시스템 프로그래밍 (2016)

강의 8 : 8.1~8.2 예외적인 제어흐름 - 프로세스

* Some slides are from Original slides of RBE

강의 일정

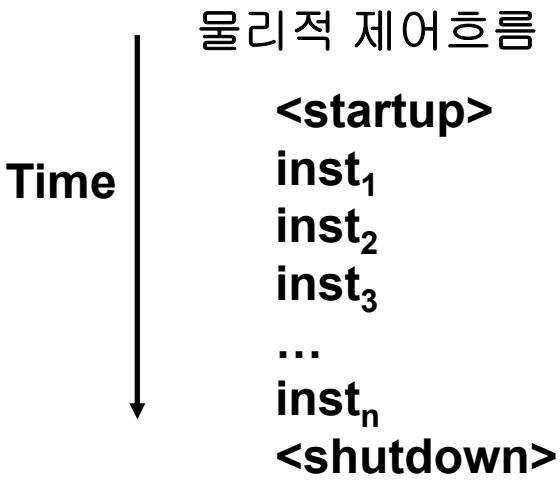
주	날짜	강의실	날짜	실습실
1	9월 1일(목)	**소개 강의**	9월 6일(화)	**리눅스 개발환경 익히기 (VI, 쉘 기본명령어들)**
2	9월 8일(목)	**정수 표현 방법**	9월 13일(화)	GCC & Make, shell script
3	9월 15일(목)	**추석 휴강**	9월 20일(화)	**C 복습과 GDB 사용하기 1 (소스 수준 디버깅)**
4	9월 22일(목)	**실수 표현 방법**	9월 27일(화)	**Data lab (GDB활용)**
5	9월 29일(목)	**어셈1 - 데이터이동**	10월 4일(화)	**어셈1 – move(실습),**
6	10월 6일(목)	**어셈2 – 제어문**	10월 11일(화)	**어셈2- 제어문 (실습)**
7	10월 13일(목)	**어셈3 – 프로시저**	10월 18일(화)	**어셈3-프로시저(실습)**
8	10월 20일(목)	**어셈보충/중간시험**	10월 25일(화)	**GDB 사용하기2(어셈수준)**
9	10월 27일(목)	**보안**(buffer overflow)	11월 1일(화)	Binary bomb 1 (GDB활용)
10	11월 3일(목)	**프로세스 1**	11월 8일(화)	Binary bomb 2 (GDB활용)
11	11월 10일(목)	**프로세스 2**	11월 15일(화)	Tiny shell 1
12	11월 17일(목)	**시그널**	11월 22일(화)	Tiny shell 2
13	11월 24일(목)	**동적메모리 1**	11월 29일(화)	Malloc lab1
14	12월 1일(목)	**동적메모리 2**	12월 6일(화)	Malloc lab2
15	12월 8일(목)	**기말시험**	12월 13일(화)	Malloc lab3

목차

1.예외적인 제어 흐름

컴퓨터의 제어 흐름

- 컴퓨터는 단순한 한가지 일만 한다
  - 전원이 들어간 이후에는 명령어(인스트럭션)들만 반복적으로 실행한다. 한번에 한 개씩.
  - 이러한 명령어의 실행흐름을 시스템의 물리적인 제어 흐름이라고 한다.



제어흐름의 변경

- 제어흐름을 변경하는 방법
  - Jumps 와 branches 명령어
  - 스택을 사용한 Call 과 return 명령어
- 이 정도로는 쓸만한 시스템을 만들기에는 부족하다
  - CPU가 시스템의 상태변화에 대응하도록 하기는 어렵다.
    - 하드디스크나 네트워크 어댑터에 데이터가 수신된 경우
    - 0으로 나누기를 시도할 때
    - 사용자가 CTRL-C를 눌렀을 때
    - 시스템 타이머가 초과되었을 때
- 시스템은 예외적인 제어흐름을 위한 메커니즘을 필요로 한다 "exceptional control flow"

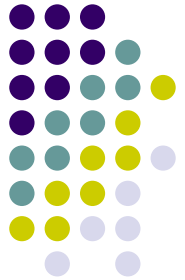
예외적인 제어 흐름

- 여러 수준에서 일어남
- 낮은 수준 →1. **예외(Exceptions)**
  - 시스템 이벤트에 대한 응답으로 제어 흐름 변화
  - 하드웨어와 OS 소프트웨어의 조합으로 구현
- 높은 수준에서

  2. **(프로세스 문맥 전환)Process context switch**
     - 하드웨어 타이머와 OS에 의한 구현

  3. **(신호) Signals**
     - OS에 의햇 구현

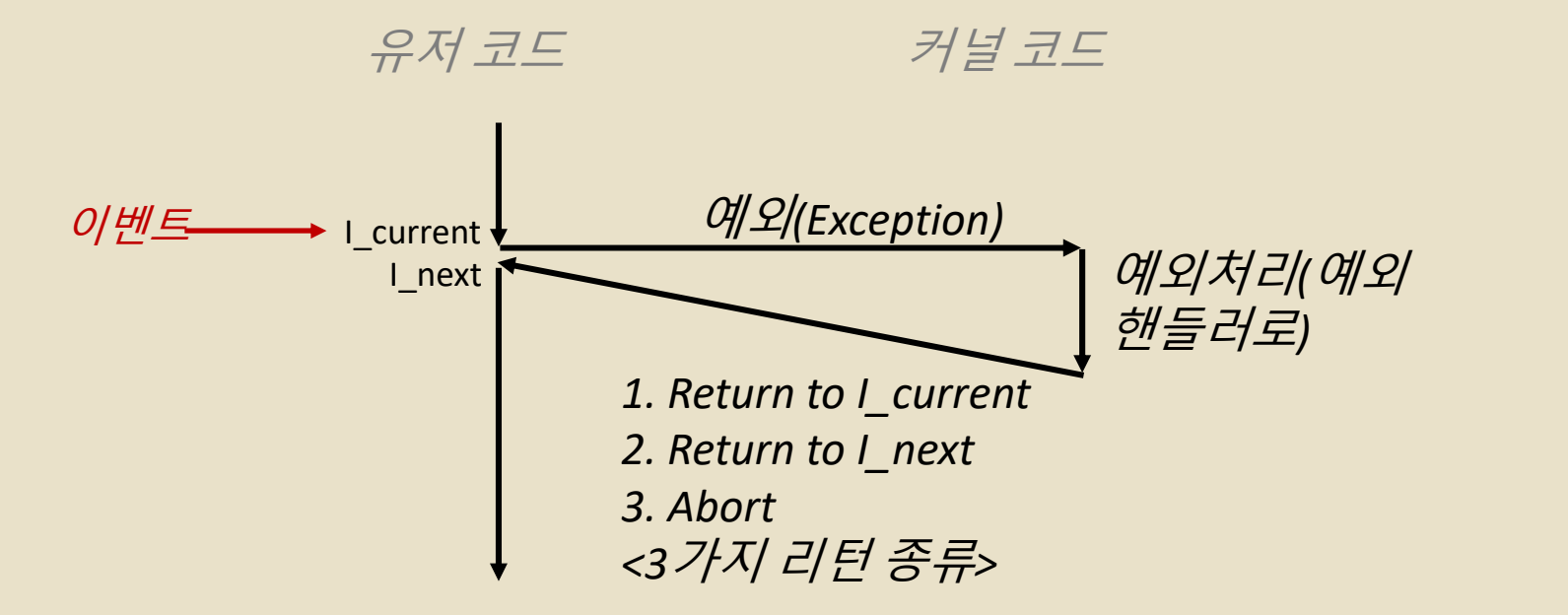
  4. **(비지역성 점프)Nonlocal jumps**: `setjmp()` and `longjmp()`
     - C 실행시간 라이브러리로 구현



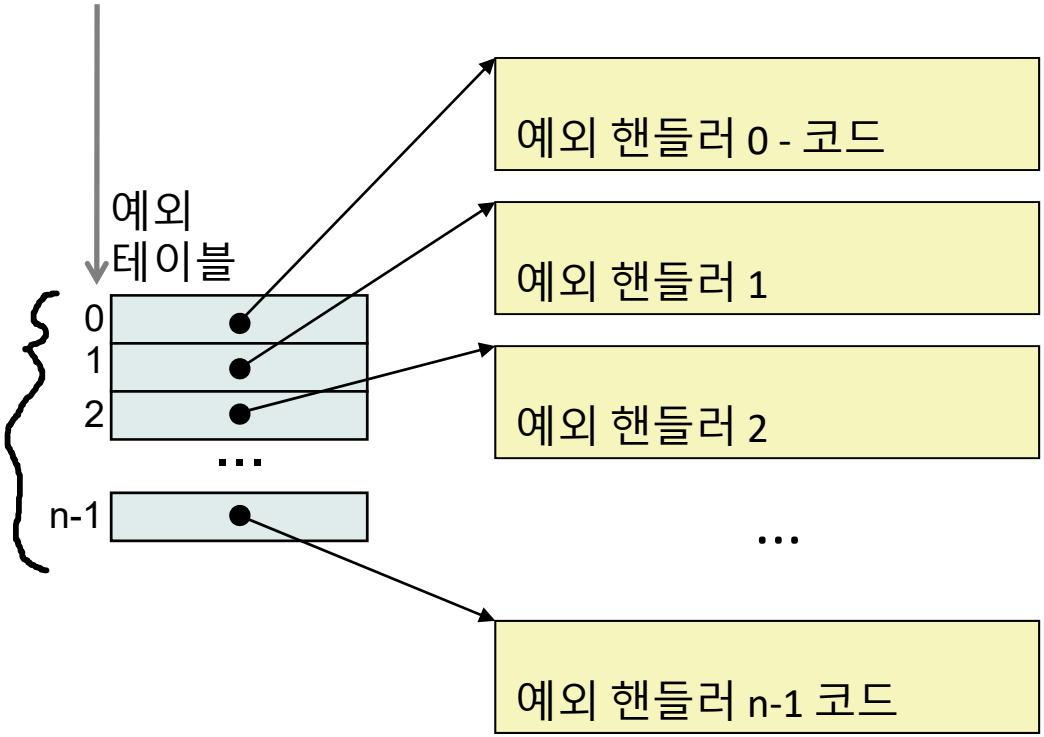
2.예외 EXCEPTIONS

예외

- 예외는 이벤트(프로세서 상태 변화)에 대응해서 OS 커널로 제어를 넘기는 것
  - 커널은 OS에서 메모리 상주 부분을 말함
  - 이벤트 예: 0으로 나눔(정수), 산술 오버플로우, 페이지 폴트(fault), I/O 요구의 완료, Ctrl-C 입력



예외 테이블(Exception Tables)



- 각 이벤트 유형은 유일한 예외 번호 k를 가짐
- k = 예외 테이블에 대한 인덱스
  (예: 인터럽트 벡터)
- 핸들러 k는 예외 k가 일어날 때 마다 호출됨

비동기형 예외(인터럽트)

- 프로세서의 외부사건으로부터 발생
  - 프로세서의 인터럽트 핀을 세팅 해서 발생을 표시
  - 핸들러 실행 후, 인터럽트 직전 실행 *명령어 다음 명령어(2. Return to I_next)로 복귀*
- 예 :
  - 타이머 인터럽트
    - 몇 ms 단위로 매번 외부 타이머 칩이 인터럽트를 트리거함
    - 커널이 사용자 프로그램을 제어할 목적으로 사용 (문맥전환 등...)
  - 외부 장치에서의 I/O 인터럽트
    - 키보드에서 Ctrl-C 입력
    - 네트워크에서 패킷이 도착함.
    - 디스크에서 (한개의 섹터) 데이터 도착함
    - 하드 리셋 인터럽트, 컴의 리셋 단추를 눌렀다, 소프트 리셋 인터럽트, 컴에서 ctl-alt-del 을 눌렀다

동기형 예외

- 명령어를 실행한 결과로 발생하는 사건들
  - Traps
    - 명령어의 결과로 발생하는 의도적인 예외
    - 예 :시스템콜(system calls), breakpoint traps, special instructions
    - 처리 후 “다음” 명령어로 복귀(*(2. Return to I_next)*)
  - Faults
    - 핸들러가 정정할 수 있는 에러의 결과로 발생
    - 예: page faults (회복가능), protection faults (회복불가), floating point exceptions.
    - Fault 를 일으킨 명령을 다시 실행하거나(*1. Return to I_current)*, Abort 한다.(*3. Abort)*
  - Aborts
    - 하드웨어 오류와 같이 복구 불가능한 에러의 결과로 발생
    - 예: 패러티 에러, 시스템 체크 에러.
    - 응용 프로그램으로 복귀할 수 없다
    - 현재 프로그램을 종료한다.(*3. Abort)*

System Calls

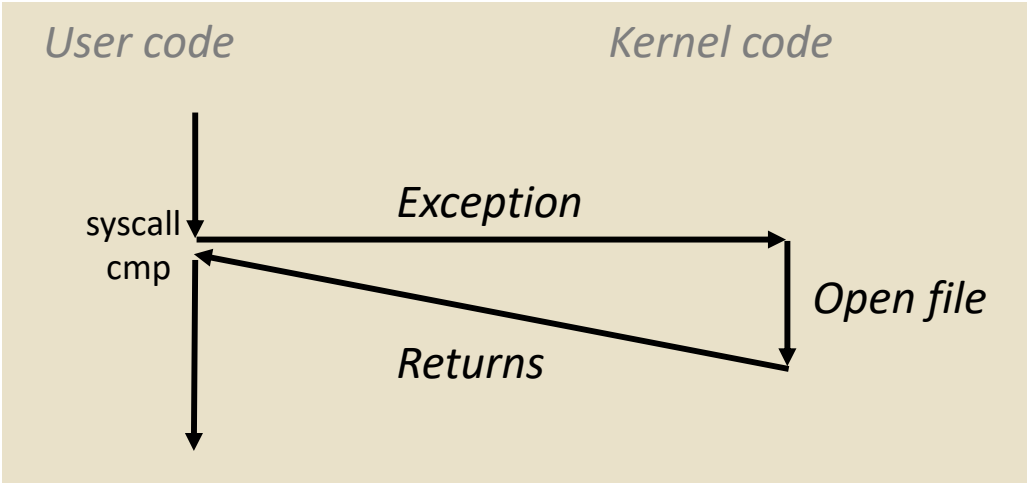
- x86-64 시스템 콜(system call)은 유일한 ID 번호 가짐
- 예:

번호	*이름*	*설명*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

Trap(시스템 콜) 예: 파일 오픈

- 호출방법: `open(filename, options)`
- `__open` 함수 호출 → `syscall` 명령어를 호출함 (호출번호 `%eax`)

```
0000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



- `%rax` 에 시스템콜번호
- 그외 인자는 `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`에
- 리턴 값은 `%rax`에
- 음수 리턴값은 오류에 해당함. `errno`

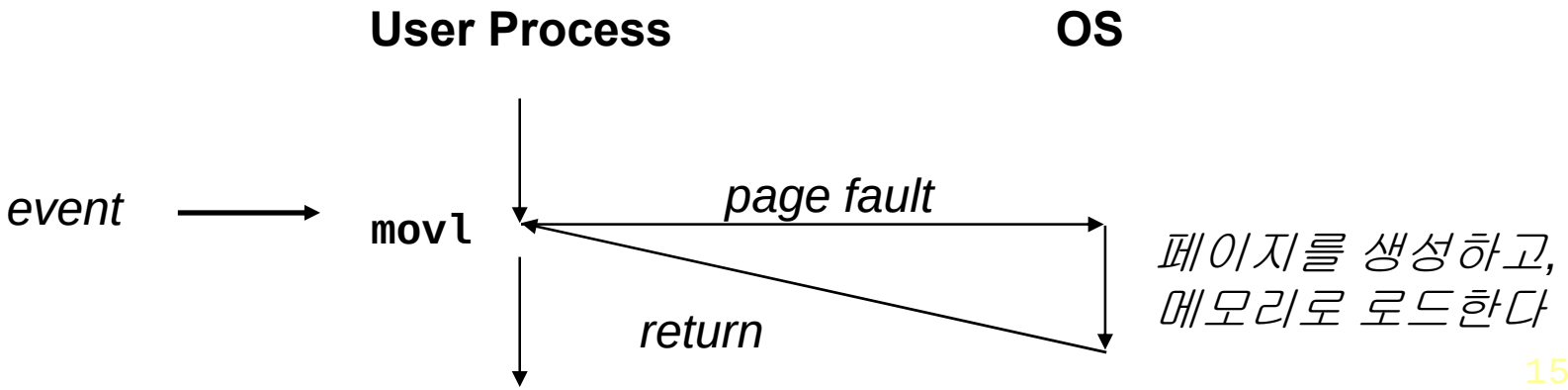
Fault 예제

```
int a[1000];
main ()
{
    a[500] = 1234;
}
```

- 메모리 참조시
  - 사용자는 메모리에 쓰기작업 수행
  - 사용자 메모리의 특정 페이지가 현재 하드디스크에 위치하는 경우

```
4004da:       c7 05 4c 13 20 00 d2   movl   $0x4d2,0x20134c(%rip)
```

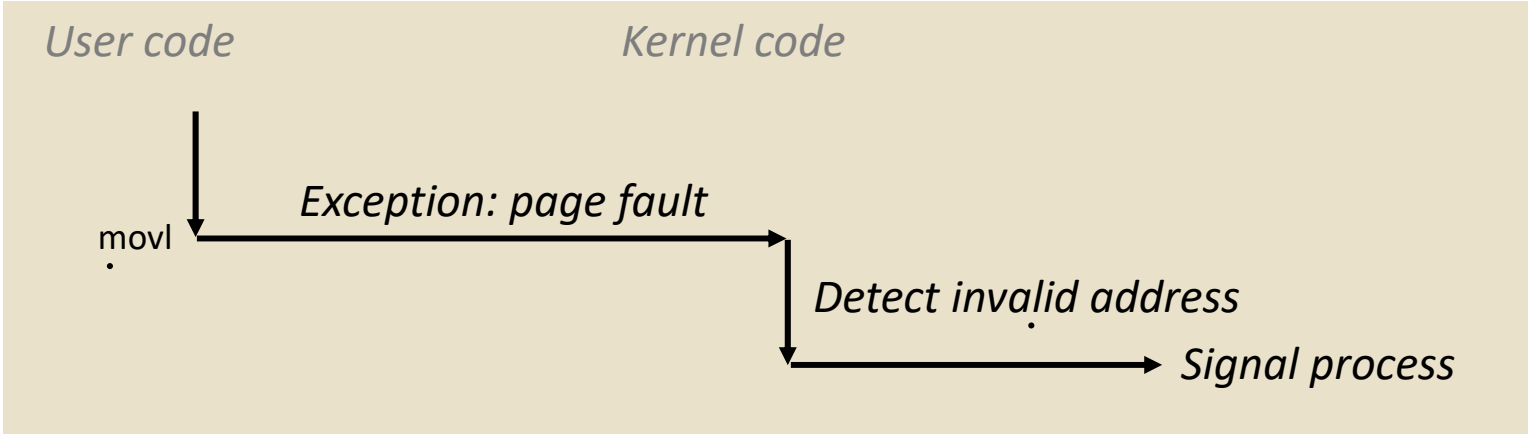
  - 페이지 핸들러는 해당 페이지를 물리메모리에 로드해야 한다
  - 이 때 페이지 오류가 발생한다
  - 오류 처리 후에 오류를 발생시킨 명령어를 다시 실행한다
  - 다시 실행할 때에는 접근이 성공한다



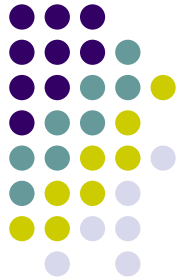
Fault 예제: 메모리 참조 오류

```
int a[1000];
main ()
{
    a[5000] = 1234;
}
```

```
4004da: c7 05 9c 59 20 00 d2   movl   $0x4d2,0x20599c(%rip)
```



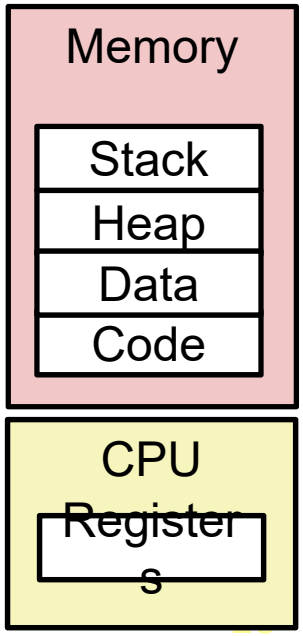
- SIGSEGV 시스널을 사용자 프로세스에 전달
- 사용자프로세스는 “세그멘테이션 폴트 " 로 종료



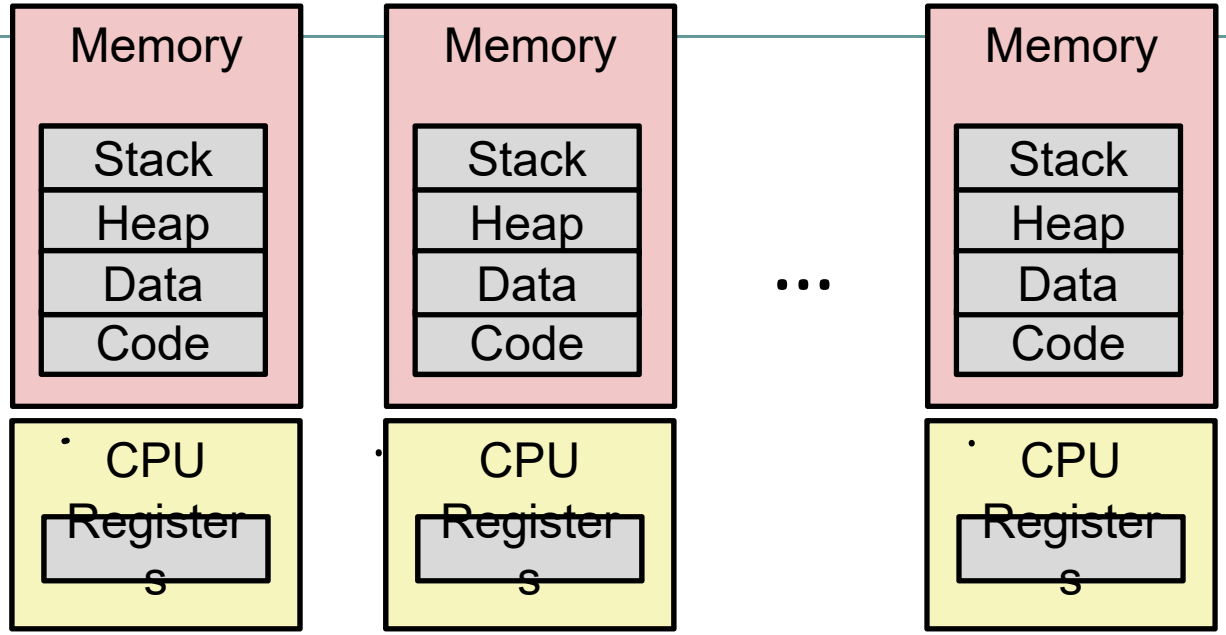
3. 프로세스

프로세스Processes

- 정의 : 프로세스는 실행하고 있는 프로그램의 한 실행 예 이다
  - 컴퓨터과학 분야에서 가장 심오한 개념중의 하나
  - 프로그램이나 프로세서를 구별 필요
- 프로세스는 프로그램에 두 개의 중요한 추상화를 제공한다:
  - 논리적인 제어흐름
    - 각 프로그램이 CPU를 독점하는 것처럼 보임.
    - 문맥교환 context switch으로 가능
  - 사적인 주소공간
    - 각 프로그램이 주 메모리를 독점하는 듯...
    - 가상메모리 기능으로 가능



다중처리: 착각

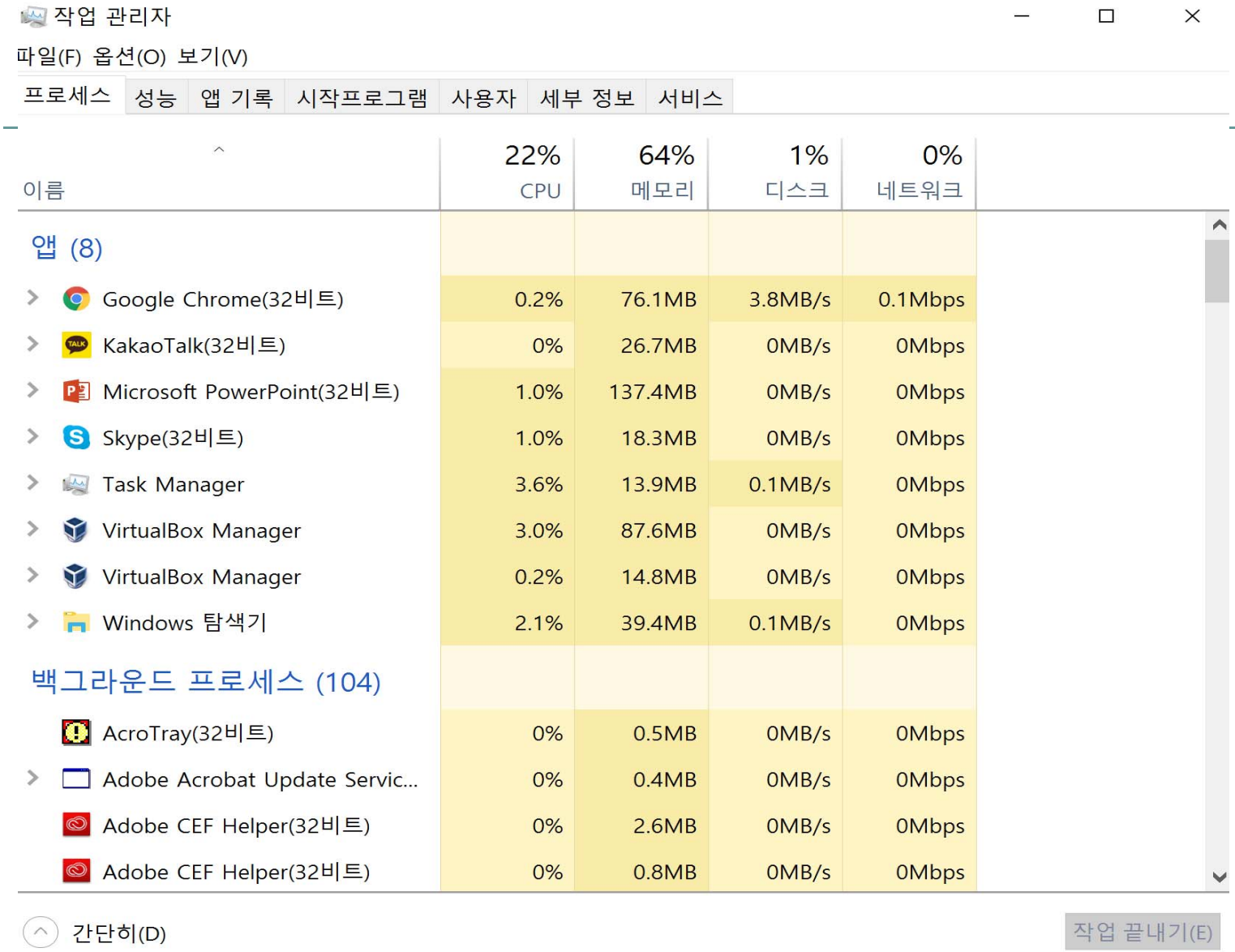


- 컴퓨터는 동시에 여러 프로세스를 실행...
  - 하나이상의 사용자 응용
    - 웹, 이메일, 편집등
  - 백그라운드 태스크들 .
    - 네트워크와 I/O 장치들의 모니터링

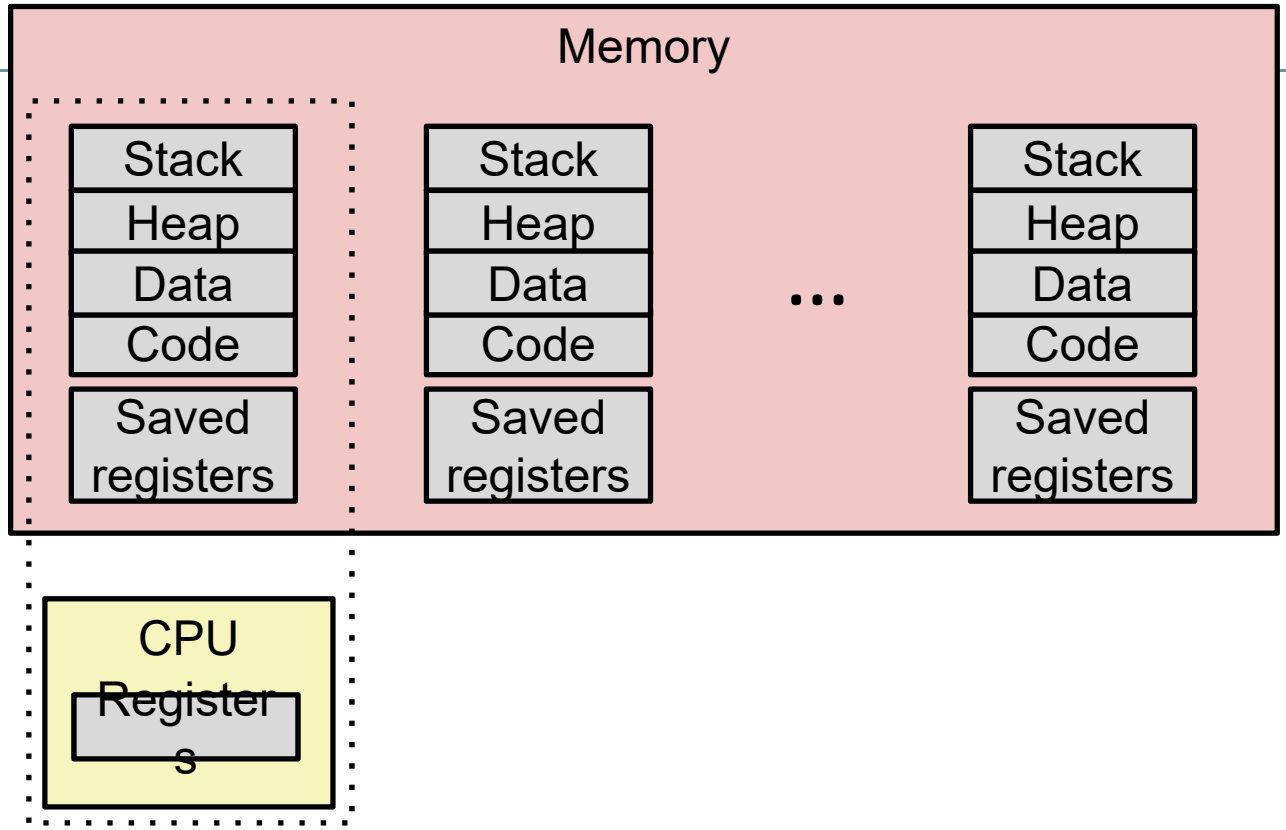
LINUX : % ps axl

```
sun@sun-virtualubuntu:~$ ps axl
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4     0     1     0  20   0 185536  6128 -      Ss   ?          0:01 /sbin/init splash
1     0     2     0  20   0      0     0 -      S    ?          0:00 [kthreadd]
1     0     3     2  20   0      0     0 -      S    ?          0:00 [ksoftirqd/0]
1     0     5     2   0 -20      0     0 -      S<   ?          0:00 [kworker/0:0H]
1     0     7     2  20   0      0     0 -      S    ?          0:00 [rcu_sched]
1     0     8     2  20   0      0     0 -      S    ?          0:00 [rcu_bh]
1     0     9     2 -100  -      0     0 -      S    ?          0:00 [migration/0]
5     0    10     2 -100  -      0     0 -      S    ?          0:00 [watchdog/0]
5     0    11     2  20   0      0     0 -      S    ?          0:00 [kdevtmpfs]
1     0    12     2   0 -20      0     0 -      S<   ?          0:00 [netns]
1     0    13     2   0 -20      0     0 -      S<   ?          0:00 [perf]
1     0    14     2  20   0      0     0 -      S    ?          0:00 [khungtaskd]
1     0    15     2   0 -20      0     0 -      S<   ?          0:00 [writeback]
1     0    16     2  25   5      0     0 -      SN   ?          0:00 [ksmd]
1     0    17     2  39  19      0     0 -      SN   ?          0:00 [khugepaged]
1     0    18     2   0 -20      0     0 -      S<   ?          0:00 [crypto]
1     0    19     2   0 -20      0     0 -      S<   ?          0:00 [kintegrityd]
1     0    20     2   0 -20      0     0 -      S<   ?          0:00 [bioset]
1     0    21     2   0 -20      0     0 -      S<   ?          0:00 [kblockd]
1     0    22     2   0 -20      0     0 -      S<   ?          0:00 [ata_sff]
1     0    23     2   0 -20      0     0 -      S<   ?          0:00 [md]
1     0    24     2   0 -20      0     0 -      S<   ?          0:00 [devfreq_wq]
1     0    28     2  20   0      0     0 -      S    ?          0:00 [kswapd0]
1     0    29     2   0 -20      0     0 -      S<   ?          0:00 [vmstat]
1     0    30     2  20   0      0     0 -      S    ?          0:00 [fsnotify_mark]
1     0    31     2  20   0      0     0 -      S    ?          0:00 [ecryptfs-kthrea]
1     0    47     2   0 -20      0     0 -      S<   ?          0:00 [kthrotld]
1     0    48     2   0 -20      0     0 -      S<   ?          0:00 [acpi_thermal_pm]
1     0    49     2   0 -20      0     0 -      S<   ?          0:00 [bioset]
```

윈도우즈 작업관리자

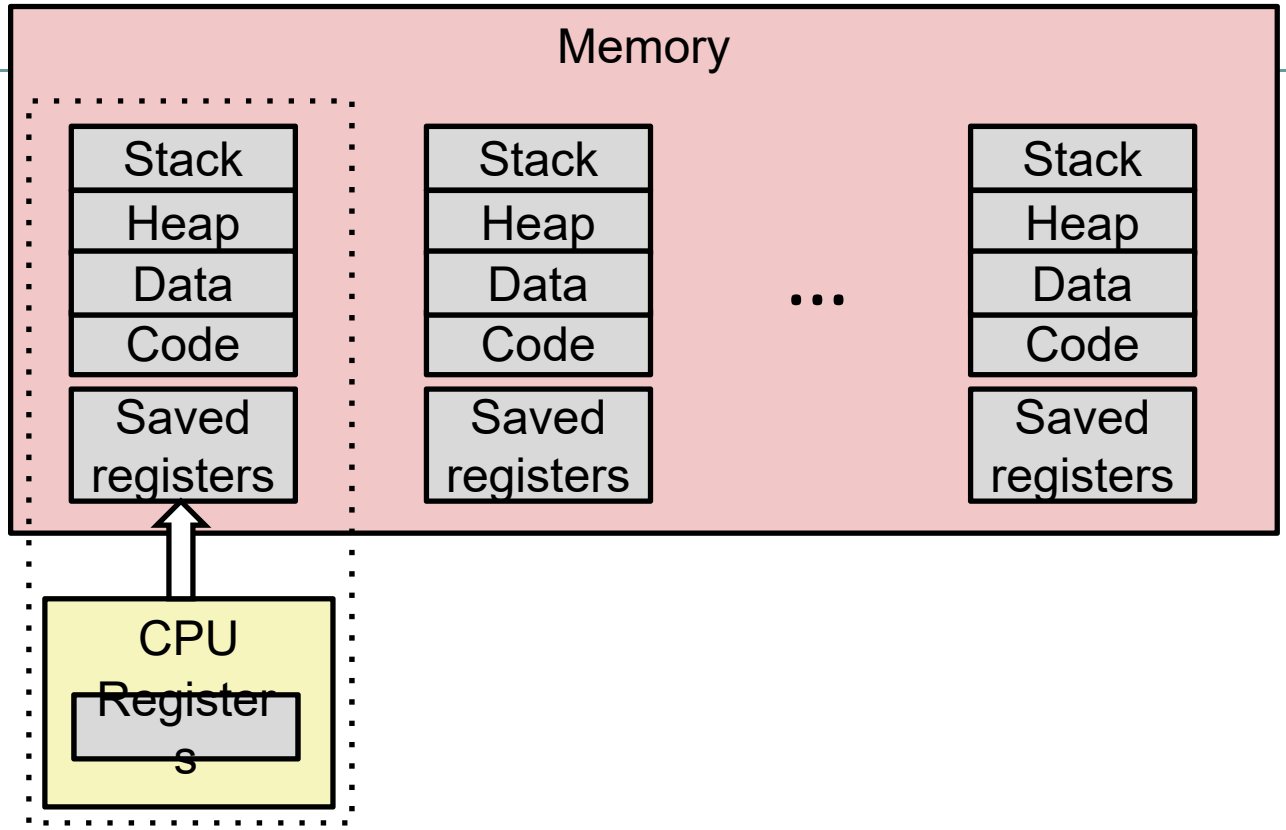

작업 관리자
파일(F) 옵션(O) 보기(V)
프로세스 성능 앱 기록 시작프로그램 사용자 세부 정보 서비스
이름
22% CPU
64% 메모리
1% 디스크
0% 네트워크
앱 (8)
Google Chrome(32비트) 0.2% 76.1MB 3.8MB/s 0.1Mbps
KakaoTalk(32비트) 0% 26.7MB 0MB/s 0Mbps
Microsoft PowerPoint(32비트) 1.0% 137.4MB 0MB/s 0Mbps
Skype(32비트) 1.0% 18.3MB 0MB/s 0Mbps
Task Manager 3.6% 13.9MB 0.1MB/s 0Mbps
VirtualBox Manager 3.0% 87.6MB 0MB/s 0Mbps
VirtualBox Manager 0.2% 14.8MB 0MB/s 0Mbps
Windows 탐색기 2.1% 39.4MB 0.1MB/s 0Mbps
백그라운드 프로세스 (104)
AcroTray(32비트) 0% 0.5MB 0MB/s 0Mbps
Adobe Acrobat Update Servic... 0% 0.4MB 0MB/s 0Mbps
Adobe CEF Helper(32비트) 0% 2.6MB 0MB/s 0Mbps
Adobe CEF Helper(32비트) 0% 0.8MB 0MB/s 0Mbps
간단히(D)
작업 끝내기(E)

다중처리의 실상



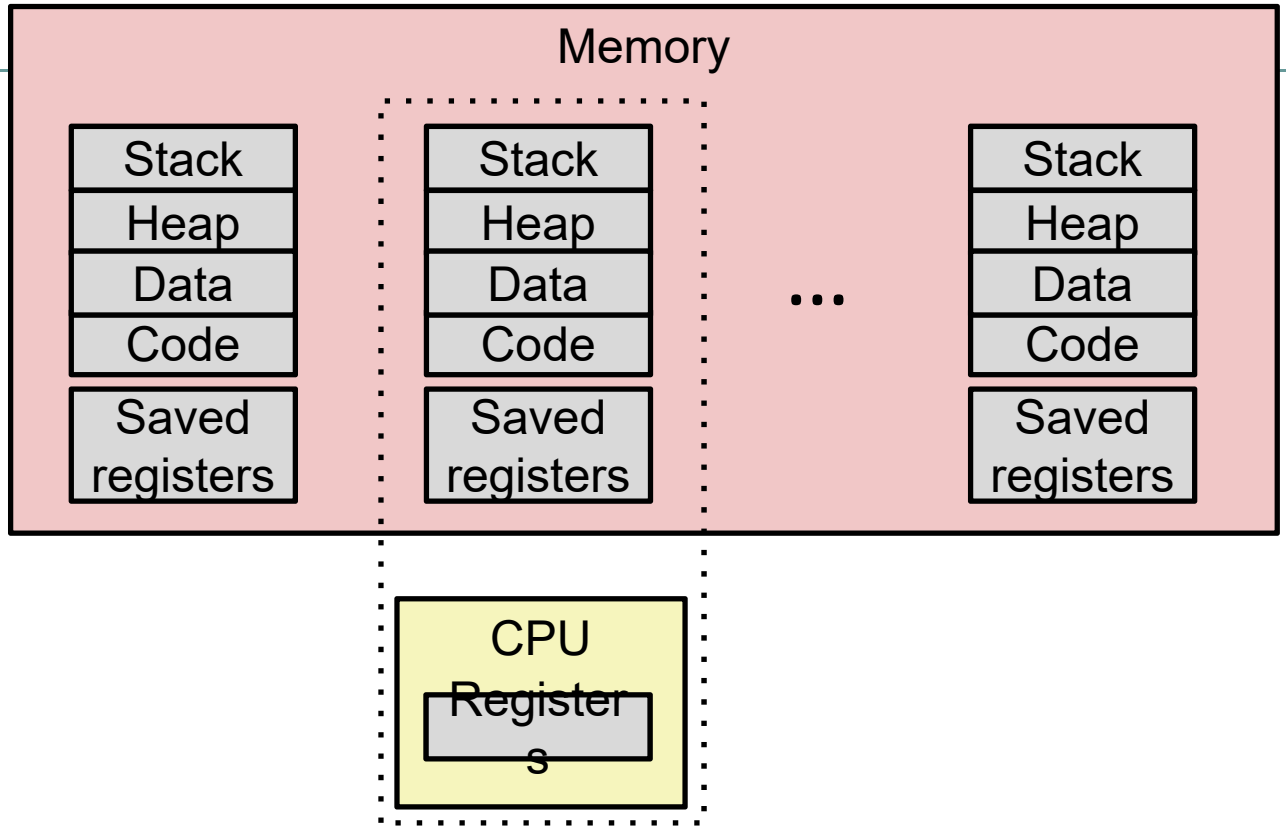
- 단일 프로세서가 여러 프로세스를 concurrently(번갈아가면서) 실행함.
  - 프로세서 실행은 interleaved(번갈아가면서 이루어짐) 멀티태스킹
  - 주소 공간은 가상메모리시스템에의해 관리됨
  - 실행중이 아닌 프로세스의 레지스터값들은 메모리에 저장됨

다중처리의 실상



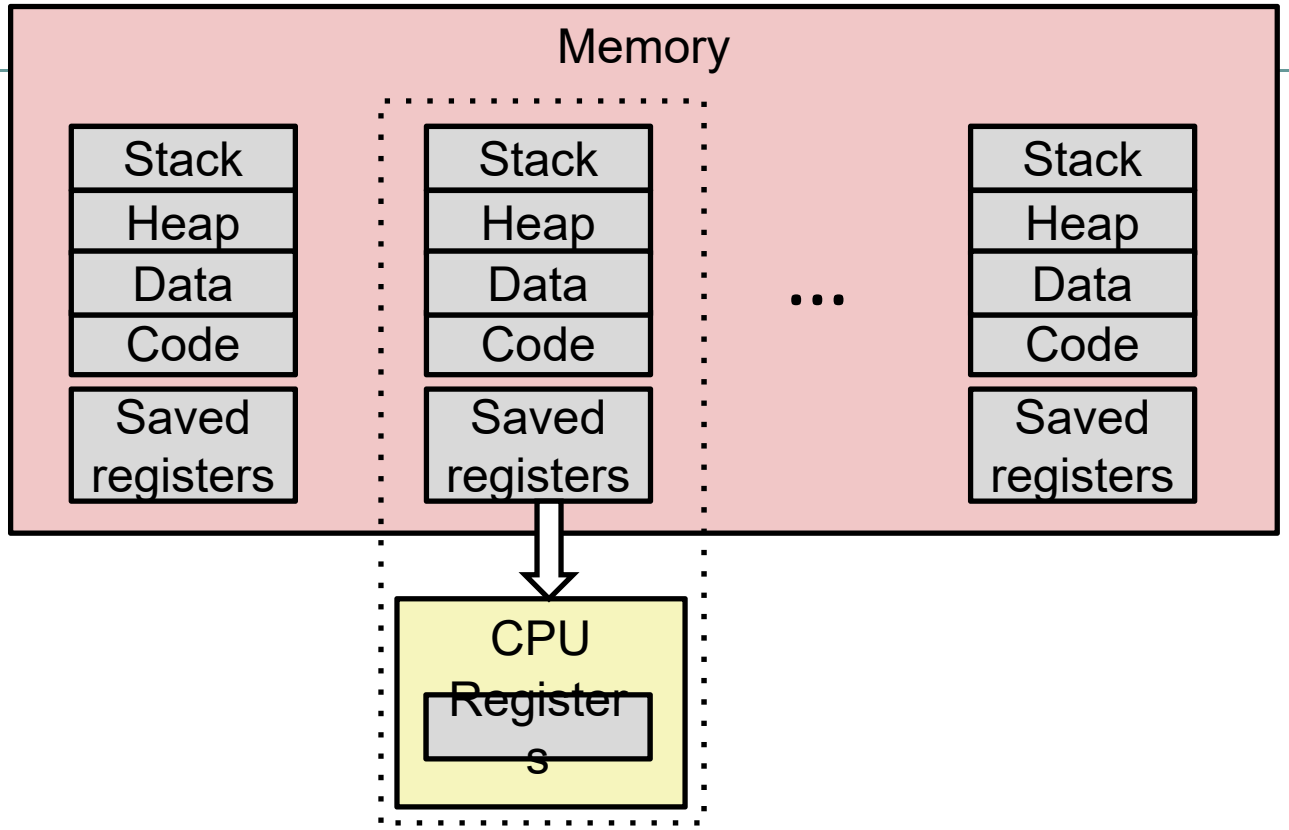
- 현재 레지스터들을 메모리에 저장

다중처리의 실상



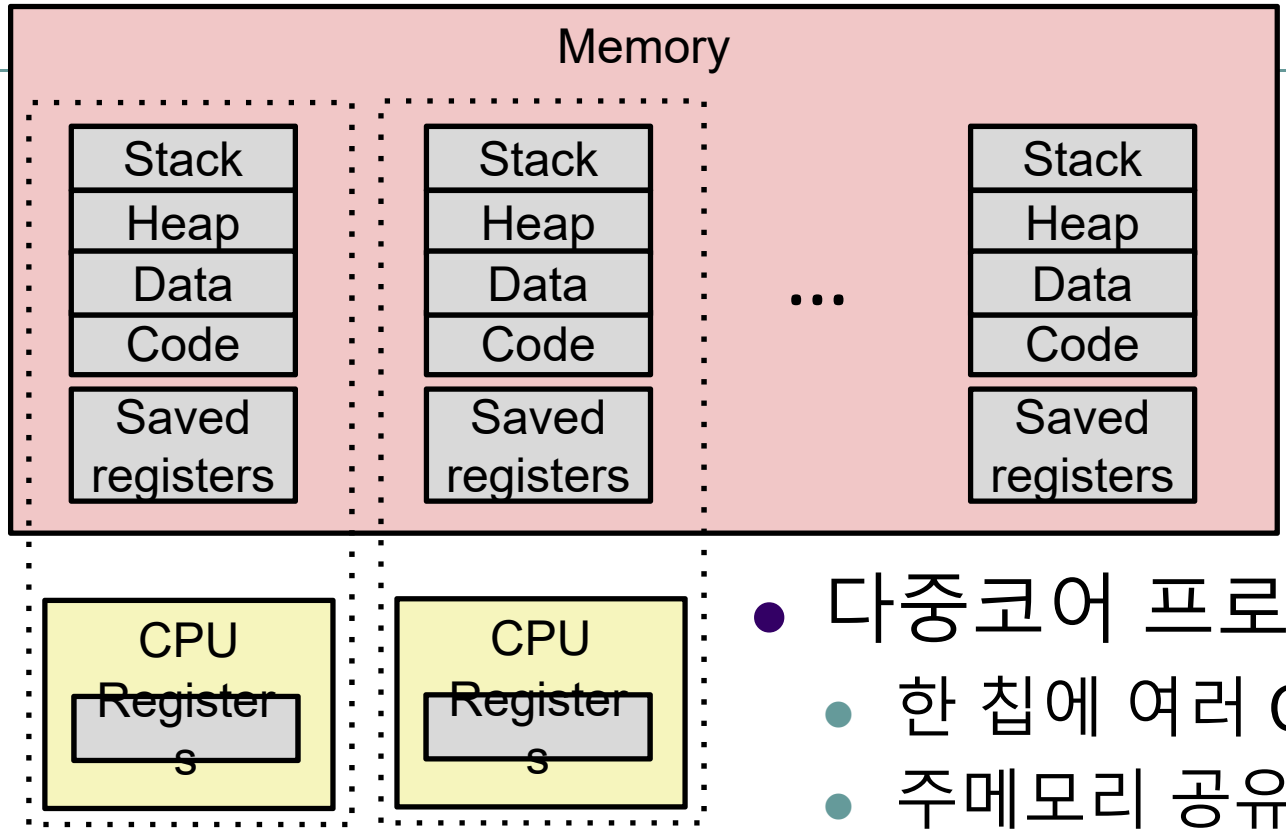
- 다음 실행 프로세스에 대한 스케줄링

다중처리의 실상



- 저장된 레지스터들을 로드하고, 주소 공간을 교체함 (문맥교환/교체)
- context switch

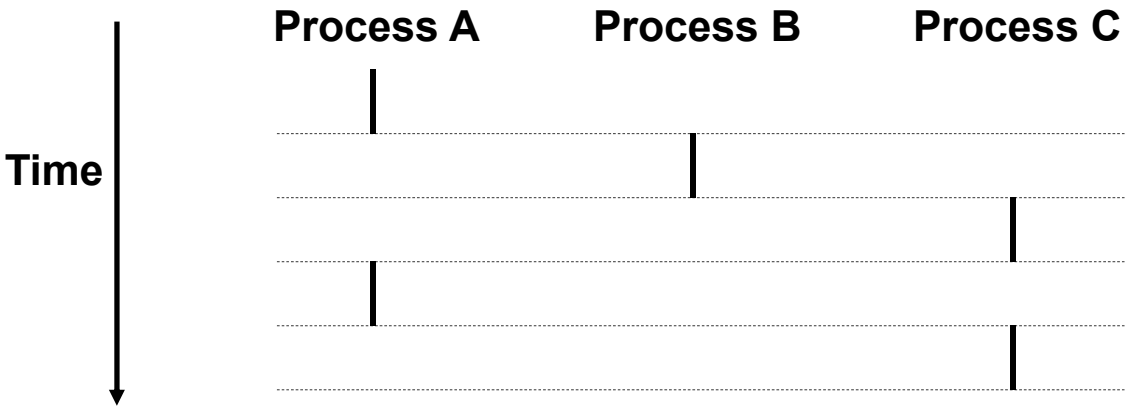
다중처리의 실상



- 다중코어 프로세서
  - 한 칩에 여러 CPU 집적
  - 주메모리 공유(캐쉬공간도)
  - 각 코어는 다른 프로세스를 동시에 실행
    - 프로세스를 코어에 스케쥴하는 것은 커널이 담당

논리적 제어흐름

각 프로세스는 자신만의 논리적인 제어흐름을 갖는다

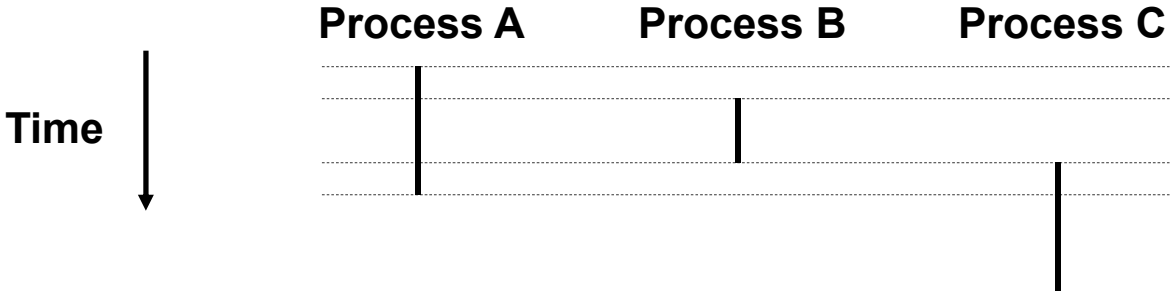


동시성 프로세스

- 두 프로세스는 그들의 실행 시간이 서로 중첩되면, 동시에 실행된다고 부른다.(*are concurrent)*
- 그렇지 않다면, 순차적으로 실행된다고 정의한다
- (*sequential.*)
- Examples:
  - 동시실행: A & B, A & C
  - 순차실행: B & C

동시프로세스의 사용자 관점

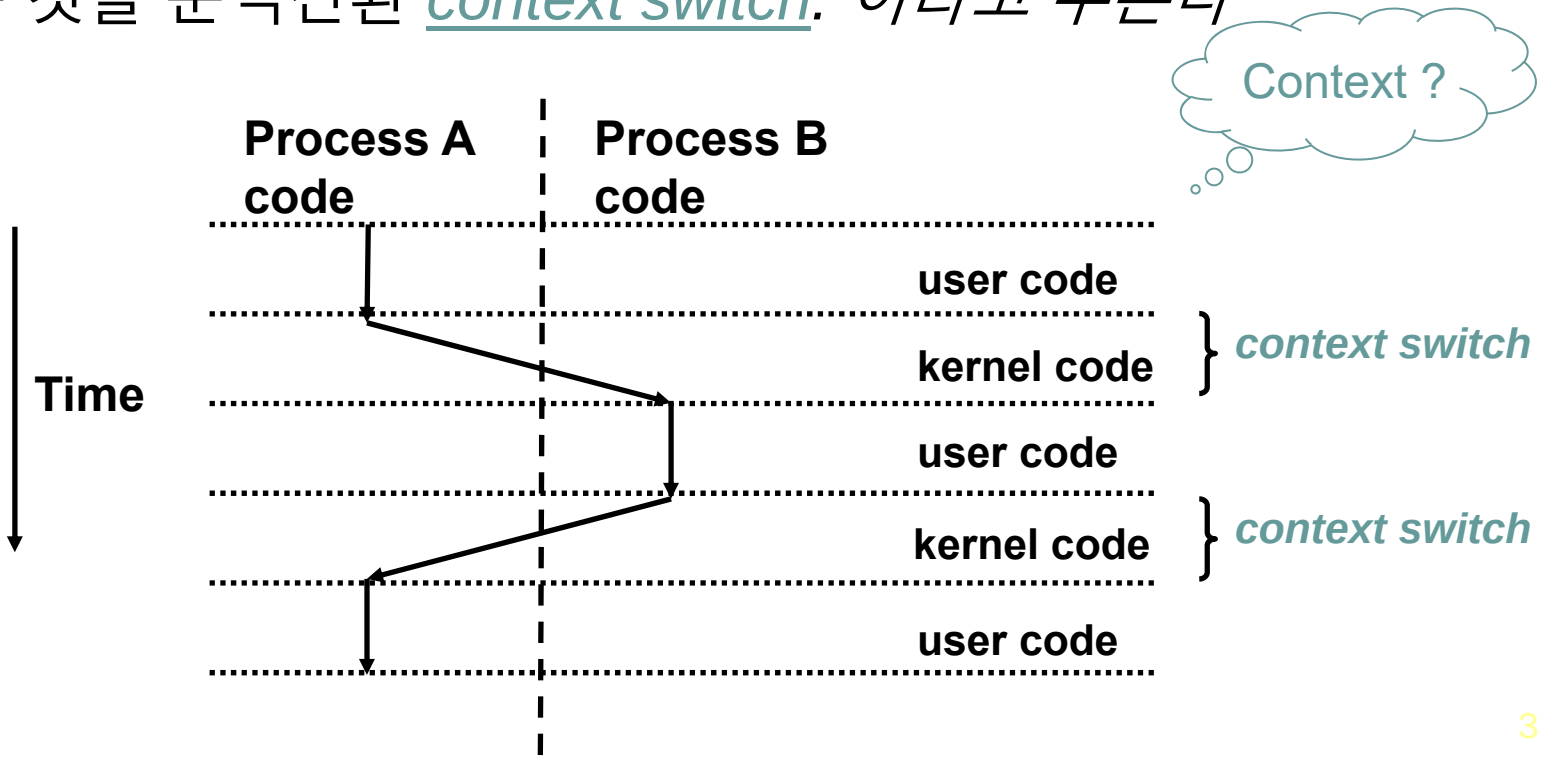
- 동시 프로세스들을 위한 제어흐름은 시간상으로는 물리적으로 분리된다.
- 그러나, 동시프로세스들이 서로 병렬로 실행된다고 생각할 수 있다.

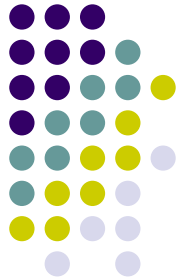


*** 멀티 태스킹 또는 타임 슬라이싱이라고 부름**

문맥전환(Context Switch)

- 프로세스는 커널이라고 부르는 운영체제에 의해서 관리된다
  - 중요 : 커널은 프로세스가 아니며, 유저 프로세스의 일부분으로 실행된다
- 한 개의 프로세스에서 다른 프로세스로 제어흐름이 넘어가는 것을 문맥전환 *context switch*. *이라고 부른다*





4. 프로세스 제어 1

프로세스의 제어

- Unix 는 C 프로그램을 이용해서 다음과 같은 프로세스 제어 기능을 제공한다
  - process ID 를 가져온다
  - 프로세스를 만들거나 종료한다
  - 자식 프로세스를 제거한다 Reaping child processes
  - 프로그램의 로딩 및 실행

시스템 콜 오류 처리

- 오류발생시, 리눅스 시스템 함수들은 대개 -1을 리턴하면, 전역변수인 errno 값에 그 원인을 집어 넣어 표시한다.
- 엄중한 규칙은:
  - 모든 시스템 함수의 리턴 상태를 체크해야 함.
  - void리턴 함수는 제외
- 예:

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s₩n", strerror(errno));
    exit(0);
}
```

오류 리포트 함수들

- 오류 리포트 함수로 단순화 가능:

```
void unix_error(char *msg) /* Unix-style error */
{
  fprintf(stderr, "%s: %s₩n", msg, strerror(errno));
  exit(0);
}
```

```
if ((pid = fork()) < 0)
  unix_error("fork error");
```

오류처러 래퍼함수

- 단순히 fork()함수 사용하는 것보다, 아래와 같은 오류 처리 래퍼 함수 사용으로 오류 처리 단순화 가능:

```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

```
pid = Fork();
```

프로세스의 제어

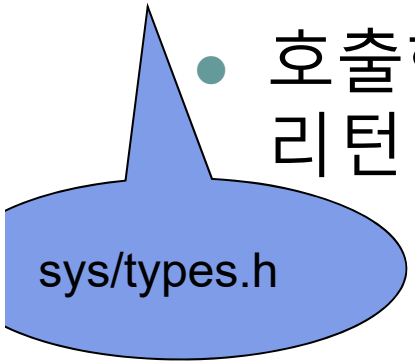
- Unix 는 C 프로그램을 이용해서 다음과 같은 프로세스 제어 기능을 제공한다
  - process ID 를 가져온다
  - 프로세스를 만들거나 종료한다
  - 자식 프로세스를 제거한다 Reaping child processes
  - 프로그램의 로딩 및 실행

프로세스 ID 얻기

- `pid_t getpid(void)`
  - 현재 프로세스의 PID 리턴
- `pid_t getppid(void)`
  - 부모 프로세스의 PID 리턴

Process ID 가져오기

- 각 프로세스는 프로세스 ID를 갖는다
- `pid_t getpid(void)`
- `pid_t getppid(void)`
  - 호출한 프로세스 또는 그 부모 프로세스의 PID 를 리턴




```
#include <stdio.h>
#include <sys/types.h>
int main()
{
        printf("hello World\n");
        printf("pid = %d\n", getpid());
        printf("ppid = %d\n", getppid());
        return 0;
}
```

```
hello World
pid = 18336
ppid = 11190
```

fork: 프로세스 만들기

- int fork(void)
  - 호출하는 프로세스(부모 프로세스)와 동일한 새 프로세스(자식 프로세스)를 생성
  - 자식 프로세스는 0을 리턴
  - 부모 프로세스는 pid 을 리턴

```
if (fork() == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Fork 함수는 한번 호출하지만, 리턴은 두번 된다는 점이 특이하다

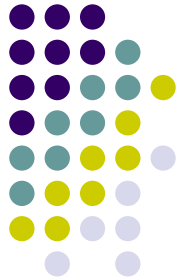
프로세스와 자식 프로세스

- fork() – 자식 프로세스 생성
- 둘 사이의 관계

1. 한번 호출, 리턴은 두번
2. 두 프로세스 동시 실행
3. 파일을 공유함(file descriptor 테이블 복사로...)
4. 변수들은 복사되지만, 서로 다른 프로세스에 속한 다른 변수가 되고, 별도의 주소 공간을 차지함.

프로세스 종료

- 종료이유들:
  - 신호를 받고 종료(그 신호의 동작이 종료하는 것인 경우)
  - main에서 리턴할 경우
  - exit 함수 호출시
- void exit(int status)
  - 종료 상태(status)를 가지고 마침
  - 관습: 정상리턴은 0을, 오류시에는 0아닌 값으로
  - main루틴에서 리턴시에, 리턴 값에 종료 상태를 설정할 수도 …
- exit은 한번 호출되고는 리턴되지 않음



5. 프로세스 제어 2 FORK()

Fork 예 #1

- Key Points
  - 부모와 자식은 동일한 코드를 실행한다
    - fork 로부터의 리턴 값으로 부모와 자식을 구분
  - 부모와 자식은 동일한 상태로 시작하지만, 각각의 사본을 갖는다
    - 출력 파일 식별자도 공유
    - 각각의 출력문의 실행 순서는 랜덤

```
$ ./fork1
Parent has x = 0
Bye from process 19242 with x = 0
Child has x = 2
Bye from process 19243 with x = 2
```

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
       printf("Child has x = %d\n", ++x);
    } else {
       printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

fork1A.c → fork1A.s

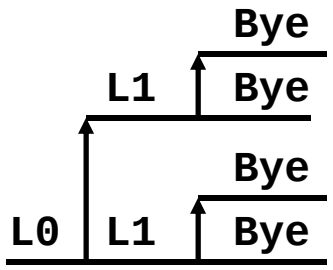
```
main:          (PARENT)
pushq   %rbx
call    fork ← parent calls fork()
testl   %eax, %eax
jne     .L2
movl    $2, %edx
movl    $.LC0, %esi
movl    $1, %edi
call    __printf_chk
movl    $2, %ebx
jmp     .L3
.L2:
movl    $0, %edx
movl    $.LC1, %esi
movl    $1, %edi
movl    $0, %eax
call    __printf_chk
movl    $0, %ebx
```

```
main:          (CHILD code copy)
pushq   %rbx
call    fork
testl   %eax, %eax ← child starts here
jne     .L2
movl    $2, %edx
movl    $.LC0, %esi
movl    $1, %edi
call    __printf_chk
movl    $2, %ebx
jmp     .L3
.L2:
movl    $0, %edx
movl    $.LC1, %esi
movl    $1, %edi
movl    $0, %eax
call    __printf_chk
movl    $0, %ebx
```

Fork 예 #2

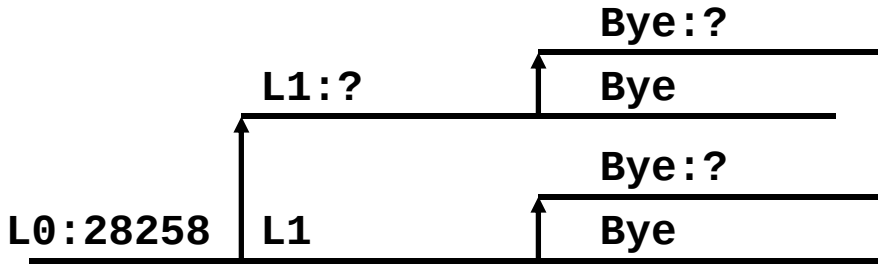
- Key Points
  - 부모와 자식이 계속 fork 하는 경우

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



프로세스 그래프를 그려서 생각하면 편리

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

int main()
{
pid_t p1=-1, p2=-1;

    printf("L0\n");
    p1 = fork();
    printf("L1, pid=%d,p1=%d,p2=%d\n",getpid(),p1,p2);
    p2 = fork();
    printf("Bye, pid=%d,p1=%d,p2=%d\n", getpid(),p1,p2);
}
```

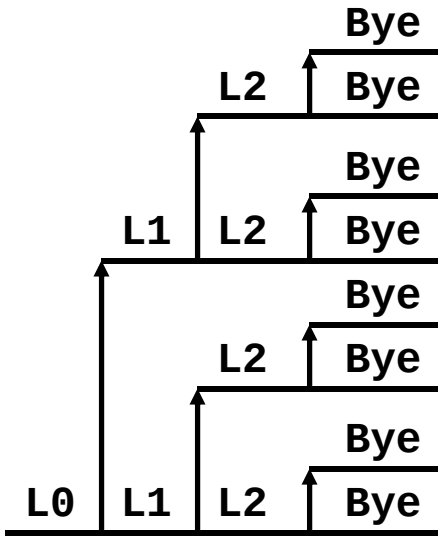



```
L0
L1, pid=7559,p1=7560,p2=-1
L1, pid=7560,p1=0,p2=-1
Bye, pid=7559,p1=7560,p2=7561
Bye, pid=7560,p1=0,p2=7562
Bye, pid=7561,p1=7560,p2=0
Bye, pid=7562,p1=0,p2=0
```

Fork 예 #3

- Key Points
  - 부모와 자식이 계속 fork 하는 경우

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Modified code

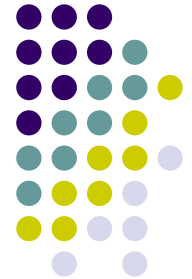
```
#include <stdio.h>
#include <sys/types.h>


int main()
{
pid_t p1=-1, p2=-1, p3=-1;

    printf("L0\n");
    p1 = fork();
    printf("L1, pid=%d,p1=%d,p2=%d, p3=%d\n",getpid(),p1,p2,p3);
    p2 = fork();
    printf("L2, pid=%d,p1=%d,p2=%d, p3=%d\n", getpid(),p1,p2,p3);
    p3 = fork();
    printf("L3, pid=%d,p1=%d,p2=%d, p3=%d\n", getpid(),p1,p2,p3);
}
```

```
L0
L1, pid=1688,p1=1689,p2=-1, p3=-1
L1, pid=1689,p1=0,p2=-1, p3=-1
L2, pid=1688,p1=1689,p2=1690, p3=-1
L2, pid=1690,p1=1689,p2=0, p3=-1
L2, pid=1689,p1=0,p2=1692, p3=-1
L3, pid=1688,p1=1689,p2=1690, p3=1691
L2, pid=1692,p1=0,p2=0, p3=-1
L3, pid=1689,p1=0,p2=1692, p3=1693
L3, pid=1694,p1=0,p2=0, p3=0
L3, pid=1691,p1=1689,p2=1690, p3=0
L3, pid=1692,p1=0,p2=0, p3=1694
L3, pid=1693,p1=0,p2=1692, p3=0
L3, pid=1690,p1=1689,p2=0, p3=1695
L3, pid=1695,p1=1689,p2=0, p3=0
```

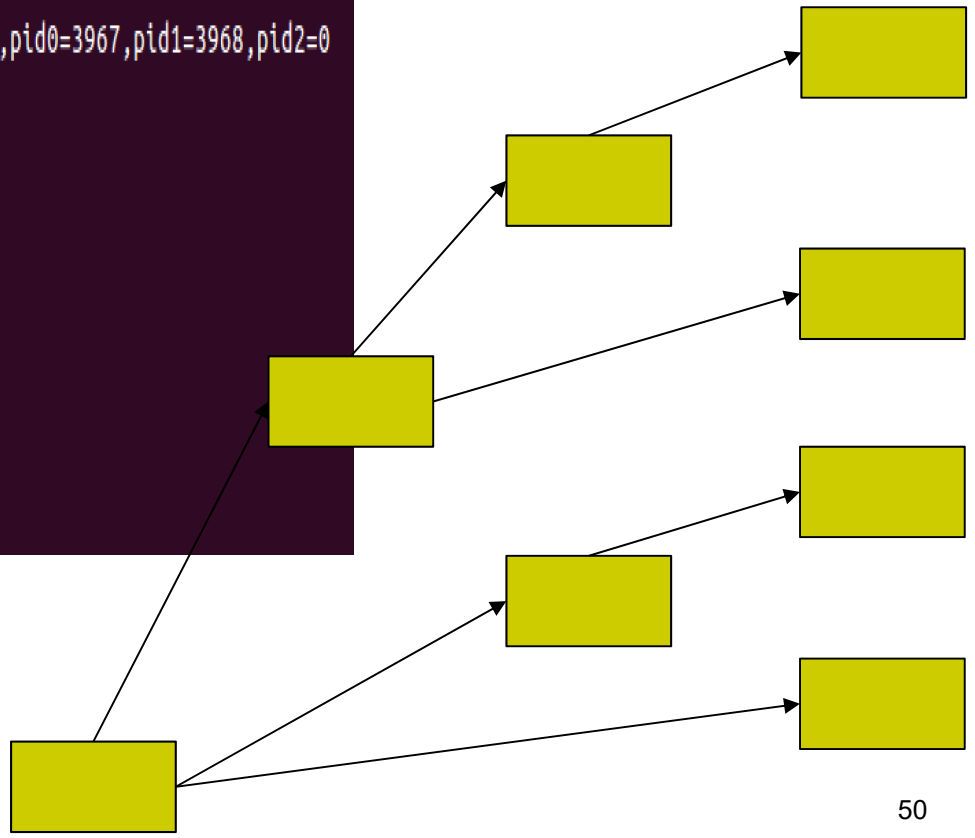
점검문제1 – fork3() 예제

```
void fork3()
{ int pid0=-1, pid1=-1, pid2=-1;
    printf("L0,pid=%d\n",getpid());
    pid0=fork();
    printf("L1,pid=%d,pid0=%d,pid1=%d,pid2=%d\n",getpid(),pid0,pid1,pid2);
    pid1=fork();
    printf("L2,pid=%d,pid0=%d,pid1=%d,pid2=%d\n",getpid(),pid0,pid1,pid2);
    pid2=fork();
    printf("Bye,pid=%d,pid0=%d,pid1=%d,pid2=%d\n",getpid(),pid0,pid1,pid2);
}
```

점검문제1: fork3() – 실행 예를 보고, 프로세스 계통도(완성하기)



```
sun@sun-virtualubuntu:~/Dropbox/assem/14-ecf-procs$ ./forks 3
L0,pid=3966
L1,pid=3966,pid0=3967,pid1=-1,pid2=-1
L2,pid=3966,pid0=3967,pid1=3968,pid2=-1
Bye,pid=3966,pid0=3967,pid1=3968,pid2=3969
sun@sun-virtualubuntu:~/Dropbox/assem/14-ecf-procs$ Bye,pid=3969,pid0=3967,pid1=3968,pid2=0
L2,pid=3968,pid0=3967,pid1=0,pid2=-1
Bye,pid=3968,pid0=3967,pid1=0,pid2=3970
L1,pid=3967,pid0=0,pid1=-1,pid2=-1
L2,pid=3967,pid0=0,pid1=3971,pid2=-1
Bye,pid=3967,pid0=0,pid1=3971,pid2=3972
Bye,pid=3972,pid0=0,pid1=3971,pid2=0
Bye,pid=3970,pid0=3967,pid1=0,pid2=0
L2,pid=3971,pid0=0,pid1=0,pid2=-1
Bye,pid=3971,pid0=0,pid1=0,pid2=3973
Bye,pid=3973,pid0=0,pid1=0,pid2=0
```



6. 프로세스 제어 3 FORK() - 프로세스그래프로 모델링하기

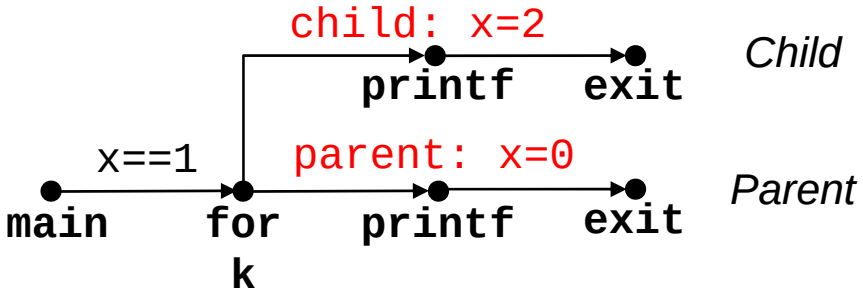
fork를 프로세스그래프로 모델링하기

- 프로세스 그래프는 concurrent 프로그램의 실행문들의 부분적인 순서관계를 표현하는데 유용한 도구 :
  - 노드는 문장의 실행
  - a -> b 는 a가 b보다 먼저 실행됨
  - 에지는 변수의 현재 값으로 레이블함
  - printf 노드는 출력으로 레이블함
  - 각 그래프는 입력 에지가 없는 노드로 시작
- 프로세스 그래프에 대한 위상적인 정렬(*topological sort*)*은 한가지 가능한 전체 순서화에 해당*
  - 노드의 전체 순서화에서 각 에지는 왼쪽에서 오른쪽으로

프로세스 그래프 예

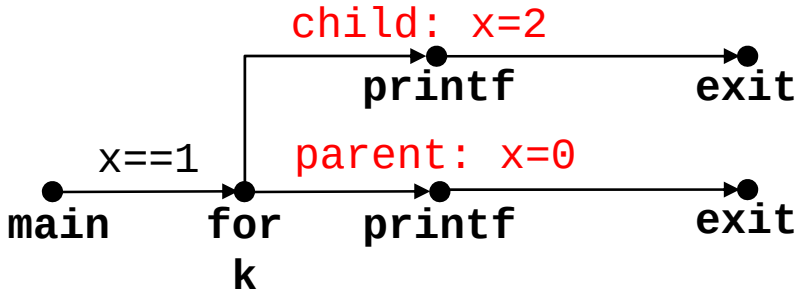
```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d₩n", ++x);
            exit(0);
    }

    /* Parent */
    printf("parent: x=%d₩n", --x);
    exit(0);
}
```

fork.c

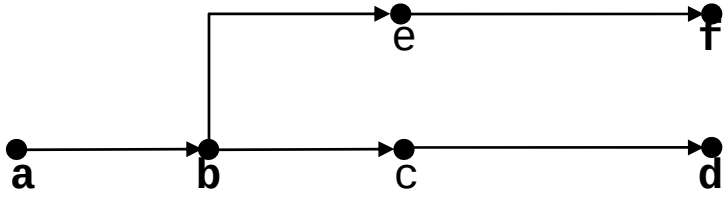


프로세스 그래프 해석

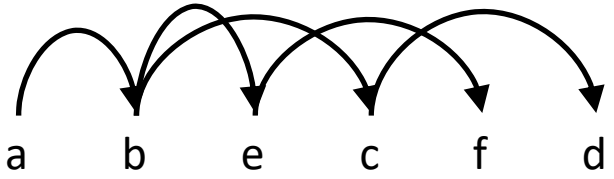
- 원래 그래프:



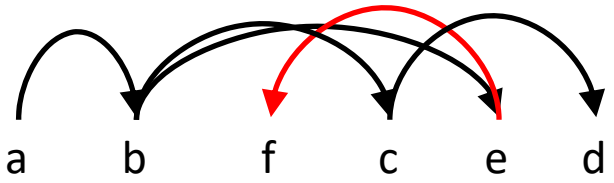
- 재 레이블한 그래프:



가능한 전체 순서화:

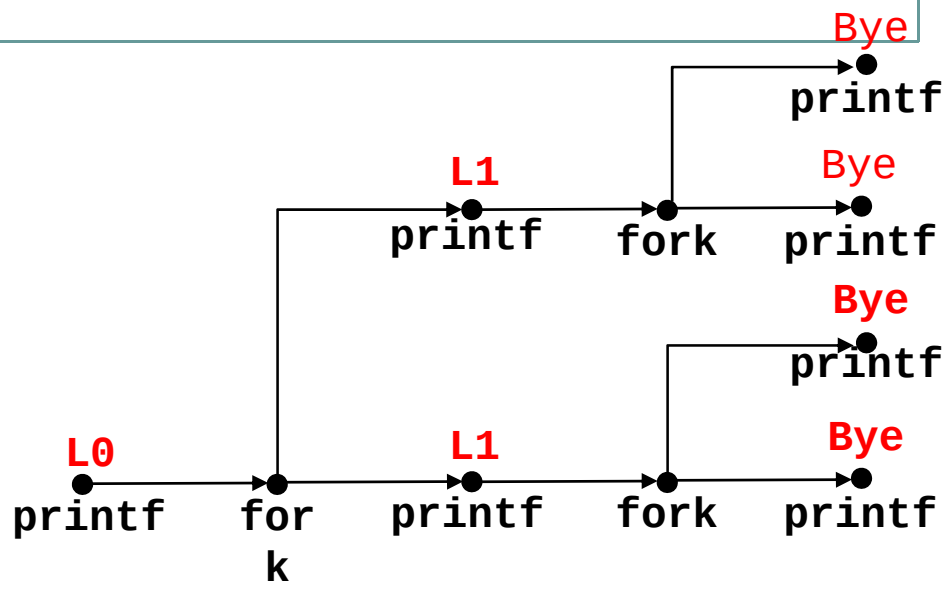


불가능한 순서화:



fork 예: 연속된 forks

```
void fork2()
{
    printf("L0₩n");
    fork();
    printf("L1₩n");
    fork();
    printf("Bye₩n");
}
```
forks.c



Feasible output:

L0
L1
Bye
Bye
L1
Bye
Bye

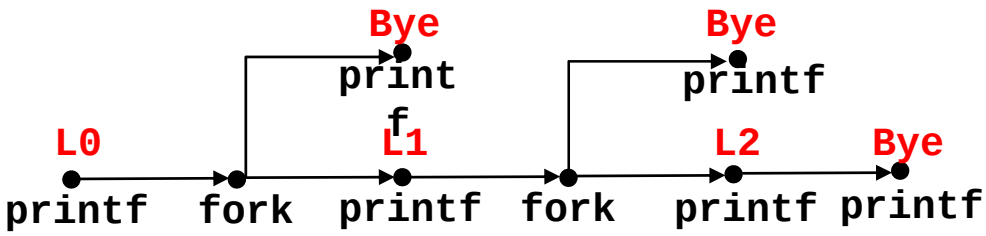
Infeasible output:

L0
Bye
L1
Bye
L1
Bye
Bye

fork 예: 중첩된 fork(부모에서)

```
void fork4()
{
    printf("L0₩n");
    if (fork() != 0) {
        printf("L1₩n");
        if (fork() != 0) {
            printf("L2₩n");
            }
    }
    printf("Bye₩n");
}
```

forks.c



Feasible output:

L0
L1
Bye
Bye
L2
Bye

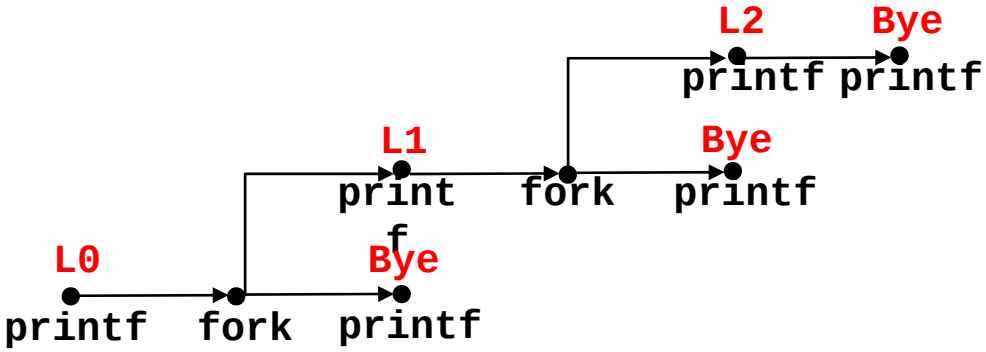
Infeasible output:

L0
Bye
L1
Bye
Bye
L2

fork 예: 중첩된 fork(자식에서)

```
void fork5()
{
    printf("L0₩n");
    if (fork() == 0) {
        printf("L1₩n");
        if (fork() == 0) {
            printf("L2₩n");
        }
    }
    printf("Bye₩n");
}
```

forks.c



Feasible output:

L0
Bye
L1
L2
Bye
Bye

Infeasible output:

L0
Bye
L1
Bye
Bye
L2

연습문제 2. fork

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

int main()
{
int x=1;

        if( fork() == 0)
                printf("pid:%d, printf1: x = %d\n",getpid(),++x);
        printf("pid:%d, printf2: x = %d\n", getpid(),--x);

}
```

```
pid:29729, printf2: x = 0
pid:29730, printf1: x = 2
pid:29730, printf2: x = 1
```

a) 위 프로그램에서 자식 프로세스의 출력을 쓰시오

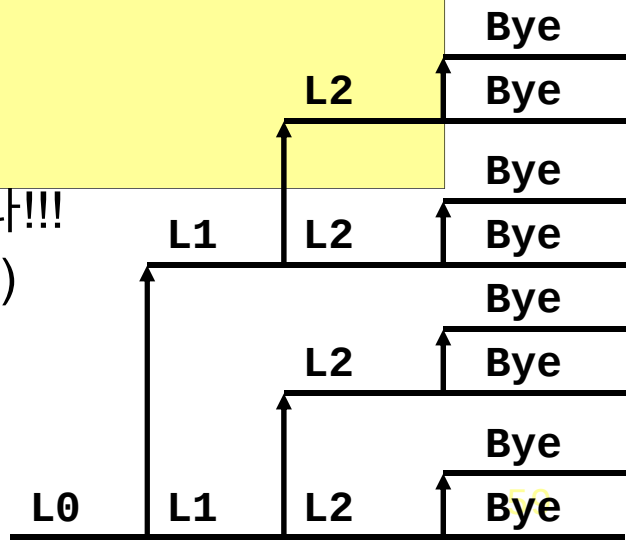
b) 위 프로그램에서 부모 프로세스의 출력을 쓰시오.

점검문제2: How to program for each child?

- 8 childs – to program for each child?

```
void fork3()
{ int pid0=-1, pid1=-1, pid2=-1;
    printf("L0,pid=%d₩n",getpid());
    pid0=fork();
    printf("L1,pid=%d,pid0=%d,pid1=%d,pid2=%d₩n",getpid(),pid0,pid1,pid2);
    pid1=fork();
    printf("L2,pid=%d,pid0=%d,pid1=%d,pid2=%d₩n",getpid(),pid0,pid1,pid2);
    pid2=fork();
    printf("Bye,pid=%d,pid0=%d,pid1=%d,pid2=%d₩n",getpid(),pid0,pid1,pid2);
    /* add here your code ..... */
    .  .  .
}
```

- **각 child가 별도의 일을 하도록** fork3.c를 수정하라!!!
- (서로 다른 일 하는 코드를 나누어서 맡길 수 있음)
- 예: 거대한 file의 1/8씩을 맡아서, 처리하는 일!!!
- pid0,pid1,pid2의 값만을 검사해서 함.
- pid값은 매번 달라짐.



요약

- 예외
  - 표준적이지 않은 제어 흐름을 필요로 하는 이벤트
  - 외부적으로 발생 (interrupts) 또는 내부적으로 발생 (traps, faults)
- 프로세스
  - 일정한 시점에 여러 활동중인 프로세스가 존재
  - 단일 코어에서는 한 순간에 한 프로세스만 수행
  - 단일 프로세스는 프로세서와 메모리를 모두 사용하는 환상을 갖게 됨

요약 (계속.)

- 프로세스 생성하기
  - `fork` 호출, 한번 호출 두번 리턴
- 프로세스 마침/종료
  - `exit` 호출
  - 한번 호출, 리턴 없음

실습과 다음 주 준비

- 실습: Binary Bomb2

- 다음주 강의 동영상: 프로세스 2
- 예습 질문, 점검문제 답은 개인과제로 올림.