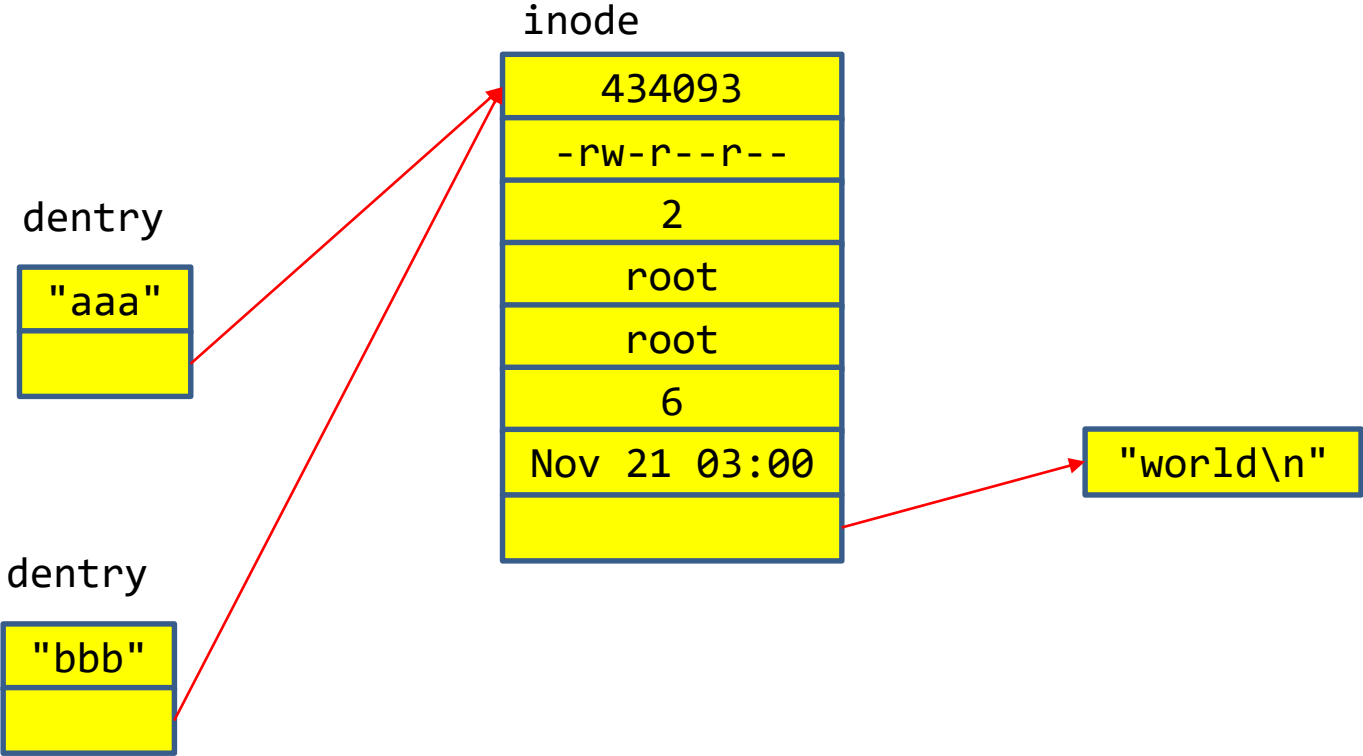
**2.5 Hard Link vs Symbolic Link**

---

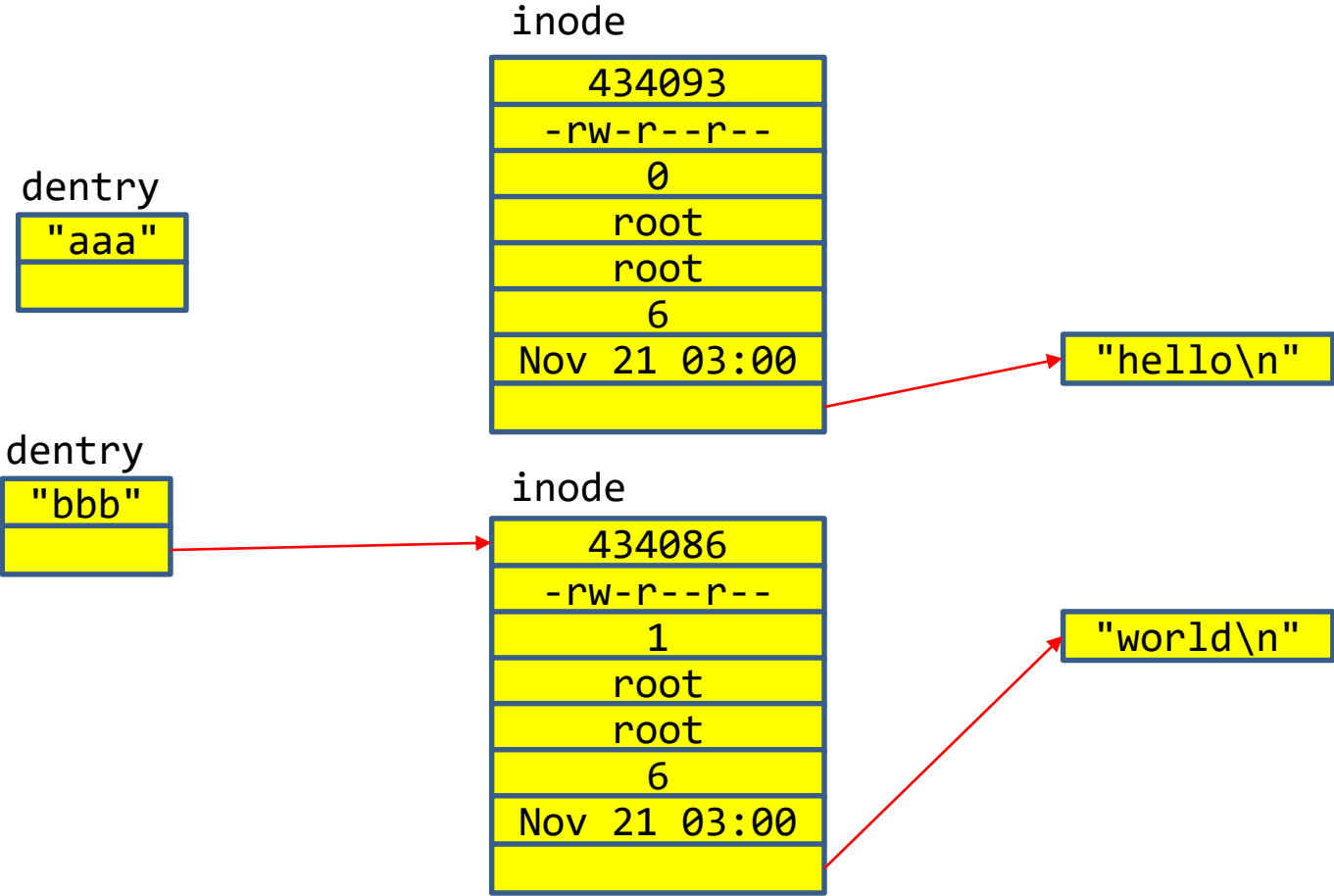
# cp aaa bbb // 파일을 복사 하면 data 및 inode까지 깊은 복사가 된다.



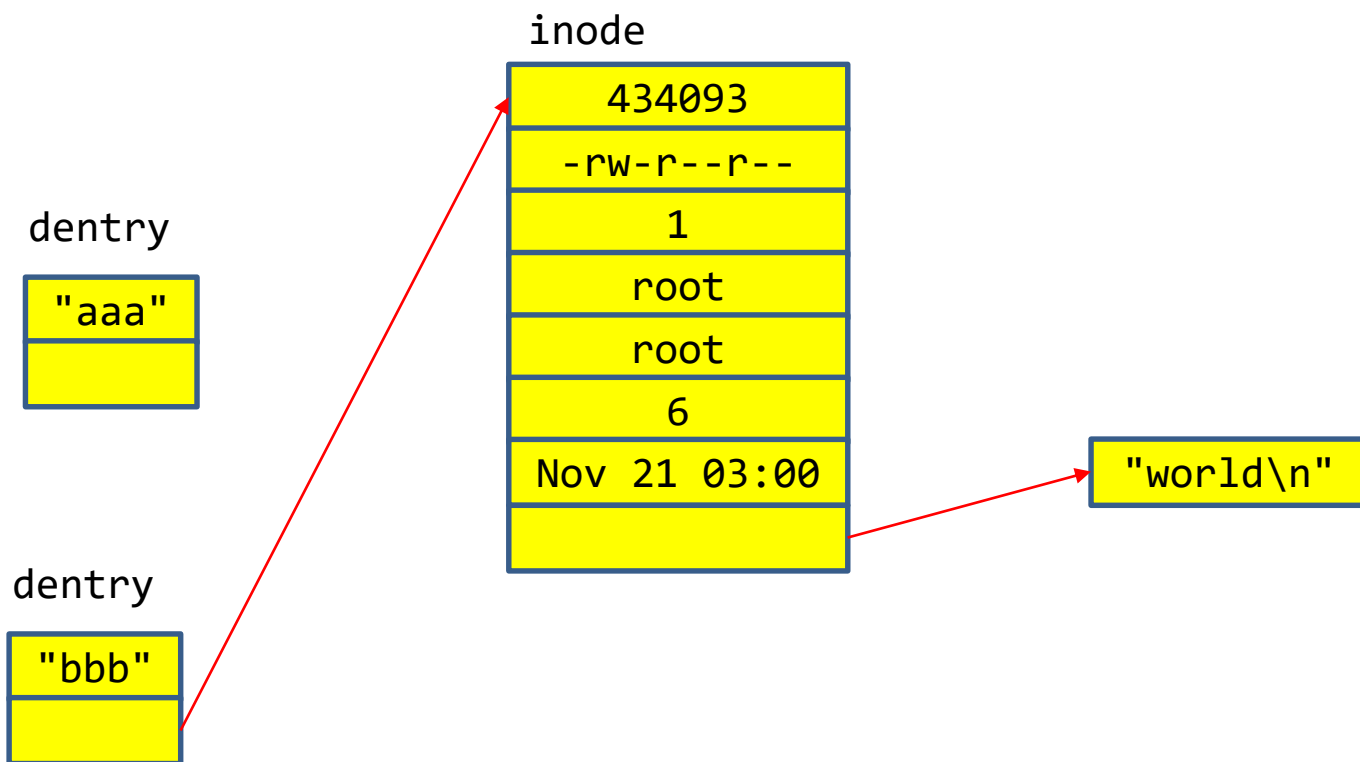
```
# ln aaa bbb // 파일을 링크 하면 같은 inode를 가리킨다.
```



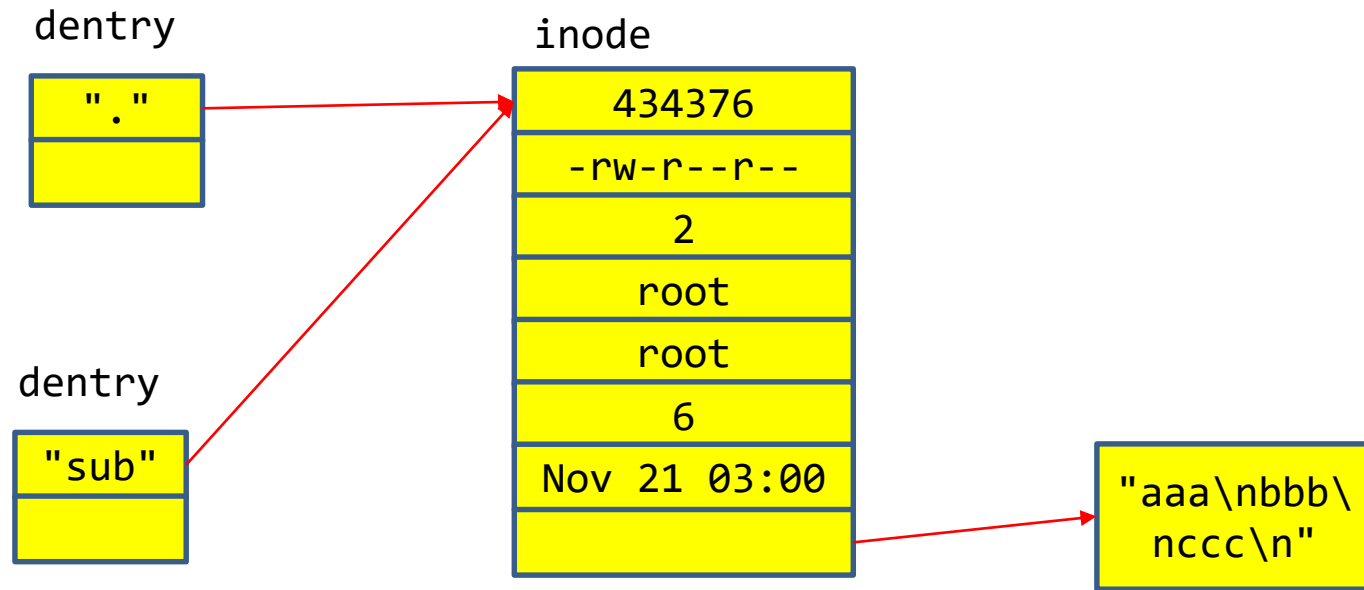
```
# cp aaa bbb
# rm aaa // unlink(); // 복제 및 삭제시 파일이 큰경우 오버헤드가 있다.
```



```
# ln aaa bbb
# rm aaa    // unlink(aaa)    // ln으로 링크후 원본을 unlink하면 실제 파일의 복사는
                                // 없어도 되므로 속도가 빠르다.
```



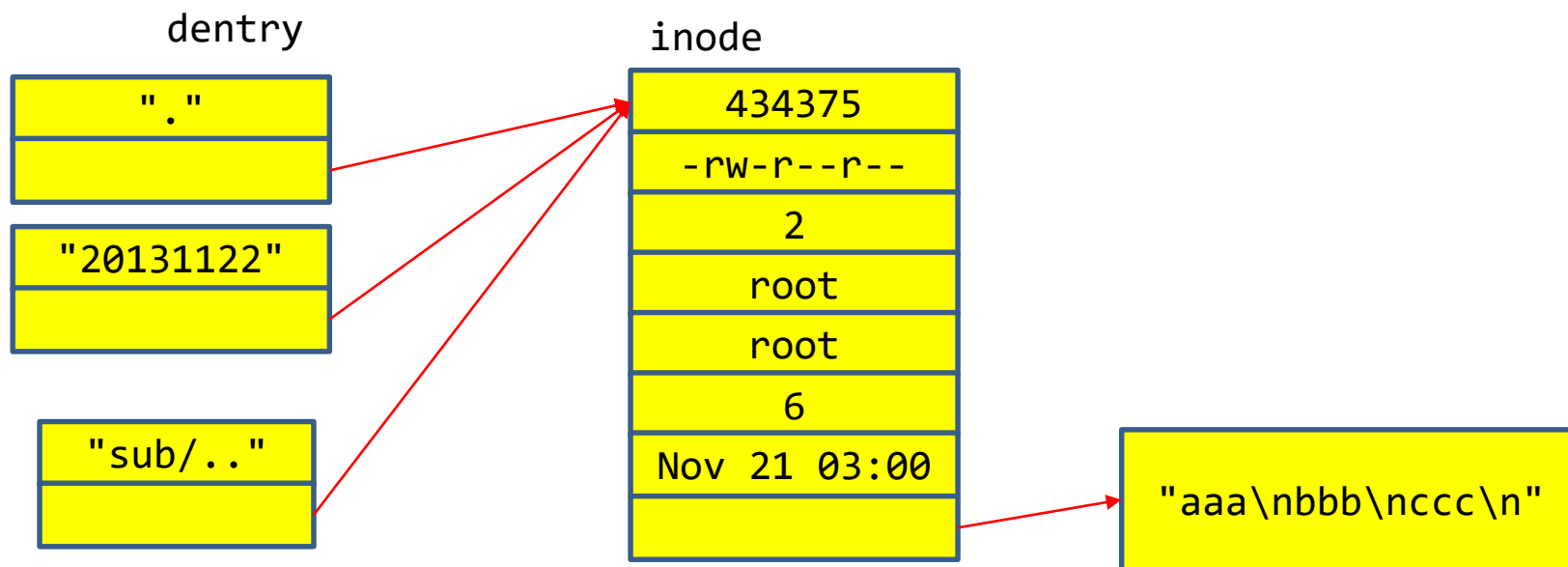
하드 링크의 필요 이유 : 상대 경로 때문





하드 링크의 필요 이유 : 상대 경로 때문

- . : 현재 디렉토리에 대한 하드링크
- .. : 부모 디렉토리에 대한 하드링크



### 하드링크의 제약 사항

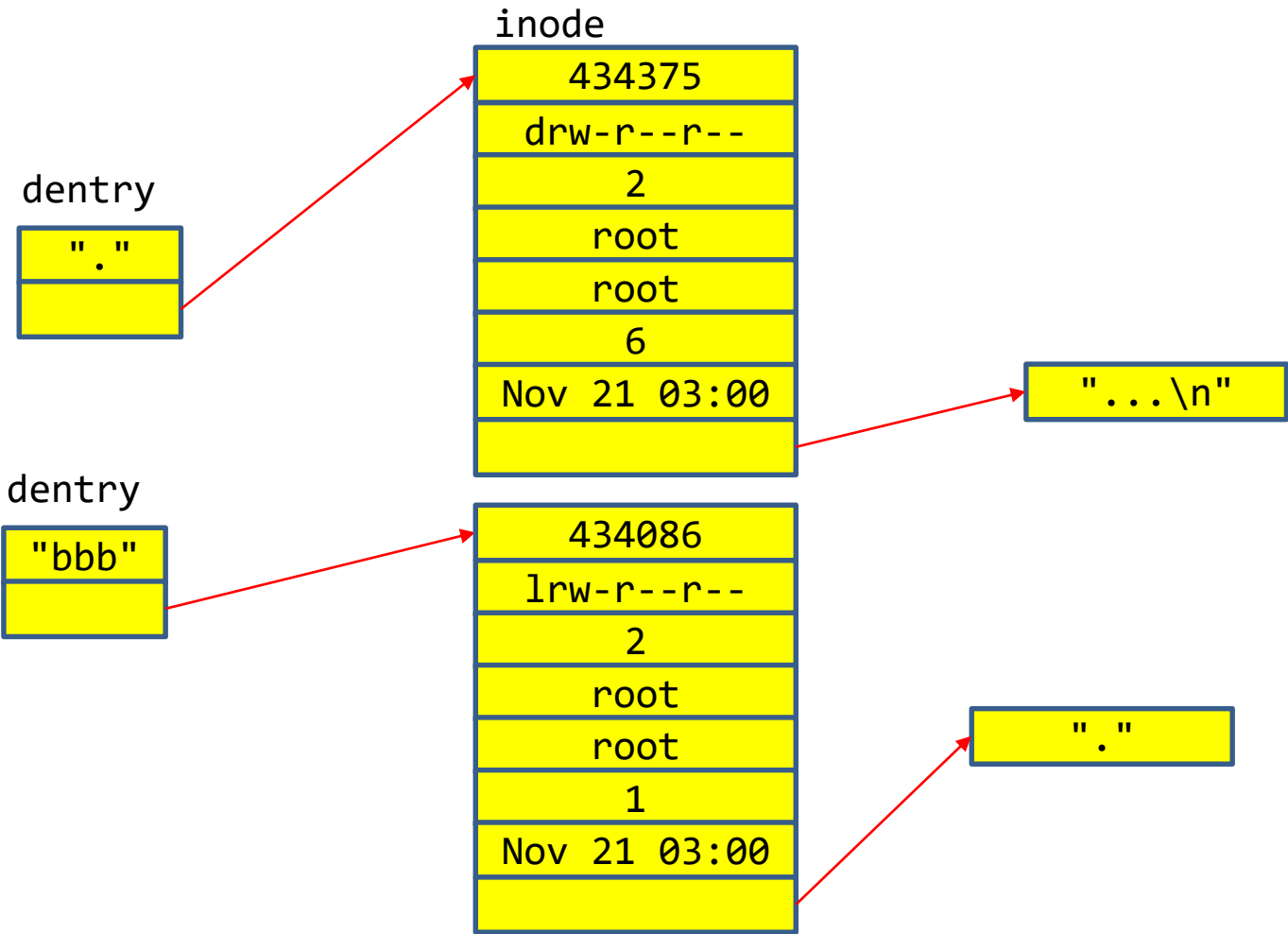
1. 디렉토리는 하드링크 할 수 없다. ( -R 옵션 때문 )
2. 파티션이 다르면 하드 링크 할 수 없다. ( inode가 파티션 마다 따로 저장 되기 때문)

바로가기 기능은 하드 링크 할 수 없다.

해결책

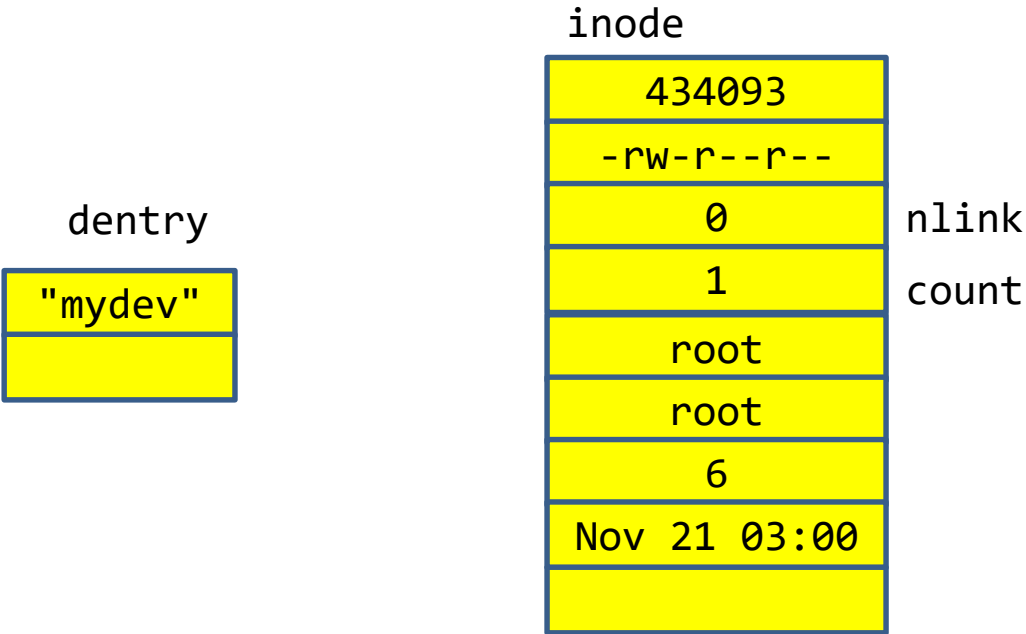
심볼릭 링크

```
# ln -s . bbb // symlink(); // 심볼릭 링크는 데이터에 경로를 포함하는
// 특수 파일이다.
```



```
mknod(name, S_IFCHR | 0600, (1 << 8) | 3); // # mknod mydev c 1 3
fd = open(name, O_RDWR);
unlink(name);                               // # rm mydev
```

파일이 지워지는 시점은 연결계수와 참조계수가 둘다 0일 때 이다.



### unlink.c

```
#include <fcntl.h>
int main()
{
    int fd, ret ;
    char buff[100];
    fd = open( "zzz", O_RDWR | O_CREAT | O_TRUNC, 0666);
    unlink("zzz"); // 이시점에서는 파일의 이름만 지워진다.

    write(fd, "hello\n", 6 ); // unlink 이후에도 파일에 쓸수 있다.
    lseek(fd, 0 , SEEK_SET );

    ret = read( fd, buff, sizeof buff );
    // unlink 이후에도 파일에서 읽을수 있다.
    write( 1, buff, ret );

    for(;;)
        ;
    return 0;
}
```

# 3. Process Programming

# 3. Process Programming

---

## **3.1 Process Structure**

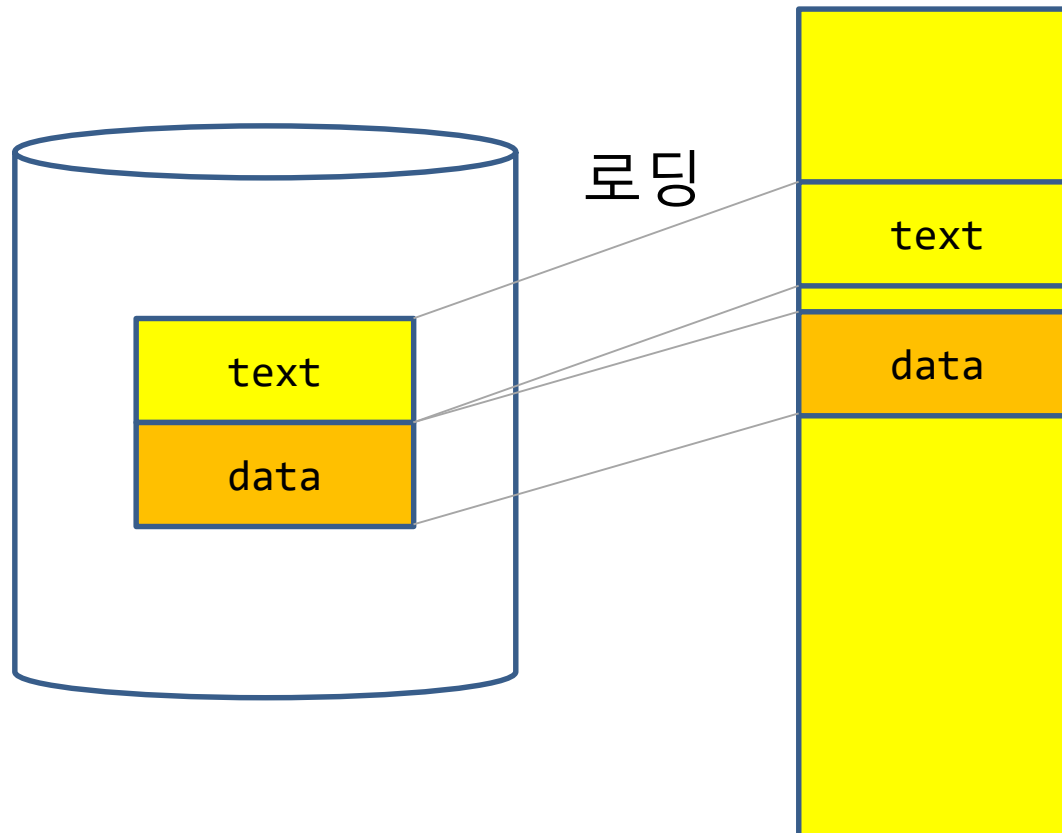
3.2 Process Control

3.3 Process Relationship

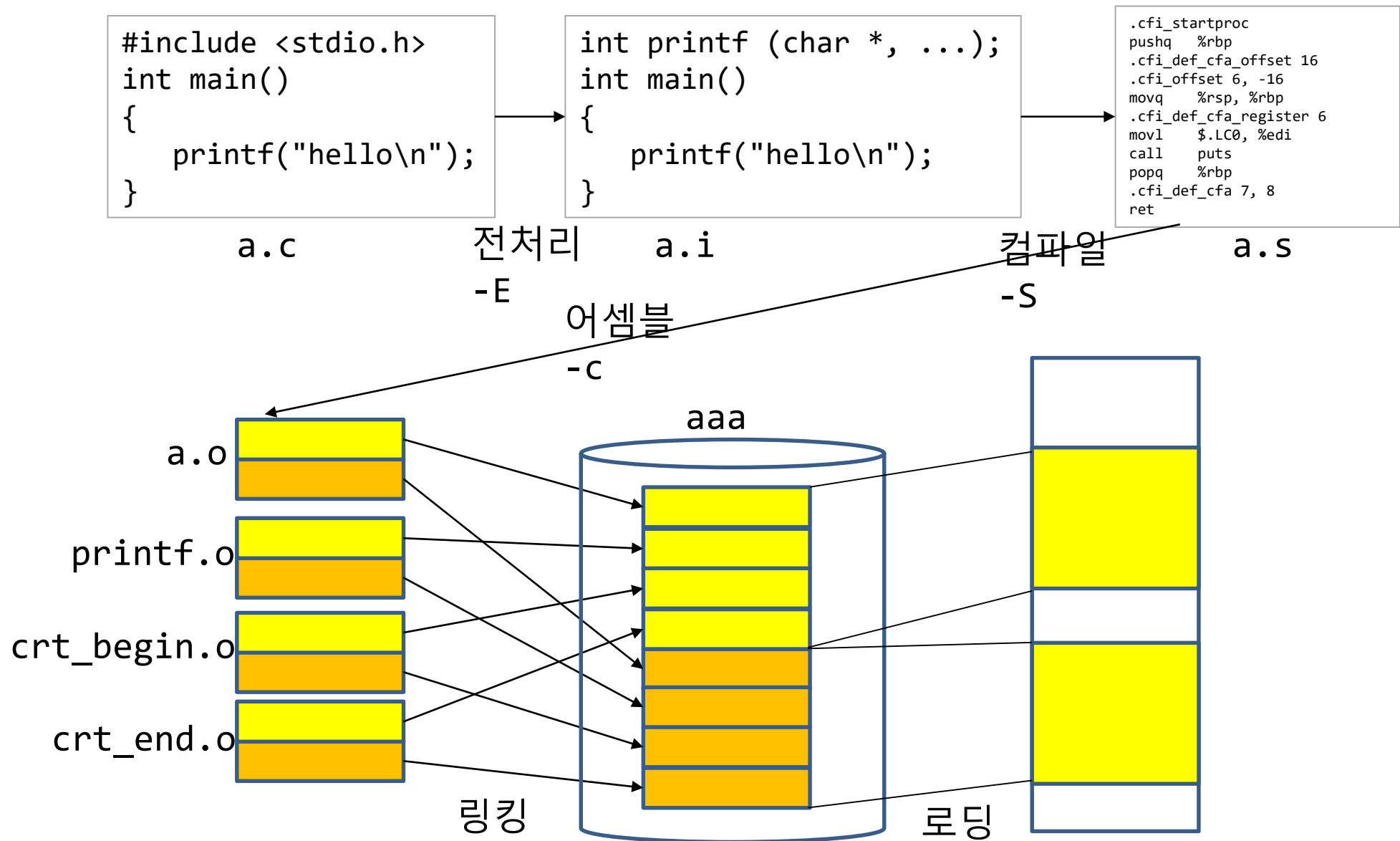
---

Process : 실행중인 프로그램  
Program : 실행가능한 파일

```
# ./aaa    system("aaa");
```







실행 파일 제작 과정

```
# gcc -v --save-temps a.c -o aaa
```

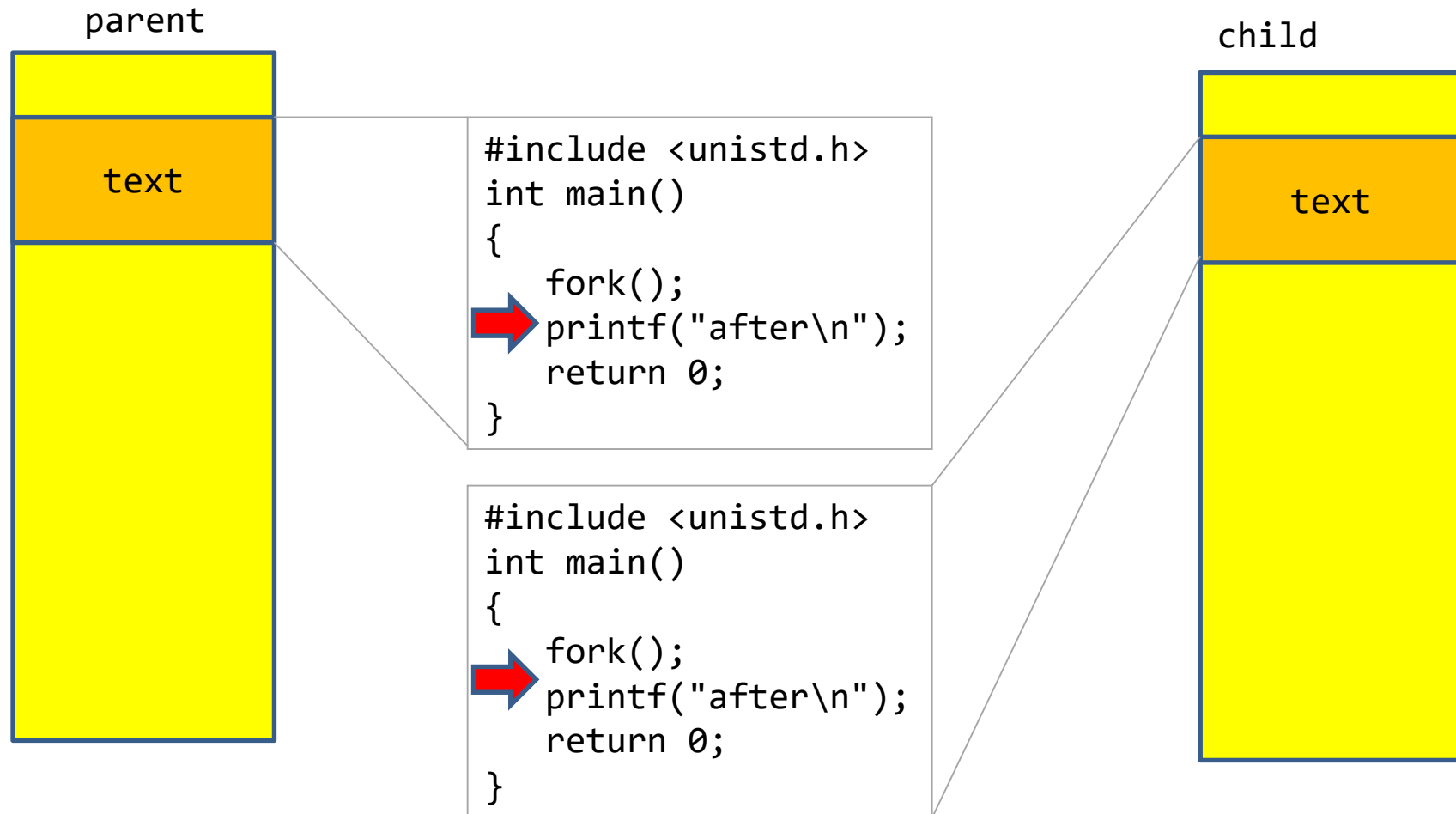
프로세스 생성 함수 fork()

: 아래 코드를 실행 하면 "after"이 두번 출력 된다. 이유는 무엇일까?

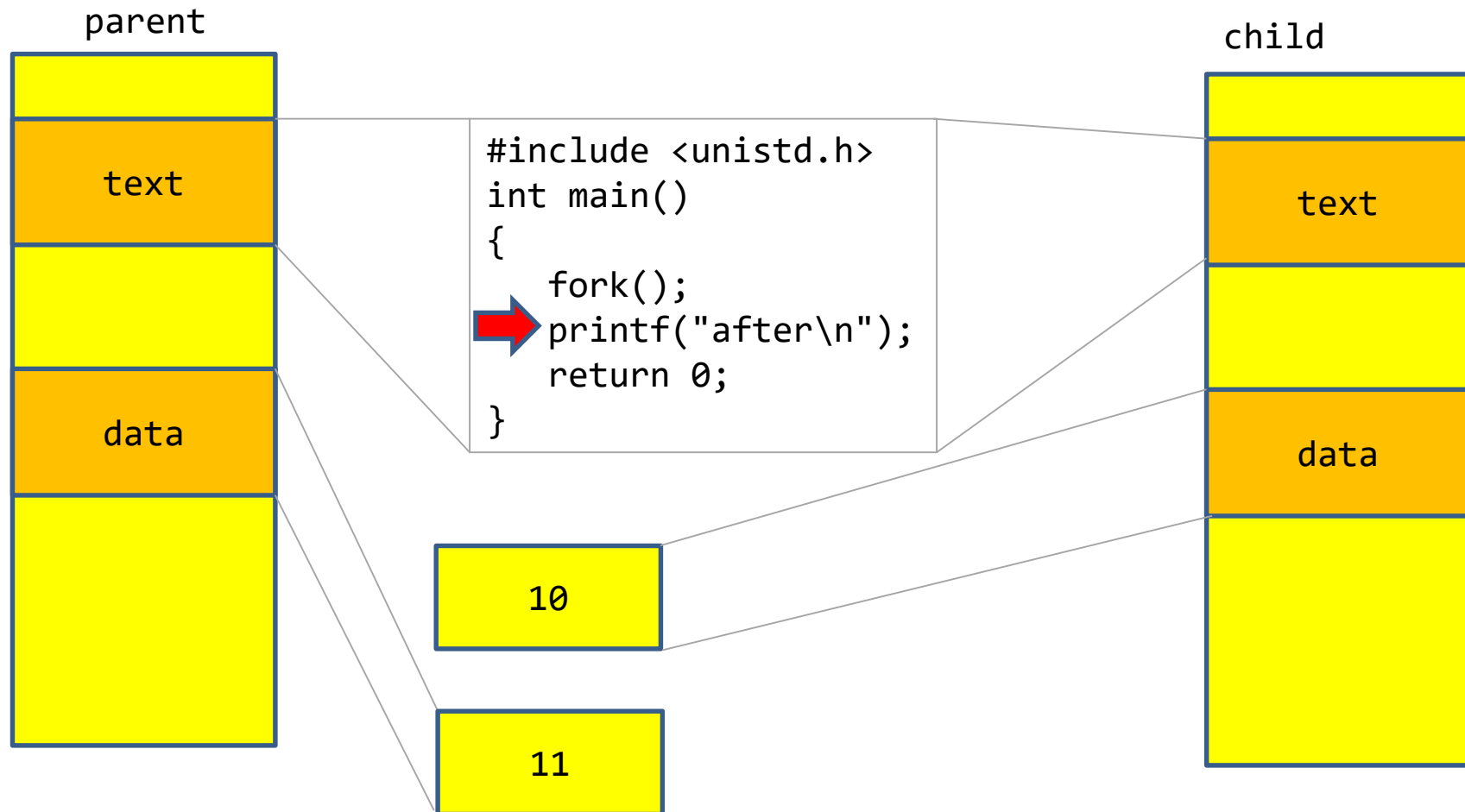
fork\_1.c

```
#include <unistd.h>
int main()
{
    fork();
    printf("after\n");
    return 0;
}
```

부모의 기계어 코드는 자식 프로세스에게 상속 된다.



COW(Copy On Write) : fork시 메모리 복제는 프로세스가 생성되었을때가 아니라 어느 한쪽이 write를 했을 때이다.



프로세스의 구분 : fork()를 하면 fork후에 부모와 자식의 제어의 흐름을 구분해야 한다.

fork는 인자가 없으므로 아래 처럼 구분 불가능

fork\_2.c

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    int status;
    fork(&status);
    if( status == 1 )
        printf("parent : after\n");
    else if( status == 2 )
        printf("child  : after\n");
    return 0;
}
```

프로세스의 구분 : fork후 리턴값은 부모쪽에는 자식의 pid가 자식쪽에는 pid로 사용하지 않는 0이 리턴된다.

fork\_3.c

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
int main()
{
    pid_t pid;
    pid = fork();
    if( pid > 0 )
        printf("parent : after\n");
    else if( pid == 0 )
        printf("child  : after\n");
    else
        perror("fork");
    return 0;
}
```

### ◆ 정상 종료

- main 함수로부터의 리턴 (exit 함수를 호출하는 것과 동일)
- exit 또는 \_exit 함수의 호출 (표준 C 라이브러리와 시스템 호출)
- 종료 상태 값을 명시적으로 지정

### ◆ 비정상 종료

- 자신이 abort 시스템 호출 (SIGABRT 시그널)
- 커널이 발생한 시그널(signal)에 의한 종료  
(0으로 나눈 경우, 잘못된 메모리 참조 등)
- 커널이 종료 상태값을 생성



### ◆ 프로세스의 종료작업

- 운영체제는 그 프로세스가 open한 file descriptor를 모두 close
- 프로세스가 차지하고 있던 메모리를 가용 메모리 풀(pool)로 반환
- exit 함수는 표준 입출력 정리 루틴을 수행하고 \_exit를 호출한다.  
(open된 file stream에 대해 fclose 호출, 버퍼에 남은 데이터를 flush)
- C 컴파일러는 main 함수에서 return하는 경우 자동적으로 exit 함수가 호출되도록 코드를 생성한다

### ◆ exit

- status - 종료상태 값
  - 표준 입출력 정리 루틴을 수행한 후, \_exit()를 호출한다.
  - open된 file stream에 대해 fclose 호출, 버퍼에 남은 데이터를 flush

### ◆ \_exit

- status - 종료상태 값
  - system call
- 자식의 종료상태 정보는 부모에게 전달될 필요성이 있다.(wait())
- status값은 하위 8byte만 사용된다.(0 ~ 255의 값)
- 보통 성공적으로 수행된 경우 0, 에러가 발생한 경우 0 이외의 값을 사용

fork\_4.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main(){
    pid_t pid;
    int i;
    pid = fork();
    if( pid == 0 ) {
        for(i=0; i<3; i++ ) {
            sleep(1);
            printf("\t\tchild\n");
        }
        exit(3);
    }
    while(1){
        sleep(1);
        printf("parent\n");
    }
    return 0;
}
```

fork의 사용 목적 :

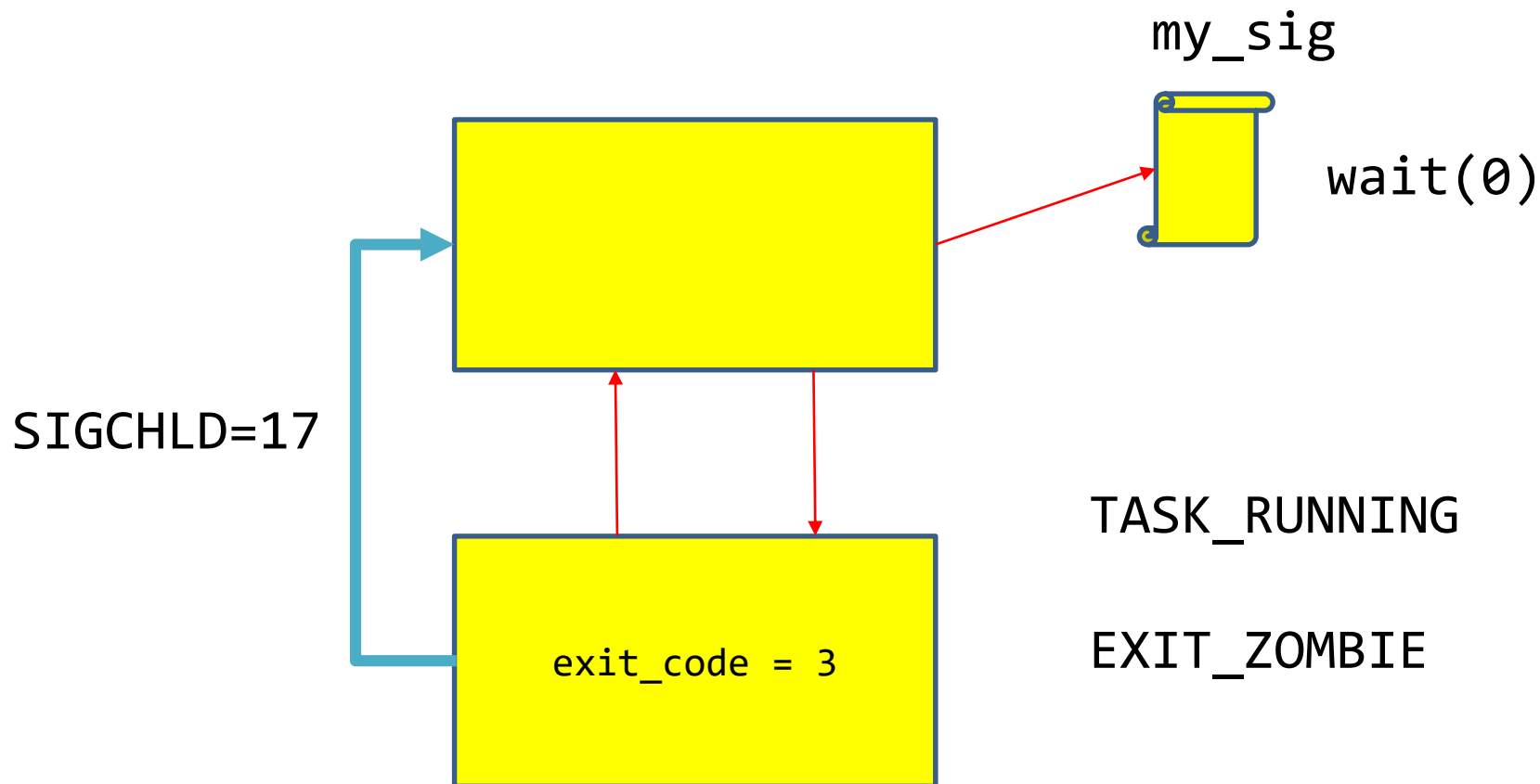
fork는 부모와 자식이  
동시에 실행 될 목적으로  
사용된다.

이를 병행성 이라 한다.

하지만 지금 코드는  
문제가 있다.

문제는 무엇인가?

프로세스의 제어권은 `exit` 시스템 콜로 잃게 되지만 프로세스의 몸체는 파괴되지 않는다. 이는 몸체 속에 자신의 종료 코드가 있기 때문이다. 부모는 자식의 종료 값을 `wait`로 가져 간후 몸체를 파괴한다.



### ◆ 고아(orphaned) 프로세스

- 자식 프로세스가 종료하기 전에 부모 프로세스가 수행을 마친 경우의 자식 프로세스
- 커널은 자식 프로세스를 찾아서 'init' 프로세스의 자식이 되도록 설정한다

### ◆ 좀비(zombie) 프로세스

- 종료했으나 부모 프로세스의 'wait' 처리가 끝나지 않은 프로세스
- 커널이 프로세스 ID, 종료 상태 값, CPU 사용 시간 등의 정보를 유지
- init 프로세스는 자식 프로세스가 종료할 때마다 wait 함수를 호출
- 지속적인 좀비 상태를 방지

fork\_5.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    pid_t pid;
    int i;
    pid = fork();
    if( pid == 0 ) {
        for(i=0; i<3; i++ ) {
            sleep(1);
            printf("\t\tchild\n");
        }
        exit(3);
    }
    wait(0);
    while(1){
        sleep(1);
        printf("parent\n");
    }
    return 0;
}
```

wait의 사용 :  
wait 시스템 콜은 자식의  
몸체를 파괴 하므로  
zombie의 문제를  
해결할 수 있다.

하지만 또 다른 문제가  
있다.

문제가 무엇인가?

fork\_6.c

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

void my_sig( int signo )
{
    printf("my_sig(%d)\n", signo );
    wait(0);
}

int main()
{
    pid_t pid;
    int i;
    signal( SIGCHLD, my_sig );
    ...
}
```

signal 의 사용 :

wait는 자식의 몸체를 수거하나  
blocking 연산이므로 부모가  
제어를 자식이 죽을때까지  
멈추게 된다. 이는 근본  
fork의 사용 목적인 병행성에  
위배 된다.

이는 시그널을 사용하면 해결  
할 수 있다.

하지만 다음 시그널의 처리는  
언뜻보면 훌륭 하나

문제가 있다.

문제는 무엇인가?

fork\_6.c

```
for( j=0; j<10; j++ )
{
    pid = fork();
    {
        for(i=0; i<3; i++ )
        {
            sleep(1);
            printf("\t\tchild\n");
        }
        exit(3);
    }
}
```

여러 프로세스 동시 종료시 문제  
: 프로세스가 여러개가 동시에  
종료한 경우는 시그널 핸들러가  
자식의 개수만큼 호출 되지 않을  
수도 있다.

그러면 자식 프로세스중 일부는  
여전히 zombie로 남게 된다.

프로세스의 개수를 늘려 가며  
그런 현상에 발생하는지 알아보자.

그러면 이 문제는 어떻게 해결해야  
하는가?



fork\_7.c

```
void my_sig( int signo )
{
    printf("my_sig(%d)\n", signo );
    while( wait(0) > 0 )
        ;
}
```

여러 프로세스 동시 종료시 문제 해결  
: 핸들러가 자식의 종료의 횟수 만큼  
호출 되지 않을 수도 있으므로  
시그널 핸들러 에서는 자식의 종료  
를 모두 수거 할때 까지 loop로  
wait를 실행 해야 한다.

하지만 이 경우에도 문제가 있다.

문제는 무엇인가?

fork\_8.c

```
for( j=0; j<10; j++ )
{
    pid = fork();
    if( pid == 0 )
    {
        for(i=0; i<j+1; i++ )
        {
            sleep(1);
            printf("\t\tchild\n");
        }
        exit(3);
    }
}
```

여러 프로세스 차등 종료시 문제

: 프로세스가 동시에 종료 되는 문제는 해결 했지만 프로세스가 종료 되었을때 아직 살아 있는 프로세스가 있을 경우는 해당 프로세스가 종료 될때까지

wait 시스템 콜은 blocking된다. wait가 blocking되면 signal handle가 멈추게 되고 signal handle가 멈추게 되면 부모 프로세스가 멈추게 되니

이는 병행성이 위배 되게 된다.

이를 해결 하려면?

fork\_9.c

```
void my_sig( int signo )
{
    printf("my_sig(%d)\n", signo );
    while( waitpid(-1, 0, WNOHANG) > 0 )
        ;
}
```

waitpid의 도입

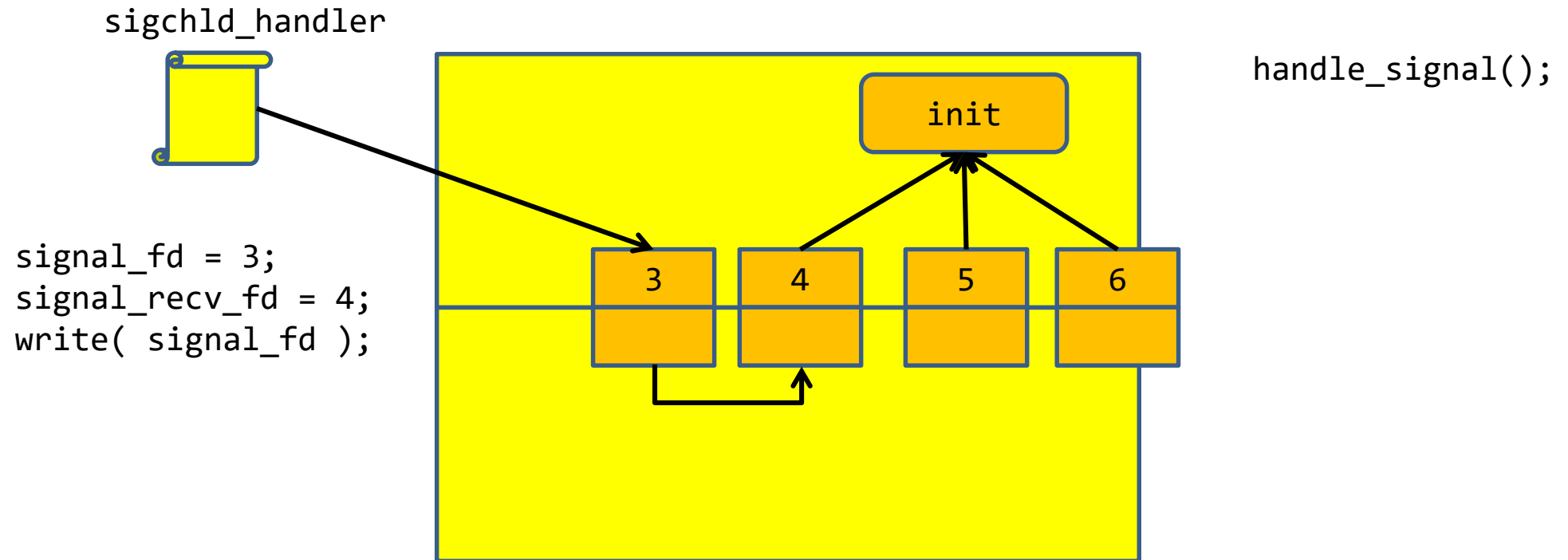
: waitpid는 wait의 family 함수로 기본적으로 wait와 동작이 같지만 flag를 추가로 지정 할 수 있어서 flag에 WNOHANG을 지정하면 살아 있는 프로세스가 있더라도 blockin되지 않고 즉각 0을 리턴한다.

안드로이드의 init 프로세스의 종료 처리

```
signal_init_action();
signal_init();
    struct sigaction act;
    memset(&act, 0, sizeof(act));
    act.sa_handler = sigchld_handler;
    act.sa_flags = SA_NOCLDSTOP;
    sigaction(SIGCHLD, &act, 0);

static void sigchld_handler(int s)
{
    write(signal_fd, &s, 1);
}
```

안드로이드의 init 프로세스의 종료 처리



# 3. Process Programming

---

3.1 Process Structure

**3.2 Process Control**

3.3 Process Relationship

---

표준 라이브러리를 이용한 ls의 실행 프로그램

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

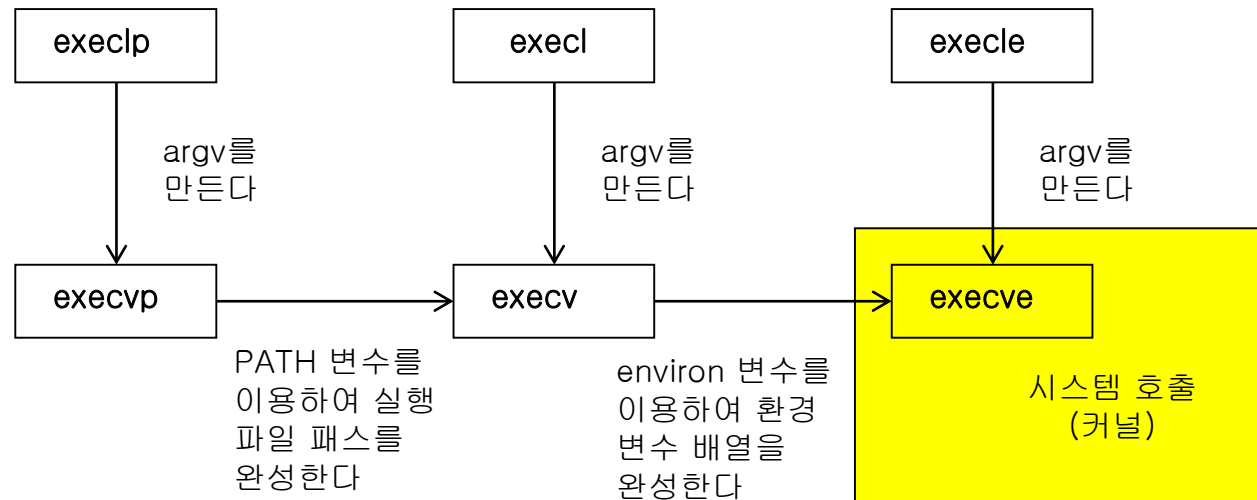
int main()
{
    system("ls -l");
    return 0;
}
```

### ◆ exec에 붙는 문자의 의미

- l : 리스트 형태의 명령라인 인자(argv[0],argv[1],argv[2]...)
- v : 벡터(vector) 형태의 명령 라인 인자(argv[])
- e : 환경변수 인자(envp[])
- p : path정보가 없는 실행파일 이름

### ◆ execve 함수만이 커널내의 시스템 함수

### ◆ 나머지 함수들은 라이브러리 함수로서 execve 함수를 호출한다.





exec\_1.c

```
#include <unistd.h>
int main()
{
    //exec1("/bin/ls", "ls", "-l", (char*)0 );
    exec1("./newpgm", "newpgm", "one", "two", (char*)0 );
    return 0;
}
```

newpgm.c

```
#include <stdio.h>

// # ./newpgm    one    two
//   argv[0]    argv[1]  argv[2]
//   argc=3
int main( int argc, char **argv )
{
    int i;

    for( i=0 ; i<argc; i++ )
        printf("argv[%d]=%s\n", i, argv[i] );

    return 0;
}
```

exec\_2.c

```
#include <unistd.h>
int main()
{
    char *argv[] = {"newpgm", "one", "two", (char*)0};
    execv("./newpgm", argv );
    return 0;
}
```

exec\_3.c

```
#include <unistd.h>
int main()
{
    char *argv[] = {"newpgm", "one", "two", (char*)0};
    char *env[] = {"name=LG", "addr=seoul", (char*)0};
    execve("./newpgm", argv, env );
    return 0;
}
```

newpgm\_1.c

```
#include <stdio.h>

// # ./newpgm    one    two
//  argv[0]    argv[1]  argv[2]
//  argc=3
int main( int argc, char **argv, char **envp )
{
    int i;

    for( i=0 ; i<argc; i++ )
        printf("argv[%d]=%s\n", i, argv[i] );
    printf("-----\n");

    for( i=0 ; envp[i]; i++ )
        printf("envp[%d]=%s\n", i, envp[i] );

    return 0;
}
```

exec\_4.c

```
#include <unistd.h>
int main()
{
    //execl("/bin/ls", "ls", "-l", (char*)0 );
    execlp("ls", "ls", "-l", (char*)0 );
    return 0;
}
```

### ◆ 새로운 프로세스의 속성

- 호출 전에 open된 파일 디스크립터들은 호출 후에도 그대로 유지되어 사용될 수 있다. (FD\_CLOEXEC이 세트되지 않은 경우)
- 새로 실행되는 프로그램의 set-user-ID가 세트되어 있는 경우 유효 사용자 ID가 프로그램 파일의 소유자 ID로 변경된다.
- 유효 그룹 ID도 새 프로그램의 set-group-ID 값에 따라 변경된다.
- 각 시그널에 대한 처리는 default 설정으로 복원된다.
- 단, 무시(SIG\_IGN)되고 있던 시그널은 계속 무시된다.

### ◆ 새로운 프로세스로 상속되는 속성

- 프로세스 ID, 부모 프로세스 ID
- 실제 사용자 ID, 실제 그룹 ID
- 프로세스 그룹 ID, 세션 (session) ID, 제어 터미널
- alarm 시그널까지 남은 시간
- 현재 작업 디렉토리, root 디렉토리
- 파일 생성 마스크, 파일 잠금
- 시그널 마스크, 보류된 시그널
- 자원 제약
- CPU 사용 시간 (tms\_utime,tms\_stime,tms\_cutime,tms\_ustime)



execve system call 이용한 ls의 실행 프로그램

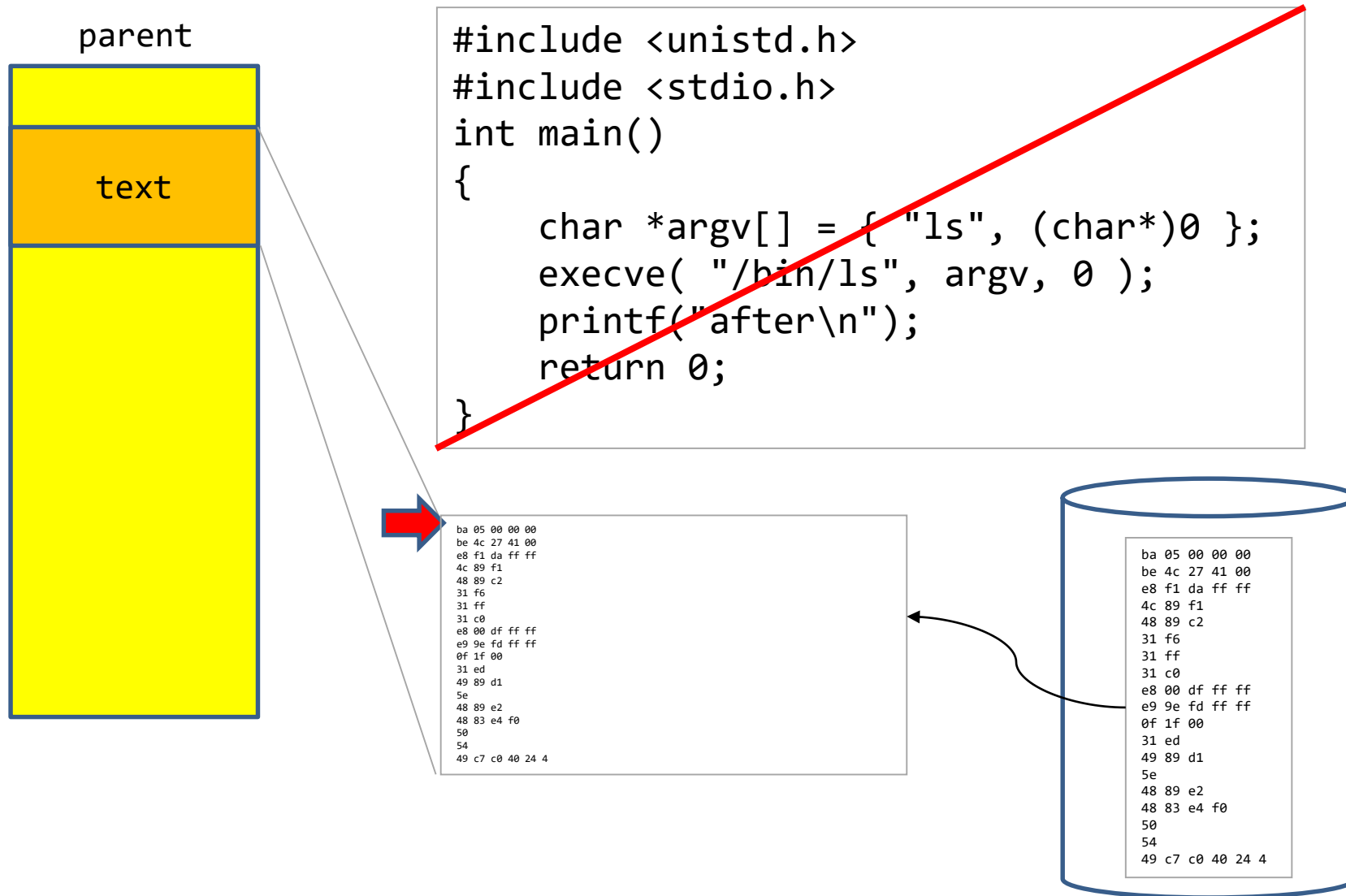
execve\_1.c

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    char *argv[] = { "ls", (char*)0 };
    execve( "/bin/ls", argv, 0 );
    printf("after\n");
    return 0;
}
```

execve 실행후 after은  
출력 되지 않는다.

이유는 무엇이고

해결책은 무엇인가?



execve system call 이용한 ls의 실행 프로그램

execve\_2.c

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    char *argv[] = { "ls", (char*)0 };
    printf("prompt> ls\n");
    execve( "/bin/ls", argv, 0 );
    printf("prompt> \n");
    return 0;
}
```

execve system call 이용한 ls의 실행 프로그램

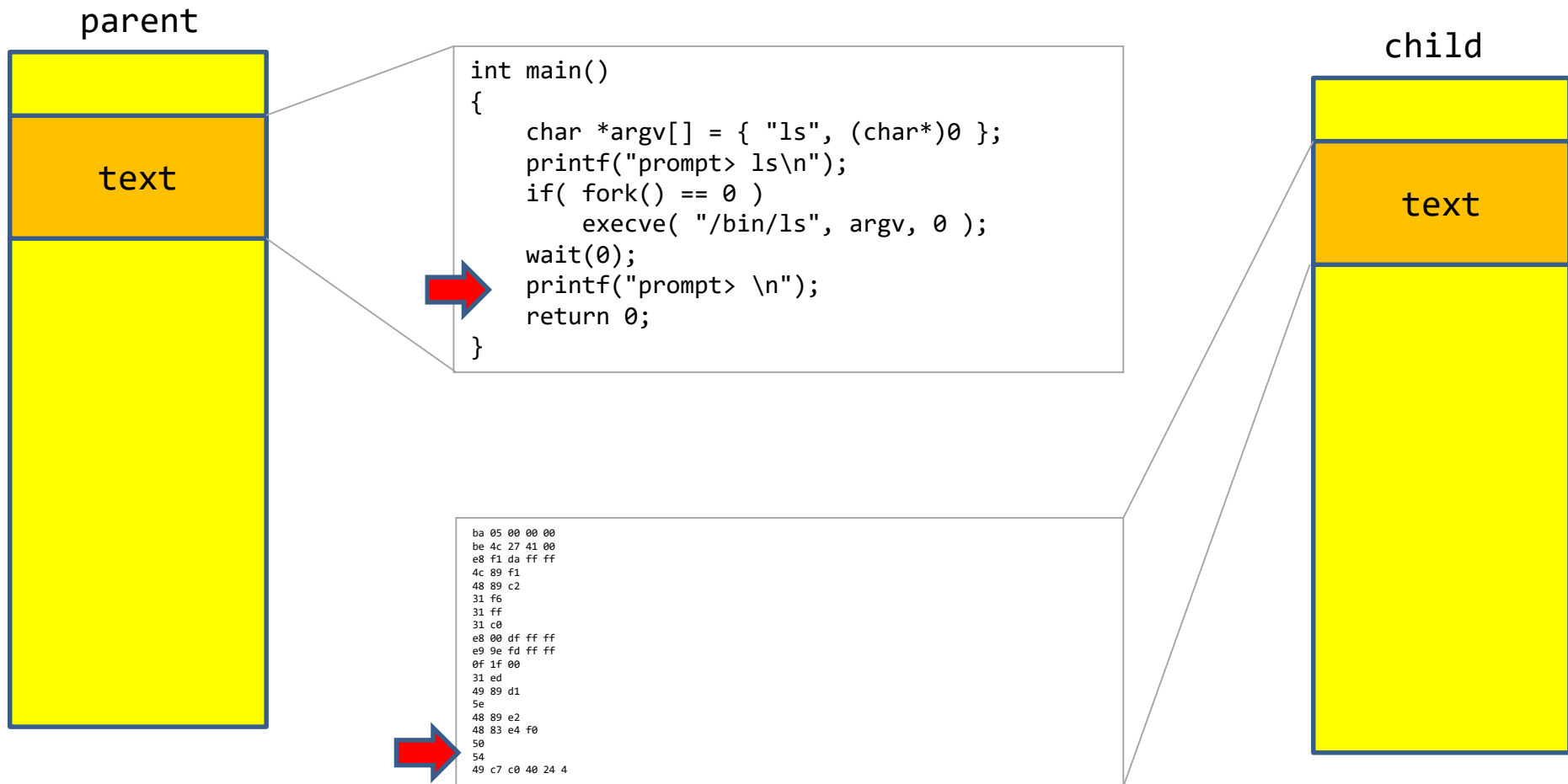
execve\_3.c

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
int main()
{
    char *argv[] = { "ls", (char*)0 };
    printf("prompt> ls\n");
    if( fork() == 0 )
        execve( "/bin/ls", argv, 0 );
    wait(0);
    printf("prompt> \n");
    return 0;
}
```

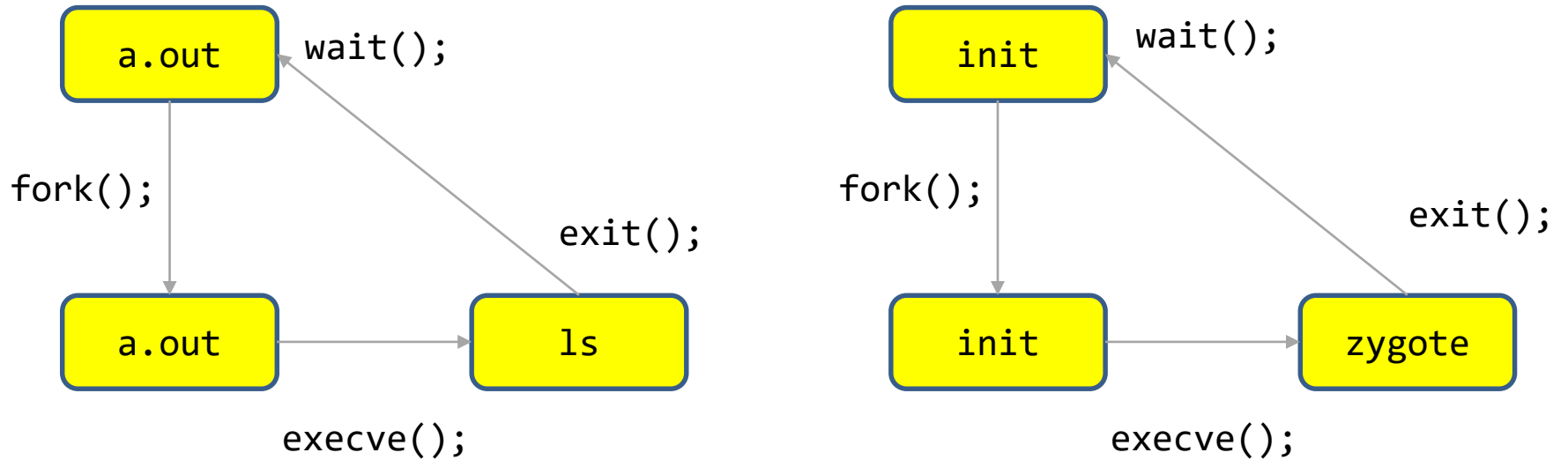
fork를 execve와 함께 사용하면 문제를 해결 할수 있다.

wait는 자식 프로세스가 종료하면 리턴 되므로

동기화도 자연스럽게 해결된다.



프로세스는 fork로 생성되어 execve로 실행되며 exit로 종료하고 wait로 소멸된다.



execve system call 이용한  
system 함수의 구현 프로그램

system.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int my_system( char *cmd){
    char *argv[] = { cmd , (char*)0 };
    char buff[128] = "/bin/";
    pid_t pid;
    int status=0;
    strcat( buff, cmd );
    pid = fork();
    if( pid == 0 ) {
        execve( buff, argv, 0 );
        _exit(127);
    } else if ( pid < 0 ){
        status = -1;
    } else{
        waitpid(pid, &status, 0 );
    }
    return status;
}
```

wait함수의 인자인 status는 2byte의 유효값을 사용하지만 프로세스의 정상적인 exit 종료와 signa에 의한 비정상 종료를 처리 하기 위해 상위 1byte와 하위 1byte가 상호 배타적으로 사용된다.



```
kill -2 1234
```



exit 시스템 콜에 의한 프로세스의 정상적인 종료

wait\_1.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int status;
    if( fork() == 0 )
        exit(7);

    wait(&status);
    if( (status & 0xff) == 0 )
        printf("정상종료 : exit(%d)\n", (status>>8)&0xff );
    return 0;
}
```

## 정상 vs 비정상 종료 처리 구분

wait\_2.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
int main()
{
    int status;
    if(fork()==0)
        while(1);

    wait(&status);
    if( (status & 0xff) == 0 )
        printf("정상종료 : exit(%d)\n", (status>>8)&0xff );
    else
        printf("비정상종료 : signo(%d)\n", status&0xff );
    return 0;
}
```

# kill -3 1234 에 의한 비정상 종료 처리

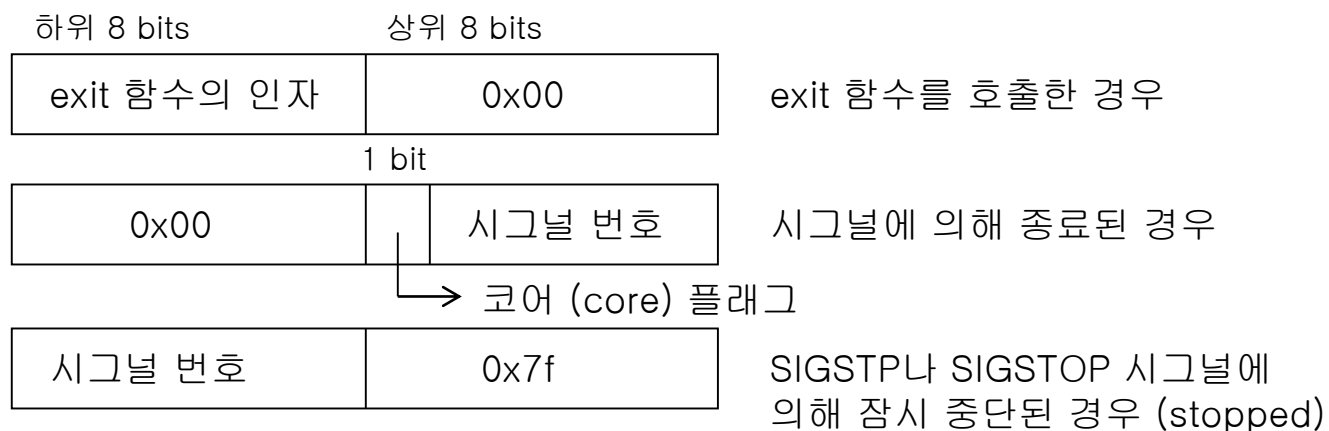
wait\_3.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
int main()
{
    int status;
    if(fork()==0)
        while(1);

    wait(&status);
    if( (status & 0x7f) == 0 )
        printf("정상종료 : exit(%d)\n", (status>>8)&0xff );
    else
    {
        printf("비정상종료 : signo(%d) %s\n", status&0xff,
            (status&0x80)?"(core dumped)":"" );
    }
    return 0;
}
```

### wait

- statloc - 종료된 프로세스의 상태값 저장, 매크로 지원



- 자식 프로세스가 여럿인 경우 그 중 하나만 종료해도 리턴한다.
- 부모 프로세스에 발생한 시그널에 의해서도 리턴될 수 있다.
- 자식 프로세스가 없는 경우에는 -1을 리턴

### ◆ statloc에 대한 매크로 함수

WIFEXITED(status)	정상적으로 종료한 경우에 참 값을 리턴
WEXITEDSTATUS(status)	exit 함수의 인자에서 하위 8 비트 값을 리턴
WIFSIGNALED(status)	시그널에 의해 종료한 경우에 참 값을 리턴
WTERMSIG(status)	시그널 번호를 리턴
WCOREDUMP(status)	코어 파일이 생성된 경우에 참 값을 리턴
WIFSTOPPED(status)	실행이 일시 중단된 경우에 참 값을 리턴
WSTOPSIG(status)	실행을 일시 중단시킨 시그널 번호를 리턴

wait\_4.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
void term_stat( int status )
{
    if( WIFEXITED(status) )
        printf("정상종료 : exit(%d)\n", WEXITSTATUS(status) );
    else if( WIFSIGNALED(status) )
    {
        printf("비정상종료 : signo(%d) %s\n", WTERMSIG(status),
            (WCOREDUMP(status))?"(core dumped)": "" );
    }
}
int main()
{
    int status;
    if(fork()==0)
        while(1);

    wait(&status);
    term_stat(status);
    return 0;
}
```

# 3. Process Programming

---

3.1 Process Structure

3.2 Process Control

**3.3 Process Relationship**

---

### ◆ 프로세스 그룹

- 하나 이상의 프로세스들의 집합
- 일관된 작업을 하는 프로세스들
- 프로세스 그룹 ID를 가진다

### ◆ 프로세스 그룹 리더(leader)

- 프로세스 그룹 ID와 동일한 값을 프로세스 ID로 가진 프로세스
- 프로세스 그룹 및 그 그룹 내의 프로세스를 생성, 종료시킬 수 있다

### ◆ 프로세스 그룹 수명(lifetime)

- 프로세스 그룹이 생성되어 그룹 내의 마지막 프로세스가 그룹을 떠날 때까지의 기간
- 그룹 리더의 존재 여부와 상관없다



### ◆ getpggrp

- 호출한 프로세스가 속한 프로세스 그룹 ID를 리턴한다.

### ◆ setpgid(pid,pgid)

- 새로운 프로세스 그룹을 생성하거나 이미 존재하는 프로세스 그룹에 합류.
- 두 인자가 동일한 경우에는 지정한 프로세스가 프로세스 그룹 리더가 된다.
- pid 인자가 0이면 호출한 프로세스의 ID가 pid로 사용된다.
- pgid 인자가 0이면 pid 인자가 pgid로 쓰인다.
- 자기 자신이나 자식 프로세스의 프로세스 그룹 ID만을 변경할 수 있다.
- exec 함수를 수행한 자식 프로세스의 프로세스 그룹 ID는 변경할 수 없다.

### ◆ 프로세스 세션

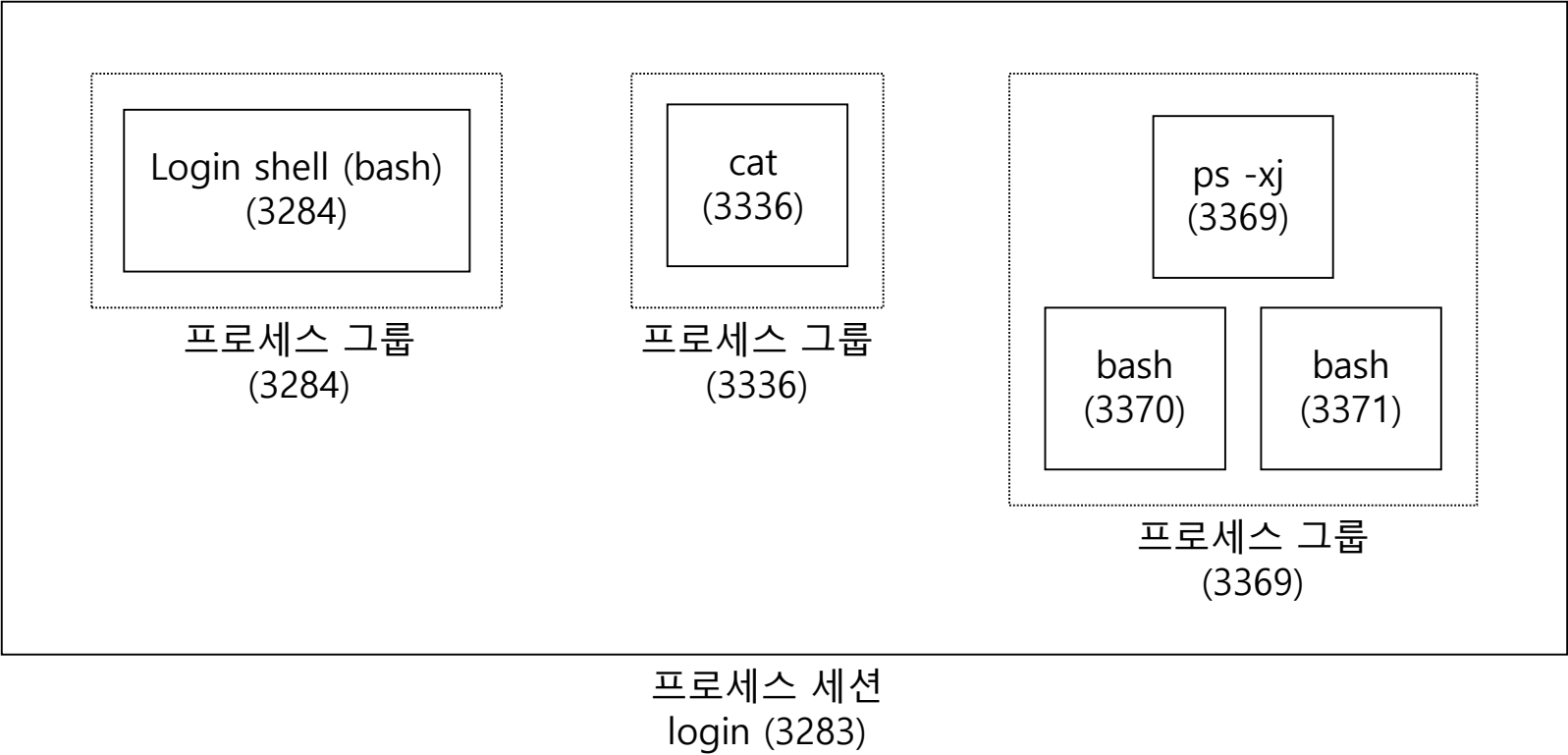
- 하나 이상의 프로세스 그룹들의 집합
- 보통 동일한 터미널에서 수행되고 있는 프로세스 그룹들

```
# cat > temp &
```

```
# ps -xj | more | more
```

PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND
3283	3284	3284	3283	pts/0	3369	R	500	0:00	-bash
3284	3336	3336	3283	pts/0	3369	T	500	0:00	cat
3284	3369	3369	3283	pts/0	3369	R	500	0:00	ps -xj
3284	3370	3369	3283	pts/0	3369	R	500	0:00	-bash
3284	3371	3369	3283	pts/0	3369	R	500	0:00	-bash

◆ 프로세스 세션



### ◆ setsid

#### ◆ 호출한 프로세스가 프로세스 그룹 리더가 아니면 새 세션을 생성한다.

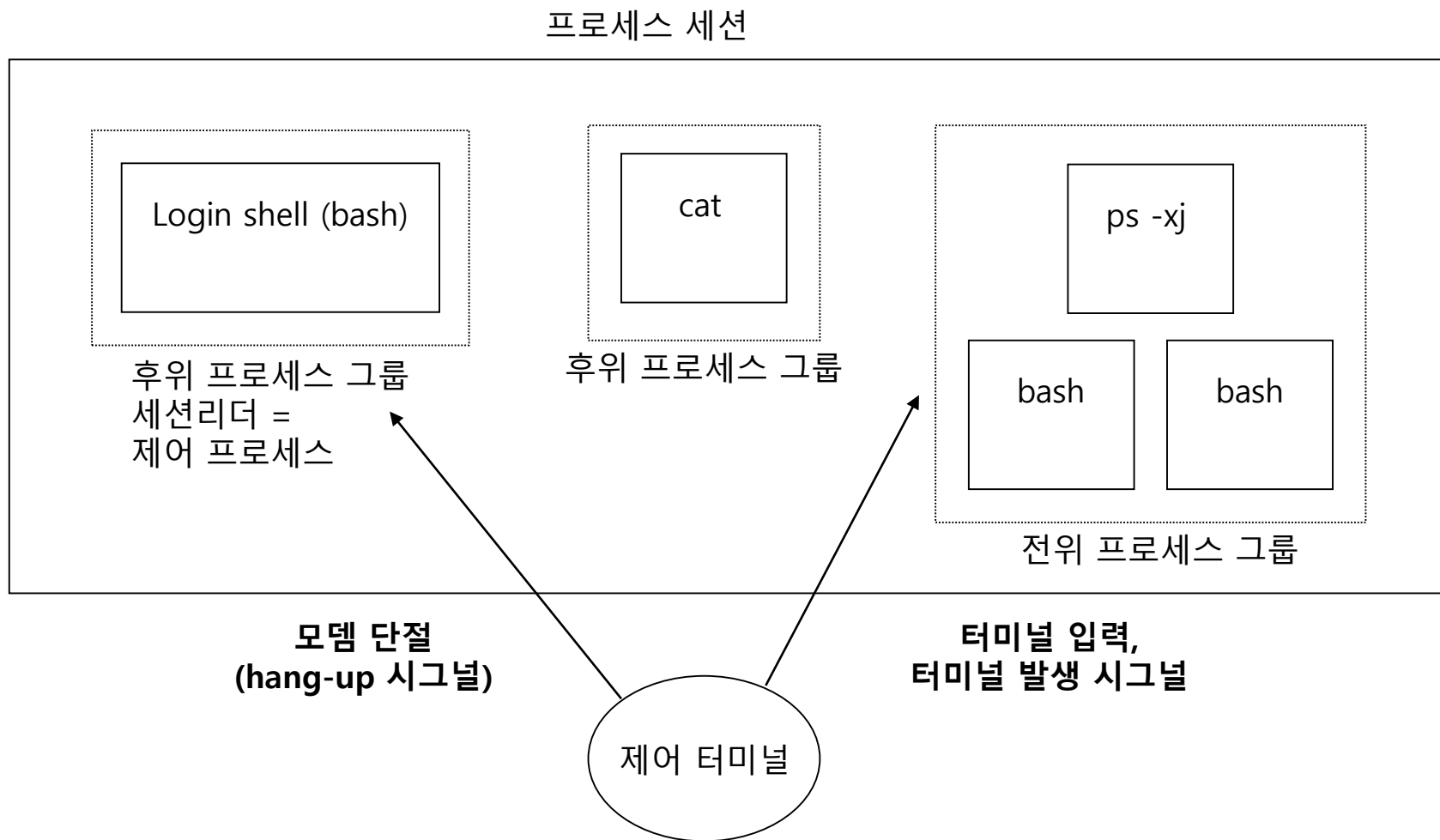
- 생성된 세션의 유일한 프로세스로서 세션 리더가 된다.
- 새 프로세스 그룹의 프로세스 그룹 리더가 된다.
- 제어 터미널을 갖지 않는다. 연결 되어 있던 제어 터미널과는 단절된다.
- 그룹 리더인 경우에 setsid는 실패하고 에러를 리턴한다

#### ◆ fork를 호출한 후 부모는 종료시키고 자식 프로세스로서 수행을 계속하여 setsid 함수를 호출한다.(그룹 리더를 만들지 않기 위해)

### ◆ 세션과 제어 터미널

- 한 세션은 하나의 제어 터미널을 가질 수 있다. (통상 로그인 작업시)
- 세션 리더는 제어 터미널과의 연결을 설정하는 제어 프로세스이다.
- 한 세션 내의 프로세스 그룹은 하나의 전위(foreground) 프로세스 그룹과 하나 이상의 후위(background) 프로세스 그룹으로 구분된다.
- 제어 터미널에서 특수키(DELETE, ^C, ^W)를 입력하면 전위 프로세스 그룹의 모든 프로세스에게 해당하는 시그널(interrupt or quit)이 전달된다.
- 모뎀과의 연결이 끊어지면 제어 프로세스(세션 리더)에게 hang-up 시그널이 전달된다

### ◆ 시그널과 세션 및 프로세스 그룹



### ◆ tcgetpgrp

- filedes 인자가 가리키는 터미널에 연결된 전위 프로세스 그룹의 프로세스 그룹 ID를 리턴한다.

### ◆ tcsetpgrp

- filedes - 세션의 제어 터미널
- pgrp\_id - 같은 세션에 속한 프로세스 그룹의 프로세스 그룹 ID
  - 제어 터미널을 가지고 있는 경우, pgrp\_id가 가리키는 프로세스 그룹을 전위 프로세스 그룹으로 설정한다.

### ◆작업

- 일관된 일을 수행하는 하나 이상의 프로세스들의 집합 (전위, 후위)

### ◆작업제어

- 작업들을 제어 터미널 안에서 전위 또는 후위로 변경
- 하나의 제어 터미널에서 다중 작업의 수행이 가능

### ◆작업제어를 위한 조건

- 셸이 작업 제어를 지원해야 한다.
- 커널의 터미널 제어기(terminal driver)가 작업 제어를 지원해야 한다.
- 작업 제어 시그널이 제공 되어야 한다



### ◆작업제어에 사용되는 특수키

- ^C 혹은 DELETE : 인터럽트 시그널 (SIGINT) 발생, 프로세스의 종료
- ^₩ : quit 시그널 (SIGQUIT) 발생, 코어 (core) 파일을 만들고 종료
- ^Z : suspend 시그널 (SIGTSTP) 발생, 프로세스의 수행을 중지

### ◆ 정의

- 어떤 사건이 발생할 때까지 기다리거나, 주기적으로 주어진 일을 수행하기 위해 기다리는 background에 있는 프로세스

### ◆ 특징

- 제어 터미널을 가지지 않으며 후위로 실행된다.
- 보통 시스템이 부팅될 때 시작되며, 셧다운(shutdown)될 때 종료한다.
- 유닉스 시스템의 일상적인 작업을 수행한다.  
( 스케줄링, 네트워크 감시, 이메일 송수신, 프린터 제어 등)
- 다른 프로세스가 발생한 시그널에 의해 간섭 받지 않아야 한다.

◆ 예 (ps -ejf)

UID	PID	PPID	PGID	SID	C	STIME	TTY	TIME	CMD
root	1	0	1	1	0	18:47	?	00:00:02	/sbin/init splash
root	2	0	0	0	0	18:47	?	00:00:00	[kthreadd]
root	4	2	0	0	0	18:47	?	00:00:00	[kworker/0:0H]
root	6	2	0	0	0	18:47	?	00:00:00	[ksoftirqd/0]
root	7	2	0	0	0	18:47	?	00:00:00	[rcu_sched]
root	8	2	0	0	0	18:47	?	00:00:00	[rcu_bh]
root	9	2	0	0	0	18:47	?	00:00:00	[migration/0]
root	10	2	0	0	0	18:47	?	00:00:00	[lru-add-drain]
root	11	2	0	0	0	18:47	?	00:00:00	[watchdog/0]

### ◆ 실행 방법

- the system initialization scripts ( /etc/rc )
- the inetd superserver
- cron daemon
- the at command
- from user terminals

### ◆ 코딩 규칙

1. background로 수행되도록 한다.
  - fork()를 호출하여 parent를 종료한다.
  - child 프로세스는 프로세스 그룹의 리더가 아니다.

```
if(fork()>0)  
    exit(0);
```

2. 새로운 세션을 생성한다.
  - 세션의 리더가 된다.
  - 제어 터미널과 단절된다.

```
setsid();
```

### ◆ 코딩 규칙

3. 열려진 모든 file descriptor를 닫는다.
  - daemon을 실행시킨 프로세스로 부터 상속받은 모든 fd를 닫는다.  
for(l=0;l<64;l++)  
    close(l);
4. 작업 디렉토리를 root로 바꾼다.
  - mount된 파일 시스템을 unmount 가능하도록 하기 위해  
    chdir("/");
5. 파일 생성 마스크를 제거한다.
  - parent(자기를 기동한 프로세스)로 부터 물려받은 umask를 제거  
    umask(0);
6. SIGCLD 시그널을 처리한다.
  - 자식이 발생하는 SIGCHLD를 대응 하지 않으면 zombie가 된다.  
    signal(SIGCHLD,SIG\_IGN);

daemon\_1.c

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

int daemon_init(void){
    pid_t  pid;
    int i;
    if(fork() > 0)
        exit(0);
    setsid();           /* become session leader */
    chdir("/");         /* change working directory */
    umask(0);           /* clear our file mode creation mask */

    for(i=0;i<64;i++)
        close(i);
    signal(SIGCLD,SIG_IGN);
    return(0);
}
```

**daemon\_1.c**

```
main(void)
{
    daemon_init();
    sleep(20);
}
```



# 4. Pthread Programming

# 4. Pthread Programming

---

## 4.1 Pthread 개요

4.2 Pthread API

4.3 Thread 동기화

4.4 Thread 재진입 가능함수 구현

---

### ◆ 프로그램 개념

: 당장 실행 가능한 파일 ( ELF 포맷 )

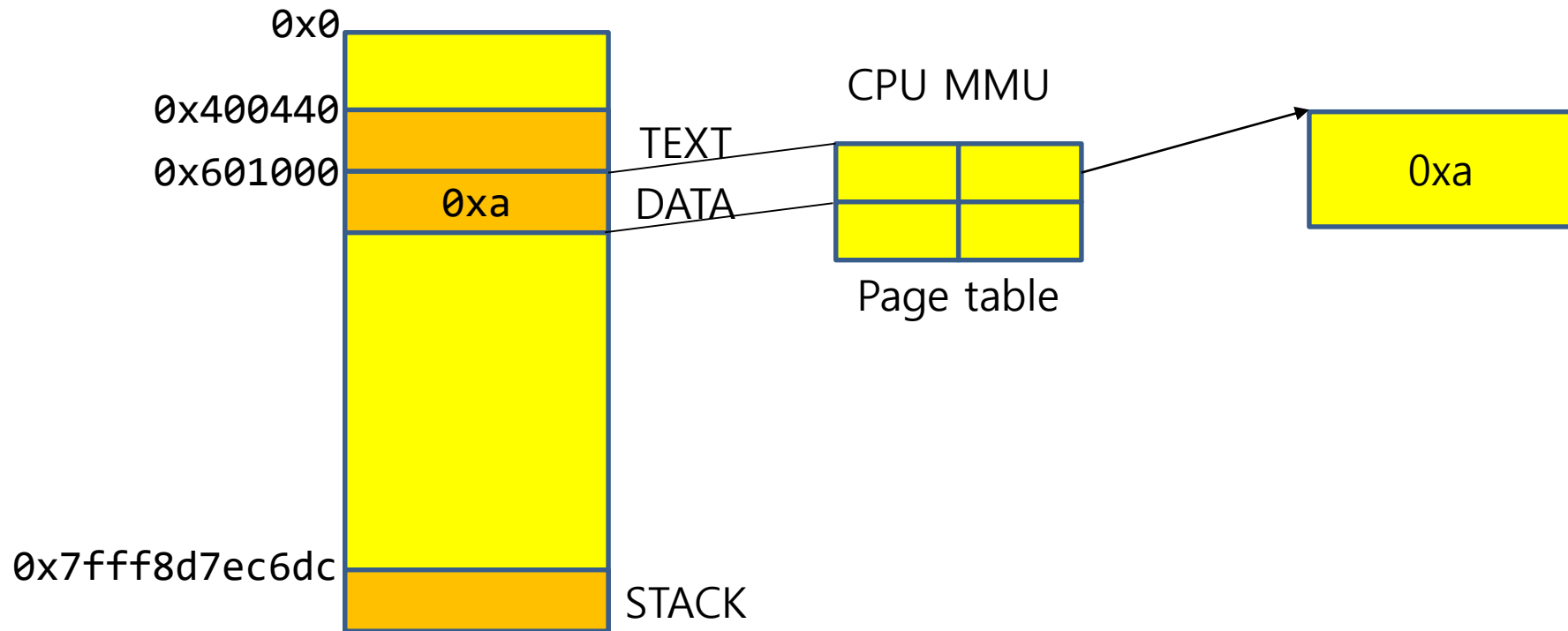
### ◆ Process 개념

: 실행 중인 프로그램 ( 32bit 보호 모드 이상 )

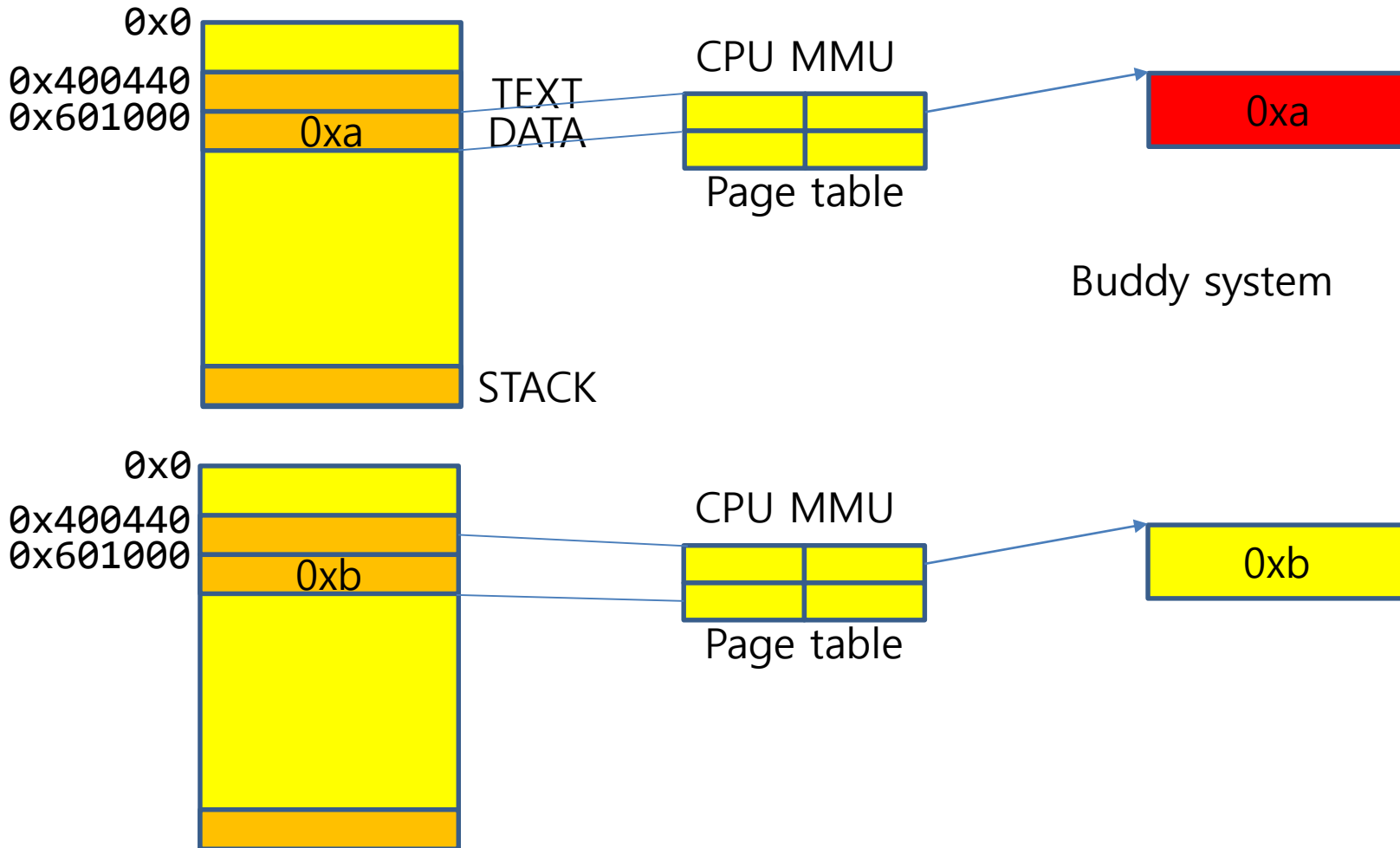
### ◆ Thread 개념

: 가상 메모리를 공유한 프로세스 ( parent, LWP )

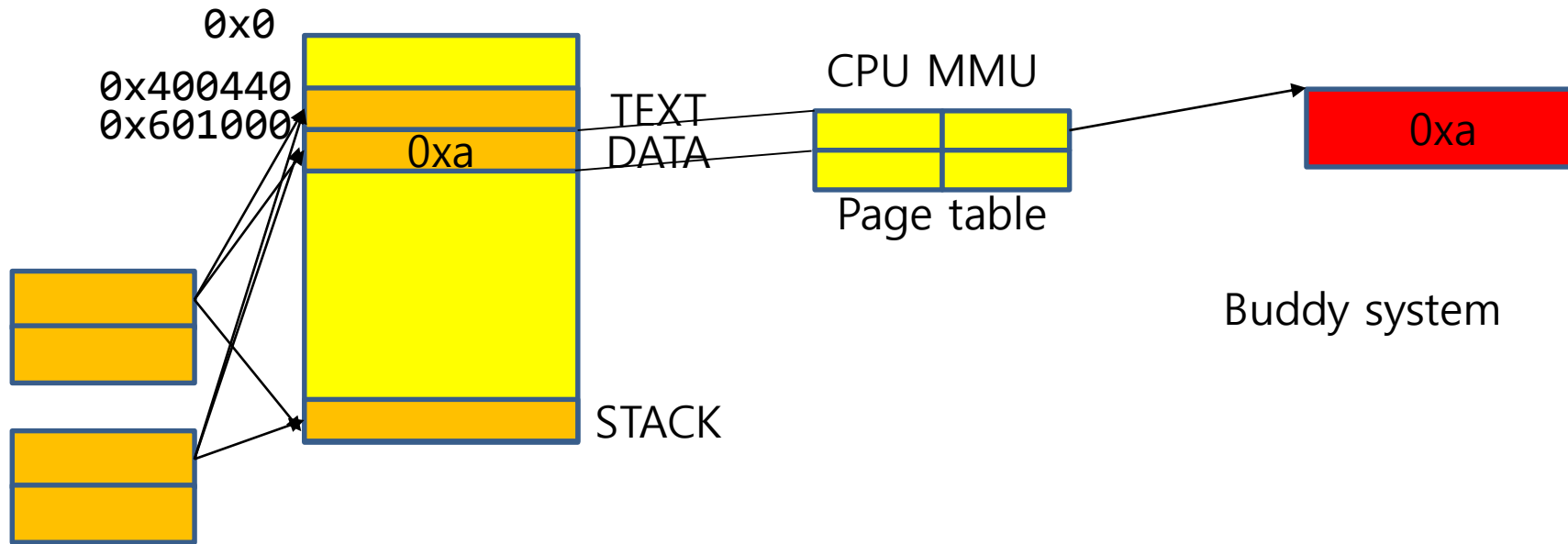
프로세스의 가상 메모리는 할당된 물리적 메모리와 매핑 된다.



프로세스가 다르다면 절대 같은 물리 메모리를 매핑 하지 않는다.



Thread는 parent 프로세스의 가상 메모리 맵을 공유 한다.



# 4. Pthread Programming

---

4.1 Pthread 개요

**4.2 Pthread API**

4.3 Thread 동기화

4.4 Thread 내부구조 분석

---

errors.h

```
#ifndef __errors_h
#define __errors_h
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef DEBUG
# define DPRINTF(arg) printf arg
#else
# define DPRINTF(arg)
#endif
#define err_abort(code,text) do { \
    fprintf(stderr, "%s at \"%s\":%d: %s\n", \
        text, __FILE__, __LINE__, strerror(code)); \
    abort(); \
}while(0)
#define errno_abort(text) do { \
    fprintf(stderr, "%s at \"%s\":%d: %s\n", \
        text, __FILE__, __LINE__, strerror(errno)); \
    abort(); \
}while(0)
#endif
```



lifecycle.c

```
#include <pthread.h>
#include "errors.h"
void *thread_routine( void * arg ){
    return arg;
}

int main( int argc, char **argv ){
    pthread_t thread_id;
    void *thread_result;
    int status;

    status = pthread_create( &thread_id,0,thread_routine,0);
    if( status != 0 )
        err_abort( status, "Create thread");

    status = pthread_join( thread_id, &thread_result );
    if( status != 0 )
        err_abort( status, "Join thread");
    if( thread_result == NULL )
        return 0;
    else
        return 1;
}
```

빌드 하기

```
# gcc lifecycle.c -lpthread
```

실행 하기

```
# ./a.out
```

빌드 하기 ( 파일 이름 설정 )

```
# gcc lifecycle.c -o lifecycle -lpthread
```

실행 하기

```
# ./lifecycle
```

중복 헤더 파일 include test

lifecycle.c

```
#include "errors.h"
```

```
#include "errors.h"
```

전처리 까지만 하라.

```
# gcc -E lifecycle.c -lpthread
```

조건부 컴파일 include test

lifecycle.c

```
DPRINTF( ("hello\n") ); => printf("hello\n");
```

전처리 까지만 하라.

```
# gcc -DDEBUG -E lifecycle.c -lpthread
```

thread 의 실행 확인  
lifecycle.c

```
void *thread_routine( void * arg )  
{  
    sleep(100);  
    return arg;  
}
```

확인 명령 ( 다른 터미널 에서 )

```
# ps -eLf | grep a.out | grep -v grep
```

pthread\_join blocking 테스트

lifecycle.c

```
void *thread_routine( void * arg )
{
    sleep(3);
    return arg;
}
...
if( thread_result == NULL )
{
    DPRINTF(("자식 쓰레드 종료\n"));
    return 0;
}
```

```
# gcc -DDEBUG lifecycle.c -lpthread
```

main thread 종료 pthread\_exit 테스트

lifecycle.c

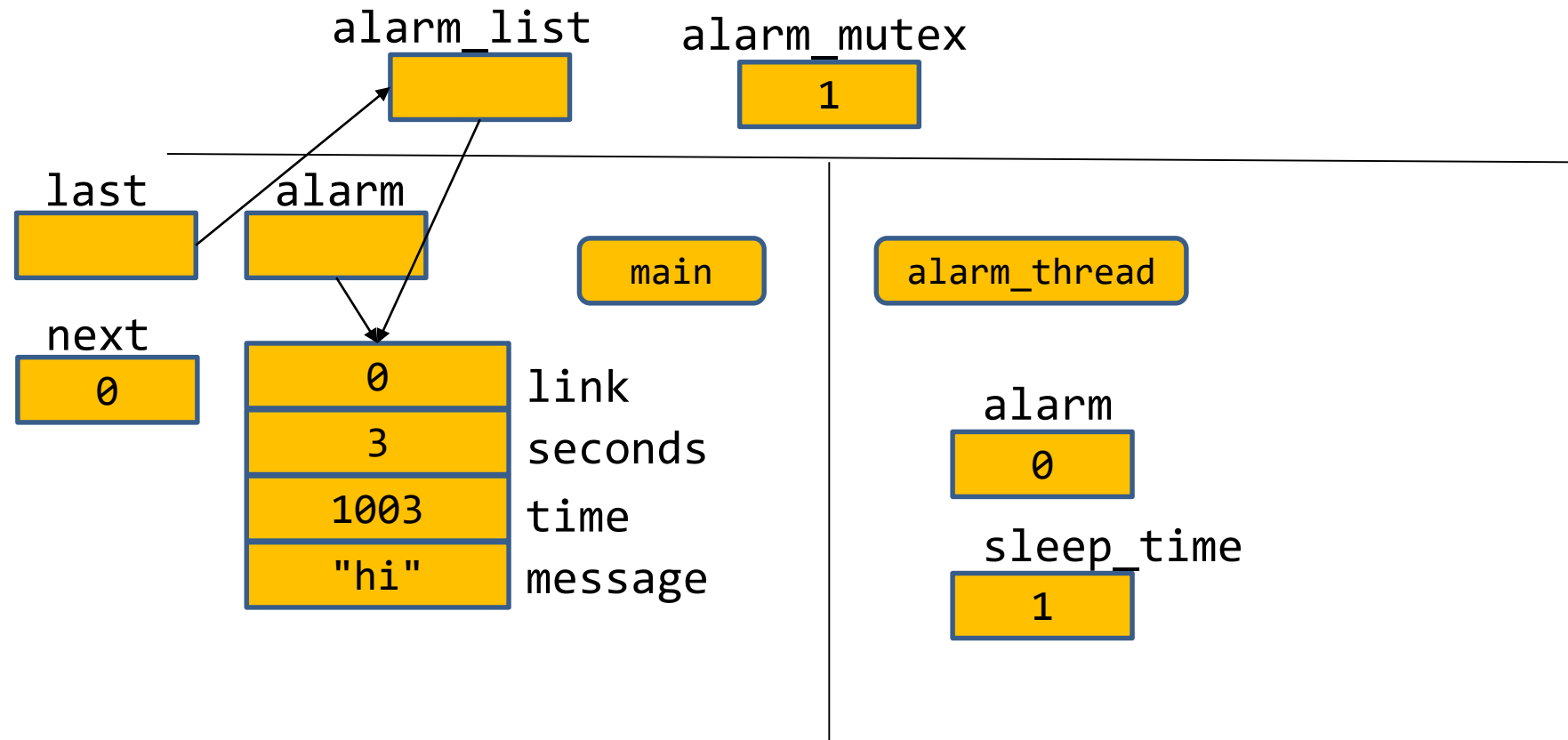
```
void *thread_routine( void * arg )
{
    sleep(1000);
    return arg;
}
...
pthread_exit(0);
/*
    status = pthread_join( thread_id, &thread_result );
*/
```

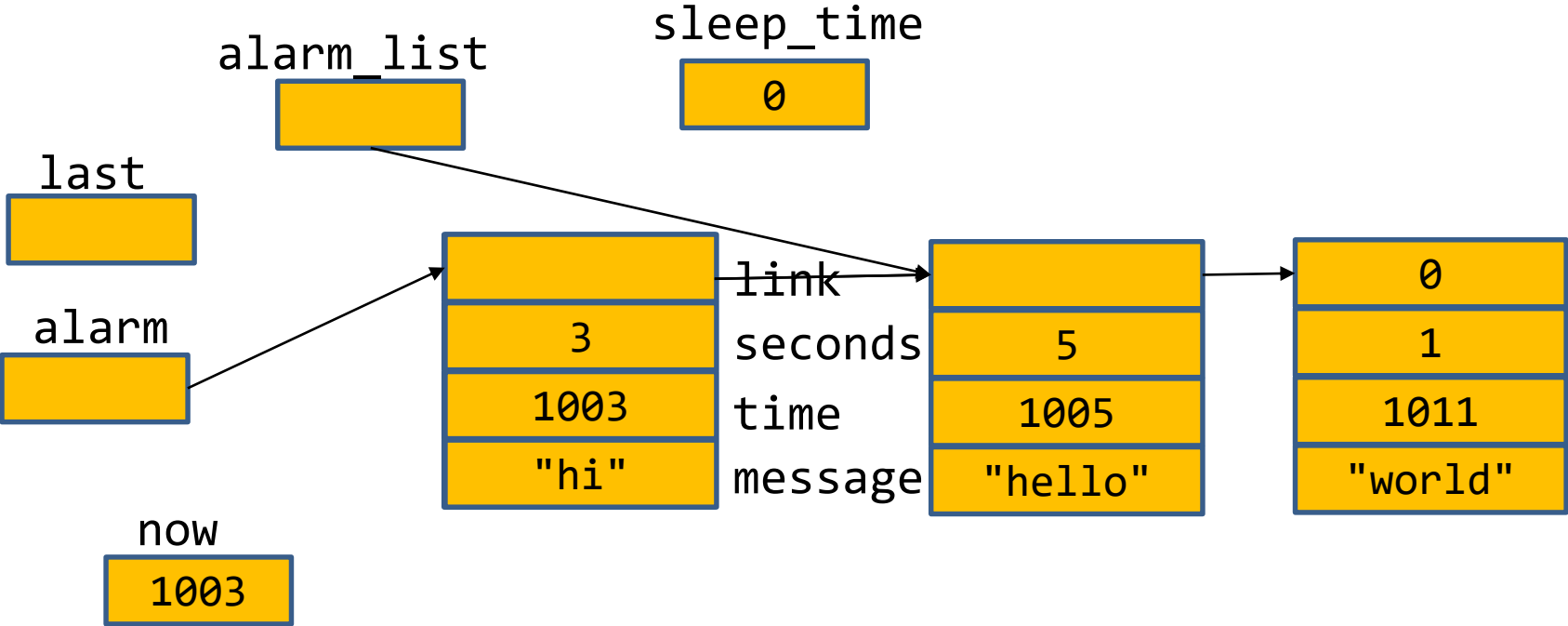
확인 명령 ( 다른 터미널 에서 )

```
# ps -eLf | grep a.out | grep -v grep
```

alarm\_mutex.c 분석

time(0) : 현재시간







# 4. Pthread Programming

---

4.1 Pthread 개요

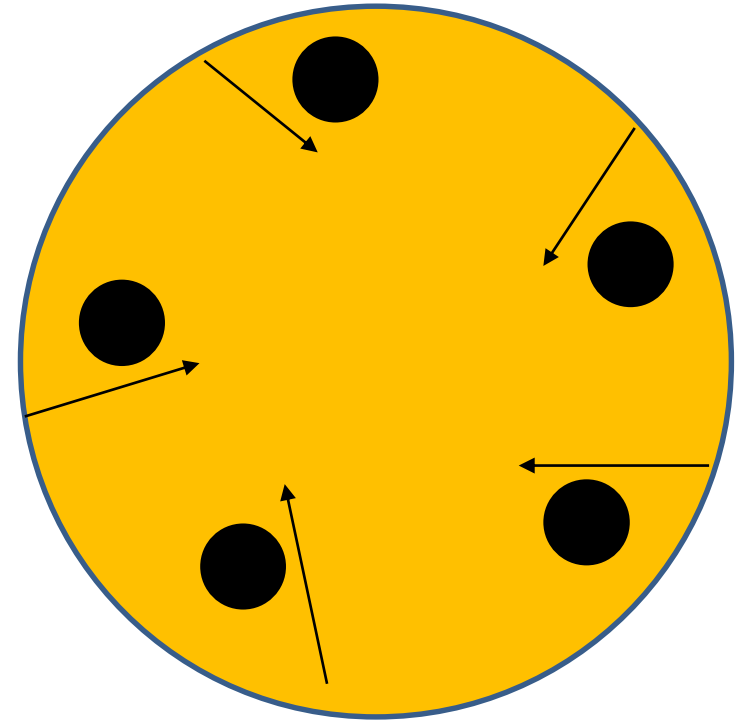
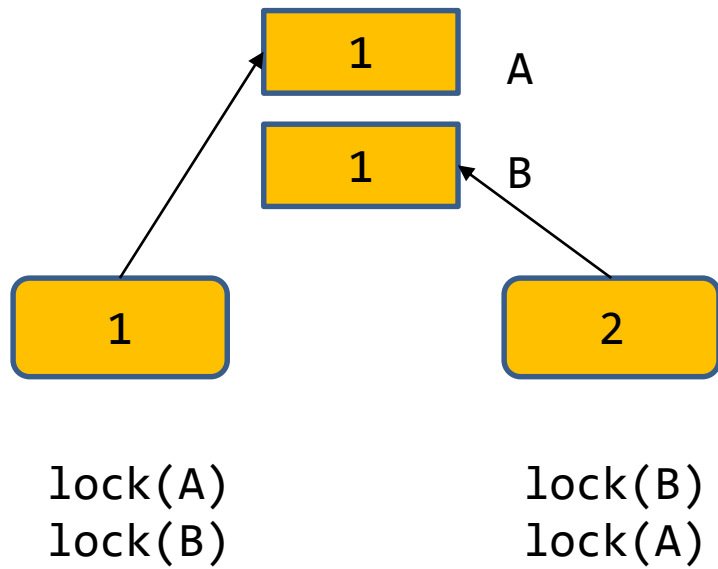
4.2 Pthread API

4.3 Thread 동기화

4.4 Thread 내부구조 분석

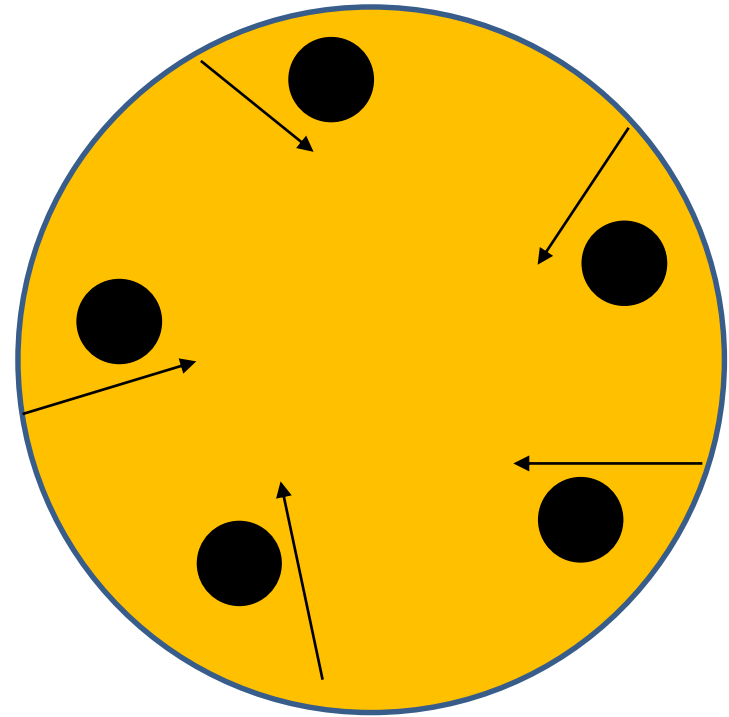
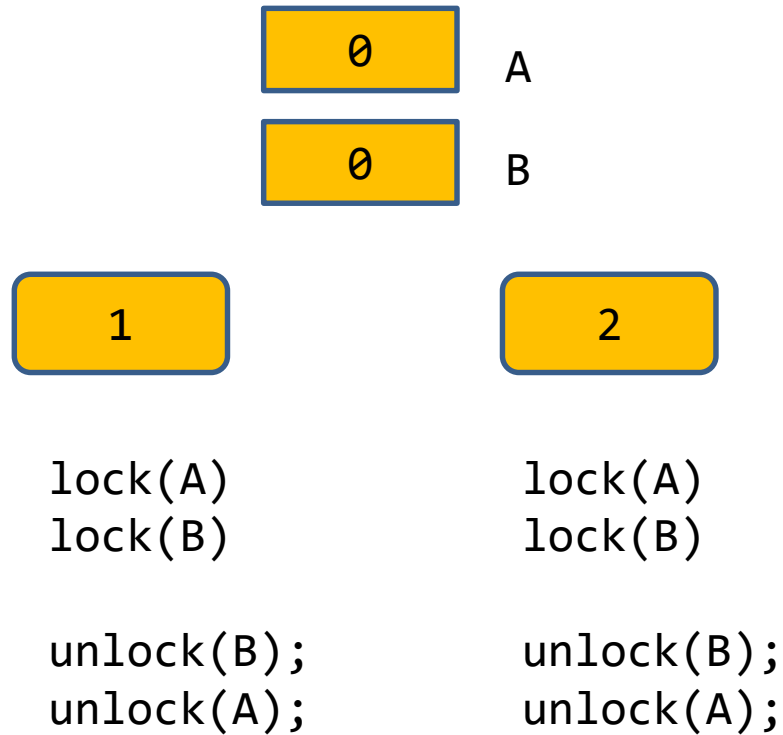
---

Dead lock : 생각 하는 철학자 비유



Dead lock : 해결 방법

1. 모든 스레드에서 lock의 순서를 지키면 됨



```
#include <pthread.h>
#include "errors.h"

#define MAX_THREAD 4
int sum = 0;
void *thread_routine( void * arg )
{
    int local,i;
    for(i=0; i<1000000000/MAX_THREAD; i++ )
    {
        local = sum;
        local = local+1;
        sum = local;
    }
    return arg;
}
```

test\_1.c

test\_1.c

```
int main( int argc, char **argv )
{
    pthread_t thread_id[MAX_THREAD];
    void *thread_result;
    int status, i;

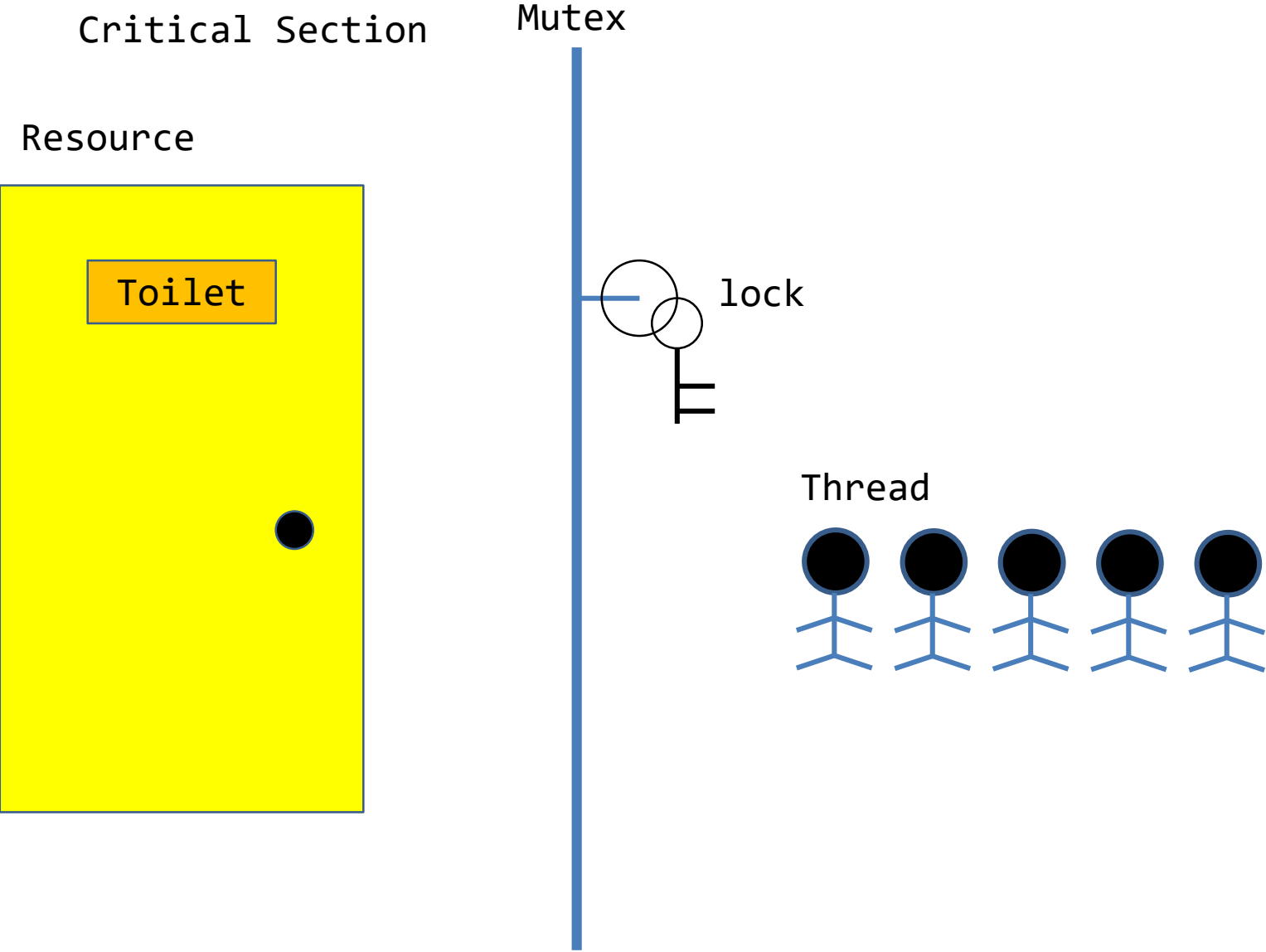
    for(i=0; i<MAX_THREAD ; i++ )
        status = pthread_create( &thread_id[i],0,thread_routine,0);

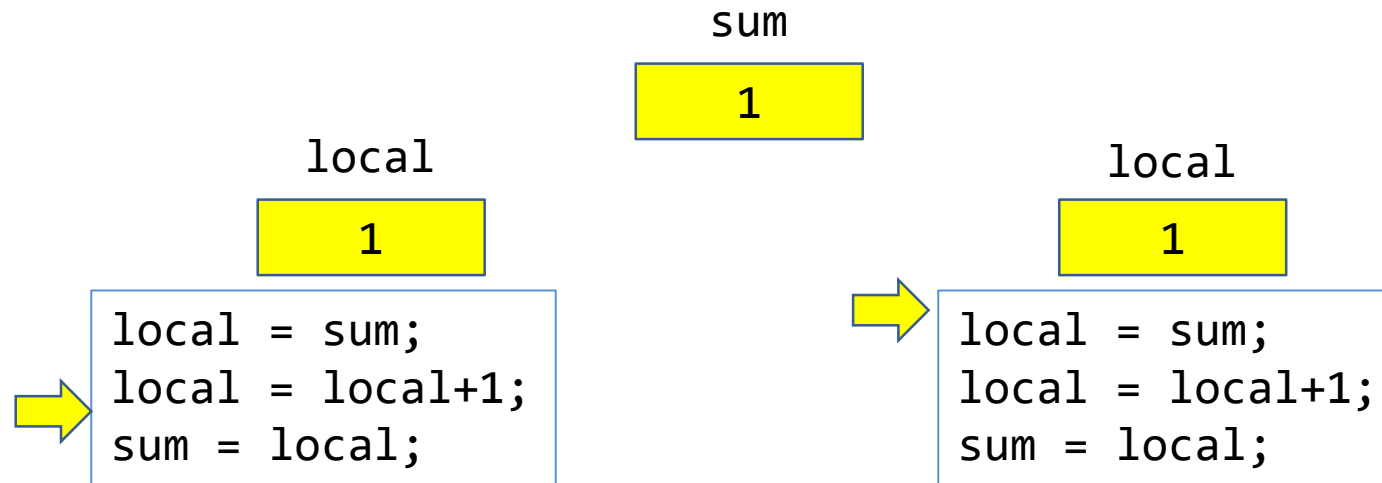
    for(i=0; i<MAX_THREAD ; i++ )
        status = pthread_join( thread_id[i], &thread_result );

    DPRINTF(("sum=%d\n", sum));
    return 0;  // exit(0);
}
```

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
void *thread_routine( void * arg )
{
    int local,i;
    for(i=0; i<1000000000/MAX_THREAD; i++ )
    {
        pthread_mutex_lock(&lock);
        local = sum;
        local = local+1;
        sum = local;
        pthread_mutex_unlock(&lock);
    }
    return arg;
}
```

test\_2.c



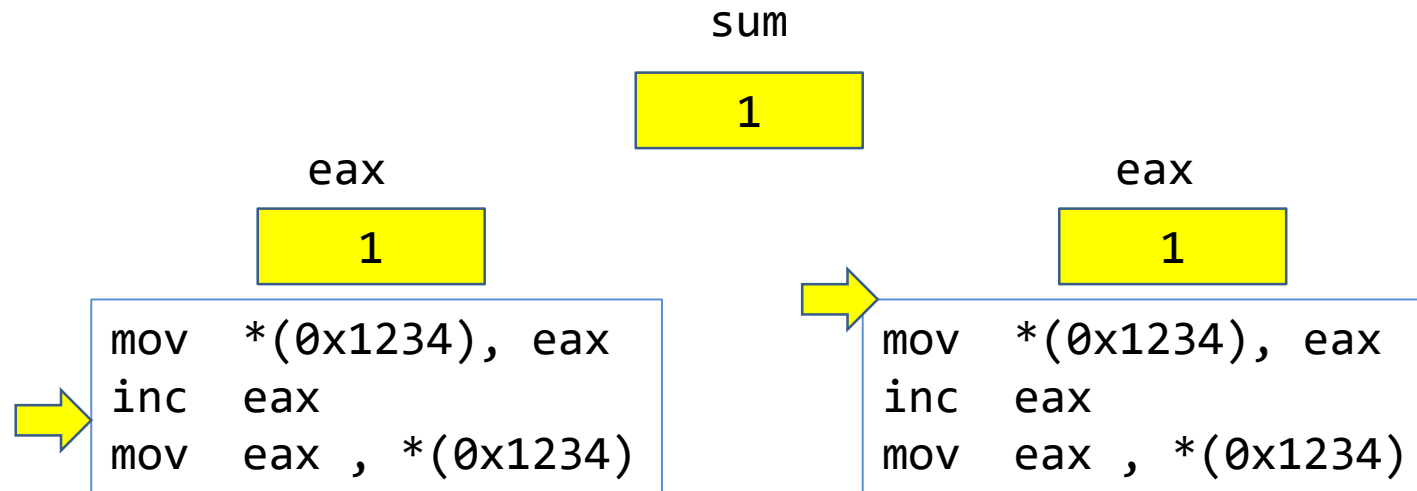




```
#include <pthread.h>
#include "errors.h"

#define MAX_THREAD 2
int sum = 0;
void *thread_routine( void * arg )
{
    int local,i;
    for(i=0; i<1000000000/MAX_THREAD; i++ )
    {
        ++sum;
    }
    return arg;
}
```

test\_3.c

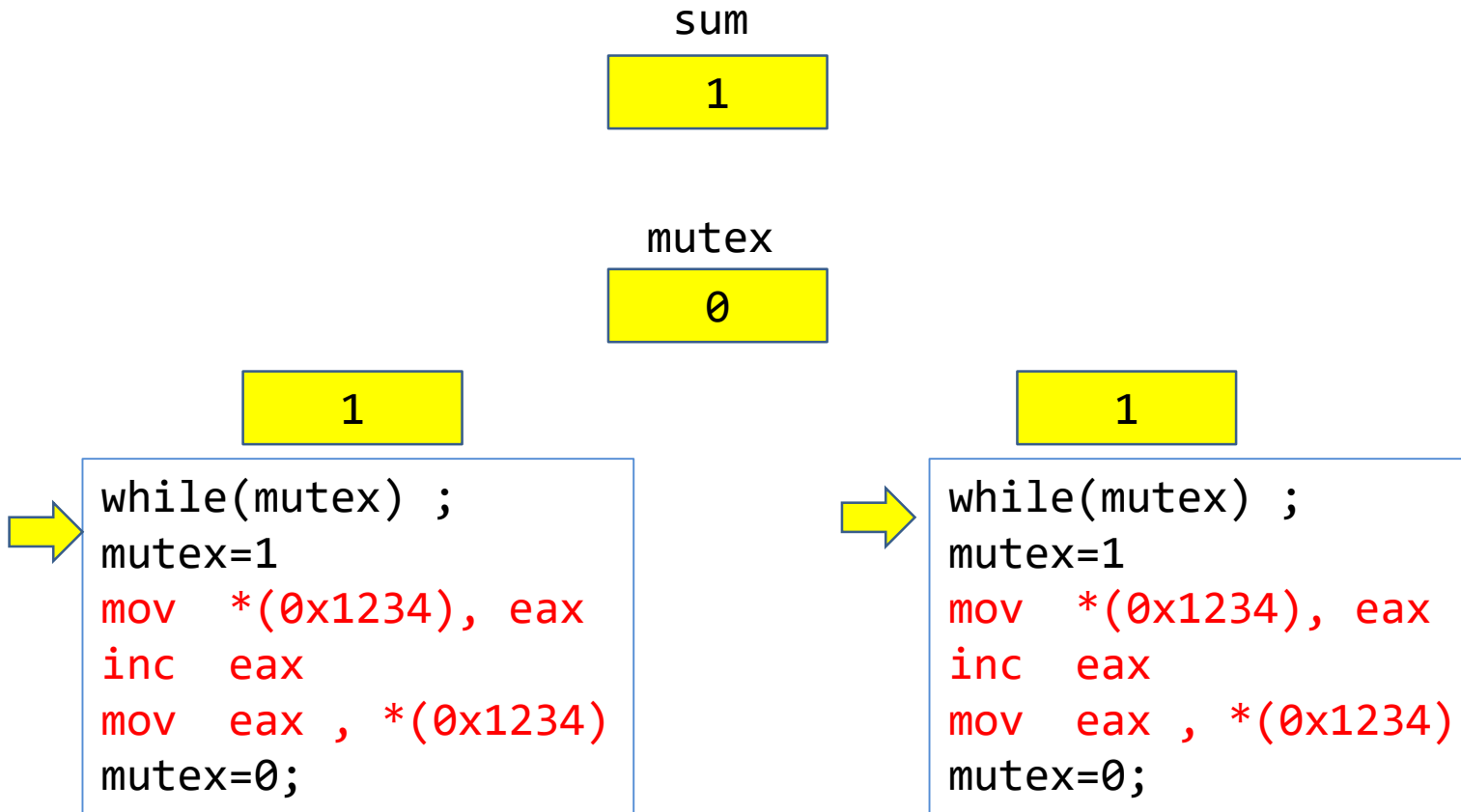


```
#include <pthread.h>
#include "errors.h"

#define MAX_THREAD 2

int sum = 0;
int mutex=0;
void *thread_routine( void * arg )
{
    int local,i;
    for(i=0; i<1000000000/MAX_THREAD; i++ )
    {
        while(mutex);
        mutex=1;
        ++sum;
        mutex=0;
    }
    return arg;
}
```

test\_4.c

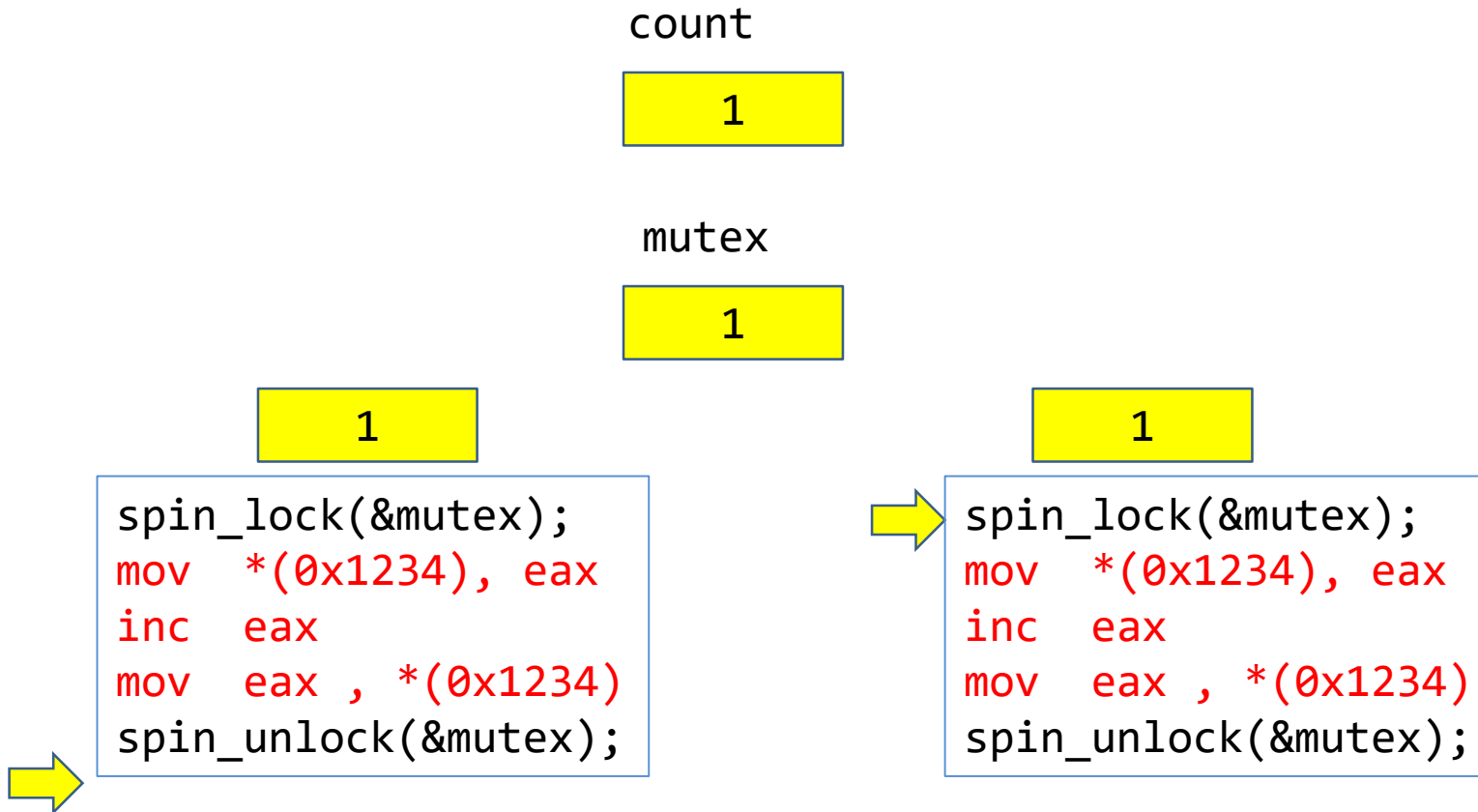


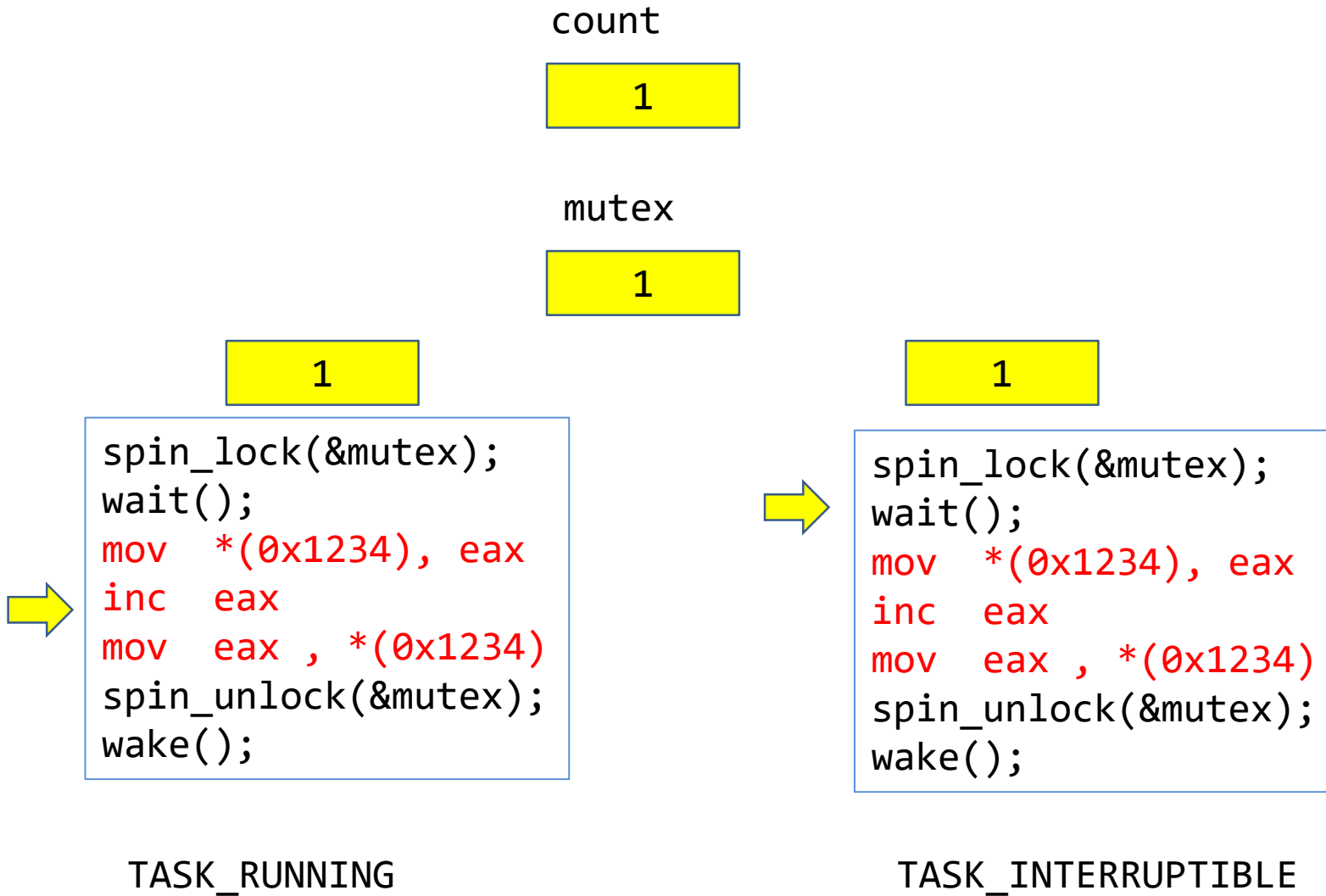
```
typedef int int32_t;
int __bionic_cmpxchg(int32_t old_value, int32_t new_value,
                    volatile int32_t* ptr)
{
    int32_t prev;
    __asm__ __volatile__ ("lock; cmpxchgl %1, %2"
                          : "=a" (prev)
                          : "q" (new_value), "m" (*ptr), "0" (old_value)
                          : "memory");
    return prev != old_value;
}
```

```
typedef int int32_t;
int __bionic_cmpxchg(int32_t old_value, int32_t new_value, volatile int32_t* ptr)
{
    int32_t prev;
    __asm__ __volatile__ ("lock; cmpxchgl %1, %2"
        : "=a" (prev)
        : "q" (new_value), "m" (*ptr), "0" (old_value)
        : "memory");
    return prev != old_value;
}

int mutex=0;

void spin_lock(int *mutex)
{
    while( __bionic_cmpxchg(0, 1, mutex) );
}
```







```
while (atomic_exchange_explicit(&mutex->state, locked_contended,  
                                memory_order_acquire) != unlocked) {  
    if (__futex_wait_ex(&mutex->state, shared, locked_contended,  
                        use_realtime_clock, abs_timeout_or_null) == -ETIMEDOUT) {  
        return ETIMEDOUT;  
    }  
}
```

futex.c

```
#include <pthread.h>
#include <sys/syscall.h>

int mutex = 1;

void * foo(void *data)
{
    syscall(202, &mutex, 0, 1, 0); // __futex_wait();
    printf("after\n");
    return 0;
}

int main()
{
    pthread_t thread;
    pthread_create( &thread, 0, foo, 0 );
    getchar();
    syscall(202, &mutex, 1, 1); // __futex_wake();
    pthread_join( thread, 0 );
    return 0;
}
```

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

pthread_mutex_t mutex;

int main()
{
    pthread_mutex_init( &mutex , 0 );

    pthread_mutex_lock( &mutex );
    pthread_mutex_lock( &mutex );
    printf("임계영역 진입\n");
    pthread_mutex_unlock( &mutex );
    return 0;
}
```

```
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t mutex;
void foo(int i)
{
    if(i==0)
        return;
    pthread_mutex_lock( &mutex );
    printf("임계영역 진입\n");
    foo(i-1);
    pthread_mutex_unlock( &mutex );
}

int main()
{
    pthread_mutex_init( &mutex , 0 );
    foo(10);

    return 0;
}
```

```
#include <pthread.h>

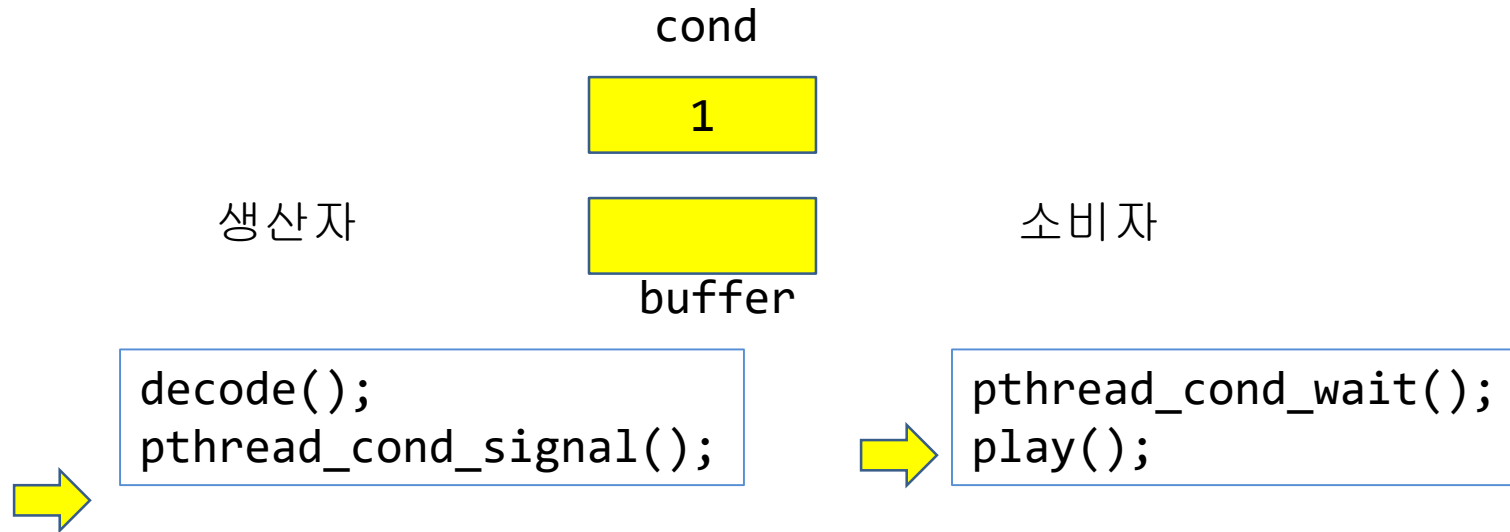
pthread_mutex_t mutex;
pthread_mutexattr_t attr;

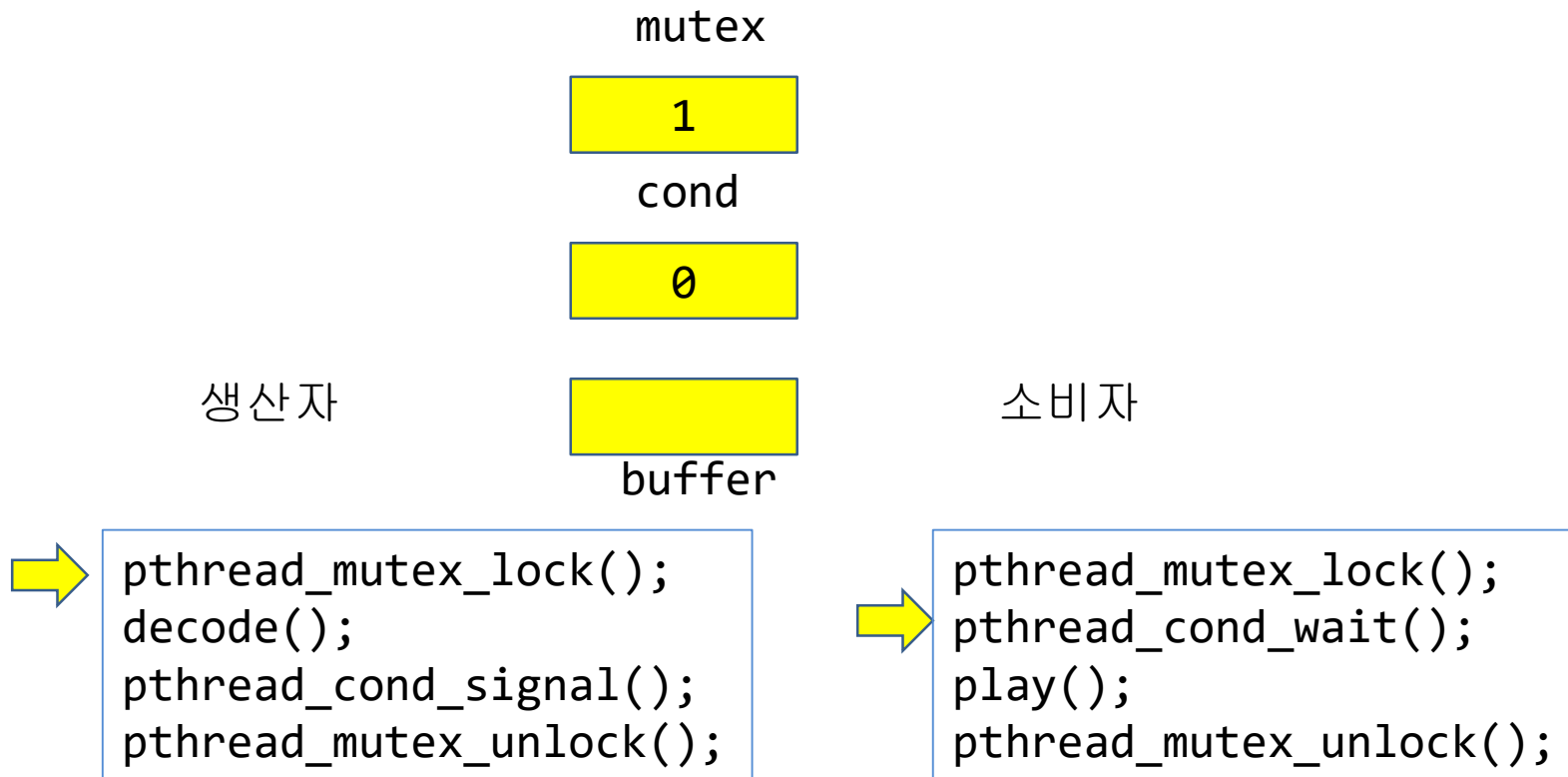
int main()
{
    pthread_mutexattr_init( &attr );
    pthread_mutexattr_settype( &attr , PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init( &mutex , &attr );

    pthread_mutex_lock( &mutex ); // 1
    pthread_mutex_lock( &mutex ); // 2
    printf("임계영역 진입\n");
    pthread_mutex_unlock( &mutex ); // 1
    pthread_mutex_unlock( &mutex ); // 0
    return 0;
}
```

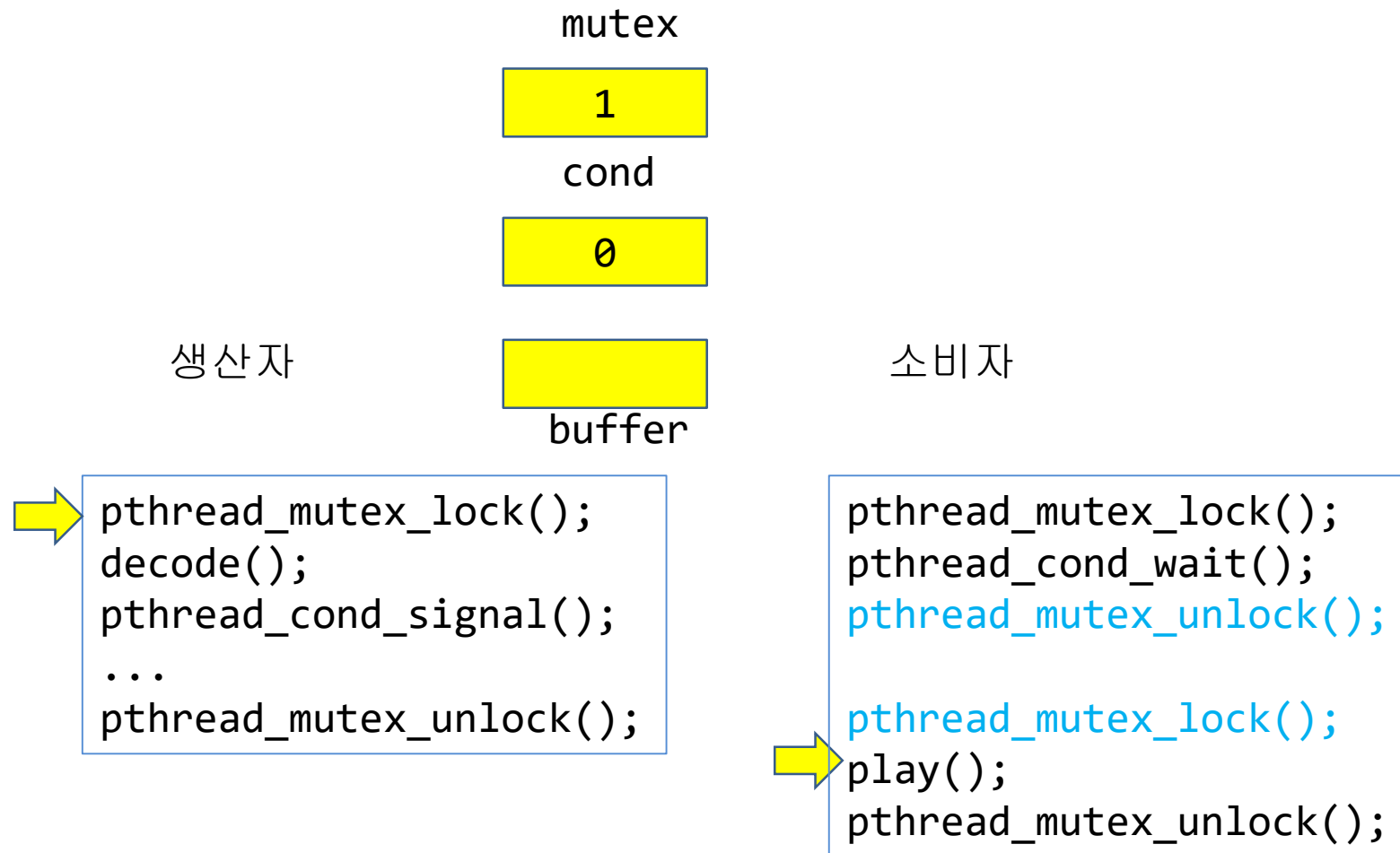
```
pthread_mutex_t mutex;
pthread_mutexattr_t attr;
void foo(int i)
{
    if(i==0)
        return;
    pthread_mutex_lock( &mutex );
    printf("임계영역 진입\n");
    foo(i-1);
    pthread_mutex_unlock( &mutex );
}
int main()
{
    pthread_mutexattr_init( &attr );
    pthread_mutexattr_settype( &attr , PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init( &mutex , &attr );
    foo(10);

    return 0;
}
```









mutex

1


cond

1

생산자


소비자

buffer



```
pthread_mutex_lock(&mutex);  
decode();  
pthread_cond_signal(&cond);  
...  
pthread_mutex_unlock(&mutex);
```

```
pthread_mutex_lock(&mutex);  
pthread_cond_wait(&cond, &mutex);  
pthread_mutex_unlock(&mutex);  
  
pthread_mutex_lock(&mutex);  
play();  
pthread_mutex_unlock(&mutex);
```



# 4. Pthread Programming

---

4.1 Pthread 개요

4.2 Pthread API

4.3 Thread 동기화

**4.4 Thread 재진입 가능함수 구현**

---

# vi strtok.c

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>

void * my_handler1(void* data ){
    char hp[] = "010-1234-5678";
    char *p;
    p = strtok( hp, "-");
    while(p)    {
        printf("[%s]\n", p );
        p = strtok( 0, "-");
    }
}

void * my_handler2(void* data ){
    char ip[] = "192.168.100.128";
    char *p;
    p = strtok( ip, ".");
```

```
# vi strtok.c
```

```
    while(p)
    {
        printf("[%s]\n", p );
        p = strtok( 0, ".");
    }

int main()
{
    pthread_t tid1, tid2;

    pthread_create(&tid1, 0, my_handler1, 0);
    pthread_create(&tid2, 0, my_handler2, 0);
    pthread_join( tid1, 0 );
    pthread_join( tid2, 0 );
    return 0;
}
```

```
# vi strtok_r.c
```

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>

void * my_handler1(void* data ){
    char hp[] = "010-1234-5678";
    char *p;
    char *saveptr;
    p = strtok_r( hp, "-", &saveptr );
    while(p) {
        printf("[%s]\n", p );
        p = strtok_r( 0, "-" , &saveptr);
    }
}

void * my_handler2(void* data ){
    char ip[] = "192.168.100.128";
    char *p;
    char *saveptr;
    p = strtok_r( ip, "." , &saveptr );
```

```
# vi strtok_r.c
```

```
    while(p)
    {
        printf("[%s]\n", p );
        p = strtok_r( 0, "." , &saveptr);
    }
}

int main(){
    pthread_t tid1, tid2;

    pthread_create(&tid1, 0, my_handler1, 0);
    pthread_create(&tid2, 0, my_handler2, 0);
    pthread_join( tid1, 0 );
    pthread_join( tid2, 0 );
    return 0;
}
```

# vi tls.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
pthread_key_t key;
void foo(void){
    int *tsd;
    tsd = pthread_getspecific( key );
    if( tsd == 0 )    {
        tsd = calloc( 1, sizeof(int) );
        pthread_setspecific( key, tsd );
    }
    ++*tsd;
    printf("tsd=%d\n", *tsd );
}
void * my_handler1(void* data ){
    foo();
    foo();
    foo();
}
```



```
# vi reent.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
pthread_key_t key;
void foo(int *count)
{
    printf("%d\n", ++*count);
}

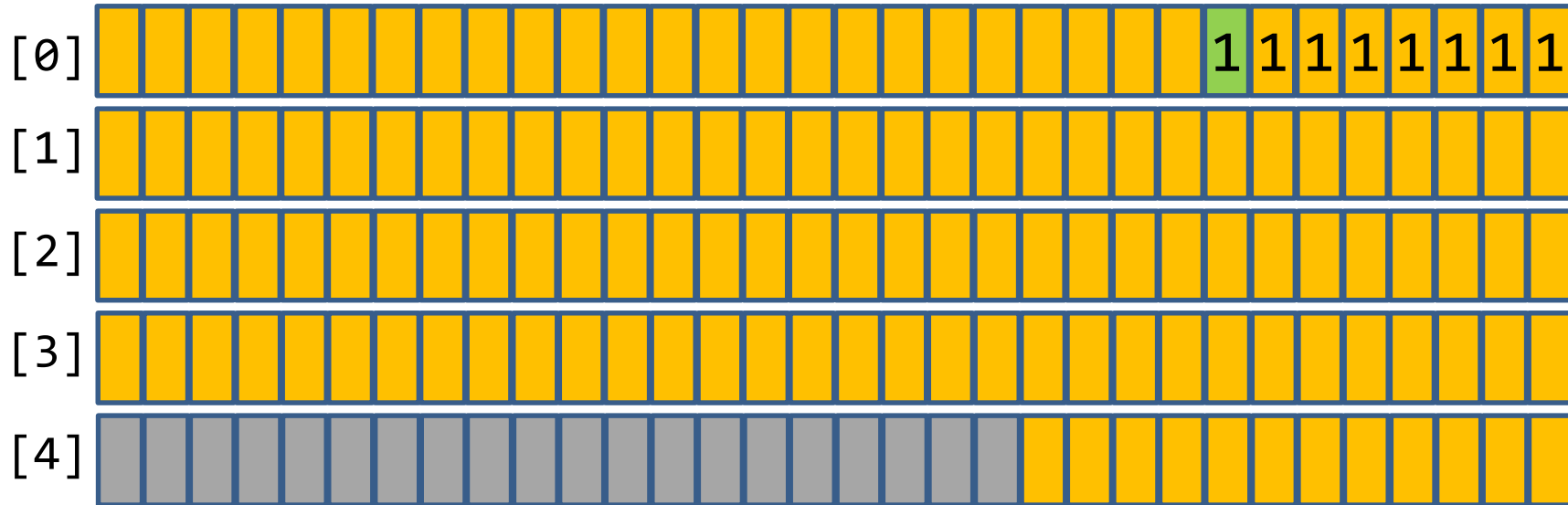
void * my_handler1(void* data )
{
    int count;
    foo(&count);
    foo(&count);
    foo(&count);
}
```

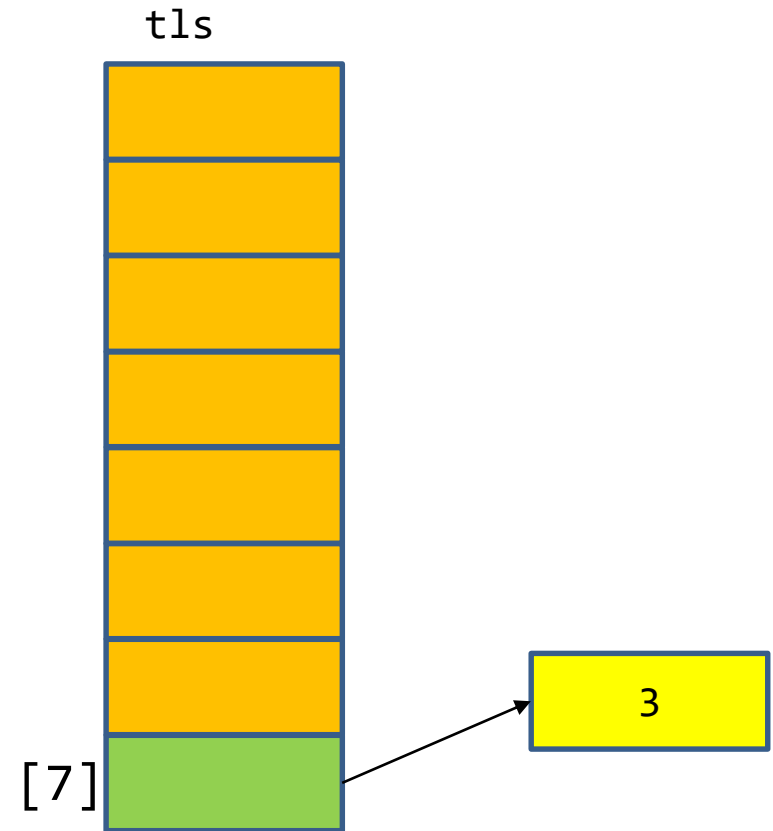
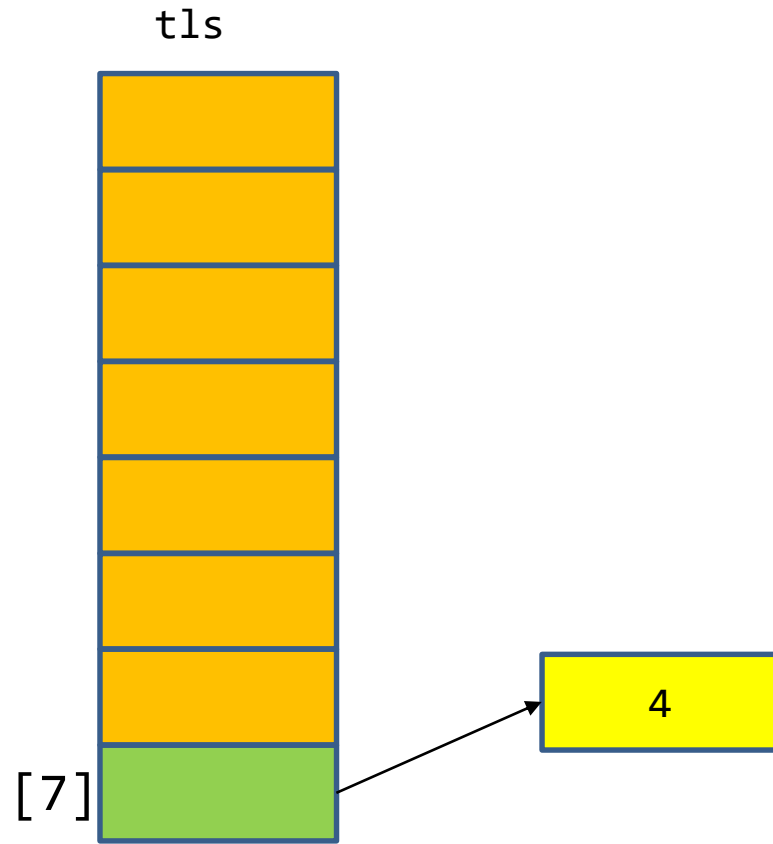
```
# vi tls.c
```

```
void * my_handler2(void* data ){
    foo();
    foo();
    foo();
    foo();
}
void destr_function (void *p){
    free(p);
}
int main(){
    pthread_t tid1, tid2;
    pthread_key_create( &key, destr_function );
    pthread_create(&tid1, 0, my_handler1, 0);
    pthread_create(&tid2, 0, my_handler2, 0);
    pthread_join( tid1, 0 );
    pthread_join( tid2, 0 );
    pthread_key_delete(key);
    return 0;
}
```

```
pthread_key_t key;  
pthread_key_create(&key, fn );
```

7  
key





### readline\_1.c

```
ssize_t readline(int fd, void *vptr, size_t maxlen){
    int      n, rc;
    char      c, *ptr;
    ptr = vptr;
    for (n = 1; n < maxlen; n++) {
        if ( (rc = read(fd, &c, 1)) == 1) {
            *ptr++ = c;
            if (c == '\n')
                break; /* newline is stored, like fgets() */
        } else if (rc == 0) {
            if (n == 1)
                return(0); /* EOF, no data read */
            else
                break; /* EOF, some data was read */
        } else
            return(-1); /* error, errno set by read() */
    }
    *ptr = 0; /* null terminate like fgets() */
    return(n);
}
```

comman를 처리하는 데몬은 \n가 flush 시점이 된다.

putty

sshd

# ls

l	s	\
		n

```
readline( fd, buff, sizeof buff );  
system( buff );
```

readline\_2.c

```
#define MAXLINE 4096

static ssize_t my_read(int fd, char *ptr)
{
    static int    read_cnt = 0;
    static char   *read_ptr;
    static char   read_buf[MAXLINE];

    if (read_cnt <= 0) {
        if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
            return(-1);
        } else if (read_cnt == 0)
            return(0);
        read_ptr = read_buf;
    }

    read_cnt--;
    *ptr = *read_ptr++;
    return(1);
}
```

readline\_3.c

```
typedef struct
{
    int            read_cnt;
    char          *read_ptr;
    char          read_buf[MAXLINE];
} Rline;

static ssize_t my_read(int fd, char *ptr, Rline *buff)
{
    if (buff->read_cnt <= 0) {
        if ( (buff->read_cnt = read(fd, buff->read_buf,
                                   sizeof(buff->read_buf))) < 0) {
            return(-1);
        } else if (buff->read_cnt == 0)
            return(0);
        buff->read_ptr = buff->read_buf;
    }

    buff->read_cnt--;
    *ptr = *buff->read_ptr++;
    return(1);
}
```



getline\_3.c

```
ssize_t  getline(int fd, void *vptr, size_t maxlen)
{
    int      n, rc;
    char     c, *ptr;
    Rline    buff = {0,};
    ptr = vptr;
    for (n = 1; n < maxlen; n++) {
        if ( (rc = my_read(fd, &c, &buff)) == 1) {
            *ptr++ = c;
            if (c == '\n')
                break;
        } else if (rc == 0) {
            if (n == 1)
                return(0);
            else
                break;
        } else
            return(-1);
    }
    *ptr = 0;
    return(n);
}
```

readline\_4.c

```
static pthread_key_t    rl_key;
static pthread_once_t  rl_once = PTHREAD_ONCE_INIT;

static void readline_destructor(void *ptr)
{
    free(ptr);
}

static void readline_once(void)
{
    pthread_key_create(&rl_key, readline_destructor);
}

typedef struct {
    int      rl_cnt;                /* initialize to 0 */
    char     *rl_bufptr;            /* initialize to rl_buf */
    char     rl_buf[MAXLINE];
} Rline;
```

readline\_4.c

```
static ssize_t my_read(Rline *tsd, int fd, char *ptr)
{
    if (tsd->rl_cnt <= 0) {
        if ( (tsd->rl_cnt = read(fd, tsd->rl_buf, MAXLINE)) < 0) {
            return(-1);
        } else if (tsd->rl_cnt == 0)
            return(0);
        tsd->rl_bufptr = tsd->rl_buf;
    }

    tsd->rl_cnt--;
    *ptr = *tsd->rl_bufptr++;
    return(1);
}
```

readline\_4.c

```
ssize_t readline(int fd, void *vptr, size_t maxlen)
{
    int      n, rc;
    char      c, *ptr;
    Rline     *tsd;

    pthread_once(&rl_once, readline_once);
    if ( (tsd = pthread_getspecific(rl_key)) == NULL) {
        tsd = calloc(1, sizeof(Rline));          /* init to 0 */
        pthread_setspecific(rl_key, tsd);
    }

    ...
    return(n);
}
```

# 5. IPC Programming

# 5. IPC Programming

---

## **5.1 Signal**

5.2 Pipe

5.3 Message Queue

5.4 Semaphores

5.5 Shared Memory

---

### ◆ Signal?

- 프로세스간 동기화에 이용
- 한 프로세스가 다른 프로세스에게 보냄
- 커널이 프로세스에게 보냄
- 소프트웨어 인터럽트

### ◆ Signal의 발생

- 인위적 발생 : kill()
- 사건발생 : 알람, 프로세스 종료 등
- 에러상황 : 잘못된 메모리 접근
- 외부상황 : 키보드 입력(Cntl + C)

### ◆ Signal의 처리

- 무시, 보류, 종료, 특정함수 호출

### ◆ signal 종류

```
#define SIGHUP 1 /* hangup */
#define SIGINT 2 /* interrupt (rubout) */
#define SIGQUIT 3 /* quit (ASCII FS) */
#define SIGILL 4 /* illegal instruction (not reset when caught) */
#define SIGTRAP 5 /* trace trap (not reset when caught) */
#define SIGIOT 6 /* IOT instruction */
#define SIGABRT 6 /* used by abort, replace SIGIOT in the future */
#define SIGEMT 7 /* EMT instruction */
#define SIGFPE 8 /* floating point exception */
#define SIGKILL 9 /* kill (cannot be caught or ignored) */
#define SIGBUS 10 /* bus error */
#define SIGSEGV 11 /* segmentation violation */
#define SIGSYS 12 /* bad argument to system call */
#define SIGPIPE 13 /* write on a pipe with no one to read it */
#define SIGALRM 14 /* alarm clock */
#define SIGTERM 15 /* software termination signal from kill */
#define SIGUSR1 16 /* user defined signal 1 */
#define SIGUSR2 17 /* user defined signal 2 */
```



```
#define SIGCLD  18      /* child status change */
#define SIGPWR  19      /* power-fail restart */
#define SIGWINCH 20     /* window size change */
#define SIGURG  21      /* urgent socket condition */
#define SIGPOLL 22      /* pollable event occurred */
#define SIGIO   SIGPOLL /* socket I/O possible (SIGPOLL alias) */
#define SIGSTOP 23      /* stop (cannot be caught or ignored) */
#define SIGTSTP 24      /* user stop requested from tty */
#define SIGCONT 25      /* stopped process has been continued */
#define SIGTTIN 26      /* background tty read attempted */
#define SIGTTOU 27      /* background tty write attempted */
#define SIGVTALRM 28     /* virtual timer expired */
#define SIGPROF 29      /* profiling timer expired */
#define SIGXCPU 30      /* exceeded cpu limit */
#define SIGXFSZ 31      /* exceeded file size limit */
#define SIGWAITING 32    /* process's lwps are blocked */
#define SIGLWP  33      /* special signal used by thread library */
#define SIGFREEZE 34     /* special signal used by CPR */
#define SIGTHAW 35      /* special signal used by CPR */
#define SIGCANCEL 36     /* thread cancellation signal used by libthread */
#define SIGLOST 37      /* resource lost (eg, record-lock lost) */
```

### ◆ 대표적 Signal

- SIGABRT : abort 함수의 호출에 의해 발생. 프로세스는 비정상적으로 종료
- SIGALRM : alarm 함수에 의해 설정된 타이머에 의해 발생
- SIGCHLD : 프로세스가 종료하거나 정지한 경우에 부모 프로세스에게 전달
- SIGFPE : 산술 연산 에러에 의해 발생 (0으로 나눈 경우 등)
- SIGHUP : 터미널 연결이 단절되는 경우에 제어 프로세스에게 전달
- SIGINT : 인터럽트 키(DELETE, ^C)를 누르면 터미널 드라이버에서 발생  
전위 프로세스 그룹의 모든 프로세스에게 전달
- SIGKILL : 무시하거나 임의의 처리(catch)를 할 수 없는 시그널.

### ◆ 대표적 Signal

- SIGQUIT : quit 키(^W)를 누르면 터미널 드라이버에서 발생  
종료된 프로세스는 core 파일을 생성
- SIGSEGV : 잘못된 메모리 참조에 의해 발생
- SIGSTOP : 프로세스를 정지시키는 작업 제어 시그널  
무시하거나 임의의 처리(catch)를 할 수 없는 시그널
- SIGTERM : 작업 종료 시그널
- SIGUSR1 : 응용 프로그램에서 사용자가 정의하여 사용할 수 있는 시그널
- SIGUSR2 : 응용 프로그램에서 사용자가 정의하여 사용할 수 있는 시그널

### ◆ 시스널 무시(ignore)

- SIGKILL과 SIGSTOP 시그널을 제외한 모든 시그널을 무시할 수 있다.
- 하드웨어 오류에 의해 발생한 시그널에 대해서는 주의해야 한다.

### ◆ 시스널 처리(catch)

- 시그널이 발생하면 미리 등록된 함수(handler)가 수행된다.
- SIGKILL과 SIGSTOP 시그널에는 처리할 함수를 등록할 수 없다

### ◆ 기본 처리(default)

- 특별한 처리 방법을 선택하지 않은 경우
- 대부분 시그널의 기본 처리 방법은 프로세스를 종료시키는 것이다

### ◆ Signal 무시

- `signal(SIGINT, SIG_IGN);`  
    `<==` 인터럽트 키(^C)를 무시함

### ◆ Signal 복구

- `signal(SIGINT, SIG_DFL);`
- 여러개의 시그널을 무시할 수도 있음.  
    `signal(SIGINT, SIG_IGN);`  
    `signal(SIGQUIT, SIG_IGN);`

인터럽트로부터 보호해야 하는 중요한  
작업 수행시 의도적으로 보호

### ◆ sig 인자

- 시그널 번호
- null signal (0) : 실제로 시그널을 보내지 않고 프로세스의 존재여부 파악
- 프로세스 미존재시 -1 리턴(errno= ESRCH)

### ◆ pid 인자

- pid > 0 : 프로세스 ID가 pid인 프로세스에게 시그널을 전달한다.
- pid == 0 : 호출한 프로세스와 같은 프로세스 그룹 ID를 가지고 있는 모든 프로세스에게 시그널을 전달한다.
- pid < 0 : pid의 절대값에 해당하는 프로세스 그룹 ID를 가지고 있는 모든 프로세스에게 시그널을 전달한다.

signal\_1.c

```
#include <signal.h>
int catchint(int signo)
{
    printf(" SIGINT Receive\n");
}

main()
{
    signal(SIGINT,(void *) catchint);

    printf("sleep call #1\n");    sleep(1);
    printf("sleep call #2\n");    sleep(1);
    printf("sleep call #3\n");    sleep(1);
    printf("sleep call #4\n");    sleep(1);
    printf("Exiting");
    exit(0);
}
```



signal\_2.c

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

void handler(int sig){
    printf("signal no(%d) Received\n",sig);
}

main(){
    if(signal(SIGUSR1,handler)==SIG_ERR) {
        fprintf(stderr,"cannot set USR1\n");
        exit(1);
    }
    if(signal(SIGUSR2,handler)==SIG_ERR) {
        fprintf(stderr,"cannot set USR2\n");
        exit(1);
    }
    for(;;)        pause();
}
```



% kill -USR1 PID

% kill -USR2 PID



### sig\_parent.c

```
#include <signal.h>
#define NUMCHILD 3
main(int argc, char *argv[]){
    int pid, chpid[NUMCHILD];
    int i, status;

    for(i=0;i<NUMCHILD;i++) {
        if((pid=fork())==0)
            execlp("./sig_child","./sig_child",
                (char *)0);
        chpid[i] = pid;
    }
    printf("sig_parent : %d
        child process run\n",NUMCHILD);
    sleep(10);
    for(i=0;i<NUMCHILD;i++)
        kill(chpid[i],SIGINT);
}
```

### sig\_child.c

```
#include <signal.h>
void sig(int sig) {
    printf("child die(%d)\n",getpid());
}

main() {
    signal(SIGINT,sig);
    pause();
}
```



인위적 발생

### ◆ Description

- 지정된 시간 후에 SIGALRM 시그널이 발생하도록 타이머를 설정
- SIGALRM의 기본 처리 방법은 프로세스의 종료
- 일반적으로 시그널 처리 함수를 등록하여 사용한다.
- 한 프로세스에는 하나의 알람 시계만 존재
- 이미 알람이 설정된 상태에서 다시 alarm 함수를 호출하면  
이전 알람의 남은 시간이 리턴되고 새로운 알람으로 설정된다.
- seconds 인자가 0인 경우,  
이미 설정된 알람이 존재하면 남은 시간이 리턴되고 알람은 해제된다.

alarm\_1.c

```
#include <signal.h>
#include <unistd.h>
int alm(signo)
int signo;
{
    printf("SIGALRM Receive");
}

void main()
{
    signal(SIGALRM, alm);
    alarm(10);
    printf("process id pause\n");
    pause();
    printf("process is wakeup\n");
}
```

### ◆ Description

- sigemptyset : set의 모든 시스널을 0으로 set(모든 시스널 제외)
- sigfillset : set의 모든 시스널 1로 set(모든 시스널 포함)
- sigaddset : set의 멤버로서 signo로 지정된 시그널 추가
- sigdelset : set에서 signo로 지정된 시그널 제거
- sigismember : signo 시그널이 set의 멤버인지를 검사

### ◆ Description

- signalmask를 변경하거나 검사한다.
- signalmask : 프로세스에게 전달되지 않도록 블럭된 시그널의 집합
- how - 시그널 set을 변경시키는 방법
  - SIG\_BLOCK : set 인자로 지정된 시그널들을 시그널 마스크에 추가
  - SIG\_UNBLOCK : set 인자로 지정된 시그널들을 시그널 마스크에서 제외
  - SIG\_SETMASK : set 인자로 시그널 마스크를 대체
- set - 변경될 시그널 마스크
- oset - sigprocmask 함수 호출 이전의 시그널 마스크 내용
- oset이 NULL이 아니면 이전의 블럭된 시그널 세트값이 저장된다.
- set이 NULL이면 how는 의미가 없으며 에러체크로 사용된다.

sigprocmask\_1.c

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

void show_mask()
{
    sigset_t sigset;

    if (sigprocmask(0, NULL, &sigset) < 0)
        printf("sigprocmask error");

    if (sigismember(&sigset, SIGINT))    printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))   printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))    printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))    printf("SIGALRM ");

    printf("\n");
}
```

sigprocmask\_1.c

```
int main(void)
{
    sigset_t newmask, oldmask;

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);

    /* add SIGQUIT signal to blocked signal list */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        printf("sigprocmask error");

    show_mask();

    /* restore previous signal mask */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        printf("sigprocmask error");

    exit (0);
}
```

## ◆ Description

- 호출한 프로세스에 대해 발생한 후 블록되어 있는 시그널 집합을 리턴한다.
- 블록된 시그널 집합은 시그널 세트로 표현된다.
- 중복 발생한 시그널은 누적되지 않는다.



sigpending\_1.c

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

static void sig_quit(int);

int main(void){
    sigset_t newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        printf("can't catch SIGQUIT");

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);

    /* block SIGQUIT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        printf("SIG_BLOCK error");

    sleep(5);      /* SIGQUIT here will remain pending */
}
```

sigpending\_1.c

```
    if (sigpending(&pendmask) < 0)
        printf("sigpending error");
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nSIGQUIT pending\n");

    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        printf("SIG_SETMASK error");
    printf("SIGQUIT unblocked\n");
    sleep(5);      /* SIGQUIT here will terminate with core file */
    exit(0);
}

static void sig_quit(int signo){
    printf("caught SIGQUIT\n");

    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        printf("can't reset SIGQUIT");
    return;
}
```

# 5. IPC Programming

---

5.1 Signal

**5.2 Pipe**

5.3 Message Queue

5.4 Semaphores

5.5 Shared Memory

---

### ◆ Pipe?

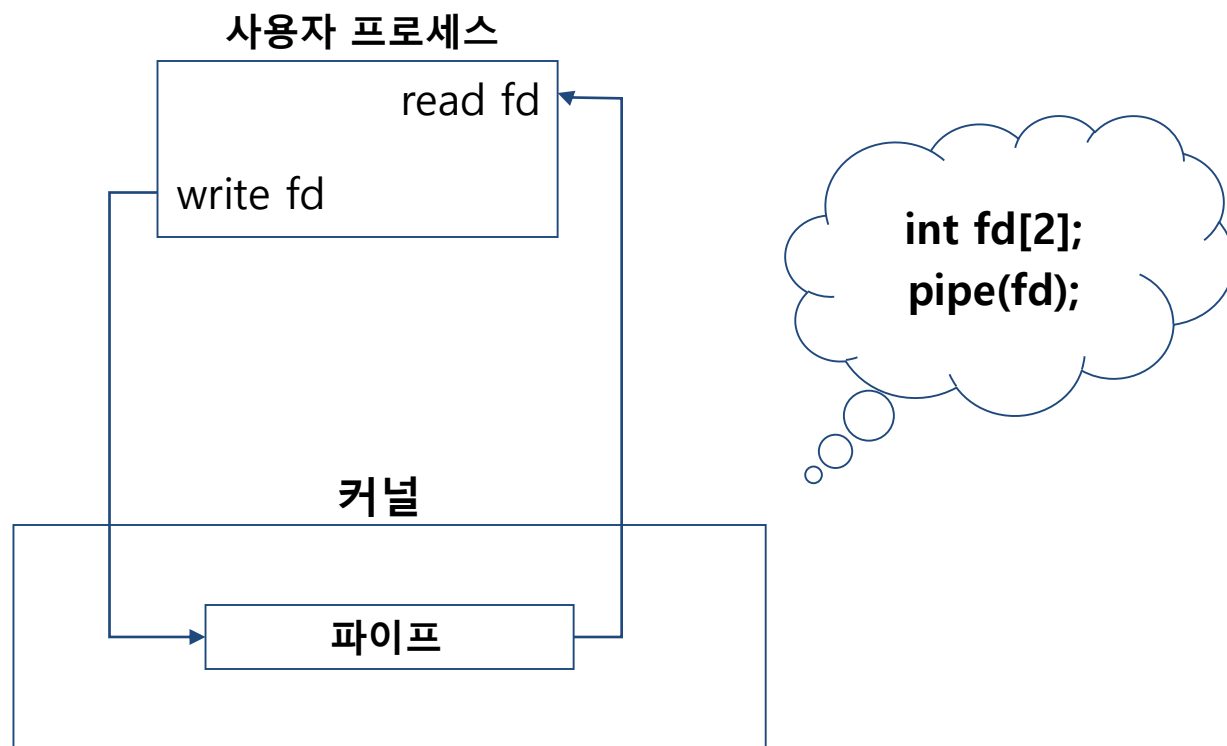
- 한 프로세스의 표준출력을 다른 프로세스의 표준 입력에 연결
- 프로세스간 단 방향 통신의 한 방법
- 동기화를 기본적으로 제공
- 가득 차거나 비어 있을때 자동으로 block된다.
- descriptor 상속으로 IPC 조인의 문제 해결

```
% who | sort
```

## ◆파이프 생성

- pipe 시스템 호출을 이용하여 만들어짐.
- 파일 처럼 동작한다.
- data를 FIFO방식으로 처리
- `filedes[0]` : 읽기 위하여 사용됨
- `filedes[1]` : 쓰기 위하여 사용됨
- 파이프의 사용을 마쳤을 때: `close()`
- 파이프에서 데이터를 읽을 때: `read()`
- 파이프에 데이터를 쓸 때: `write()`
- 파이프와 표준 입출력을 연결할 때 : `dup()`
- block을 해제할때 : `fcntl()`

## ◆ 프로세스와 파이프와의 관계



```
#include <stdio.h>

#define MSGSIZE 16

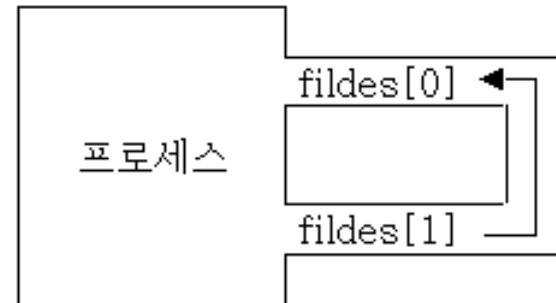
char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";

main()
{
    char inbuf[MSGSIZE];
    int p[2], j;

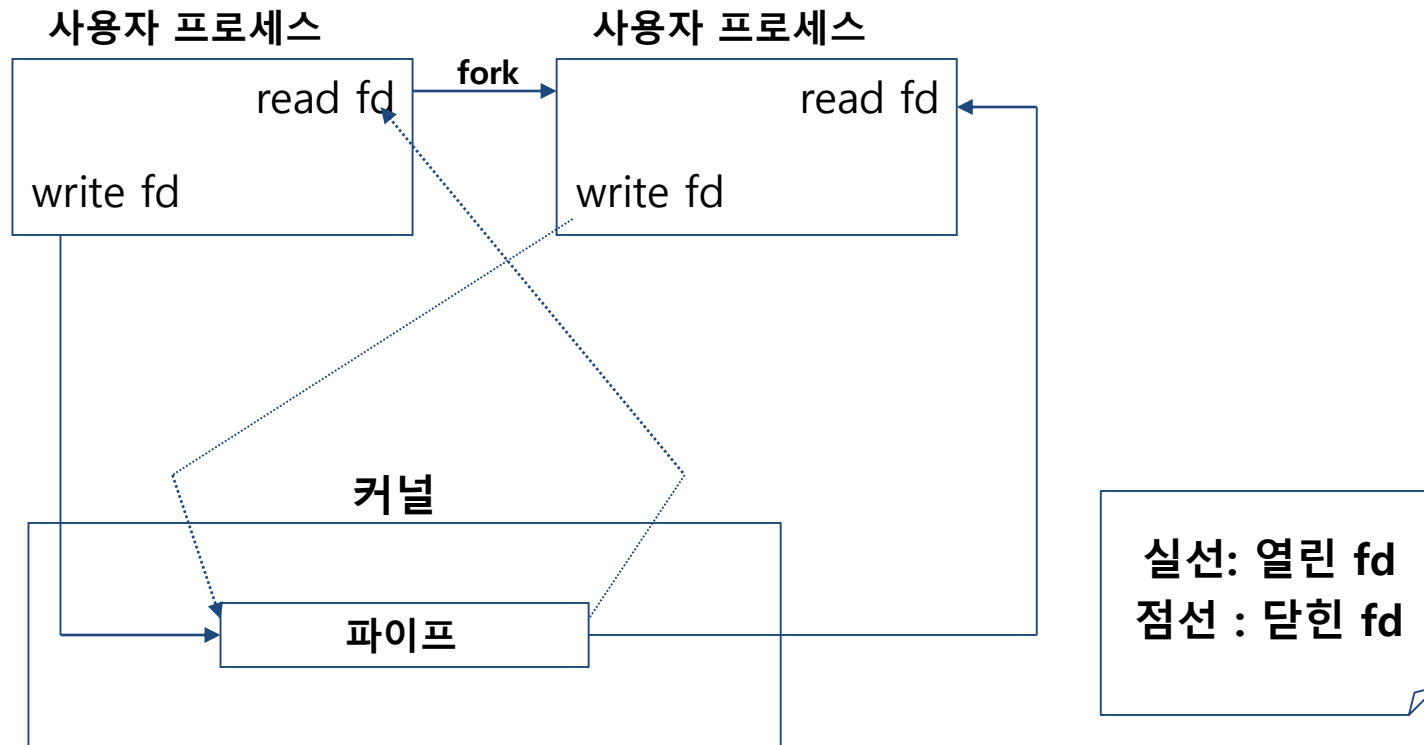
    if (pipe(p) < 0) {
        perror("pipe call");
        exit(1);
    }
```

```
    /* write down pipe */
    write(p[1], msg1, MSGSIZE);
    write(p[1], msg2, MSGSIZE);
    write(p[1], msg3, MSGSIZE);

    for (j=0; j<3; j++) {
        read(p[0], inbuf, MSGSIZE);
        printf("%s", inbuf);
    }
    exit(0);
}
```



## ◆ fork후에 프로세스와 파이프와의 관계



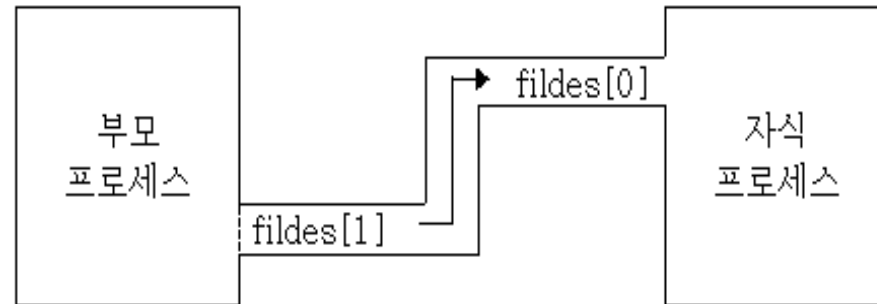


```
#include <stdio.h>

#define MSGSIZE 20
int main() {
    int fd[2],pid;
    char msgout[MSGSIZE] = "Hello,
    world\n";
    char msgin[MSGSIZE];

    if (pipe(fd) == -1)
    {
        perror("pipe()");
        exit(1);
    }
    if ((pid = fork()) > 0)
    {
        close(fd[0]);
        write(fd[1], msgout, MSGSIZE);
    }
```

```
    else if (pid == 0)
    {
        close(fd[1]);
        read(fd[0],msgin,MSGSIZE);
        puts(msgin);
    }
    else
    {
        perror("fork()");
        exit(2);
    }
}
```



```
#include <stdio.h>

int main(int argc, char **argv) {
    int fd[2], pid;

    if (pipe(fd) == -1)
    {
        perror("pipe()");
        exit(1);
    }
    if ((pid = fork()) == 0)
    {
        /* child process */
        close(fd[1]);
        dup2(fd[0], 0);
        execlp(argv[2], argv[2],
            (char *)0);
    }
}
```

```
/* parent process */
close(fd[0]);
dup2(fd[1], 1);
execlp(argv[1], argv[1], (char *)0);
}
```

```
% who
% who | sort
% a.out who sort
```

bipipen\_1.c

```
int main(void){
    int    childpid, pipe1[2], pipe2[2];
    if(pipe(pipe1) < 0 || pipe(pipe2) < 0)
        printf("pipe error");
    childpid = fork();
    if(childpid > 0) {                /* parent process */
        close(pipe1[0]);
        close(pipe2[1]);
        client(pipe2[0], pipe1[1]);
        while(wait((int *) 0) != childpid);
        close(pipe1[1]);
        close(pipe2[0]);
    } else {                          /* child process */
        close(pipe1[1]);
        close(pipe2[0]);
        server(pipe1[0], pipe2[1]);
        close(pipe1[0]);
        close(pipe2[1]);
    }
}
```

bipipen\_2.c

```
#include <stdio.h>
#define MAXBUFF 1024
client(int readfd, int writefd){
    char buff[MAXBUFF];
    int n;

    if(fgets(buff, MAXBUFF, stdin) == NULL)
        printf("client: filename read error");
    n = strlen(buff);
    if(buff[n-1] == '\n')
        n--;
    if(write(writefd, buff, n) != n) /* 파일 이름 전송 */
        printf("client: filename write error");

    while((n = read(readfd, buff, MAXBUFF)) > 0) /* 파일 데이터 수신 */
        if(write(1, buff, n) != n)
            printf("client: data write error");

    if(n < 0)
        printf("client: data read error");
}
```

bipipen\_3.c

```
#include <stdio.h>
#define MAXBUFF 1024
server(int readfd, int writefd){
    char buff[MAXBUFF];
    int n, fd; extern int errno;

    if ((n = read(readfd, buff, MAXBUFF)) <= 0) /* 파일 이름 수신 */
        printf("server: filename read error");
    buff[n] = '\0';
    if ((fd = open(buff, 0)) < 0) {
        strcat(buff, " can't open\n");
        n = strlen(buff);
        if(write(writefd, buff, n) != n) /* 에러 메시지 전송 */
            printf("server: errmsg write error");
    }
    else {
        while((n = read(fd, buff, MAXBUFF)) > 0)
            if(write(writefd, buff, n) != n) /* 파일 데이터 전송 */
                printf("server: data write error");
    }
}
```



Result of `fp = popen(command,"r");`

- `command`의 표준 출력을 반환된 파일 포인터로 읽음



Result of `fp = popen(command,"w");`

- 반환된 파일 포인터로의 출력을 `command`의 표준 입력으로

명령어("`command`")를 사용하여 파이프를 만들고  
`fork/exec`를 수행한다.

```
#include <stdio.h>
int main(void)
{
    FILE *pipein_fp, *pipeout_fp;
    char readbuf[80];

    /*popen() 호출을 사용하여 단방향 파이프를 만든다*/
    pipein_fp = popen("ls", "r");

    /*popen() 호출을 사용하여 단방향 파이프를 만든다*/
    pipeout_fp = popen("sort", "w");

    /*반복 처리*/
    while(fgets(readbuf, 80, pipein_fp))
        fputs(readbuf, pipeout_fp);
```

```
    /*파이프를 닫는다*/
    pclose(pipein_fp);
    pclose(pipeout_fp);

    return(0);
}
```

## ◆ Block되지 않는 Read와 Write

- 파이프에 자료가 없을때 read : block된다.
- 파이프에 빈공간이 없을때 write : block된다.
- alarm()을 이용한 대기시간 조절
- fstate
- fcntl(filedes, F\_SETFL, O\_NDELAY)



# 5. IPC Programming

---

5.1 Signal

5.2 Pipe

**5.3 Message Queue**

5.4 Semaphores

5.5 Shared Memory

---

### ◆제공되는 기능

- - System V에서 처음 제공된 IPC
- - 채널 조인 방법과 인터페이스에 공통점 존재
- 메시지 큐(Message Queue) : 메시지 전달
- 세마포어(Semaphore) : 동기화
- 공유메모리(Shared Memory) : 메시지 전달

◆ Name Space

- 생성 가능한 채널 이름의 집합
- IPC 채널 식별자의 필요
- 생성 가능한 채널번호의 집합
- Key\_t의 key를 name space값으로 사용
- 여러 프로세스들이 IPC자원을 쉽게 공유하도록 해줌

IPC 종류	Name space	채널 식별자
파이프	없음	File descriptor
FIFO	Pathname	File descriptor
메세지큐	Key_t key	identifier
공유메모리	Key_t key	identifier
세마포어	Key_t key	identifier
소켓(unix domain)	Pathname	File descriptor

관련 없는 프로세스간의 IPC채널 충돌이 없도록 고려해야 함.

## ◆ Interface

- 동일한 name space를 갖고있기 때문에 Interface가 유사
- 동일한 structure에 의해 커널이 관리

IPC 종류	메시지 큐	세마포어	공유메모리
Include 화일 IPC 채널 생성, 열기 IPC 채널 제어 IPC 오퍼레이션	<sys/msg.h> msgget msgctl msgsnd, msgrcv	<sys/sem.h> semget semctl semop	<sys/shm.h> shmget shmctl shmat, shmdt

```

struct ipc_perm{
    ushort uid;
    ushort gid;
    ushort cuid;
    ushort cgid;
    ushort mode;
    ushort seq;
    key_t key;
}

```

◆ Permission & Creation

- file 접근권한과 동일
- superuser는 언제나 어떤 채널에도 접근가능

플래그	Key 존재 않음	Key 이미 존재
없음 IPC_CREAT IPC_CREAT   IPC_EXCL	errno = ENOENT 새로운 채널 ID 리턴 새로운 채널 ID 리턴	기존 채널 ID 리턴 기존 채널 ID 리턴 Errno = EEXIST

### ◆ Shell Command를 통한 IPC 제어

- ipcs
  - 모든 IPC 객체의 상태를 얻는데 사용
  - IPC 객체에 대한 커널의 저장조직을 볼수있다.
  - ipcs -q: 메세지 큐(message queues)만을 보여준다.
  - ipcs -s: 세마퍼(semaphore)만을 보여준다.
  - ipcs -m: 공유 메모리(shared memory)만을 보여준다
  - ipcs --help: 부가적인 아규먼트(arguments)
- lpcrm
  - 커널로부터 IPC객체를 제거한다.
  - lpcrm <msg | sem | shm> <IPC ID>

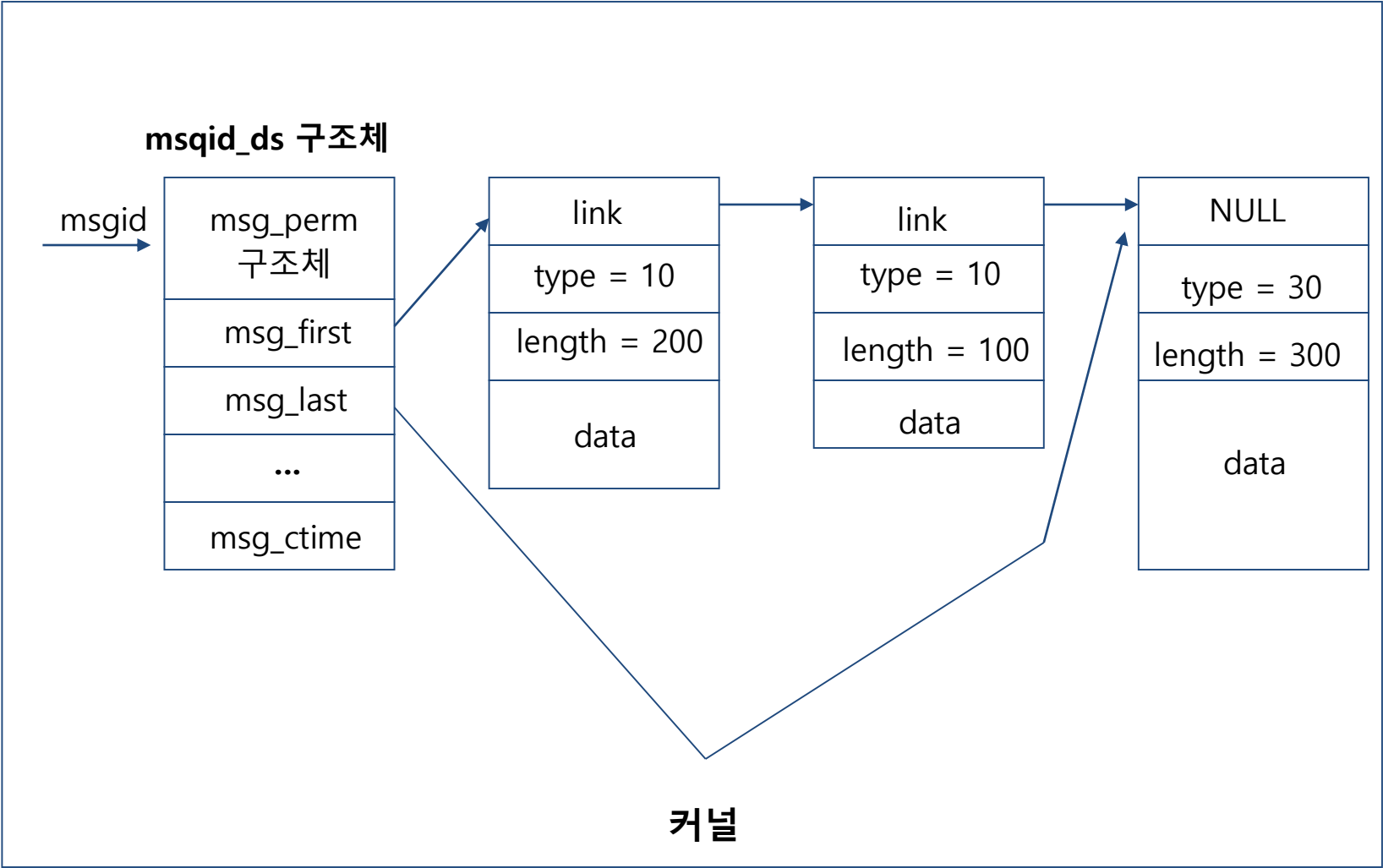
◆ msgflag

8진수값	플래그	기능
0400	MSG_R	큐 소유자에 의해
0200	MSG_W	read가능
0040	MSG_R >> 3	큐 소유자에 의해
0020	MSG_W >> 3	write가능
0004	MSG_R >> 6	그룹에 의해 read 가능
0002	MSG_W >> 6	그룹에 의해 write 가능
	IPC_CREAT	누구나 read 가능
	IPC_EXCL	누구나 write 가능
		앞에서 설명했음
		앞에서 설명했음

```
int CreateMQ(long key)
{
    return( msgget(key, IPC_CREAT | 0777) );
}

int OpenMQ(long key)
{
    return( msgget(key, 0) );
}
```





```
/* 시스템상에서 각 큐에 대한 msqid 구조 */
#include <sys/types.h>
#include <sys/ipc.h>

struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first; /* first message on queue 큐의 처음 메세지*/
    struct msg *msg_last; /* last message in queue 큐의 마지막 메세지*/
    time_t msg_stime; /* last msgsnd time 마지막으로 msgsnd가 수행된 시간*/
    time_t msg_rtime; /* last msgrcv time 마지막으로 msgrcv가 수행된 시간*/
    time_t msg_ctime; /* last change time 마지막으로 change가 수행된 시간*/
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes; /* max number of bytes on queue 큐의 최대 바이트 수*/
    ushort msg_lspid; /* pid of last msgsnd 마지막으로 msgsnd를 수행한 pid*/
    ushort msg_lrpid; /* last receive pid 마지막으로 받은 pid*/
};
```

```
/* 메시지 버퍼 */
#include <sys/types.h>
#include <sys/ipc.h>

/* msgsnd와 msgrcv 호출을 위한 메세지 버퍼 */
struct msgbuf {
    long mtype;      /* type of message 메세지 타입 */
    char mtext[1];   /* message text 메세지 내용 */
};

/* 사용자 재정의 */
typedef struct msg_q {
    long to_mtype;    /* type of message 메세지 타입 */
    long fm_mtype;    /* process id */
    char mtext[100]; /* message text 메세지 내용 */
}MSG_t;
```

데이터 영역의 크기는 가변적으로 정의 할 수 있으며  
커널은 데이터 영역의 포인터와 length를 통하여 데이터를 관리한다.

### ◆ **nbyte** - 전송되는 메시지 Size

- 메시지 타입을 제외한 데이터만의 길이

### ◆ **flag** - **block, nonblock**의 지정

- **IPC\_NOWAIT** :
  - 메시지 큐에 새로운 메시지를 저장할 공간이 없을때 즉시 리턴
  - erron는 EAGAIN
- 0 : block된다.

```
long SendMQ(int qid, long mtype, MSG_t msg)
{
    int st;

    msg.to_mtype = mtype;
    msg.fm_mtype = getpid();

    /* msgsnd() : Return 0 if OK, -1 on error */
    st = msgsnd(qid, &msg, (sizeof(MSG_t)-sizeof(long)), IPC_NOWAIT);

    if(st < 0)
        return -1L;

    return msg.fm_mtype;
}
```

### ◆ **nbyte** - 수신 메시지 Size

- 메시지 타입을 제외한 데이터만의 길이

### ◆ **flag**

- **IPC\_NOWAIT** :
  - 메시지 큐에 메시지가 없는 경우 즉시 리턴
  - `errno`는 `ENOMSG`
- **MSG\_NOERROR** :
  - 지정 : 메시지 버퍼보다 메시지가 큰 경우 데이터 절단
  - 미지정 : `msgrcv`가 실패
- 0 : block된다.

### ◆ **type**

- `type == 0` : 메시지 큐에 있는 첫번째 메시지 리턴
- `type > 0` : 메시지 큐에서 `type`와 메시지 타입이 같은 첫번째 메시지 리턴
- `type < 0` : 메시지 큐에서 `type`절대값과 같거나 작은 첫번째 메시지 리턴

```
long RecvMQ(int qid, long mtype, MSG_t *msg)
{
    int st;

    /* msgrcv() : Return recv bytes if OK, -1 on error */
    st = msgrcv(qid, msg, sizeof(MSG_t)- sizeof(long),
                mtype, IPC_NOWAIT);
    if(st < 0)
        return -1L;
    return msg->fm_mtype;
}
```

### ◆ cmd - 해당 메세지큐에 수행될 명령

- IPC\_STAT
  - 큐에서 msqid\_ds 구조체를 조회하여 버퍼 아규먼트의 주소지에 저장한다.
- IPC\_SET
  - 큐의 msqid\_ds 구조체의 ipc\_perm 멤버의 값을 지정한다.
  - msg\_perm.uid, msg\_perm.gid, msg\_perm.mode, msg\_perm.qbyte(root)
- IPC\_RMID
  - 커널로 부터 큐를 제거한다.



```
int GetFreeSizeMQ(int qid, long *freesize)
{
    int rtn;
    struct msqid_ds stat_q;

    if(qid<0)
        return -1;

    /* msgctl() : Return 0 if OK, -1 on error */
    rtn = msgctl(qid,IPC_STAT,&stat_q);
    if(rtn < 0)
        return -1;

    *freesize = stat_q.msg_qbytes - stat_q.msg_cbytes;

    return rtn;
}
```

```
int RemoveMQ(int qid)
{
    return(msgctl(qid, IPC_RMID, 0));
}
```

msglib.h

```
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

typedef struct msg_q
{
    long to_mtype;

    long fm_mtype;
    char mtext[100];
} MSG_t;

int  CreateMQ( key_t );
int  OpenMQ( key_t );
long RecvMQ( int qid, long mtype, MSG_t *msgbuf );
long SendMQ( int qid, long mtype, MSG_t msgbuf );
int  RemoveMQ( int qid );
```

msgclient.c

```
#include "msglib.h"
#include <stdio.h>

int main()
{
    int qid,st;
    MSG_t msg;

    qid = OpenMQ(5000);

    if(qid < 0)
    {
        printf("q open fail: %d\n",errno);
        return -1;
    }
}
```

msgclient.c

```
while(1) {
    memset(msg.mtext, '\0', 100);
    printf("\ninput : ");
    gets(msg.mtext);

    if(SendMQ(qid, 1L, msg) <= 0) {
        printf("q send fail: %d\n", errno);
        break;
    }

    if(!strcmp(msg.mtext, "exit")) {
        printf("Client Process Exit\n");
        break;
    }
    sleep(1);
    st = RecvMQ(qid, getpid(), &msg);
    if(st > 0)
        printf("recv : %s\n", msg.mtext);
}
return 0;
}
```

msgserver.c

```
#include "msglib.h"
main()
{
    int qid;
    MSG_t msg;

    long mtype;

    qid=CreateMQ(5000);

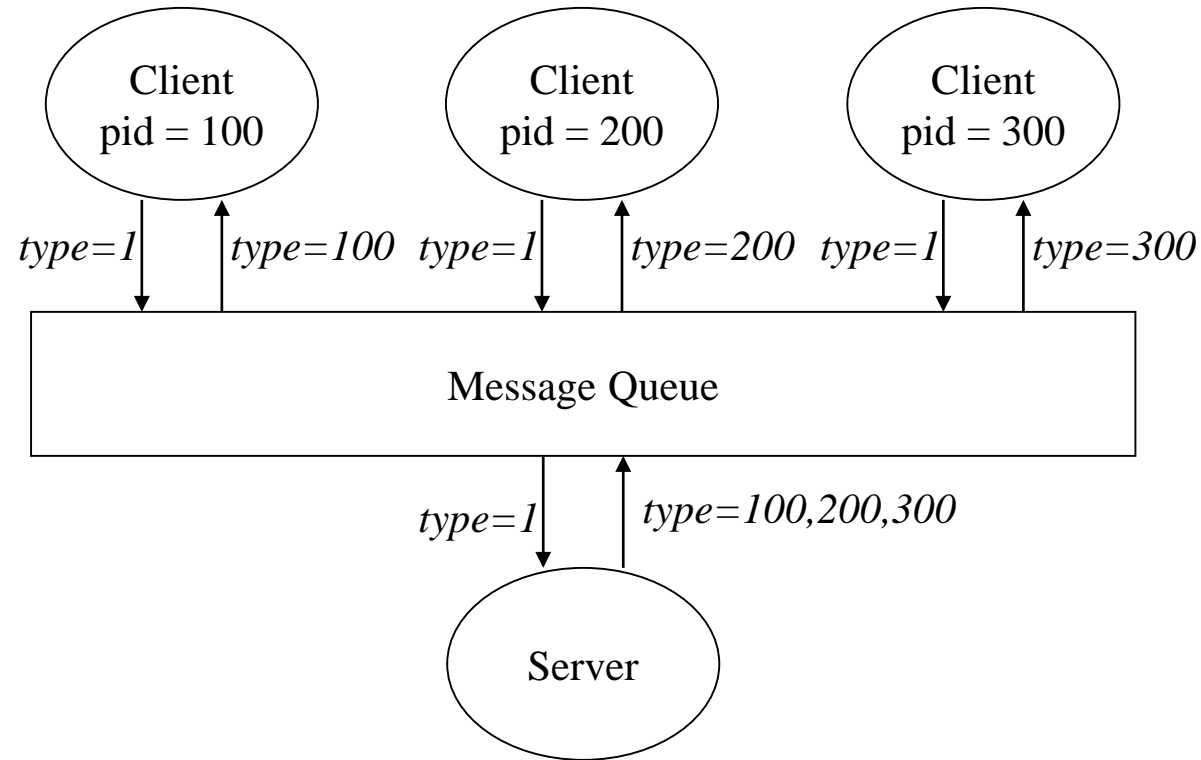
    if(qid<0)
    {
        printf("q open fail: %d\n",errno);
        return -1;
    }
}
```

msgserver.c

```
while(1)
{
    mtype = RecvMQ(qid, 1L,&msg);

    if(mtype > 0)
    {
        if(!strcmp(msg.mtext, "exit"))
        {
            printf("Server Process Exit\n");
            break;
        }
        printf("recv : %s\n",msg.mtext);
        if(SendMQ(qid, mtype, msg)<0)
            break;
    }
    RemoveMQ(qid);

    return 0;
}
```





# 5. IPC Programming

---

5.1 Signal

5.2 Pipe

5.3 Message Queue

**5.4 Semaphores**

5.5 Shared Memory

---

8진수값	플래그	기능
0400	SEM_R	소유자에 의해 읽기 가능
0200	SEM_A	소유자에 의해 변경 가능
0040	SEM_R >> 3	그룹에 의해 읽기 가능
0020	SEM_A >> 3	그룹에 의해 변경 가능
0004	SEM_R >> 6	누구나 읽기 가능
0002	SEM_A >> 6	누구나 변경 가능
	IPC_CREAT	앞에서 설명했음
	IPC_EXCL	앞에서 설명했음

```
/* 시스템 상에서의 각각의 세마포어 집합에 대한 semid 자료 구조 */
#include <sys/types.h>
#include <sys/ipc.h>

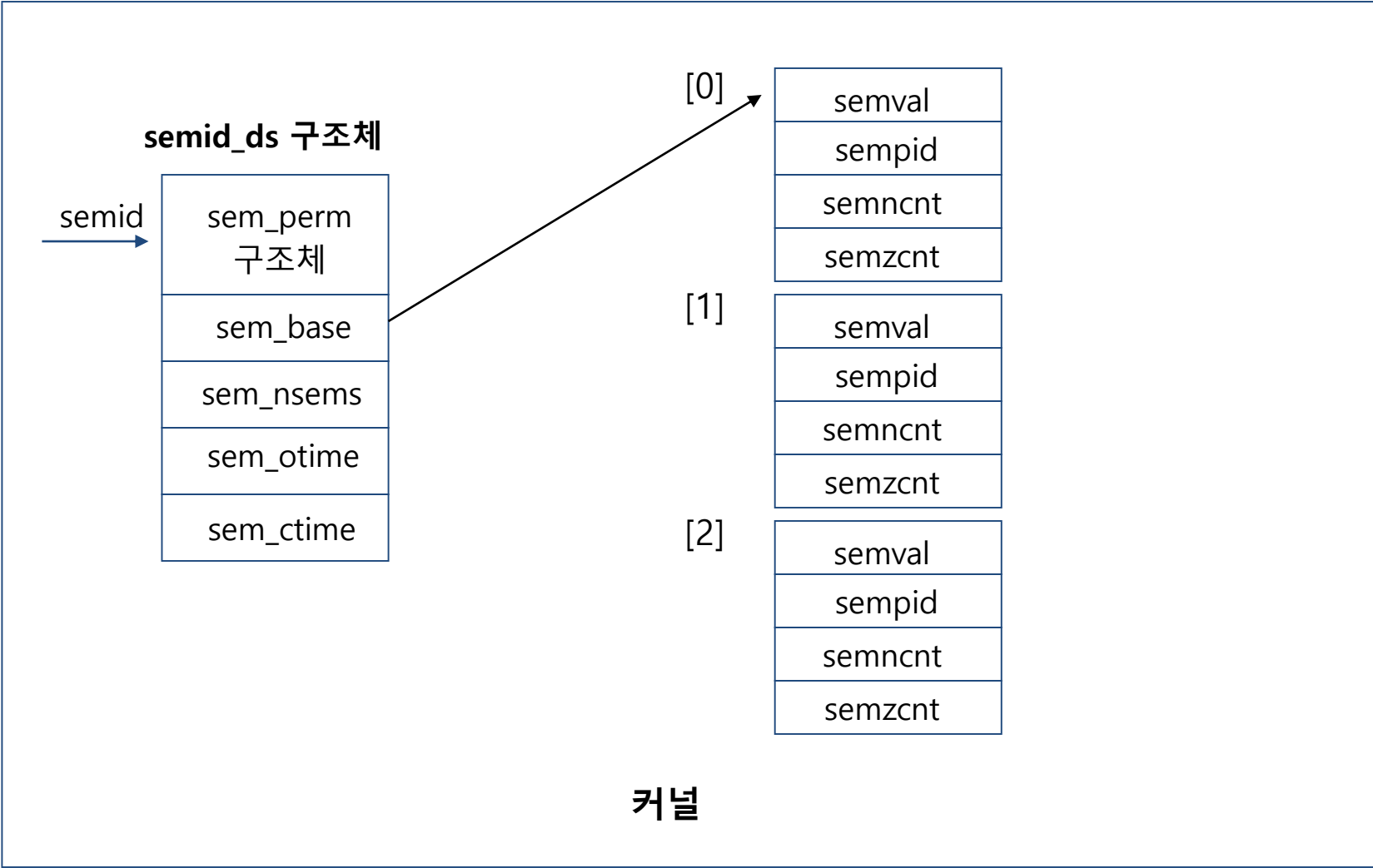
struct semid_ds {
    struct ipc_perm sem_perm;      /* permissions .. see ipc.h */
    time_t          sem_otime;     /* last semop time */
    time_t          sem_ctime;     /* last change time */
    struct sem      *sem_base;     /* ptr to first semaphore in array */
    struct wait_queue *eventn;
    struct wait_queue *eventz;
    struct sem_undo  *undo;        /* undo requests on this array */
    ushort           sem_nsems;    /* no. of semaphores in array */
};
```

```
/* 시스템에서의 각 세마포어를 위한 세마포어 구조 */  
struct sem {  
    short   sempid;      /* pid of last operation */  
    ushort  semval;      /* current value */  
    ushort  semncnt;     /* num procs awaiting increase in semval */  
    ushort  semzcnt;     /* num procs awaiting semval = 0 */  
};
```

```
int CreateSEM(key_t semkey)
{
    int status=0,semid;

    /* semget() : Return nonnegative int if OK, -1 on error */
    if((semid = semget(semkey,1,SEMPER|IPC_CREAT|IPC_EXCL))== -1)
    {
        if(errno==EEXIST)
            semid = semget(semkey,1,0);
    }
    else
        /* semctl() : Return 0 id OK, -1 on error */
        status = semctl(semid,0,SETVAL,1);

    if(semid== -1 || status== -1)
        return -1;
    return semid;
}
```



```
/* semop 시스템 호출은 이러한 배열을 갖는다 */
struct sembuf {
    ushort  sem_num;      /* 배열안에서의 세마포어 인덱스, 0부터 시작 */
    short   sem_op;       /* 세마포어 동작 */
    short   sem_flg;      /* 동작 플래그 ,IPC_NOWAIT*/
};
```

- `sem_op > 0`           // 자원반납
  - 공유변수에 `sem_op`값 만큼 더한다.
- `sem_op == 0`
  - `semop`를 호출한 프로세스는 공유변수가 0이될때 까지 기다린다.
- `sem_op < 0`           // 자원확보
  - 공유변수의 값을 `sem_op`의 절대값 만큼 빼면서 `semop`를 리턴한다.

```
int p(int semid)
{
    struct sembuf p_buf;

    p_buf.sem_num = 0;
    p_buf.sem_op  = -1;
    p_buf.sem_flg = SEM_UNDO;

    /* semop : Return : 0 if OK, -1 on error */
    if(semop(semid,&p_buf,1)==-1)
        return -1;
    return 0;
}
```



```
int v(int semid)
{
    struct sembuf p_buf;

    p_buf.sem_num = 0;
    p_buf.sem_op  = 1;
    p_buf.sem_flg = SEM_UNDO;

    /* semop : Return : 0 if OK, -1 on error */
    if(semop(semid,&p_buf,1)==-1)
        return -1;

    return 0;
}
```

```
/* semctl 시스템 호출에 대한 아규먼트 */
union semun {
    int val;                /* SETVAL을 위한 값 */
    struct semid_ds *buf;   /* IPC_STAT & IPC_SET을 위한 버퍼 */
    ushort *array;         /* GETALL & SETALL를 위한 배열 */
};
```

# 5. IPC Programming

---

5.1 Signal

5.2 Pipe

5.3 Message Queue

5.4 Semaphores

**5.5 Shared Memory**

---

```
/* 시스템안에서 각 공유 메모리 세그먼트에 대한 shmid 자료 구조 */
struct shmid_ds {
    struct ipc_perm shm_perm;          /* 동작 허가사항 */
    int      shm_segsz;                /* 세그먼트의 크기(bytes) */
    time_t   shm_atime;                /* 마지막 attach 시간 */
    time_t   shm_dtime;                /* 마지막 detach 시간 */
    time_t   shm_ctime;                /* 마지막 change 시간 */
    unsigned short shm_cpid;           /* 생성자의 pid */
    unsigned short shm_lpid;           /* 마지막 동작자의 pid */
    short     shm_nattch;              /* 현재 attaches no. */

                                        /* the following are private */

    unsigned short  shm_npages;         /* 세그먼트의 크기 (pages) */
    unsigned long   *shm_pages;         /* array of ptrs to frames -> SHMMAX */
    struct vm_area_struct *attaches;    /* descriptors for attaches */
};
```

```
int CreateSHM(long key)
{
    return( shmget(key, sizeof(SHM_t), IPC_CREAT | 0777) );
}
```

```
int OpenSHM(long key)
{
    return( shmget(key, sizeof(SHM_t), 0) );
}
```

```
void *addr
```

- `addr == 0` : 커널이 적당한 메모리 위치 선정
- `addr != 0` : 지정된 메모리 번지에 지정

```
flag
```

- `SHM_RND` : lower boundary address

```
SHM_t *GetPtrSHM(int shmid)
{
    return( (SHM_t *)shmat(shmid,(char *)0,0) );
}
```

```
int FreePtrSHM(SHM_t *shmptr)
{
    return( shmdt((char *)shmptr) );
}
```



```
int RemoveSHM(int shmid)
{
    return( shmctl(shmid,IPC_RMID,(struct shmid_ds *)0) );
}
```

### ◆ 고려사항

- IPC 채널은 프로세스 접근에 따른 참조 카운터가 없다.
- 파일 디스크립터와 다르므로 표준 입출력과 다중화 함수를 이용 할수 없다.  
(각 IPC마다 필요한 기능이 추가로 작성)
- 통신을 원하는 프로세스간에 IPC채널을 어떻게 식별할 것인가?

파일 시스템과 관련이 없어 불편하기는 하지만  
특정한 목적을 위하여 개발된 만큼 훌륭한 효과를 발휘한다.