시스템 프로그래밍 (2016)

프로세스와 스레드

목차

1. 프로세스 제어 – 좀비와 자식프로세스의 거두기(reaping)
2. 프로그램 로딩과 실행
3. tiny shell 과제 관련
4. 스레드와 프로세스
5. posix thread
6. 세마포와 상호배제

1. 프로세스 제어 – 좀비와 자식 프로세스의 거두기(REAPING)

자식프로세스의 수확(거두기)

- 개념
  - 프로세스 종료되어도, 시스템 자원을 사용함
    - 예: Exit 상태, 여러 OS 테이블들
  - 좀비라고 부름
    - 살아있는 송장, 반은 살고 반은 죽은 것
- 거두기
  - 부모가 종료된 자식에 대해 (`wait/waitpid`호출)
  - 부모에게는 exit 상태 정보가 주어짐
  - 커널은 좀비 자식 프로세스를 제거함
- 부모가 거두지 않으면?
  - 자식을 거두지 않고 부모가 종료하면, 고아가 된 자식은 init 프로세스(pid==1)에 의해 나중에 거두어질 것임.
  - 그래서, 오래동안 실행되는 프로세스들은 명시적인 거두기 (reaping)을 해야 함(쉡이나, 서버들...)

좀비 예

```c
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d₩n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d₩n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

forks.c

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- ps 명령으로 보면 자식 프로세스는 “defunct”로 나옴 (즉, 좀비)
- 부모프로세스를 kill하면 자식 프로세스도 수확

종료하지 않는 자식프로세스

```c
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d₩n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d₩n",
               getpid());
        exit(0);
    }
}
```

forks.c

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- 부모는 종료했지만 자식은 여전히 active
- 명시적으로 자식프로세스를 kill해야 함, 아니면 무한히 수행계속

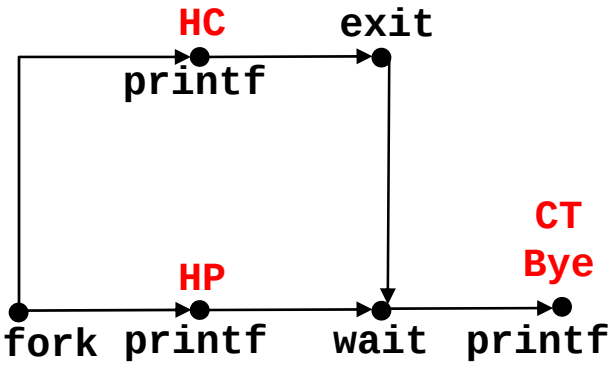
wait: 자식 프로세스와 동기화 하기

- `int wait(int *child_status)`
  - 현재 프로세스를 자신의 자식프로세스들 중의 하나가 종료될 때까지 정지시킨다
  - 리턴 값은 종료한 자식 프로세스의 pid 가 된다
    - 만일 `child_status != NULL`, 인 경우는, 자식 프로세스의 종료 이유를 나타내는 상태 정보를 갖는다.
    - 매크로로 체크(`wait.h`)
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
  - 그리고 나서, 커널은 자식 프로세스를 제거한다
  - 이와 같이 부모는 자식 프로세스를 명시적으로 삭제한다.

wait: 자식프로세스와 동기화

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child₩n");
            exit(0);
    } else {
        printf("HP: hello from parent₩n");
        wait(&child_status);
        printf("CT: child has terminated₩n");
    }
    printf("Bye₩n");
}
```

forks.c



Feasible output:

HC
HP
CT
Bye

Infeasible output:

HP
CT
Bye
HC

다른 wait 사용 예

- 여러 자식 프로세스가 있다면, wait로 거둬지는 순서는 임의로...
- 매크로 WIFEXITED 와 WEXITSTATUS로 exit 상태 정보 얻음

```
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d₩n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally₩n", wpid);
    }
}
```

forks.c

Wait() Example

- 만일 여러개의 자식들이 종료되었다면, 무순으로 자식을 제거한다
- exit 상태 정보를 얻기 위해서 WIFEXITED 와WEXITSTATUS 등의 매크로를 사용한다

```
sun@sun-virtualubuntu:~/Dropbox/assem/14-ecf-procs$ ./forks 10
Child 2051 terminated with exit status 106
Child 2052 terminated with exit status 107
Child 2053 terminated with exit status 108
Child 2050 terminated with exit status 105
Child 2054 terminated with exit status 109
Child 2049 terminated with exit status 104
Child 2048 terminated with exit status 103
Child 2047 terminated with exit status 102
Child 2046 terminated with exit status 101
Child 2045 terminated with exit status 100
sun@sun-virtualubuntu:~/Dropbox/assem/14-ecf-procs$
```

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

정상적인 종료(exit, 또는 리턴)라면 참을 리턴

정상종료라면 exit 상태정보를 리턴

Waitpid(): 특정 pid를 청소한다

- waitpid(pid, &status, options)
  - 특정 프로세스 아이디(pid>0)를 기다린다 (waitset이 특정 child)
  - 임의의 child를 기다림(pid == -1, waitset이 모든 child)
  - Options : 0(종료된 자식을 기다린다), WNOHANG(==1 한번만 체크), WUNTRACED(==2, 정지되거나 종료된 자식을 기다린다)

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status)) // normal termination
            printf("Child %d terminated with exit status %d\n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
```

Wait/Waitpid 실행예

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

Using waitpid (fork11)

```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminated with exit status 103
Child 3572 terminated with exit status 104
```

연습문제 1. waitpid

- 아래 프로그램의 가능한 출력을 모두 쓰시오.

```
int main()
{
      if(Fork() == 0)
            printf("a \n");
      } else {
          printf("b \n");
          waitpid(-1,NULL,0);
      }
      printf("c \n");
}
```

Ideally, B-C^1, A-C^2 조합가능하나
실제로BAC^2C^1

주) waitpid시에 pid=-1 이면 부모의 모든 자식프로세스를 기다린다??
기다리는 waitset이 모든 자식 프로세스,
기다리는 것은 그중 하나만 기다림.

점검문제 1. waitpid

- 다음 프로그램을 실행하면 몇 줄을 출력하게 되는가? 프로세스 그래프를 그려서 보이시오.

code/ecf/waitprob1.c

```
1   int main()
2   {
3       int status;
4       pid_t pid;
5
6       printf("Hello\n");
7       pid = Fork();
8       printf("%d\n", !pid);
9       if (pid != 0) {
10          if (waitpid(-1, &status, 0) > 0) {
11              if (WIFEXITED(status) != 0)
12                  printf("%d\n", WEXITSTATUS(status));
13          }
14      }
15      printf("Bye\n");
16      exit(2);
17  }
```

code/ecf/waitprob1.c

프로세스를 sleep 시키기

- `unsigned int sleep(unsigned int secs)`
  - 자기 자신을 secs 초 동안 suspend 시킨다
  - 정상적으로 깨어날 때는 0을 리턴하고, 그 외의 경우에는 잔여 시간을 초로 리턴한다
- `int pause(void)`
  - 호출하는 프로세스를 시그널(signal)을 받을 때까지 잠재운다

연습문제 3. sleep

- sleep 함수를 이용하여 다음과 같은 snooze( ) 함수를 완성하시오.

```
unsigned int snooze(unsigned int secs);
```

- 이 함수는 현재까지 잠을 잔 시간을 초로 다음과 같이 출력하는 것 외에는 sleep과 동일하게 동작한다

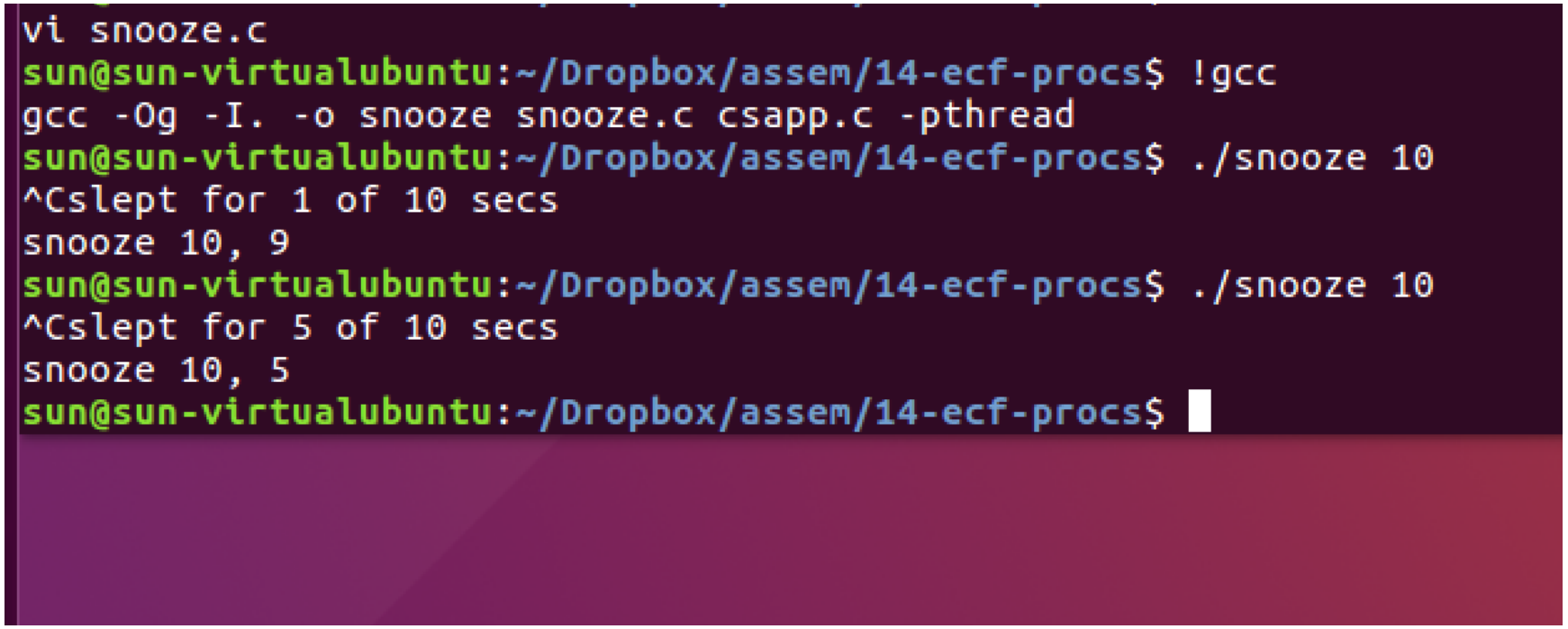
```
Slept for 4 of 5 secs.
```

Snooze()

```
unsigned int snooze(unsigned int secs)
{
        unsigned int rc = sleep(secs);
        printf("Slept for %u of %u secs.\n", secs - rc, secs);
        return rc;
}
```

```
#include "csapp.h"
unsigned int snooze(unsigned int secs) {
        unsigned int rc = sleep(secs);
        printf("slept for %u of %u secs\n", secs-rc, secs);
        return rc;
}
void handler(int sig) { return; }

int main(int argc, char *argv[])
{

        if(signal(SIGINT, handler) == SIG_ERR)
                unix_error("signal error\n");

        if(argc >= 2)
                printf("snooze %s, %u\n", argv[1], snooze(atoi(argv[1])) );
        return 0;

}
```

실제 구현시, signal 필요, 실행예



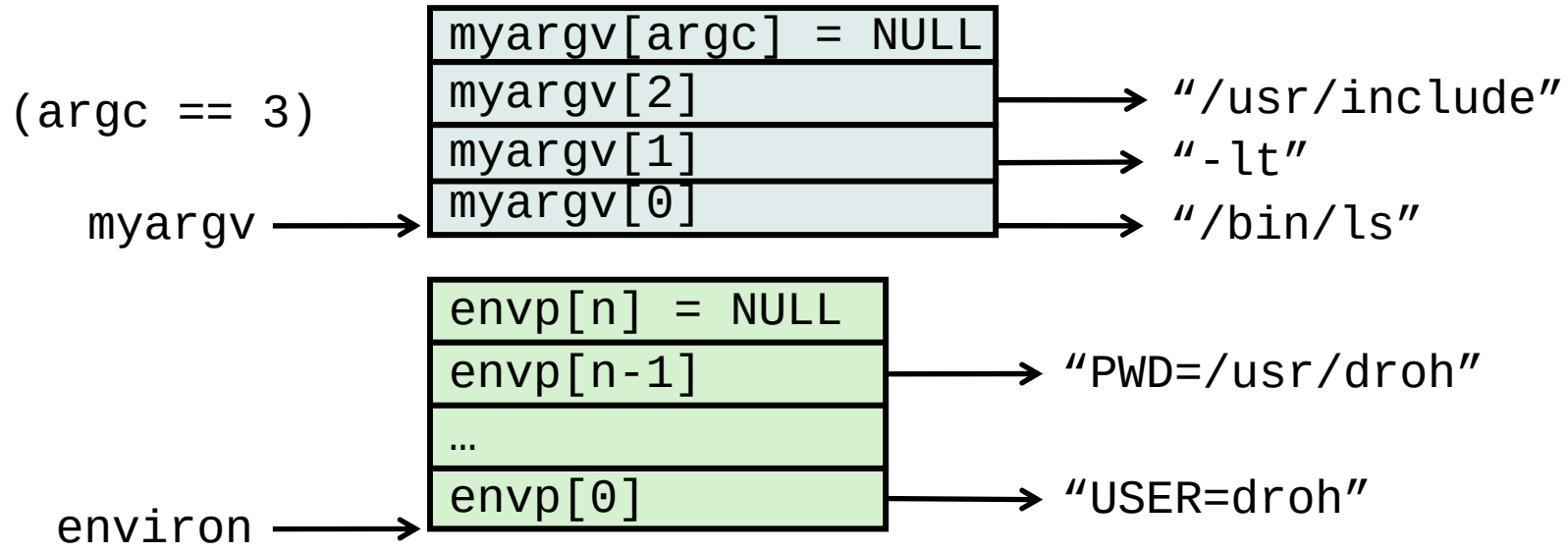
2. 프로그램 로딩과 실행

execve: 프로그램 로딩과 실행

- `int execve(char *filename, char *argv[], char *envp[])`
- 현재 프로세스 안에서 로드하고 실행:
  - 실행 파일이름 **`filename`**
    - 오브젝트 파일이나 스크립트 파일(`#!interpreter`) (e.g., `#!/bin/bash`)
  - …인자 리스트는 **`argv`**
    - 관습상 **`argv[0]==filename`**
  - …환경 변수 리스트는 **`envp`**
    - "name=value" 스트링 (예, `USER=droh`)
    - `getenv, putenv, printenv`
- 현재 프로세스의 코드, 데이터, 스택 영역을 덮어 쓰기 함
  - PID와 열린 파일들과 시그널 context만 유지
- 한번 호출되고 결코 리턴하지 않음
  - …오류가 있을 경우에는 예외적으로 리턴함.

execve 예

- 실행 “/bin/ls -lt /usr/include” 이 자식프로세스에서 현재 환경 하에서:



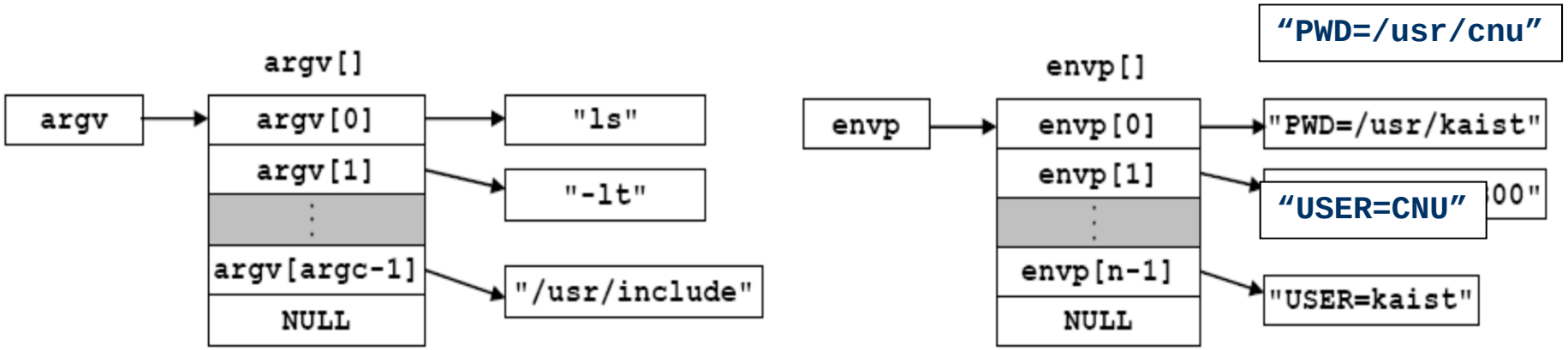

```
if ((pid = Fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.₩n", myargv[0]);
        exit(1);
    }
}
```

exec: 새 프로그램을 실행하기(1)

- `int execl(char *path, char *arg0, char *arg1, ..., 0)`
  - path에 위치한 프로그램을 로딩해서 args arg0, arg1, ...를 인자로 실행시킨다
    - path 는 실행프로그램의 완전한 경로를 가져야 한다
    - arg0 는 프로세스의 이름이다
      - 대개 arg0 는 path와 같거나, path 중에서 실행파일부분으로 표시한다
    - 실행프로그램의 실제 인자들은 arg1 부터 시작된다
    - 인자 리스트는 널 문자로 종료된다
  - 에러가 발생하면 -1을 리턴하고, 그 외의 경우에는 리턴하지 않는다 !

새 프로그램을 실행하기(2)

- int execve (char *path, char *argv[], char *envp[])
  - argv, envp: 널 문자로 끝나는 포인터 배열
  - envp는 환경 변수 스트링을 가리키는 char **environ 전역변수로부터 생성됨



새 프로그램을 실행하기(3)

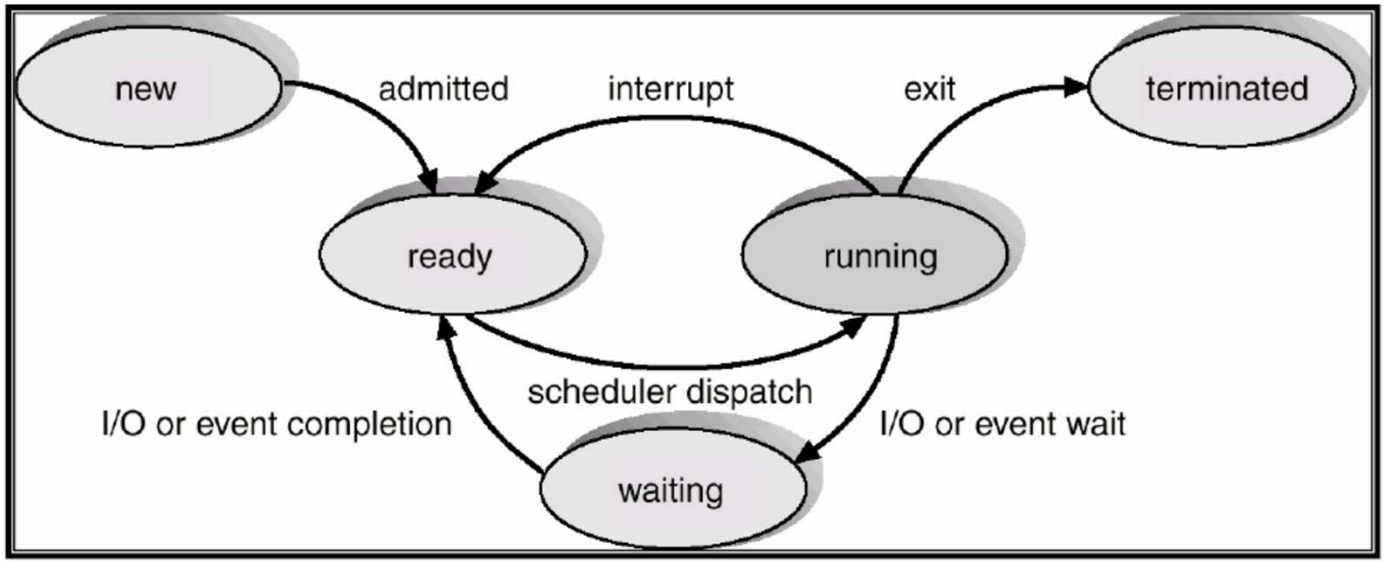
- Example : running /bin/ls

```
main() {
    if (fork() == 0) {
        execl("/bin/ls", "ls", "/", 0);
    }
    wait(NULL);
    printf("completed\n");
    exit();
}
```

```
main() {
    char *args[] = {"ls", "/", NULL};
    if (fork() == 0) {
        execv("/bin/ls", args);
    }
    wait(NULL);
}
```

세 개의 함수(로드와 실행): execl, execv, execve

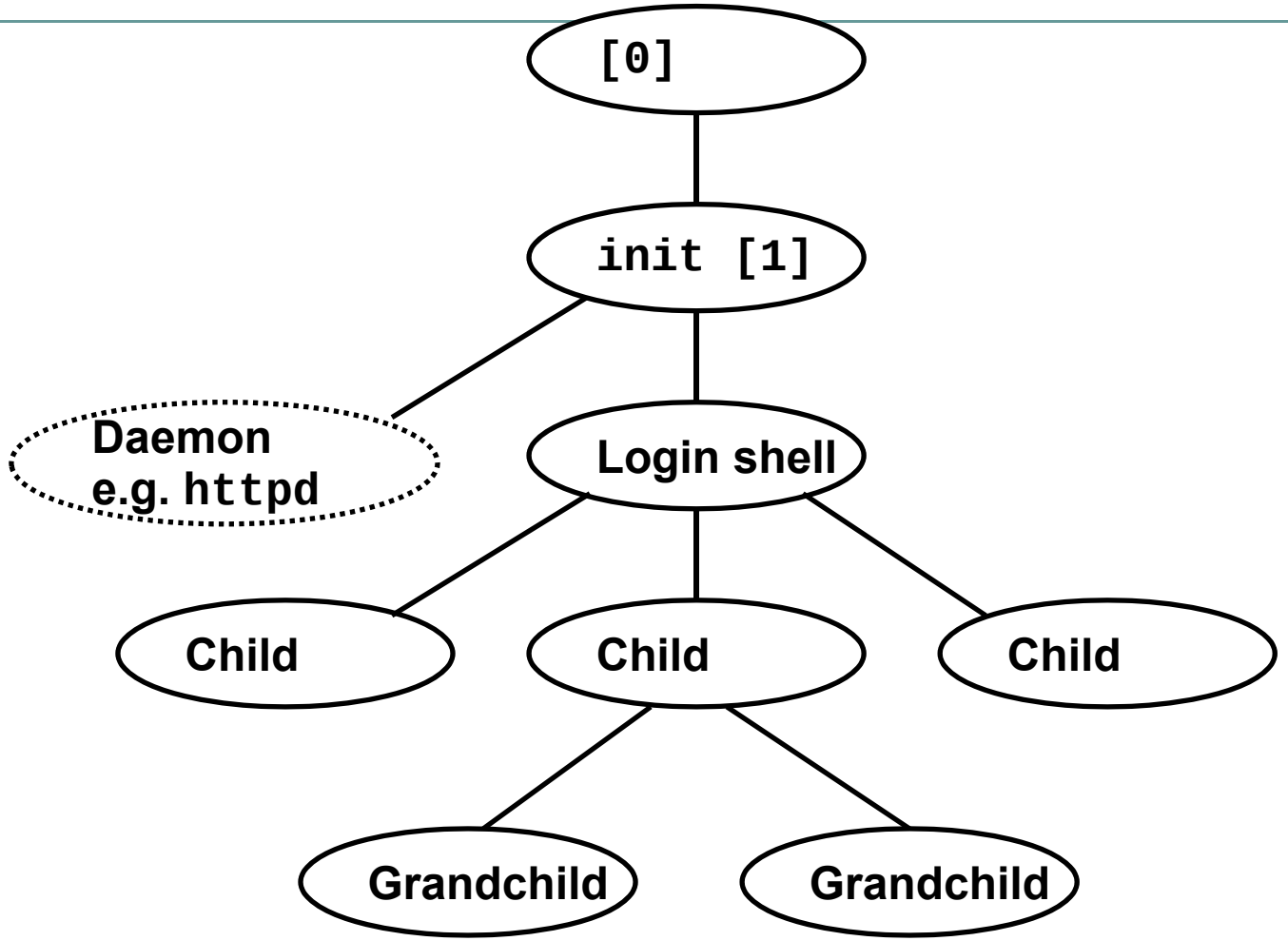
프로세스 상태



- 정지 또는 대기 **Stopped or waiting**
  - **SIGSTOP, SIGSTP, SIGTTIN or SIGTTOU** 과 같은 시그널을 받으면, 프로세스는 정지하며, **SIGCONT** 시그널을 받으면, 다시 실행된다
  - 시그널은 일종의 소프트웨어 인터럽트다
- 종료 **Terminated**
  - 종료 시그널을 수신했을 때
  - 메인 함수에서 리턴했을 때
  - **exit** 함수를 호출했을 때

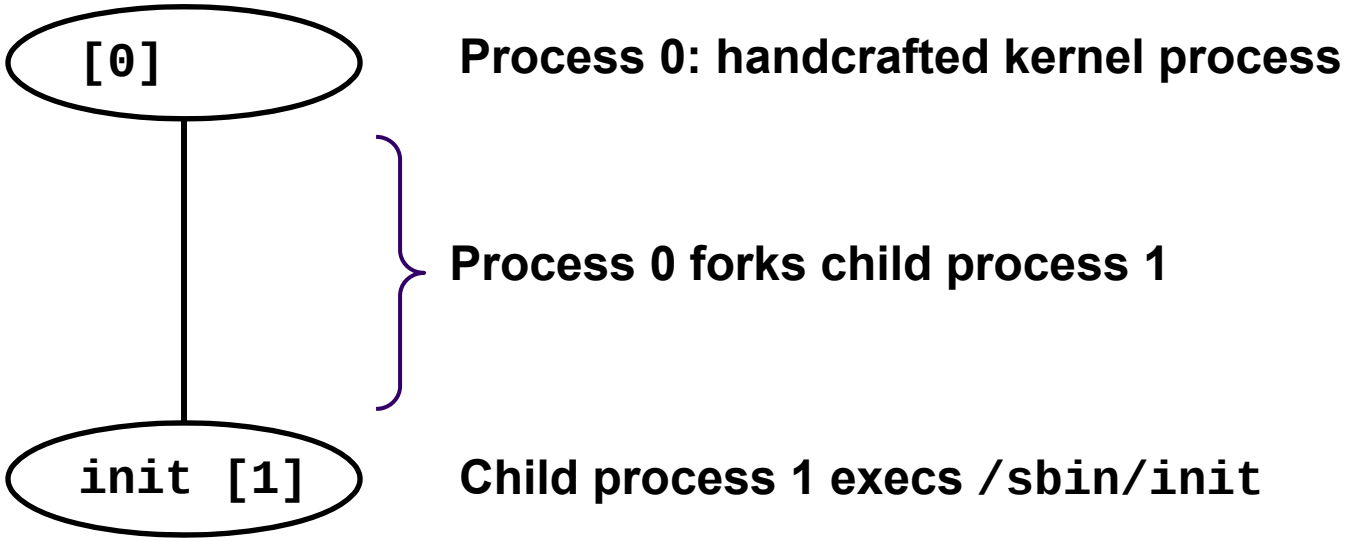
3.B. UNIX/LINUX 프로세스 체계

fork와 exec의 사용예 : Unix 프로세스 체계

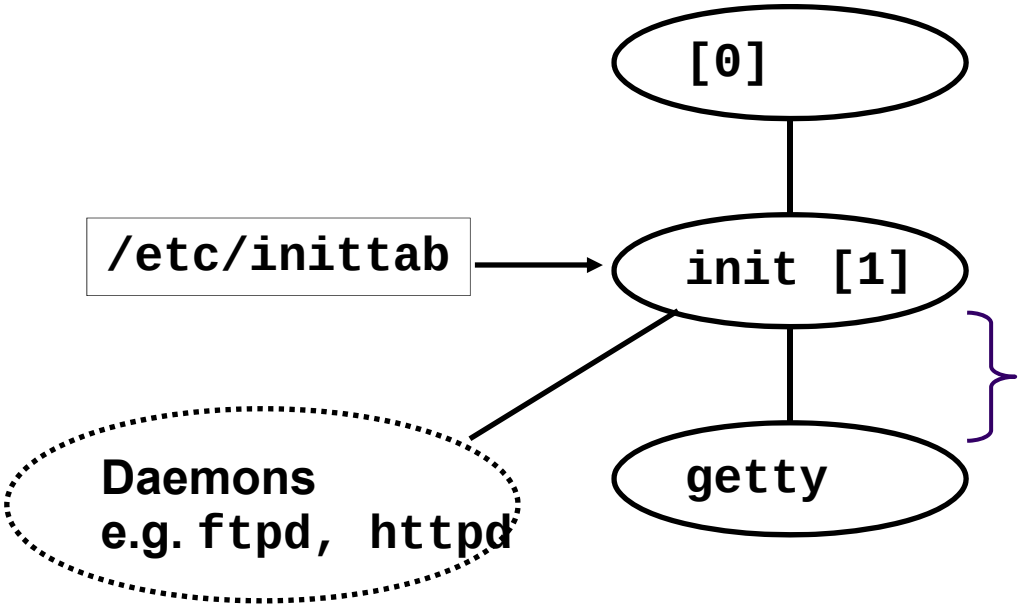


Unix 초기화: Step 1

1. Pushing reset button loads the PC with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel binary (e.g., `/boot/vmlinux`)
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for process 0.

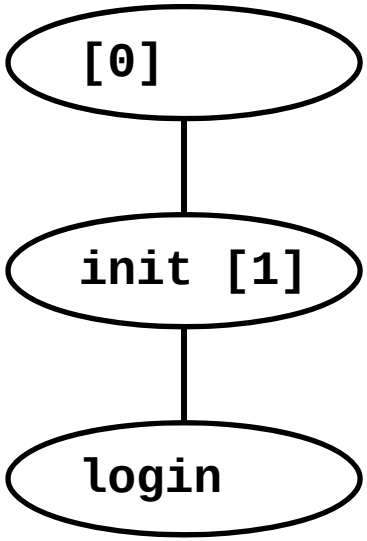


Unix 초기화: Step 2



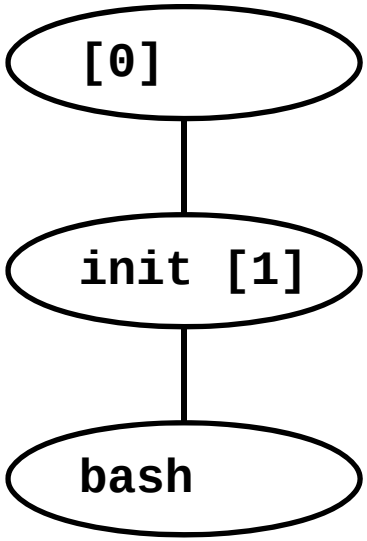
`init` forks and execs daemons per `/etc/inittab`, and forks and execs a `getty` program for the console

Unix 초기화: Step 3



The `getty` process execs a `login` program

Unix 초기화: Step 4

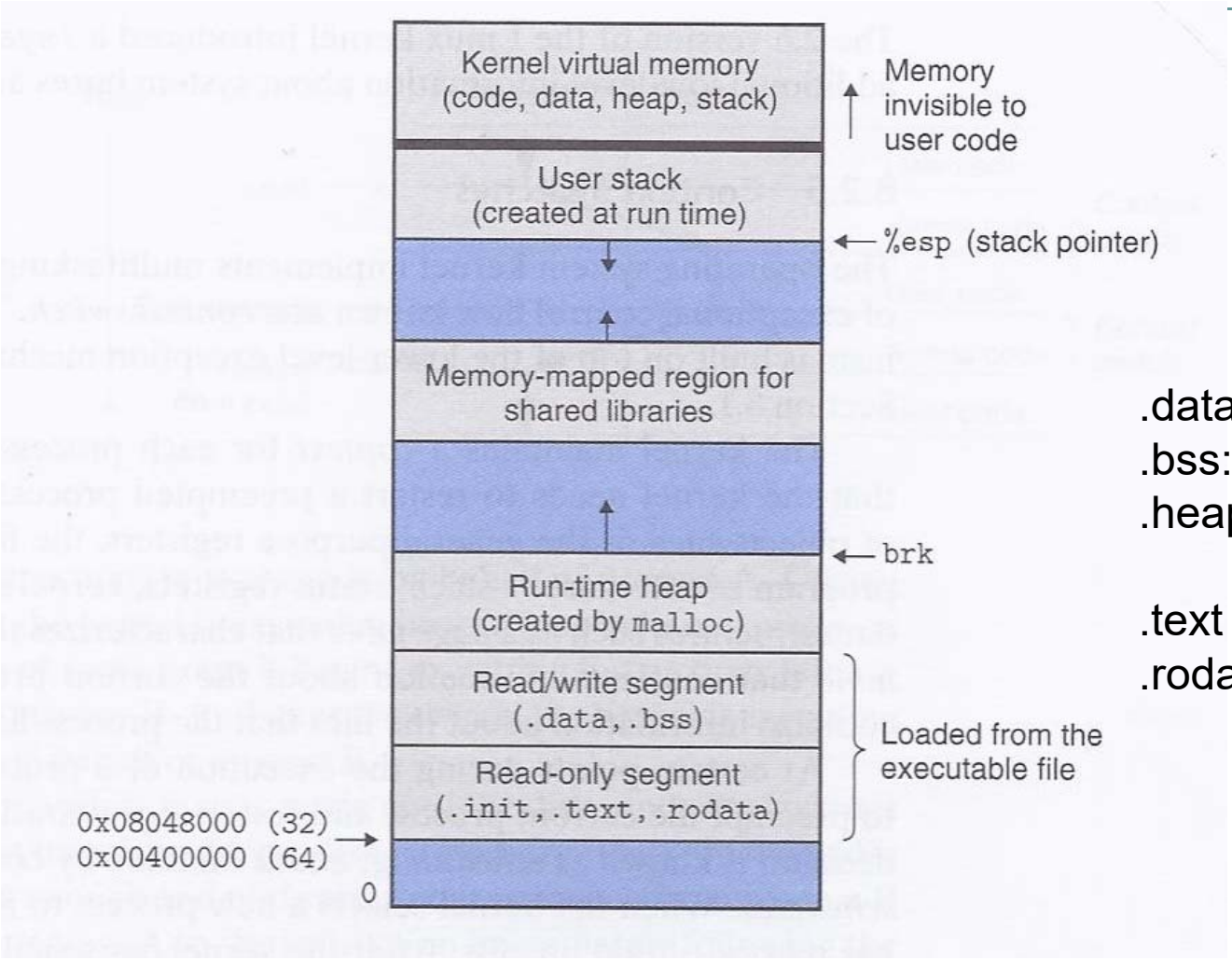


`login` reads login-ID and passwd.
if OK, it execs a *shell.*
if not OK, it execs another `getty`

In case of **`login`** on the console
`xinit` may be used instead of
a shell to start the window manger

4. THREAD와 PROCESS

Private address space(process)



.data : 초기화된 global, static
.bss: 초기화안된 global,static
.heap: malloc(), free()로 관리
되는 공간
.text : code
.rodata : const 공간

User and kernel modes

- Kernel mode : 모든 명령 수행, 모든 주소 접근
- User mode : privileged instruction 수행 불가, mode bit 변화 불가, I/O 시작 불가, 커널 영역/다른 user 영역 접근 불가 → protection fault
- User -> kernel mode 변환: exception 통해서만 가능함. Exception handler 수행시에 mode bit 변경, 커널모드 수행
- Kernel -> user mode (appl. 코드로 return시)
- User mode accessible kernel information
  - /proc/cpuinfo: cpu 관련 정보들 …
  - /proc/<pid>/* maps: 메모리 세그먼트 정보(pid별)

Context (switch)

- A context : 커널이 교체된 process 재시작에 필요한 상태 정보들
  - 레지스터 – 일반용, 부동소숫점용, PC(**eip**), 사용자스택, 상태 레지스터, 커널 스택, 여러가지 커널 자료구조(예: page table – 주소공간 사용관련, process table, file table,등)
- Scheduling – process간 preemption(교체) 결정 및 실행, 이전 process와 새 process에 대해서 일정 작업수정
- Context switch :
  - (exception handler(timer, page fault, abort 등) 수행이 주 원인)
  1. 현재 process의 context 저장,
  2. 이전 preempted process의 context 복원
  3. 새 process에 control을 넘김

프로세스에 대한 전통적인 관점

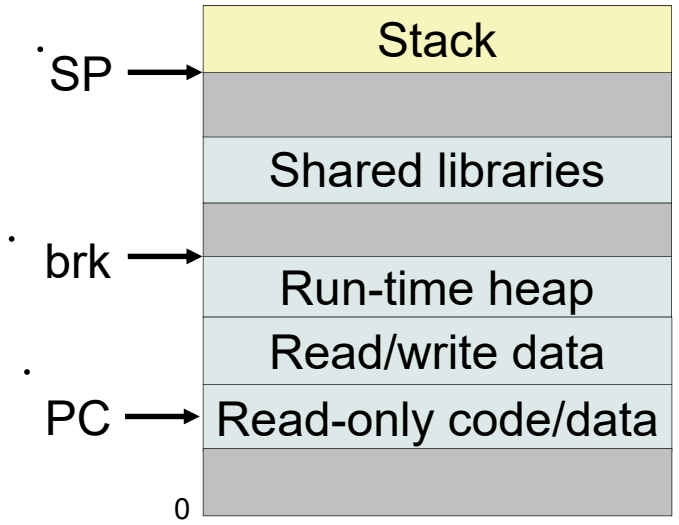
- 프로세스는 = 프로세스 context+code, data, stack

프로세스 context

프로그램컨텍스트:
- 데이터레지스터
- 컨디션 코드
- 스택 포인터 (SP)
- 프로그램 카운터 (PC)

커널 컨텍스트:
- VM 구조
- 디스크립터 테이블
- brk 포인터(heap)

Code, data, and stack

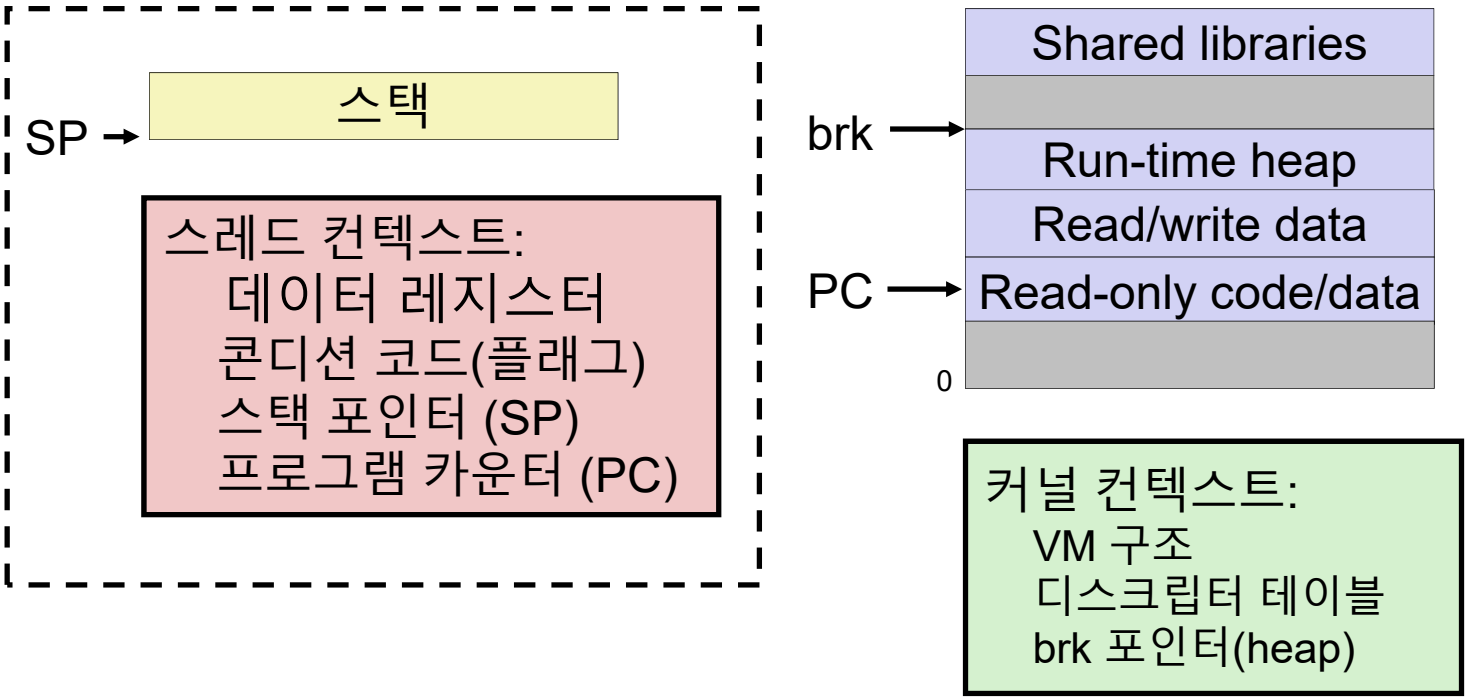


프로세스에 대한 다른 관점

- 프로세스= 스레드 + 코드, 데이터, 커널 컨텍스트

스레드 (주 스레드)

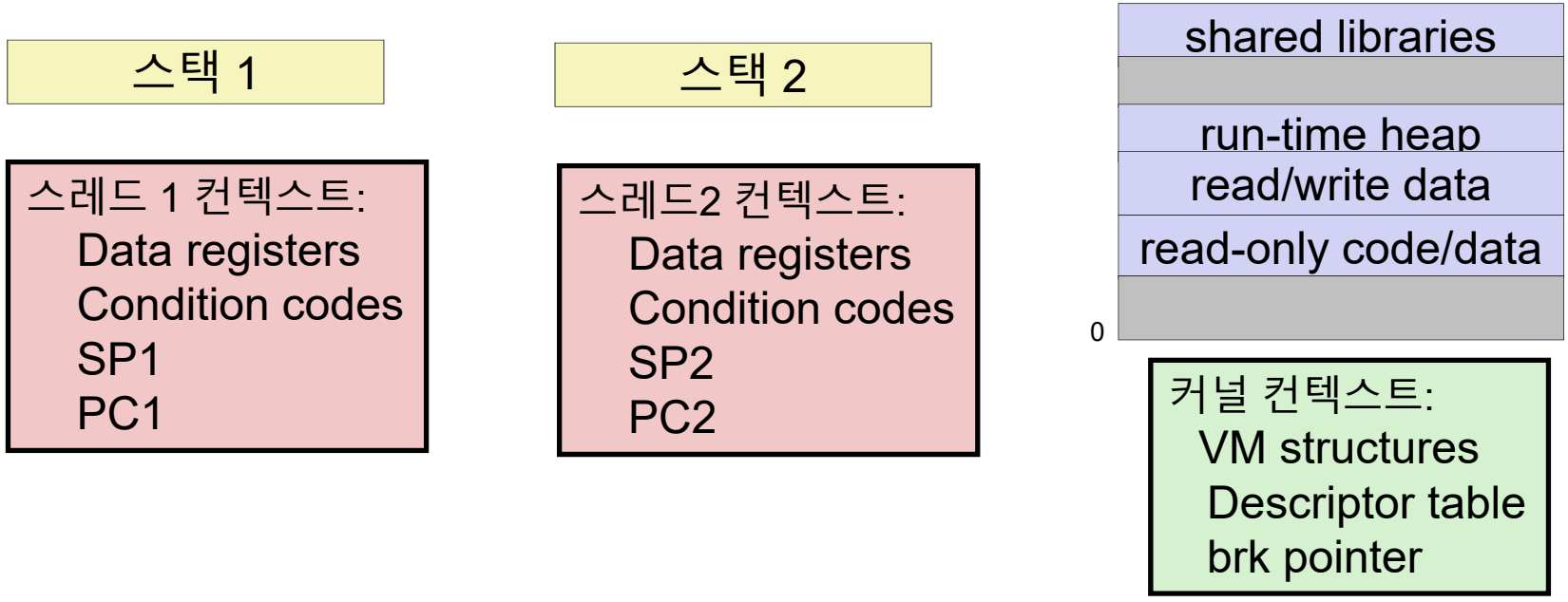
코드, 데이터, 커널 컨텍스트



다중 스레드를 갖는 프로세스

- 여러 스레드가 한 프로세스에 포함되는 경우
  - 각 스레드는 각자의 논리적인 제어 흐름을 가짐
  - 각 스레드는 같은 코드, 데이터, 커널 컨텍스트를 공유함
  - 각 스레드는 지역 변수에 대한 각자의 스택을 가짐.
    - 그러나 다른 스레드로부터 보호되지는 않음ㅁ
  - 각 스레드는 각자의 스레드 아이디(TID)를 가짐

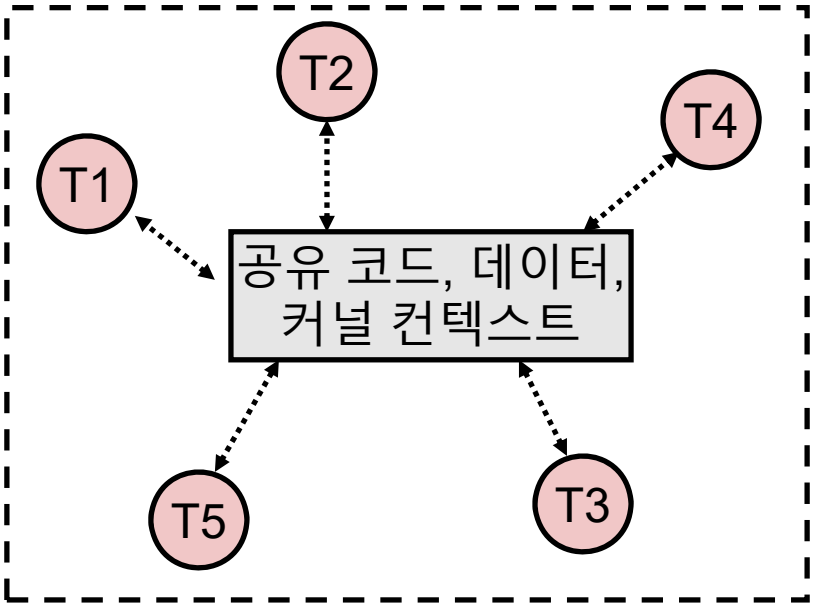
스레드1 (주 스레드) 스레드 2 (동료 스레드 공유되는 코드와 데이터



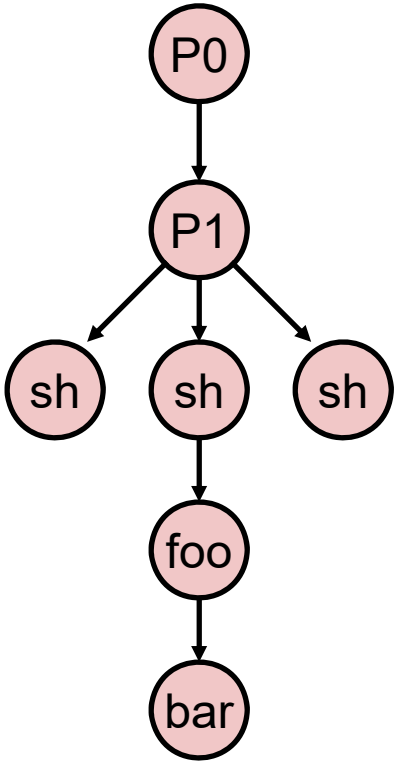
스레드에 대한 논리적인 관점

- 프로세스에 연관된 스레드는 동료들의 풀을 형성
  - 프로세스(트리 계층형성)와 다름

프로세스 foo와 연관된 스레드들



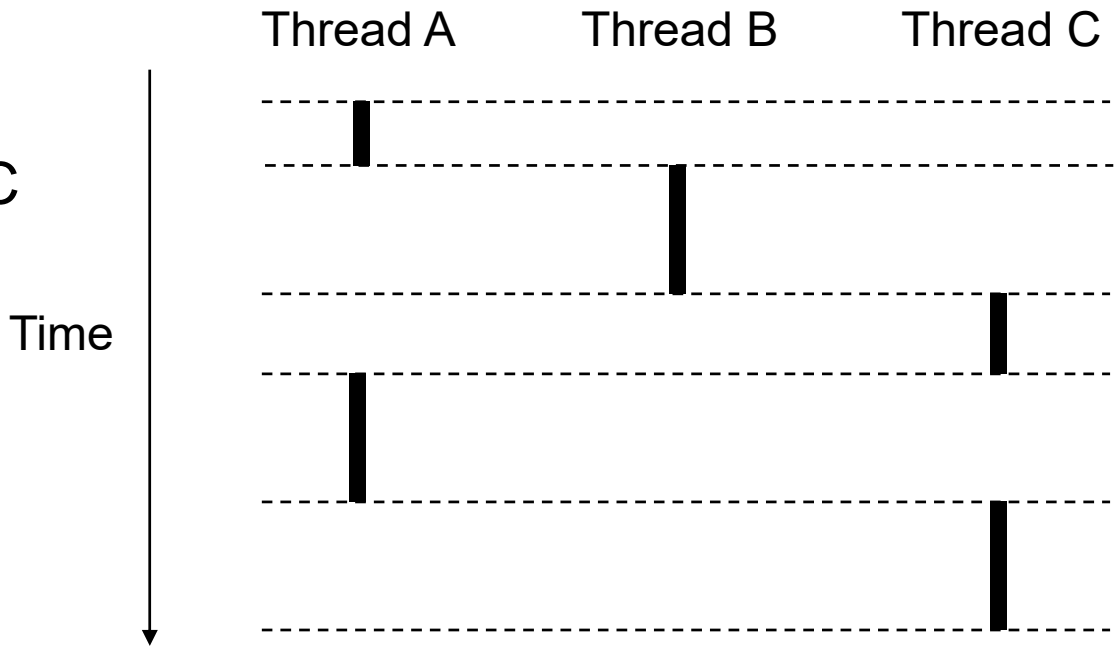
프로세스 계층 구조



동시성 스레드들

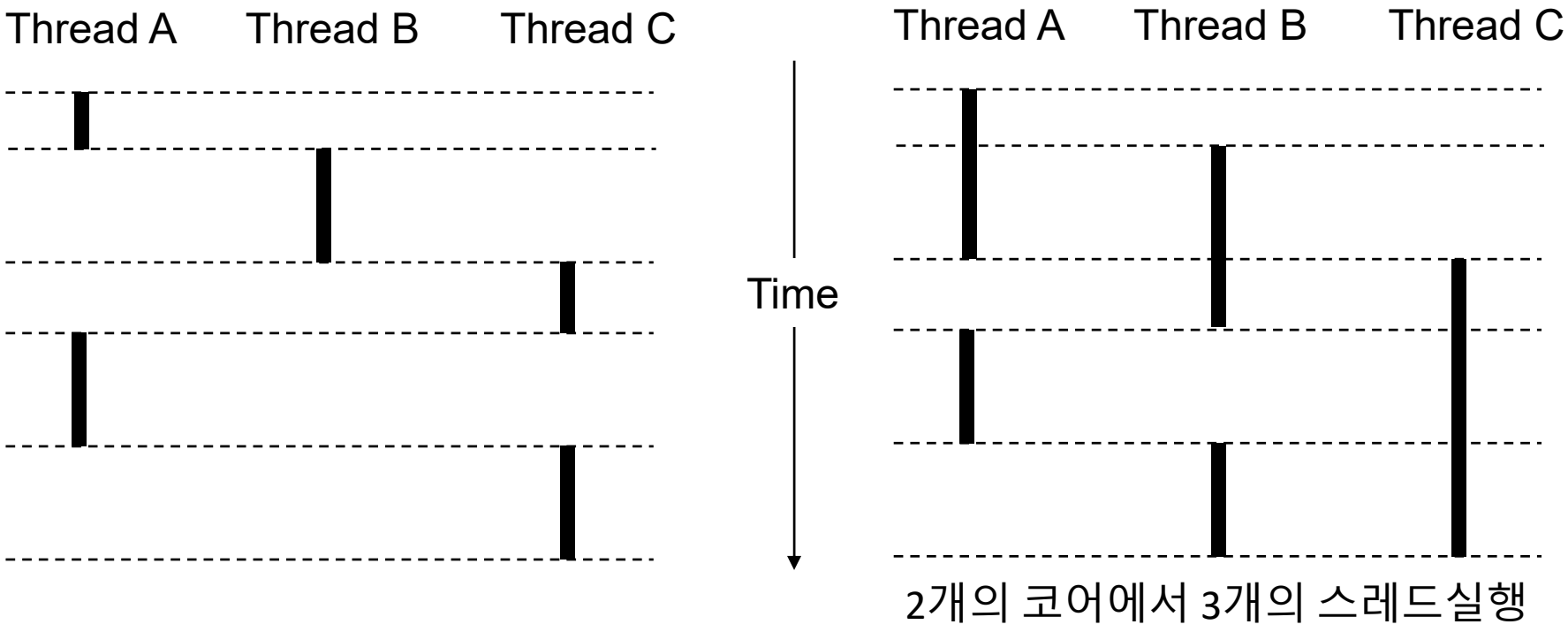
- 두 스레드의 시간 상의 흐름에서 겹칠 때, 동시성 스레드라고 함
- 그이외에는, 순차적이라고 함

- 예:
  - 동시성: A & B, A&C
  - 순차적: B & C



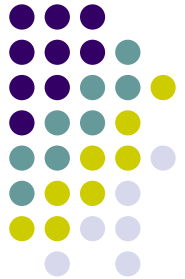
동시성 스레드 실행

- 단일 코어 프로세서
  - 시간나누기(time slicing)으로 병렬성을 시뮬레이션함
- 다중 코어 프로세서
  - 진정한 병렬성을 실현함



스레드와 프로세스

- 스레드와 프로세스의 유사성
  - 각자의 논리적인 제어 흐름을 가짐
  - 각자는 다른 것들와 동시적으로 실행될 수있음(대개는 서로 다른 코어에서)
  - 각각은 컨텍스트 전환(문맥전환)됨
- 스레드와 프로세스의 상이성
  - 스레드들은 모두 데이터와 코드를 공유함 (로컬 스택은 제외)
    - 프로세스는 공유하지 않음
  - 스레드는 프로세스에 비해 비용이 많이 들지 않는다/비싸지 않다.
    - 프로세스 제어(만들고,수확하기)에는 스레드 제어보다 두배로 비용이 비쌈(시간이나 메모리/디스크 측면에서)
    - 리눅스에서 수자로 보면:
      - ~20K 싸이클 걸려서 프로세스를 만들고 수확함
      - ~10K 싸이클(또는 그 이하)) 걸려서 스레드 만들고 수확함

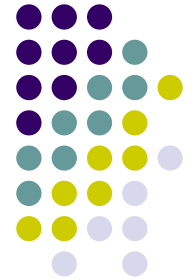


5. POSIX THREAD

Posix 스레드(Pthreads) 인터페이스

- *Pthreads:* 스레드 조작용 표준 인터페이스(60개정도의 함수를 포함함, C 언어용)
  - 만들고 수확하기
    - `pthread_create()`
    - `pthread_join()`
  - 스레스 ID 결정
    - `pthread_self()`
  - 스레드 종료하기
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [모든 스레드 종료] ; `RET` [현재 스레드 종료]
  - 공유 변수에 대한 동기화된 접근
    - `pthread_mutex_init`
    - `pthread_mutex_lock`
    - `pthread_mutex_unlock`

Pthreads "hello, world" 프로그램



```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

hello.c

스레드 ID

스레드 특성 (대개 NULL)

스레드 루틴(함수)

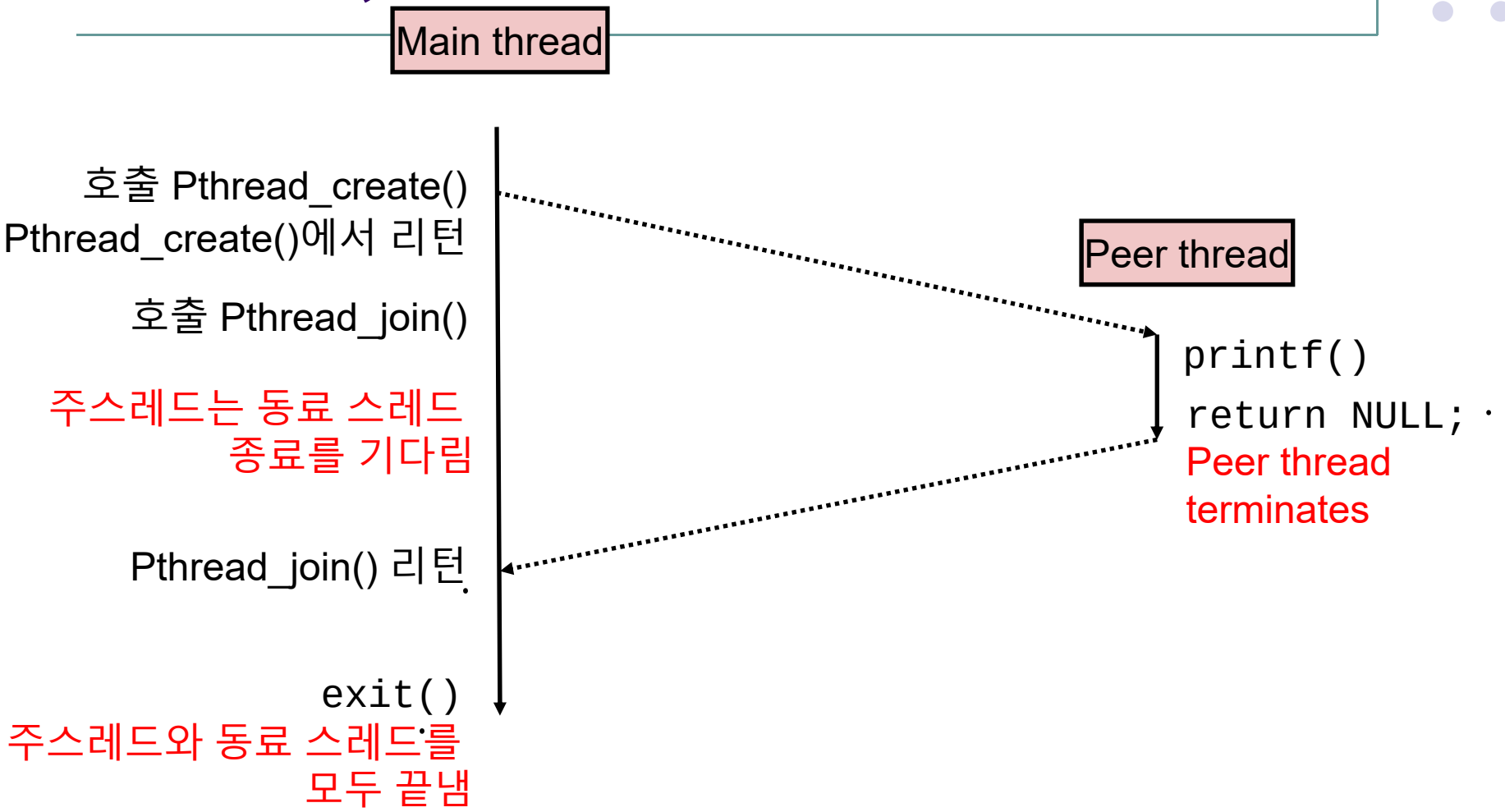
스레드인자 (void *p)

리턴 값 (void **p)

```c
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!₩n");
    return NULL;
}
```

hello.c

스레드 버전의 실행 “hello, world”



스레드 메모리 모델

- 개념적인 모델:
  - 다중 스레드는 단일 프로세스의 컨텍스트 내에서 실행됨
  - 각 스레드는 별도의 스레드 컨텍스르를 가짐
    - 스레드 ID, 스택, 스택포인터, PC, 컨디션코드, GP 레지스터
  - 모든 스레드는 그 외 프로세스 컨텍스트는 공유함
    - 코드, 데이터, 힙, 공유 라이브러리(프로세스의 가상주소 공간안에있는)
- 실제적으로는, 이 모델이 그대로 적용안됨:
  - 레지스터 값들은 그대로 분리되고 보호되지만…
  - 모든 스레드는 다른 스레드의 스택을 읽고 쓸 수 있음

이런 불일치로 혼란과 오류가 야기됨

공유의 사례 - 프로그램

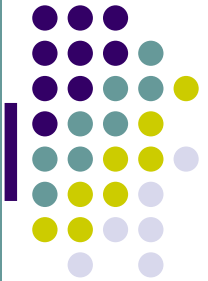
```c
char **ptr;  /* global var */

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)₩n",
        myid, ptr[myid], ++cnt);
    return NULL;
}
```

동료 스레드들은 주 스레드의 스택을 전역 변수의 포인터를 통해 간접적으로 접근한다.

변수인스턴스를 메모리에 매핑하기



- 전역변수
  - *정의: 함수 밖에 정의된 변수*
  - **가상 메모리는 각 전역 변수의 한 인스턴스만 유지**
- 지역 변수
  - *정의* : static이 붙지 않은 함수내 선언된 변수
  - **각 스레드는 각 지역 변수에 대한 별도의 인스턴스 가짐**
- 지역 static 변수
  - *정의:static 이 붙은 함수내 선언된 변수*
  - **가상 메모리는 임의의 지역 static변수의 한 인스턴스만 유지**

변수 인스턴스를 메모리에 매핑하기

전역변수: 1 인스턴스 (ptr [데이터)

지역변수: 1 인스턴스 (i.m, msgs.m)

```
char **ptr;  /* global var */

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

지역변수: 2 인스턴스 (
myid.p0 [동료스레드0의 스택에],
myid.p1 [동료스레드1의 스택에]
)

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)₩n",
        myid, ptr[myid], ++cnt);
    return NULL;
}
```

지역 static 변수: 1 인스턴스 (cnt [데이터])

badcnt.c: 부적절한 동기화의 예

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld₩n", cnt);
    else
        printf("OK cnt=%ld₩n", cnt);
    exit(0);
}
```

badcnt.c

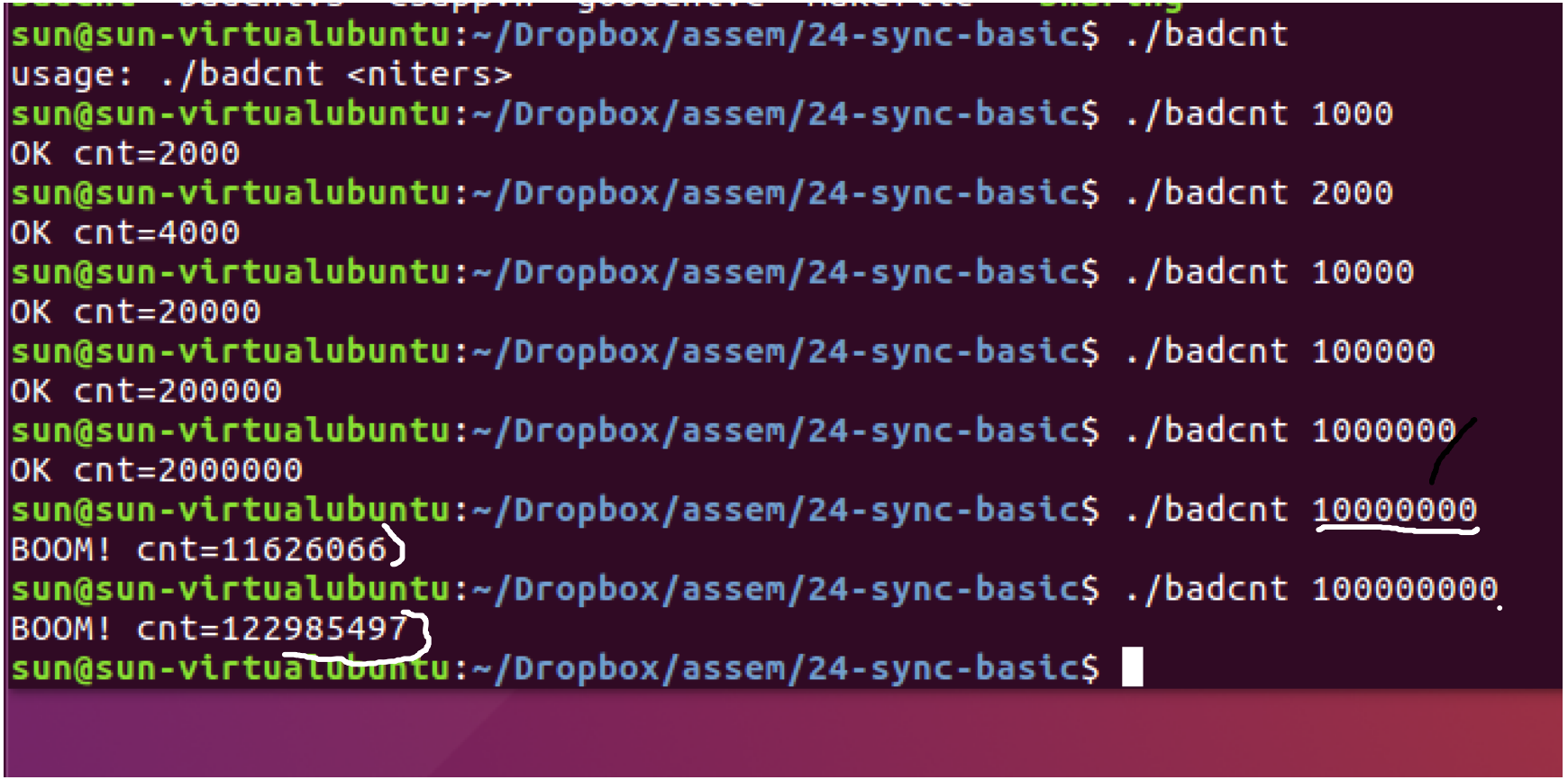
```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
            *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

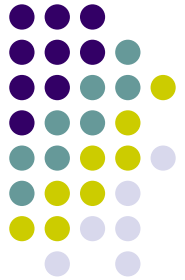
```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

`cnt` should equal 20,000.

What went wrong?

count가 커지면...


sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./badcnt
usage: ./badcnt <niters>
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./badcnt 1000
OK cnt=2000
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./badcnt 2000
OK cnt=4000
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./badcnt 10000
OK cnt=20000
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./badcnt 100000
OK cnt=200000
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./badcnt 1000000
OK cnt=2000000
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./badcnt 10000000
BOOM! cnt=11626066
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./badcnt 100000000
BOOM! cnt=122985497
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$



6. 세마포와 상호배제

세마포

- ***세마포:*** 음수가 아닌 전역 정수. P,V 동작으로
- P(s)
  - s 가 0 아니면, s를 1 감소시키고 바로 리턴함.
    - 테스트(비교)와 감소 동작이 원소동작(나누어질수없는)으로 실행
  - s가 0이면, s가 0 아닌 값이 될 때까지 대기함. 다른 스레드의V 동작으로 다시 시작하게 됨.
  - 다시 시작한 후에, P 동작은 s를 감소시키고, caller로 리턴함
- ***V(s):***
  - s를 1 증가시킴
    - Increment 동작은 원소 동작으로 실행
  - s가 0아닌 값이 되기를 기다라는 P 동작으로 블록된 스레드가 있으면, 그 스레드 중 하나가 다시 시작되면서, P 동작을 마침으로써 s를 감소 시킴
- 세마포의 변치않는 조건: *(s >= 0)*

C 세마포 동작

Pthreads 함수:

```
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */
```

CS:APP 래퍼 함수:

```
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

badcnt.c: 부적절한 동기화

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld₩n", cnt);
    else
        printf("OK cnt=%ld₩n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
            *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

세마포로 어떻게 해결할 수 있을까?

세마포를 이용한 상호배제 (mutual exclusion)

- 기본 개념
  - 각 공유 변수(또는 관련 변수들)에 대해서 유일한 세마포 mutex(처음에 1인 값)를 연결시킨다.
  - 해당 critical section(변수 변경 부분)을 *P(mutex)* 와 *V(mutex)* 동작으로 감싼다.
- 용어:
  - *이진 세마포*: 세마포의 값이 0이나 1임
  - *Mutex:* 상호배제에 사용되는 이진 세마포
    - P 동작: "locking" the mutex, 잠금동작
    - V 동작: "unlocking" or "releasing" the mutex, 잠금해제동작
    - *유지 "Holding"* a mutex: 잠겨있고 아직 풀리지 않은 상태.

goodcnt.c: 적절한 동기화

- 공유변수에 대한 mutex의 정의와 초기화

```
volatile long cnt = 0;  /* Counter */
sem_t mutex;            /* Semaphore that protects cnt */

Sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- **감싸기**critical section 을 *P와 V로 감싸기*:

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
```

goodcnt.c

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

경고: I매우 느려짐(badcnt.c비하여).

goodcnt.c:

```
#include "csapp.h"

void *thread(void *vargp); /* Thread routine prototype */
volatile long cnt = 0; /* Counter */
sem_t mutex;           /* Semaphore that protects counter */

int main(int argc, char **argv)
{    int niters;
    pthread_t tid1, tid2;

    if (argc != 2) {  printf("usage: %s <niters>₩n", argv[0]);
                exit(0);     }
    niters = atoi(argv[1]);

    /* Create threads and wait for them to finish */
/* $begin goodcntseminit */
    Sem_init(&mutex, 0, 1);  /* mutex = 1 */
/* $end goodcntseminit */
    Pthread_create(&tid1, NULL, thread, &niters);
    Pthread_create(&tid2, NULL, thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
                printf("BOOM! cnt=%ld₩n", cnt);
    else
                printf("OK cnt=%ld₩n", cnt);
    exit(0);
}
```

```
/* Thread routine */
void *thread(void *vargp)
{
    int i, niters = *((int *)vargp);

/* $begin goodcntthread */
    for (i = 0; i < niters; i++) {
            P(&mutex);
            cnt++;
            V(&mutex);
    }
/* $end goodcntthread */
    return NULL;
}
```

실행결과와 badcnt와 시간 비교

```
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./goodcnt 1000
OK cnt=2000
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./goodcnt 1000000
OK cnt=2000000
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./goodcnt 10000000
OK cnt=20000000
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ ./goodcnt 100000000
OK cnt=200000000
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ time ./goodcnt 1000000
OK cnt=2000000

real    0m0.100s
user    0m0.056s
sys     0m0.000s
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$ time ./badcnt 1000000
BOOM! cnt=1000000

real    0m0.006s
user    0m0.004s
sys     0m0.000s
sun@sun-virtualubuntu:~/Dropbox/assem/24-sync-basic$
```

mutex 버전

```
void *thread();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;
int main(int argc, char **argv)
{   int niters;
    pthread_t tid1, tid2;
    niters = atoi(argv[1]);
/* Create threads and wait for them to finish */
   pthread_create(&tid1, NULL, thread, &niters);
    pthread_create(&tid2, NULL, thread, &niters);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /* Check result */
    if (counter != (2 * niters))
            printf("BOOM! cnt=%ld\n", counter);
    else
            printf("OK cnt=%ld\n", counter);
    exit(0);
}
```

```
/* Thread routine */
void *thread(void *vargp)
{
    int i, niters = *((int *)vargp);

/* $begin goodcntthread */
    for (i = 0; i < niters; i++) {
        pthread_mutex_lock( &mutex1 );
            counter++;
            pthread_mutex_unlock( &mutex1 );
    }
/* $end goodcntthread */
    return NULL;
}
```

점검문제2(O,X 문제)

1. thread 루틴에 정의된 지역 변수는 동료스레드간에 공유된다.
2. 주 스레드는 thread 루틴에 정의된 지역 static 변수에 접근할 수 없다.
3. 스레드 컨텍스트에는 스택포인터는 포함되나, PC는 포함되지 않는다.
4. 스레드 비용이 프로세스 비용에 비하여 낮지 않다.
5. 스레드를 이용하면, 프로세스의 경우보다 2배이상 빠르게 실행도 가능하다.
6. 세마포가 필요한 이유는 공유 변수에 대한 Load, Update, Store 동작을 원소동작으로 확보하기 위한 것이다.
7. 세마포의 V동작과 pthread_mutex_lock() 동작은 같은 기능을 한다.