

Pintos

Hanyang University
Embedded Software Systems Lab.



Contents

0. Introduction
1. Command Line Parsing
2. System Call
3. Hierarchical Process Structure
4. File Descriptor
5. Denying Write to Executable
6. Alarm System Call
7. Priority Scheduling
8. Priority Scheduling and Synchronization
9. Priority Inversion Problem
10. Multi-Level Feedback Queue Scheduler
11. Virtual Memory
12. Memory Mapped File
13. Swapping
14. Stack
15. Buffer Cache
16. Extensible File
17. Subdirectory



Project 상관관계

User Program

1. Command Line Parsing
2. System Call
3. Hierarchical Process Structure
4. File Description
5. Denying Write to Executable

Thread

6. Alarm System Call
7. Priority Scheduling
8. Priority Scheduling and Synchronization
9. Priority Inversion Problem
10. Multi-Level Feedback Queue Scheduler

Virtual Memory

11. Virtual Memory
12. Memory Mapped File
13. Swapping
14. Stack

Filesystem

15. Buffer cache
16. Extensible File
17. Subdirectory

Introduction

▣ 핀토스가 뭐지?

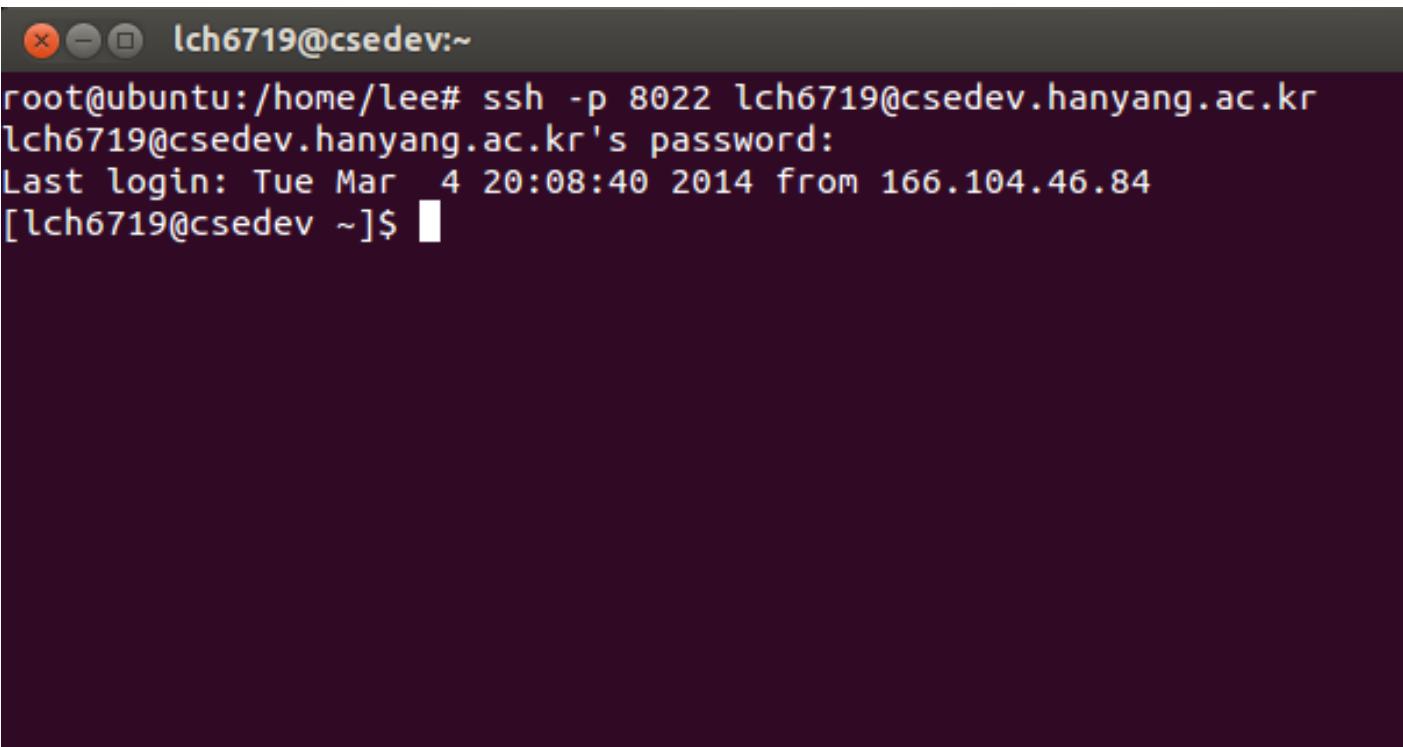
- ◆ x86 아키텍처를 위한 교육용 운영체제
- ◆ 2004년 스탠포드 대학에 Ben Pfaff에 의해 만들어짐
- ◆ kernel threads, loading and running user programs, file system 등을 지원
- ◆ Bochs나 QEMU등 x86 시뮬레이터를 사용

▣ 왜 핀토스를 사용할까?

- ◆ 운영체제에서 실제로 각종 개념(쓰레드, 프로세스, 메모리 관리, 파일 시스템)을 구현해보는 것이 매우 중요
- ◆ 리눅스와 같은 상용운영체제는 매우 큼(100만라인). 80%이상이 다양한 하드웨어지원을 위한 디바이스 드라이버코드
- ◆ 리눅스 컴파일 : 최소한 한시간 이상걸림.
- ◆ 간단하고, 이해하기 쉽고, 컴파일도 쉬운 운영체제 핀토스(PintOS)

Pintos 실습 서버 접속 방법

- ▣ SSH 설치 (\$ sudo apt-get install ssh)
- ▣ 서버 접속 (\$ ssh -p 8022 사용자계정@csedev.hanyang.ac.kr)



A screenshot of a terminal window titled "lch6719@csedev:~". The window shows an SSH session from a root user on an Ubuntu system to a host at port 8022. The command entered was "ssh -p 8022 lch6719@csedev.hanyang.ac.kr". The password was requested, and the last login information was displayed. The prompt "[lch6719@csedev ~]\$" is visible at the end of the session.

```
lch6719@csedev:~  
root@ubuntu:/home/lee# ssh -p 8022 lch6719@csedev.hanyang.ac.kr  
lch6719@csedev.hanyang.ac.kr's password:  
Last login: Tue Mar  4 20:08:40 2014 from 166.104.46.84  
[lch6719@csedev ~]$
```

Pintos 설치 및 환경 설정

- ▣ 해당 명령어를 통해 핀토스 소스코드 다운로드

```
$ wget http://dmclab.hanyang.ac.kr/wikidata/download/pintos.tar.gz
```

- ▣ 압축 해제

```
$ tar xvf pintos.tar.gz
```

- ▣ pintos/src/threads 디렉토리로 이동 후 make 명령어 입력



Pintos 설치 확인

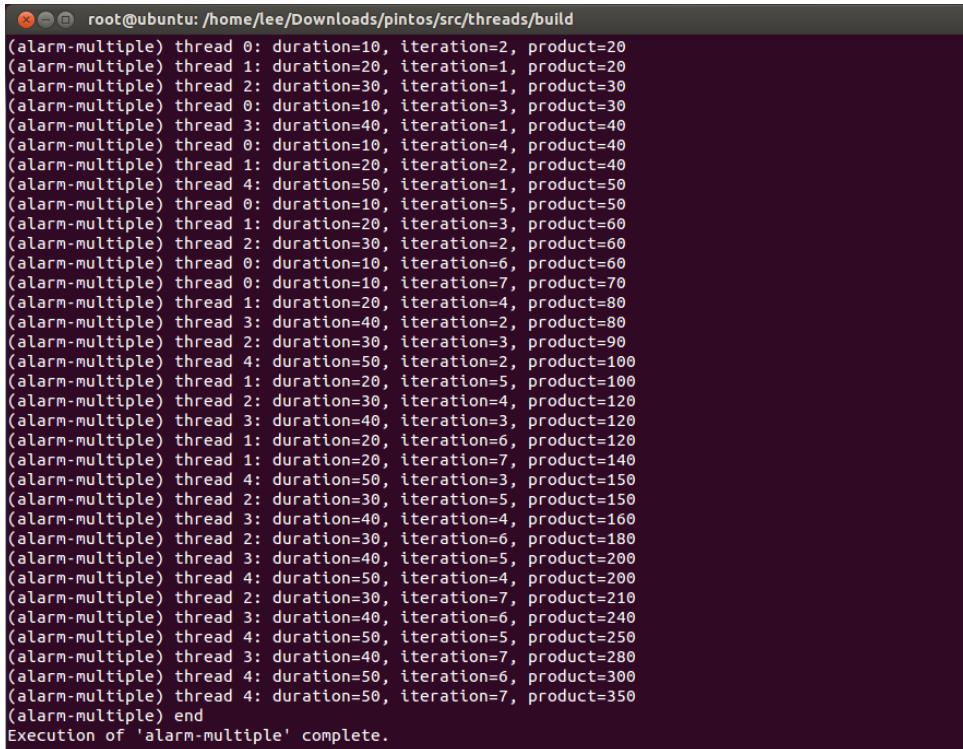
- ▣ pintos/src/threads/ 디렉토리로 이동

```
$ make
```

```
$ cd build
```

```
$ pintos -- run alarm-multiple
```

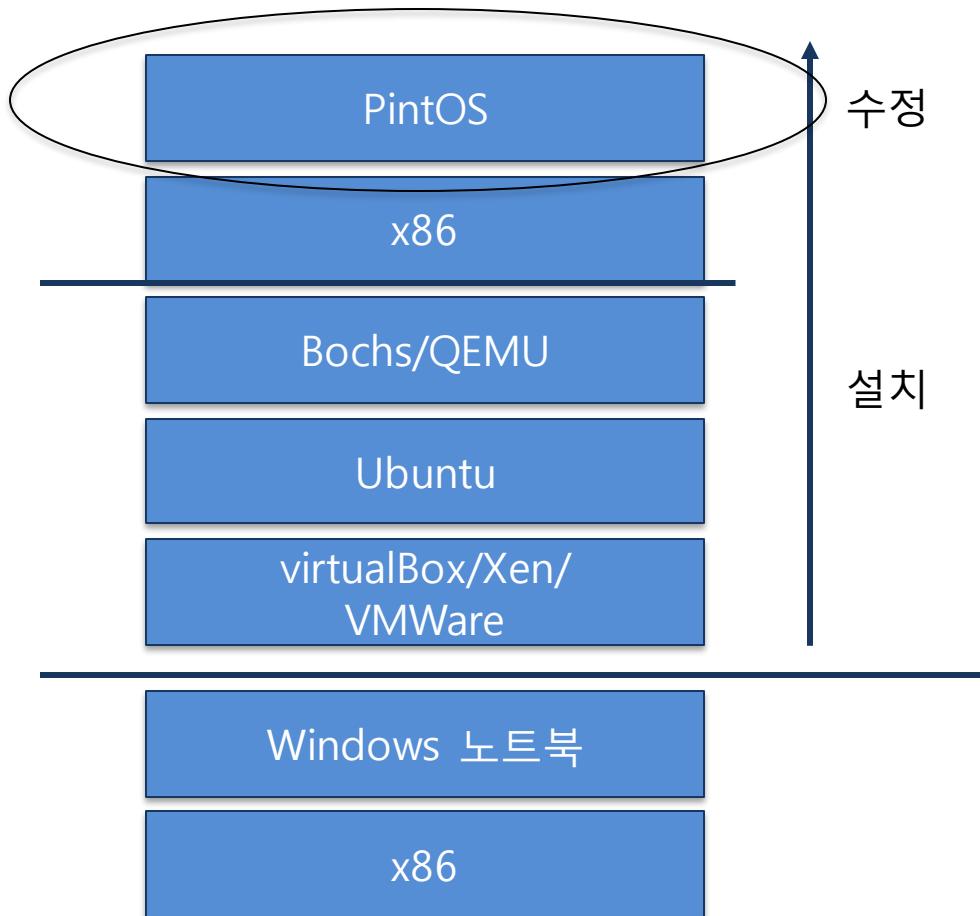
- ◆ 프로그램 동작 확인



The screenshot shows a terminal window titled "root@ubuntu: /home/lee/Downloads/pintos/src/threads/build". The window displays the output of the "alarm-multiple" program. The output consists of multiple lines of text, each representing a thread's progress. Each line contains the thread identifier (0, 1, 2, 3, or 4), duration, iteration count, and product value. The threads are running simultaneously, with their progress interleaved. The final message at the bottom indicates the completion of the program.

```
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 2: duration=30, iteration=1, product=30
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
(alarm-multiple) thread 3: duration=40, iteration=1, product=40
(alarm-multiple) thread 0: duration=10, iteration=4, product=40
(alarm-multiple) thread 1: duration=20, iteration=2, product=40
(alarm-multiple) thread 4: duration=50, iteration=1, product=50
(alarm-multiple) thread 0: duration=10, iteration=5, product=50
(alarm-multiple) thread 1: duration=20, iteration=3, product=60
(alarm-multiple) thread 2: duration=30, iteration=2, product=60
(alarm-multiple) thread 0: duration=10, iteration=6, product=60
(alarm-multiple) thread 0: duration=10, iteration=7, product=70
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
(alarm-multiple) thread 3: duration=40, iteration=3, product=120
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 4: duration=50, iteration=3, product=150
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
```

핀토스의 실행



개인 PC에서 Pintos 사용하기 1: Virtual Box 설치

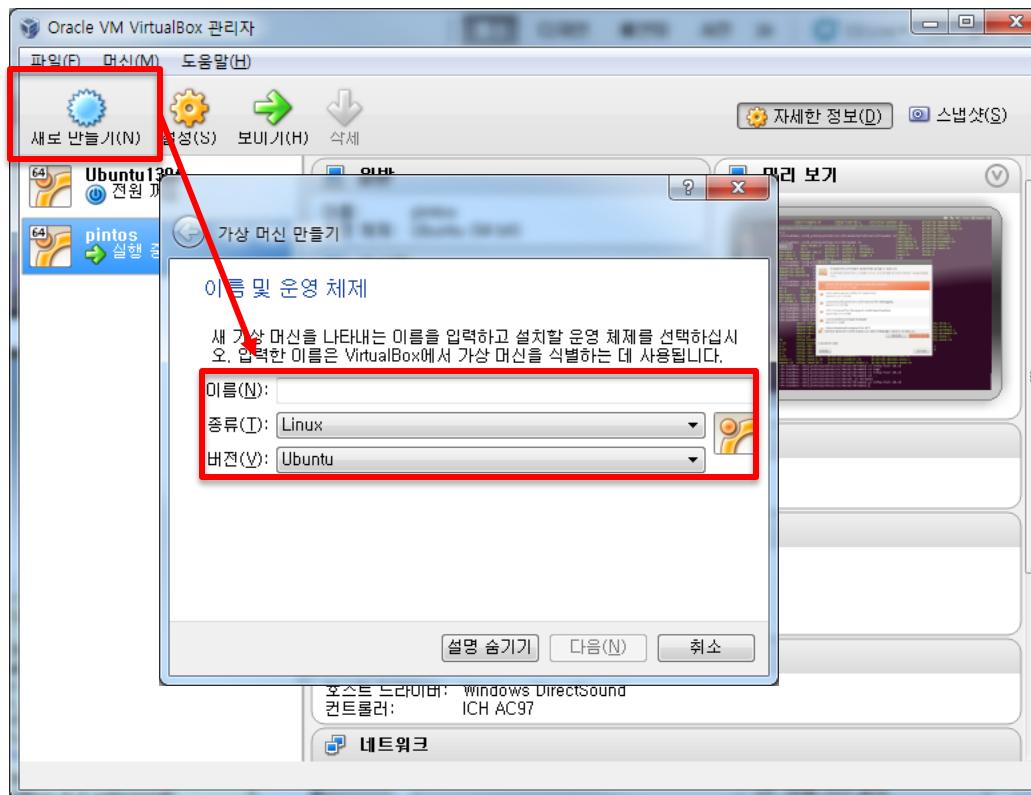
- <http://www.virtualbox.org> 에서 Virtual Box 다운로드 후 설치

여기서 다운로드

The screenshot shows a web browser window displaying the official VirtualBox website at <https://www.virtualbox.org>. The page features a large 'VirtualBox' logo and a 'Welcome to VirtualBox.org!' message. On the left, a sidebar lists navigation links: About, Screenshots, **Downloads** (which is highlighted with a red box), Documentation, End-user docs, Technical docs, Contribute, and Community. The main content area contains information about VirtualBox's capabilities, a 'Hot picks:' section with links to developer tools, and a 'News Flash' box with updates from July 4th, 2013, September 13th, 2012, and January 20th, 2012. At the bottom, there's an Oracle logo and links to Contact, Privacy policy, and Terms of Use.

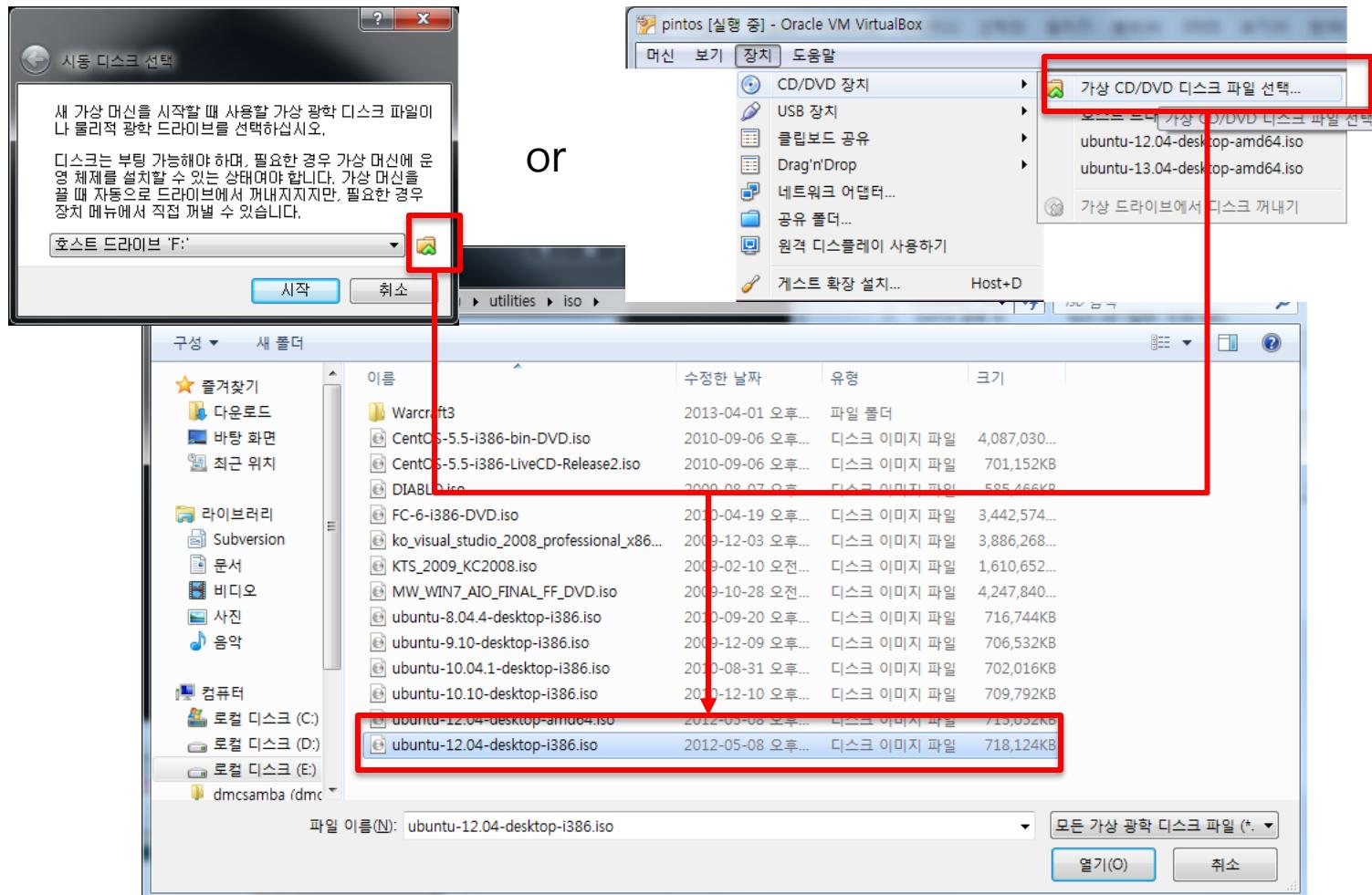
개인 PC에서 Pintos 사용하기 2.1: 우분투 설치

- VirtualBox상에 리눅스 가상 머신을 생성한다. 리눅스 설치(Ubuntu 12.04 LTS)
 - ubuntu-12.04-desktop-i386.iso 다운로드: <http://www.ubuntu.com/download/desktop>
 - 가상 머신 만들기



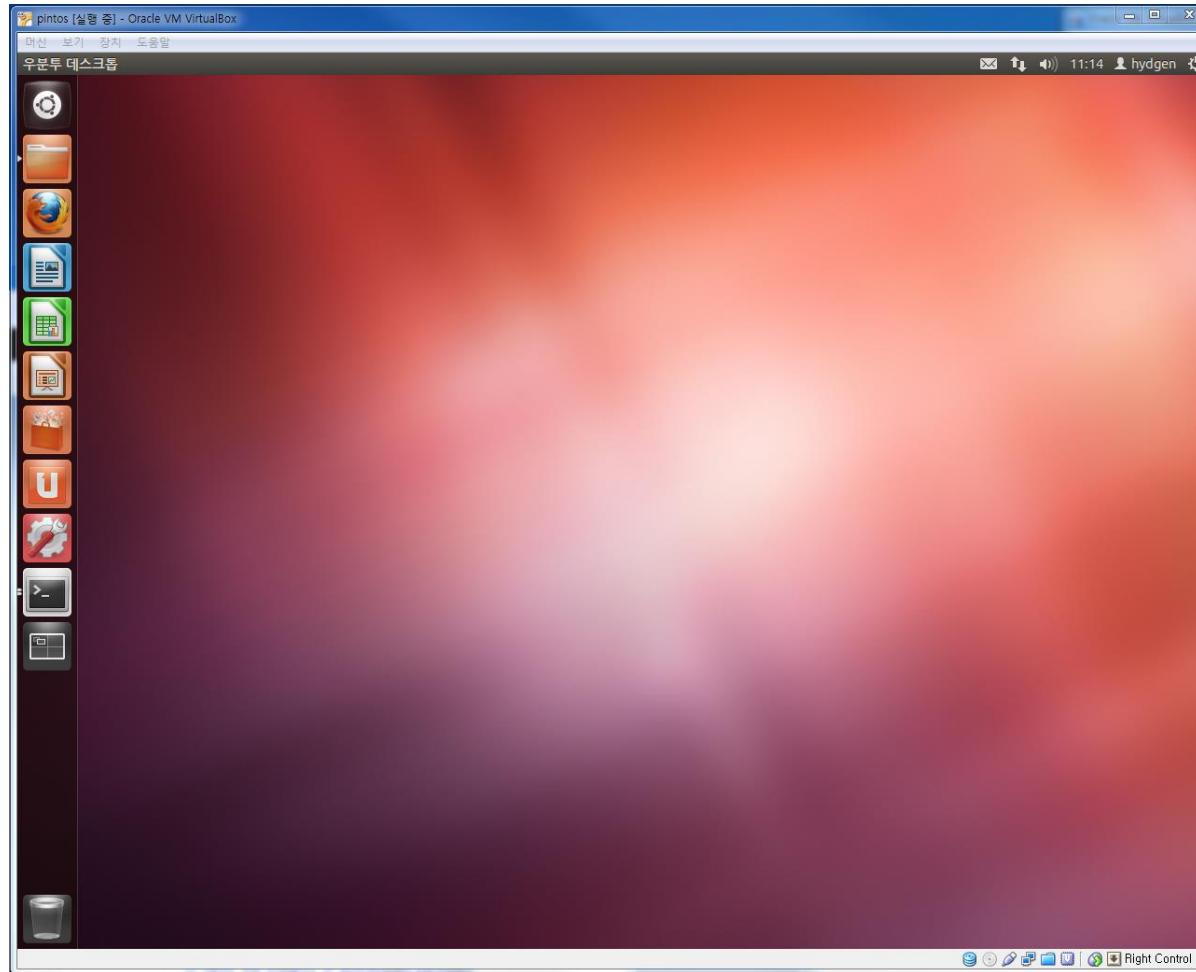
개인 PC에서 Pintos 사용하기 2.2: 우분투 설치

- 다운 받은 우분투 이미지 파일을 마운트하고, 설치



개인 PC에서 Pintos 사용하기 2.3: 우분투 설치 완료

- ◆ Ubuntu 설치 완료 및 부팅



개인 PC에서 Pintos 사용하기 3: Bochs 설치

- <http://bochs.sourceforge.net> 사이트에서 Bochs 다운로드

bochs: The Open Source IA-32 Emulation Project (Home Page) - Mozilla Firefox

bochs: The Open Source IA-32 ... +

bochs.sourceforge.net

Anonymous SVN Support

- Bochs Wiki
- Bochs FAQ
- Documentation
- User Guide
- Devel. Guide
- Doc. Guide
- Mailing Lists
- Discussion Boards
- SF Site Docs

Feedback

- Bug Reports
- Feature Requests

Development

- Get Involved
- View the Source
- Patches
- Tech Specs Pages
- Subversion Access

Resources

- Bochs History
- Bochs Links
- Related Links
- Screen Shots

allows you to run OS's and software within the emulator on your workstation, much like you have a machine inside of a machine. For instance, let's say your workstation is a Unix/X11 workstation, but you want to run Win'95 applications. Bochs will allow you to run Win 95 and associated software on your Unix/X11 workstation, displaying a window on your workstation, simulating a monitor on a PC.

여기서 다운로드

 Bochs 2.6.2 released on May 26, 2013 !
New Bochs 2.6.2 release is now available. You can download it from the [SourceForge project page](#).
See the [CHANGES](#) file for details on what has changed since release 2.6.1

Bochs IRC Chat Transcripts
The Bochs community held an IRC open discussion chat on Sunday, February 1, 2004. We talked about current and future developments ([Transcript](#)). Here are some transcripts of earlier conversations: [October 13, 2002](#), [April 7, 2002](#), [June 19, 2001](#), [May 30, 2001](#).

Bochs at ISCA-35
Bochs was presented at ISCA-35 in Beijing, China at "The 1st Workshop on Architectural and Microarchitectural Support for Binary Translation" by a paper "[Virtualization without direct execution - designing a portable VM](#)". Download [paper](#) and [presentation slides](#).

Want to know more about Bochs architecture ? [How the Bochs works under the hood \(2nd edition\)](#)

개인 PC에서 Pintos 사용하기 3: Bochs 설치

- 파일 압축 해제

```
$ tar xvf bochs-2.6.2.tar.gz
```

- bochs 폴더로 이동

```
$ ./configure --enable-gdb-stub --with-nogui
```

```
$ make
```

```
$ sudo make install
```



개인 PC에서 Pintos 사용하기 3: Bochs 설치

▣ 오류 1: C compiler cannot create executables

- ◆ gcc, g++ 및 라이브러리 패키지 설치

```
$ sudo apt-get install libc6-dev g++ gcc
```

▣ 오류 2: X windows libraries were not found

- ◆ X windows 라이브러리 설치

```
$ sudo apt-get install xorg-dev
```



개인 PC에서 Pintos 사용하기 3: QEMU 설치

```
$ sudo apt-get install qemu
```

```
$ sudo ln -s /usr/bin/qemu-system-i386 /usr/bin/qemu
```

개인 PC에서 Pintos 사용하기 4: Pintos 설치 및 환경 설정

- ▣ 해당 명령어를 통해 핀토스 소스코드 다운로드

```
$ wget http://dmclab.hanyang.ac.kr/wikidata/download/pintos.tar.gz
```

- ▣ 압축 해제

```
$ tar xvf pintos.tar.gz
```

- ▣ pintos/src/threads 디렉토리로 이동 후 make 명령어 입력



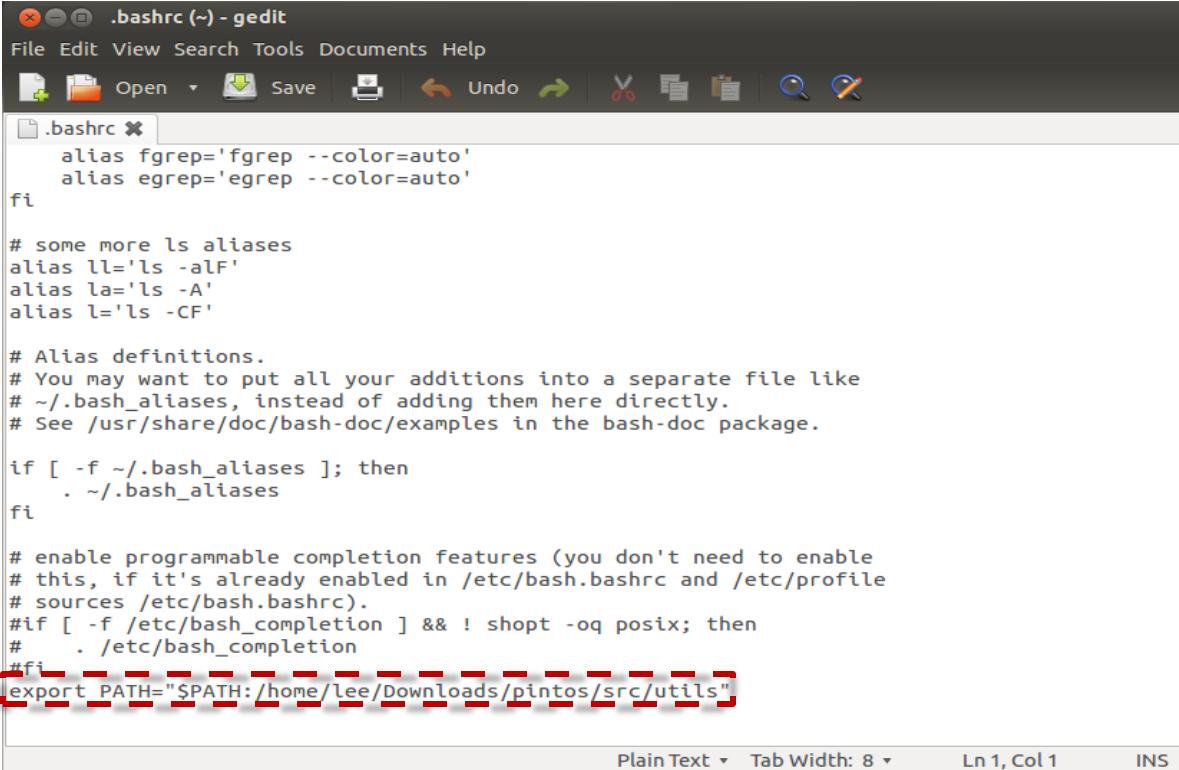
개인 PC에서 Pintos 사용하기 4: Pintos 설치 및 환경 설정

▣ ~/.bashrc 파일에 환경변수 설정

- ◆ pintos를 설치한 디렉토리 경로를 추가

```
export PATH="$PATH:/home/user/pintos/src/utils" 추가
```

- ◆ \$source ~/.bashrc 수정사항 적용



```
.bashrc (~) - gedit
File Edit View Search Tools Documents Help
Open Save Undo Redo Cut Copy Paste Find Replace
.bashrc *
alias fgrep='fgrep --color=auto'
alias egrep='egrep --color=auto'
fi

# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash_completion).
#if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
#    . /etc/bash_completion
#fi
export PATH="$PATH:/home/lee/Downloads/pintos/src/utils"
```

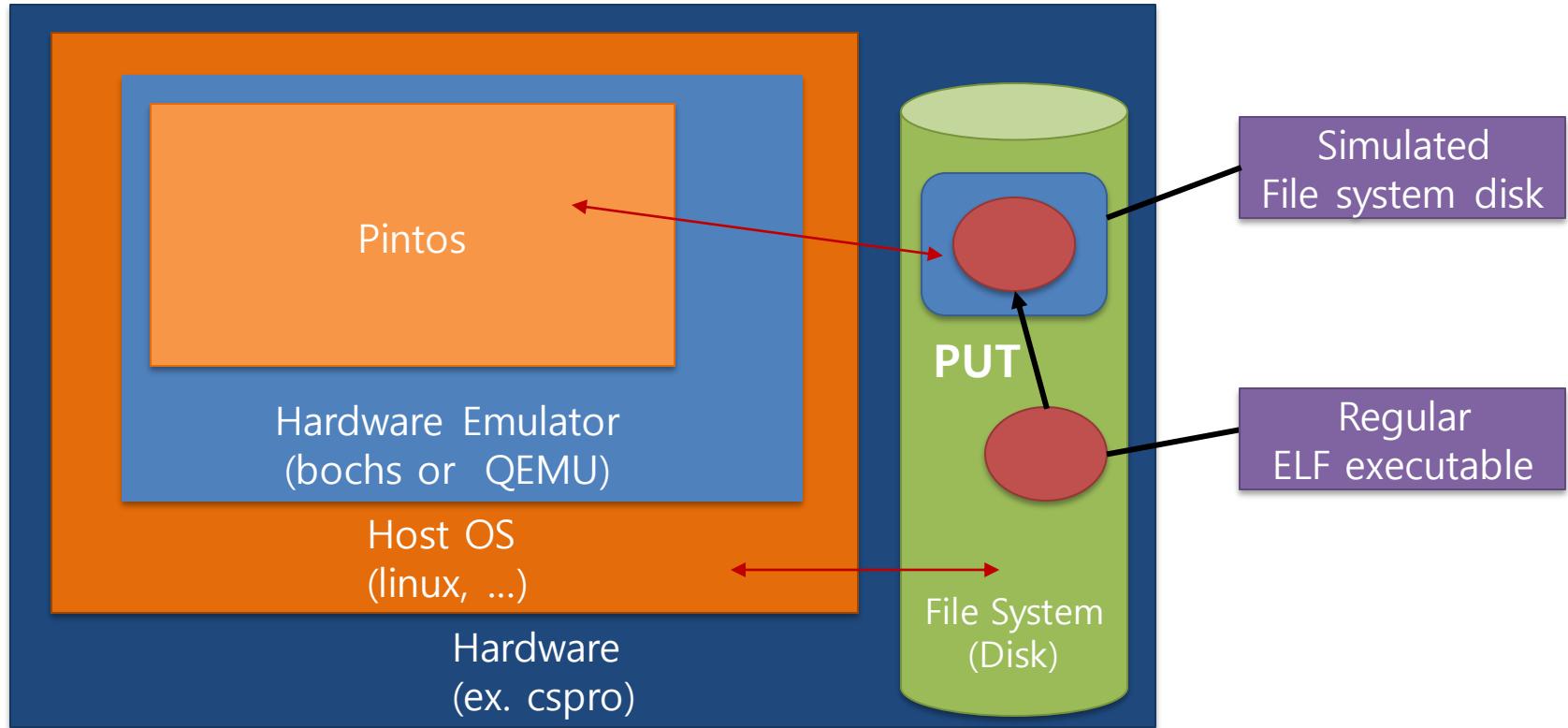
개인 PC에서 Pintos 사용하기 5: GCC 다운그레이드

- gcc버전을 pintos 권장인 4.5 버전으로 다운그레이드

```
$ sudo apt-get install gcc-4.5
```

```
$ sudo update-alternatives --install /usr/bin/gcc gcc  
/usr/bin/gcc-4.5 50
```

How User Program Work



Pintos File System 사용 방법

▣ Pintos에서 User program 실행하기

```
$ cd ~/pintos/src/userprog/  
$ make  
$ cd build
```

1. 파일시스템 디스크를 생성

```
$ pintos-mkdisk filesys.dsk --filesys-size=2
```

- `pintos-mkdisk` : pintos에서 제공하는 가상 디스크 생성 도구.
- 파일시스템 디스크의 이름은 무조건 `filesys.dsk`로 해야 한다.
- `--filesys-size` 옵션 : 가상디스크의 용량을 지정하기 위한 옵션.

2. 파일시스템 디스크 초기화

```
$ pintos -f -q
```

- `-f` 옵션 : `filesys.dsk`를 포맷. pintos의 자체 파일시스템 구조 사용.
- `-q` 옵션 : 모든 작업이 끝나면 pintos를 종료하게 하는 옵션.

Pintos File System 사용 방법 (Cont.)

▣ Pintos에서 User program 실행하기(cont.)

```
$ cd ~/pintos/src/examples/  
$ make  
$ cd ../userprog/bulid
```

3. 파일시스템 디스크에 실행할 응용 프로그램 복사(ex: echo 프로그램)

```
$ pintos -p ../../examples/echo -a echo -- -q
```

- -p 옵션 : 시스템 디스크로 복사, -a 옵션 : 프로그램 이름 설정
- -- : Pintos 프로그램에 전달할 인자와 Pintos에서 동작하는 가상 OS에 전달할 인자를 구분하기 위해 사용
- 실행 프로그램은 Regular ELF executable 파일 형식으로 만들어져야 함

4. 프로그램 실행 (ex: echo프로그램을 x라는 인자를 넣어서 실행)

```
$ pintos -q run 'echo x'
```

- run 옵션 : 응용 프로그램을 실행
- pintos의 현재 상태로는 응용 프로그램이 실행되지 않음



Pintos File System 사용 방법 (Cont.)

▣ Pintos에서 User program 실행하기(cont.)

- ◆ 결합된 명령어

```
$ pintos --filesys-size=2 -p ../../examples/echo -a echo -- -f  
-q run 'echo x'
```

- 앞의 모든 옵션을 한번에 넣어서 프로그램을 실행할 수 있음.
- 옵션은 반드시 -p, -a, -- , -f, -q, -run 순서로 실행되어야 함.



Patch 파일 만들기

- Patch 파일 생성은 'diff' 명령을 사용

```
$ diff [옵션] [원본파일] [수정된파일] > 출력파일.patch
```

```
$ diff [옵션] [원본디렉토리] [수정된디렉토리] > 출력파일.patch
```

- 주요 옵션(-urN)

- u : 수정된 부분의 앞뒤 동일한 내용을 3줄만 표시
 - unified=2 을 사용하면 동일한 라인을 2줄만 표시

- r : recursive. 재귀적 검색, 하위 디렉토리를 모두 검색하여 비교

- N : new file. 두 디렉토리 중 어느 한 디렉토리에만 파일이 있는 경우, 두 디렉토리 모두에 같은 파일이 존재한다고 가정하고 비교를 진행

- ◆ 자세한 옵션은 '\$ diff --help'로 확인



Patch 파일 만들기

▣ 예제

- ◆ 파일 vs 파일 비교하여 patch 파일 생성 예시

```
$ diff -u process.c process_new.c > process.c.patch
```

- ◆ 디렉토리 vs 디렉토리 비교하여 patch 파일 생성 예시(-r 옵션 사용)

```
$ diff -urN pintos/ pintos_new/ > pintos.patch
```



Patch 파일 적용하기

▣ Patch 파일 적용은 'patch' 명령을 사용

```
$ patch [옵션] < 결과파일.patch
```

▣ 주요 옵션

-p : -pNUM 형식으로 사용(NUM=숫자)

- NUM은 patch에 명시된 파일 경로에서 NUM 만큼 하위 디렉터리로 이동하여 patch를 적용하라는 의미
- Ex) patch 파일 안에 '--- driver/src/display.c'라는 경로가 있는 경우, '-p1'을 사용하면 'src/display.c' 경로에 위치한 파일에 해당 patch를 적용

▣ 패치 파일 적용 예시

```
$ patch -p0 < pintos.patch
```



1. 명령어 실행 기능의 구현

명령어의 실행

▣ 과제 목표

- ◆ 커맨드 라인에서 명령어의 실행

▣ 과제 설명

- ◆ 커맨드 라인의 문자열을 토큰으로 분리하는 기능 개발

- Pintos는 프로그램과 인자를 구분하지 못하는 구조

예: \$ls -a /* Pintos는 'ls -a'를 하나의 프로그램명으로 인식 */

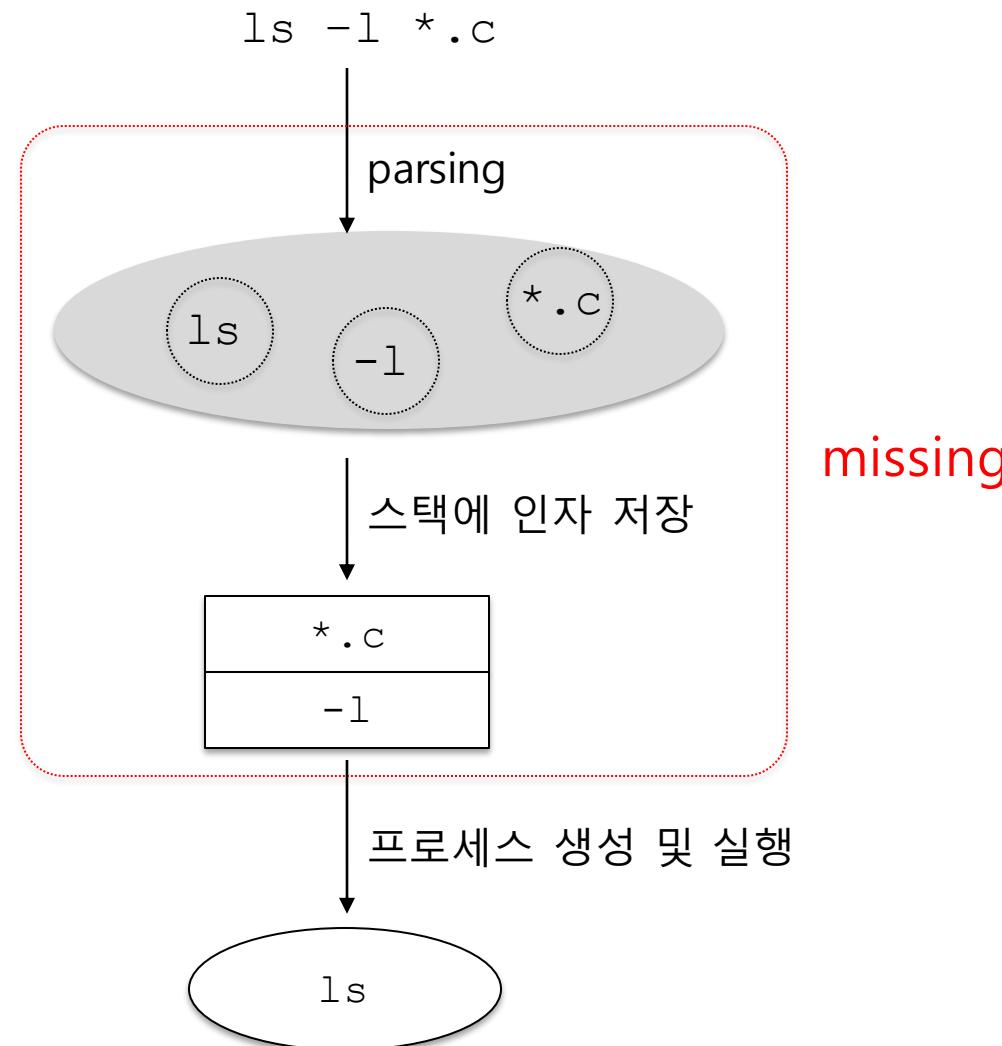
- ◆ 프로그램 이름과 인자를 구분하여 스택에 저장, 인자를 프로그램에 전달

▣ 수정 파일

- ◆ pintos/src/userprog/process.*



프로그램의 실행



핀토스에서 프로그램의 실행

run옵션(응용프로그램 실행)일 경우 run_task() 호출

```
int main(void)
{
    ...
    run_action(argv);
    shutdown_power_off();
    ...
}
```

```
static void run_action (char **argv)
{
    static const struct action
actions[] =
    {
        {"run", 2, run_task },
        ...
    };
}
```

프로그램의 실행

유저 프로세스 생성 후 핀토스는 프로세스 종료 대기

```
static void run_task(char ** argv)
{
    ...
    process_wait(process_execute(argv));
    ...
}
```

```
int process_wait (tid_t
                  child_tid UNUSED)
{
    return -1;
}
```

자식 프로세스가 종료될 때까지 대기(현재는 구현이 안됨)

프로그램의 실행

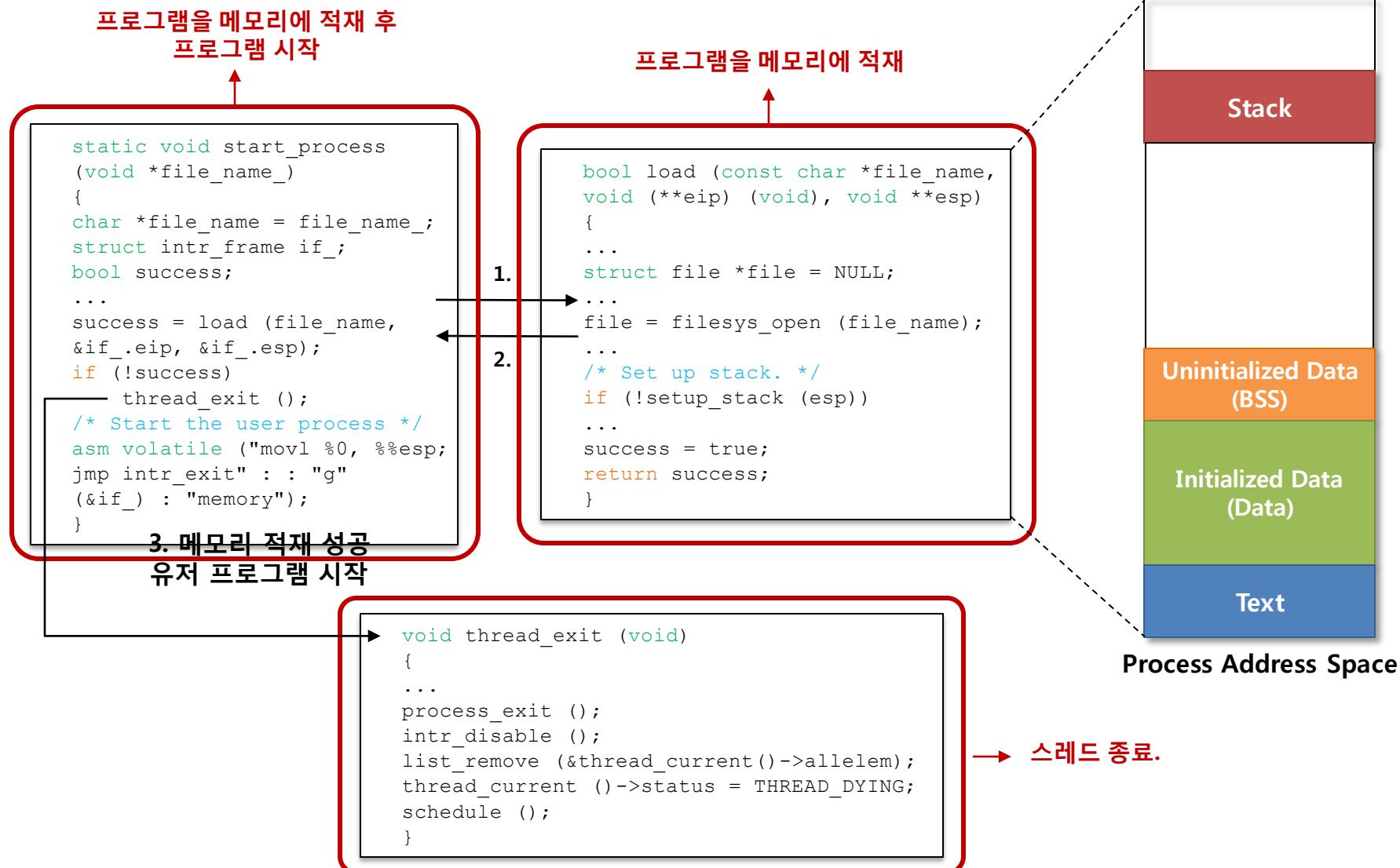
```
tid_t process_execute (const  
char *file_name)  
{  
    ...  
    tid = thread_create  
        (, start_process, );  
    ...  
    return tid;  
}
```

프로세스(스레드)생성 함수를
호출하고 tid 리턴

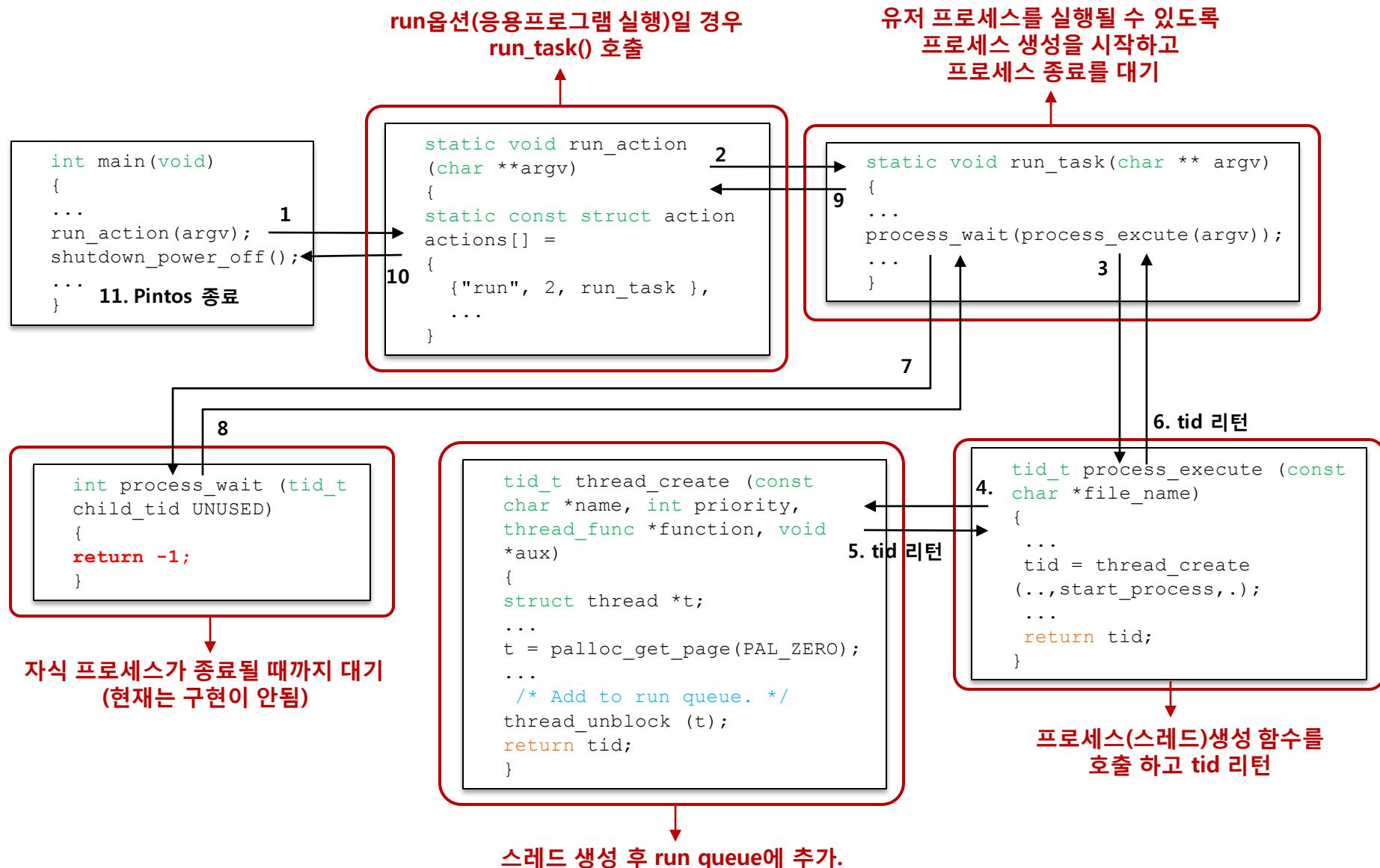
```
tid_t thread_create (const char  
*name, int priority,  
thread_func *function, void *aux)  
{  
    struct thread *t;  
    struct kernel_thread_frame *kf;  
    ...  
    t = palloc_get_page(PAL_ZERO);  
    ...  
    kf->function = function;  
    ...  
    /* Add to run queue. */  
    thread_unblock (t);  
    return tid;  
}
```

스레드 생성 후 run queue에 추가

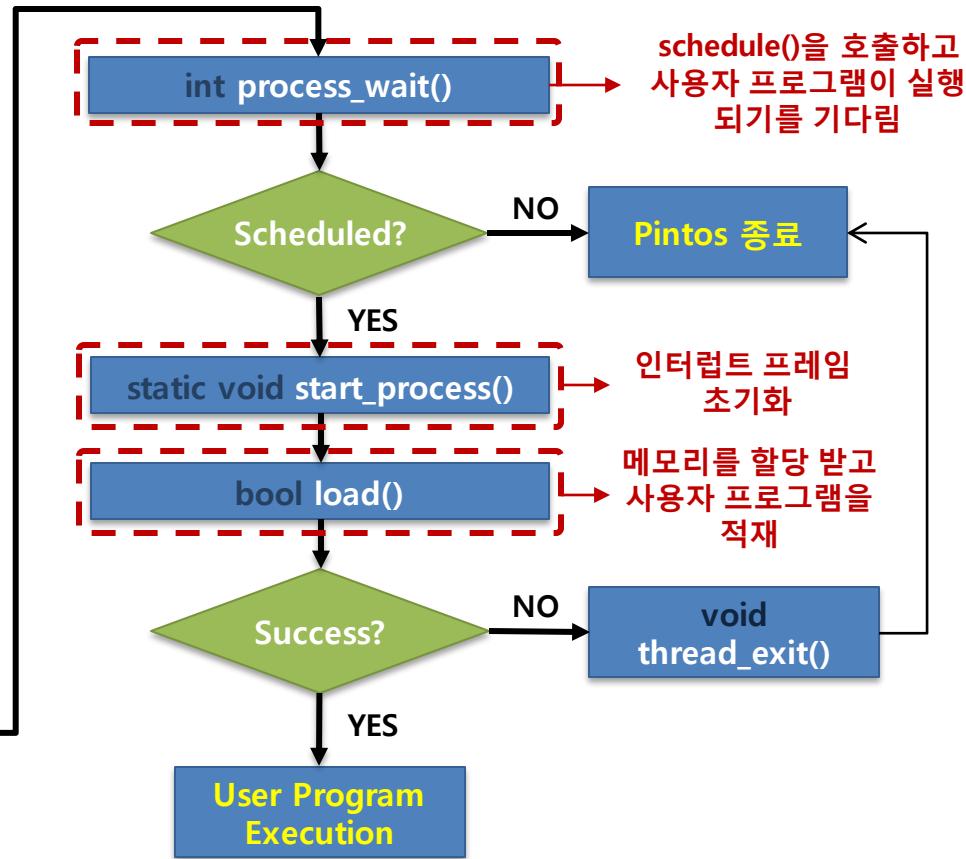
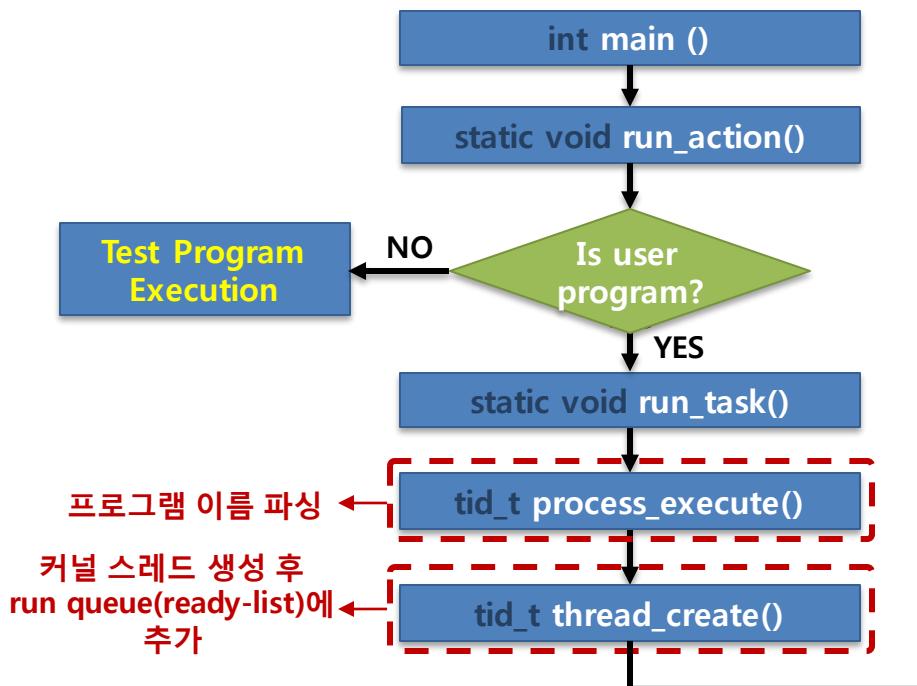
pintos의 프로그램 실행 모델



프로그램의 실행

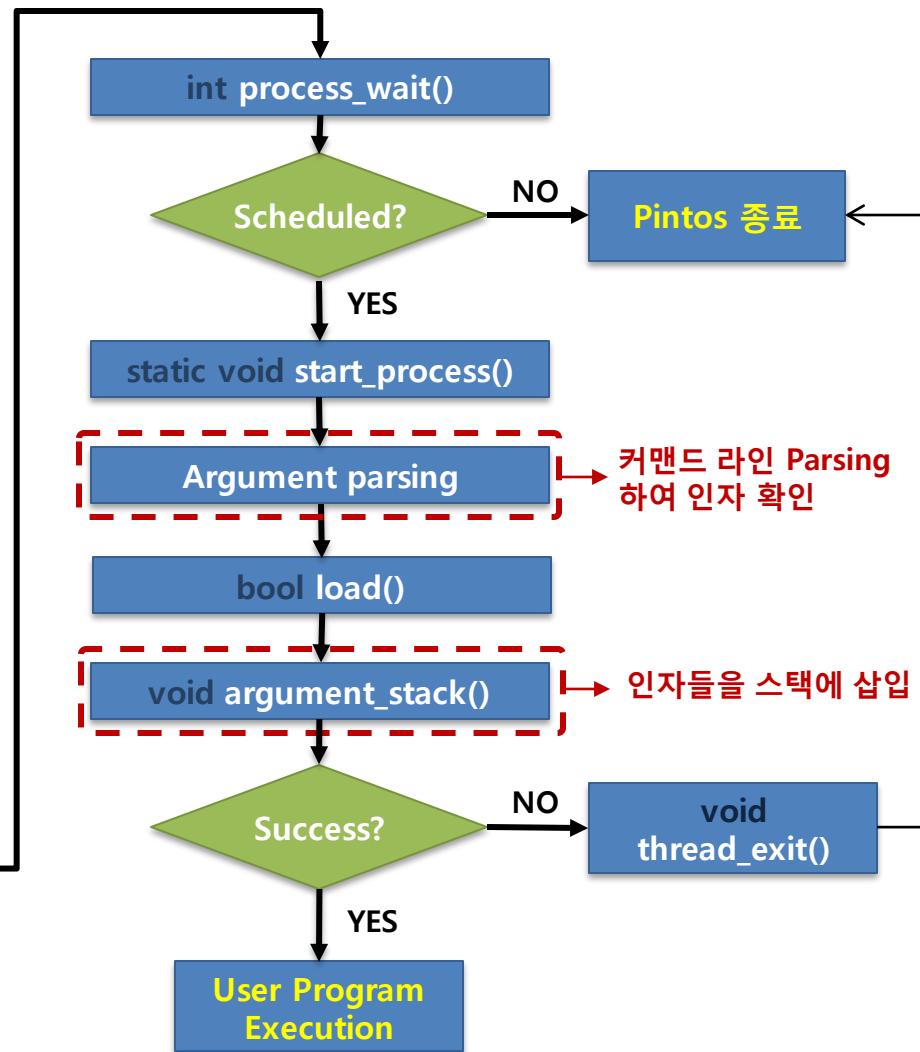
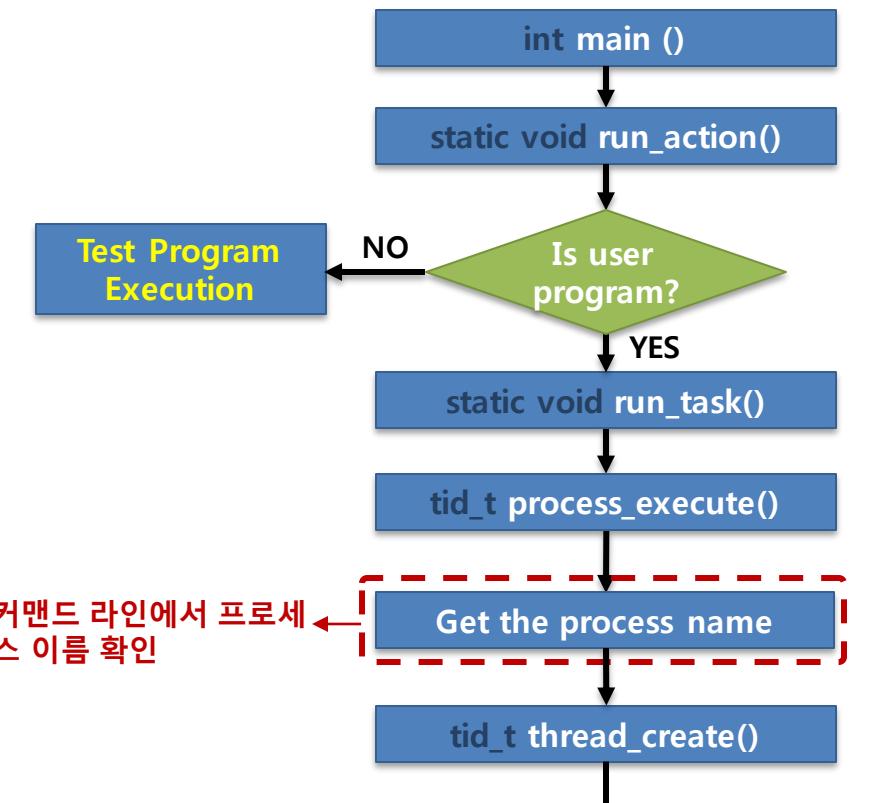


응용 프로그램 실행의 흐름



과제 후 응용 프로그램 실행의 흐름

추가할 부분



프로그램의 실행

- ▣ process_execute(char * str)
 - ◆ tid_t process_execute(const char *str);
 - ◆ 예) process_execute("echo");
 - echo 실행
 - ◆ load(char* str) 을 호출

스레드 생성

▣ 해당 함수를 수행하는 커널 스레드 생성

- ◆ 프로세스 디스크립터(struct thread) 생성 및 초기화
- ◆ 페이지 테이블을 저장하기 위한 메모리 할당
- ◆ 커널 스택 할당 후 커널 스레드가 수행할 함수를 등록
- ◆ 커널 스레드를 run queue(ready list)에 추가

pintos/src/threads/thread.c

```
tid_t thread_create (const char *name, int priority,
                      thread_func *function, void *aux)
{
    struct thread *t;
    struct kernel_thread_frame *kf;
    ...
    t = palloc_get_page (PAL_ZERO); /* 페이지 할당 */
    init_thread (t, name, priority); /* thread 구조체 초기화*/
    tid = t->tid = allocate_tid (); /* tid 할당 */
    /* Stack frame for kernel_thread(). */
    kf = alloc_frame (t, sizeof *kf); /* 커널 스택 할당 */
```

스레드 생성 (Cont.)

pintos/src/threads/thread.c – thread_create() 계속

```
kf->eip = NULL;
kf->function = function; /* 스레드가 수행할 함수 */
kf->aux = aux;           /* 수행할 함수의 인자 */
...
/* Add to run queue. */
thread_unblock (t);
return tid;
}
```



프로세스 탑재

▣ 프로그램을 메모리에 적재하고 실행하는 함수

- ◆ load(): file_name의 프로그램을 메모리에 적재
- ◆ 메모리 적재 성공 시 응용 프로그램 실행, 실패 시 스레드 종료
 - thread_exit(): 실행중인 스레드 종료

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    char *file_name = file_name_;
    ...
    /* if_.esp는 스택 포인터 */
    success = load (file_name, &if_.eip, &if_.esp);
    if (!success)
        thread_exit ();
    /* start user program */
    asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g"
                 (&if_) : "memory");
}
```

프로세스 탑재 (Cont.)

프로그램을 메모리에 적재 후 프로그램 시작

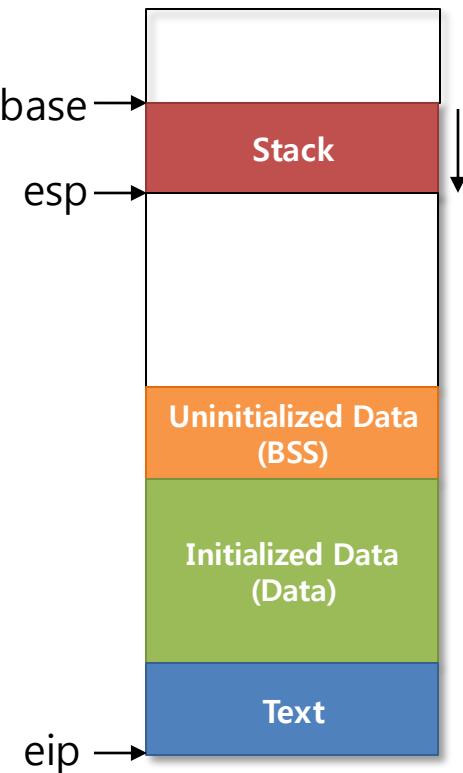
```
static void start_process (void *file_name_)
{
    char *file_name = file_name_;
    struct intr_frame if_;
    bool success;

    ...
    success = load (file_name, &if_.eip, &if_.esp);
    if (!success)
        thread_exit ();
    /* Start the user process */
    asm volatile ("movl %0, %%esp; jmp intr_exit" :: "g"
                 (&if_) : "memory");
}
```

load(char* str)

```
bool load (const char *file_name, void (**eip)(void), void **esp);
```

- ◆ User process의 페이지 테이블 생성
- ◆ 파일을 open하고, ELF 헤더정보를 메모리로 읽어 들임.
- ◆ 각 세그먼트의 가상주소공간 위치를 읽어 들임. Stack base →
- ◆ 각 세그먼트를 파일로부터 읽어 들임.
- ◆ 스택 생성 및 초기화
 - esp: 스택 포인터 주소
 - eip: text 세그먼트 시작주소



load(char *str): 페이지 테이블 생성

- ▣ pagedir_create(): 유저 프로세스의 페이지 테이블을 생성
- ▣ process_activate(): PDBR(cr3) 레지스터 값을 실행중인 스레드의 페이지 테이블 주소로 변경

pintos/src/userprog/process.c

```
bool load (const char *file_name, void (**eip) (void), void
**esp)
{
    struct thread *t = thread_current ();
    struct Elf32_Ehdr ehdr;
    struct file *file = NULL;
    ...
    t->pagedir = pagedir_create ();           /* 페이지 디렉토리 생성 */
    process_activate ();                      /* 페이지 테이블 활성화 */
    file = filesys_open (file_name);          /* 프로그램 파일 Open */
```

load(char *str)

pintos/src/userprog/process.c - load() 계속

```
/* ELF파일의 헤더 정보를 읽어와 저장 */
if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
    || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
    || ehdr.e_type != 2
    || ehdr.e_machine != 3
    || ehdr.e_version != 1
    || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
    || ehdr.e_phnum > 1024)
/* 배치 정보를 읽어와 저장. */
struct Elf32_Phdr phdr;
if (file_ofs < 0 || file_ofs > file_length (file))
    file_seek (file, file_ofs);
if (file_read (file, &phdr, sizeof phdr) != sizeof phdr)
...
/* 배치정보를 통해 파일을 메모리에 적재. */
if (!load_segment (file, file_page, (void *) mem_page,
                  read_bytes, zero_bytes, writable))
...
if (!setup_stack (esp)) /* 스택 초기화 */
...
}
```



pintos의 프로그램 실행 모델

스레드 종료

```
void thread_exit (void)
{
    ...
    process_exit ();
    intr_disable ();
    list_remove (&thread_current ()->allelem);
    thread_current ()->status = THREAD_DYING;
    schedule ();
}
```



함수 수정

- ▣ 문자열을 파싱하고 유저 스택에 인자 값이 저장 되도록 수정

pintos/src/userprog/process.c

```
tid_t process_execute() (const char *file_name)
```

- ◆ file_name 문자열을 파싱
- ◆ 첫 번째 토큰을 thread_create() 함수에 스레드 이름으로 전달

```
static void start_process() (void *file_name_)
```

- ◆ file_name 문자열 파싱
- ◆ argument_stack() 함수를 이용해 스택에 토큰들을 저장



process_execute() 수정

▣ 프로그램을 실행할 스레드 생성 (thread_create())

- ◆ file_name : 스레드 이름 (문자열)
- ◆ PRI_DEFAULT : 스레드 우선순위 (31)
- ◆ start_process : 생성된 스레드가 실행할 함수를 가리키는 포인터
- ◆ fn_copy : start_process 함수를 수행할 때 사용하는 인자 값

pintos/src/userprog/process.c

```
tid_t process_execute (const char *file_name)
{
    char *fn_copy;
    tid_t tid;
    ...
    tid = thread_create (file_name, PRI_DEFAULT, start_process,
                         fn_copy);
    ...
    return tid;
}
```

실행 파일 이름 전달

▣ 스레드의 이름을 실행 파일명으로 지정

- ◆ 커맨드 라인의 첫 번째 토큰을 `thread_create()` 함수의 첫 인자로 전달 되도록 프로그램을 수정
- ◆ 현재는 커맨드 라인 전체가 `thread_create()`에 전달되고 있음

pintos/src/userprog/process.c

```
tid_t process_execute (const char *file_name)
{
    char *fn_copy;
    tid_t tid;
    ...
    /* 첫번째 공백 전까지의 문자열 파싱 */
    ...
    /* Create a new thread to execute FILE_NAME. */
    tid = thread_create (file_name, PRI_DEFAULT, start_process,
    fn_copy);
    ...
}
```

스레드 이름

실행 파일 로드 기능 추가

▣ 실행 파일 로드

- ◆ 인자들을 토큰화(strtok_r() , string.h) 및 토큰의 개수 계산
- ◆ 실행 파일 이름을 load() 함수의 첫 번째 인자로 전달

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    char *file_name = file_name_;
    struct intr_frame if_;
    bool success;

    ...
    /* 인자들을 띄어쓰기 기준으로 토큰화 및 토큰의 개수 계산
       (strtok_r() 함수 이용) */

    /* Initialize interrupt frame and load executable. */
    memset (&if_, 0, sizeof if_);
    ...
    success = load (file_name, &if_.eip, &if_.esp);
    ...
}
```

프로그램 이름

함수 추가

▣ 유저 스택에 파싱된 토큰을 저장하는 함수 구현

pintos/src/userprog/process.c

```
void argument_stack(char **parse ,int count ,void **esp)
```

- ◆ 유저 스택에 프로그램 이름과 인자들을 저장하는 함수
- ◆ parse: 프로그램 이름과 인자가 저장되어 있는 메모리 공간, count: 인자의 개수, esp: 스택 포인터를 가리키는 주소

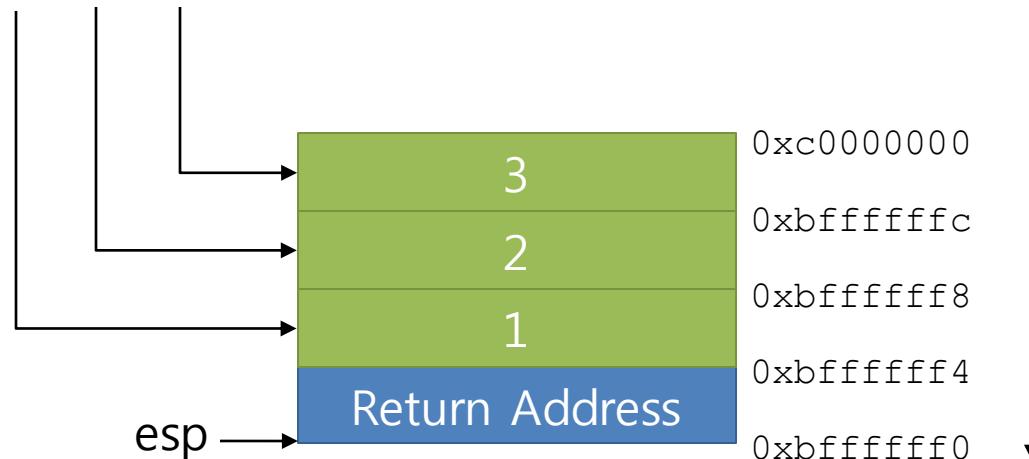


함수 인자 전달

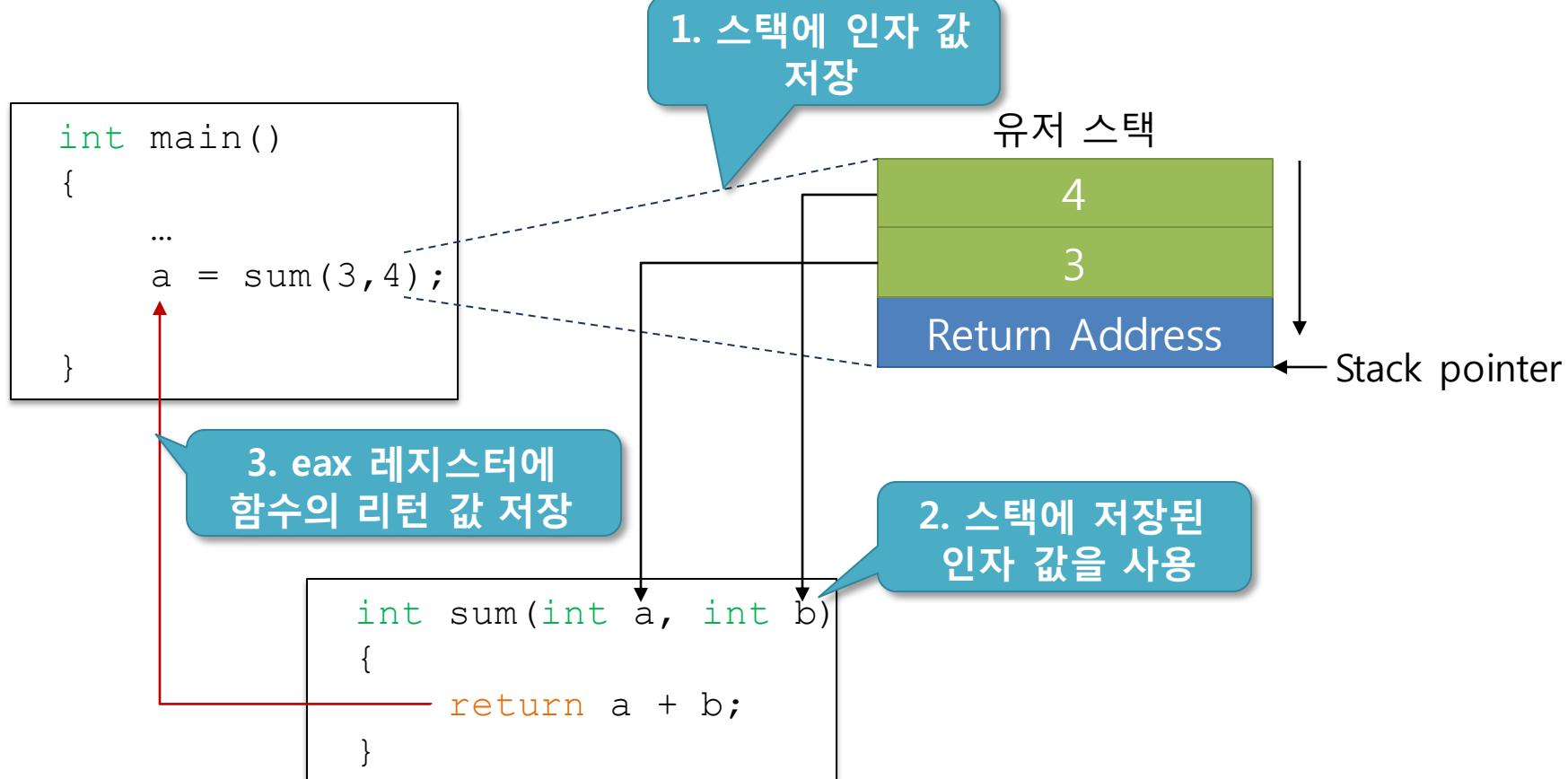
▣ 함수호출 방식 (80x86)

- ◆ 함수 호출 시 인자 값은 오른쪽 -> 왼쪽 순으로 스택에 저장
- ◆ Return Address : Caller(함수를 호출하는 부분)의 다음 수행 명령어 주소
- ◆ Callee(호출 받은 함수)의 리턴 값은 eax 레지스터에 저장

ex) `function(1, 2, 3)`



Argument Passing 흐름



프로그램의 실행과 스택을 통한 인자 전달

```
$bin/ls -l foo bar
```

- argc = 4
- argv[0] = "bin/ls", argv[1] = "-l", argv[2] = "foo", argv[3] = "bar"

Address	Name	Data	Type	
0xbfffffffcc	argv[3][...]	'bar\0'	char[4]	Argument(문자열)
0xbfffffff8	argv[2][...]	'foo\0'	char[4]	
0xbfffffff5	argv[1][...]	'-l\0'	char[3]	
0xbfffffff4	word-align	0	uint8_t	
0xbffffffec	argv[0][...]	'/bin/ls\0'	char[8]	Argument의 주소
0xbffffffe8	argv[4]	0	char *	
0xbffffffe4	argv[3]	0xbfffffc	char *	
0xbffffffe0	argv[2]	0xbfffff8	char *	
0xbffffffdcc	argv[1]	0xbfffff5	char *	main(int argc , char **argv)
0xbffffffd8	argv[0]	0xbfffffed	char *	
0xbffffffd4	argv	0xbfffffd8	char **	
0xbffffffd0	argc	4	int	
0xbfffffcc	return address	0 (fake address)	void (*) () → fake address(0)	
stack top →				

Pintos의 인터럽트 프레임

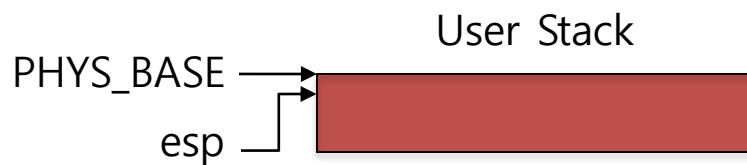
- 실행중인 프로세스의 레지스터 정보, 스택 포인터, Instruction Count를 저장하는 자료구조
 - ◆ 인터럽트나 시스템 콜 호출 시 사용
 - ◆ esp : 스택 포인터 위치
 - ◆ eax : 함수의 반환 값을 저장

pintos/src/threads/interrupt.h

```
struct intr_frame
{
    /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
    ...
    uint32_t eax;                      /* Saved EAX. */
    ...
    void *esp;                         /* Saved stack pointer. */
    ...
}
```

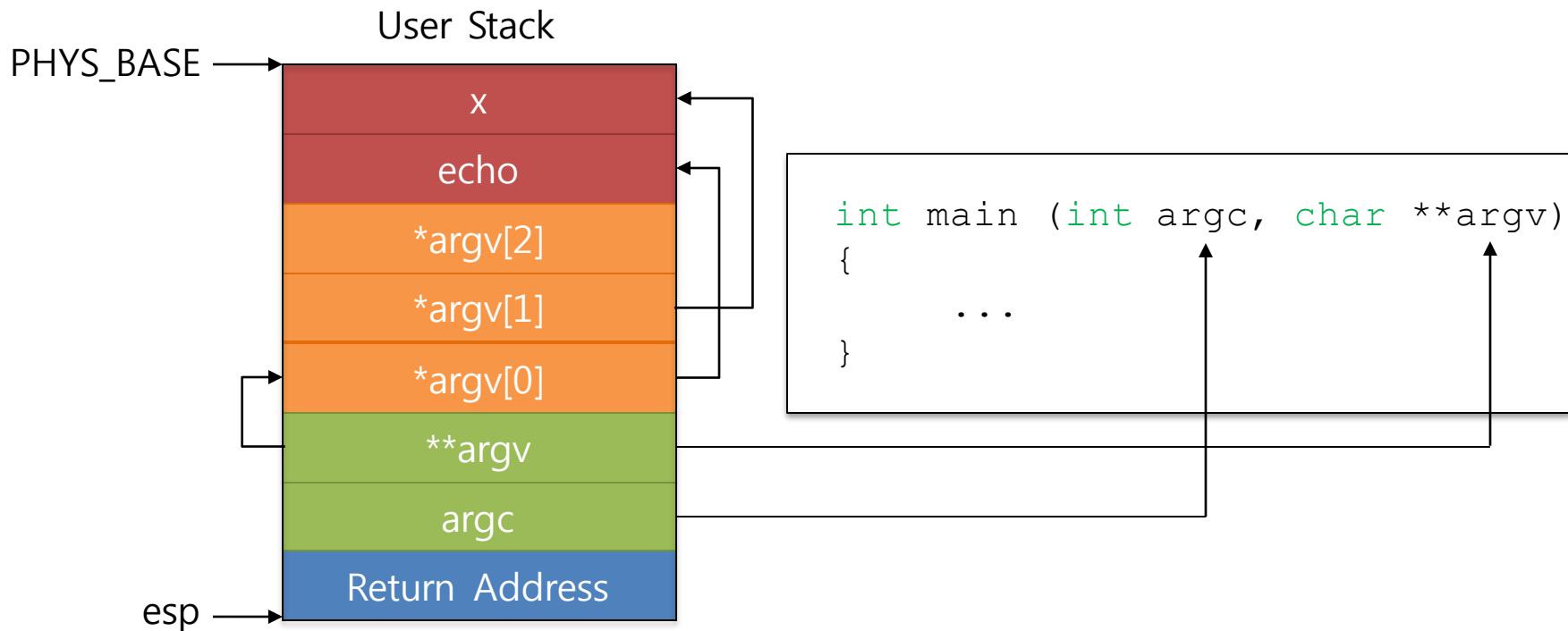
유저 스택에 인자 삽입

- ▣ 기존 Pintos에서는 유저 스택에 프로그램 인자 값이 삽입되지 않음
 - ◆ 유저 프로그램 실행 시, User Stack은 비어있는 상태
 - ◆ 인자들을 스택 삽입하는 기능 추가 필요



유저 스택에 인자 삽입 (Cont.)

- 프로그램 실행 시 유저 스택에 저장된 값들이 main 함수의 인자
- ex) \$pintos run 'echo x'



esp 스택 포인터 다루기

- esp는 높은 메모리 주소에서 낮은 메모리 주소로 이동하면서 인자를 스택에 삽입
 - parse: 프로그램 이름과 인자가 저장되어 있는 메모리 공간 (`char **parse`)
 - count: 인자의 개수
 - esp: 스택 포인터를 가리키는 주소 값 (`void **esp`)

```
...
int i, j;
for(i = count - 1 ; i > -1 ; i--)
{
    for(j = strlen(parse[i]) ; j > -1 ; j--)
    {
        *esp = *esp - 1;
        **(char **)esp = parse[i][j];
    }
    ...
}
...
```

스택 주소를 감소시키면서
인자를 스택에 삽입

실행 파일에 인자 전달

▣ 스택에 프로그램명, 함수인자 저장

- ◆ parse: 프로그램 이름과 인자가 저장되어 있는 메모리 공간
- ◆ count: 인자의 개수
- ◆ esp: 스택 포인터를 가리키는 주소 값

pintos/src/userprog/process.c

```
void argument_stack(char **parse , int count , void **esp)
{
    /* 프로그램 이름 및 인자(문자열) push */
    /* 프로그램 이름 및 인자 주소들 push */
    /* argv (문자열을 가리키는 주소들의 배열을 가리킴) push*/
    /* argc (문자열의 개수 저장) push */
    /* fake address(0) 저장 */
}
```

argument_stack() 호출

- 유저 프로그램이 실행 되기 전에 argument_stack() 함수를 호출하여 스택에 인자 저장

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    ...
    success = load (file_name, &if_.eip, &if_.esp);
    ...
    argument_stack(parse , count , &if_.esp); 추가
    /* Start the user process */
    asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g"
(&if_) : "memory");
    NOT_REACHED ();
}
```

argument_stack() 호출 (Cont.)

```
asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) :  
"memory");
```

- ◆ 인터럽트 프레임(if_)에 저장된 유저 프로그램 컨텍스트를 유저 스택으로 복사 후 'intr_exit' 위치로 실행흐름 변경
- ◆ 유저 스택에 저장된 유저 프로그램 컨텍스트를 각 레지스터에 세팅
 - movl %0, %%esp : 유저 스택(esp)에 인터럽트 프레임(if_)을 복사
 - jmp intr_exit : intr_exit 위치로 실행 흐름을 변경
 - intr_exit: 유저 스택에 저장된 각 레지스터의 값을 각 레지스터에 저장

중간 점검

▣ hex_dump()(stdio.h)

- ◆ pintos에서 제공하는 디버깅 툴
- ◆ 메모리 내용을 16진수로 화면에 출력
- ◆ 유저 스택에 인자를 저장 후 유저 스택 메모리 확인

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    ...
    success = load (file_name, &if_.eip, &if_.esp);
    ...
    argument_stack(parse , count , &if_.esp);
    hex_dump(if_.esp , if_.esp , PHYS_BASE - if_.esp ,
true);추가

    asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g"
(&if_) : "memory");
    NOT_REACHED ();
}
```



중간 점검 (Cont.)

Result

```
$pintos -v -- run 'echo x'
```

```
Execution of 'echo x' complete.  
'echo'  
'x'  
Success : 1  
esp : bfffffe0  
bfffffe0 00 00 00 00 02 00 00 00-ec ff ff bf f9 ff ff bf |.....|  
bffffff0 fe ff ff bf 00 00 00 00-00 65 63 68 6f 00 78 00 |.....echo.x.|  
system call!
```

return address
(fake)

argc

argv

echo

x

수정 & 추가 함수

```
tid_t process_execute() (const char *file_name)
/* 프로그램을 실행 할 프로세스 생성 */
```

```
static void start_process() (void *file_name_)
/* 프로그램을 메모리에 적재하고 응용 프로그램 실행 */
```

```
void argument_stack(char **parse ,int count ,void **esp)
/* 함수 호출 규약에 따라 유저 스택에 프로그램 이름과 인자들을 저장 */
```

2. 시스템 콜 핸들러 구현

시스템 콜 핸들러 구현 개요

▣ 과제 목표

- ◆ 시스템 콜 핸들러 및 시스템 콜 (halt, exit, create, remove) 구현

▣ 과제 설명

- ◆ pintos는 시스템 콜 핸들러가 구현되어 있지 않아 시스템 콜이 호출될 수 없으므로 응용 프로그램이 정상적으로 동작하지 않는다.
- ◆ 사용자는 Pintos의 시스템 콜 메커니즘을 이해하고 시스템 콜 핸들러를 구현한다.
- ◆ 시스템 콜(halt, exit, create, remove)을 구현하고 시스템 콜 핸들러를 통해 호출 한다.

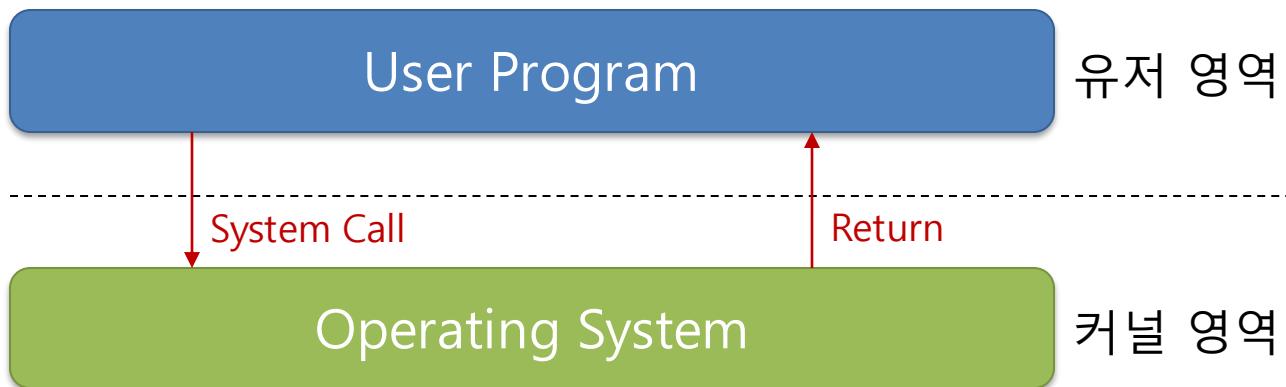
▣ 수정 파일

- ◆ pintos/src/userprog/syscall.*

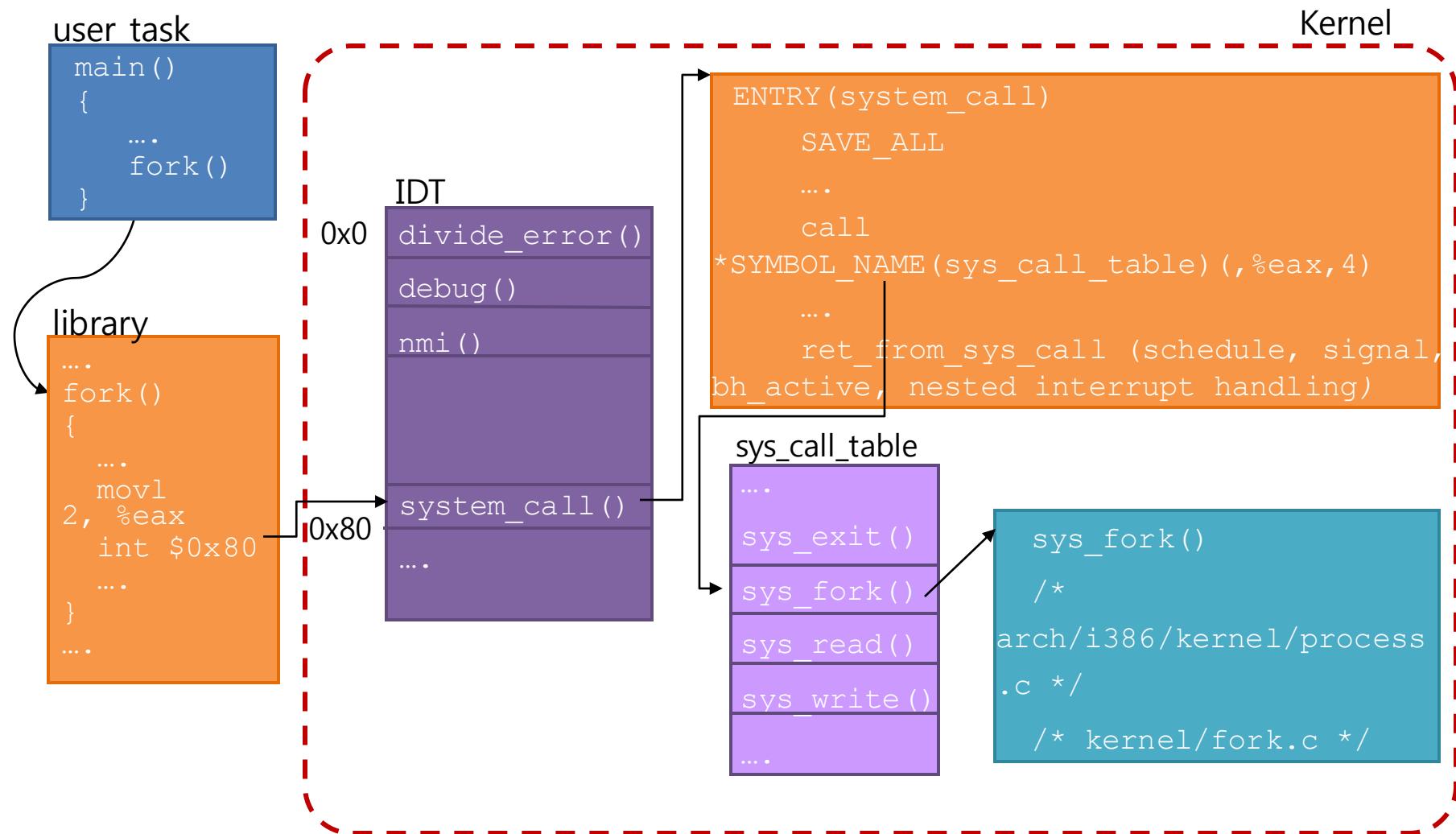


시스템 콜

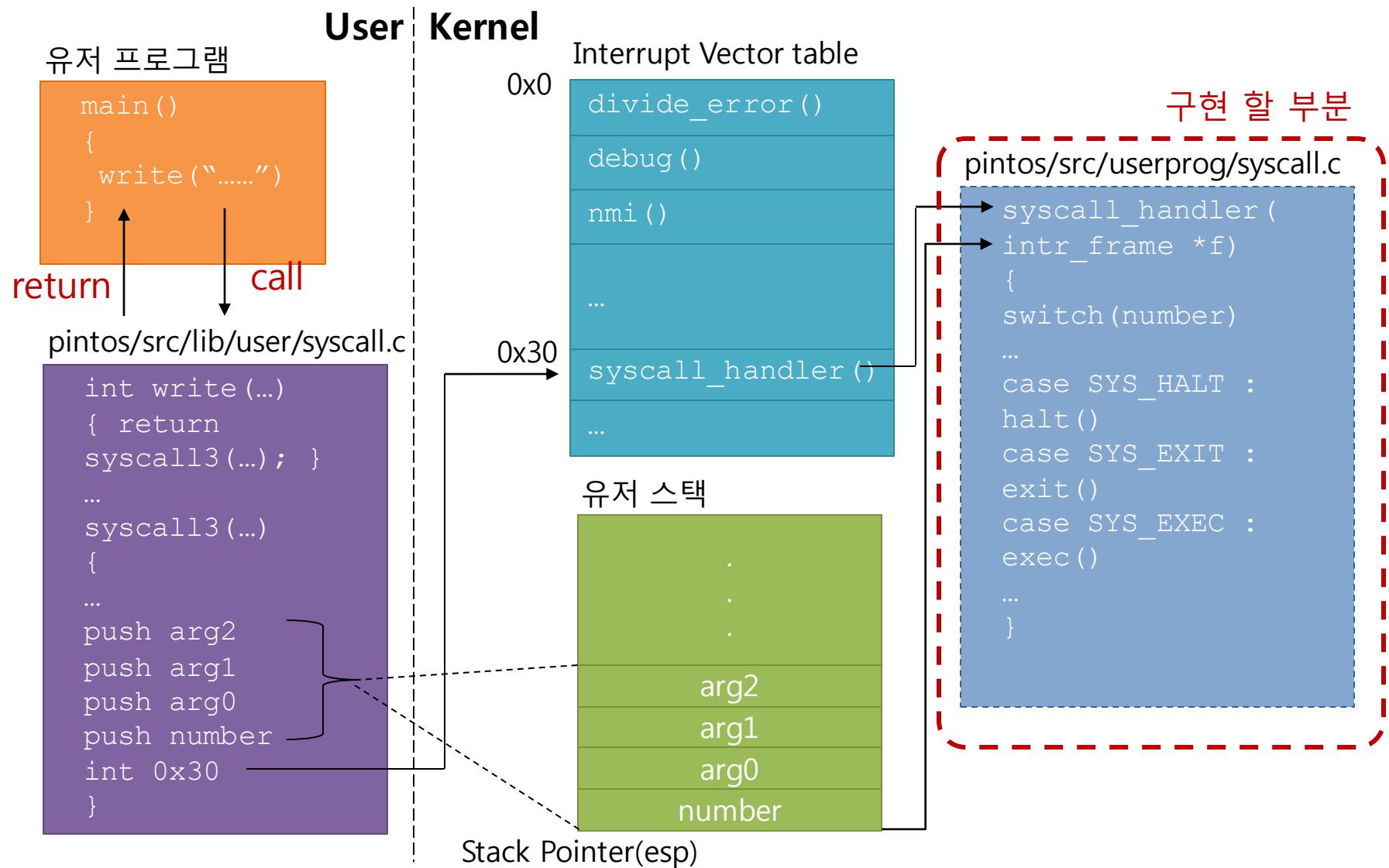
- ▣ 시스템 콜은 운영 체제가 제공하는 서비스에 대한 프로그래밍 인터페이스
- ▣ 사용자 모드 프로그램이 커널 기능을 사용할 수 있도록 함
- ▣ 시스템 콜은 커널 모드에서 실행되고, 처리 후 사용자 모드로 복귀됨



시스템 콜 호출 과정 (리눅스)



시스템 콜 호출 과정 (Pintos)



시스템 콜 핸들러 구현 요구사항

▣ 시스템 콜 핸들러 구현

- ◆ 시스템 콜 핸들러에서 시스템 콜 번호에 해당하는 시스템 콜 호출
- ◆ 시스템 콜 핸들러에서 유저 스택 포인터(esp) 주소와 인자가 가리키는 주소(포인터)가 유저 영역인지 확인
 - pintos는 유저영역을 벗어난 주소를 참조할 경우 페이지 폴트 발생
- ◆ 유저 스택에 있는 인자들을 커널에 저장
- ◆ 시스템 콜의 함수의 리턴 값은 인터럽트 프레임의 eax에 저장

▣ 시스템 콜 구현

- ◆ halt(), exit(), create(), remove() 시스템 콜을 구현



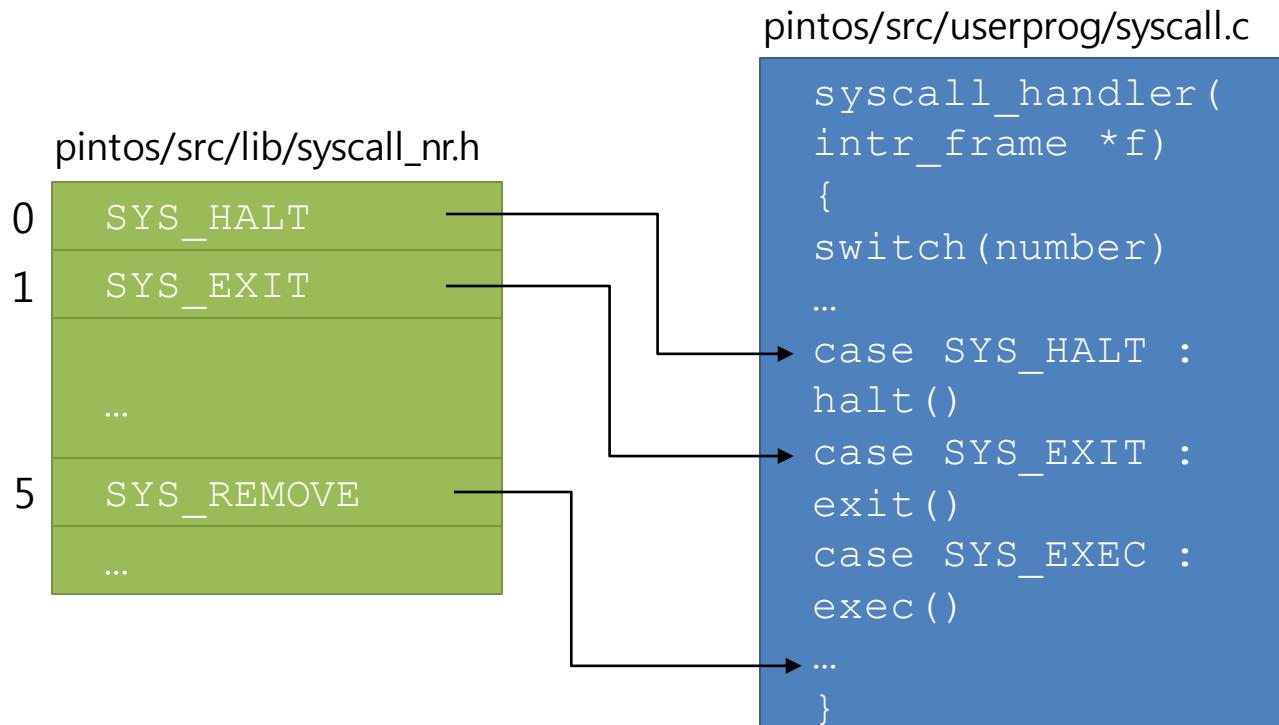
시스템 콜 핸들러 구현 요구사항 (Cont.)

▣ Pseudo code

```
procedure syscall_handler (interrupt frame)
    get stack pointer from interrupt frame
    get system call number from stack
    switch (system call number) {
        case the number is halt:
            call halt function;
            break;
        case the number is exit:
            call exit function;
            break;
        ...
        default
            call thread_exit function;
    }
```

시스템 콜 핸들러 구현 예시

- 시스템 콜 번호를 이용하여 시스템 콜 핸들러에서 해당 시스템 콜을 호출
 - 시스템 콜 넘버는 pintos/src/lib/syscall_nr.h에 정의



check_address() 구현

- ▣ 주소유효성 검사: 포인터가 가리키는 주소가 사용자 영역(0x8048000~0xc0000000)인지 확인
 - ◆ 유저 영역을 벗어난 영역일 경우 프로세스 종료(exit(-1))

pintos/src/userprog/syscall.c

```
void check_address(void *addr)
{
    /* 포인터가 가리키는 주소가 유저영역의 주소인지 확인 */
    /* 잘못된 접근일 경우 프로세스 종료 */
}
```



get_argument 구현

▣ 유저 스택에 있는 인자들을 커널에 저장

- ◆ 스택에서 인자들을 4byte 크기로 꺼내어 arg 배열에 순차적으로 저장
- ◆ count 개수 만큼의 인자를 스택에서 가져옴

pintos/src/userprog/syscall.c

```
void get_argument(void *esp, int *arg, int count)
{
    /* 유저 스택에 저장된 인자값들을 커널로 저장 */
    /* 인자가 저장된 위치가 유저영역인지 확인 */
}
```

시스템 콜 핸들러 구현

- 유저 스택에 저장된 시스템 콜 넘버에 해당하는 시스템 콜을 호출

pintos/src/userprog/syscall.c

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
    /* 유저 스택에 저장되어 있는 시스템 콜 넘버를 이용해 시스템 콜
     * 핸들러 구현 */
    /* 스택 포인터가 유저 영역인지 확인 */
    /* 저장된 인자 값이 포인터일 경우 유저 영역의 주소인지 확인 */
    /* 0 : halt */
    /* 1 : exit */
    /* . . . */
}
```



시스템 콜 구현

`void halt(void)`

- ◆ pintos를 종료시키는 시스템 콜

`void exit(int status)`

- ◆ 현재 프로세스를 종료시키는 시스템 콜
- ◆ 종료 시 “프로세스 이름: exit(status)” 출력 (Process Termination Message)
- ◆ 정상적으로 종료 시 status는 0
- ◆ status : 프로그램이 정상적으로 종료됐는지 확인



시스템 콜 구현 (Cont.)

```
bool create (const char *file , unsigned initial_size)
```

- ◆ 파일을 생성하는 시스템 콜
- ◆ 성공 일 경우 true, 실패 일 경우 false 리턴
- ◆ file : 생성할 파일의 이름 및 경로 정보
- ◆ initial_size : 생성할 파일의 크기

```
bool remove (const char *file)
```

- ◆ 파일을 삭제하는 시스템 콜
- ◆ file : 제거할 파일의 이름 및 경로 정보
- ◆ 성공 일 경우 true, 실패 일 경우 false 리턴

시스템 콜 구현 시 사용되는 API

```
#include <devices/shutdown.h>
```

```
void shutdown_power_off(void)
```

- ◆ pintos를 종료시키는 함수

```
#include <threads/thread.h>
```

```
void thread_exit(void)
```

- ◆ 스레드를 종료시키는 함수

```
#include <filesys/filesys.h>
```

```
bool filesys_create(const char *name, off_t initial_size)
```

- ◆ 파일 이름과 파일 사이즈를 인자 값으로 받아 파일을 생성하는 함수

```
bool filesys_remove(const char *name)
```

- ◆ 파일 이름에 해당하는 파일을 제거하는 함수



halt(), exit() 시스템 콜 구현

pintos/src/userprog/syscall.c

```
void halt (void)
{
    /* shutdown_power_off()를 사용하여 pintos 종료 */
}
```

pintos/src/userprog/syscall.c

```
void exit (int status)
{
    /* 실행중인 스레드 구조체를 가져옴 */
    /* 프로세스 종료 메시지 출력,
       출력 양식: "프로세스이름: exit(종료상태)" */
    /* 스레드 종료 */
}
```



create(), remove() 시스템 콜 구현

pintos/src/userprog/syscall.c

```
bool create(const char *file , unsigned initial_size)
{
    /* 파일 이름과 크기에 해당하는 파일 생성 */
    /* 파일 생성 성공 시 true 반환, 실패 시 false 반환 */
}
```

pintos/src/userprog/syscall.c

```
bool remove(const char *file)
{
    /* 파일 이름에 해당하는 파일을 제거 */
    /* 파일 제거 성공 시 true 반환, 실패 시 false 반환 */
}
```

추가 함수

```
void check_address(void *addr)
/* 주소 값이 유저 영역에서 사용하는 주소 값인지 확인 하는 함수
Pintos에서는 시스템 콜이 접근할 수 있는 주소를 0x8048000~0xc0000000으로 제한함
유저 영역을 벗어난 영역일 경우 프로세스 종료(exit(-1)) */
```



```
void get_argument(void *esp, int *arg, int count)
/* 유저 스택에 있는 인자들을 커널에 저장하는 함수
스택 포인터(esp)에 count(인자의 개수) 만큼의 데이터를 arg에 저장 */
```



3. 프로세스 계층구조 구현

프로세스 계층구조 개요

▣ 과제 목표

- ◆ 프로세스간의 부모와 자식관계를 구현하고, 부모가 자식프로세스의 종료를 대기하는 기능 구현

▣ 과제 설명

- ◆ Pintos는 프로세스 구조체에 부모와 자식관계를 명시하는 정보가 없음
 - 부모와 자식의 구분이 없고, 자식 프로세스의 정보를 알지 못하기 때문에, 자식의 시작/종료 전에 부모 프로세스가 종료되는 현상이 발생 → 프로그램이 실행되지 않음

예: init 프로세스는 자식 프로세스의 정보를 알지 못하여, 유저 프로그램이 실행 되기 전에 Pintos를 종료



프로세스 계층구조 개요 (Cont.)

▣ 과제 설명(cont.)

- ◆ 프로세스 디스크립터(struct thread)에 부모와 자식필드를 추가하고, 이를 관리하는 함수를 구현
 - 부모 프로세스를 가리키는 포인터 추가
 - 자식 프로세스 : 리스트로 구현
 - 자식 리스트에서 원하는 프로세스를 검색, 삭제하는 함수 구현
- ◆ exec(), wait() 구현 (세마포어를 이용)

▣ 수정 파일

- ◆ pintos/src/threads/thread.*
- ◆ pintos/src/userprog/process.*
- ◆ pintos/src/userprog/syscall.*

유저 프로그램이 실행 되지 않는 원인

- init 프로세스가 유저 프로그램이 실행 되기 전 종료

pintos/src/threads/init.c

```
static void run_task(char ** argv)
{
    ...
    process_wait(process_execute(argv));
    ...
}
```

3. process_wait
호출

1. 유저 프로세스
생성

```
tid_t process_execute (const
char *file_name)
{
    ...
    tid = thread_create (...);
    ...
    return tid;
}
```

2. 생성 후
Ready List에 추가

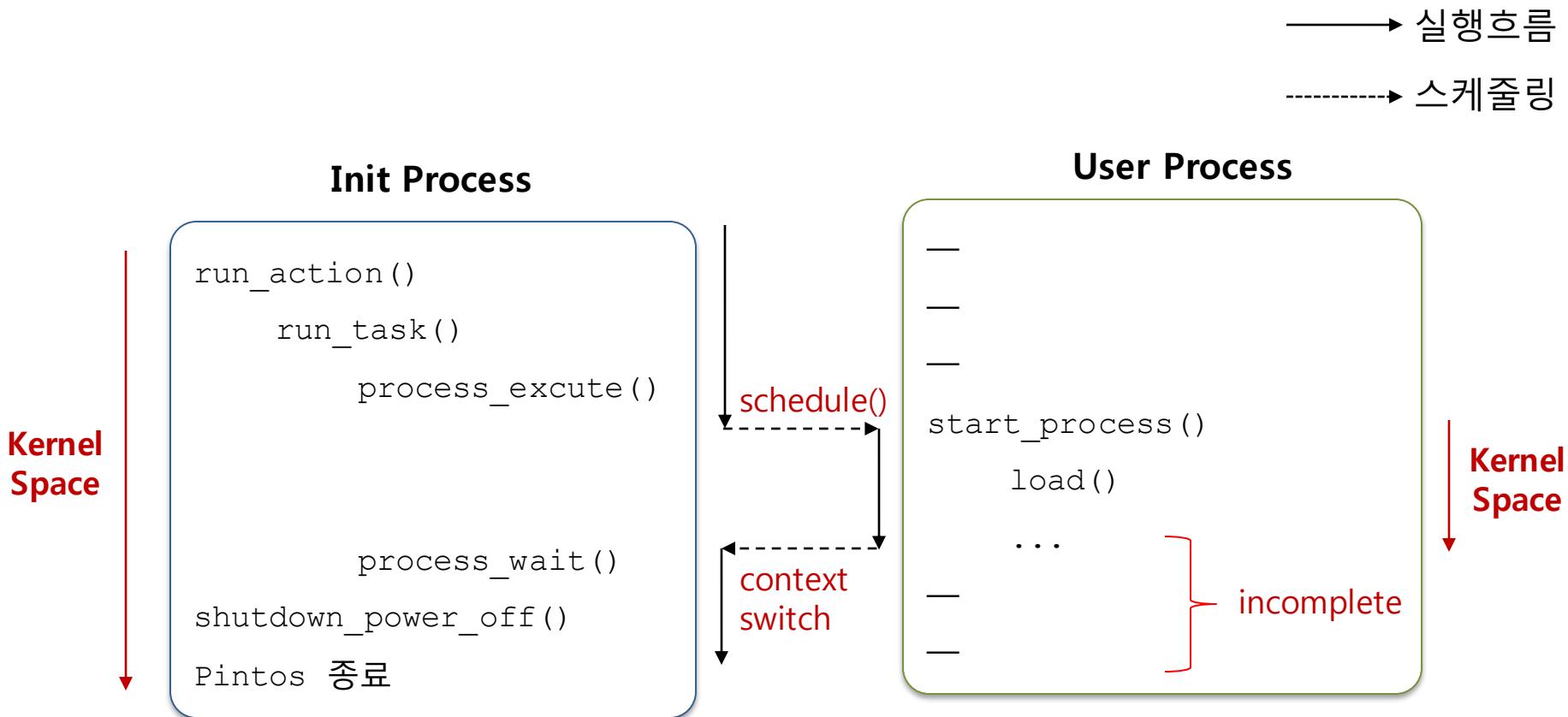
```
int process_wait (tid_t child_tid UNUSED)
{
    return -1;
}
```

4. Pintos 종료

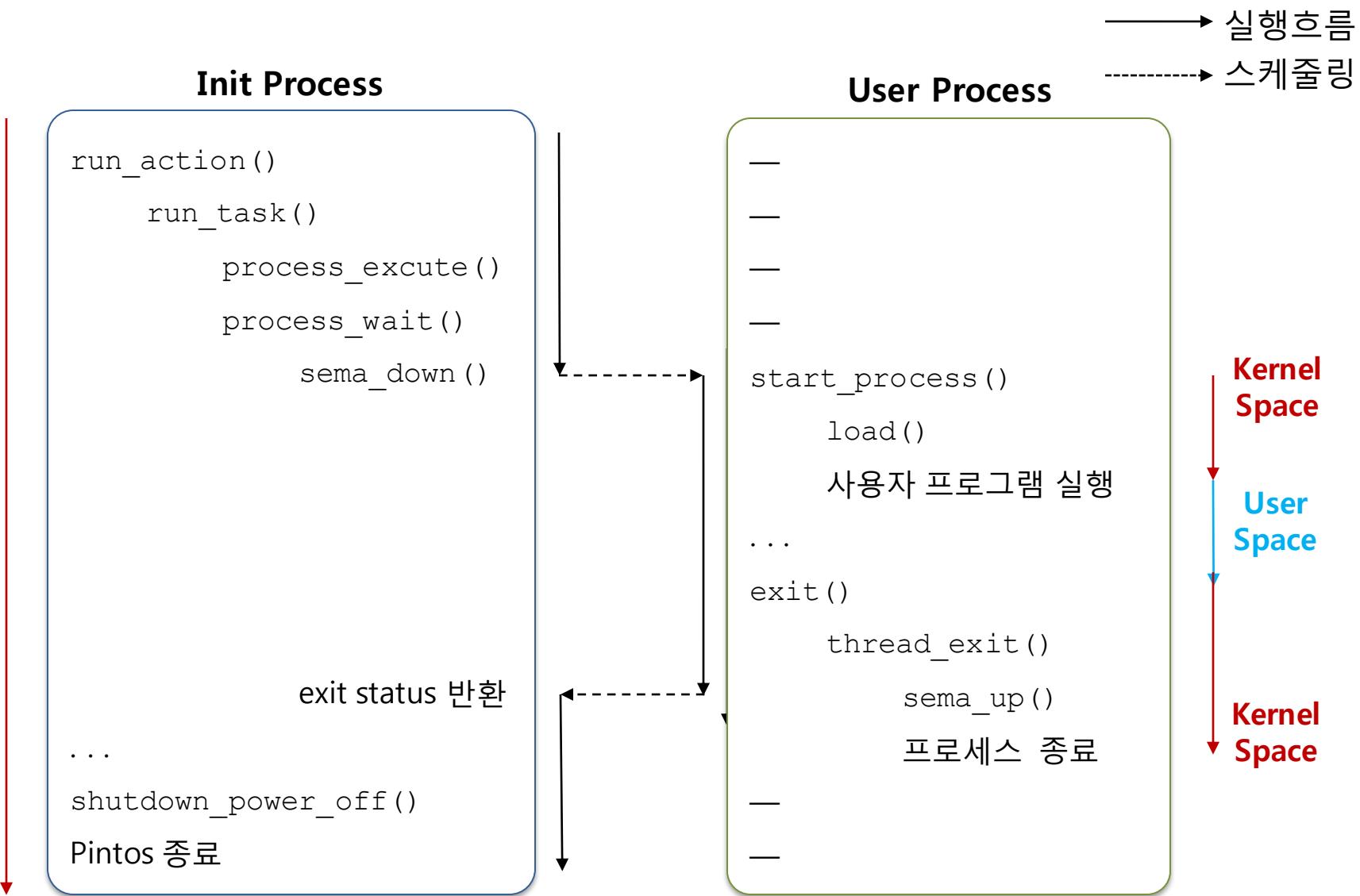
pintos/src/threads/init.c

```
int main(void)
{
    ...
    run_action(argv);
    shutdown_power_off();
    ...
}
```

기존 Pintos의 실행 흐름

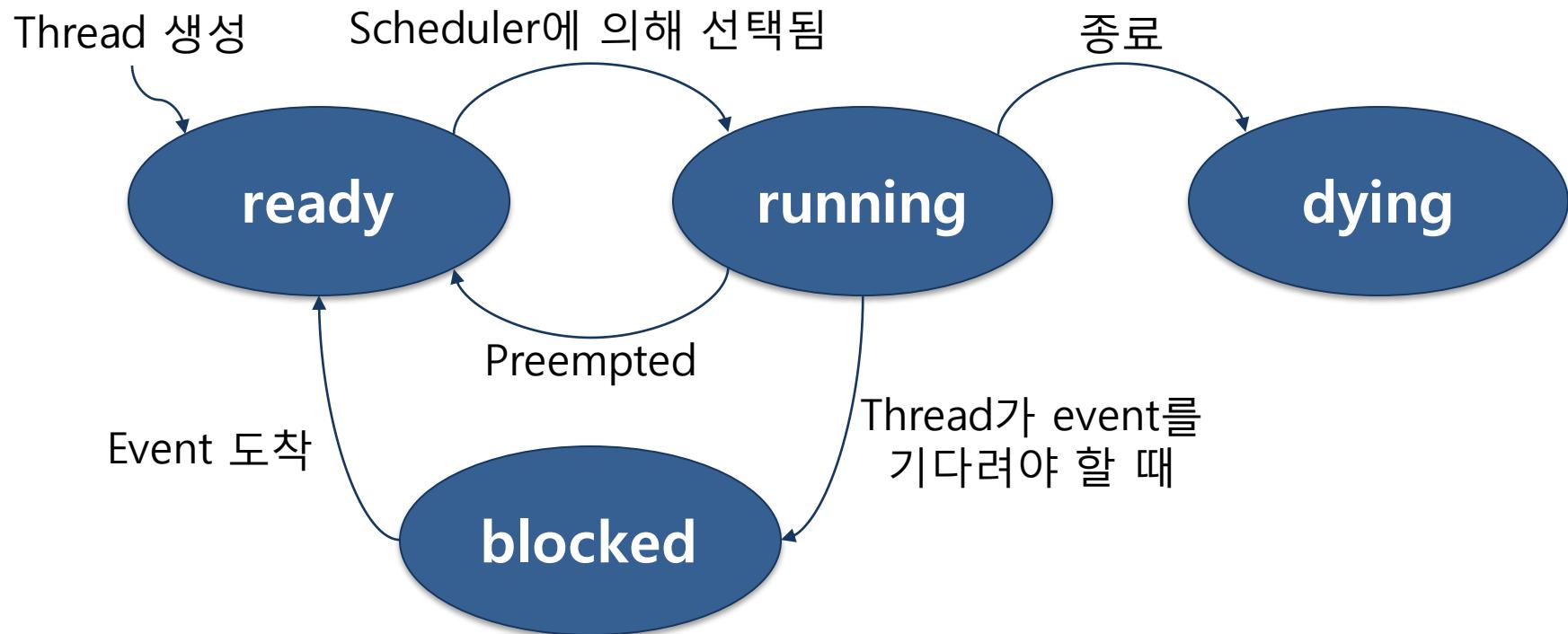


최종 목표



Pintos의 프로세스와 스레드

- ▣ Pintos의 프로세스는 1개의 스레드로 구성
- ▣ Pintos의 thread lifecycle



thread 구조체(프로세스 디스크립터)

pintos/src/threads/thread.h

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;
    enum thread_status status;
    char name[16];
    uint8_t *stack;
    int priority;
    struct list_elem allelem;

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;           /* List element. */

#ifndef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;              /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                 /* Detects stack overflow. */
};
```

enum thread_status

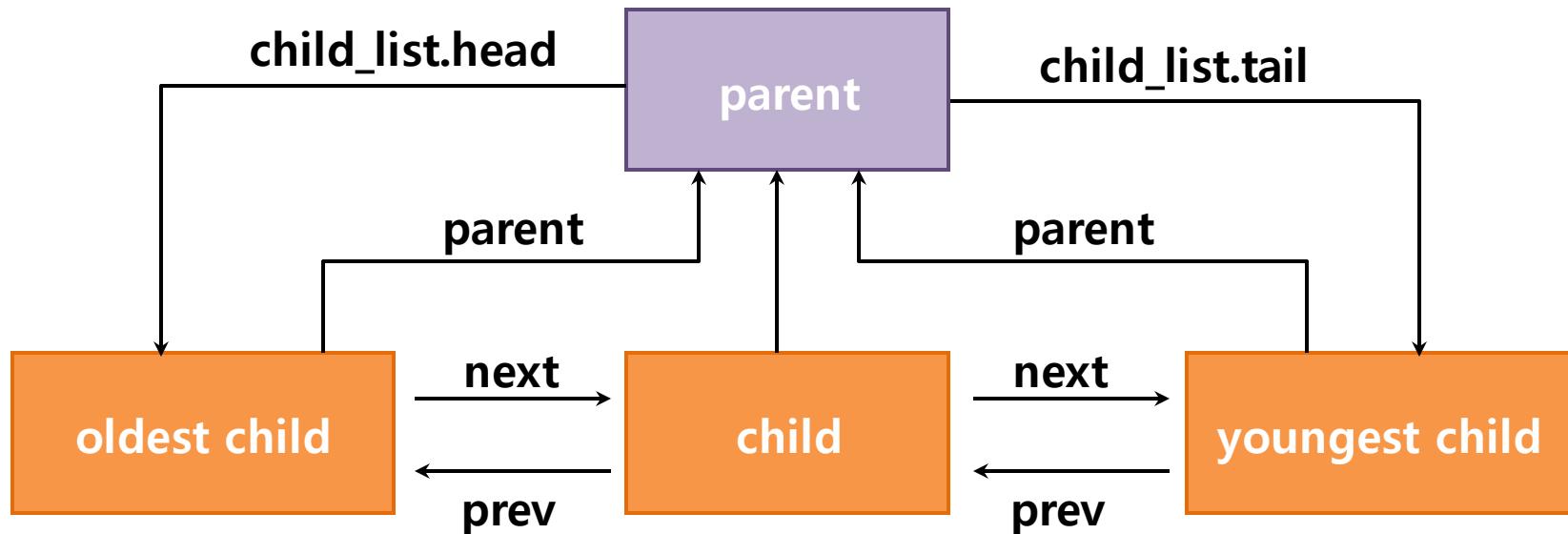
```
{
    THREAD_RUNNING,      /* Running thread. */
    THREAD_READY,        /* Not running but ready to run. */
    THREAD_BLOCKED,      /* Waiting for an event to trigger. */
    THREAD_DYING         /* About to be destroyed. */
};
```

/* Thread identifier. */
/* Thread state. */
/* Name (for debugging purposes). */
/* Saved stack pointer. */
/* Priority. */
/* List element for all threads list. */

프로세스 계층 구조 설계

▣ 부모 프로세스와 자식 프로세스

- ◆ 부모는 자식 프로세스 디스크립터들을 리스트를 이용하여 관리



프로세스 계층 구조 구현

- ▣ 프로세스 디스크립터(`struct thread`)에 정보 추가
 - ◆ 프로세스의 생성 성공 여부를 확인하는 플래그 추가 (실행 파일이 로드에 실패하면 -1)
 - ◆ 프로세스의 종료 유무를 확인하는 필드 추가
 - ◆ 프로세스의 종료 상태를 나타내는 필드 추가
 - ◆ 자식 프로세스의 생성/종료 대기를 위한 세마포어 추가
 - ◆ 자식 프로세스 리스트 필드 추가
 - ◆ 부모 프로세스 디스크립터를 가리키는 필드 추가



프로세스 계층 구조 구현 (Cont.)

▣ 프로세스 디스크립터에 프로세스의 정보 추가

pintos/src/threads/thread.h

```
struct thread
{
    ...
    /* 부모 프로세스의 디스크립터 */
    /* 자식 리스트 element */
    /* 자식 리스트 */

    /* 프로세스의 프로그램 메모리 적재 유무 */
    /* 프로세스가 종료 유무 확인 */
    /* exit 세마포어 */
    /* load 세마포어 */
    /* exit 호출 시 종료 status */
    ...
}
```



프로세스 계층 구조 구현 (Cont.)

▣ 자료구조 초기화

- ◆ 스레드 생성 시 자식 리스트 초기화

pintos/src/threads/thread.c

```
static void init_thread (struct thread *t, const char *name,
int priority)
{
    ...
    memset (t, 0, sizeof *t);
    t->status = THREAD_BLOCKED;
    strlcpy (t->name, name, sizeof t->name);
    t->stack = (uint8_t *) t + PGSIZE;
    t->priority = priority;
    t->magic = THREAD_MAGIC;
    list_push_back (&all_list, &t->allelem);

    /* 자식 리스트 초기화 */
}
```

프로세스 계층 구조 구현 (Cont.)

▣ 자료구조 초기화

- ◆ 생성된 프로세스 디스크립터의 정보를 초기화
- ◆ 부모 프로세스의 자식 리스트에 추가

pintos/src/threads/thread.c

```
tid_t thread_create (const char *name, int priority,
                      thread_func *function, void *aux)
{
    ...
    /* 부모 프로세스 저장 */
    /* 프로그램이 로드되지 않음 */
    /* 프로세스가 종료되지 않음 */
    /* exit 세마포어 0으로 초기화 */
    /* load 세마포어 0으로 초기화 */
    /* 자식 리스트에 추가 */

    /* Add to run queue. */
    thread_unblock (t);
}
```

자식 프로세스 검색 함수 구현

- ▣ 자식 리스트를 pid로 검색하여 해당 프로세스 디스크립터를 반환
 - ◆ pid가 없을 경우 NULL 반환

pintos/src/userprog/process.c

```
struct thread *get_child_process (int pid)
{
    /* 자식 리스트에 접근하여 프로세스 디스크립터 검색 */
    /* 해당 pid가 존재하면 프로세스 디스크립터 반환 */
    /* 리스트에 존재하지 않으면 NULL 리턴 */
}
```



자식 프로세스 제거 함수 구현

- ▣ 부모 프로세스의 자식 리스트에서 프로세스 디스크립터 제거
- ▣ 프로세스 디스크립터 메모리 해제

pintos/src/userprog/process.c

```
void remove_child_process(struct thread *cp)
{
    /* 자식 리스트에서 제거 */
    /* 프로세스 디스크립터 메모리 해제 */
}
```

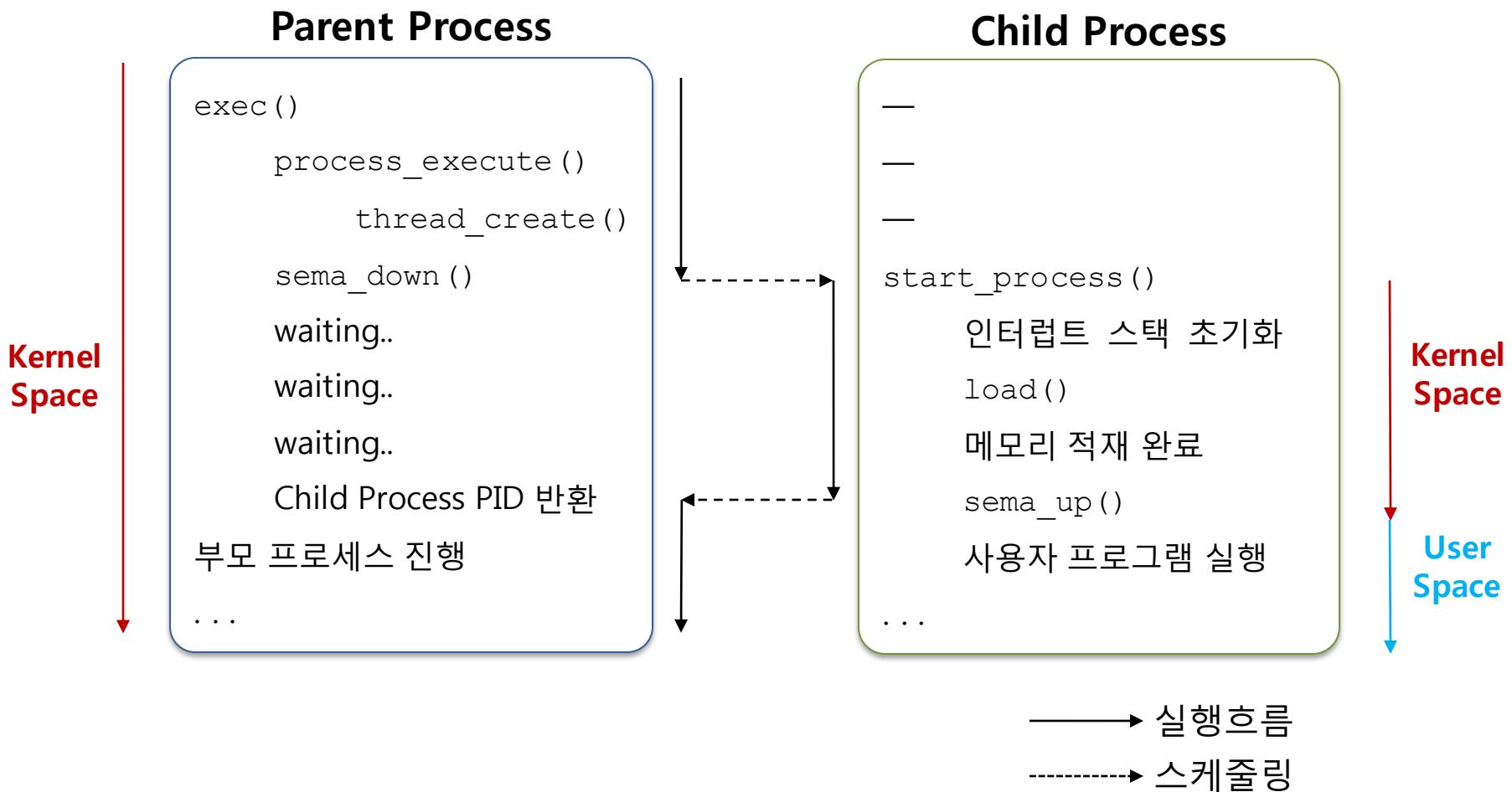
exec() 시스템 콜 구현

```
pid_t exec(const *cmd_line)
```

- ◆ 자식 프로세스를 생성하고 프로그램을 실행시키는 시스템 콜
- ◆ 프로세스 생성에 성공 시 생성된 프로세스에 pid 값을 반환, 실패 시 -1 반환
- ◆ 부모 프로세스는 생성된 자식 프로세스의 프로그램이 메모리에 적재 될 때까지 대기
 - 세마포어를 사용하여 대기
- ◆ cmd_line: 새로운 프로세스에 실행할 프로그램 명령어
- ◆ pid_t: int 자료형



exec() 시스템 콜 흐름



세마포어

- ▣ 공유 자원에 접근할 수 있는 프로세스의 수를 제한
 - ◆ 공유자원에 접근 가능한 프로세스 수를 초과한 경우 프로세스는 대기상태 진입
 - ◆ value : 현재 공유자원에 접근할 수 있는 프로세스의 수
 - ◆ waiter : 공유 자원 접근을 대기중인 프로세스의 리스트

pintos/src/threads/synch.h

```
struct semaphore
{
    unsigned value;                      /* Current value. */
    struct list waiters;                /* List of waiting threads. */
};
```



세마포어 관련 함수

- ▣ 프로세스가 대기 상태로 진입, 이탈할 수 있도록 세마포어 사용

pintos/src/threads/synch.h

```
void sema_init (struct semaphore *, unsigned value)
```

- ◆ 세마포어의 value 값을 초기화

```
void sema_down (struct semaphore *)
```

- ◆ 세마포어의 value가 0일 경우 현재 스레드를 THREAD_BLOCK 상태로 변경 후 schedule() 호출

```
void sema_up (struct semaphore *sema)
```

- ◆ 대기 리스트에 스레드가 존재하면 리스트 맨 처음에 위치한 스레드를 THREAD_READY 상태로 변경 후 schedule() 호출



exec() 시스템 콜 구현

```
pid_t exec(const cmd_line)
```

- ◆ 자식 프로세스를 생성하고 프로그램을 실행시키는 시스템 콜
- ◆ 프로세스 생성에 성공 시 생성된 프로세스에 pid 값을 반환, 실패 시 -1 반환
- ◆ 부모 프로세스는 자식 프로세스의 응용 프로그램이 메모리에 적재 될 때까지 대기
- ◆ cmd_line : 새로운 프로세스에 실행할 프로그램 명령어
- ◆ pid_t는 tid_t와 동일한 int 자료형

pintos/src/userprog/syscall.c

```
tid_t exec(const char *cmd_line)
{
    /* process_execute() 함수를 호출하여 자식 프로세스 생성 */
    /* 생성된 자식 프로세스의 프로세스 디스크립터를 검색 */
    /* 자식 프로세스의 프로그램이 적재될 때까지 대기 */
    /* 프로그램 적재 실패 시 -1 리턴 */
    /* 프로그램 적재 성공 시 자식 프로세스의 pid 리턴 */
}
```

부모 프로세스 대기 상태 이탈 구현

- ▣ 메모리 적재 완료 시 부모 프로세스 다시 진행 (세마포어 이용)
 - ◆ 메모리 적재 성공 시 유저 프로그램 실행, 실패 시 스레드 종료

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    ...
    success = load (file_name, &if_.eip, &if_.esp);

    /* 메모리 적재 완료 시 부모 프로세스 다시 진행 (세마포어 이용) */
    /* If load failed, quit. */
    palloc_free_page (file_name);
    if (!success)
        /* 메모리 적재 실패 시 프로세스 디스크립터에 메모리 적재 실패 */
        thread_exit ();
    /* 메모리 적재 성공 시 프로세스 디스크립터에 메모리 적재 성공 */
}

구현
구현
구현
```



시스템 콜 핸들러 설정

- exec 시스템 콜을 시스템 콜 핸들러에 추가 (예시)

pintos/src/userprog/syscall.c

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
    uint32_t *sp = f -> esp      /* 유저 스택 포인터 */
    check_address((void *)sp);
    int syscall_n = *sp;        /* 시스템 콜 넘버 */

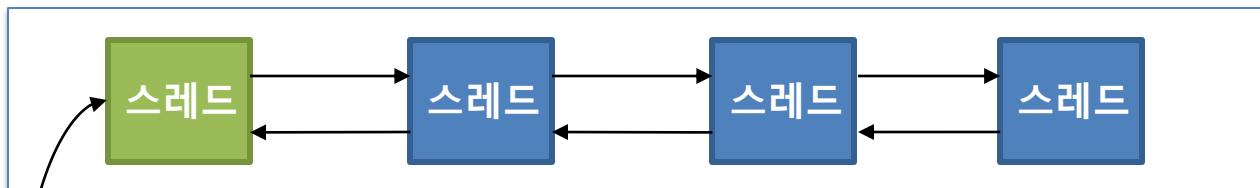
    switch(syscall_n)
    {
        ...
        case SYS_EXEC :
            get_argument(sp , arg , 1);
            check_address((void *)arg[0]);
            f -> eax = exec((const char *)arg[0]);
            break;
        ...
    }
}
```

wait() 구현

- ▣ 현재 process_wait()는 -1을 리턴
 - ◆ init 프로세스는 유저 프로세스가 종료될 때까지 대기하지 않고 Pintos 종료
- ▣ process_wait() 기능을 구현
 - ◆ 자식프로세스가 모두 종료될 때까지 대기(sleep state)
 - ◆ 자식 프로세스가 올바르게 종료 됐는지 확인
- ▣ wait() 시스템 콜 구현
 - ◆ process_wait() 함수를 호출

wait() 구현 (Cont.)

ready list



1. 프로세스 생성

자식 프로세스

Process 2

부모 프로세스

Process 1

2. ready 리스트
추가

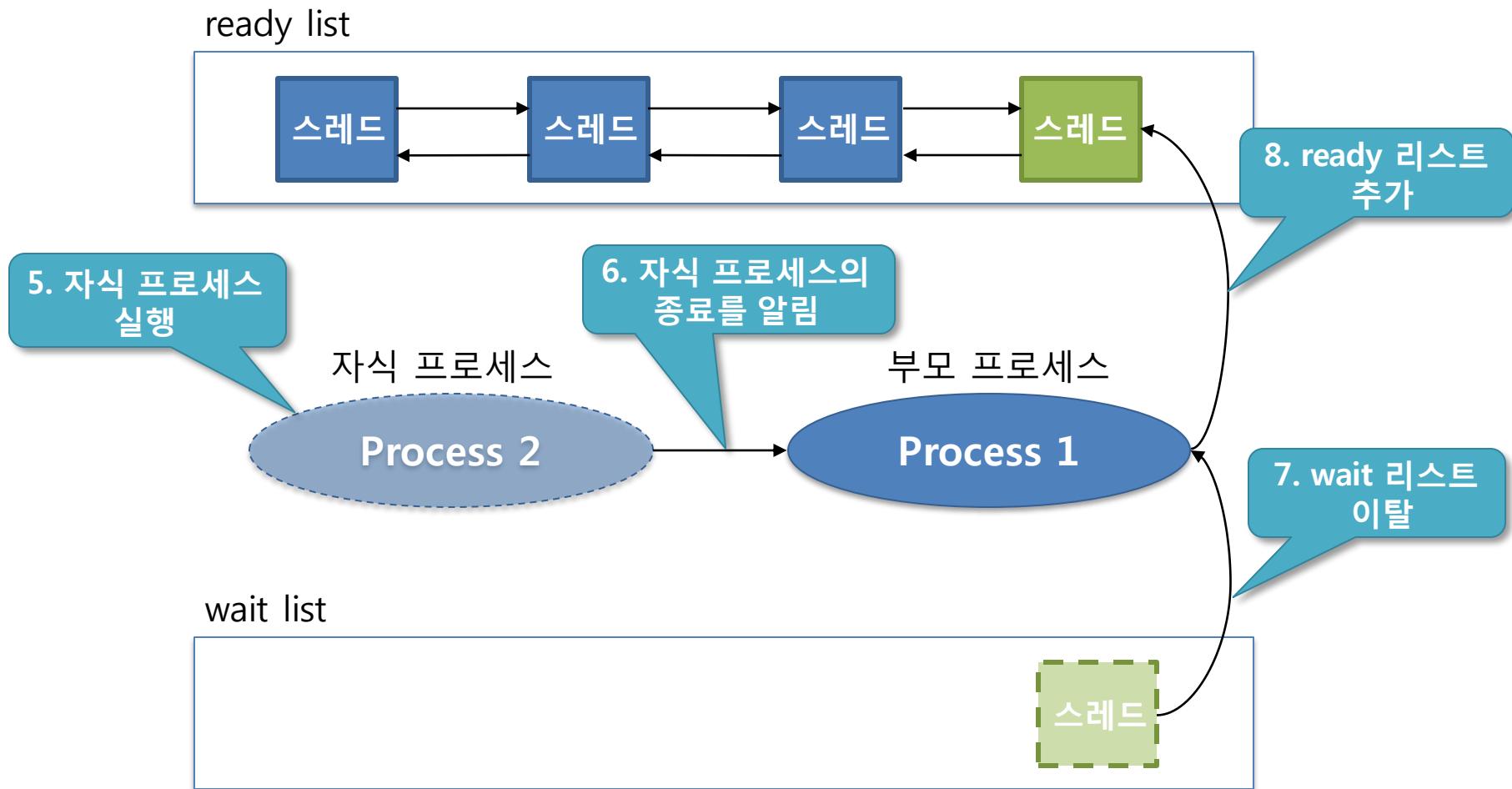
3. wait 호출

4. wait 리스트
추가

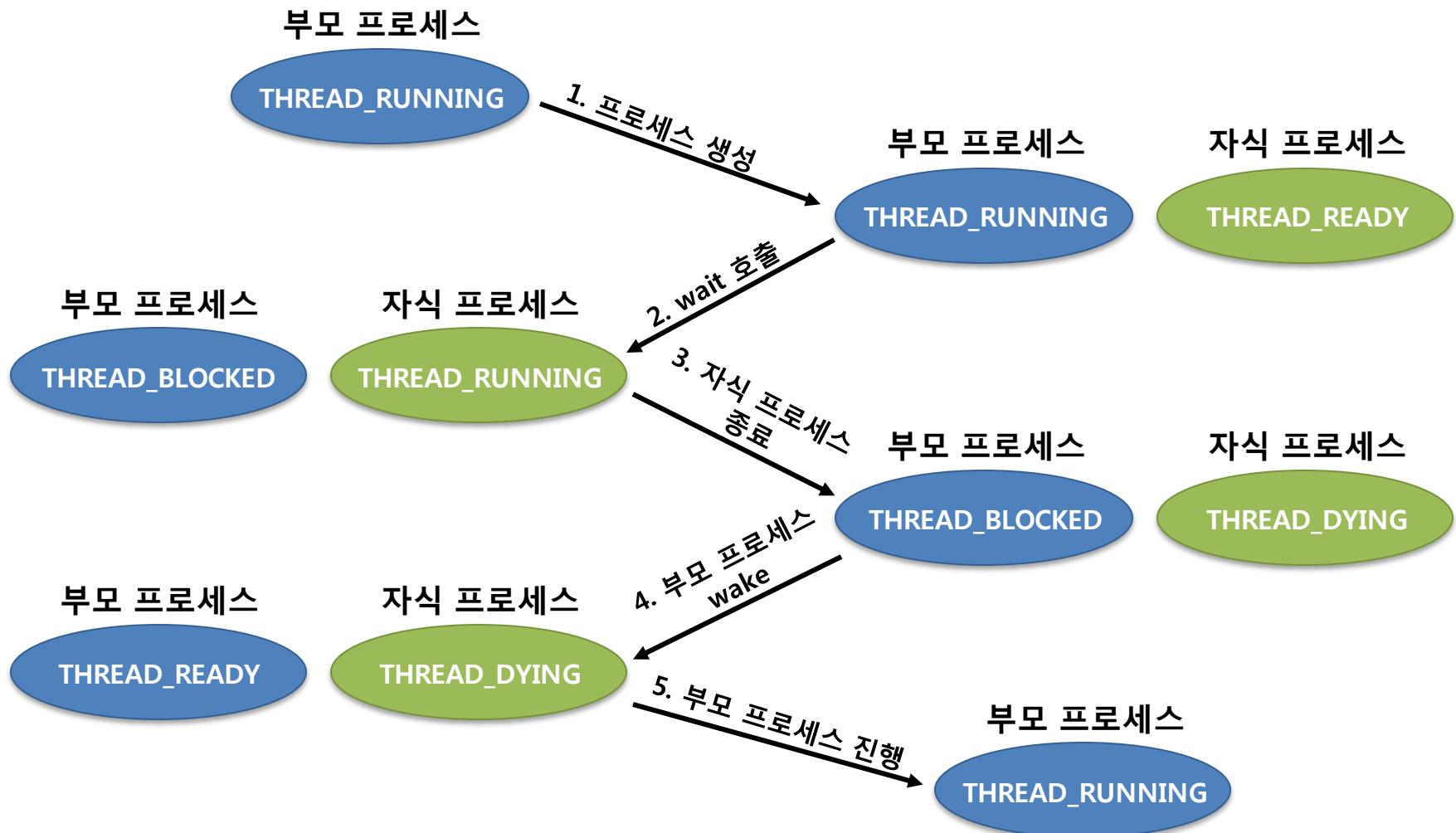
wait list



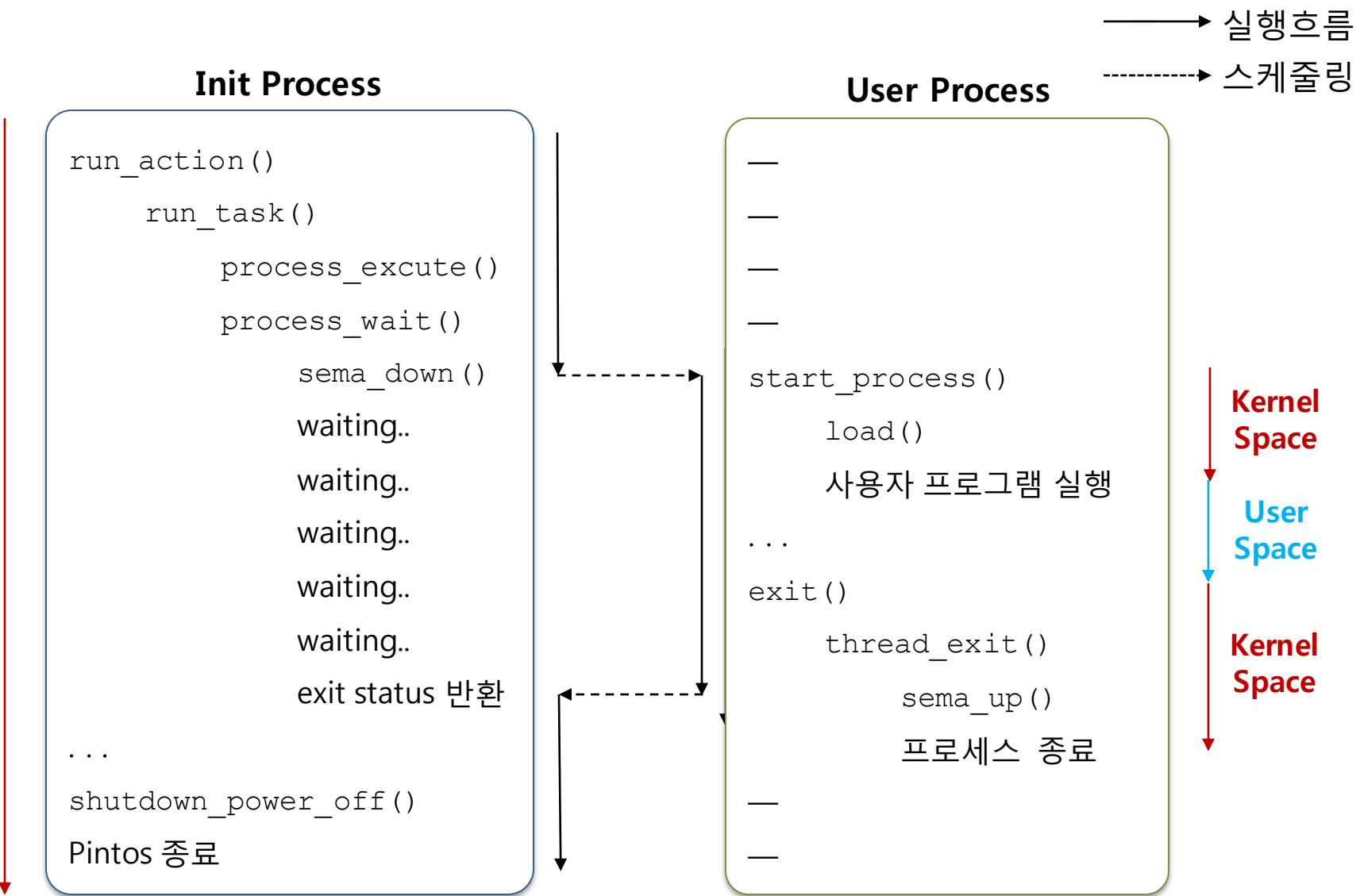
wait() 구현 (Cont.)



부모프로세스와 자식 프로세스의 상태



wait()을 사용한 유저 프로그램 실행 흐름



wait() 구현 (Cont.)

- ▣ 자식 프로세스가 수행 되고 종료될 때 까지 부모 프로세스 대기
 - ◆ 프로세스의 정보를 알기 위해 프로세스 디스크립터 검색
 - ◆ 자식 프로세스가 종료되지 않았으면 부모 프로세스 대기
 - ◆ 유저 프로세스가 정상적으로 종료 시 exit status 반환, 아닐 시 -1 반환

pintos/src/userprog/process.c

```
int process_wait (tid_t child_tid UNUSED)
{
    /* 자식 프로세스의 프로세스 디스크립터 검색 */
    /* 예외 처리 발생시 -1 리턴 */
    /* 자식프로세스가 종료될 때까지 부모 프로세스 대기 (세마포어 이용) */
    /* 자식 프로세스 디스크립터 삭제 */
    /* 자식 프로세스의 exit status 리턴 */
}
```

thread_exit() 수정

▣ 프로세스 종료 시 thread_exit() 호출

- ◆ 유저 프로세스가 종료되면 부모 프로세스는 대기 상태 이탈 후 진행
- ◆ 프로세스 디스크립터에 프로세스 종료 됨을 표시

pintos/src/threads/thread.c

```
void thread_exit (void)
{
    struct thread *t = thread_current ();
    ...
    list_remove (&t->allelem);
    /* 프로세스 디스크립터에 프로세스 종료를 알림 */
    /* 부모프로세스의 대기 상태 이탈(세마포어 이용) */
    t -> status = THREAD_DYING;
    schedule ();
    ...
}
```



thread_schedule_tail() 수정

- ▣ 프로세스 디스크립터를 삭제하지 않도록 수정

pintos/src/threads/thread.c

```
void thread_schedule_tail (struct thread *prev)
{
    struct thread *cur = running_thread ();
    ASSERT (intr_get_level () == INTR_OFF);

    /* Mark us as running. */
    cur->status = THREAD_RUNNING;
    ...
    if (prev != NULL && prev->status == THREAD_DYING &&
        prev != initial_thread)
    {
        ASSERT (prev != cur);
        palloc_free_page(prev); /* 프로세스 디스크립터 삭제 */
    }
}
```



exit() 시스템 콜 수정

- ▣ 프로그램 종료 시 exit() 시스템 콜을 호출
 - ◆ 정상적으로 종료가 됐는지 확인하기 위해 exit status 저장

pintos/src/userprog/syscall.c

```
void exit (int status)
{
    struct thread *cur = thread_current ();
    /* 프로세스 디스크립터에 exit status 저장 */
    printf ("%s: exit(%d)\n" , cur -> name , status);
    thread_exit ();
}
```



wait() 시스템 콜 구현

pintos/src/userprog/syscall.c

```
int wait (tid_t tid)
{
    /* 자식 프로세스가 종료 될 때까지 대기 */
    /* process_wait() 사용 */
}
```



Pintos의 list 자료 구조

pintos/src/lib/kernel/list.h

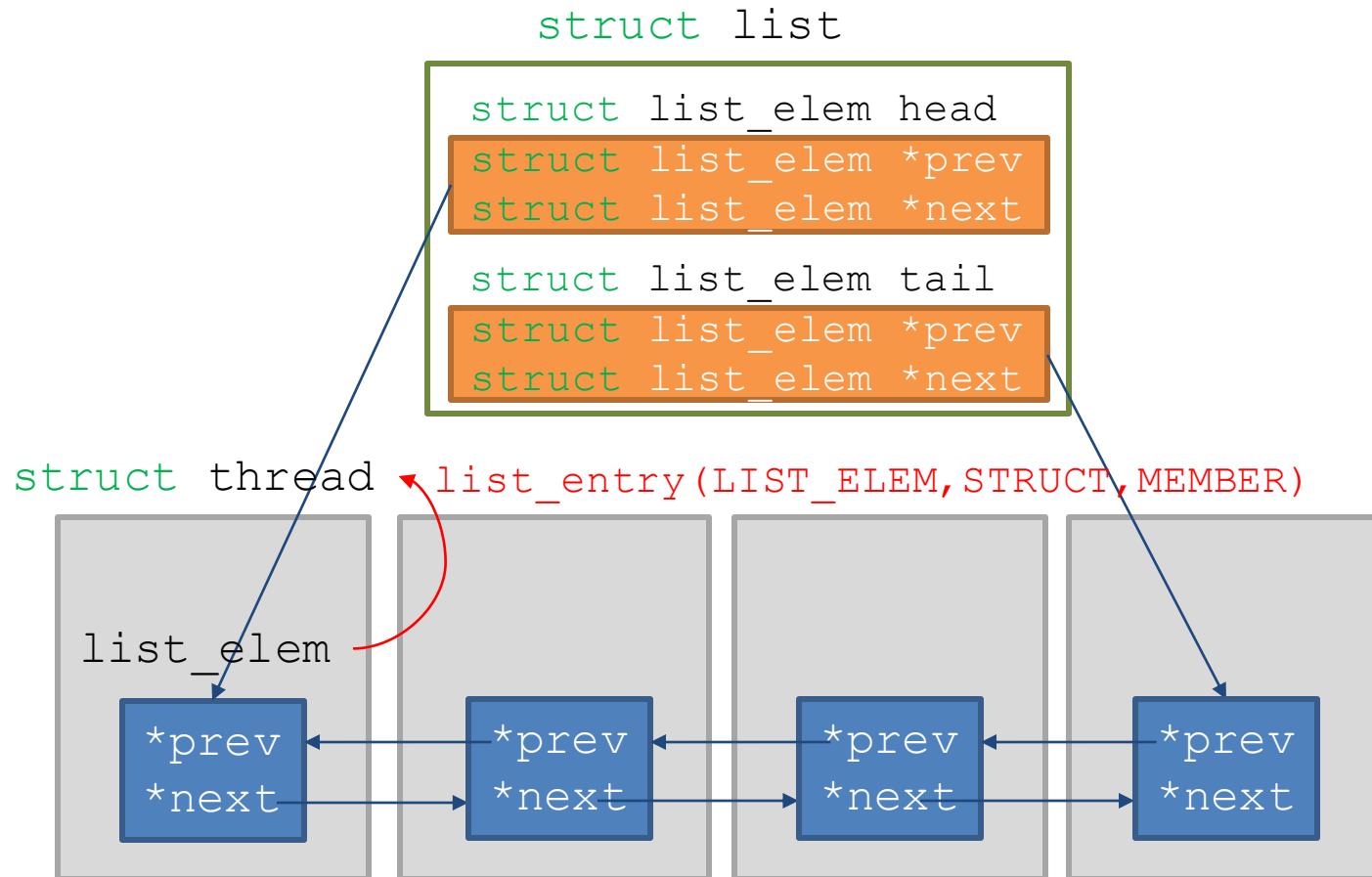
```
/* List element. */
struct list_elem
{
    struct list_elem *prev;          /* Previous list element. */
    struct list_elem *next;          /* Next list element. */
};

/* List. */
struct list
{
    struct list_elem head;          /* List head. */
    struct list_elem tail;          /* List tail. */
};
```



Pintos의 list 자료 구조 (Cont.)

- ▣ Pintos의 모든 스레드는 list로 관리



list 관련 함수 소개

▣ #include <list.h>

```
void list_init(struct list *list)
```

- ◆ list 자료 구조를 초기화

```
void list_push_back(struct list *list, struct list_elem *elem)
```

- ◆ elem을 list의 끝에 삽입

```
void list_push_front(struct list *list, struct list_elem *elem)
```

- ◆ elem을 list의 처음에 삽입

```
struct list_elem * list_pop_front(struct list *list)
```

- ◆ list의 처음 list_elem을 반환

```
struct list_elem * list_pop_back(struct list *list)
```

- ◆ list의 마지막 list_elem을 반환

```
#define list_entry(LIST_ELEM, STRUCT, MEMBER)
```

- ◆ list에서 해당 list_elem이 포함된 struct의 포인터를 반환

수정 함수

```
static void init_thread (struct thread *, const char *name,
                        int priority)

/* 프로세스 디스크립터 초기화 */

tid_t thread_create (const char *name, int priority,
                     thread_func *function, void *aux)

/* function 함수를 수행하는 스레드 생성 */

void thread_exit (void)

/* 현재 실행중인 스레드 종료 */

static void start_process (void *file_name_)

/* 프로그램을 메모리에 적재하고 응용 프로그램 실행 */
```



수정 및 추가 함수

```
int process_wait (tid_t child_tid UNUSED)
/* 자식 프로세스가 종료될 때까지 부모 프로세스 대기 */

void exit (int status)
/* 프로그램을 종료하는 시스템 콜 */

struct thread *get_child_process(int pid)
/* 자식 리스트를 검색하여 프로세스 디스크립터의 주소 리턴 */

void remove_child_process(struct thread *cp)
/* 프로세스 디스크립터를 자식 리스트에서 제거 후 메모리 해제 */
```



추가 함수 및 자료구조 수정

```
pid_t exec(const *cmd_line)  
/* 자식 프로세스를 생성하고 프로그램을 실행시키는 시스템 콜 */
```

```
int wait (tid_t tid)  
/* 자식 프로세스가 종료될 때 까지 대기하는 시스템 콜 */
```

```
void thread_schedule_tail (struct thread *prev)  
/* 프로세스를 스케줄링 하는 함수 */
```

- ▣ 프로세스 디스크립터 수정

```
struct thread  
/* 프로세스의 정보를 가진 자료구조 */
```

4. File Descriptor

File Descriptor 개요

▣ 과제 목표

- ◆ 파일 디스크립터 및 관련 시스템 콜 구현

▣ 과제 설명

- ◆ Pintos에는 파일 디스크립터 부분이 누락되어 있음. 파일 입출력을 위해서는 파일 디스크립터의 구현이 필수

▣ 수정 파일

- ◆ pintos/src/threads/thread.*
- ◆ pintos/src/userprog/process.*
- ◆ pintos/src/userprog/syscall.*



File Descriptor 개요 (Cont.)

▣ 과제를 해결하기 위해서는?

- ◆ 파일 디스크립터의 개념을 이해하고 구현 한다.
- ◆ 파일 시스템 관련 함수의 사용법을 익힌다.
- ◆ 시스템 콜의 요구사항을 분석하고 시스템 콜을 구현 한다.

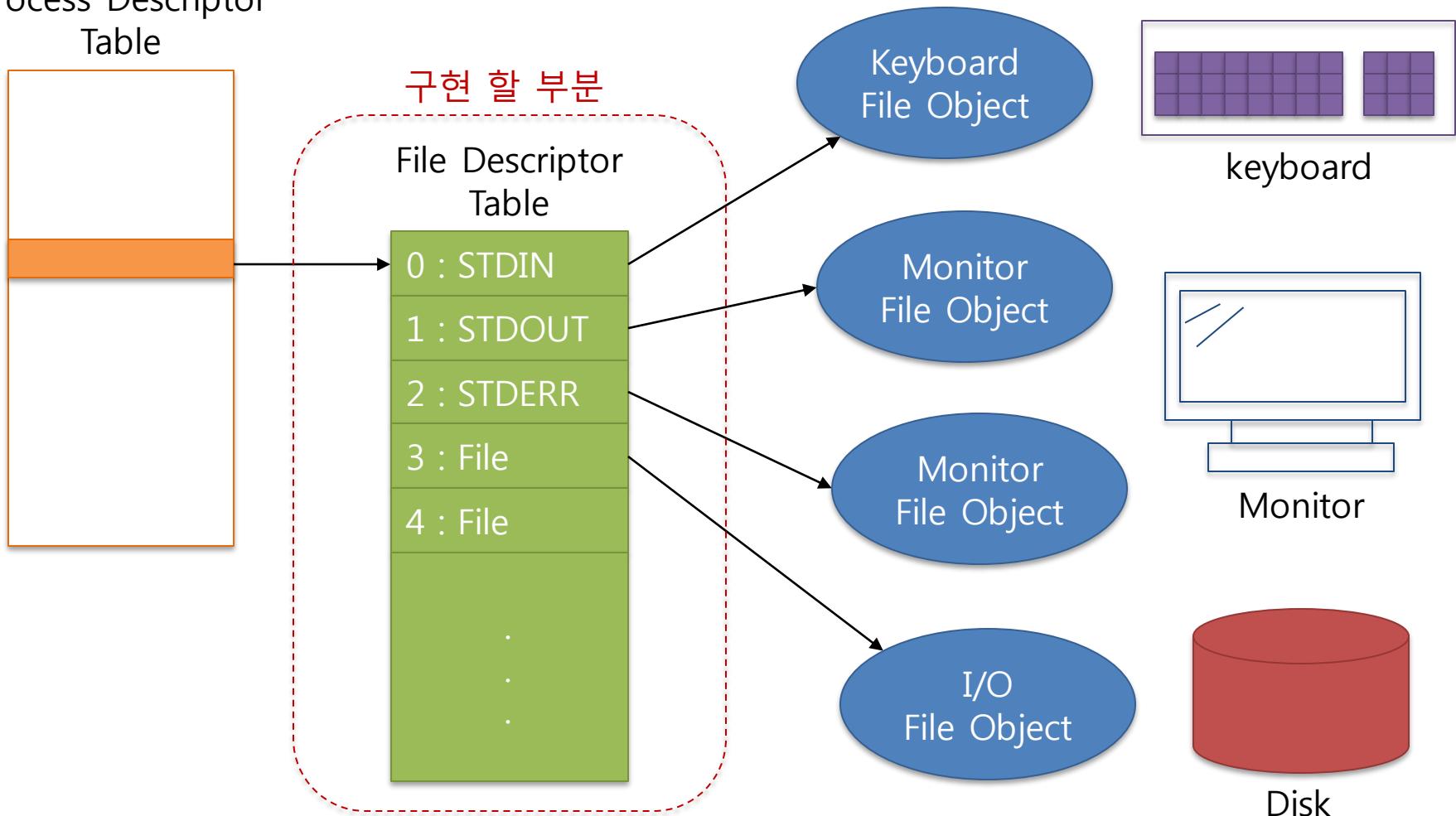


유닉스의 File Descriptor

- File Descriptor를 통해 파일에 접근

Process Descriptor Table

구현 할 부분



File Descriptor 구현

▣ File Descriptor 테이블 추가

- ◆ 파일 객체 포인터의 배열
- ◆ File Descriptor 등록 시 2부터 순차적으로 File Descriptor 1씩 증가
- ◆ open() 시 File Descriptor를 리턴
- ◆ close() 시 File Descriptor 테이블에 해당 엔트리 초기화
- ◆ 각 프로세스는 자신의 File Descriptor 테이블을 가지고 있음

File Descriptor 구현 (Cont.)

▣ 스레드 생성 시 File Descriptor를 초기화 하도록 수정

pintos/src/threads/thread.c

```
tid_t thread_create (const char *name, int priority,  
thread_func *function, void *aux)
```

- ◆ `struct thread`에 추가된 필드 초기화
- ◆ File Descriptor 테이블 메모리 할당

▣ 프로세스 종료 시 열린 파일을 모두 닫도록 수정

pintos/src/userprog/process.c

```
void process_exit (void)
```

- ◆ 프로세스 종료가 일어날 경우 프로세스에 열려있는 모든 파일을 닫음



File Descriptor 구현 (Cont.)

▣ File Descriptor 사용을 위한 함수 추가

pintos/src/userprog/process.c

```
int process_add_file (struct file *f)
```

- ◆ 파일 객체에 대한 파일 디스크립터 생성

```
struct file *process_get_file (int fd)
```

- ◆ 프로세스의 파일 디스크립터 테이블을 검색하여 파일 객체의 주소를 리턴

```
void process_close_file (int fd)
```

- ◆ 파일 디스크립터에 해당하는 파일을 닫고 해당 엔트리 초기화



File Descriptor 구현 (Cont.)

▣ File Descriptor 테이블 추가

- ◆ 파일 객체 포인터의 배열
- ◆ 현재 테이블에 할당된 파일 File Descriptor의 최대값 + 1

pintos/src/threads/thread.h

```
struct thread
{
    ...
    /* 파일 디스크립터 테이블 */
    /* 현재 테이블에 존재하는 fd값의 최대값 + 1 */
    ...
};
```



File Descriptor 구현 (Cont.)

▣ File Descriptor 초기화

- ◆ File Descriptor 초기 값을 2로 설정
- ◆ File Descriptor 테이블 메모리 할당

pintos/src/threads/thread.c

```
tid_t thread_create (const char *name, int priority,
                      thread_func *function, void *aux)
{
    /* fd 값 초기화(0,1은 표준 입력, 출력) */
    /* File Descriptor 테이블에 메모리 할당 */

    /* Add to run queue. */
    thread_unblock (t);
    ...
}
```

File Descriptor 구현 (Cont.)

▣ File Descriptor 생성

- ◆ 파일 객체(`struct file`)를 File Descriptor 테이블에 추가
- ◆ 프로세스의 File Descriptor의 최대값 1 증가
- ◆ 파일 객체의 File Descriptor 반환

pintos/src/userprog/process.c

```
int process_add_file (struct file *f)
{
    /* 파일 객체를 파일 디스크립터 테이블에 추가
     * 파일 디스크립터의 최대값 1 증가 */
    /* 파일 디스크립터 리턴 */
}
```

File Descriptor 구현 (Cont.)

▣ 파일 객체(`struct file`)를 검색

- ◆ File Descriptor값에 해당하는 파일 객체 반환. (File Descriptor 테이블 이용)
- ◆ 해당 테이블에 파일 객체가 없을 시 NULL 반환

pintos/src/userprog/process.c

```
struct file *process_get_file(int fd)
{
    /* 파일 디스크립터에 해당하는 파일 객체를 리턴 */
    /* 없을 시 NULL 리턴 */
}
```



File Descriptor 구현 (Cont.)

▣ 파일 닫기

- ◆ File Descriptor에 해당하는 파일 객체의 파일을 닫음
- ◆ File Descriptor 테이블에 해당 엔트리를 초기화

pintos/src/userprog/process.c

```
void process_close_file(int fd)
{
    /* 파일 디스크립터에 해당하는 파일을 닫음 */
    /* 파일 디스크립터 테이블 해당 엔트리 초기화 */
}
```



File Descriptor 구현 (Cont.)

▣ 모든 열린 파일 닫기

- ◆ 프로세스가 종료될 때 메모리 누수를 방지하기 위해 프로세스에 열린 모든 파일을 닫음
- ◆ File Descriptor 테이블 메모리 해제

pintos/src/userprog/process.c

```
void process_exit (void)
{
    struct thread *cur = thread_current ();
    uint32_t *pd;

    /* 프로세스에 열린 모든 파일을 닫음 */
    /* 파일 디스크립터 테이블의 최대값을 이용해 파일 디스크립터
    의 최소값인 2가 될 때까지 파일을 닫음 */
    /* 파일 디스크립터 테이블 메모리 해제 */
    ...
}
```



파일 시스템 관련 API

▣ 시스템 콜 구현에 필요한 함수

```
#include <filesys/filesys.h>
```

```
struct file *filesys_open(const char *name)
```

- ◆ 파일 이름에 해당하는 파일을 여는 함수

```
#include <filesys/file.h>
```

```
void file_close(struct file *)
```

- ◆ 파일을 닫는 함수

```
off_t file_read(struct file *, void *, off_t)
```

- ◆ 파일에 데이터를 읽는 함수

```
off_t file_write(struct file *, const void *, off_t);
```

- ◆ 파일에 데이터를 기록하는 함수

파일 시스템 관련 API (Cont.)

```
void file_seek(struct file *, off_t);
```

- ◆ 파일의 위치(offset)를 이동하는 함수

```
off_t file_tell(struct file *);
```

- ◆ 파일의 위치(offset)를 알려주는 함수

```
off_t file_length(struct file *);
```

- ◆ 파일의 크기를 알려주는 함수



파일 시스템 관련 API (Cont.)

- read(), write() 시스템 콜 구현 시 File Descriptor 0(표준 입력), File Descriptor 1(표준 출력)에 사용되는 함수

- ◆ read(), write() 시스템 콜은 pintos가 제공하는 file_read(), file_write() 인터페이스를 사용하여 구현

- #include <devices/input.h>

```
uint8_t input_getc (void)
```

- ◆ 키보드로 입력 받은 문자를 반환 하는 함수

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

```
void putbuf (const char *, size_t)
```

- ◆ 문자열을 화면에 출력해주는 함수

동기화 관련 API

- File에 대한 동시 접근이 일어날 수 있으므로 lock을 사용

pintos/src/threads/synch.h

```
struct lock
{
    struct thread *holder;          /* Thread holding lock */
    struct semaphore semaphore;     /* Binary semaphore
                                    controlling access. */
};
```

- ◆ `struct lock filesys_lock` 추가
 - userprog/syscall.h 헤더 파일에 전역변수로 추가
 - `read()`, `write()` 시스템 콜에서 파일 접근하기 전에 lock을 획득하도록 구현
 - 파일에 대한 접근이 끝난 뒤 lock 해제
- ◆ `syscall_init()` 함수에서 `filesys_lock` 초기화 코드 추가
 - userprog/syscall.c에 구현
 - `lock_init(struct lock*)` 인터페이스 사용

동기화 관련 API

pintos/src/threads/synch.h

```
void lock_init (struct lock *)
```

- ◆ lock을 초기화

```
void lock_acquire (struct lock *)
```

- ◆ 다른 프로세스가 접근하지 못하도록 lock을 잠금

```
void lock_release (struct lock *)
```

- ◆ 다른 프로세스가 접근할 수 있도록 lock을 해제



시스템 콜 설명

▣ int open (const char *file)

- ◆ 파일을 열 때 사용하는 시스템 콜
- ◆ 성공 시 fd를 생성하고 반환, 실패 시 -1 반환
- ◆ File : 파일의 이름 및 경로 정보

```
int filesize (int fd)
```

- ◆ 파일의 크기를 알려주는 시스템 콜
- ◆ 성공 시 파일의 크기를 반환, 실패 시 -1 반환

open(), filesize() 시스템 콜 구현

pintos/src/userprog/syscall.c

```
int open(const char *file)
{
    /* 파일을 open */
    /* 해당 파일 객체에 파일 디스크립터 부여 */
    /* 파일 디스크립터 리턴 */
    /* 해당 파일이 존재하지 않으면 -1 리턴 */
}
```

pintos/src/userprog/syscall.c

```
int filesize (int fd)
{
    /* 파일 디스크립터를 이용하여 파일 객체 검색 */
    /* 해당 파일의 길이를 리턴 */
    /* 해당 파일이 존재하지 않으면 -1 리턴 */
}
```

시스템 콜 설명 (Cont.)

```
int read (int fd , void *buffer unsigned size)
```

- ◆ 열린 파일의 데이터를 읽는 시스템 콜
- ◆ 성공 시 읽은 바이트 수를 반환, 실패 시 -1 반환
- ◆ buffer: 읽은 데이터를 저장할 버퍼의 주소 값
- ◆ size: 읽을 데이터 크기
- ◆ fd 값이 0일 때 키보드의 데이터를 읽어 버퍼에 저장. (input_getc() 이용)

```
int write (int fd , void *buffer unsigned size)
```

- ◆ 열린 파일의 데이터를 기록 시스템 콜
- ◆ 성공 시 기록한 데이터의 바이트 수를 반환, 실패시 -1 반환
- ◆ buffer: 기록 할 데이터를 저장한 버퍼의 주소 값
- ◆ size: 기록할 데이터 크기
- ◆ fd 값이 1일 때 버퍼에 저장된 데이터를 화면에 출력. (putbuf() 이용)



read(), write() 시스템 콜 구현

pintos/src/userprog/syscall.c

```
int read (int fd, void *buffer, unsigned size)
{
    /* 파일에 동시 접근이 일어날 수 있으므로 Lock 사용 */
    /* 파일 디스크립터를 이용하여 파일 객체 검색 */
    /* 파일 디스크립터가 0일 경우 키보드에 입력을 버퍼에 저장 후
    버퍼의 저장한 크기를 리턴 (input_getc() 이용) */
    /* 파일 디스크립터가 0이 아닐 경우 파일의 데이터를 크기만큼 저
    장 후 읽은 바이트 수를 리턴 */
}
```

pintos/src/userprog/syscall.c

```
int write(int fd, void *buffer, unsigned size)
{
    /* 파일에 동시 접근이 일어날 수 있으므로 Lock 사용 */
    /* 파일 디스크립터를 이용하여 파일 객체 검색 */
    /* 파일 디스크립터가 1일 경우 버퍼에 저장된 값을 화면에 출력
    후 버퍼의 크기 리턴 (putbuf() 이용) */
    /* 파일 디스크립터가 1이 아닐 경우 버퍼에 저장된 데이터를 크기
    만큼 파일에 기록후 기록한 바이트 수를 리턴 */
}
```

시스템 콜 설명 (Cont.)

```
void seek (int fd , unsigned position)
```

- ◆ 열린 파일의 위치(offset)를 이동하는 시스템 콜
- ◆ Position: 현재 위치(offset)를 기준으로 이동할 거리

```
unsigned tell (int fd)
```

- ◆ 열린 파일의 위치(offset)를 알려주는 시스템 콜
- ◆ 성공 시 파일의 위치(offset)를 반환, 실패 시 -1 반환

```
void close (int fd)
```

- ◆ 열린 파일을 닫는 시스템 콜
- ◆ 파일을 닫고 File Descriptor를 제거

seek(), tell(), close() 시스템 콜 구현

pintos/src/userprog/syscall.c

```
void seek (int fd, unsigned position)
{
    /* 파일 디스크립터를 이용하여 파일 객체 검색 */
    /* 해당 열린 파일의 위치(offset)를 position만큼 이동 */
}
```

pintos/src/userprog/syscall.c

```
unsigned tell (int fd)
{
    /* 파일 디스크립터를 이용하여 파일 객체 검색 */
    /* 해당 열린 파일의 위치를 반환 */
}
```

pintos/src/userprog/syscall.c

```
void close (int fd)
{
    /* 해당 파일 디스크립터에 해당하는 파일을 닫음 */
    /* 파일 디스크립터 엔트리 초기화 */
}
```



Page Fault 수정

- Stack pointer(%esp)가 가리키는 주소에서 Page fault가 발생할 경우, exit(-1) 시스템 콜을 호출 하도록 수정
 - Stack pointer(%esp) 주소를 임의 위치로 변경하는 테스트 (ex. child-bad.c)의 pass를 위해 수정 필요
 - Page fault의 관한 자세한 내용은 project 3에서 다룬다.

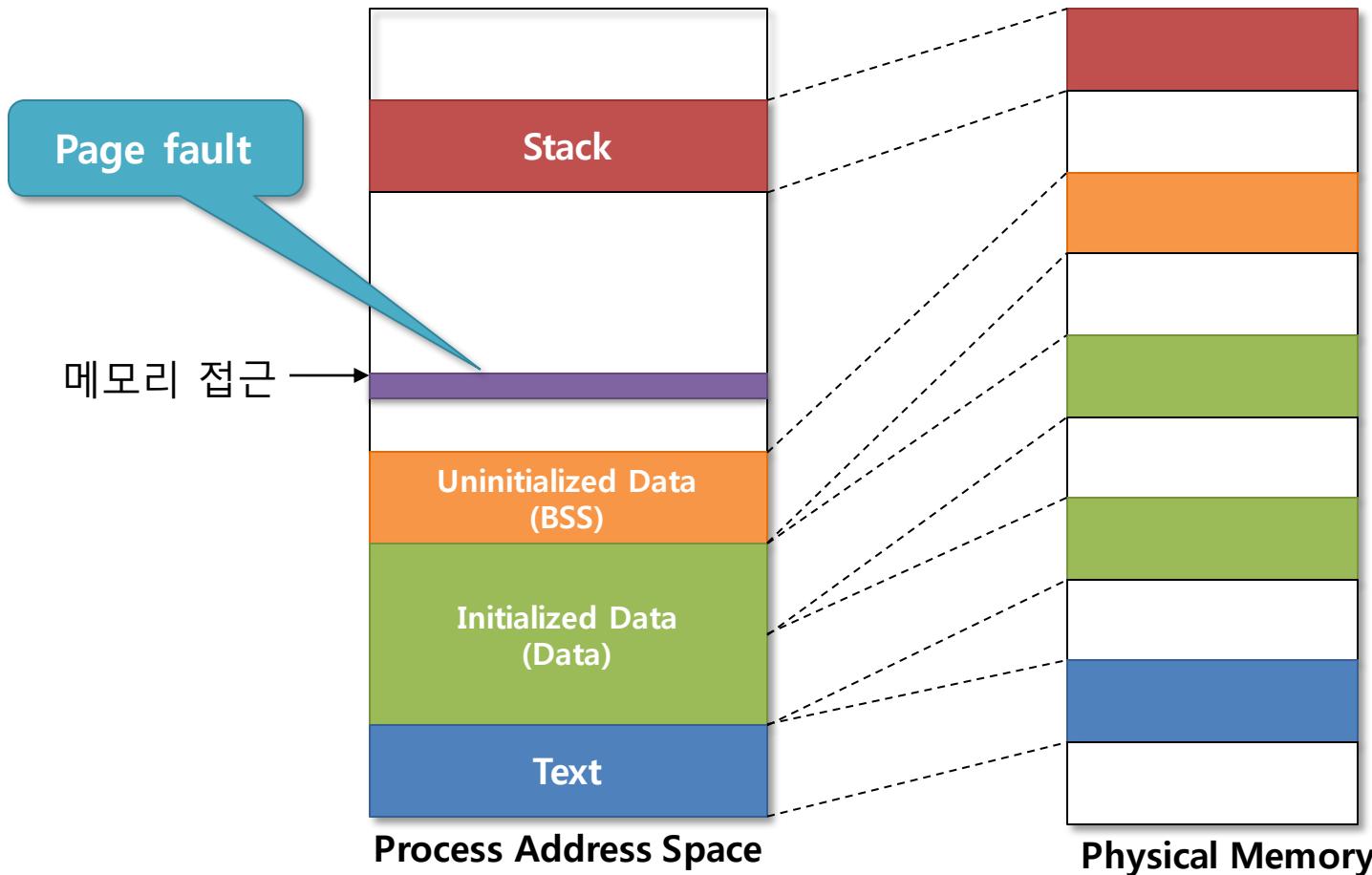
pintos/src/userprog/exception.c

```
static void page_fault (struct intr_frame *f)
{
    ...
    not_present = (f->error_code & PF_P) == 0;
    write = (f->error_code & PF_W) != 0;
    user = (f->error_code & PF_U) != 0;

    /* 페이지 폴트 발생 시 exit(-1) 호출 */
    ...
}
```

Page Fault

- 유저 영역의 주소이지만 물리 메모리에 존재하지 않으므로 page fault 발생



수정 및 추가 함수

```
tid_t thread_create (const char *name, int priority,  
                     thread_func *function, void *aux)  
  
/* function 함수를 수행하는 스레드 생성 */  
  
void process_exit (void)  
  
/* 프로세스 종료 시 호출되어 프로세스의 자원을 해제 */  
  
int process_add_file (struct file *f)  
  
/* 파일 객체에 대한 파일 디스크립터 생성 */  
  
struct file *process_get_file (int fd)  
  
/* 프로세스의 파일 디스크립터 테이블을 검색하여 파일 객체의 주소를 리턴 */
```



추가 함수

```
void process_close_file (int fd)
    /* 파일 디스크립터에 해당하는 파일을 닫고 해당 엔트리 초기화 */

int open (const char *file)
    /* 파일을 열 때 사용하는 시스템 콜 */

int filesize (int fd)
    /* 파일의 크기를 알려주는 시스템 콜 */

int read (int fd , void *buffer unsigned size)
    /* 열린 파일의 데이터를 읽는 시스템 콜 */

int write (int fd , void *buffer unsigned size)
    /* 열린 파일의 데이터를 기록 시스템 콜 */
```



추가 함수 및 자료구조 수정

```
void seek (int fd , unsigned position)  
/* 열린 파일의 위치(offset)를 이동하는 시스템 콜 */  
  
unsigned tell (int fd)  
/* 열린 파일의 위치(offset)를 알려주는 시스템 콜 */  
  
void close (int fd)  
/* 열린 파일을 닫는 시스템 콜 */
```

▣ 자료구조 수정

```
struct thread  
/* 프로세스의 정보를 가진 자료구조 */
```



5. Denying Write to Executable

Denying Writes to Executables

▣ 과제 목표

- ◆ 실행 중인 파일에 데이터를 기록하는 것을 방지

▣ 과제 설명

- ◆ 실행중인 사용자 프로그램의 파일 데이터가 변경되면 프로그램이 원래 예상했던 데이터와 다르게 변경된 데이터를 읽어올 가능성이 있다.
- ◆ 이런 현상이 발생하면 되면 현재 실행중인 프로그램이 올바른 연산 결과를 도출할 수 없기 때문에 OS 차원에서 실행 중인 유저 프로그램에 대한 쓰기 접근을 막아야 한다.

Denying Writes to Executables

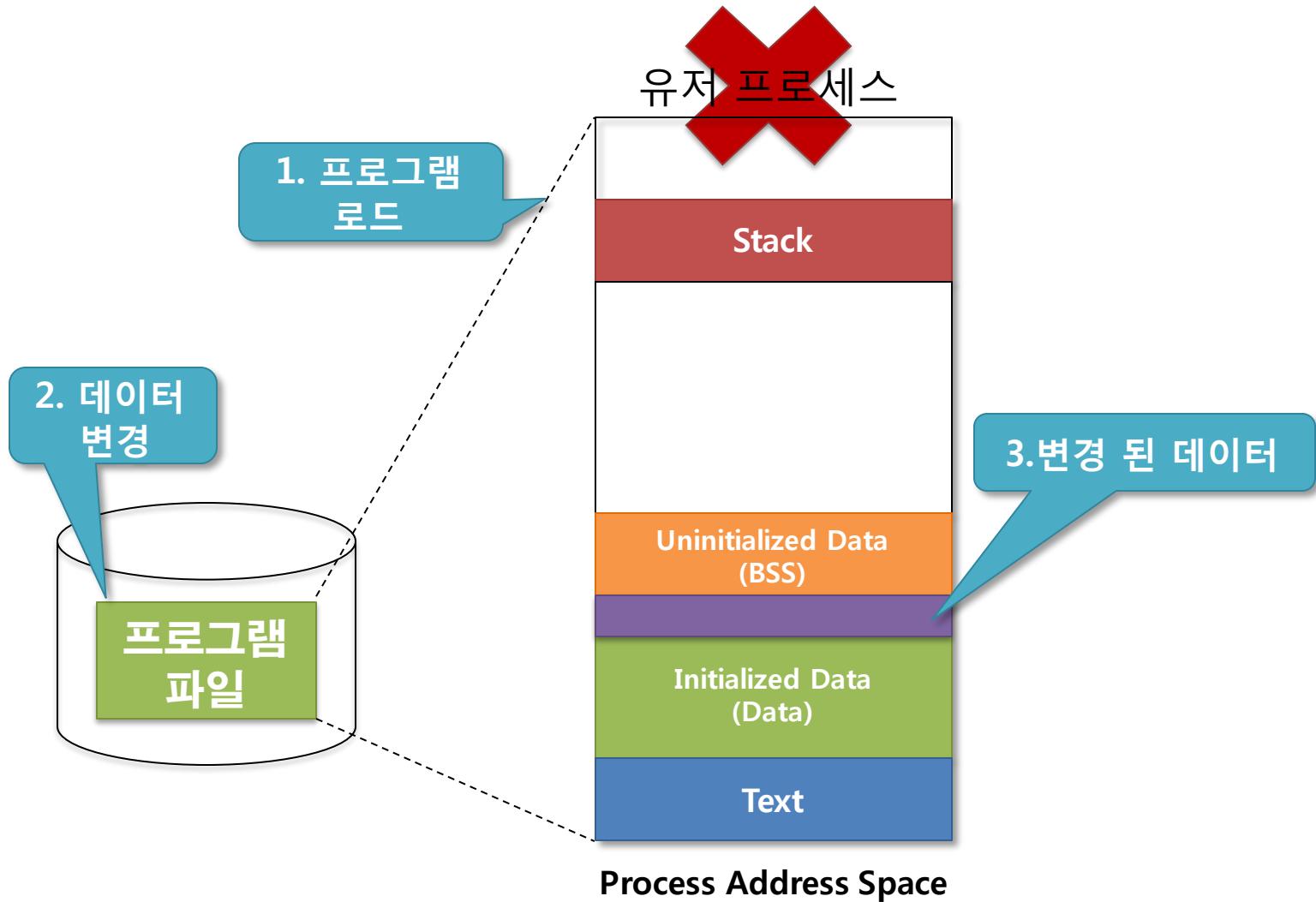
▣ 과제를 해결하기 위해서는?

- ◆ 파일이 실행되는 위치를 정확히 파악 해야 한다.
- ◆ 실행 중인 프로그램 파일의 데이터가 변경되는 것을 예방할 수 있도록 수정
 - pintos에서는 데이터 변경을 막는 함수가 이미 구현되어 있음
- ◆ 프로그램이 종료가 되었을 경우에만 실행중인 파일의 데이터가 변경될 수 있도록 수정



Denying Writes to Executables

- 변경된 데이터로 인해 프로그램이 정상적으로 동작하지 않음



Denying Writes to Executables 구현

- ▣ 구현 요구 사항: 메모리에 적재된 프로그램의 디스크 상의 파일이 변경되는 것을 방지
 - ◆ 메모리에 프로그램 적재 시, 프로그램 파일에 쓰기 권한 제거
 - `file_deny_write()` 인터페이스 사용
 - ◆ 프로그램이 종료되었을 때, 프로그램 파일에 쓰기 권한 부여
 - `file_allow_write()` 인터페이스 사용



Denying Writes to Executables 관련 함수

▣ 파일의 데이터변경 예방과 허락을 위한 함수

```
#include <filesys/file.h>
```

```
void file_deny_write (struct file *)
```

- ◆ 열린 파일의 데이터가 변경되는 것을 예방

```
void file_allow_write (struct file *)
```

- ◆ 파일의 데이터가 변경되는 것을 허락
- ◆ `file_close()` 함수 호출 시 해당 함수가 호출됨



Denying Writes to Executables 구현

pintos/src/userprog/process.c

```
static bool load (const char * cmdline, void (**eip) (void),  
void **esp)
```

- ◆ 프로그램의 파일을 open 할 때 file_deny_write() 함수를 호출
- ◆ 실행중인 파일 구조체를 thread 구조체에 추가

```
void process_exit (void)
```

- ◆ 현재 프로세스가 실행 중인 파일을 닫도록 수정

실행중인 파일에 대한 포인터 추가

- 현재 실행중인 파일의 정보를 수정 할 수 있도록 thread 구조체에 추가

pintos/src/thread/thread.h

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;                                /* Thread identifier. */
    enum thread_status status;                 /* Thread state. */
    ...
    struct list file_list;
    /* 현재 실행중인 파일 추가 */

    /* Owned by thread.c. */
    unsigned magic;                            /* Detects stack overflow. */
};
```



실행 중인 파일의 데이터 변경 예방

- 실행 할 파일을 열 때 데이터 변경을 예방 하도록 수정

pintos/src/userprog/process.c

```
bool load (const char *file_name, void (**eip) (void), void **esp)
{
    struct thread *t = thread_current ();
    struct Elf32_Ehdr ehdr;

    /* 락 획득 */
    /* Open executable file. */
    file = filesys_open (file_name);
    if (file == NULL)
    {
        /* 락 해제 */
        printf ("load: %s: open failed\n", file_name);
        goto done;
    }
    /* thread 구조체의 run_file을 현재 실행할 파일로 초기화 */
    /* file_deny_write()를 이용하여 파일에 대한 write를 거부 */
    /* 락 해제 */
    ...
}
```

실행 중인 파일의 데이터 변경 허락

- ▣ 프로세스 종료 시 실행 중인 파일의 데이터가 변경 될 수 있도록 수정

pintos/src/userprog/process.c

```
void process_exit (void)
{
    struct thread *cur = thread_current ();
    uint32_t *pd;

    /* Destroy the current process's page directory and switch
back to the kernel-only page directory. */
    /* 실행 중인 파일 close */
    ...
}
```



결과

▣ 모든 과정 테스트 결과 확인

- ◆ 경로 : pintos/src/userprog

```
$make grade
```

SUMMARY BY TEST SET				
Test Set	Pts	Max	% Ttl	% Max
tests/userprog/Rubric.functionality	108/108	35.0%	/ 35.0%	
tests/userprog/Rubric.robustness	88/ 88	25.0%	/ 25.0%	
tests/userprog/no-vm/Rubric	1/ 1	10.0%	/ 10.0%	
tests/filesys/base/Rubric	30/ 30	30.0%	/ 30.0%	
Total	100.0%/100.0%			

6. Alarm System Call

Alarm 개요

▣ 과제 목표

- ◆ Alarm : 호출한 프로세스를 정해진 시간 후에 다시 시작하는 커널 내부 함수
- ◆ Pintos에서는 알람 기능이 Busy waiting을 이용하여 구현되어 있음
- ◆ 본 과제에서는 알람 기능을 sleep/wake up을 이용하여 다시 구현 한다.

▣ Busy waiting

- ◆ Thread가 CPU를 점유하면서 대기하고 있는 상태
- ◆ CPU 자원이 낭비 되고, 소모 전력이 불필요하게 낭비될 수 있다.

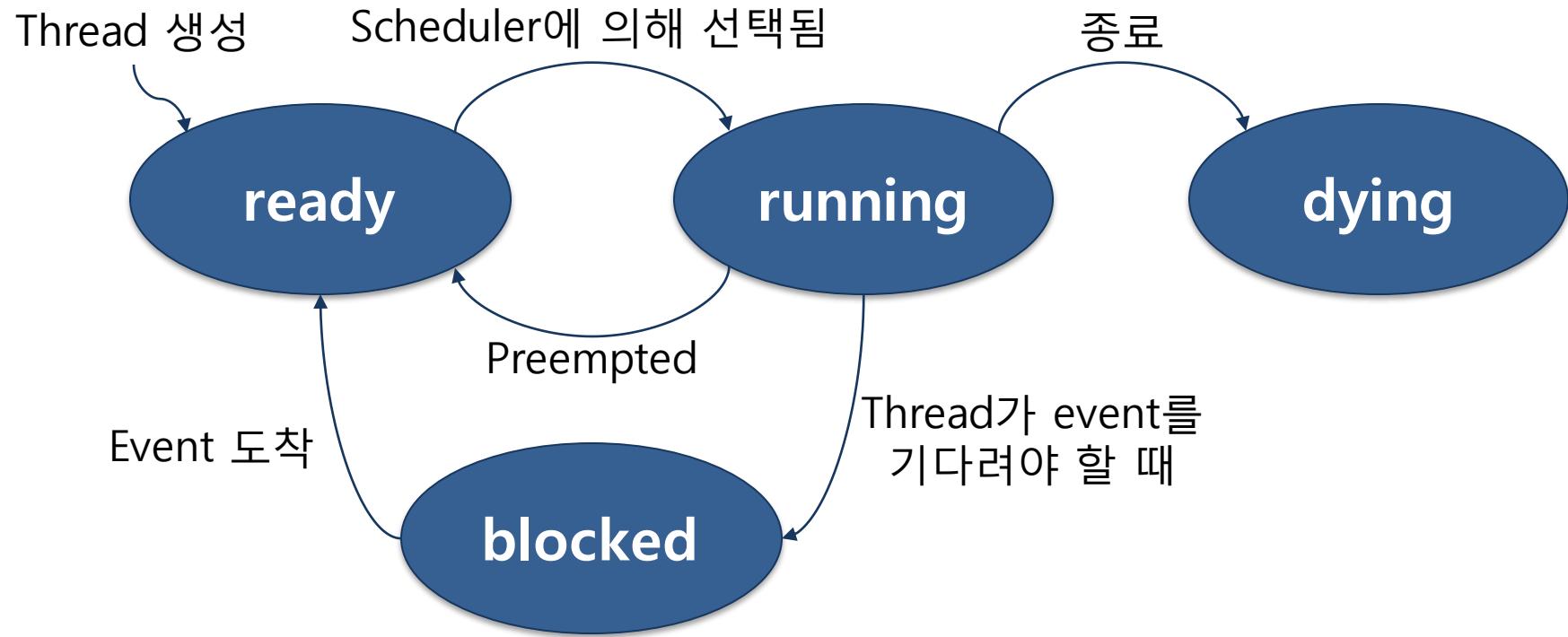
▣ 수정해야 할 주요 파일

- ◆ thread/thread.* , devices/timer.*



Thread 기초

▣ Pintos의 thread lifecycle



Thread 기초 (Cont.)

▣ Thread 자료구조

pintos/src/threads/thread.h

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;
    enum thread_status status;
    char name[16];
    uint8_t *stack;
    int priority;
    struct list_elem allelem;

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;           /* List element. */

#ifndef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;              /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                 /* Detects stack overflow. */
};
```

enum thread_status

{

THREAD_RUNNING, /* Running thread. */
 THREAD_READY, /* Not running but ready to run. */
 THREAD_BLOCKED, /* Waiting for an event to trigger. */
 THREAD_DYING /* About to be destroyed. */
};

/* Thread identifier. */
/* Thread state. */
/* Name (for debugging purposes). */
/* Saved stack pointer. */
/* Priority. */
/* List element for all threads list. */

Thread 기초 (Cont.)

▣ Pintos의 list 자료 구조

- ◆ Pintos의 모든 thread는 list로 관리

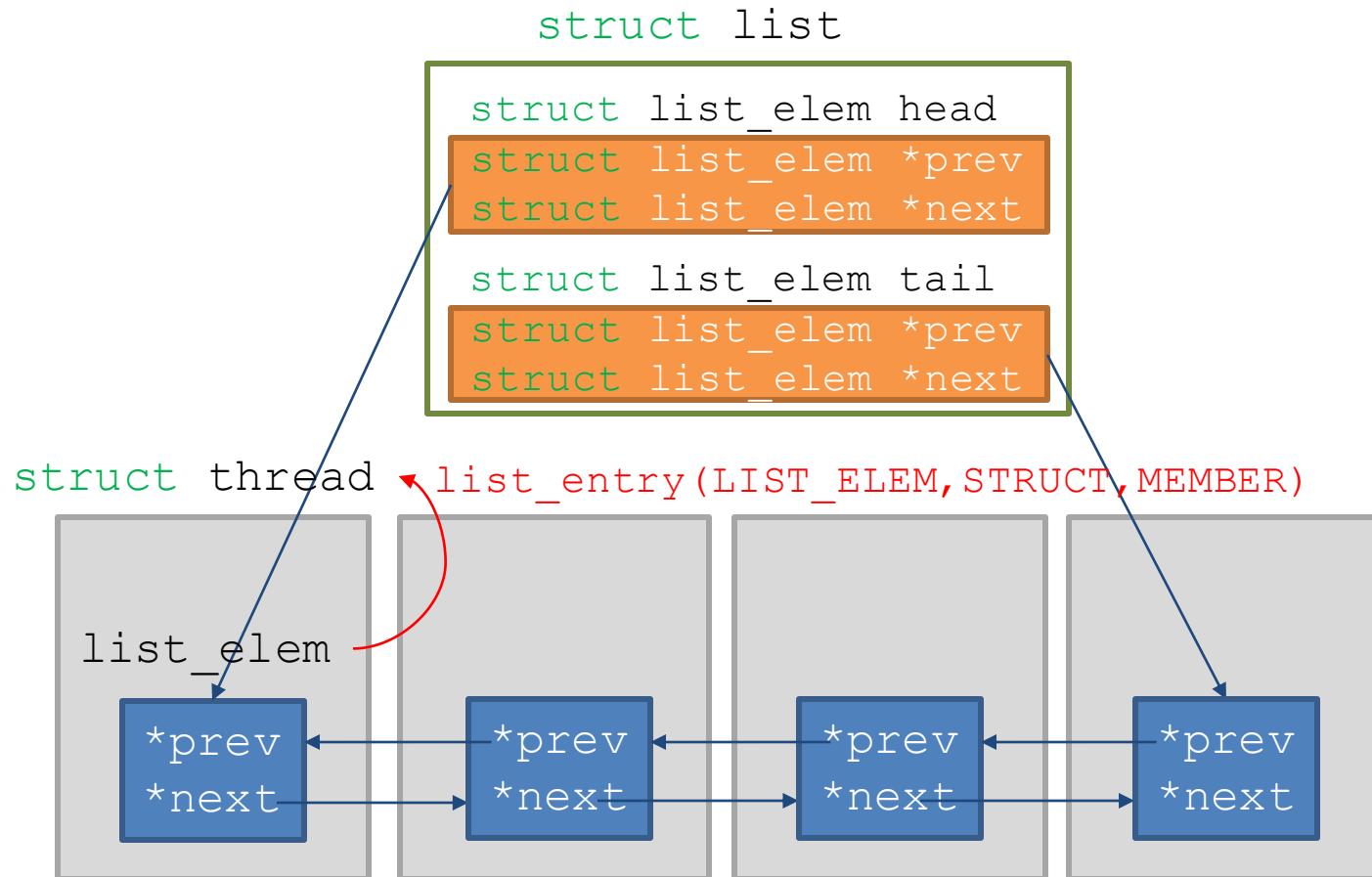
pintos/src/lib/kernel/list.h

```
/* List element. */
struct list_elem
{
    struct list_elem *prev;          /* Previous list element. */
    struct list_elem *next;          /* Next list element. */
};

/* List. */
struct list
{
    struct list_elem head;          /* List head. */
    struct list_elem tail;          /* List tail. */
};
```

Thread 기초 (Cont.)

▣ Pintos의 list 자료 구조



Thread 기초 (Cont.)

▣ Pintos의 list 자료 구조

pintos/src/threads/thread.c

```
/* List of processes in THREAD_READY state, that is, processes that
   are ready to run but not actually running. */
static struct list ready_list;

/* List of all processes. Processes are added to this list
   when they are first scheduled and removed when they exit. */
static struct list all_list;
```

- ◆ ready_list
 - Ready 상태의 thread를 관리하는 list
- ◆ all_list
 - 모든 thread를 관리하는 list



Thread 기초 (Cont.)

▣ list 관련 함수 소개 (pintos/src/lib/kernel/list.*)

```
void list_init(struct list *list)
```

- ◆ list 자료 구조를 초기화 한다

```
void list_push_back(struct list *list,  
                     struct list_elem *elem)
```

- ◆ elem을 list의 끝에 삽입 한다

```
void list_push_front(struct list *list,  
                     struct list_elem *elem)
```

- ◆ elem을 list의 처음에 삽입 한다

Thread 기초 (Cont.)

▣ list 관련 함수 소개 (pintos/src/lib/kernel/list.*)

```
struct list_elem * list_pop_front(struct list *list)
```

- ◆ list의 처음 list_elem을 반환한다

```
struct list_elem * list_pop_back(struct list *list)
```

- ◆ list의 마지막 list_elem을 반환한다

```
#define list_entry(LIST_ELEM, STRUCT, MEMBER)
```

- ◆ list_elem을 인자로 list_elem이 포함된 struct의 포인터를 반환한다

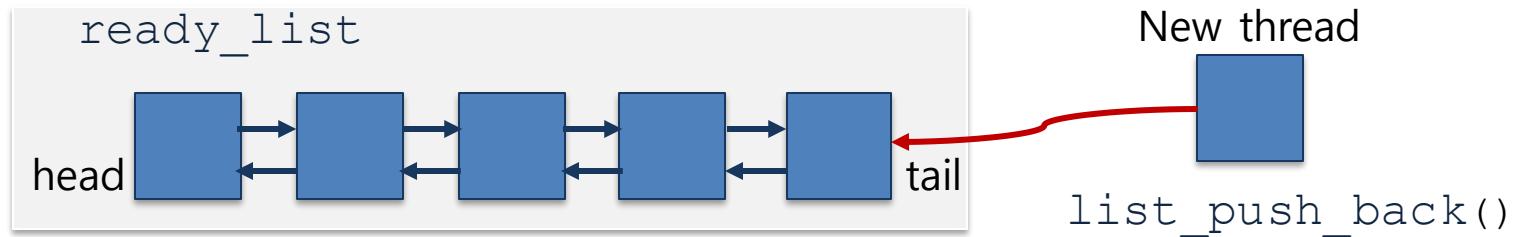
수정 및 추가 자료구조

```
struct thread  
  
static struct list sleep_list;  
  
int64_t next_tick_to_awake = INT64_MAX;
```



Pintos의 스케줄링 방식

- Ready_list: CPU를 기다리는 쓰레드들의 배열
 - ◆ 새로운 쓰레드는 리스트 맨뒤에 추가(list_push_back())

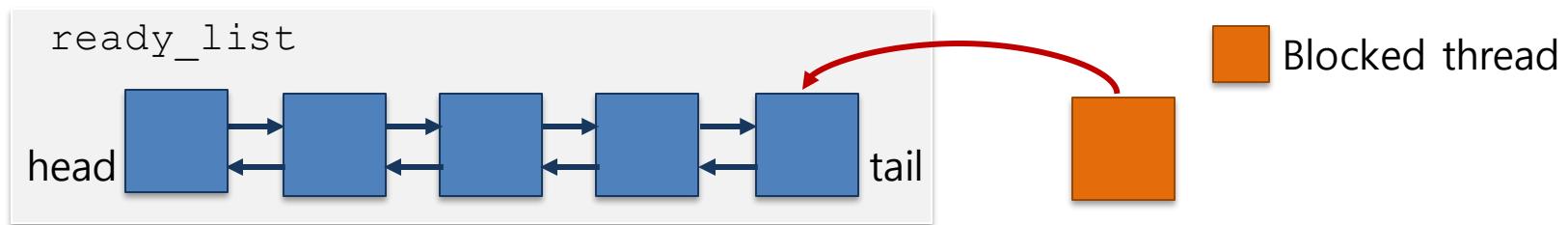


Pintos의 스케줄링 방식

▣ timer_sleep()

- ◆ 주어진 tick만큼 thread_yield()를 통해 CPU를 양보
- ◆ 주어진 tick 경과후 ready_list에 삽입된다.

```
while (timer_elapsed (start) < ticks)  
    thread_yield();
```



Pintos에서의 loop 기반 waiting

- 전달된 tick의 만큼 loop 기반 waiting

pintos/src/device/timer.c

```
void timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();

    while (timer_elapsed (start) < ticks)
        thread_yield ();
}
```

- ◆ `thread_yield()` : CPU를 양보하고, `thread`를 `ready_list`에 삽입
- ◆ `timer_ticks()` : 현재 진행되고 있는 tick의 값을 반환
- ◆ `timer_elased()` : 인자로 전달 된 tick이후 몇 tick이 지났는지 반환

Pintos에서의 loop 기반 waiting (Cont.)

- CPU를 양보하고, thread를 ready_list에 삽입

pintos/src/device/timer.c

```
void thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    old_level = intr_disable ();
    if (cur != idle_thread)
        list_push_back (&ready_list, &cur->elem);
    cur->status = THREAD_READY;
    schedule ();
    intr_set_level (old_level);
}
```

Pintos에서의 loop 기반 waiting (Cont.)

▣ thread_yield() 함수 설명

thread_current()

- ◆ 현재 실행 되고 있는 thread를 반환

intr_disable()

- ◆ 인터럽트를 비활성하고 이전 인터럽트의 상태를 반환

intr_set_level(old_level)

- ◆ 인자로 전달된 인터럽트 상태로 인터럽트를 설정 하고 이전 인터럽트 상태를 반환

list_push_back(&ready_list, &cur->elem)

- ◆ 주어진 entry를 list의 마지막에 삽입

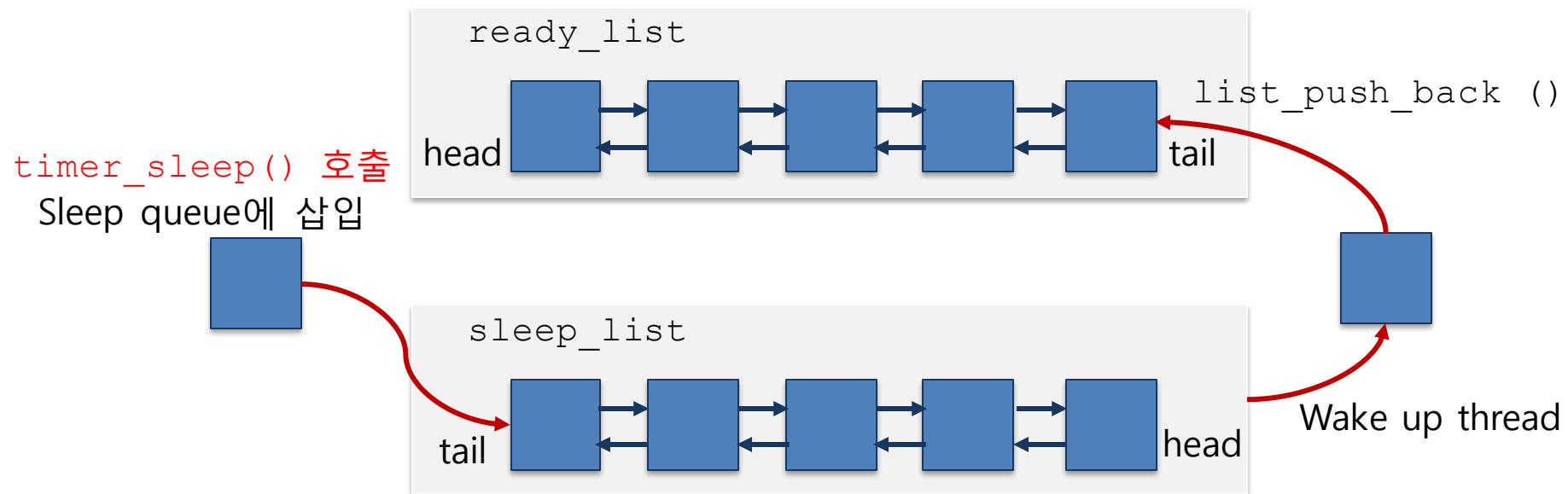
schedule()

- ◆ 컨텍스트 스위치 작업을 수행



Alarm clock 솔루션

- Sleep/wake up 기반의 alarm clock 구현



Alarm clock 솔루션

▣ Sleep queue의 도입

- ◆ Sleep된 thread를 저장하는 자료구조
- ◆ timer_sleep()을 호출한 thread를 저장

▣ 수정

- ◆ loop 기반 wait() → sleep/wakeup 으로 변경
- ◆ timer_sleep() 호출시 thread를 ready_list에서 제거, sleep queue에 추가
- ◆ wake up 수행
 - timer interrupt가 발생시 tick 체크
 - 시간이 다 된 thread는 sleep queue에서 삭제하고, ready_list에 추가

Alarm clock 구현

▣ thread 자료 구조 수정

- ◆ 깨어나야 할 tick 저장 (wakeup_tick)

pintos/src/thread/thread.h

```
struct thread
{
    ...
    /* 깨어나야 할 tick을 저장할 변수 추가 */
    ...
}
```



Alarm clock 구현 (Cont.)

▣ 전역변수 추가

- ◆ Sleep queue 자료구조 추가
- ◆ next_tick_to_awake 전역 변수 추가
 - sleep_list에서 대기중인 스레드들의 wakeup_tick 값 중 최소값을 저장

pintos/src/threads/thread.c

```
...
/* THREAD_BLOCKED 상태의 스레드를 관리하기 위한 리스트 자료 구조 추가 */
/* sleep_list에서 대기중인 스레드들의 wakeup_tick 값 중 최소값을 저장
하기위한 변수 추가 */
...
```

Alarm clock 구현 (Cont.)

▣ 구현할 함수 선언

pintos/src/threads/thread.h

```
...
void thread_sleep(int64_t ticks); /* 실행 중인 스레드를 슬립으로 만듬 */
void thread_awake(int64_t ticks); /* 슬립큐에서 깨워야할 스레드를 깨움 */
void update_next_tick_to_awake(int64_t ticks); /*최소 틱을 가진
                                                 스레드 저장 */
int64_t get_next_tick_to_awake(void); /* thread.c의
                                         next_tick_to_awake 반환 */
...
```



Alarm clock 구현 (Cont.)

- thread_init() 함수 수정
 - Sleep queue 자료구조 초기화 코드 추가

pintos/src/threads/thread.c

```
void thread_init (void)
{
    /* sleep_list 를 초기화 */
    ...
}
```



Alarm clock 구현 (Cont.)

- ▣ timer_sleep() 함수 수정
 - ◆ Sleep queue를 이용하도록 함수 수정

pintos/src/devices/timer.c

```
void timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks ();

    /* 기존의 busy waiting을 유발하는 코드를 삭제 */
    /* 새로 구현한 thread를 sleep queue에 삽입하는 함수를 호출 */
}
```



Alarm clock 구현 (Cont.)

▣ thread_sleep() 함수 구현

- ◆ Thread를 blocked 상태로 만들고 sleep queue에 삽입하여 대기
- ◆ devices/timer.c : timer_sleep() 함수에 의해 호출

pintos/src/threads/thread.c

```
void thread_sleep(int64_t ticks){  
  
    /* 현재 스레드가 idle 스레드가 아닐경우  
       thread의 상태를 BLOCKED로 바꾸고 깨어나야 할 ticks을 저장,  
       슬립 큐에 삽입하고, awake함수가 실행되어야 할 tick값을 update */  
    /* 현재 스레드를 슬립 큐에 삽입한 후에 스케줄한다. */  
    /* 해당 과정중에는 인터럽트를 받아들이지 않는다. */  
}
```



Alarm clock 구현 (Cont.)

- timer_interrupt() 함수 수정

pintos/src/devices/timer.c

```
static void timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick ();

    /* 매 tick마다 sleep queue에서 깨어날 thread가 있는지 확인하여,
     * 깨우는 함수를 호출하도록 한다. */
}
```

Alarm clock 구현 (Cont.)

▣ thread_awake() 함수 구현

- ◆ wakeup_tick값이 ticks보다 작거나 같은 스레드를 깨움
- ◆ 현재 대기중인 스레드들의 wakeup_tick변수 중 가장작은 값을 next_tick_to_awake 전역 변수에 저장

pintos/src/threads/thread.c

```
void thread_awake(int64_t ticks){  
  
    /* sleep list의 모든 entry 를 순회하며 다음과 같은 작업을 수행한다.  
     현재 tick이 깨워야 할 tick 보다 크거나 같다면 슬립 큐에서 제거하고  
     unblock 한다.  
     작다면 update_next_tick_to_awake() 를 호출한다.  
    */  
  
}
```



Alarm clock 구현 (Cont.)

- ▣ update_next_tick_to_awake() 함수 추가
 - ◆ next_tick_to_awake 변수를 업데이트

pintos/src/threads/thread.c

```
void update_next_tick_to_awake(int64_t ticks)
{
    /* next_tick_to_awake 가 깨워야 할 스레드중 가장 작은 tick을 갖도록
       업데이트 한다 */
}
```

Alarm clock 구현 (Cont.)

- get_next_tick_to_awake() 함수 추가

pintos/src/threads/thread.c

```
int64_t get_next_tick_to_awake(void)
{
    /* next_tick_to_awake 을 반환한다. */
}
```



결과 비교

```
$ pintos -- -q run alarm-multiple
```

- ◆ Busy waiting

```
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
Thread: 0 idle ticks, 860 kernel ticks, 0 user ticks
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
```

- ◆ Busy waiting 제거 (sleep queue 사용)

```
(alarm-multiple) thread 2: duration=50, iteration=0, product=180
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
Thread: 550 idle ticks, 312 kernel ticks, 0 user ticks
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
```

- ◆ Sleep 상태에서도 CPU를 점유하고 있어서 idle tick이 0이었으나, busy waiting을 제거한 후에는 idle tick이 증가하였다.

수정 및 추가 함수

```
void timer_sleep (int64_t ticks)
/* 인자로 주어진 ticks 동안 스레드를 block */

void thread_sleep(int64_t ticks)
/* Thread를 blocked 상태로 만들고 sleep queue에 삽입하여 대기 */

void thread_awake(int64_t ticks)
/* Sleep queue에서 깨워야 할 thread를 찾아서 wake */

void update_next_tick_to_awake(int64_t ticks)
/* Thread들이 가진 tick 값에서 최소 값을 저장 */

int64_t get_next_tick_to_awake(void)
/* 최소 tick값을 반환 */
```



7. Priority Scheduling

Priority Scheduling 개요

▣ 과제 목표

- ◆ 현재 pintos의 스케줄러는 라운드 로빈으로 구현되어 있다. 이를 우선순위를 고려하여 스케줄링 하도록 수정한다.
 - Time Quantum: 40msec (src/thread.c)
 - Ready list에 새로 추가된 thread의 우선순위가 현재 CPU를 점유중인 thread의 우선순위보다 높으면, 기존 thread를 밀어내고 CPU를 점유하도록 한다
 - 여러 thread가 lock, semaphore 를 얻기 위해 기다릴 경우 우선순위가 가장 높은 thread가 CPU를 점유 한다

▣ 수정해야 할 주요 파일

- ◆ thread.* , synch.*

Priority 기반 Scheduling

- ▣ 새로운 thread의 우선순위가 현재 실행중인 thread의 우선순위보다 높으면, 새로운 thread가 실행중인 thread를 선점.
- ▣ 기존 pintos 에서는 ready_list 에 새로운 thread가 삽입 되어도 선점이 발생하지 않음.
- ▣ Pintos는 ready list에 삽입된 순으로 thread가 CPU를 점유.
 - ◆ pintos는 thread가 unblock되거나 생성될 때 우선순위와 관련 없이 ready_list의 가장 뒤에 삽입.

pintos에서의 우선순위

- ▣ PRI_MIN(=0) 부터 PRI_MAX(=63)까지의 범위를 가지며 숫자가 클수록 우선순위가 높음.
- ▣ thread_create() 함수로 thread를 생성할 때 인자로 초기 우선순위를 전달(기본 값으로 PRI_DEFAULT(=31))
- ▣ 생성된 thread의 우선순위는 다음 함수를 통해 변경 가능.
 - ◆ void thread_set_priority (int new_priority)
 - 현재 thread의 우선순위를 new_priority로 변경
 - ◆ int thread_get_priority (void)
 - 현재 thread의 우선순위를 반환

Priority Scheduling 문제 솔루션

- ▣ 새로 추가된 thread가 실행 중인 thread보다 우선순위가 높은 경우 CPU를 선점하도록 하기 위해 `thread_create()` 함수 수정.
- ▣ `ready_list`를 우선순위로 정렬 하기 위해 `thread_unblock()`, `thread_yield()` 함수들을 수정하고 `list_insert_ordered()` 함수에 사용 되는 `cmp_priority()` 함수를 추가한다
- ▣ 새로운 thread가 생성 되거나(`thread_create()`) block 되었던 thread가 `unblock` 됐을때(`thread_unblock()`) `ready_list`에 thread가 추가 된다

Priority Scheduling 구현

▣ 구현할 함수 선언

pintos/src/threads/thread.h

```
void test_max_priority (void);
/* 현재 수행중인 스레드와 가장 높은 우선순위의 스레드의 우선순위를 비교하여
스케줄링 */
bool cmp_priority (const struct list_elem *a,
                   const struct list_elem *b,
                   void *aux UNUSED);
/* 인자로 주어진 스레드들의 우선순위를 비교 */
```



Priority Scheduling 구현 (Cont.)

▣ thread_create() 함수 수정

- ◆ Thread의 ready_list 삽입시 현재 실행중인 thread와 우선순위를 비교하여, 새로 생성된 thread의 우선순위가 높다면 thread_yield()를 통해 CPU를 양보.

pintos/src/threads/thread.c

```
tid_t thread_create (const char *name, int priority,
                      thread_func *function, void *aux)
{
    ...
    thread_unblock (t);

    /* 생성된 스레드의 우선순위가 현재 실행중인 스레드의
       우선순위 보다 높다면 CPU를 양보한다. */

    return tid;
}
```

Priority Scheduling 구현 (Cont.)

▣ thread_unblock() 함수 수정

- ◆ Thread가 unblock 될때 우선순위 순으로 정렬 되어 ready_list에 삽입되도록 수정

pintos/src/threads/thread.c

```
void thread_unblock (struct thread *t)
{
    ...
    /* 스레드가 unblock 될때 우선순위 순으로 정렬 되어
       ready_list에 삽입되도록 수정 */
    ...
}
```



Priority Scheduling 구현 (Cont.)

▣ thread_yield() 함수 수정

- ◆ 현재 thread가 CPU를 양보하여 ready_list에 삽입 될 때 우선순위 순서로 정렬되어 삽입 되도록 수정

pintos/src/threads/thread.c

```
void thread_yield (void)
{
    ...
    /* 현재 thread가 CPU를 양보하여 ready_list에 삽입 될 때
       우선순위 순서로 정렬되어 삽입 되도록 수정 */
    ...
}
```



Priority Scheduling 구현 (Cont.)

▣ thread_set_priority() 함수 설정

pintos/src/threads/thread.c

```
void thread_set_priority (int new_priority)
{
    thread_current ()->priority = new_priority;

    /* 스레드의 우선순위가 변경되었을때 우선순위에 따라
       선점이 발생하도록 한다. */
}
```

▣ test_max_priority() 함수 추가

pintos/src/threads/thread.c

```
void test_max_priority (void)
{
    /* ready_list에서 우선순위가 가장 높은 스레드와 현재 스레드의
       우선순위를 비교하여 스케줄링 한다.
       (ready_list 가 비여있지 않은지 확인) */

}
```



Priority Scheduling 구현 (Cont.)

▣ cmp_priority() 함수 추가

- ◆ 첫 번째 인자의 우선순위가 높으면 1을 반환, 두 번째 인자의 우선순위가 높으면 0을 반환
- ◆ list_insert_ordered() 함수에서 사용

pintos/src/threads/thread.c

```
bool cmp_priority (const struct list_elem* a_,  
                   const struct list_elem* b_, void* aux UNUSED)  
{  
    /* list_insert_ordered() 함수에서 사용 하기 위해 정렬 방법을 결정하기  
     위한 함수 작성 */  
}
```

결과

```
$ make check
```

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
FAIL tests/threads/priority-donate-one
FAIL tests/threads/priority-donate-multiple
FAIL tests/threads/priority-donate-multiple2
FAIL tests/threads/priority-donate-nest
FAIL tests/threads/priority-donate-sema
FAIL tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
FAIL tests/threads/priority-sema
FAIL tests/threads/priority-condvar
FAIL tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
16 of 27 tests failed.
```

추가 함수

```
void test_max_priority (void)  
/* 현재 수행중인 스레드와 가장 높은 우선순위의 스레드의 우선순위를 비교하여  
스케줄링 */  
  
bool cmp_priority (const struct list_elem *a,  
                   const struct list_elem *b,  
                   void *aux UNUSED)  
/* 인자로 주어진 스레드들의 우선순위를 비교 */
```



수정 함수

```
tid_t thread_create (const char *name, int priority,  
                     thread_func *function, void *aux)  
  
/* 새로운 스레드를 생성 */  
  
void thread_unblock (struct thread *t)  
  
/* block된 스레드를 unblock */  
  
void thread_yield (void)  
  
/* 현재 수행중인 스레드가 사용중인 CPU를 양보 */  
  
void thread_set_priority (int new_priority)  
  
/* 현재 수행중인 스레드의 우선순위를 변경 */
```



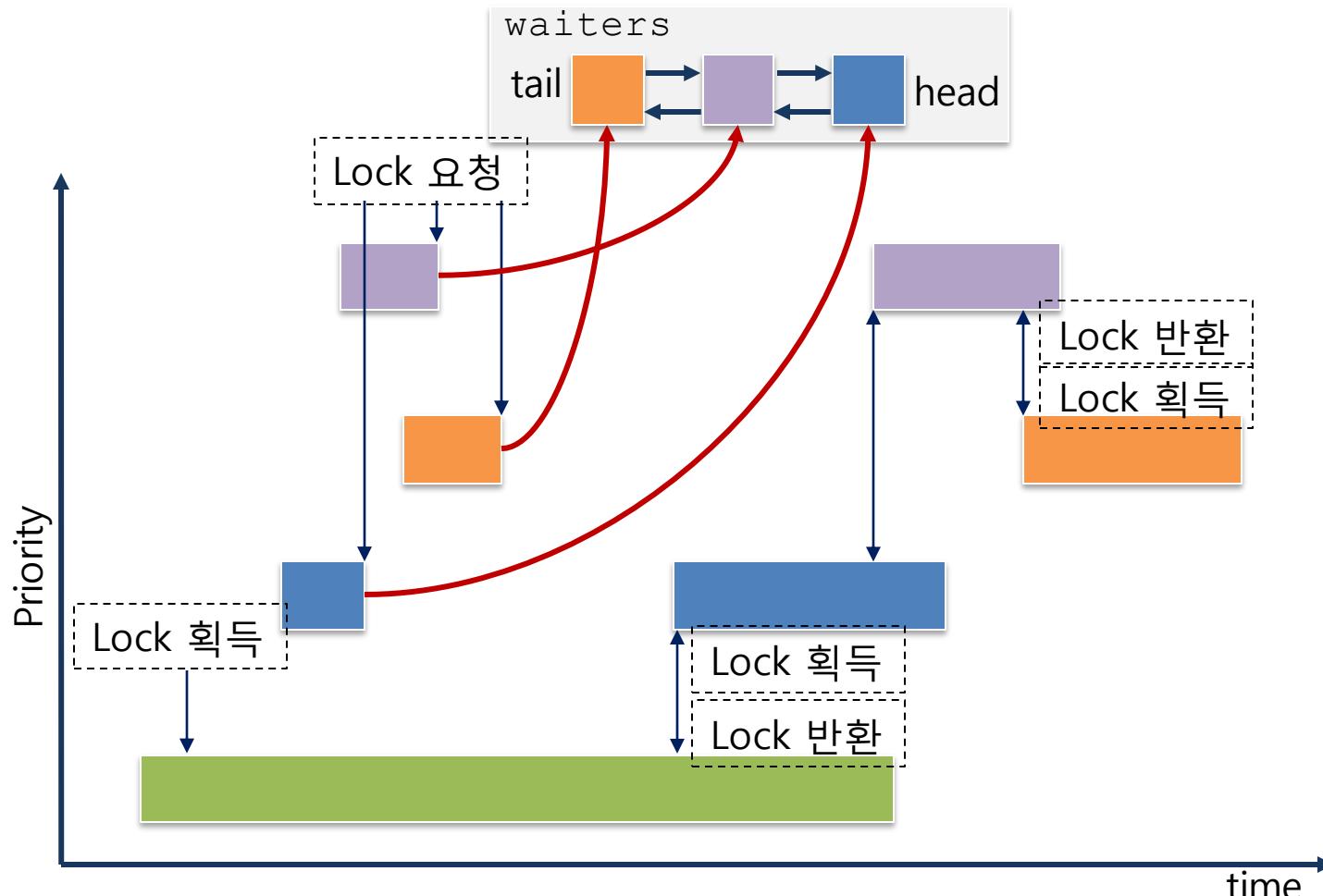
8. Priority Scheduling and Synchronization

Priority Scheduling-Synchronization 개요

- ▣ 여러 스레드가 lock, semaphore, condition variable 을 얻기 위해 기다릴 경우 우선순위가 가장 높은 thread가 CPU를 점유 하도록 구현
 - ◆ 현재 pintos는 semaphore를 대기 하고 있는 스레드들의 list인 `waiters`가 FIFO로 구현되어있다
- ▣ 수정해야 할 주요 파일
 - ◆ `thread.*`, `synch.*`

Priority Scheduling-Synchronization

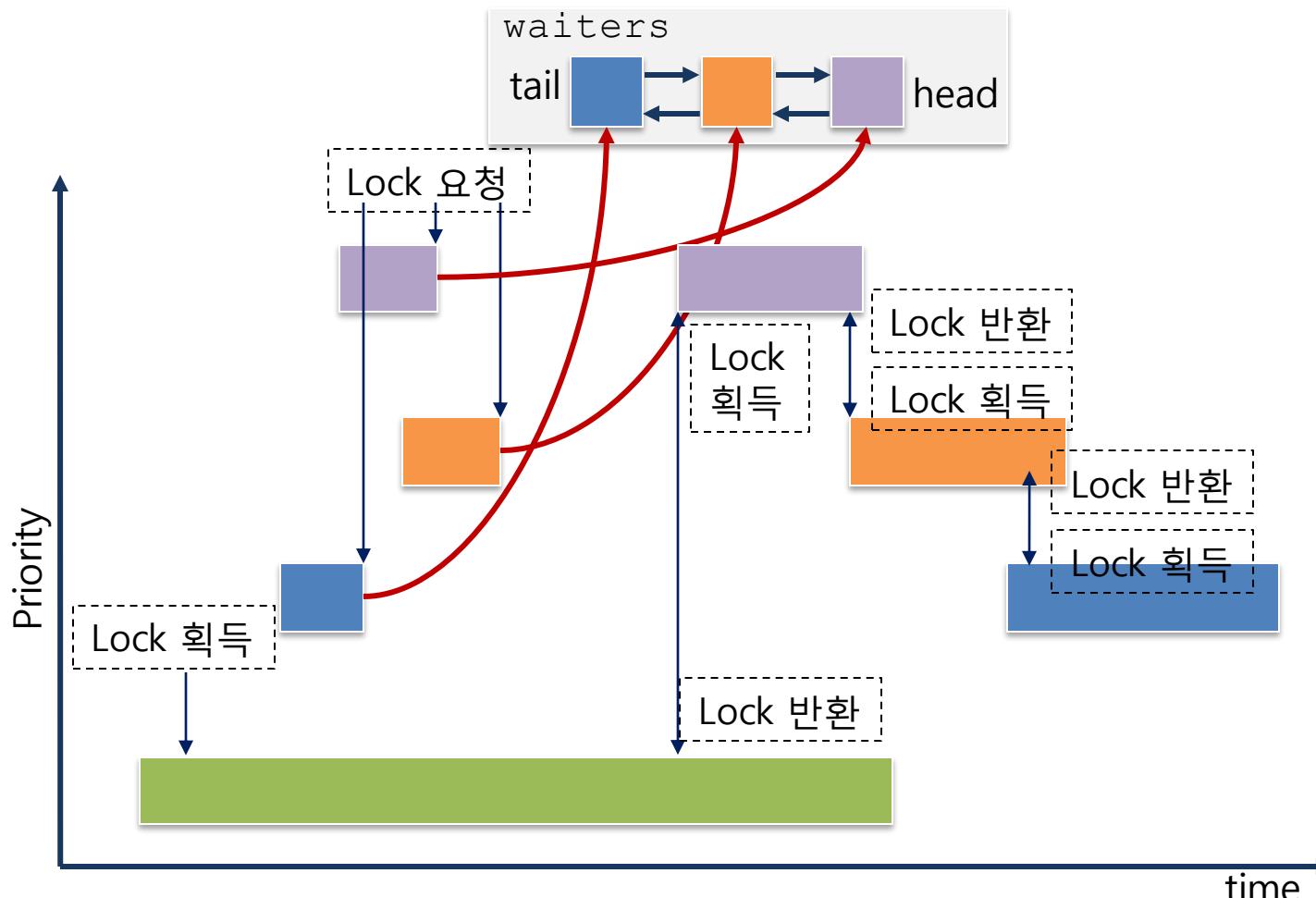
- 우선순위를 무시하고 waiters list에 삽입되는 순서대로 lock을 획득



Priority Scheduling-Synchronization 솔루션

- Semaphore를 요청 할때 대기 리스트를 우선순위로 정렬

- sema_down ()에서 waiters list를 우선수위로 정렬 하도록 수정



Pintos 의 semaphore

pintos/src/threads/synch.h

```
struct semaphore {
    unsigned value;           /* Current value. */
    struct list waiters;     /* List of waiting
                                threads. */
};
```

`void sema_init(struct semaphore *sema, unsigned value)`

- ◆ semaphore를 주어진 value로 초기화

`void sema_down(struct semaphore *sema)`

- ◆ semaphore를 요청하고 획득했을 때 value를 1 낮춤

`void sema_up(struct semaphore *sema)`

- ◆ semaphore를 반환하고 value를 1 높임



Pintos 의 lock

pintos/src/threads/synch.h

```
struct lock
{
    struct thread *holder;          /* Thread holding lock */
    struct semaphore semaphore;    /* Binary semaphore
                                    controlling access. */
};
```

void lock_init (struct lock *lock)

- ◆ lock 자료구조를 초기화

void lock_acquire (struct lock *lock)

- ◆ lock을 요청

void lock_release (struct lock *lock)

- ◆ lock을 반환



Pintos 의 condition variable

pintos/src/threads/synch.h

```
struct condition {
    struct list waiters;      /* List of waiting threads. */
};
```

`void cond_init(struct condition *cond)`

- ◆ condition variable 자료구조를 초기화

`void cond_wait(struct condition *cond, struct lock *lock)`

- ◆ condition variable을 통해 signal이 오는지 기림

`void cond_signal(struct condition *cond,
 struct lock *lock UNUSED)`

- ◆ condition variable에서 기다리는 가장 높은 우선순위의 스레드에 signal을 보냄

`void cond_broadcast(struct condition *cond, struct lock *lock)`

- ◆ condition variable에서 기다리는 모든 스레드에 signal을 보냄



Priority Scheduling-Synchronization 구현

▣ sema_down() 함수 수정

- ◆ Semaphore를 얻고 waiters 리스트 삽입 시, 우선순위대로 삽입되도록 수정

pintos/src/threads/synch.c

```
void sema_down (struct semaphore *sema)
{
    ...
    /* waiters 리스트 삽입 시, 우선순위대로 삽입되도록 수정 */
    ...
}
```

Priority Scheduling-Synchronization 구현 (Cont.)

▣ sema_up() 함수 수정

pintos/src/threads/synch.c

```
void sema_up (struct semaphore *sema)
{
    ...
    if (!list_empty (&sema->waiters))
    {

        /* 스레드가 waiters list에 있는 동안 우선순위가 변경 되었을
           경우를 고려 하여 waiters list 를 우선순위로 정렬 한다. */
        thread_unblock (list_entry (list_pop_front
                                    (&sema->waiters), struct thread, elem));
    }
    sema->value++;

    /* priority preemption 코드 추가*/
    ...
}
```



Priority Scheduling-Synchronization 구현 (Cont.)

▣ 구현할 함수 선언

pintos/src/threads/synch.h

```
bool cmp_sem_priority (const struct list_elem *a,  
                      const struct list_elem *b,  
                      void *aux);
```

Priority Scheduling-Synchronization 구현 (Cont.)

▣ cmp_sem_priority() 함수 추가

- 첫 번째 인자의 우선순위가 두 번째 인자의 우선순위보다 높으면 1을 반환 낮으면 0을 반환

pintos/src/threads/synch.c

```
bool cmp_sem_priority (const struct list_elem *a, const struct
                      list_elem *b, void *aux UNUSED) {
    struct semaphore_elem *sa = list_entry(a,
                                            struct semaphore_elem, elem);
    struct semaphore_elem *sb = list_entry(b,
                                            struct semaphore_elem, elem);

    /* 해당 condition variable 을 기다리는 세마포어 리스트를
       가장 높은 우선순위를 가지는 스레드의 우선순위 순으로 정렬하도록 구현 */
}
```

Priority Scheduling-Synchronization 구현 (Cont.)

▣ cond_wait() 함수 수정

- ◆ condition variable의 waiters list에 우선순위 순서로 삽입되도록 수정

pintos/src/threads/synch.c

```
void cond_wait (struct condition *cond, struct lock *lock)
{
    ...
    /* condition variable의 waiters list에 우선순위 순서로
       삽입되도록 수정 */
    ...
}
```

Priority Scheduling-Synchronization 구현 (Cont.)

▣ cond_signal() 함수 수정

- ◆ condition variable의 waiters list를 우선순위로 재 정렬
 - 대기 중에 우선순위가 변경되었을 가능성이 있음

pintos/src/threads/synch.c

```
void cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
    ...
    if (!list_empty (&cond->waiters))
    {
        /* condition variable의 waiters list를 우선순위로 재 정렬 */

        sema_up (&list_entry (list_pop_front (&cond->waiters),
                             struct semaphore_elem, elem)->semaphore);
    }
}
```



결과

```
$ make check
```

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
FAIL tests/threads/priority-donate-one
FAIL tests/threads/priority-donate-multiple
FAIL tests/threads/priority-donate-multiple2
FAIL tests/threads/priority-donate-nest
FAIL tests/threads/priority-donate-sema
FAIL tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
FAIL tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
14 of 27 tests failed.
```



수정 및 추가 함수

```
void sema_down (struct semaphore *sema)
void sema_up (struct semaphore *sema)
void cond_wait (struct condition *cond, struct lock *lock)
void cond_signal (struct condition *cond,
                  struct lock *lock UNUSED)

bool cmp_sem_priority(const struct list_elem *a,
                      const struct list_elem *b,
                      void *aux UNUSED)

/* 첫번째 인자로 주어진 세마포어를 위해 대기 중인 가장 높은 우선순위의
스레드와 두번째 인자로 주어진 세마포어를 위해 대기 중인 가장 높은
우선순위의 스레드와 비교 */
```

9. Priority Inversion Problem

Priority inversion Problem 개요

▣ 과제 목표

- ◆ Priority donation 구현
- ◆ Multiple donation 구현
- ◆ Nested donation 구현

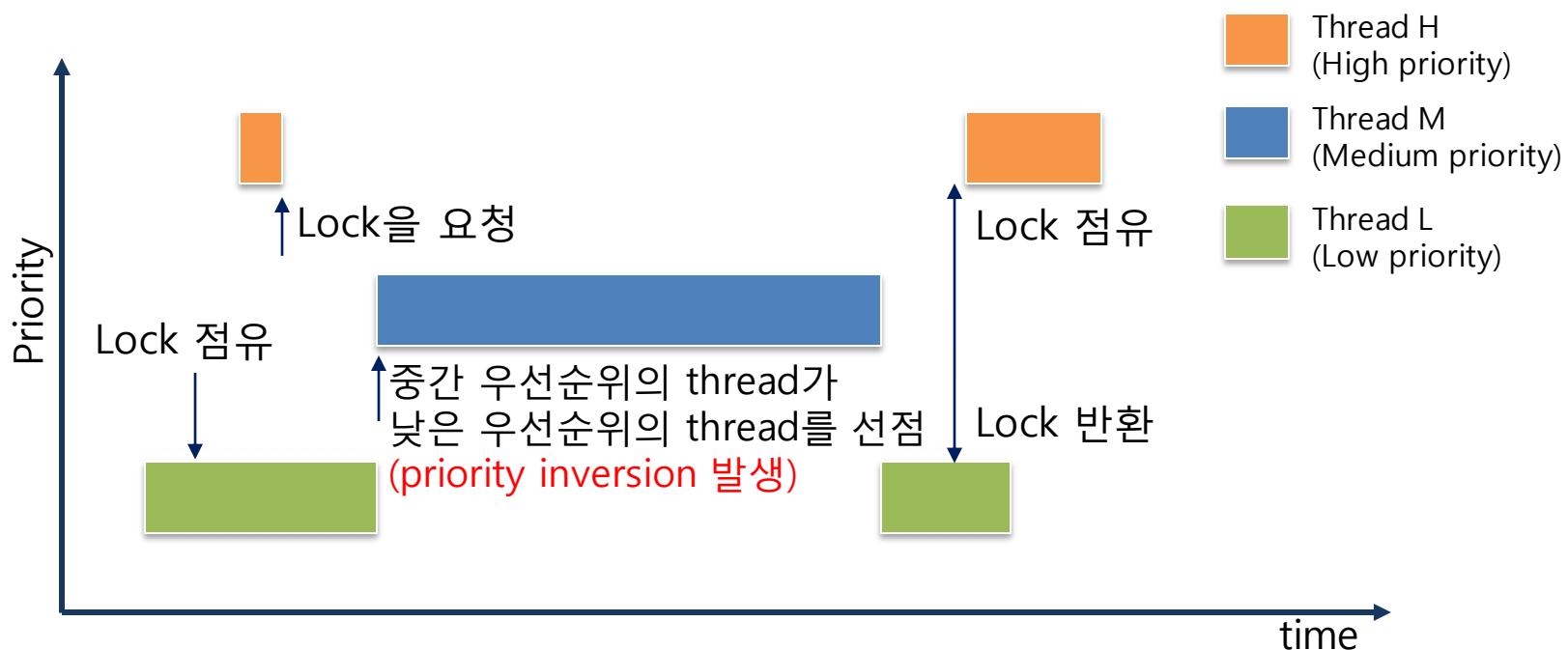
▣ 수정해야 할 주요 파일

- ◆ thread.* , synch.*



Priority inversion Problem

- 우선순위가 높은 쓰레드가 우선순위 낮은 쓰레드를 기다리는 현상



Priority inversion Problem 솔루션

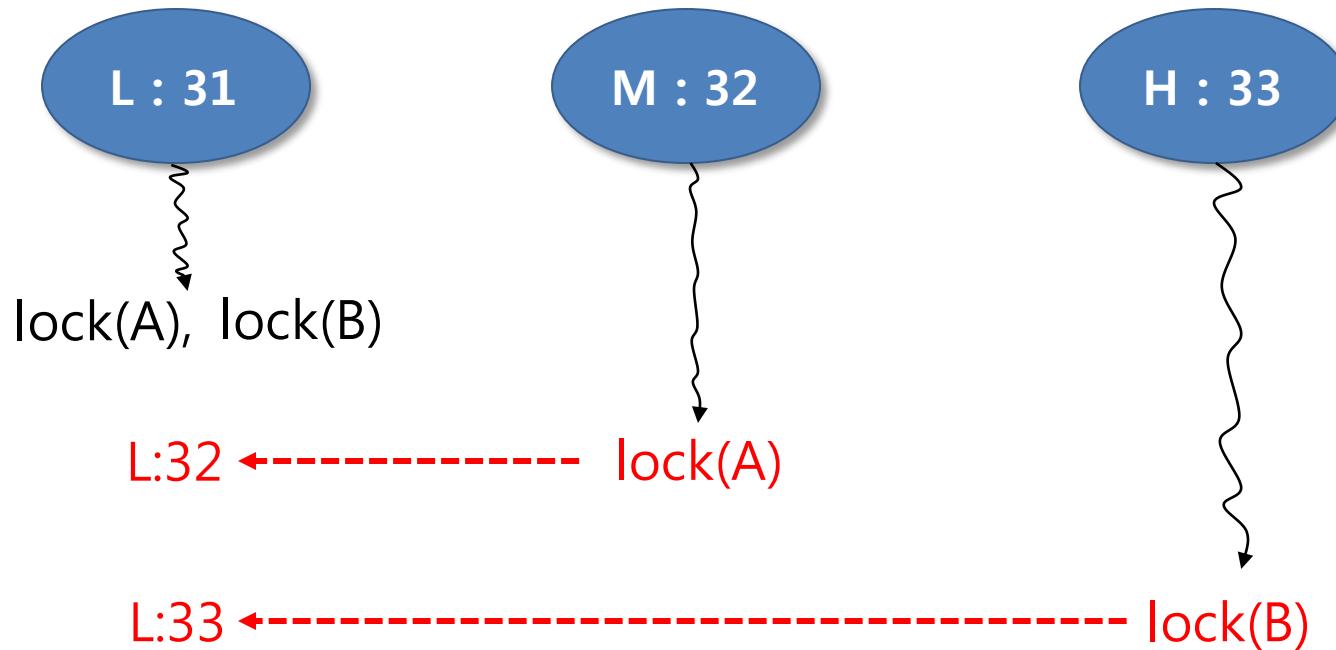
▣ Priority donation



Multiple donation

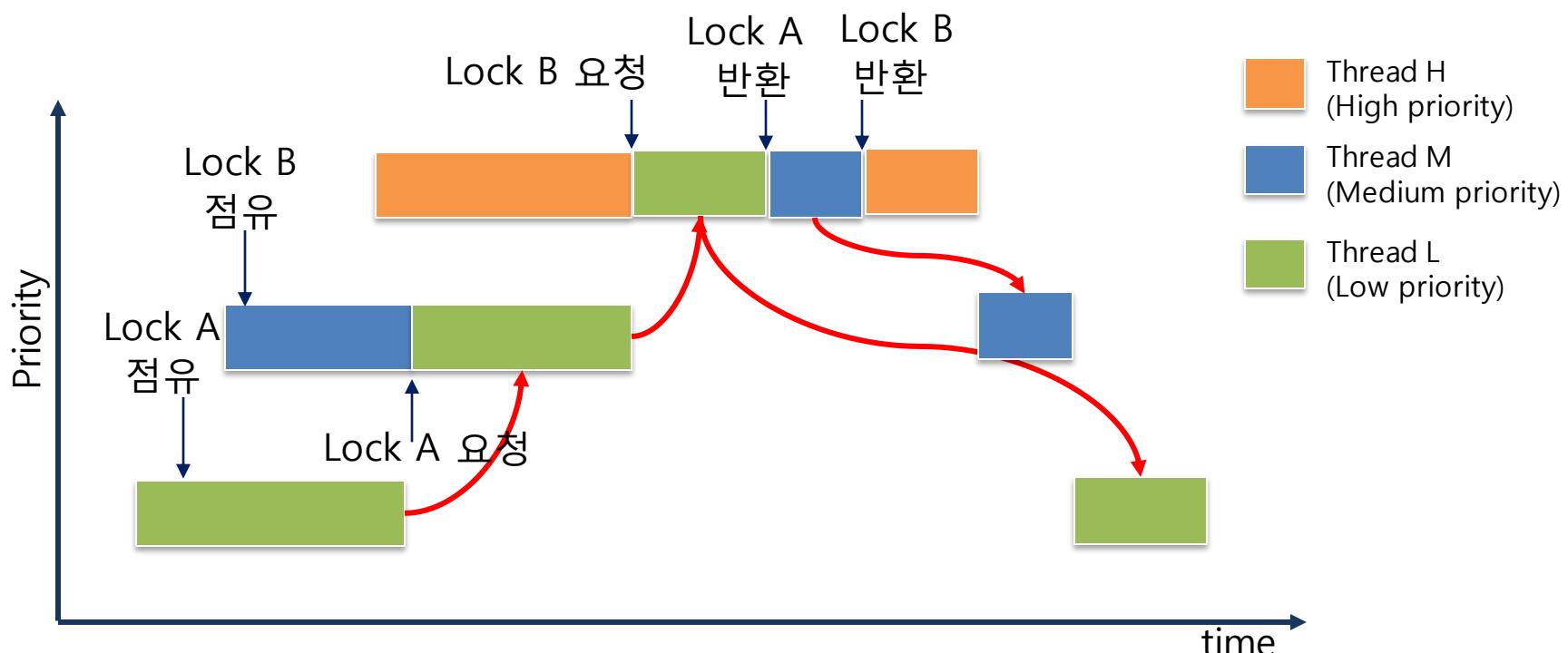
- ▣ 스레드가 두 개 이상의 lock 보유 시, 각 lock에 의해 도네이션이 발생 가능 → 이전 상태의 우선순위를 기억하고 있어야 함
- ▣ 예시
 - ◆ 스레드 *L*(priority: 31)이 *lock A*와 *lock B*를 점유
 - ◆ 스레드 *M*(priority: 32)이 *lock A*를 요청, 우선순위 도네이션 발생 (스레드 *L*, priority = 32)
 - ◆ 스레드 *H*(priority: 33)이 *lock B*를 요청, 다시 우선순위 도네이션 발생 (스레드 *L*, priority = 33)
 - ◆ 스레드 *L*이 *lock B*를 해지하여 스레드 *L*의 우선순위를 초기화(priority: 31)
 - 아직 *lock A*를 해지하지 않았지만 우선순위가 31로 변경
 - 이전 상태의 우선순위 (스레드 *M*에 의해 도네이션 받은 priority: 32)를 기억하고 있다가 스레드 *L*의 우선순위를 32로 변경해야 한다.

Multiple donation

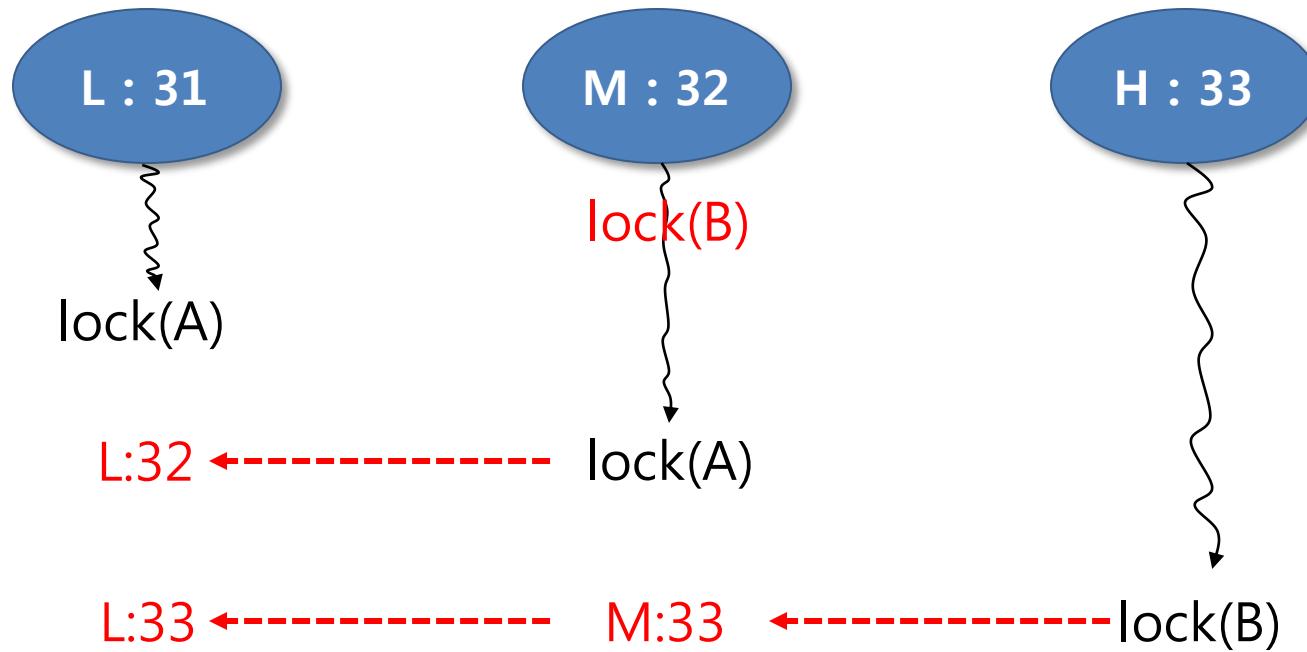


Nested donation

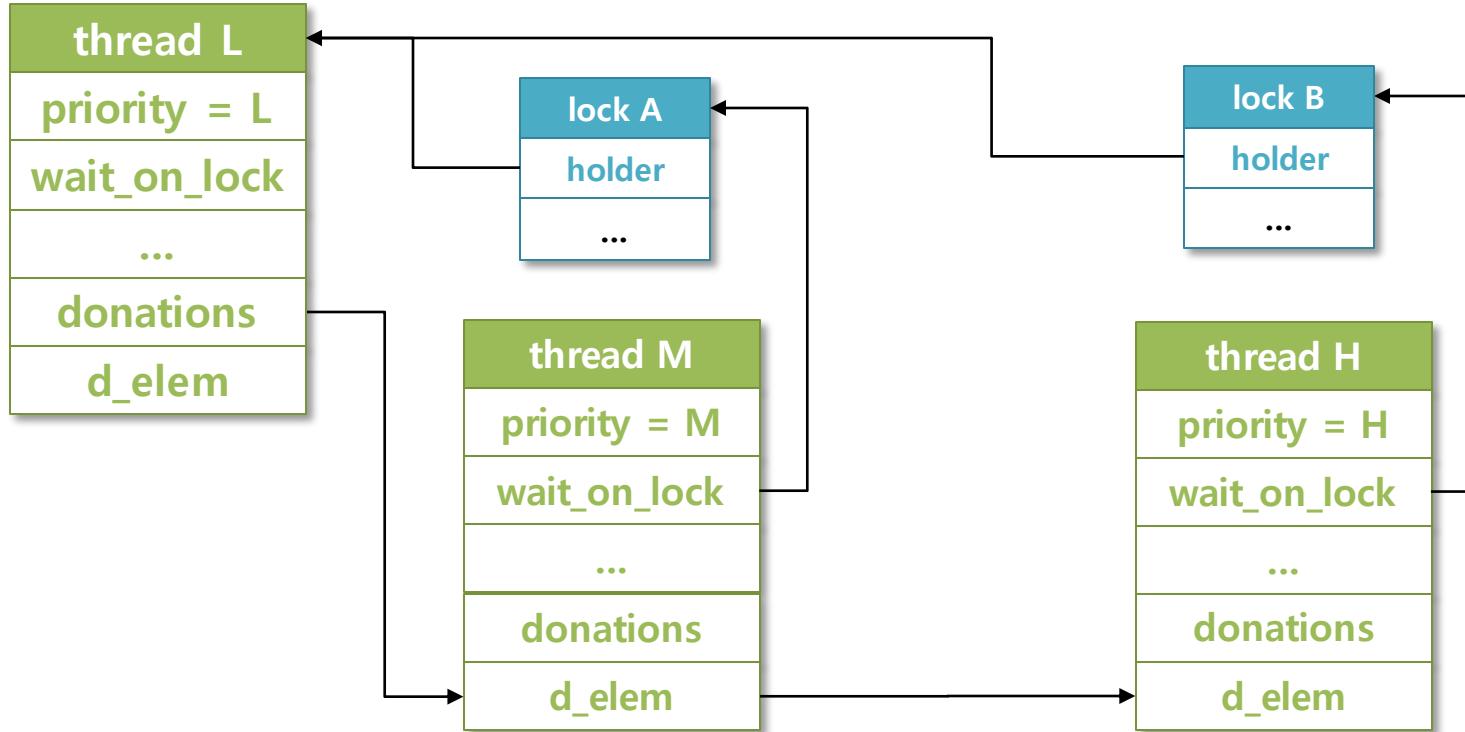
- 스레드 *H* 가 *lock B* (스레드 *M* 점유) 를 얻기 위해 대기. 이때 스레드 *M* 은 *lock A* (스레드 *L* 점유) 를 얻기 위해 대기 하고 있다.
- 스레드 *H* 의 우선순위는 스레드 *L,M* 에게 모두 도네이션 되어야 한다.



Nested donation



Multiple Donation



Nested donation



Priority donation 구현

- struct thread에 priority donation 관련 항목 추가

pintos/src/threads/thread.h

```
struct thread{  
    ...  
    int init_priority;  
    struct lock *wait_on_lock;  
    struct list donations;  
    struct list_elem donation_elem;  
    ...  
}
```

- init_priority: donation 이후 우선순위를 초기화하기 위해 초기값 저장
- wait_on_lock: 해당 스레드가 대기하고 있는 lock자료구조의 주소를 저장
- donations: multiple donation을 고려하기 위해 사용
- donation_elem: multiple donation을 고려하기 위해 사용

Priority donation 구현 (Cont.)

- ▣ `init_thread()` 함수 수정
 - ◆ Priority donation 관련 자료구조 초기화 코드 삽입

pintos/src/threads/thread.c

```
static void init_thread (struct thread *t, const char *name,
                        int priority)
{
    ...
    /* Priority donation 관련 자료구조 초기화 */
    ...
}
```



Priority donation 구현 (Cont.)

▣ lock_acquire() 함수 수정

pintos/src/threads/synch.c

```
void lock_acquire (struct lock *lock) {
    ...
    /* 해당 lock 의 holder가 존재 한다면 아래 작업을 수행한다. */
    /* 현재 스레드의 wait_on_lock 변수에 획득 하기를 기다리는 lock의 주소를 저장 */
    /* multiple donation 을 고려하기 위해 이전상태의 우선순위를 기억,
       donation 을 받은 스레드의 thread 구조체를 list로 관리한다. */
    /* priority donation 수행하기 위해 donate_priority() 함수 호출 */

    sema_down (&lock->semaphore);
    thread_current ()->wait_on_lock = NULL;

    /* lock을 획득 한 후 lock holder 를 갱신한다. */
}
```

Priority donation 구현 (Cont.)

▣ 구현할 함수 선언

pintos/src/threads/thread.h

```
...
void donate_priority(void);
void remove_with_lock(struct lock *lock);
void refresh_priority(void);
...
```



Priority donation 구현 (Cont.)

- donate_priority() 함수 추가

pintos/src/threads/thread.c

```
void donate_priority(void)
{
    /* priority donation 을 수행하는 함수를 구현한다.
    현재 스레드가 기다리고 있는 lock 과 연결 된 모든 스레드들을 순회하며
    현재 스레드의 우선순위를 lock 을 보유하고 있는 스레드에게 기부 한다.
    (Nested donation 그림 참고, nested depth 는 8로 제한한다. ) */
}
```



Priority donation 구현 (Cont.)

▣ lock_release() 함수 수정

pintos/src/threads/synch.c

```
void lock_release (struct lock *lock)
{
    ...
    lock->holder = NULL;

    /* remove_with_lock() 함수 추가 */
    /* refresh_priority() 함수 추가 */

    sema_up (&lock->semaphore);
}
```

Priority donation 구현 (Cont.)

- remove_with_lock() 함수 추가

pintos/src/threads/thread.c

```
void remove_with_lock(struct lock *lock)
{
    /* lock 을 해지 했을때 donations 리스트에서 해당 엔트리를
       삭제 하기 위한 함수를 구현한다. */

    /* 현재 스레드의 donations 리스트를 확인하여 해지 할 lock 을
       보유하고 있는 엔트리를 삭제 한다. */

}
```

Priority donation 구현 (Cont.)

- refresh_priority() 함수 추가

pintos/src/threads/thread.c

```
void refresh_priority(void)
{
    /* 스레드의 우선순위가 변경 되었을때 donation 을 고려하여
       우선순위를 다시 결정 하는 함수를 작성 한다. */

    /* 현재 스레드의 우선순위를 기부받기 전의 우선순위로 변경 */

    /* 가장 우선수위가 높은 donations 리스트의 스레드와
       현재 스레드의 우선순위를 비교하여 높은 값을 현재 스레드의
       우선순위로 설정한다. */
}
```

Priority donation 구현 (Cont.)

- thread_set_priority() 함수 수정

pintos/src/threads/thread.c

```
void thread_set_priority (int new_priority)
{
    /* donation 을 고려하여 thread_set_priority() 함수를 수정한다 */

    /* refresh_priority() 함수를 사용하여 우선순위를 변경으로 인한
       donation 관련 정보를 갱신한다.

       donation_priority(), test_max_pariorty() 함수를 적절히
       사용하여 priority donation 을 수행하고 스케줄링 한다. */
}
```



결과

```
$ make check
```

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
7 of 27 tests failed.
```



수정 함수

```
void lock_acquire (struct lock *lock)
/* lock을 점유하고 있는 스레드와 요청 하는 스레드의 우선순위를 비교하여
priority donation을 수행하도록 수정 */

void lock_release (struct lock *lock)
/* donation list에서 스레드를 제거하고 우선순위를 다시 계산하도록
remove_with_lock(), refresh_prioriy() 함수를 호출 */

void thread_set_priority (int new_priority)
/* 스레드 우선순위 변경시 donation의 발생을 확인 하고 우선순위 변경을
위해 donation_priority() 함수 추가 */
```



추가 함수 및 자료구조 수정

```
void donate_priority(void)
    /* priority donation을 수행 */

void remove_with_lock(struct lock *lock)
    /* donation list에서 스레드 엔트리를 제거 */

void refresh_priority(void)
    /* 우선순위를 다시 계산 */
```

▣ 수정 자료구조

```
struct thread
```



10. Multi-Level Feedback Queue Scheduler

Advanced Scheduler 개요

- ▣ 과제 목표

- ◆ Multi-Level Feedback Queue 스케줄러(4.4 BSD 스케줄러와 유사) 구현

- ▣ 수정해야 할 주요 파일

- ◆ thread.* , timer.c

- ▣ 추가해야 할 파일

- ◆ pintos/src/threads/fixed_point.h

Advanced Scheduler 요구사항

▣ Priority

- ◆ 숫자가 클수록 우선순위가 높음
- ◆ 스레드 생성시 초기화 (default : 31)
- ◆ All thread : 1초 마다 priority 재 계산 (아래 수식 사용)
- ◆ Current thread : 4 clock tick 마다 priority 재 계산 (아래 수식 사용)
 - 1tick은 10msec (src/thread.c)

$$\text{priority} = \text{PRI_MAX} - (\text{recent_cpu} / 4) - (\text{nice} * 2)$$

Advanced Scheduler 요구사항

▣ Nice 값 (-20~20)

- ◆ Nice (0) : 우선순위에 영향을 주지 않음
- ◆ Nice (양수) : 우선순위를 감소
- ◆ Nice (음수) : 우선순위를 증가
- ◆ 초기 값 : 0



Advanced Scheduler 요구사항

recent_cpu

$$\text{recent_cpu} = (2 * \text{load_avg}) / (2 * \text{load_avg} + 1) * \text{recent_cpu} + \text{nice}$$

- ◆ 최근에 얼마나 많은 CPU time을 사용했는가를 표현
- ◆ init 스레드의 초기 값은 '0', 다른 스레드들은 부모의 recent_cpu값
- ◆ recent_cpu는 timer interrupt마다 1씩 증가, 매 1초마다 재 계산
- ◆ int thread_get_recent_cpu (void) 함수 구현
 - 스레드의 현재 recent_cpu의 100배 (rounded to the nearest integer) 를 반환



Advanced Scheduler 요구사항

▣ load_averge

$$\text{load_avg} = (59/60) * \text{load_avg} + (1/60) * \text{ready_threads}$$

- ◆ 최근 1분 동안 수행 가능한 프로세스의 평균 개수, exponentially weighted moving average 를 사용
- ◆ ready_threads : ready_list에 있는 스레드들과 실행 중인 스레드의 개수 (idle 스레드 제외)
- ◆ int thread_get_load_avg(void) 함수 구현
 - 현재 system load average의 100배 (rounded to the nearest integer) 를 반환
 - timer_ticks() % TIMER_FREQ == 0

Advanced Scheduler 구현

▣ 함수 기능 구현

- ◆ 아래 함수들은 빠대만 존재한다.

pintos/src/threads/thread.h

```
int thread_get_nice (void);
void thread_set_nice (int);
int thread_get_recent_cpu (void);
int thread_get_load_avg (void);
```

▣ 스케줄러를 위해 추가로 구현할 함수 선언

pintos/src/threads/thread.h

```
void mlfqs_priority (struct thread *t);
void mlfqs_recent_cpu (struct thread *t);
void mlfqs_load_avg (void);
void mlfqs_increment (void);
void mlfqs_recalc (void);
```



Advanced Scheduler 구현 (Cont.)

▣ 스케줄러를 위한 자료구조 추가

pintos/src/threads/thread.h

```
struct thread
{
    ...
    int nice;
    int recent_cpu;
    ...
}
```

추가

Advanced Scheduler 구현 (Cont.)

▣ init_thread() 함수 수정

pintos/src/threads/thread.c

```
static void init_thread (struct thread *t, const char *name,
                        int priority)
{
    ...
    t->nice = NICE_DEFAULT;
    t->recent_cpu = RECENT_CPU_DEFAULT;
}
```

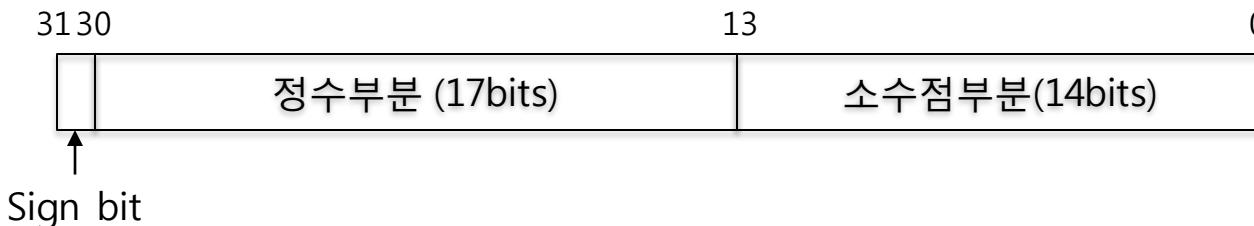
추가



Advanced Scheduler 구현 (Cont.)

▣ 소수점 연산 구현

- ◆ priority, nice, ready_threads값은 정수, recent_cpu, load_avg값은 실수이다.
- ◆ pintos는 커널에서 부동소수점 연산을 지원하지 않는다
- ◆ recent_cpu와 load_avg값을 계산을 위하여 소수점 연산이 필요하다.
- ◆ 17.14 fixed-point number representation을 이용하여 소수점 연산을 구현
 - 오른쪽 14비트를 소수점
 - 그다음 17 비트를 정수
 - 제일 왼쪽 한비트



Advanced Scheduler 구현 (Cont.)

▣ 소수점 연산 구현

- ◆ Fixed-point arithmetic operations

- n : integer x, y : fixed-point numbers f : 17.14로 표현한 1

Convert n to fixed point:	$n * f$
Convert x to integer (rounding toward zero):	x / f
Convert x to integer (rounding to nearest):	$(x + f / 2) / f$ if $x \geq 0$, $(x - f / 2) / f$ if $x \leq 0$.
Add x and y:	$x + y$
Subtract y from x:	$x - y$
Add x and n:	$x + n * f$
Subtract n from x:	$x - n * f$
Multiply x by y:	$((int64_t) x) * y / f$
Multiply x by n:	$x * n$
Divide x by y:	$((int64_t) x) * f / y$
Divide x by n:	x / n

Advanced Scheduler 구현 (Cont.)

▣ fixed point 연산 함수 구현

pintos/src/threads/fixed_point.h

```
#define F (1 << 14)      //fixed point 1
#define INT_MAX ((1 << 31) - 1)
#define INT_MIN (-(1 << 31))
// x and y denote fixed_point numbers in 17.14 format
// n is an integer

int int_to_fp(int n);           /* integer를 fixed point로 전환 */
int fp_to_int_round(int x);     /* FP를 int로 전환(반올림) */
int fp_to_int(int x);          /* FP를 int로 전환(버림) */
int add_fp(int x, int y);       /* FP의 덧셈 */
int add_mixed(int x, int n);    /* FP와 int의 덧셈 */
int sub_fp(int x, int y);       /* FP의 뺄셈(x-y) */
int sub_mixed(int x, int n);    /* FP와 int의 뺄셈(x-n) */
int mult_fp(int x, int y);      /* FP의 곱셈 */
int mult_mixed(int x, int y);   /* FP와 int의 곱셈 */
int div_fp(int x, int y);       /* FP의 나눗셈(x/y) */
int div_mixed(int x, int n);    /* FP와 int 나눗셈(x/n) */

/* 함수 본체 작성 */
```



Advanced Scheduler 구현 (Cont.)

- fixed_point.h 인클루드, 스케줄러 관련상수 정의, 변수 선언 및 초기화
pintos/src/threads/thread.c

```
#include "threads/fixed_point.h"  
...  
#define NICE_DEFAULT 0  
#define RECENT_CPU_DEFAULT 0  
#define LOAD_AVG_DEFAULT 0  
...  
int load_avg;
```

pintos/src/threads/thread.c

```
void thread_start (void)  
{  
    ...  
    thread_create ("idle", PRI_MIN, idle, &idle_started);  
    load_avg = LOAD_AVG_DEFAULT; } 추가  
    ...  
}
```

Advanced Scheduler 구현 (Cont.)

- mlfq_priority() 함수 구현
 - recent_cpu와 nice값을 이용하여 priority를 계산

pintos/src/threads/thread.c

```
void mlfqs_priority (struct thread *t)
{
    /* 해당 스레드가 idle_thread 가 아님지 검사 */

    /*priority계산식을 구현 (fixed_point.h의 계산함수 이용) */
}
```



Advanced Scheduler 구현 (Cont.)

- mlfq_s_recent_cpu() 함수 정의
 - recent_cpu 값 계산

pintos/src/threads/thread.c

```
void mlfqs_recent_cpu (struct thread *t)
{
    /* 해당 스레드가 idle_thread 가 아닌지 검사 */

    /*recent_cpu계산식을 구현 (fixed_point.h의 계산함수 이용) */
}
```

Advanced Scheduler 구현 (Cont.)

- mlfqes_load_avg() 함수 정의
 - load_avg 값 계산

pintos/src/threads/thread.c

```
void mlfqes_load_avg (void) {  
  
    /* load_avg 계산식을 구현 (fixed_point.h의 계산함수 이용) */  
    /* load_avg 는 0 보다 작아질 수 없다. */  
  
}
```

Advanced Scheduler 구현 (Cont.)

- mlfq_increment() 함수 정의
 - recent_cpu 값 1증가

pintos/src/threads/thread.c

```
void mlfqs_increment (void)
{
    /* 해당 스레드가 idle_thread 가 아닌지 검사 */
    /* 현재 스레드의 recent_cpu 값을 1증가 시킨다. */
}
```

Advanced Scheduler 구현 (Cont.)

- mlfq_s_recalc() 함수 정의
 - 모든 thread의 recent_cpu와 priority값 재계산

pintos/src/threads/thread.c

```
void mlfqs_recalc (void)
{
    /* 모든 thread의 recent_cpu와 priority값 재계산 한다. */
}
```



Advanced Scheduler 구현 (Cont.)

▣ thread_set_priority() 함수 수정

- ◆ mlfqs 스케줄러를 활성 하면 thread_mlfqs 변수는 true로 설정됨
- ◆ Advanced scheduler에서는 우선순위를 임의로 변경할 수 없다.

pintos/src/threads/thread.c

```
void thread_set_priority (int new_priority)
{
    /* mlfqs 스케줄러 일때 우선순위를 임의로 변경할수 없도록 한다. */
    ...
}
```



Advanced Scheduler 구현 (Cont.)

▣ thread_set_nice() 함수 구현

- ◆ 현재 thread의 nice값을 nice로 설정

pintos/src/threads/thread.c

```
void thread_set_nice (int nice UNUSED)
{
    /* 현재 스레드의 nice값을 변경하는 함수를 구현하다.
       해당 작업중에 인터럽트는 비활성화 해야 한다. */

    /* 현재 스레드의 nice 값을 변경한다.
       nice 값 변경 후에 현재 스레드의 우선순위를 재계산 하고
       우선순위에 의해 스케줄링 한다. */
}
```

Advanced Scheduler 구현 (Cont.)

- thread_get_nice() 함수 구현
 - 현재 thread의 nice 값을 반환

pintos/src/threads/thread.c

```
int thread_get_nice (void)
{
    /* 현재 스레드의 nice 값을 반환한다.
       해당 작업중에 인터럽트는 비활성되어야 한다. */
}
```



Advanced Scheduler 구현 (Cont.)

- thread_get_load_avg() 함수 구현
 - load_avg에 100을 곱해서 반환

pintos/src/threads/thread.c

```
int thread_get_load_avg (void)
{
    /* load_avg에 100을 곱해서 반환 한다.
       해당 과정중에 인터럽트는 비활성되어야 한다. */
}
```



Advanced Scheduler 구현 (Cont.)

- thread_get_recent_cpu() 함수 구현
 - recent_cpu에 100을 곱해서 반환

pintos/src/threads/thread.c

```
int thread_get_recent_cpu (void)
{
    /* recent_cpu 에 100을 곱해서 반환 한다.
       해당 과정중에 인터럽트는 비활성되어야 한다. */
}
```



Advanced Scheduler 구현 (Cont.)

▣ timer_interrupt() 함수 수정

- ◆ 1초마다 load_avg, 모든 스레드의 recent_cpu, priority 재 계산
- ◆ 4tick마다 현재 스레드의 priority 재 계산

pintos/src/device/timer.c

```
static void timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick ();

    /* mlfqs 스케줄러일 경우
       timer_interrupt 가 발생할때마다 recent_cpu 1증가,
       1초마다 load_avg, recent_cpu, priority 계산,
       매 4tick마다 priority 계산 */
    ...
}
```



Advanced Scheduler 구현 (Cont.)

▣ lock_acquire() 함수 수정

- ◆ mlfqs 스케줄러를 사용시 priority donation 사용 금지

pintos/src/threads/synch.c

```
void lock_acquire (struct lock *lock) {  
    ...  
    /* mlfqs 스케줄러 활성화시 priority donation 관련 코드 비활성화 */  
    ...  
}
```



Advanced Scheduler 구현 (Cont.)

▣ lock_release() 함수 수정

- ◆ mlfqs 스케줄러를 사용시 priority donation 사용 금지

pintos/src/threads/synch.c

```
void lock_release (struct lock *lock)
{
    ...
    /* mlfqs 스케줄러 활성화시 priority donation 관련 코드 비활성화 */
    ...
}
```



결과

```
$ make check
```

```
pass tests/threads/mlfqs-block
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
```



수정 함수

```
void thread_set_priority(int new_priority)
    /* 현재 수행중인 스레드의 우선순위를 변경 */

static void timer_interrupt(struct intr_frame *args UNUSED)
    /* timer interrupt 핸들러 */
```



추가 함수

```
void mlfqs_priority (struct thread *t)
    /* 인자로 주어진 스레드의 priority를 업데이트 */

void mlfqs_recent_cpu (struct thread *t)
    /* 인자로 주어진 스레드의 recent_cpu를 업데이트 */

void mlfqs_load_avg (void)
    /* 시스템의 load_avg를 업데이트 */

void mlfqs_increment (void)
    /* 현재 수행중인 스레드의 recent_cpu를 1증가 시킴 */

void mlfqs_recalc (void)
    /* 모든 스레드의 priority, recent_cpu를 업데이트 */
```



12. Virtual Memory

핀토스의 메모리 개요

- ▣ 현재 주소공간은 4개의 세그먼트로 구성

- ◆ Stack
- ◆ Initialized data
- ◆ Uninitialized data
- ◆ Code
- ◆ **Heap은 현재 없음 !**

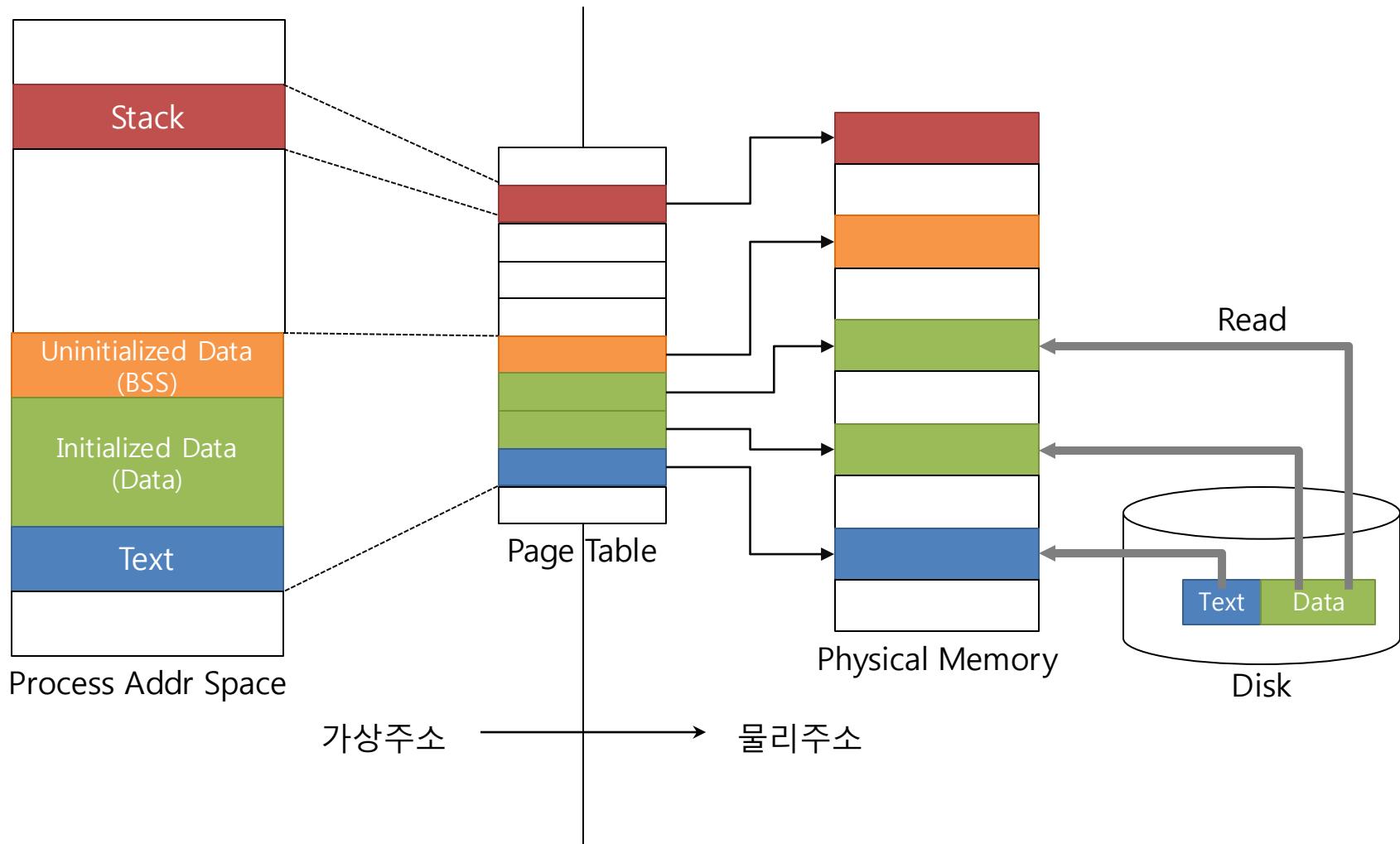
- ▣ 프로세스의 메모리 탑재 과정

- ◆ 각 세그멘트(Stack, Data, BSS, Code)가 물리페이지에 탑재
- ◆ 페이지 테이블 초기화



Pintos의 프로세스 주소 공간

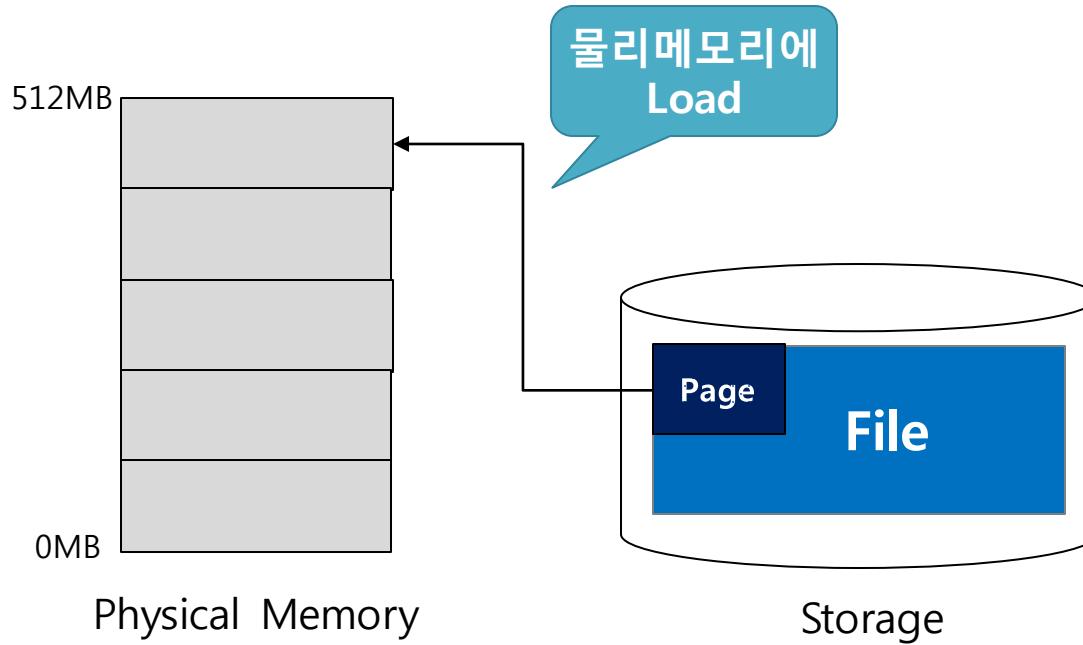
- ▣ 과제 수행 전 pintos memory layout



- ▣ Swap을 사용할 수 없음
- ▣ Demand paging 사용 불가
- ▣ Virtual memory가 구현되어 있지 않음!!!
- ▣ Pintos에서 virtual memory를 구현해 봅시다!!!

기본 개념: Demand Paging

요구 페이징 (Demand Paging) 이란?

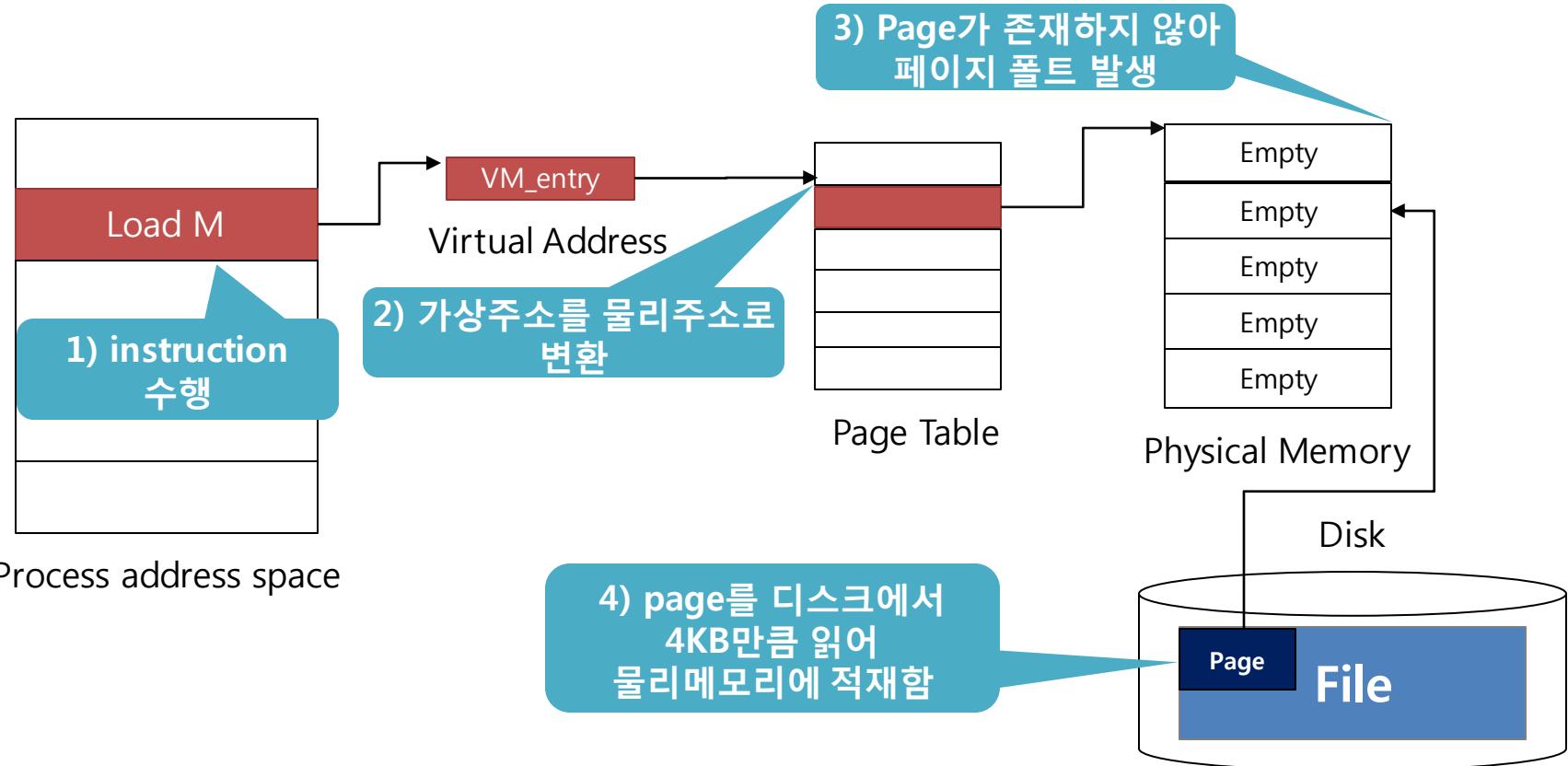


- 요구된 Page들만 물리메모리에 load 하는 기법

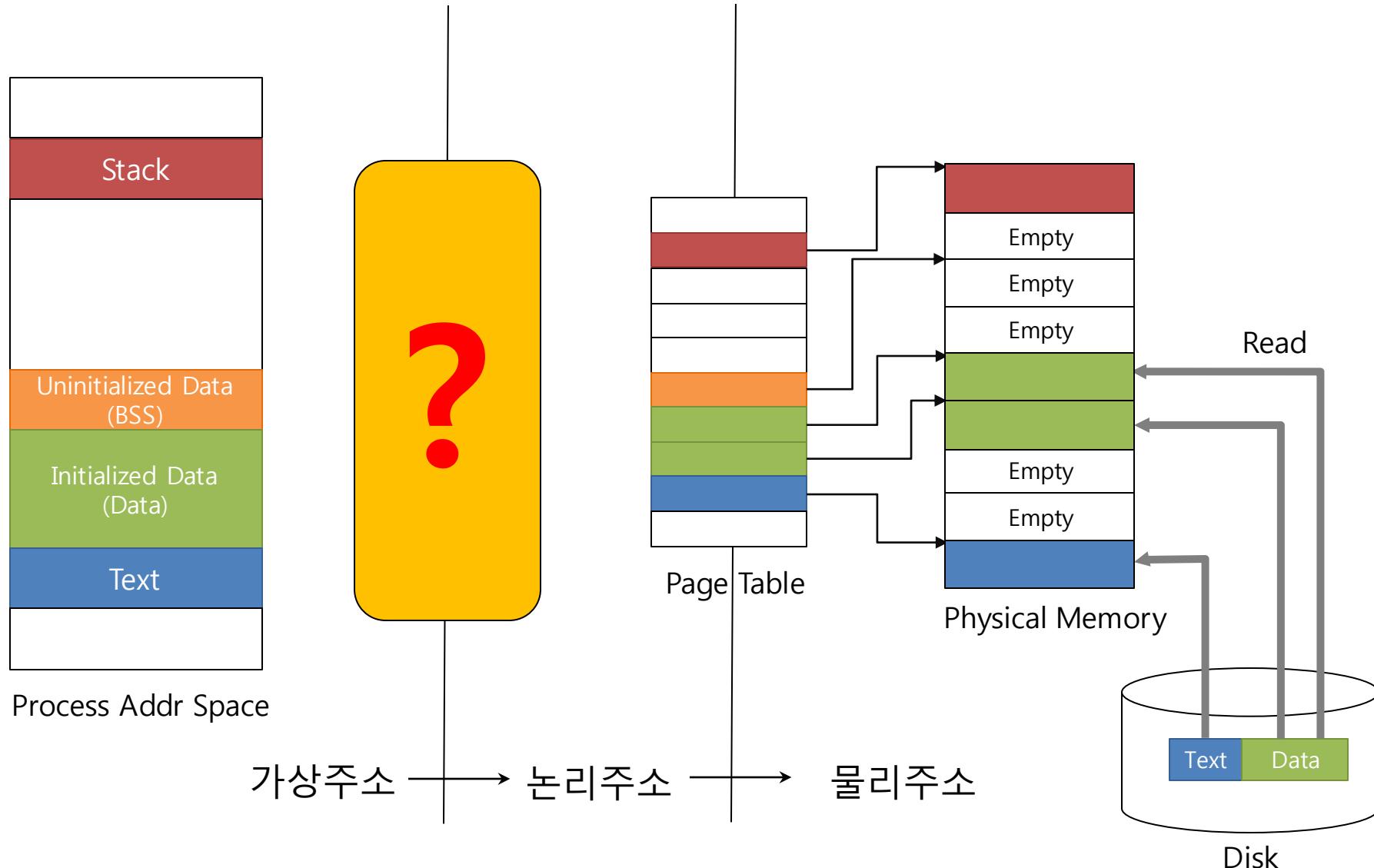
요구 페이징 (Demand Paging) 과정 (Cont.)

1. Instruction 실행
2. 가상주소로부터 가상 페이지 번호 추출
3. 페이지 테이블 참조
4. 페이지 테이블에 물리 페이지 부재시 페이지 폴트 발생
5. 페이지 폴트 발생시 페이지 프레임 할당하고 페이지 테이블 갱신
6. 해당 페이지를 디스크에서 페이지 프레임에 탑재

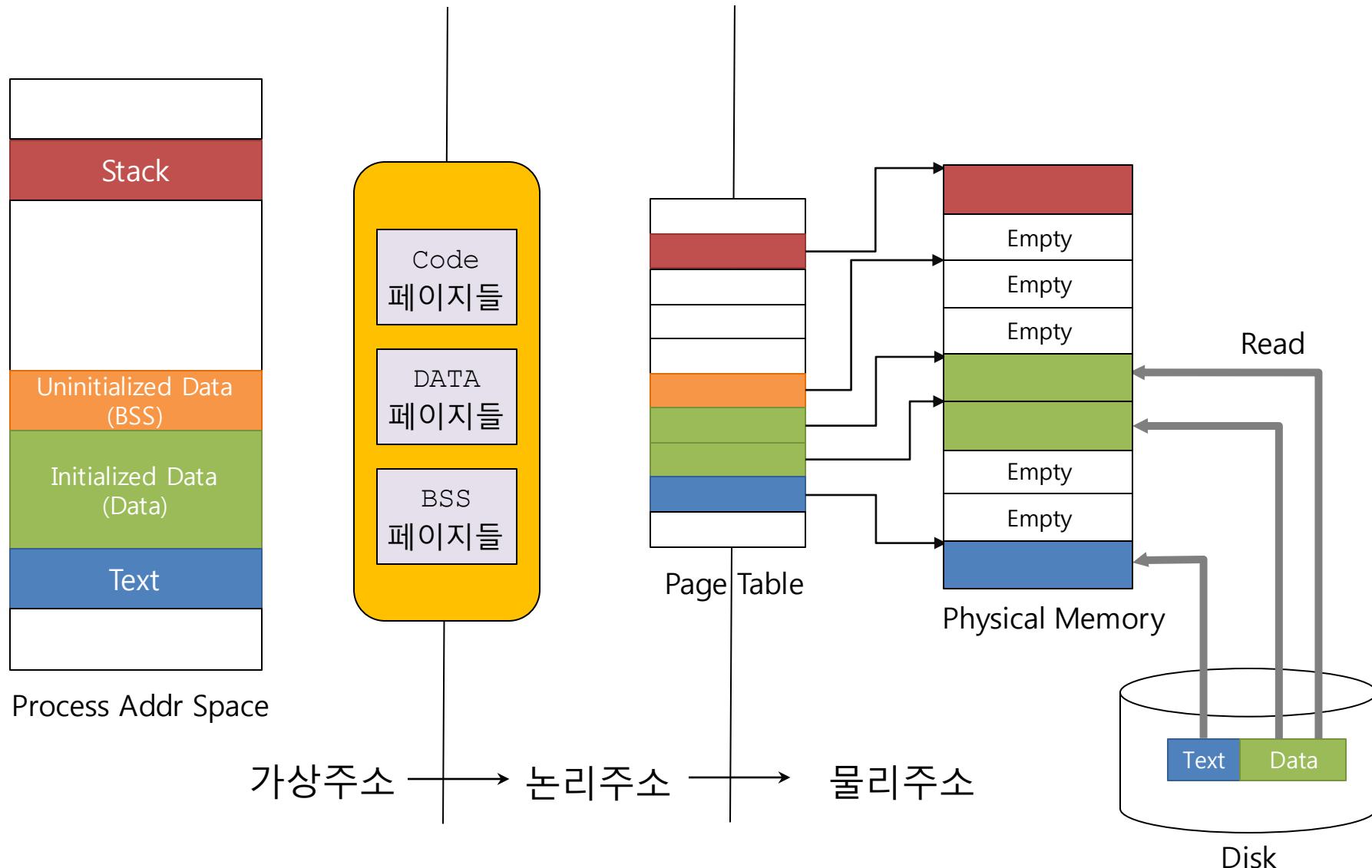
요구 페이징 (Demand Paging) 과정



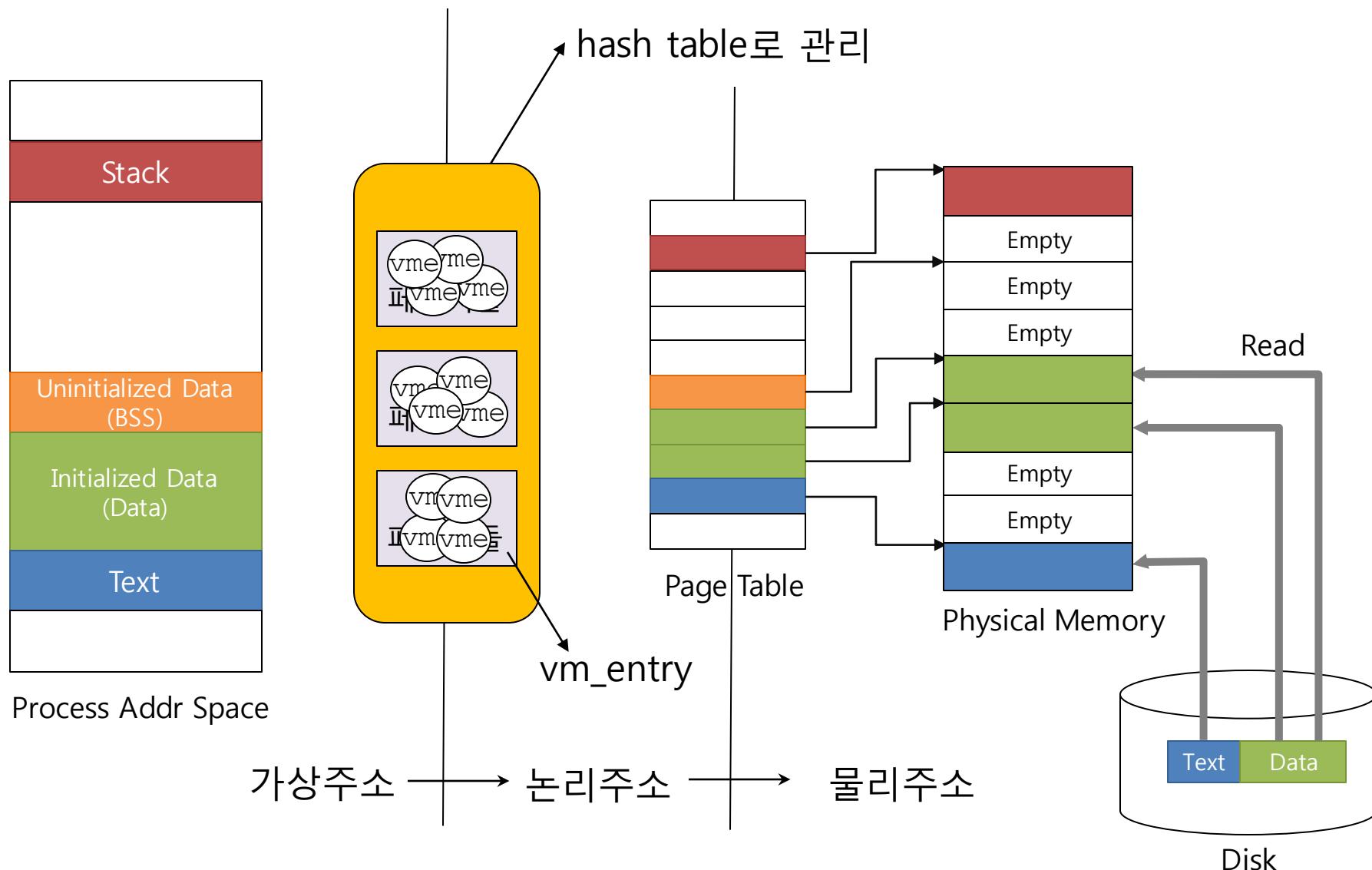
할일 1: 가상주소 공간의 구현



할일 1: 가상주소 공간의 구현 (Cont.)



할일 1: 가상주소 공간의 구현 (Cont.)



할일 2: 가상주소 공간에서의 페이지

- ▣ 가상주소 페이지 별 자료구조 정의
- ▣ vm_entry
 - ◆ 페이지당 하나
 - ◆ 각 페이지의 파일 포인터, 오프셋, 크기를 저장
 - ◆ 프로그램 초기 탑재시 가상 주소공간 각 페이지에 vm_entry 할당
 - ◆ 프로그램 실행시
 - 페이지 테이블 탐색
 - 페이지폴트 시, 가상주소에 해당하는 vm_entry를 탐색
 - vm_entry에 없는 가상 주소는 Segmentaion fault
 - vm_entry가 존재할 경우,
 - 페이지 프레임 할당
 - vm_entry에 있는 파일포인터, 읽기 시작할 오프셋, 읽어야 할 크기 등을 참조해서 페이지 로드
 - 페이지 테이블 갱신



할일 3: 가상주소 공간의 초기화

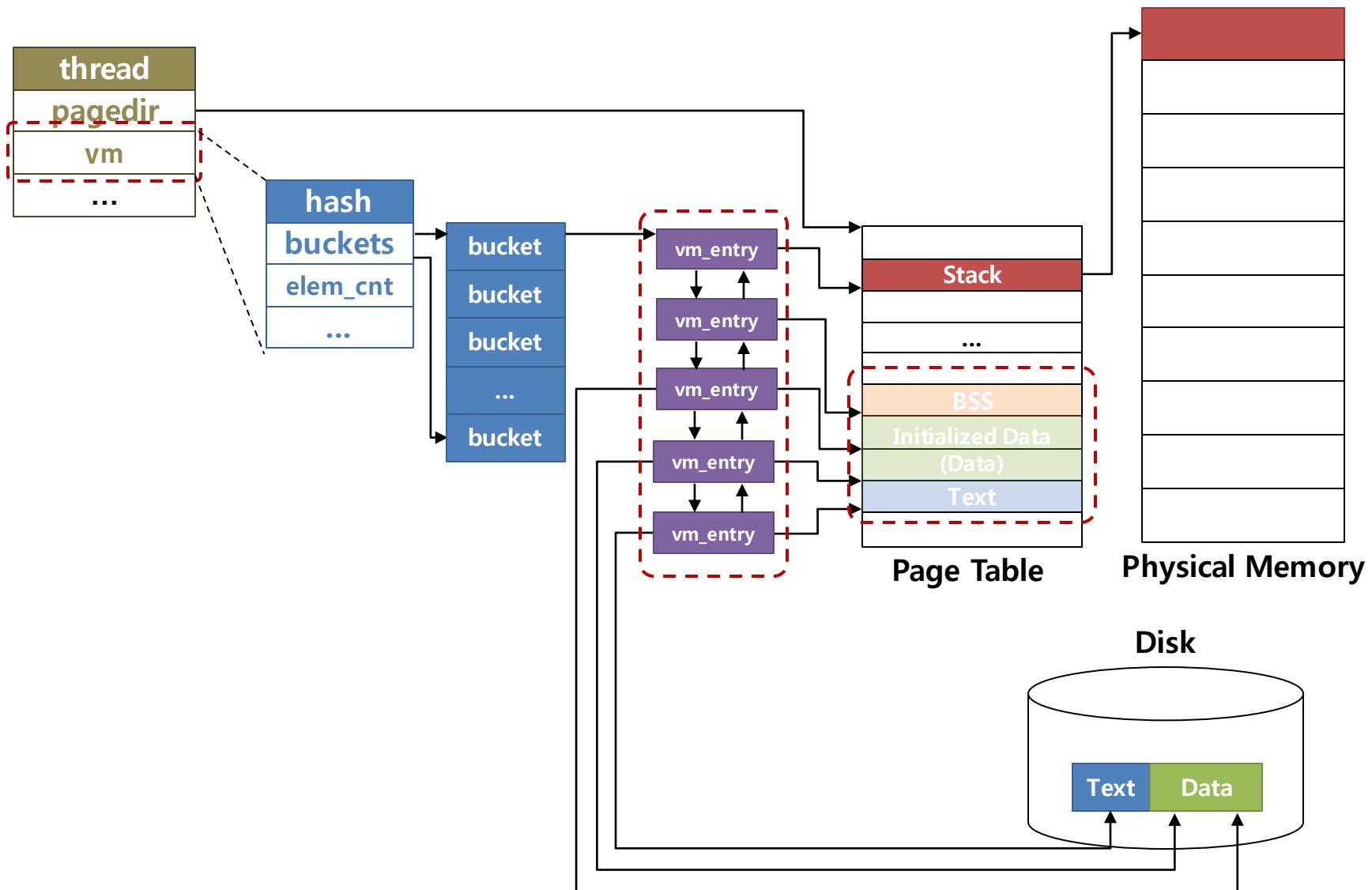
- ▣ 기존의 핀토스의 가상주소 공간 초기화 과정
 - ◆ ELF 이미지 각 페이지를 물리메모리로 읽어들임.
 - load_segment()로 Data, Code 세그먼트 읽음
 - setup_stack()로 Stack에 물리페이지 할당
- ▣ 수정된 핀토스의 가상주소 공간 초기화 과정
 - ◆ 디스크 이미지의 세그먼트를 전부 올리는 것은 물리 메모리의 낭비를 초래
 - ◆ 물리메모리 할당 대신, 가상 페이지마다 vm_entry를 통해 적재할 정보들만 관리

할일 4: 요구페이지를 위한 페이지 폴트 수정

▣ Demand Paging 구현

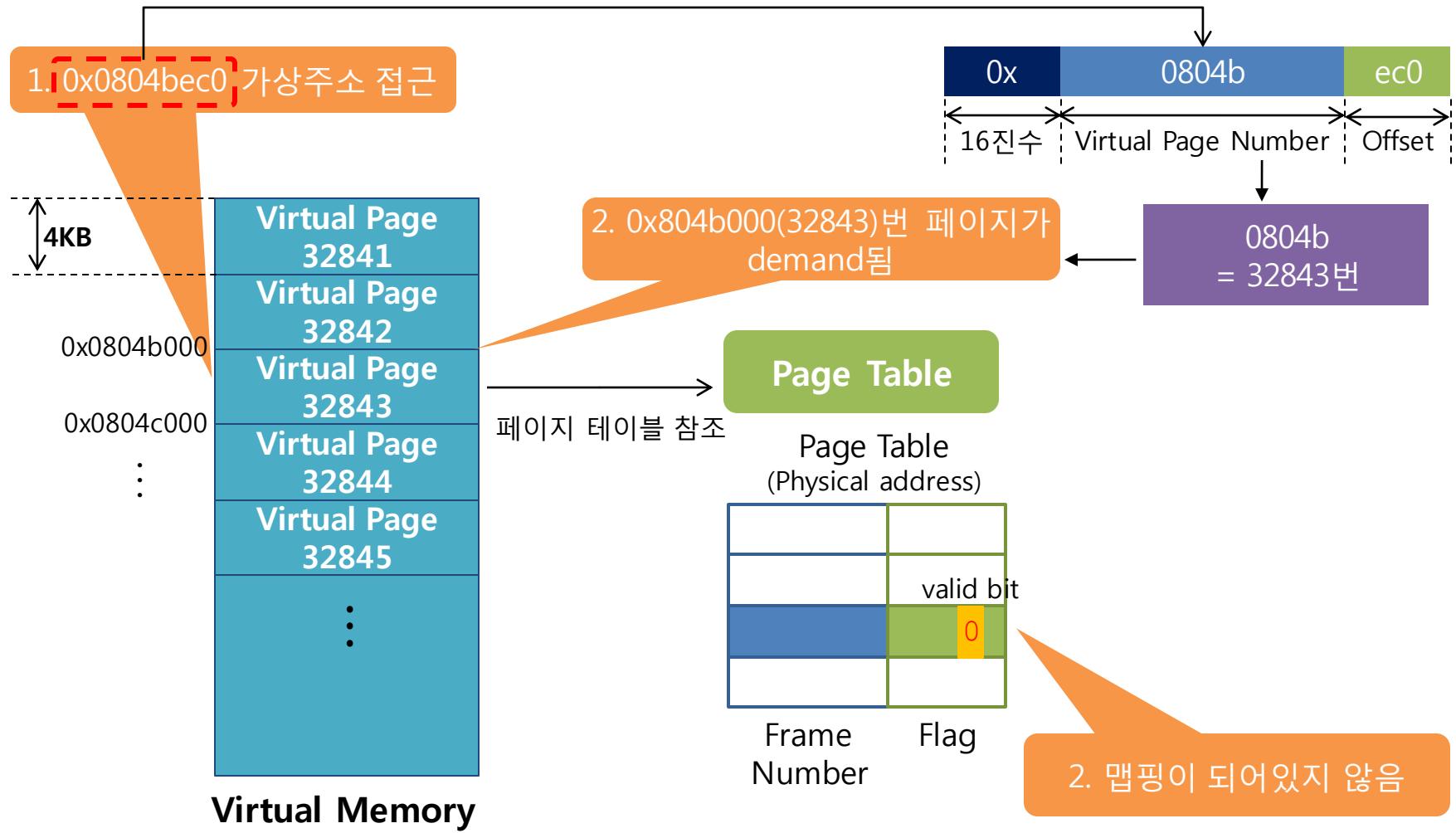
- ◆ 프로세스가 요청한 페이지들에 대해서만 물리페이지를 할당해주는 기법
 - 현재 Pintos는 프로그램의 모든 세그먼트에 대해 물리페이지 할당
 - 페이지 폴트 발생 시, 강제 종료(kill)
 - Demand Paging을 위해 요청한 페이지에 대해서만 물리페이지 할당을 수행
 - 페이지 폴트 발생 시, 해당 vm_entry의 존재 유무 확인
 - vm_entry의 가상주소에 해당하는 물리페이지 할당
 - vm_entry의 정보를 참조하여, 디스크에 저장되어 있는 실제 데이터 로드

구현 후 Pintos 가상메모리

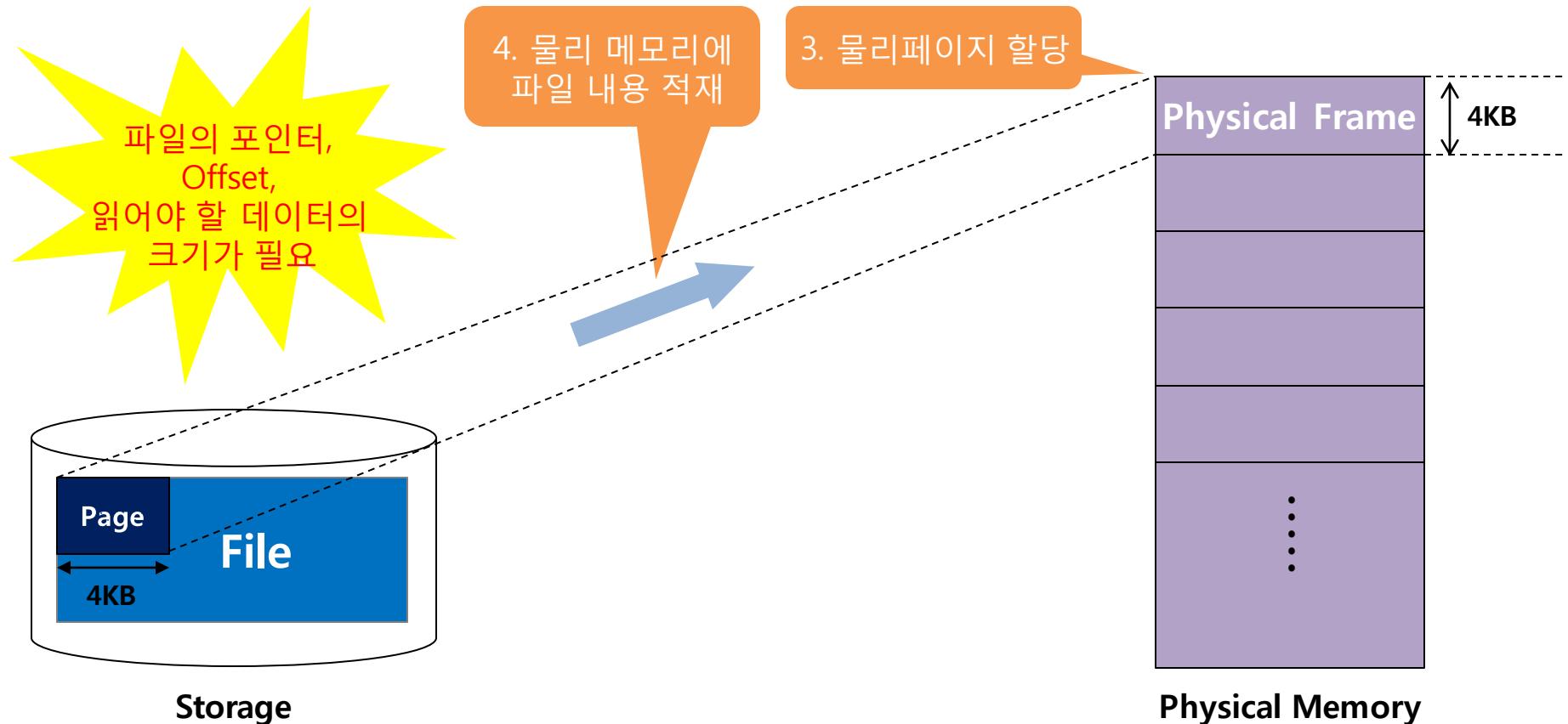


vm_entry

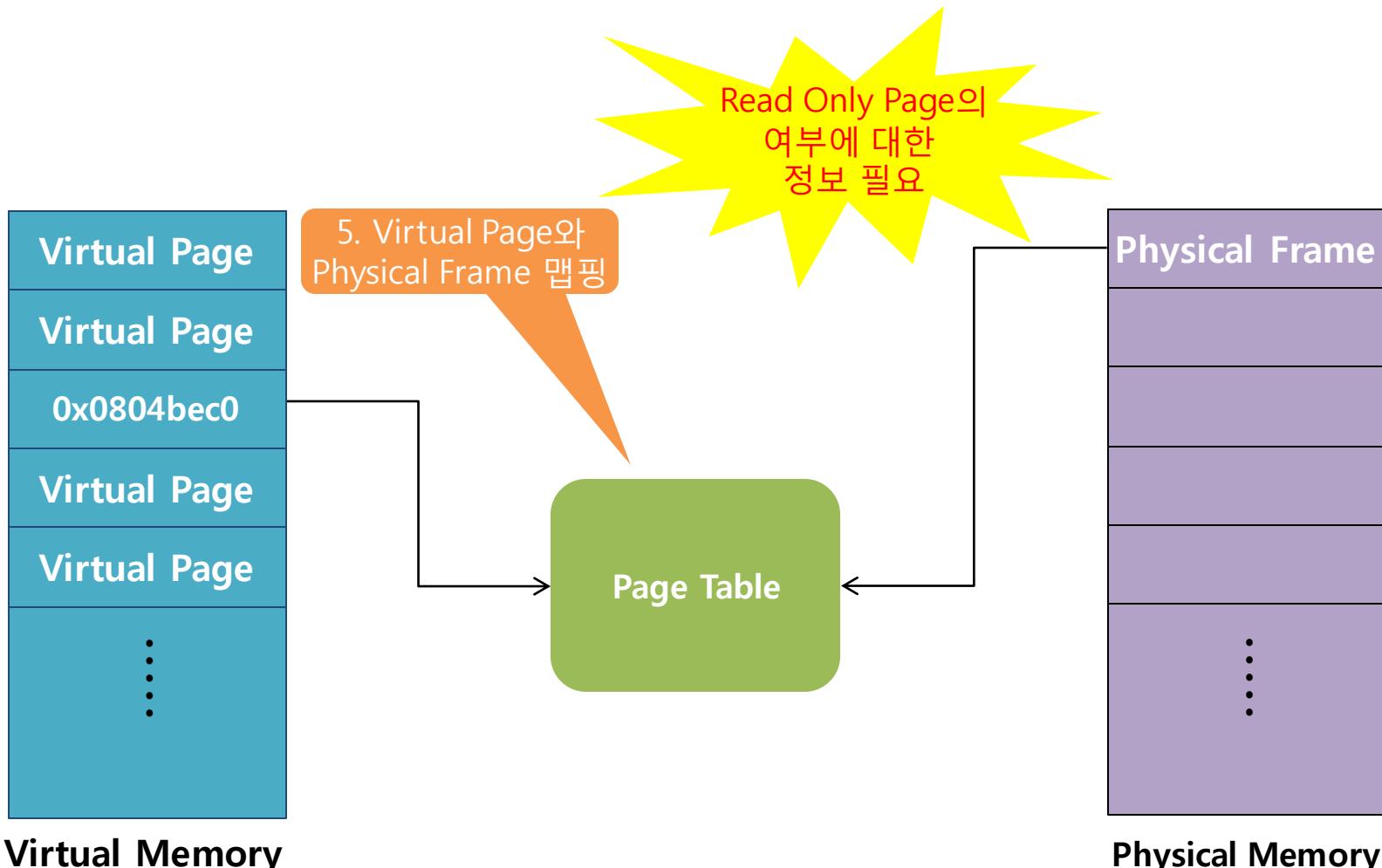
vm_entry 등장의 Motivation



vm_entry 등장의 Motivation (Cont.)

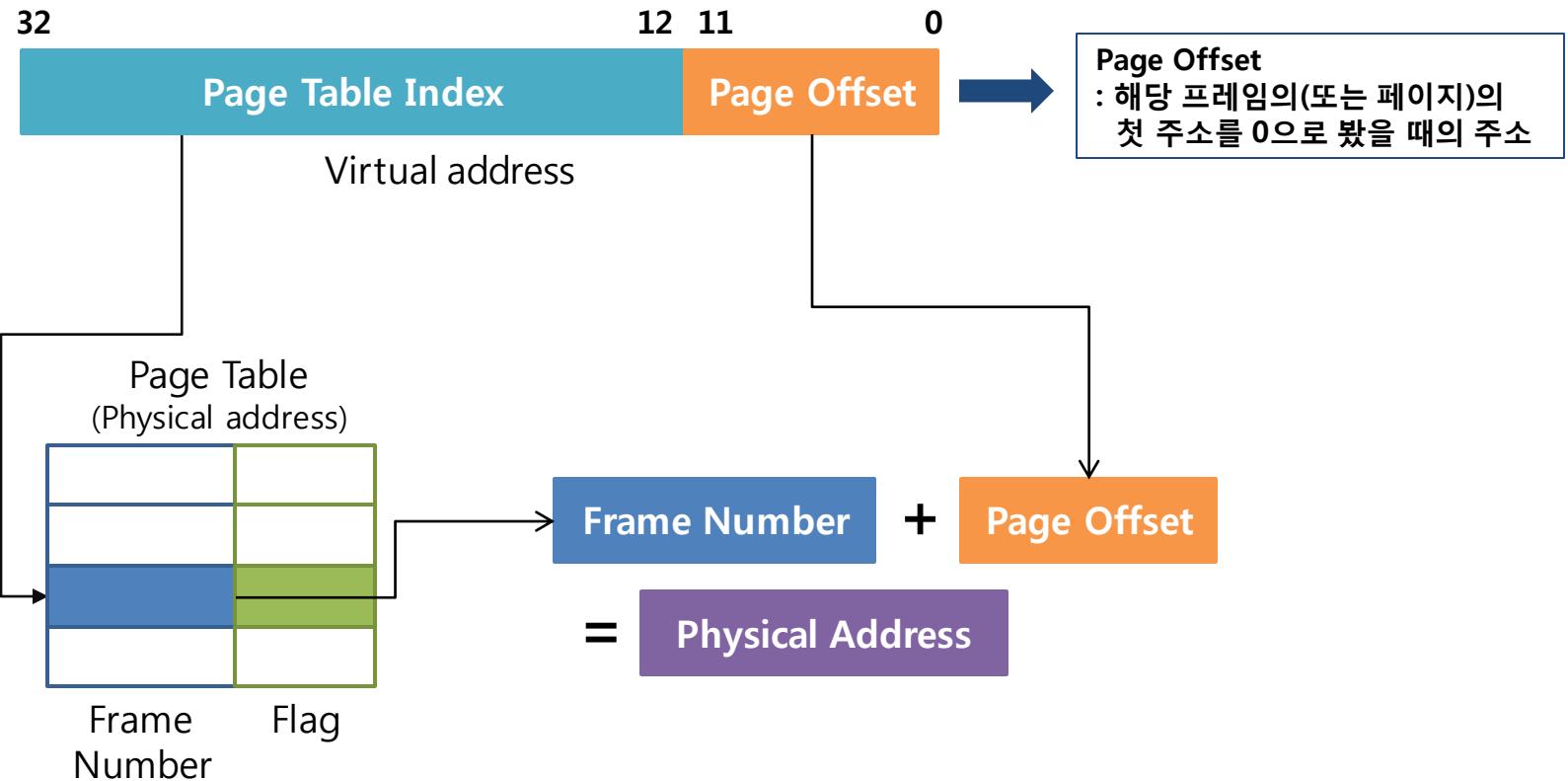


vm_entry 등장의 Motivation (Cont.)

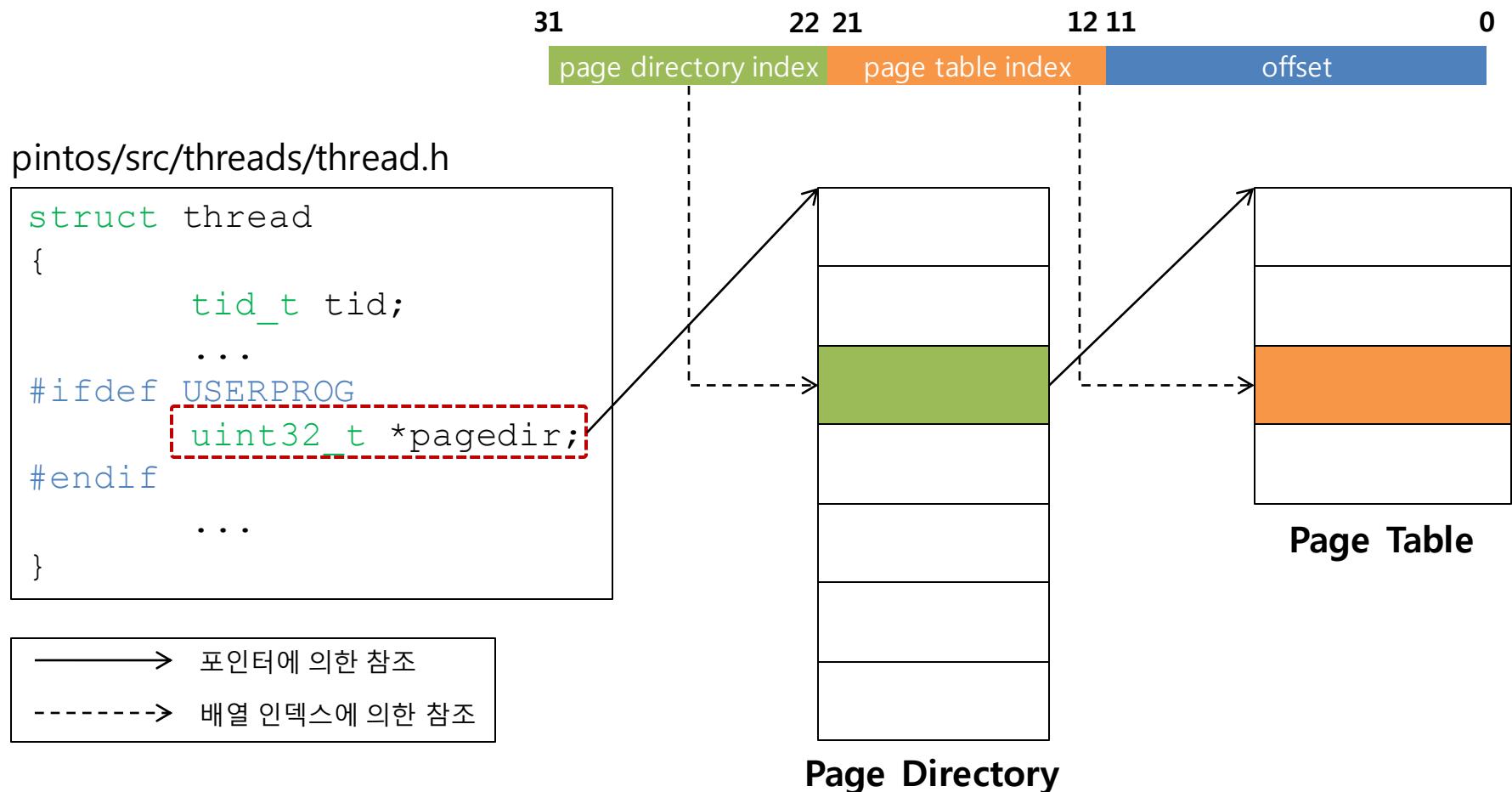


물리페이지의 할당 시에 파일의 Offset에 대한 정보를 읽을 필요가 있음

가상 주소 변환(Translation) 과정



Pintos의 페이지 테이블 구조 (2단계 페이지테이블 구조사용)



vm_entry 의 타입

▣ 가상 주소 페이지를 3가지 타입으로 분류

- ◆ vm_entry의 type 필드에 가상 주소의 타입을 저장
- ◆ VM_BIN: 바이너리 파일로 부터 데이터를 로드
- ◆ VM_FILE: 매핑된 파일로 부터 데이터를 로드
 - Memory Mapped File 과제에서 다룰 예정
- ◆ VM_ANON: 스왑영역으로 부터 데이터를 로드
 - Swapping 과제에 다룰 예정

pintos/src/vm/page.h

```
#define VM_BIN          0
#define VM_FILE          1
#define VM_ANON          2
```



vm_entry 자료구조 정의

pintos/src/vm/page.h

```
struct vm_entry{
    uint8_t type;                      /* VM_BIN, VM_FILE, VM_ANON의 타입 */
    void *vaddr;                       /* vm_entry가 관리하는 가상페이지 번호 */
    bool writable;                     /* True일 경우 해당 주소에 write 가능
                                         False일 경우 해당 주소에 write 불가능 */
    bool is_loaded;                    /* 물리메모리의 탑재 여부를 알려주는 플래그 */
    struct file* file;                /* 가상주소와 맵핑된 파일 */

    /* Memory Mapped File에서 다룰 예정 */
    struct list_elem mmap_elem; /* mmap 리스트 element */

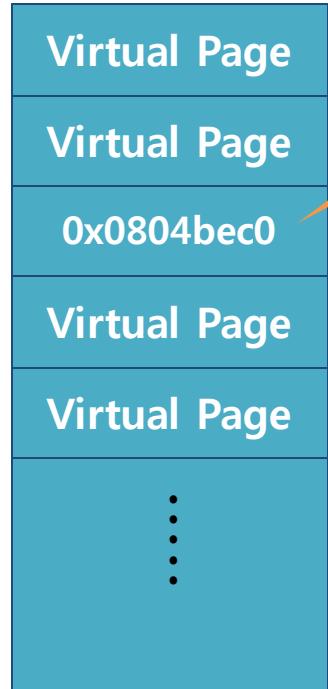
    size_t offset;                     /* 읽어야 할 파일 오프셋 */
    size_t read_bytes;                /* 가상페이지에 쓰여져 있는 데이터 크기 */
    size_t zero_bytes;                /* 0으로 채울 남은 페이지의 바이트 */

    /* Swapping 과정에서 다룰 예정 */
    size_t swap_slot;                /* 스왑 슬롯 */

    /* 'vm_entry들을 위한 자료구조' 부분에서 다룰 예정 */
    struct hash_elem elem;           /* 해시 테이블 Element */
}
```

vm_entry들을 관리하기 위한 자료구조

vm_entry 관리 Motivation



1. 해당 주소에 접근

페이지 테이블 참조

2. 페이지테이블에
정보가 없음
(맵핑이 필요)

즉 페이지 폴트가 일어날 때마다
가상 주소에 해당하는 vm_entry를 탐색해야 함

해당 가상주소의
Virtual Page를 표현하는
vm_entry 필요

Page Table

3. 가상페이지에 대한
정보가 필요

Virtual Memory

vm_entry 들의 관리

- ▣ 메모리 접근시 해당 주소의 가상페이지를 표현하는 vm_entry 탐색
 - ◆ vm_entry들은 탐색이 가능하도록 묶어서 관리되어야 함
- ▣ 탐색이 빠른 해시로 vm_entry 관리
 - ◆ vaddr으로 해시 값 추출



Pintos에서의 해시 테이블

- ▣ Pintos는 체이닝 해시 테이블을 제공
 - ◆ 체이닝 해시 테이블
 - 해시 값이 충돌할 경우, 충돌한 해시 값의 element들을 리스트로 관리
 - ◆ src/lib/kernel/hash.* 에 자료구조와 해시테이블을 관리하는 함수 정의

Pintos가 제공하는 해시테이블 인터페이스

- ▣ `bool hash_init (struct hash *h ,hash_hash_func *, hash_less_func *, void *aux)`
 - ◆ 해시 테이블을 초기화 해주는 함수
 - ◆ h : 초기화 할 Hash table
 - ◆ hash_hash_func : 해시값을 구해주는 함수의 포인터
 - ◆ hash_less_func : 해시 element 들의 크기를 비교해주는 함수의 포인터
 - hash_find()에서 사용
- ▣ `void hash_destroy (struct hash *, hash_action_func *)`
 - ◆ 해시 테이블을 삭제하는 함수
 - ◆ hash_action_func : hash bucket의 entry를 삭제 해주는 함수.
- ▣ `struct hash_elem *hash_insert (struct hash *, struct hash_elem *)`
 - ◆ 해시 테이블에 Element 삽입
- ▣ `struct hash_elem *hash_delete (struct hash *, struct hash_elem *)`
 - ◆ 해시 테이블에서 Element 제거
- ▣ `struct hash_elem *hash_find (struct hash *, struct hash_elem *)`
 - ◆ 해시 테이블에서 Element 검색

해시테이블을 이용해서 구현해야 할 부분

- ▣ thread 구조체에 해시 테이블 자료구조 추가
- ▣ 프로세스 생성시
 - ◆ 해시 테이블 초기화
 - ◆ vm_entry들을 해시 테이블에 추가
- ▣ 프로세스 실행 중
 - ◆ 페이지 폴트가 발생 시, vm_entry를 해시 테이블에서 탐색
- ▣ 프로세스 종료시
 - ◆ 해시테이블의 버킷리스트와 vm_entry들 제거



thread 구조체에 해시 테이블 자료구조 추가

```
struct thread
```

프로세스마다 가상주소 공간이 할당되므로, 가상페이지들을 관리할 수 있는 자료구조인 해시테이블 정의

pintos/src/threads/thread.h

```
struct thread{
    /* Owned by thread.c. */
    tid_t tid;                      /* Thread identifier. */
    enum thread_status status;       /* Thread state. */
    ...
    /* Owned by thread.c. */
    unsigned magic;                 /* Detects stack overflow. */

    struct hash vm;                /* 스레드가 가진 가상 주소 공간을 관리하는 해시테이블 */
}
```



해시 테이블 초기화 함수 구현

- ▣ `void vm_init (struct hash *vm)`
 - ◆ `hash_init()` 함수를 사용하여 해시 테이블 초기화
 - `hash_init()` 함수 설명 참조
- ▣ `static unsigned vm_hash_func (const struct hash_elem *e, void *aux)`
 - ◆ `vm_entry`의 `vaddr`을 인자값으로 `hash_int()` 함수를 사용하여 해시 값 반환
- ▣ `static bool vm_less_func (const struct hash_elem *a, const struct hash_elem *b, void *aux)`
 - ◆ 입력된 두 `hash_elem`의 `vaddr` 비교
 - `a`의 `vaddr`이 `b`보다 작을 시 `true` 반환
 - `a`의 `vaddr`이 `b`보다 클 시 `false` 반환

해시 테이블 초기화 및 제거 및 해시 함수 구현 (Cont.)

pintos/src/vm/page.c

```
void vm_init (struct hash *vm)
{
    /* hash_init()으로 해시테이블 초기화 */
    /* 인자로 해시 테이블과 vm_hash_func과 vm_less_func 사용 */
}
```

pintos/src/vm/page.c

```
static unsigned vm_hash_func (const struct hash_elem *e, void *aux)
{
    /* hash_entry()로 element에 대한 vm_entry 구조체 검색 */
    /* hash_int()를 이용해서 vm_entry의 멤버 vaddr에 대한 해시값을
       구하고 반환 */
}
```

pintos/src/vm/page.c

```
static bool vm_less_func (const struct hash_elem *a, const struct
hash_elem *b)
{
    /* hash_entry()로 각각의 element에 대한 vm_entry 구조체를 얻은
       후 vaddr 비교 (b가 크다면 true, a가 크다면 false) */
}
```



해시 테이블 초기화 코드 추가

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    ...
    /* vm_init() 함수를 이용해서 해시테이블 초기화 */

    /* Initialize interrupt frame and load executable */
    memset (&if_, 0, sizeof if_);
    ...
}
```



해시 테이블에 element를 삽입 및 제거 함수 구현

- ▣ `bool insert_vme (struct hash *vm, struct vm_entry
*vme)`
 - ◆ `hash_insert()` 함수를 이용하여 `vm_entry`를 해시 테이블에 삽입
 - ◆ 삽입 성공 시 `true` 반환
 - ◆ 실패 시 `false` 반환
- ▣ `bool delete_vme (struct hash *vm, struct vm_entry
*vme)`
 - ◆ `hash_delete()` 함수를 이용하여 `vm_entry`를 해시 테이블에서 제거

해시 테이블에 element를 삽입 및 제거 함수 구현 (Cont.)

pintos/src/vm/page.c

```
bool insert_vme (struct hash *vm, struct vm_entry *vme)
{
    /* hash_insert() 함수 사용 */
}
```

pintos/src/vm/page.c

```
bool delete_vme (struct hash *vm, struct vm_entry *vme)
{
    /* hash_delete() 함수 사용 */
}
```

‘가상 주소공간 초기화’ 부분에서 사용 예정



해시 테이블 내 vm_entry 검색 함수 구현

- ▣ `struct vm_entry *find_vme (void *vaddr)`
 - ◆ 인자로 받은 vaddr에 해당하는 vm_entry를 검색 후 반환
 - 가상 메모리 주소에 해당하는 페이지 번호 추출 (pg_round_down())
 - hash_find() 함수를 이용하여 vm_entry 검색 후 반환

해시테이블 인터페이스 구현

pintos/src/vm/page.c

```
struct vm_entry *find_vme (void *vaddr)
{
    /* pg_round_down()으로 vaddr의 페이지 번호를 얻음 */
    /* hash_find() 함수를 사용해서 hash_elem 구조체 얻음 */
    /* 만약 존재하지 않는다면 NULL 리턴 */
    /* hash_entry()로 해당 hash_elem의 vm_entry 구조체 리턴 */
}
```

‘요구 페이징 구현’ 부분에서 사용 예정



해시 테이블 제거 함수 구현

- ▣ `void vm_destroy (struct hash *vm)`
 - ◆ `hash_destroy()` 함수를 사용하여 해시 테이블의 버킷리스트와 `vm_entry`들을 제거

pintos/src/vm/page.c

```
void vm_destroy (struct hash *vm)
{
    /* hash_destroy() 으로 해시테이블의 버킷리스트와 vm_entry들을 제거 */
}
```

process_exit() 함수 수정

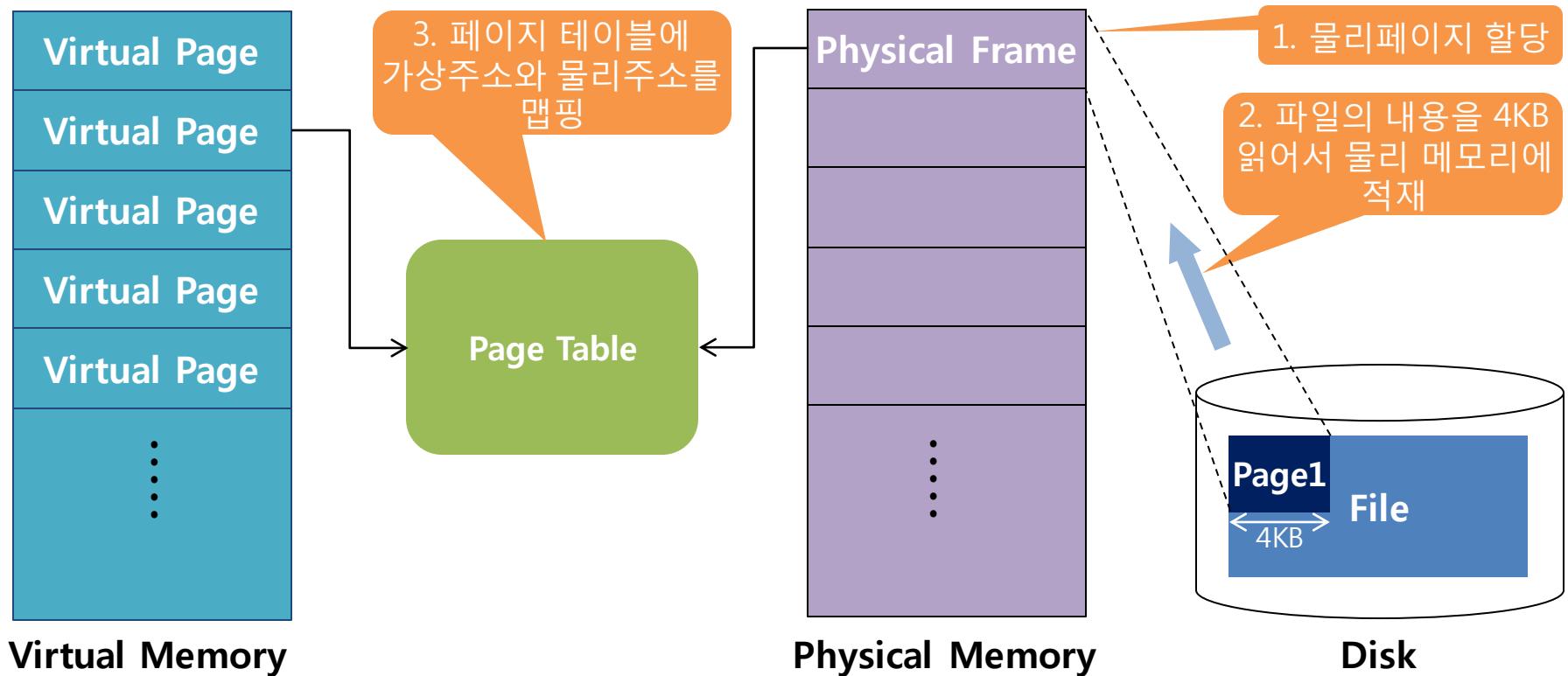
- ▣ 프로세스 종료 시 vm_entry들을 제거하도록 코드 수정

pintos/src/userprog/process.c

```
void process_exit (void) {
    struct thread *cur = thread_current();
    uint32_t *pd;
    ...
    palloc_free_page(cur -> fd);
    /* vm_entry들을 제거하는 함수 추가 */
    pd = cur->pagedir;
    ...
}
```

가상 주소공간 초기화

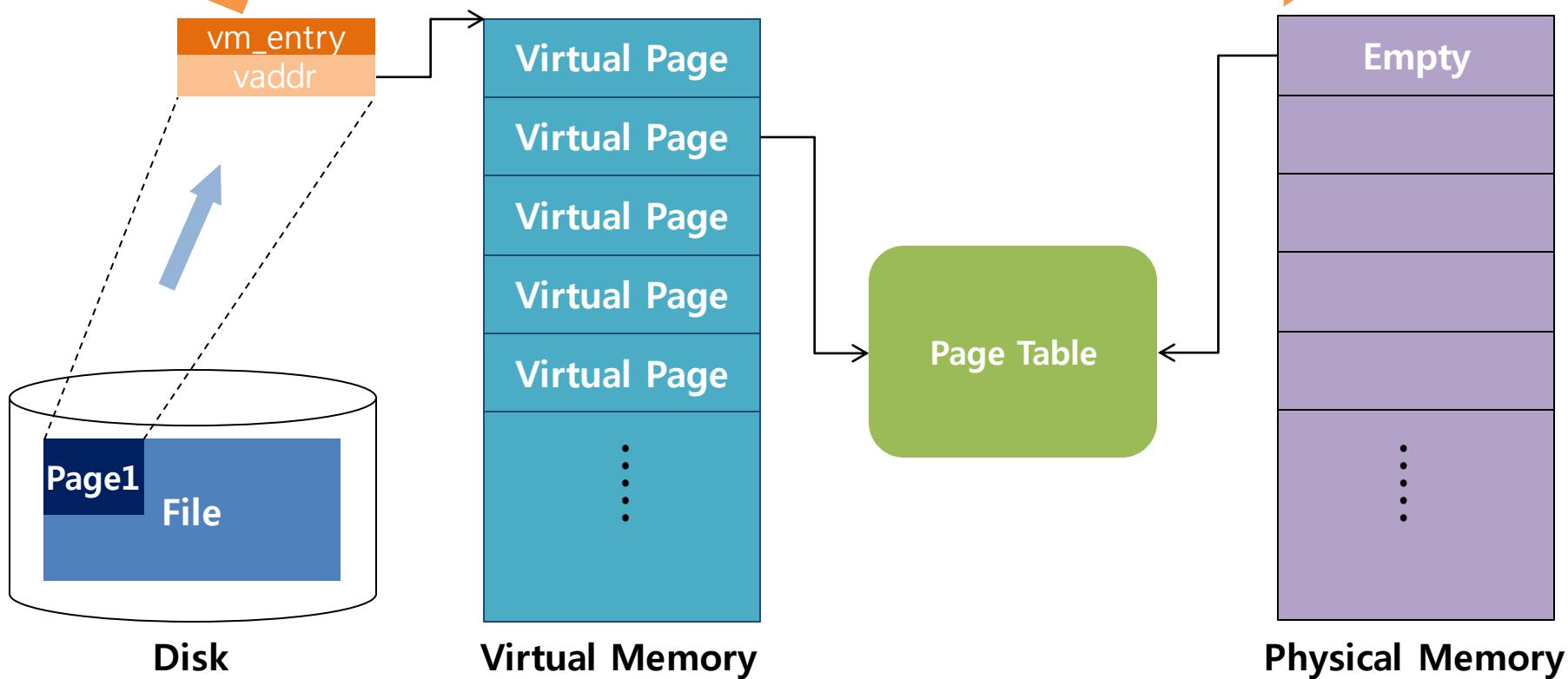
가상 주소공간 초기화 구현 전 Pintos



가상 주소공간 초기화 구현 후 Pintos

파일의 포인터, 파일의 오프셋,
읽어야 할 크기 등 정보를
vm_entry에 저장

물리페이지는 가상주소에
접근할 때 할당한다



가상주소 접근 시, 물리페이지가 맵핑되어 있지 않다면 해당 가상 주소에 해당하는 vm_entry 탐색 후
vm_entry 정보들을 참조하여 디스크의 데이터를 읽어 물리프레임에 적재

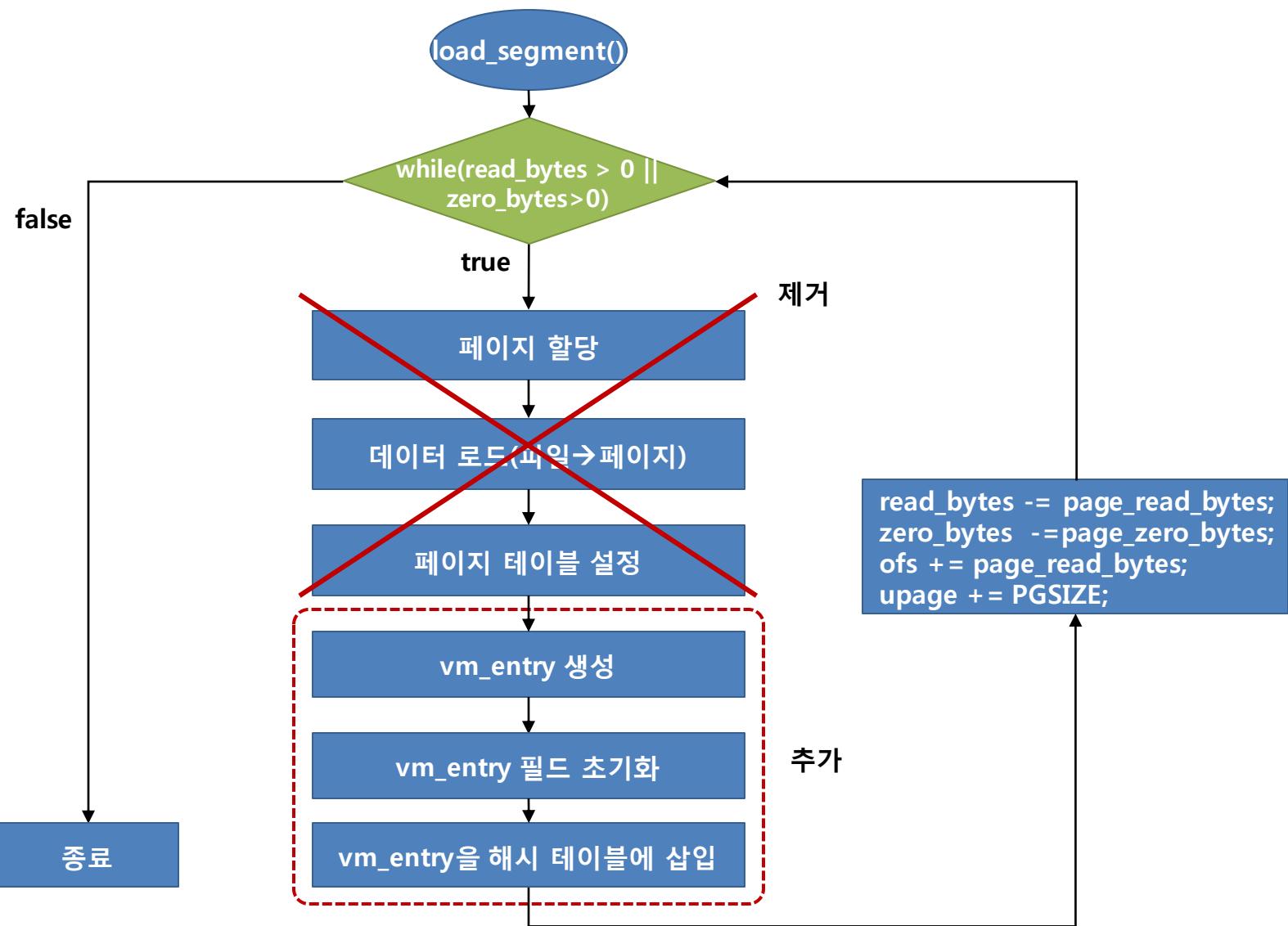
주소 공간 초기화 관련 함수 수정 (ELF 세그먼트)

▣ pintos/src/userprog/process.c

```
static bool load_segment(struct file *file, off_t ofs,  
uint8_t *upage, uint32_t read_bytes, uint32_t zero_bytes,  
bool writable)
```

- ◆ ELF포맷 파일의 세그먼트를 프로세스 가상주소공간에 탑재하는 함수이다.
- ◆ 이 함수에 프로세스 가상메모리 관련 자료구조를 초기화하는 기능을 추가한다.
 - 프로세스 가상주소공간에 메모리를 탑재하는 부분을 제거하고, vm_entry 구조체의 할당, 필드값 초기화, 해시 테이블 삽입을 추가한다.

load_segment() 함수 수정



load_segment() 함수 수정 (Cont.)

pintos/src/userprog/process.c

```
static bool load_segment (struct file *file, off_t ofs, uint8_t *upage,
                         uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
    ...
    while (read_bytes > 0 || zero_bytes > 0)
    {
        size_t page_read_byters = read_bytes < PGSIZE
                                ? read_bytes : PGSIZE;
        size_t page_zero_bytes = PGSIZE - page_read_byters;

        물리 페이지를 할당하고
        맵핑하는 부분 삭제
        -----> .....
        /* vm_entry 생성 (malloc 사용) */
        /* vm_entry 멤버들 설정, 가상페이지가 요구될 때 읽어야할 파일의 오프
           셋과 사이즈, 마지막에 패딩할 제로 바이트 등등 */
        /* insert_vme() 함수를 사용해서 생성한 vm_entry를 해시테이블에 추
           가 */

        read_bytes -= page_read_byters;
        zero_bytes -= page_zero_bytes;
        ofs += page_read_byters;
        upage += PGSIZE;
    }
}
```

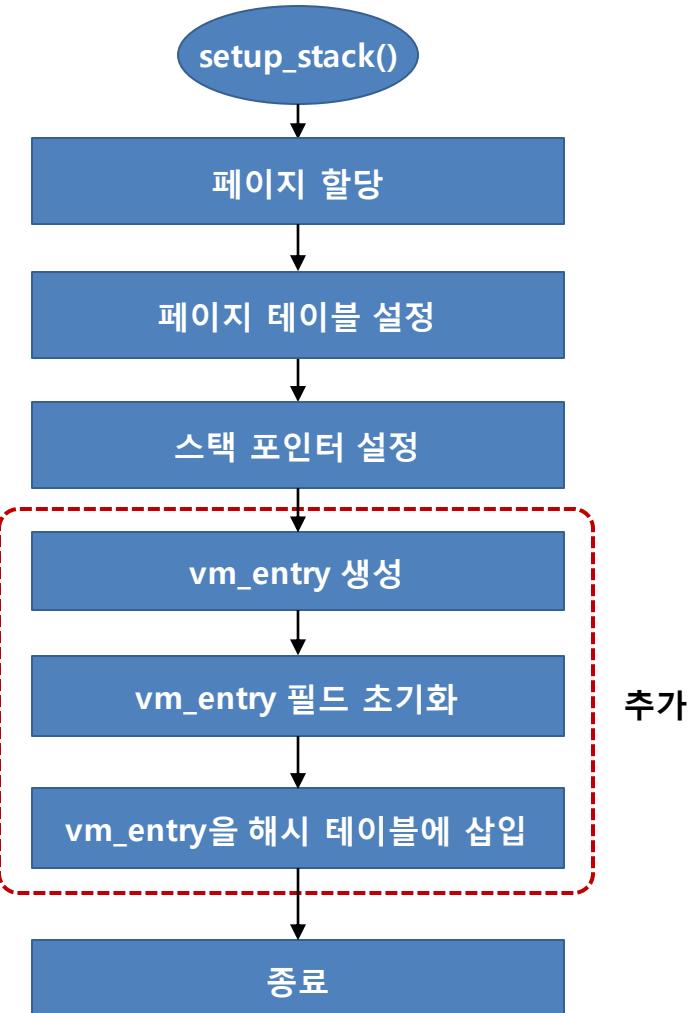
스택 초기화 함수 설정

▣ 기존

- ◆ 단일 페이지 할당
- ◆ 페이지 테이블 설정
- ◆ 스택 포인터 설정(esp)

▣ 추가할 부분

- ◆ 4KB 스택의 vm_entry 생성
- ◆ 생성한 vm_entry 필드값 초기화
- ◆ vm 해시 테이블에 삽입



추가

setup_stack() 함수 수정

pintos/src/vm/page.c

```
static bool setup_stack (void **esp)
{
    ...
    if (kpage != NULL)
    {
        ...
    }
    /* vm_entry 생성 */
    /* vm_entry 멤버들 설정 */
    /* insert_vme() 함수로 해시테이블에 추가 */
    ...
}
```



가상주소 유효성 검사

주소 유효성 검사란?

▣ 가상주소에 해당하는 vm_entry가 존재하는지 검사

- ◆ 시스템 콜(read/write) 사용 시 인자로 주어지는 문자열이나 Buffer의 주소에 해당하는 vm_entry가 존재하는지 검사

read(fd, **buffer**, size)

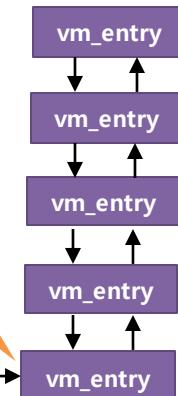
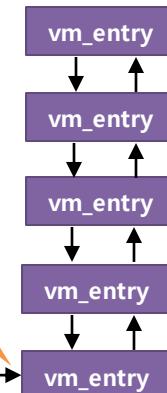


Buffer의 시작주소에 해당하는
vm_entry의 존재여부 검사

write(fd, **buffer**, size)



문자열의 시작주소에 해당하는
vm_entry의 존재여부 검사



check_address() 함수 수정

- ▣ 기존 check_address() 함수는 esp에 대한 유저 메모리 영역 체크
- ▣ vm_entry를 사용하여 유효성 검사 작업을 수행하도록 코드 수정
- ▣ vm_entry를 반환하도록 코드 수정

pintos/src/userprog/syscall.c

```
struct vm_entry * check_address (void* addr, void* esp /*Unused*/)  
{  
  
    if(addr < (void *) 0x08048000 || addr >= (void *) 0xc0000000)  
    {  
        exit(-1);  
    }  
    /*addr이 vm_entry에 존재하면 vm_entry를 반환하도록 코드 작성 */  
    /*find_vme() 사용*/  
}
```

check_valid_buffer() 함수 구현

```
void check_valid_buffer (void* buffer, unsigned size,  
                        void* esp, bool to_write)
```

- ◆ Buffer를 사용하는 read() system call의 경우 buffer의 주소가 유효한 가상주소인지 아닌지 검사할 필요성이 있음
- ◆ Buffer의 유효성을 검사하는 함수
- ◆ Check_valid_buffer 구현시 check_address() 함수를 사용
- ◆ to_write 변수는 buffer에 내용을 쓸 수 있는지 없는지 검사하는 변수

pintos/src/userprog/syscall.c

```
case SYS_READ :  
    ...  
    /* 기존 check_address /*인자삽입*/ 함수는 삭제*/  
    /*check_valid_buffer /*인자삽입*/ 함수 구현*/  
    ...  
    break;
```



check_valid_string() 함수 구현

```
void check_valid_string (const void* str, void* esp)
```

- ◆ System call에서 사용할 인자의 문자열의 주소값이 유효한 가상주소인지 아닌지 검사하는 함수
- ◆ check_valid_string 구현시 check_address() 함수를 사용

Characteristic	Read()	Write()
유효성	검사함	검사함
to_write	사용함	사용하지 않음
사용 함수	check_valid_buffer	check_valid_string

check_valid_buffer() 함수 추가

pintos/src/vm/page.c

```
void check_valid_buffer (void *buffer, unsigned size, void *esp,
                        bool to_write)
{
    /* 인자로 받은 buffer부터 buffer + size까지의 크기가 한 페이지의
     * 크기를 넘을 수도 있음 */

    /* check_address를 이용해서 주소의 유저영역 여부를 검사함과 동시에
     * vm_entry 구조체를 얻음 */

    /* 해당 주소에 대한 vm_entry 존재여부와 vm_entry의 writable 멤버가 true인지 검사 */

    /* 위 내용을 buffer부터 buffer + size까지의 주소에 포함되는
     * vm_entry들에 대해 적용 */
}
```



check_valid_string() 함수 추가

pintos/src/vm/page.c

```
void check_valid_string (const void *str, void *esp)
{
    /* str에 대한 vm_entry의 존재 여부를 확인*/
    /* check_address() 사용*/
}
```

syscall_handler() 함수 수정

- ▣ 함수 내 check_address() 함수의 인자값 변경
 - ◆ check_address() 함수의 두 번째 인자값은 f->esp
- ▣ 시스템 콜 호출 시 인자값의 유효성 검사를 하도록 코드 수정

pintos/src/userprog/syscall.c

```
static void syscall_handler(struct intr_frame *f UNUSED) {  
    ...  
    check_address(esp, esp); /* 인자값 변경 */  
    ...  
    switch(syscall_n) {  
        ...  
        case SYS_EXEC :  
            get_argument(esp , arg , 1);  
            /* buffer 사용 유무를 고려하여 유효성 검사를 하도록 코드 추가 */  
            f -> eax = exec((const char *)arg[0]);  
            break;  
        ...  
    }  
}
```

syscall_handler() 함수 수정

pintos/src/userprog/syscall.c

```
...
case SYS_OPEN :
get_argument(sp, arg , 1);
/* buffer 사용 유무를 고려하여 유효성 검사를 하도록 코드 추가 */
f -> eax = open((const char *)arg[0]);
break;
...
case SYS_READ :
get_argument(sp, arg , 3);
/* buffer 사용 유무를 고려하여 유효성 검사를 하도록 코드 추가 */
f -> eax = read(arg[0] , (void *)arg[1] , (unsigned)arg[2]);
break;
...
```

syscall_handler() 함수 수정

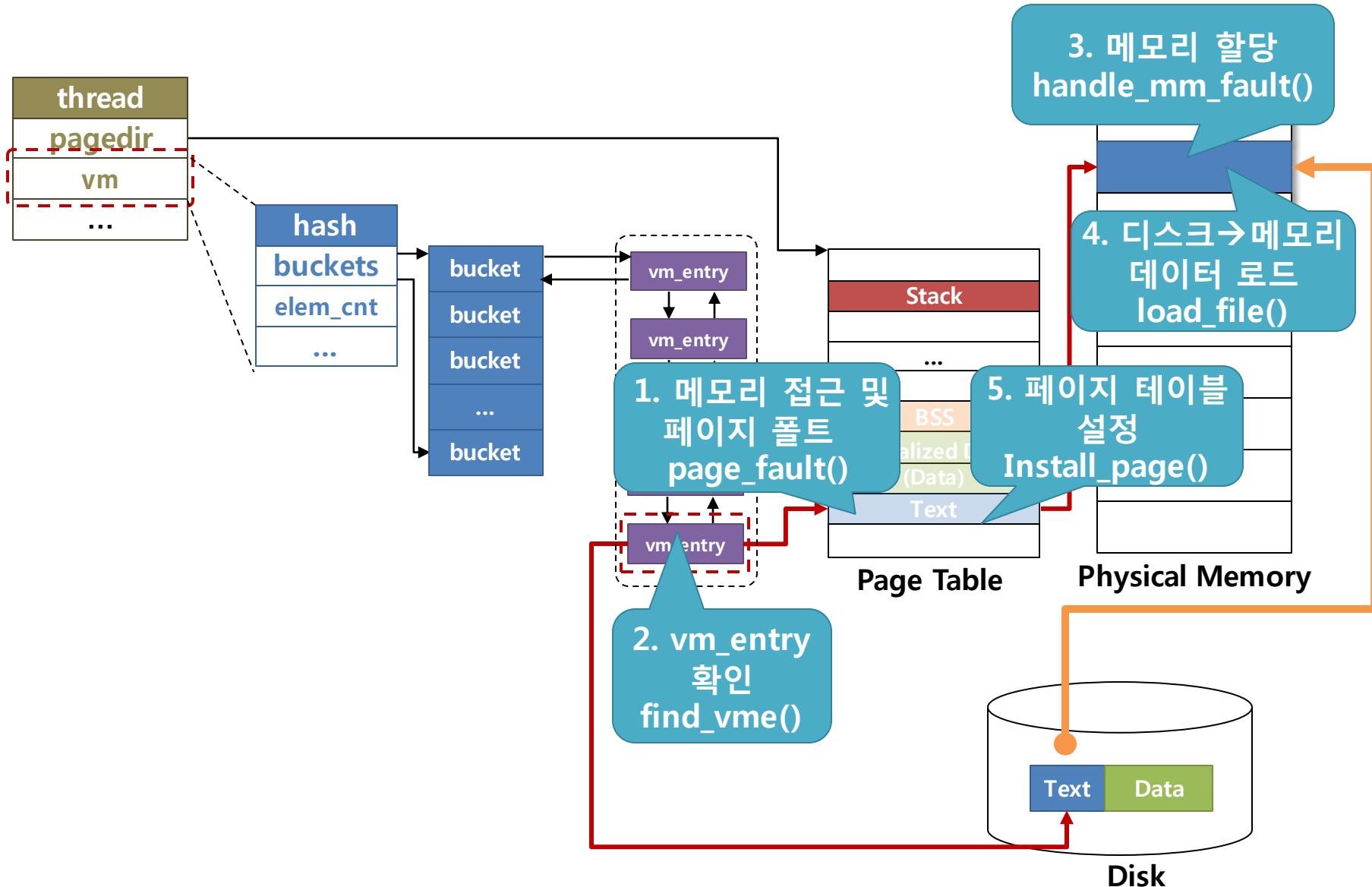
pintos/src/userprog/syscall.c

```
...
case SYS_WRITE :
    get_argument(sp, arg , 3);
    /* buffer 사용 유무를 고려하여 유효성 검사를 하도록 코드 추가 */
    f -> eax = write(arg[0] , (const void *)arg[1] ,
                      (unsigned)arg[2]);
    break;
...
default :
    thread_exit ();
}
}
```



요구 페이지 구현

요구 페이징 구현 순서



할일 1: 페이지 폴트 처리(Cont.)

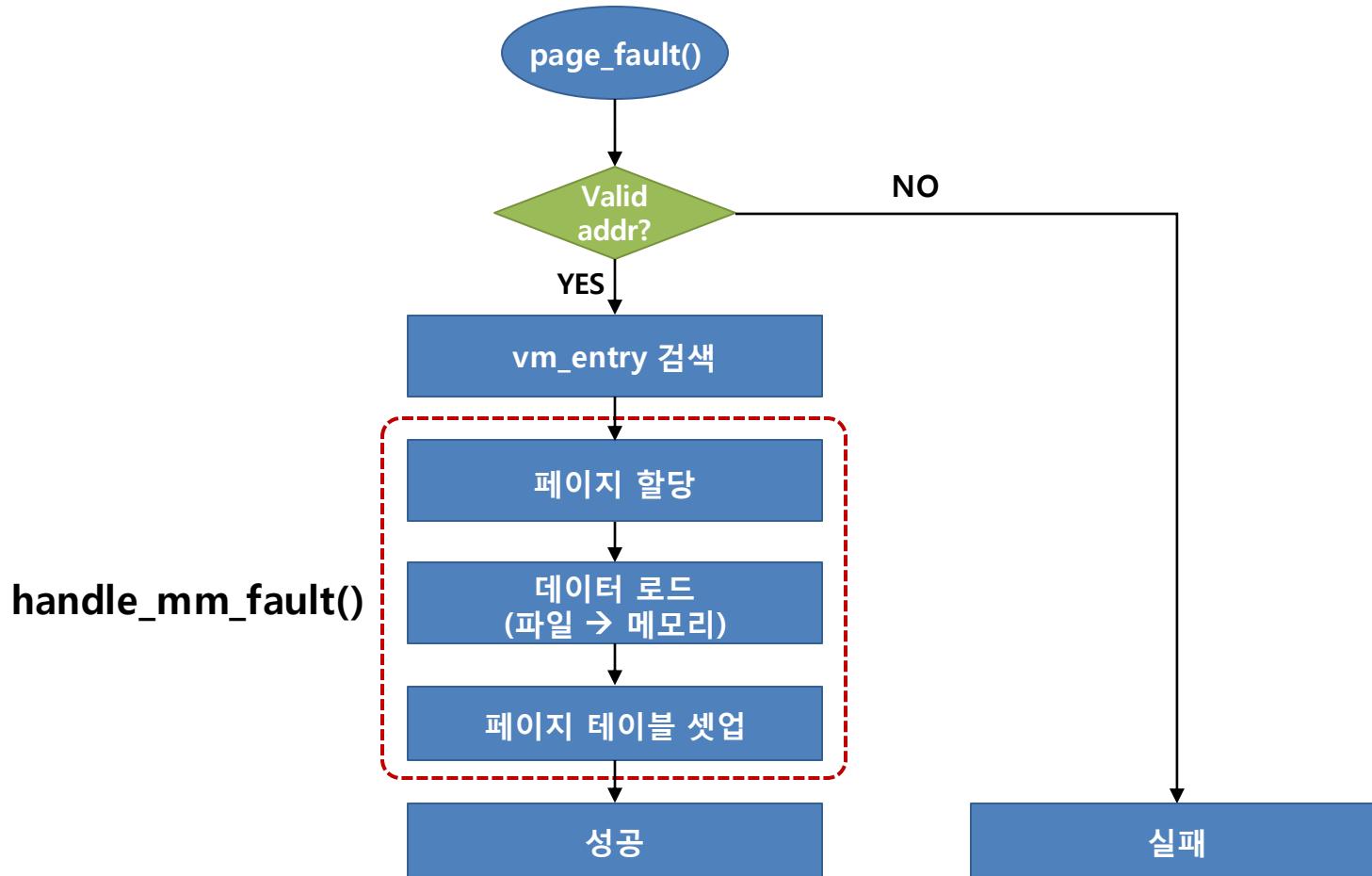
Pintos는 페이지 폴트 발생시 처리를 위해 page_fault()함수 존재

- ◆ pintos/src/userprog/exception.c
 - static void page_fault (struct intr_frame *f)
 - 현재의 pintos의 page_fault 처리는 permission, 주소 유효성 검사 후 오류발생시 무조건 "segmentation fault"를 발생시키고 kill(-1)을 하여 종료
 - kill(-1)처리 관련 코드 삭제
 - fault_addr의 유효성 검사
 - 페이지폴트 핸들러 함수 호출
 - handle_mm_fault (struct vm_entry *vme)



할일1: 페이지 폴트 처리(Cont.)

▣ 페이지폴트 처리 알고리즘



할일 1: 페이지 폴트 처리(Cont.)

- vm_entry 검색 후 페이지를 할당하도록 코드 수정

pintos/src/userprog/exception.c

```
static void page_fault (struct intr_frame *f) {
    ...
    /* Determine cause. */
    not_present = (f->error_code & PF_P) == 0;
    write = (f->error_code & PF_W) != 0;
    user = (f->error_code & PF_U) != 0;

    exit(-1);
    /* To implement virtual memory, delete the rest of the function
       body, and replace it with code that brings in the page to
       which fault_addr refers. */
    printf ("Page fault at %p: %s error %s page in %s context.\n",
           fault_addr,
           not_present ? "not present" : "rights violation",
           write ? "writing" : "reading",
           user ? "user" : "kernel");
    kill (f);
}
```

삭제 & 코드 구현

할일1: 페이지 폴트 처리(Cont.)

- vm_entry 검색 후 페이지를 할당하도록 코드 수정

pintos/src/userprog/exception.c

```
static void page_fault (struct intr_frame *f)
{
    void *fault_addr; /*page_fault가 발생한 가상주소*/
    ...
    /* Determine cause. */
    not_present = (f->error_code & PF_P) == 0;
    write = (f->error_code & PF_W) != 0;
    user = (f->error_code & PF_U) != 0;
    /* read only 페이지에 대한 접근이 아닐 경우 (not_present 참조)*/
    /* 페이지 폴트가 일어난 주소에 대한 vm_entry 구조체 탐색 */
    /* vm_entry를 인자로 넘겨주며 handle_mm_fault() 호출 */
    /* 제대로 파일이 물리 메모리에 로드 되고 맵핑 됬는지 검사 */
    ...
}
```

할일2: 페이지풀트 핸들러 구현

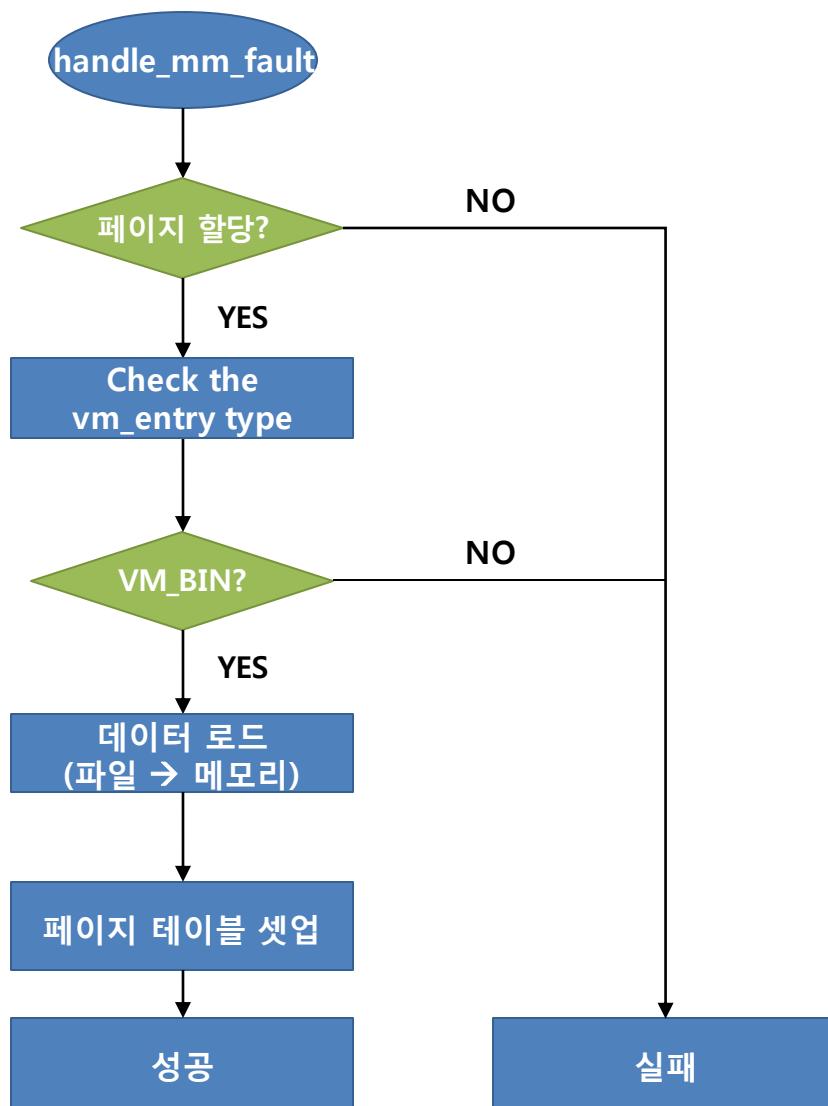
- ◆ pintos/src/userprog/process.c

- `bool handle_mm_fault(struct vm_entry *vme)`

- `handle_mm_fault`는 페이지 폴트 발생시 핸들링을 위해 호출 되는 함수
 - 페이지 폴트 발생시 물리페이지를 할당
 - Disk에 있는 file을 물리페이지로 load
 - `load_file (void* kaddr, struct vm_entry *vme)`를 사용
 - 물리메모리에 적재가 완료되면 가상주소와 물리주소를 페이지테이블로 맵핑
 - `static bool install_page(void *upage, void *kpage, bool writable)`를 사용



할일2: 페이지풀트 핸들러 구현(Cont.)



할일2: 페이지풀트 핸들러 구현(Cont.)

pintos/src/userprog/process.c

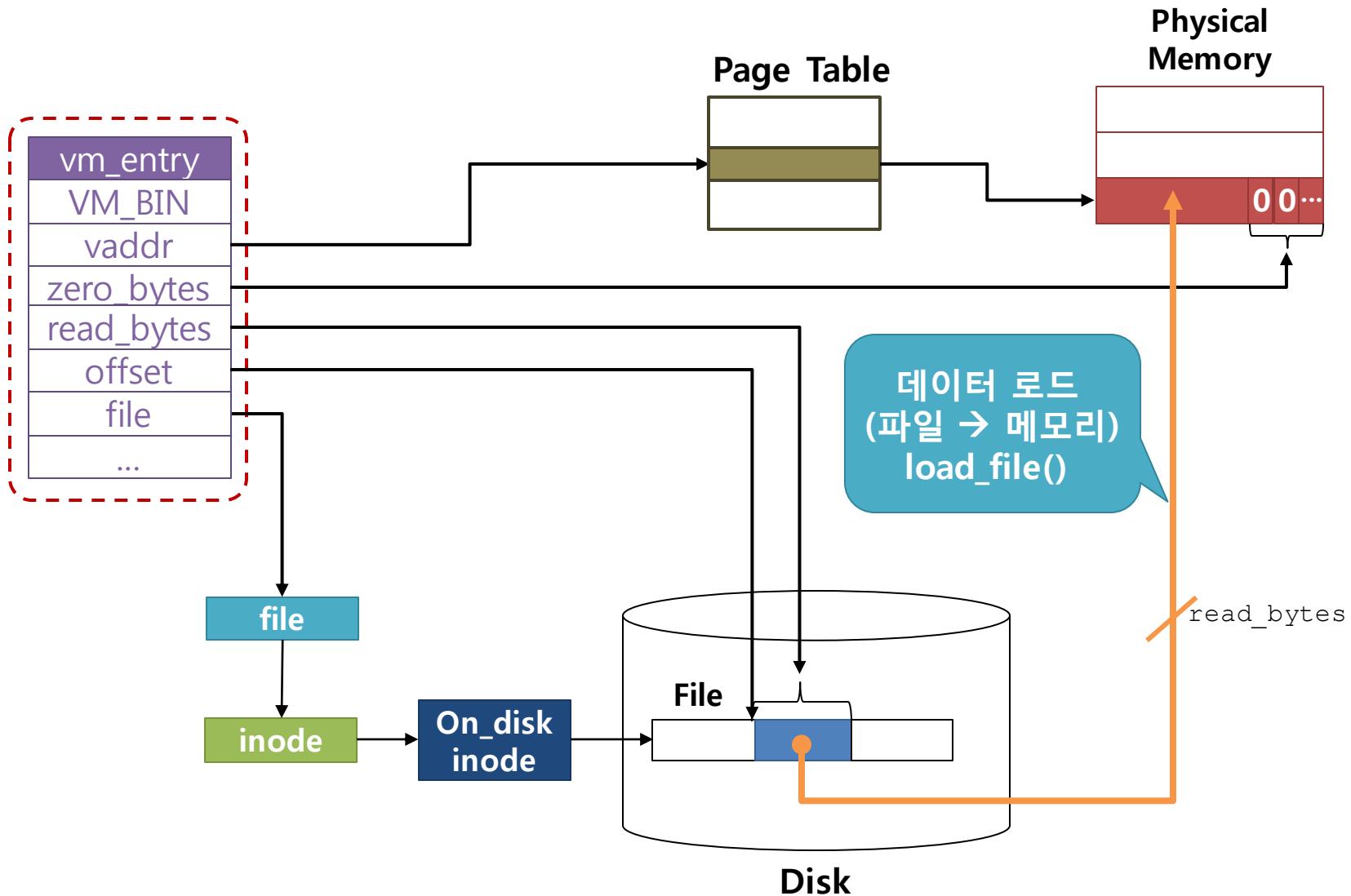
```
bool handle_mm_fault (struct vm_entry *vme)
{
    /* palloc_get_page()를 이용해서 물리메모리 할당 */
    /* switch문으로 vm_entry의 타입별 처리 (VM_BIN외의 나머지 타입은 mmf
    와 swapping에서 다름*/
    /* VM_BIN일 경우 load_file() 함수를 이용해서 물리메모리에 로드 */
    /* install_page를 이용해서 물리페이지와 가상페이지 맵핑 */
    /* 로드 성공 여부 반환 */
}
```



할일3: 물리메모리에 파일 쓰기

- ▣ 물리메모리 할당 완료 후 실제 디스크의 파일을 물리페이지로 load
- ▣ Pintos/src/vm/page.c
 - ▣ `bool load_file (void* kaddr, struct vm_entry *vme)`
 - ▣ Disk에 존재하는 page를 물리 메모리로 load하는 함수
 - ▣ vme의 <파일,offset>으로 한 페이지를 kaddr로 읽어 들이는 함수를 구현
 - ▣ `file_read_at()` 함수 또는 `file_read() + file_seek()` 함수 이용
 - ▣ 4KB를 전부 write하지 못했다면 나머지를 0으로 채움

할일3: 물리메모리에 파일 페이지 탑재



할일3: 물리메모리에 파일 쓰기(Cont.)

pintos/src/vm/page.c

```
bool load_file (void *kaddr, struct vm_entry *vme)
{
    /*Using file_read() + file_seek() */
    /* 오프셋을 vm_entry에 해당하는 오프셋으로 설정(file_seek()) */
    /* file_read로 물리페이지에 read_bytes만큼 데이터를 씀*/
    /* zero_bytes만큼 남는 부분을 '0'으로 패딩 */
    /* file_read 여부 반환 */
}
```

pintos/src/vm/page.c

```
bool load_file (void *kaddr, struct vm_entry *vme)
{
    /*Using file_read_at()*/
    /* file_read_at으로 물리페이지에 read_bytes만큼 데이터를 씀*/
    /* file_read_at 여부 반환 */
    /* zero_bytes만큼 남는 부분을 '0'으로 패딩 */
    /*정상적으로 file을 메모리에 loading 하면 true 리턴*/
}
```



요구 페이징 구현 관련 함수 추가 및 수정

- ▣ pintos/src/userprog/exception.c

```
static void page_fault (struct intr_frame *f)  
/*현재는 page fault가 발생하면 kill(-1)을 하여 종료함*/  
/*kill(-1) 처리 관련 코드 삭제*/  
/*vm_entry 검색 후 페이지를 할당하도록 코드 수정, handle_mm_fault() 이용 */
```

- ▣ pintos/src/vm/page.c

```
bool load_file (void* kaddr, struct vm_entry *vme)  
/*Disk에 존재하는 page를 물리 메모리로 load하는 함수 */  
/* vme의 <파일,offset>로부터 한 페이지를 kaddr로 읽어 들이는 함수를 구현 */  
/* file_read_at() 함수 또는 file_read() + file_seek() 함수 이용 */
```

- ▣ pintos/src/userprog/process.c

```
bool handle_mm_fault(struct vm_entry *vme)  
/*handle_mm_fault는 페이지 폴트 발생시 핸들링을 위해 호출 되는 함수 */  
/* 페이지 폴트 발생시 물리페이지를 할당하는 함수를 구현 */
```

수정 사항

▣ 수정 파일

- ◆ pintos/src/userprog/process.c
- ◆ pintos/src/userprog/exception.c
- ◆ pintos/src/userprog/syscall.*
- ◆ pintos/src/threads/thread.h
- ◆ pintos/Makefile.build
- ◆ pintos/tests/Make.tests

▣ 추가 파일

- ◆ pintos/src/vm/page.h
- ◆ pintos/src/vm/page.c

수정 파일

▣ Makefile.build 수정

- ◆ 추가한 page파일을 사용하기 위해 코드 추가

pintos/Makefile.build

```
...
userprog_SRC += userprog/tss.c          # TSS management.

# No virtual memory code yet.
#vm_SRC = vm/file.c                      # Some file.

vm_SRC = vm/page.c

# Filesystem code.
filesys_SRC = filesys/filesys.c          # Filesystem core.
filesys_SRC += filesys/free-map.c         # Free sector
bitmap.
filesys_SRC += filesys/file.c            # Files.
filesys_SRC += filesys/directory.c       # Directories.
filesys_SRC += filesys/inode.c            # File headers.
filesys_SRC += filesys/fsutil.c           # Utilities.

...
```



수정 파일 (Cont.)

- ▣ Makefile.tests 수정
- ▣ 변경하지 않으면 make check시 fail 발생
 - ◆ 환경에 따라 테스트 수행시간을 지나칠 수 있음.

pintos/tests/Makefile.tests

```
...
ifdef PROGS
include ../../Makefile.userprog
endif

TIMEOUT = 60 /*Pintos의 테스트 수행시간을 60초에서 120초로 변경*/
clean::
    rm -f $(OUTPUTS) $(ERRORS) $(RESULTS)

grade:: results
    $(SRCDIR)/tests/make-grade $(SRCDIR) $< $(GRADING_FILE)
| tee $@

...
...
```



추가 함수

```
void vm_init(struct hash* vm)
```

/* 해시테이블 초기화 */

```
void vm_destroy(struct hash *vm)
```

/* 해시테이블 제거 */



추가 함수(Cont.)

```
struct vm_entry* find_vme(void *vaddr)
/* 현재 프로세스의 주소공간에서 vaddr에 해당하는 vm_entry를 검색 */
```

```
bool insert_vme(struct hash *vm, struct vm_entry *vme)
/* 해시테이블에 vm_entry 삽입 */
```

```
bool delete_vme(struct hash *vm, struct vm_entry *vme)
/* 해시 테이블에서 vm_entry삭제 */
```

추가 함수(Cont.)

```
static unsigned vm_hash_func(const struct hash_elem *e, void  
*aux UNUSED)
```

/* 해시테이블에 vm_entry 삽입 시 어느 위치에 넣을 지 계산 */

```
static bool vm_less_func(const struct hash_elem *a, const  
struct hash_elem *b, void *aux UNUSED)
```

/* 입력된 두 hash_elem의 주소 값 비교 */

```
static void vm_destroy_func(struct hash_elem *e, void *aux  
UNUSED)
```

/* vm_entry의 메모리 제거 */

추가 함수(Cont.)

```
bool handle_mm_fault(struct vm_entry *vme)
    /* 페이지 폴트 발생시 물리페이지를 할당 */

bool load_file(void* kaddr, struct vm_entry *vme)
    /* vme의 <파일, offset>로부터 한 페이지를 kaddr로 읽어들이는 함수 */

void check_valid_buffer(void* buffer, unsigned size,
                       void* esp, bool to_write)
    /* buffer내 vm_entry가 유효한지 확인하는 함수 */

void check_valid_string(const void* str, void* esp)
    /* 문자열의 주소값이 유효한지 확인하는 함수 */
```

수정 함수

```
static bool load_segment(struct file *file, off_t ofs,
                        uint8_t *upage, uint32_t read_bytes,
                        uint32_t zero_bytes, bool writable)

/* ELF포맷 파일의 세그먼트를 프로세스 가상주소공간에 적절히 탑재하는 함수
실행 파일을 로드 시 호출 */
```

```
static bool setup_stack(void **esp)

/* 가상 메모리의 스택부분을 초기화 하는 함수 */
```



수정 함수(Cont.)

```
static void page_fault(struct intr_frame *f)
/* 페이지 폴트 핸들러 */

static void syscall_handler(struct intr_frame *f UNUSED)
/* 시스템 콜 넘버에 해당하는 시스템 콜을 호출 하는 함수 */

void check_address(void *addr, void* esp)
/* 주소 값이 유저 영역에서 사용하는 주소 값인지 확인 하는 함수 */

void process_exit (void)
/* 프로세스 종료 시 호출되어 프로세스의 자원을 해제 */
```

추가 및 수정 자료구조

```
struct vm_entry
```

```
/* 논리주소와 물리주소를 분리하여 “반드시” 필요한 페이지들만 탑재  
시키도록 하는 자료구조 */
```

```
struct thread
```

```
/* 스레드의 정보를 가지고 있는 자료구조 */
```

가상메모리 과제 검증

▣ 가상메모리 과제를 완료 후 코드 동작 확인

- ◆ 경로 : pintos/src/vm

```
$ make check
```

▣ 실행 결과 109개의 테스트 중 28개에서 fail 발생

- ◆ pt-grow-stack
- ◆ page-linear
- ◆ page-merge-stk
- ◆ mmap-unmap
- ◆ mmap-exit
- ◆ mmap-inherit
- ◆ mmap-over-data
- ◆ pt-grow-pusha
- ◆ page-parallel
- ◆ page-merge-mm
- ◆ mmap-overlap
- ◆ mmap-shuffle
- ◆ mmap-misalign
- ◆ mmap-over-stk
- ◆ pt-big-stk-obj
- ◆ page-merge-seq
- ◆ mmap-read
- ◆ mmap-twice
- ◆ mmap-bad-fd
- ◆ mmap-null
- ◆ mmap-remove
- ◆ pt-grow-stk-sc
- ◆ page-merge-par
- ◆ mmap-close
- ◆ mmap-write
- ◆ mmap-clean
- ◆ mmap-over-code
- ◆ mmap-zero

```
gaya@gaya: ~/바탕화면/PintOS/project3/3_1/answer/vm
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
28 of 109 tests failed.
make[1]: *** [check] 오류 1
make[1]: Leaving directory `/home/gaya/바탕화면/PintOS/project3/3_1/answer/vm/build'
make: *** [check] 오류 2
gaya@gaya:~/바탕화면/PintOS/project3/3_1/answer/vm$
```

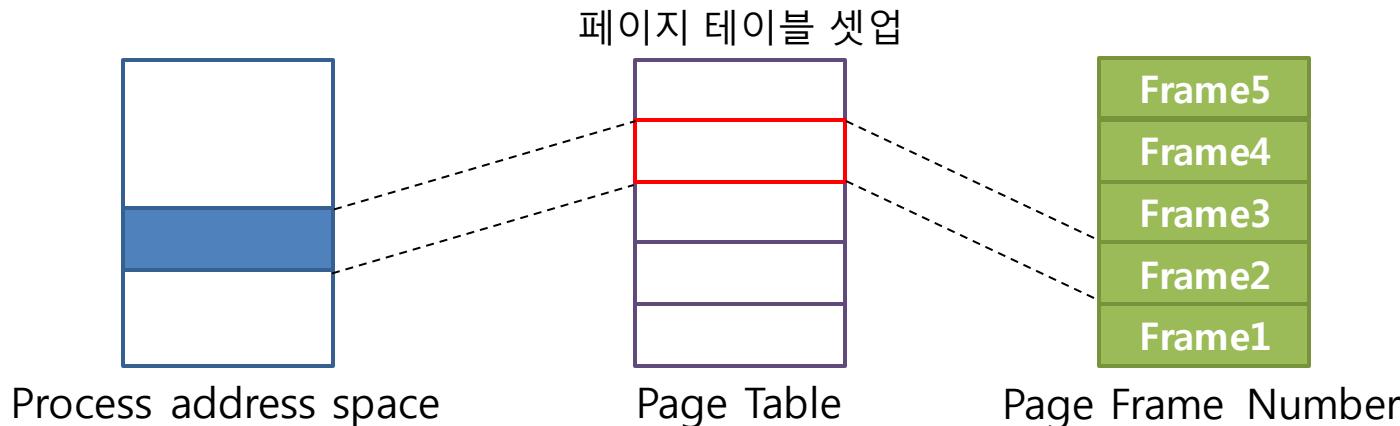
Appendix

페이지 주소 맵핑함수

```
#include "usrprog/process.c"

static bool install_page(void *upage, void *kpage,
                        bool writable)
```

- ◆ 페이지 테이블에 물리 주소와 가상주소를 맵핑 시켜주는 함수
- ◆ 물리 페이지 kpage와 가상 페이지 upage를 맵핑
- ◆ writable: 쓰기 가능(1), 읽기전용(0)



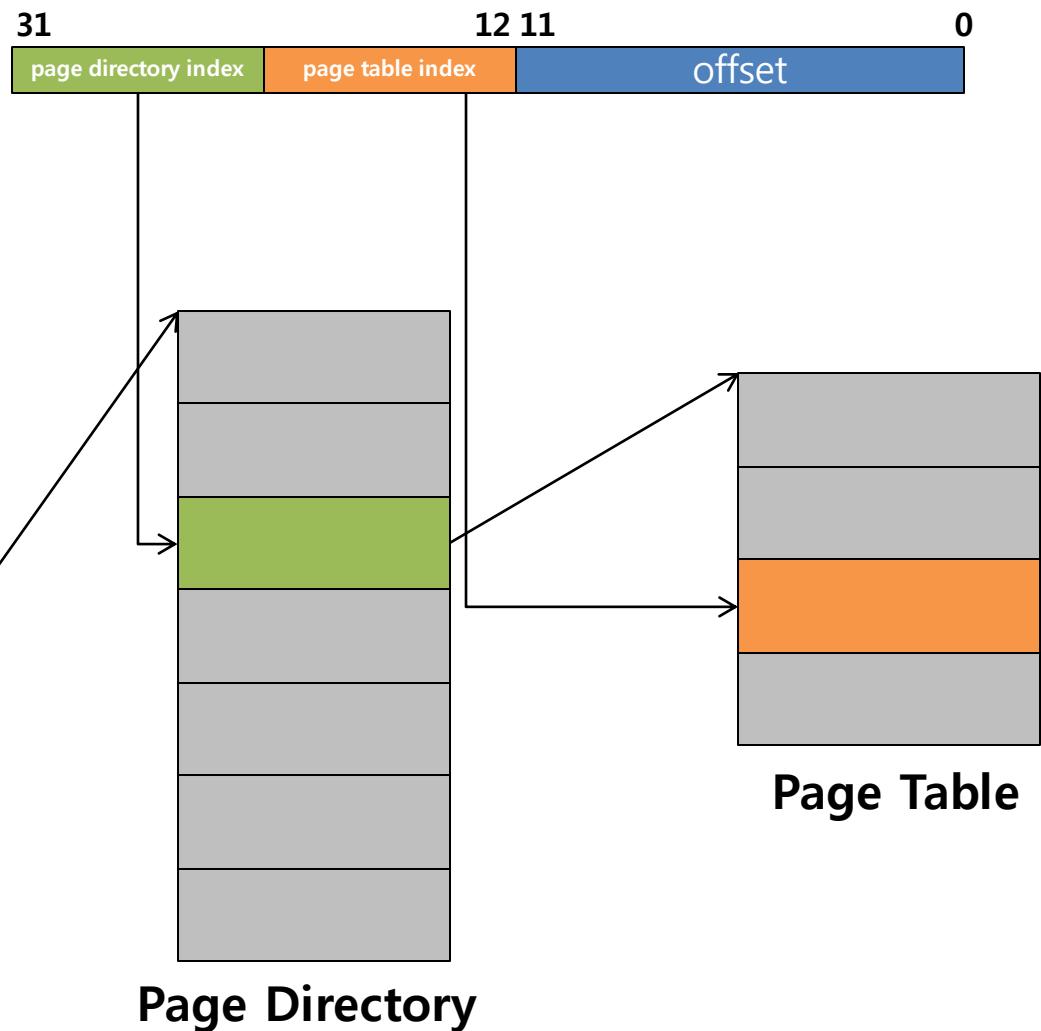
Pintos의 페이지 테이블 구조

- ▣ 페이지 테이블과 페이지 디렉토리를 이용해서 물리 주소와 가상주소의 맵핑 관리
 - ◆ 페이지 디렉토리
 - 페이지 테이블의 주소를 갖고 있는 테이블
 - 모든 가상페이지에 대한 엔트리 사용은 비효율적이므로 한 단계 상위 개념인 페이지 디렉토리를 사용
 - ◆ 페이지 테이블
 - 가상 주소에 맵핑된 물리 주소를 갖고 있는 엔트리들의 집합
- ▣ 가상 메모리 주소에 포함된 index로 entry 접근

Pintos의 페이지 테이블 구조 (Cont.)

pintos/src/threads/thread.h

```
struct thread
{
    tid_t tid;
    ...
#ifndef USERPROG
    uint32_t *pagedir;
#endif
    ...
}
```



물리 페이지 할당 및 해제 인터페이스

```
#include <threads/palloc.h>
```

```
void *palloc_get_page(enum palloc_flags flags)
```

- ◆ 4KB의 페이지를 할당
- ◆ 페이지의 물리 주소를 리턴
- ◆ flags
 - PAL_USER: 유저 메모리풀에서 페이지 할당
 - PAL_KERNEL: 커널 메모리 풀에서 페이지 할당
 - PAL_ZERO: 페이지를 '0'으로 초기화

```
void palloc_free_page(void *page)
```

- ◆ 페이지의 물리주소를 인자로 사용
- ◆ 페이지를 다시 여유 메모리 풀에 넣음



12. Memory Mapped File

Memory Mapped File 개요

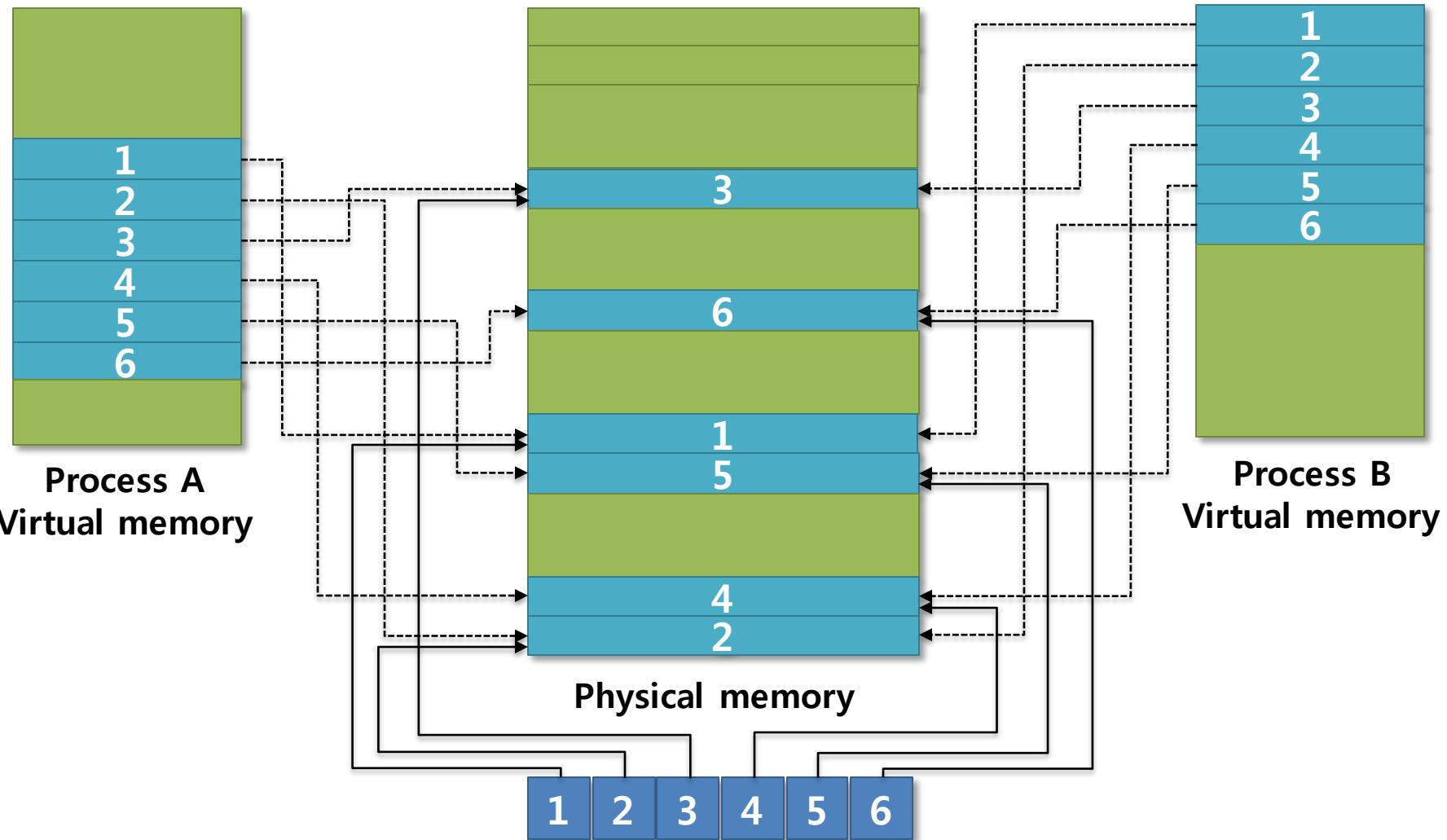
- ▣ 현재의 핀토스는 mmap()과 munmap() 함수가 구현되어 있지 않다. 본 과제에서는 mmap()과 munmap() 함수를 구현한다.
 - ◆ 요구페이지에 의해 파일 데이터를 메모리로 로드하는 mmap() 함수 구현
 - ◆ 파일 매핑을 제거하는 munmap() 함수 구현

Memory Mapped File

- Treat file I/O as routine memory access by *mapping* a disk block to a page in memory
- A file is initially read using demand paging. → Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies file access by treating file I/O through memory rather than `read()` `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

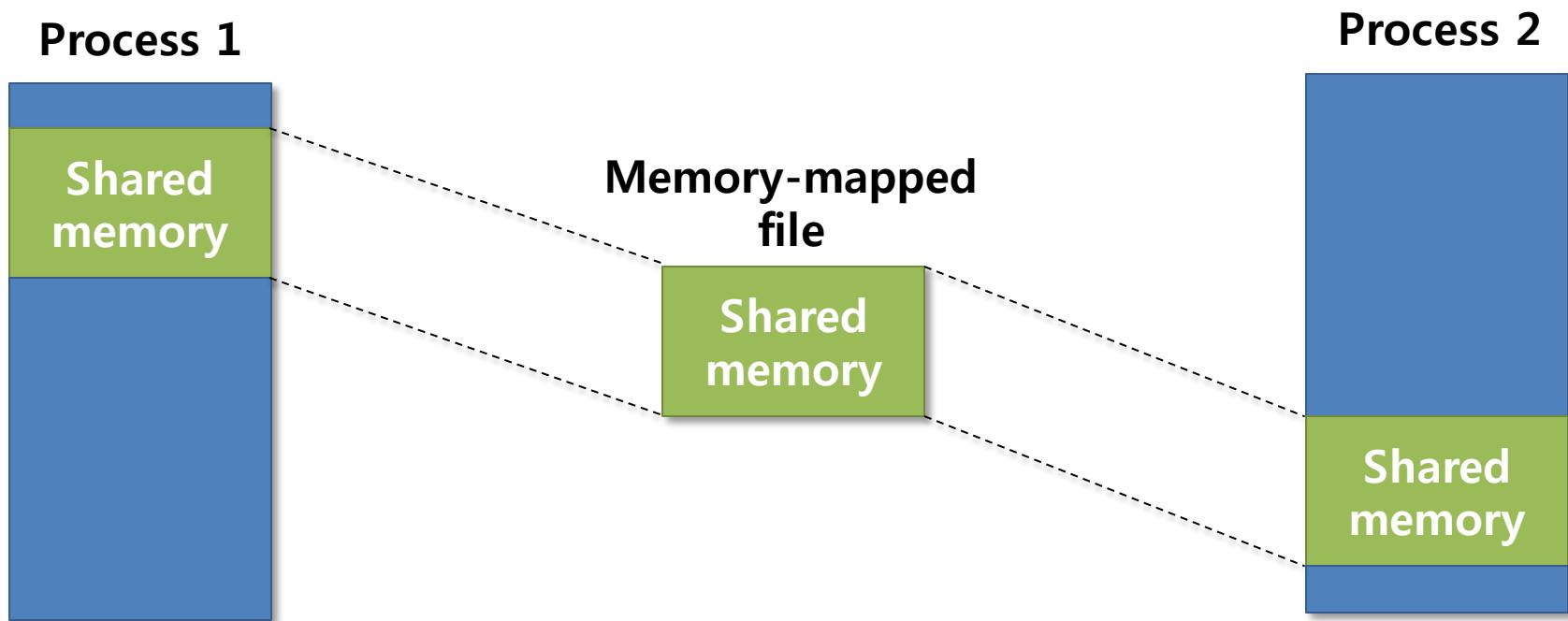
Memory-Mapped Files

Basic Mechanism



Memory-Mapped Files

- Shared memory in windows using memory-mapped I/O



Memory-Mapped I/O

- ❑ To allow more convenient access to I/O devices, many computer architectures provide memory-mapped I/O.
- ❑ In this case, ranges of memory addresses are set aside and are mapped to the device registers.
- ❑ Read and writes to these memory addresses cause the data to be transferred to and from the device registers.
- ❑ This method is appropriate for devices that have fast response times, such as video controllers.

Memory Mapped File 개요

▣ mmap() 구현

```
mapid_t mmap (int fd, void* addr)
```

- ◆ fd: 프로세스의 가상 주소공간에 맵핑할 파일
- ◆ addr: 맵핑을 시작할 주소(page 단위 정렬)
- ◆ 성공 시 mapping id를 리턴, 실패 시 에러코드(-1) 리턴
- ◆ 요구페이지에 의해 파일 데이터를 메모리로 로드

▣ munmap() 구현

```
void munmap(mapid_t mapping)
```

- ◆ mmap()의 리턴 값을 통해 파일 맵핑을 제거

▣ UNIX semantic에 따라 mmap() 된 파일은 munmap()의 호출, 혹은 프로세스가 종료 시까지 접근의 유효성 보장

예) munmap() 이전에 close() 호출되더라도 파일 맵핑은 여전히 유효함



Memory Mapped File 개요 (Cont.)

▣ 수정 파일

- ◆ pintos/src/userprog/process.c
- ◆ pintos/Makefile.build

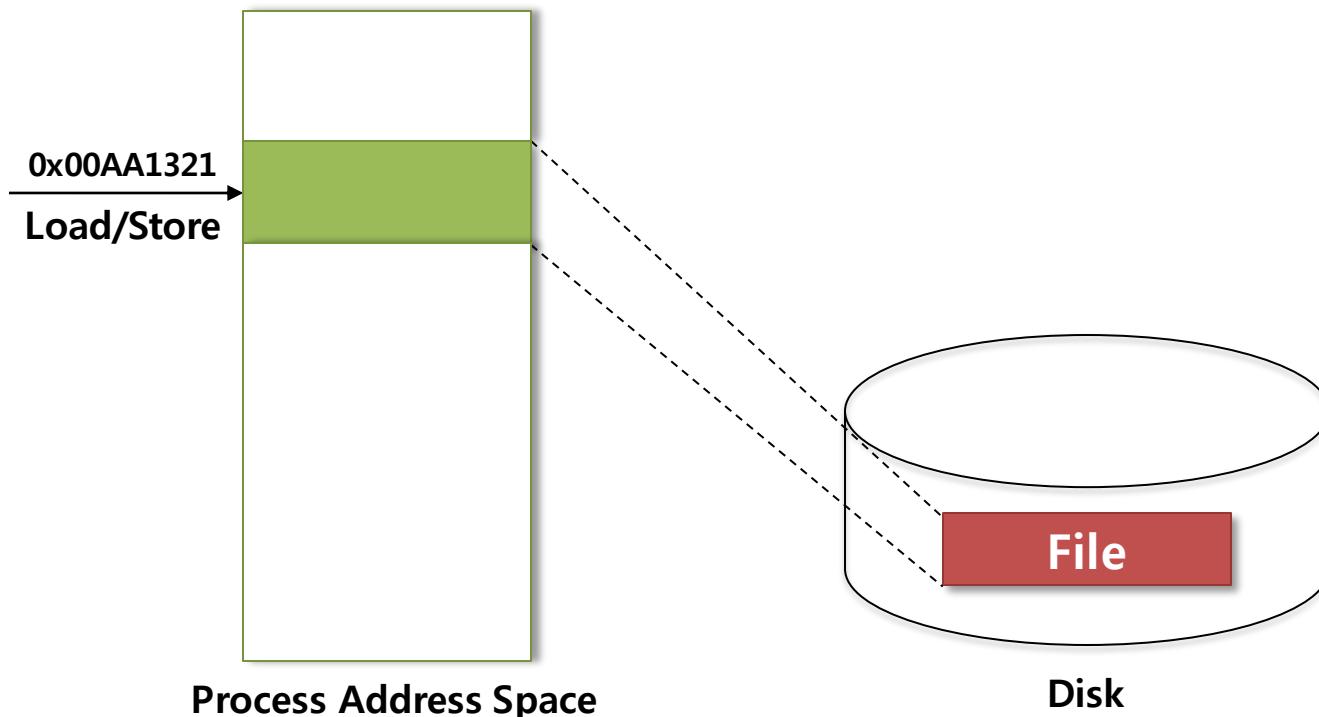
▣ 추가 파일

- ◆ pintos/src/userprog/syscall.*
- ◆ printos/src/userprog/process.c

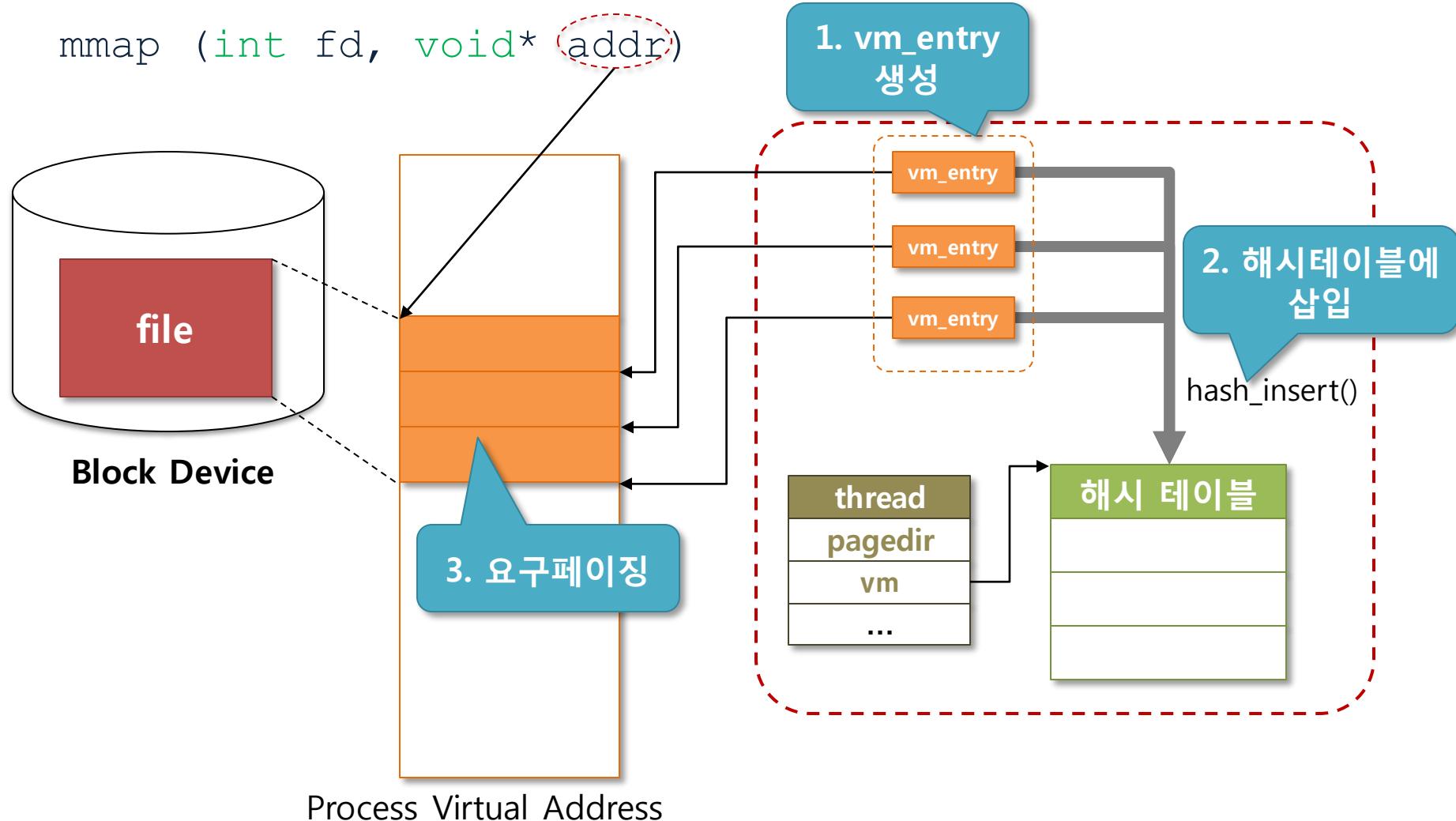


Memory Mapped File

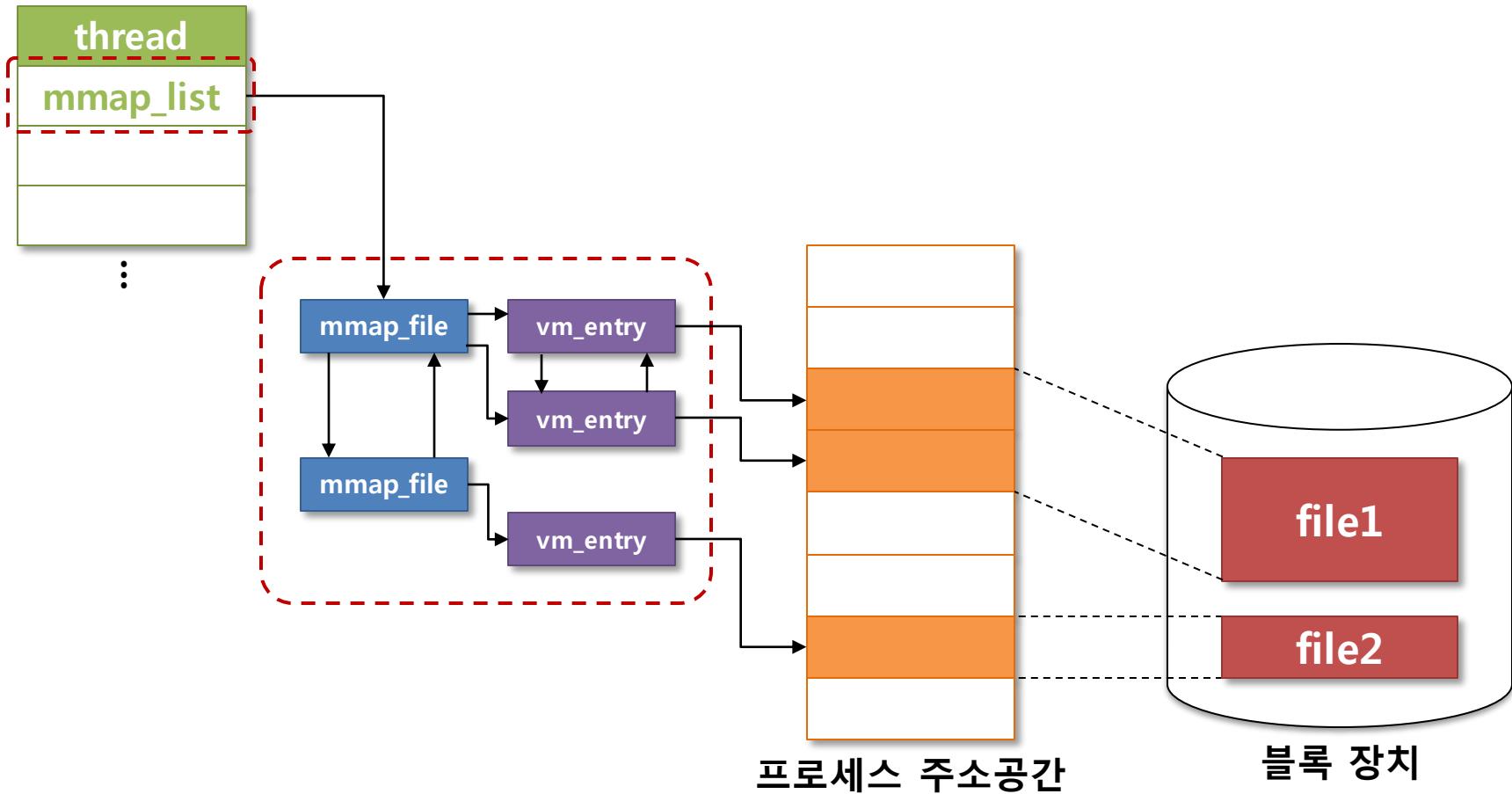
- ▣ 프로세스 주소공간에 파일을 맵핑
- ▣ `read()` / `write()` 시스템콜 대신 메모리 접근(load/store)을 통해 파일을 접근



파일 매팅 메커니즘



매핑된 파일의 관리



mmap_file 자료구조 추가

▣ 매팅된 파일의 정보를 저장

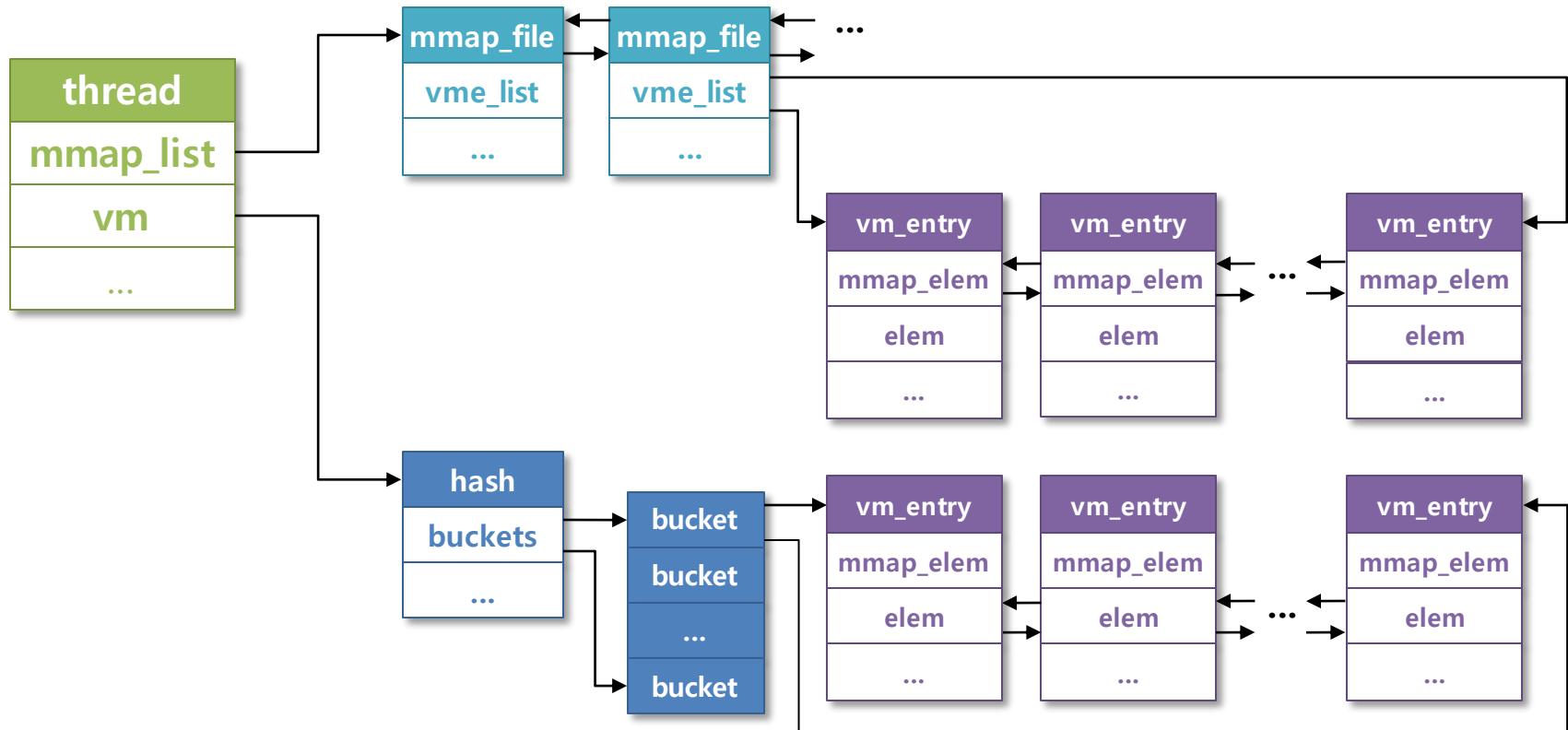
- ◆ mapid: mmap() 성공 시 리턴된 mapping id
- ◆ file: 매팅하는 파일의 파일 오브젝트
- ◆ elem: mmap_file들의 리스트 연결을 위한 구조체
 - 리스트 헤드는 struct thread의 mmap_list
- ◆ vme_list: mmap_file에 해당하는 모든 vm_entry들의 리스트

pintos/src/vm/page.h

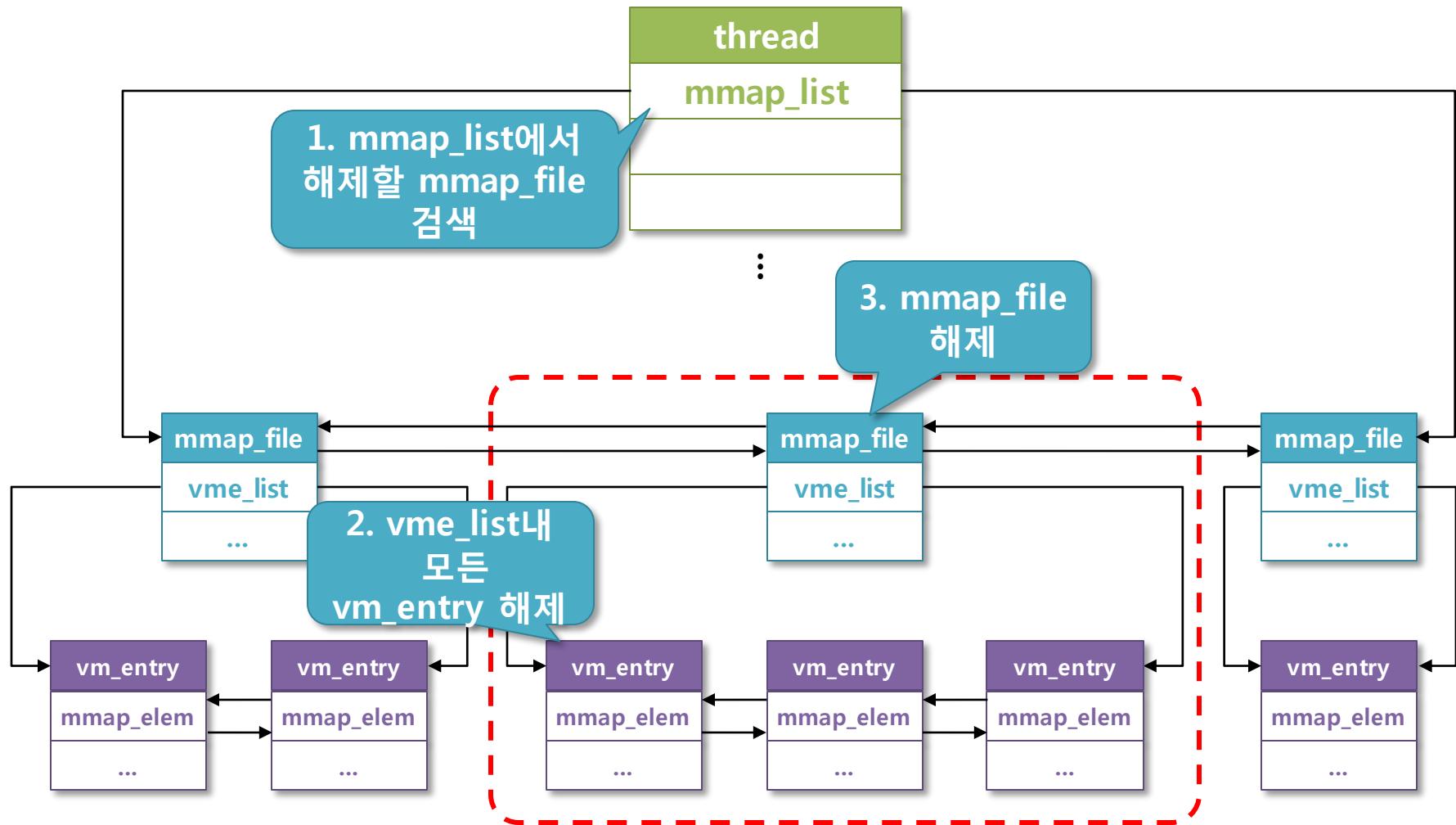
```
struct mmap_file {  
    int mapid;  
    struct file* file;  
    struct list_elem elem;  
    struct list vme_list;  
};
```

vm_entry의 리스트 구조

- thread구조체의 해시테이블 vm에서는 vm_entry의 elem필드 사용
- mmap_file의 리스트 헤드 vme_list는 vm_entry의 mmap_elem필드 사용



munmap() 동작



Memory Mapped File 과제 관련 코드

- ▣ 제공되는 구조체 및 함수 (Project 3 코드에서 기본 제공)

- ◆ pintos/src/vm/page.h

```
struct mmap_file
```

- ◆ pintos/src/userprog/process.c

```
struct file *process_get_file(int fd)
```

구현 함수

- ▣ pintos/src/userprog/syscall.c

```
int mmap(int fd, void *addr)
```

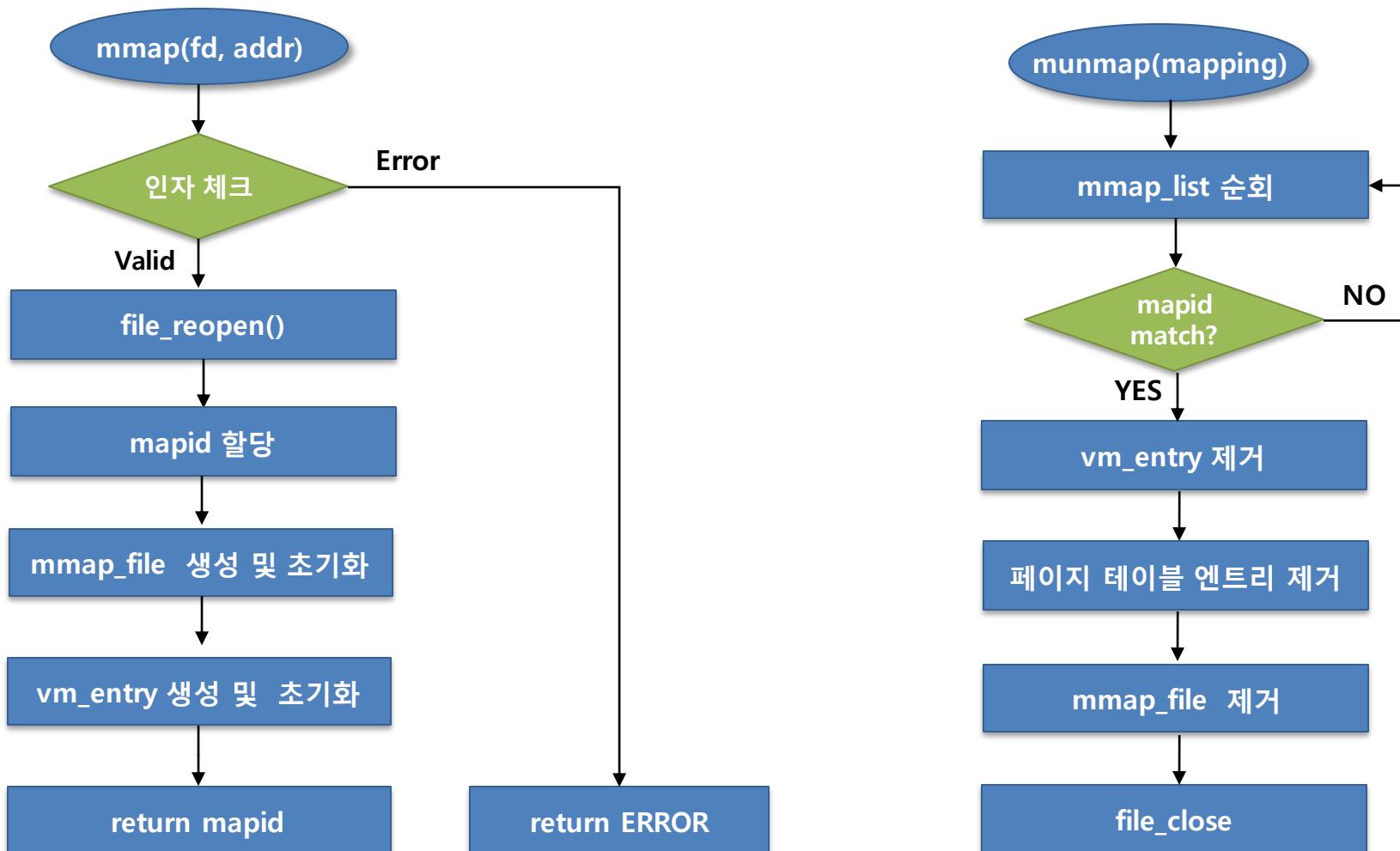
```
void munmap(mapid_t mapid)
```

- ▣ pintos/src/userprog/process.c

```
bool handle_mm_fault(struct vm_entry *vme)
```

- ◆ 가상 메모리 과제에서 구현한 함수
- ◆ 매픽된 파일에 대해 요구 페이지를 지원하도록 수정

mmap() & munmap()



map() 함수 구현

```
int mmap (int fd, void* addr)
```

- ◆ fd: 프로세스의 가상 주소공간에 매핑할 파일
- ◆ addr: 매핑을 시작할 주소(page 단위 정렬)
- ◆ 성공 시 mapping id를 리턴, 실패 시 에러코드(-1) 리턴
- ◆ 요구페이지에 의해 파일 데이터를 메모리로 로드



mmap() 구현 시 사용되는 함수

▣ pintos/src/userprog/process.c

```
struct file* process_get_file(int fd)
```

- ◆ 프로세스의 파일 디스크립터 목록을 검색하여 파일 객체의 주소 리턴

▣ pintos/src/filsys/file.c

```
struct file* file_reopen(struct file*)
```

- ◆ 파일 객체를 복제하여 복제된 객체의 주소 리턴
- ◆ close() 되더라도 mmap()의 유효성을 유지하기 위하여 파일 객체 복제



munmap() 구현

```
void munmap(int mapping)
```

- ◆ mmap_list내에서 mapping에 해당하는 mapid를 갖는 모든 vm_entry을 해제
- ◆ 인자로 넘겨진 mapping 값이 CLOSE_ALL인, 경우 모든 파일 매팅을 제거
- ◆ 페이지 테이블에서 엔트리 제거
- ◆ 매팅 제거 시 do_munmap() 함수 호출

munmap() 구현 시 사용되는 함수

- ▣ pintos/src/vm/frame.c

```
bool pagedir_is_dirty (uint32_t *pd, const void *vpage)
```

- ◆ pd가 가리키는 페이지테이블에서 vpage에 해당하는 주소가 dirty면 1 리턴

- ▣ pintos/src/filesys/file.c

```
off_t file_write_at (struct file *file, const void *buffer,  
                      off_t size, off_t file_ofs)
```

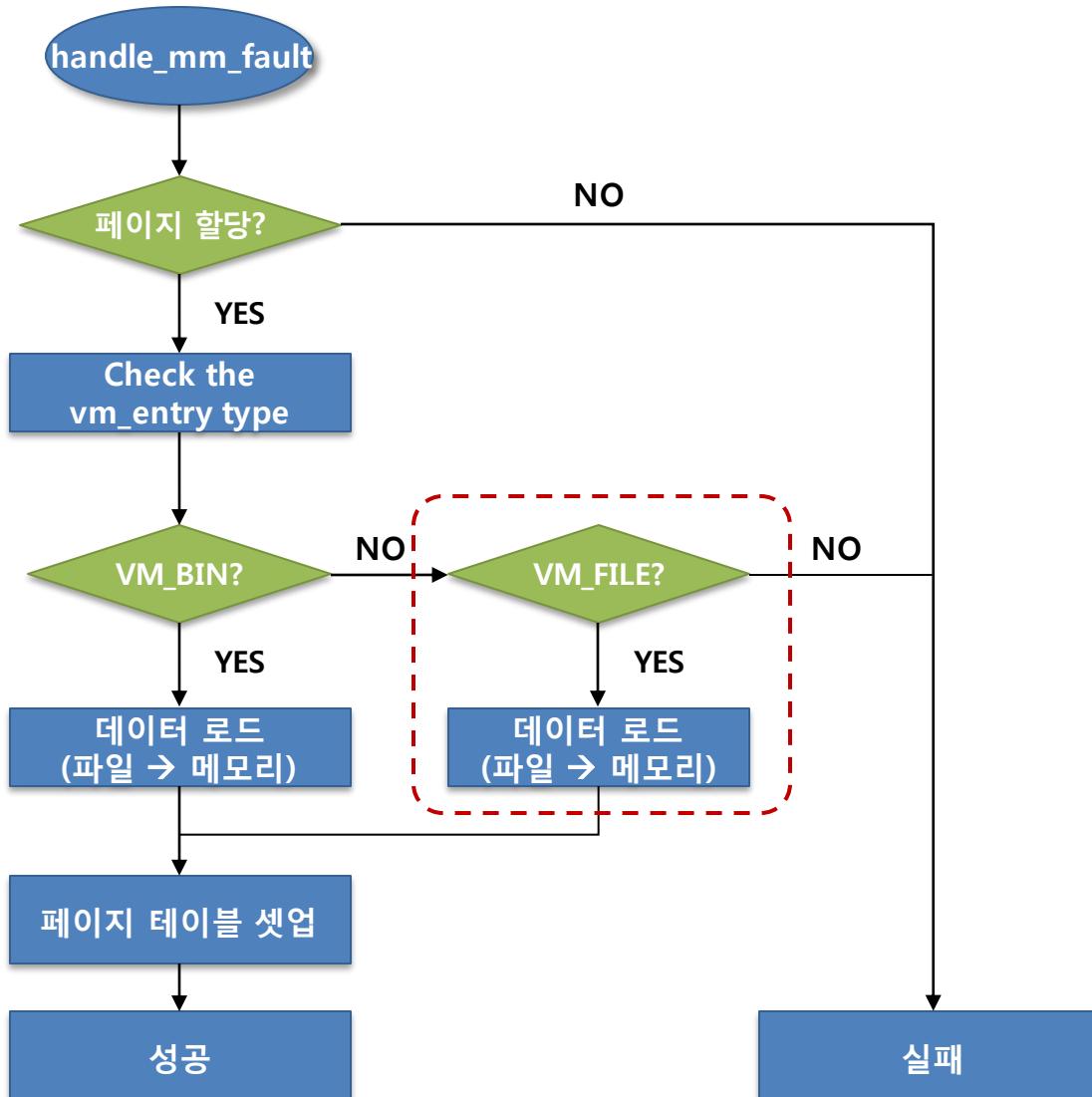
- ◆ buffer의 내용을 file의 file_ofs부터 size만큼 기록

do_munmap() 함수 구현

```
void do_munmap(struct mmap_file* mmap_file)
```

- ◆ mmap_file의 vme_list에 연결된 모든 vm_entry들을 제거
- ◆ 페이지 테이블 엔트리 제거
- ◆ vm_entry가리키는 가상 주소에 대한 물리 페이지가 존재하고, dirty하면 디스크에 메모리 내용을 기록

파일 매핑 요구 페이지 지원을 위한 페이지 폴트 핸들러 수정



handle_mm_fault() 함수 수정

- vm_entry 타입이 VM_FILE시 데이터를 로드할 수 있도록 코드 수정

pintos/src/userprog/process.c

```
bool handle_mm_fault(struct vm_entry *vme) {
    bool success = false;
    viod *kaddr;
    ...
    switch(vme->type) {
        case VM_BIN:
            success = load_file(kaddr, vme);
            break;

        case VM_FILE:
            /* VM_FILE시 데이터를 로드할 수 있도록 수정 */
            break;
    }
    ...
}
```

process_exit() 함수 수정

- ▣ 프로세스 종료 시 mmap_list에서 mapping에 해당하는 mapid를 갖는 모든 vm_entry을 해제를 제거하도록 코드 수정

pintos/src/userprog/process.c

```
void process_exit (void) {
    struct thread *cur = thread_current();
    uint32_t *pd;
    ...
    /* 메모리 leak 방지를 위한 메모리 해제 */
    palloc_free_page(cur -> fd);
    /* vm_entry를 해제하는 함수 삽입 */
    vm_entry(&cur->vm);

    pd = cur->pagedir;
    ...
}
```

Memory Mapped File 과제 검증

- Memory Mapped File 과제를 완료 후 코드 동작 확인.

- 경로 : pintos/src/vm

```
$ make check
```

- 실행 결과 109개의 테스트 중 10개에서 fail 발생.

- pt-grow-stack
- page-linear
- page-merge-stk
- pt-grow-pusha
- page-parallel
- page-merge-mm
- pt-big-stk-obj
- page-merge-seq
- pt-grow-stk-sc
- page-merge-par



```
root@gaya: /home/gaya/VM/Project3/p_3/pintos/src/vm
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
-----
10 of 109 tests failed.
```

추가 함수

```
int mmap(int fd, void *addr)
/* 요구페이지에 의해 파일 데이터를 메모리로 로드 */

void munmap(mapid_t mapid)
/* mmap_list내에서 mapping에 해당하는 mapid를 갖는 모든
   vm_entry을 해제 */

void do_munmap(struct mmap_file* mmap_file)
/* mmap_file의 vme_list에 연결된 모든 vm_entry들을 제거 */
```



수정 함수 및 추가 자료구조

```
static void page_fault(struct intr_frame *f)
    /* 페이지 폴트 핸들러 */

void process_exit (void)
    /* 프로세스 종료 시 호출되어 프로세스의 자원을 해제 */

struct mmap_file
    /* 매핑된 파일의 정보를 저장 */
```

13. Swapping

스와핑 및 페이지 교체 정책 개요

▣ 스와핑 지원

- ◆ Victim으로 선정된 페이지가 프로세스의 데이터 영역 혹은 스택에 포함될 때 이를 스왑 영역에 저장
- ◆ swap-out 된 페이지는 요구 페이징에 의해 다시 메모리에 로드
- ◆ LRU기반 알고리즘을 이용한 페이지 교체 매커니즘을 동작하도록 수정

스와핑 및 페이지 교체 정책 개요 (Cont.)

▣ 수정 파일

- ◆ pintos/src/userprog/process.c
- ◆ pintos/src/userprog/syscall.c
- ◆ pintos/src/thread/init.c
- ◆ pintos/Makefile.build

▣ 추가 파일

- ◆ pintos/src/vm/page.h
- ◆ pintos/src/vm/frame.*
- ◆ pintos/src/vm/swap.*

page 자료구조 추가

- 유저에게 할당된 물리 페이지 하나를 표현하는 자료구조
 - kaddr: 페이지의 물리주소
 - vme: 물리 페이지가 매핑된 가상 주소의 vm_entry 포인터
 - thread: 해당 물리 페이지를 사용중인 스레드의 포인터
 - lru: list 연결을 위한 필드

pintos/src/vm/page.h

```
struct page {  
    void *kaddr;  
    struct vm_entry *vme;  
    struct thread *thread;  
    struct list_elem lru;  
};
```

Makefile.build

▣ Makefile.build 수정

- ◆ 추가한 frame, swap파일을 사용하기 위해 코드 추가

pintos/Makefile.build

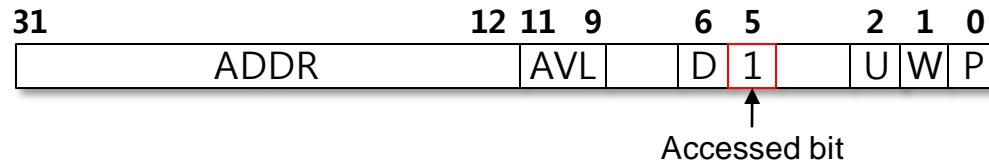
```
...
userprog_SRC += userprog/tss.c          # TSS management.

# No virtual memory code yet.
vm_SRC = vm/page.c
#vm_SRC = vm/file.c                      # Some file.

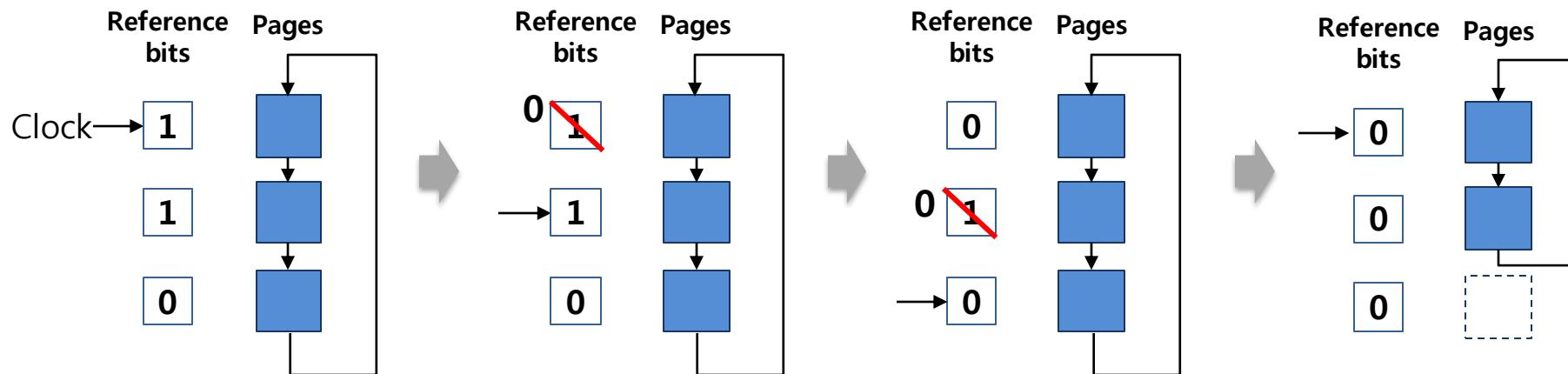
# Filesystem code.
filesys_SRC = filesys/filesys.c         # Filesystem core.
filesys_SRC += filesys/free-map.c        # Free sector
bitmap.
filesys_SRC += filesys/file.c           # Files.
filesys_SRC += filesys/directory.c      # Directories.
...
```

페이지 교체 정책 알고리즘 : Clock 알고리즘

- 페이지 테이블의 accessed bit는 페이지가 참조될 때마다 하드웨어에 의해 1로 설정됨

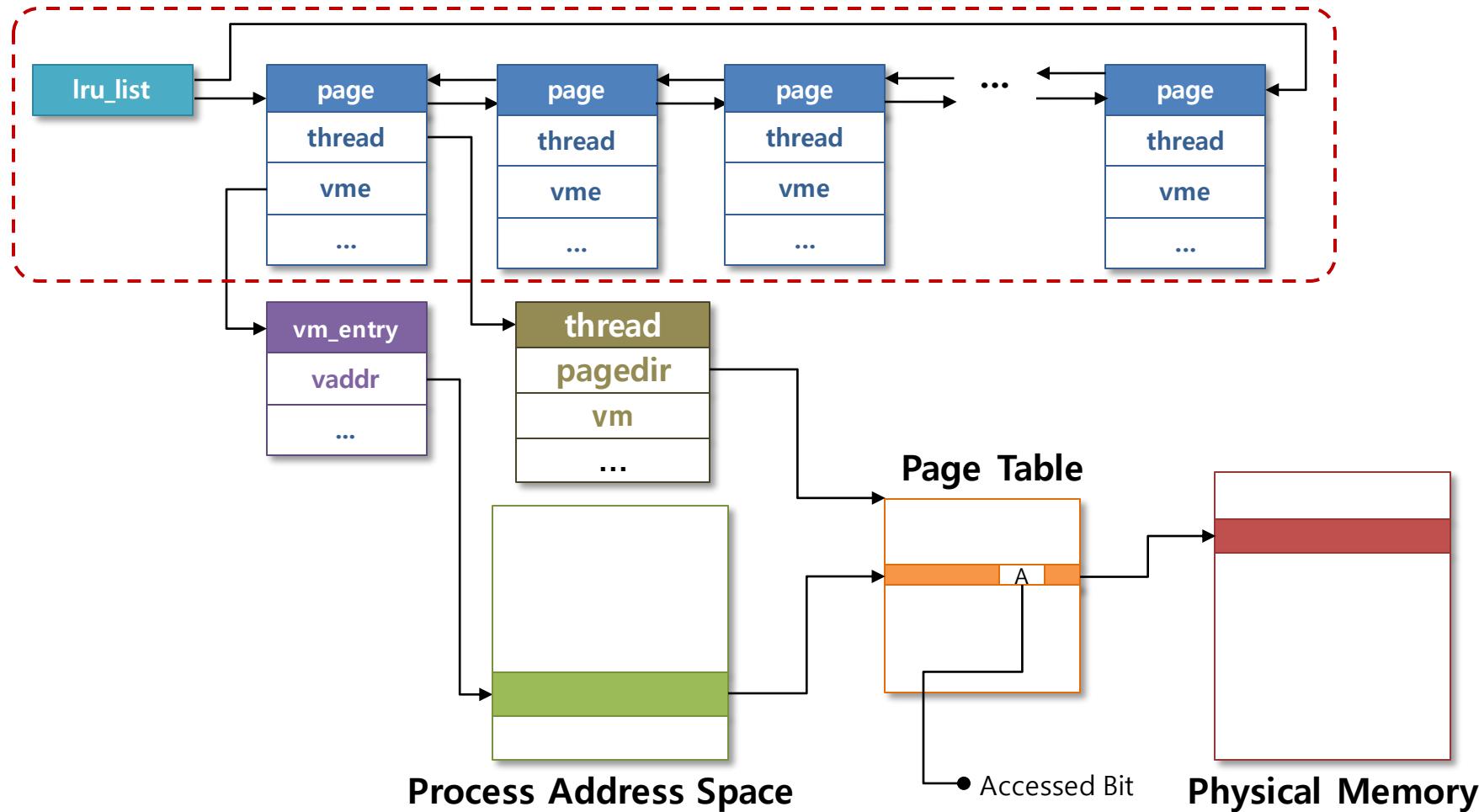


- 하드웨어는 accessed bit를 다시 0으로 만들지 않음
- 현재 포인터가 가리키고 있는 페이지의 참조비트 검사
 - '0'이면 해당 페이지를 victim으로 선정
 - '1'이면 참조 비트를 0으로 재설정



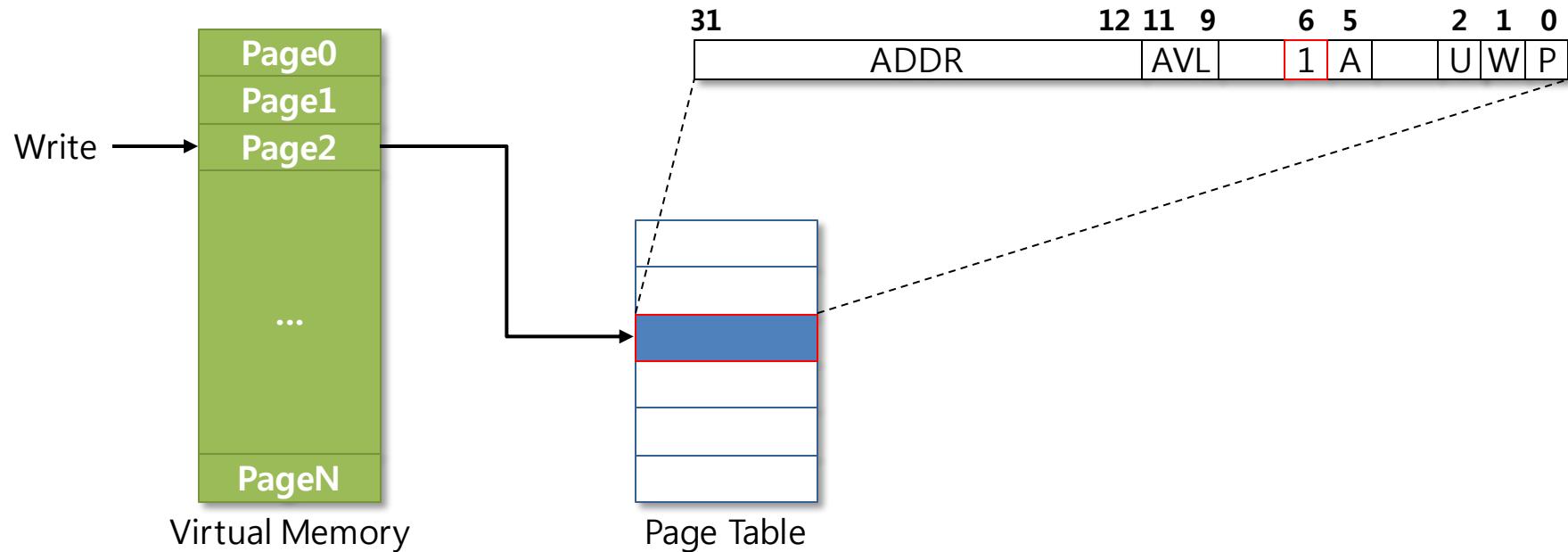
페이지 풀 관리

- page 구조체의 리스트로 프로세스에게 할당된 물리 페이지를 관리
- lru_list : 전역 변수



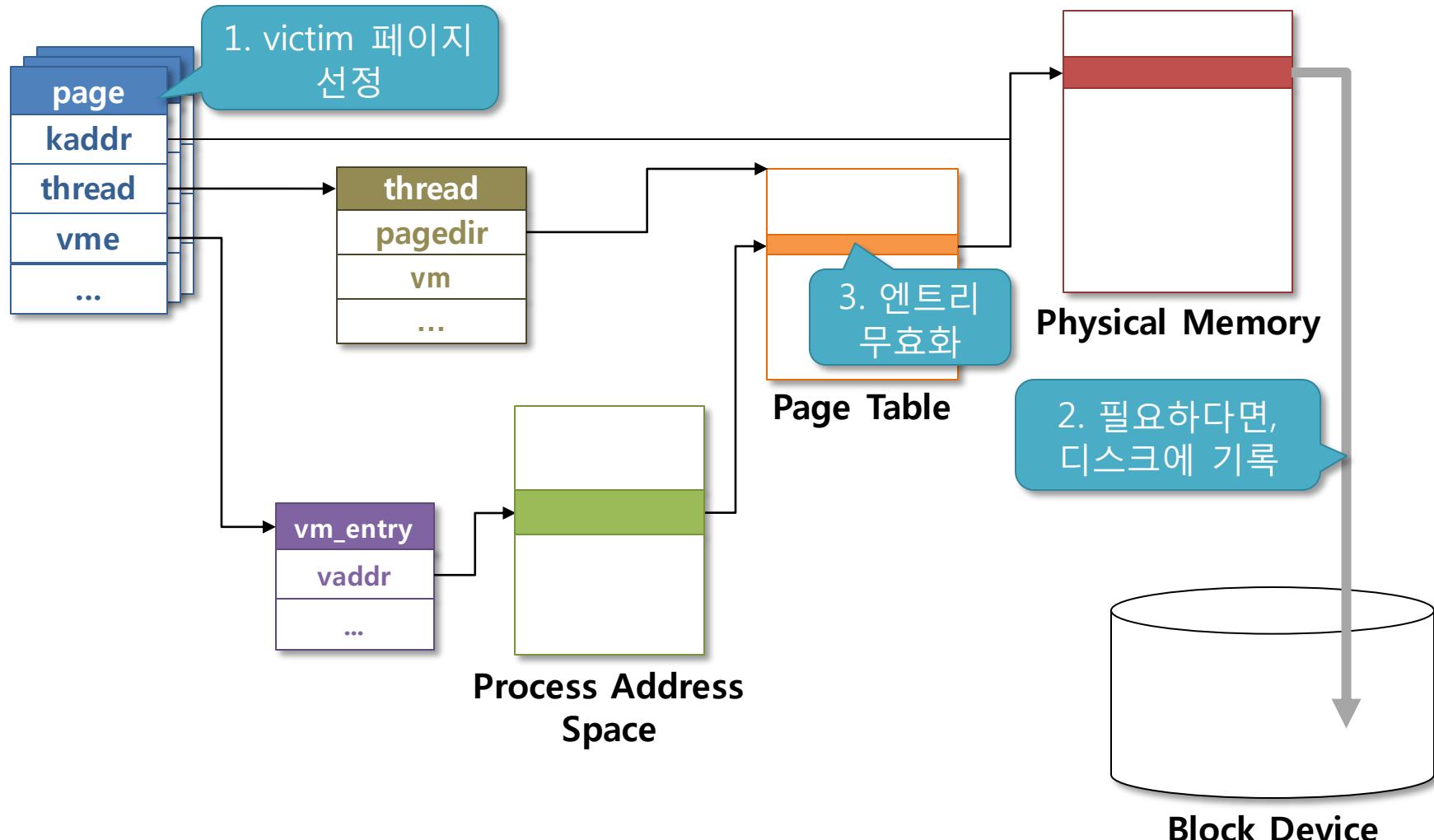
Dirty Bit

- 페이지 테이블의 dirty bit은 해당 메모리 영역에 쓰기시 하드웨어에 의하여 "1"로 설정



- Dirty bit이 "1"인 페이지가 victim으로 선정되었을 때, 변경내용을 항상 디스크에 저장해야 함

페이지 스왑 아웃



LRU 리스트 초기화, 삽입 및 삭제 함수 구현

```
void lru_list_init (void)
```

- ◆ lru_list 초기화
- ◆ lru_list_lock 초기화
- ◆ lru_clock의 값을 NULL로 설정

```
void add_page_to_lru_list(struct page* page)  
/* LRU list의 끝에 유저 페이지 삽입 */
```

```
void del_page_from_lru_list(struct page* page)  
/* LRU list에 유저 페이지 제거 */
```

LRU 리스트 삽입/삭제를 위한 페이지 할당/해제 함수 추가

- 유저페이지의 할당/삭제 시 해당 페이지를 LRU리스트를 삽입/제거

```
struct page* alloc_page(enum palloc_flags flags)
```

- palloc_get_page()를 통해 페이지 할당
- page 구조체를 할당, 초기화
- add_page_to_lru_list()를 통해 LRU 리스트에 page 구조체 삽입
- page 구조체의 주소를 리턴

```
void free_page(void *kaddr)
```

- 물리 주소 kaddr에 해당하는 page 구조체를 lru 리스트에서 검색
- 매치하는 항목을 찾으면 __free_page() 호출

```
void __free_page(struct page* page)
```

- LRU리스트의 제거
- Page 구조체에 할당받은 메모리 공간을 해제

alloc_page() 함수 구현

```
struct page* alloc_page(enum palloc_flags flags)
```

- ◆ palloc_get_page()를 통해 페이지 할당
- ◆ page 구조체를 할당, 초기화
- ◆ add_page_to_lru_list()를 통해 LRU 리스트에 page 구조체 삽입
- ◆ page 구조체의 주소를 리턴

Page 구조체 해제 함수 구현

```
void free_page(void *kaddr)
```

- ◆ 물리 주소 kaddr에 해당하는 page 구조체를 lru 리스트에서 검색
- ◆ 매치하는 항목을 찾으면 __free_page() 호출

```
void __free_page(struct page* page)
```

- ◆ LRU리스트의 제거
- ◆ Page 구조체에 할당받은 메모리 공간을 해제



get_next_lru_clock() 함수 구현

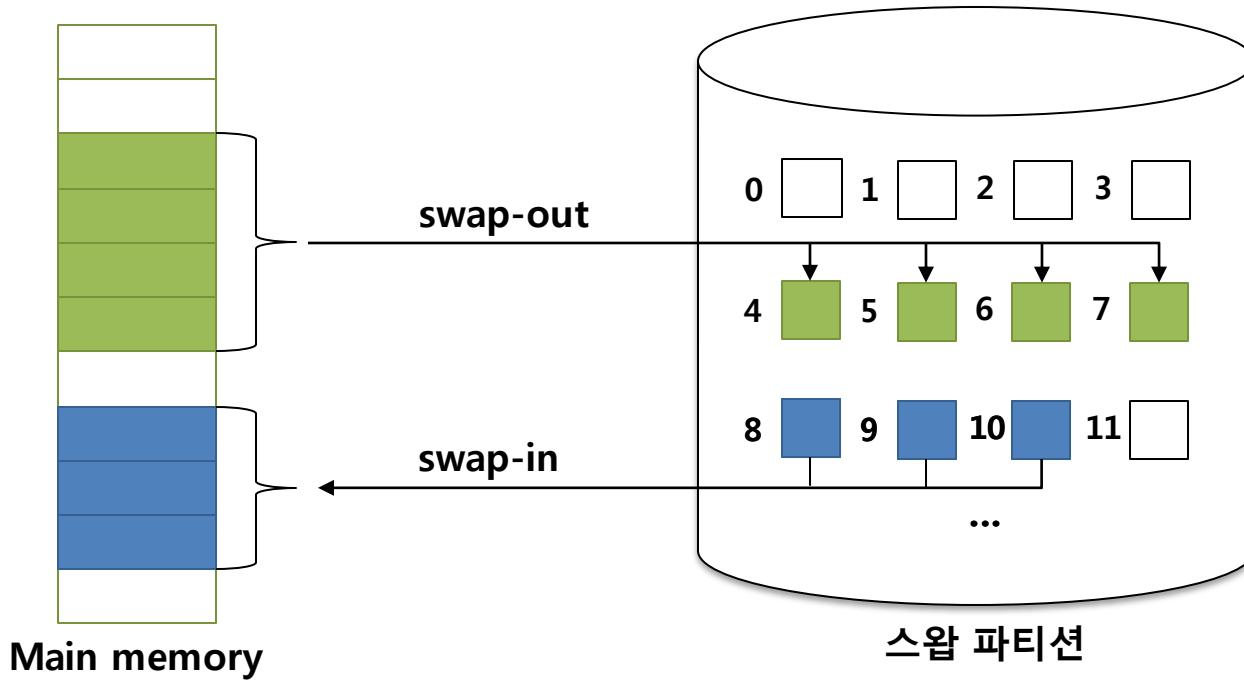
```
static struct list_elem* get_next_lru_clock()
```

- ◆ Clock 알고리즘의 LRU리스트를 이동하는 작업 수행
- ◆ LRU리스트의 다음 노드의 위치를 반환
 - 현재 LRU리스트가 마지막 노드 일 때 NULL 값 을 반환



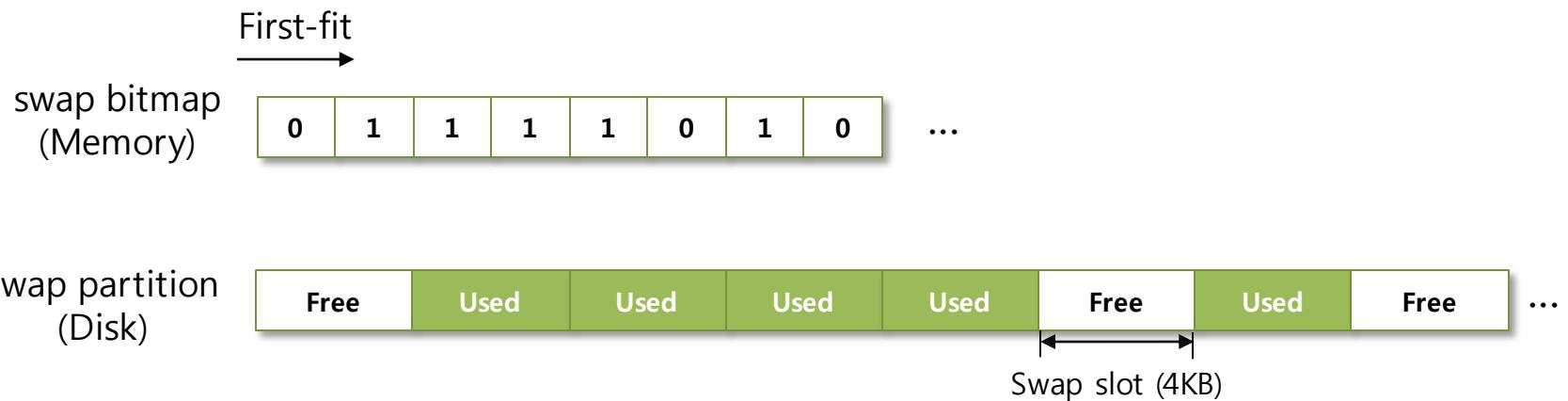
스와핑

- ▣ Pintos는 스왑 파티션을 스왑 스페이스로 제공
 - ◆ make check test에서 기본적으로 4MB의 swap partition을 사용



스왑 파티션 공간 관리

- ▣ 스왑 파티션은 swap slot(4 Kbyte) 단위로 관리
- ▣ swap bitmap(메모리에 존재)
 - ◆ swap slot의 사용가능 여부를 표시
- ▣ 여유 swap slot 탐색
 - ◆ bitmap을 first-fit 알고리즘을 이용하여 탐색



스왑 관련 함수 추가

▣ 헤더파일 추가

- ◆ #include "vm/swap.h"

```
void swap_init(size_t used_index, void* kaddr)
```

- ◆ swap 영역 초기화

```
void swap_in(size_t used_index, void* kaddr)
```

- ◆ used_index의 swap slot에 저장된 데이터를 논리 주소 kaddr로 복사

```
size_t swap_out(void* kaddr)
```

- ◆ kaddr 주소가 가리키는 페이지를 스왑 파티션에 기록
- ◆ 페이지를 기록한 swap slot 번호를 리턴



스와핑 및 페이지 교체 정책 구현 항목

▣ LRU 리스트 생성

- ◆ 할당/해제를 위해 `alloc_page()` / `free_page()` 을 사용할 것
- ◆ 가상 메모리 및 Memory Mapped File 과제에서 유저 메모리 할당/해제 하는 부분도 `alloc_page()` 및 `free_page()` 를 사용하도록 수정

▣ 물리 페이지가 부족할 때, victim 페이지를 디스크로 Swap out함으로써 여유 메모리 확보

- ◆ Clock 알고리즘을 이용하여 victim page를 선정하도록 구현

▣ 스와핑 지원을 위한 페이지 폴트 핸들러 수정

스와핑 및 페이지 교체 정책 과제 관련 코드

▣ 제공되는 구조체 및 함수

- ◆ pintos/src/vm/page.h

```
struct page
```

- ◆ pintos/src/vm/frame.c

```
void add_page_to_lru_list(struct page* page)  
void del_page_from_lru_list(struct page* page)  
static struct list_elem* get_next_lru_clock()
```

- ◆ pintos/src/vm/swap.c

```
size_t swap_out(void* kaddr)  
void swap_in(size_t used_index, void* kaddr)
```



스와핑 및 페이지 교체 정책 과제 구현 코드

▣ pintos/src/vm/frame.c

```
void try_to_free_pages(enum palloc_flags flags)
```

- ◆ 물리 페이지가 부족 할 때 clock알고리즘을 이용하여 여유메모리 확보

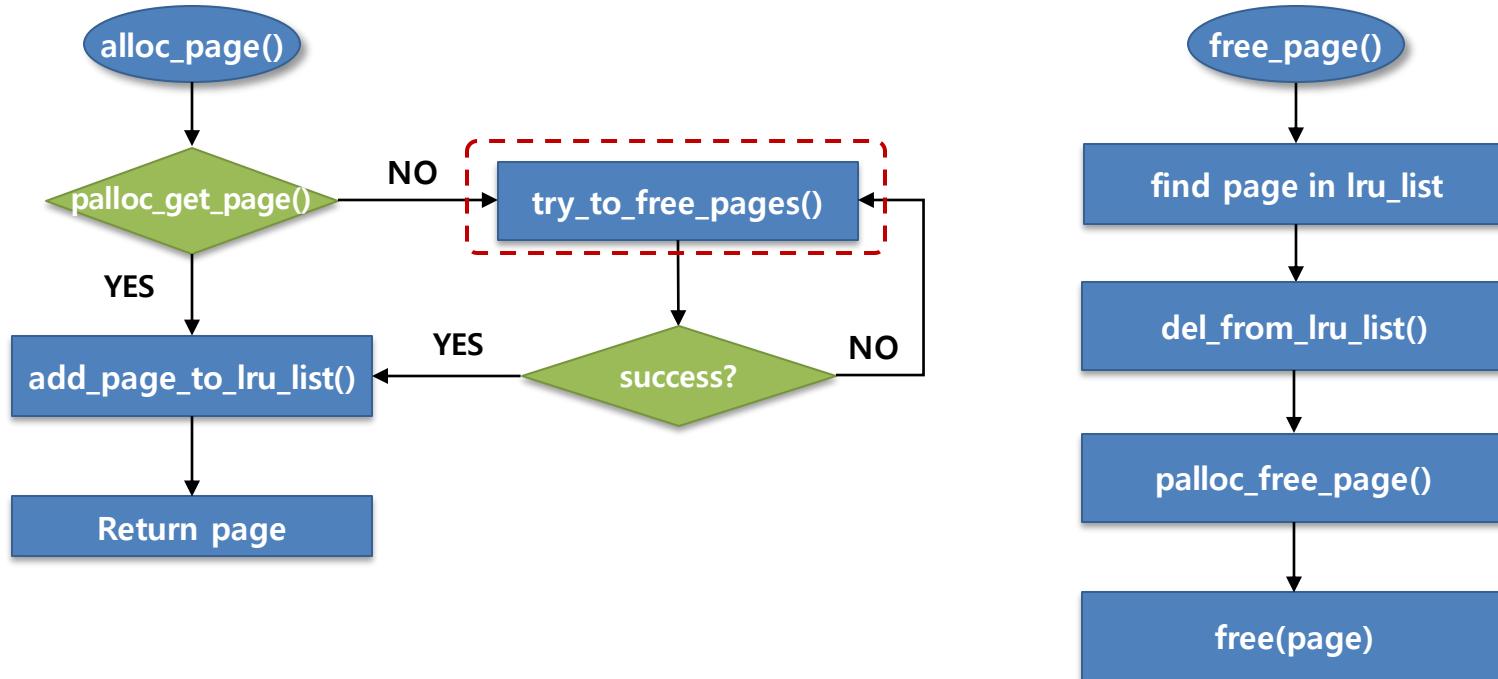
▣ pintos/src/userprog/process.c

```
bool handle_mm_fault(struct vm_entry *vme)
```

- ◆ 가상메모리 과제에서 구현한 함수
- ◆ 스와핑을 지원하도록 수정

alloc_page() & free_page() 추가

- alloc_page() 내에서 palloc_get_page()를 통해 메모리를 할당 받을 수 없을 때, 여유공간 확보를 위한 try_to_free_pages() 호출



유저 메모리 할당 및 해제 시 방법 수정

▣ 할당 함수 변경

- ◆ `palloc_get_page()` → `alloc_page()`
 - 리턴 데이터 타입: `void*` → `struct page*`
- ◆ 반환된 page 구조체의 vme 필드 초기화
 - vme 필드를 할당된 물리 페이지가 맵핑되는 논리 페이지의 vme 구조체 주소로 초기화

▣ 해제 함수 변경

- ◆ `palloc_free_page()` → `free_page()`

▣ 수정 위치

```
static bool setup_stack(void **esp)
```

```
void do_munmap(struct mmap_file* mmap_file)
```

```
bool handle_mm_fault(struct vm_entry *vme)
```



setup_stack() 함수 수정

- 할당 함수 변경
- 해제 함수 변경

pintos/src/userprog/process.c

```
static bool
setup_stack (void **esp)
{
    ...
    void *kaddr = palloc_get_page(PAL_USER | PAL_ZERO);
    vme->vaddr = ((uint8_t *) PHYS_BASE) - PGSIZE;
    vme->writable = true;
    vme->type = VM_ANON;
    vme->is_loaded = true;

    ...
    if (!install_page(vme->vaddr, kaddr, vme->writable)) {
        palloc_free_page(kaddr);
        return success;
    }
    ...
}
```

do_munmap() 함수 수정

▣ 해제 함수 변경

pintos/src/userprog/syscall.c

```
void do_munmap(struct mmap_file* mmap_file) {
    ...
    if(vme->is_loaded) {
        if(pagedir_is_dirty(t->pagedir, vme->vaddr)) {
            lock_acquire(&filesys_lock);
            file_write_at();
            lock_release(&filesys_lock);
        }
        palloc_free_page(pagedir_get_page(t->pagedir, vme->
vaddr));
    }
    ...
}
```



handle_mm_fault() 함수 수정

- 할당 함수 변경
- 해제 함수 변경

pintos/src/userprog/process.c

```
bool handle_mm_fault(struct vm_entry *vme) {
    ...
    kaddr = palloc_get_page(PAL_USER);
    switch(vme->type) {
        ...
    }
    if (!success)
    {
        palloc_free_page(kaddr);
        return false;
    }
    ...
}
```

try_to_free_pages() 함수 구현

```
void* try_to_free_pages (enum palloc_flags flags)
```

- ◆ Clock 알고리즘을 이용하여 victim page를 선정하고 방출하여, 여유 메모리 공간을 확보하고 여유 페이지의 커널 가상 주소를 리턴 하도록 구현

물리 페이지 방출

- ▣ victim의 방출은 vm_entry의 type에 따라 다른 방식이 요구
 - ◆ VM_BIN
 - Dirty bit가 1이면, 스왑 파티션에 기록 후 페이지 해제
 - 요구페이지를 위해 type을 VM_ANON으로 변경
 - ◆ VM_FILE
 - Dirty bit가 1이면, 파일에 변경 내용을 저장 후 페이지 해제
 - Dirty bit가 0이면, 바로 페이지 해제
 - ◆ VM_ANON
 - 항상 스왑 파티션에 기록

참고 함수

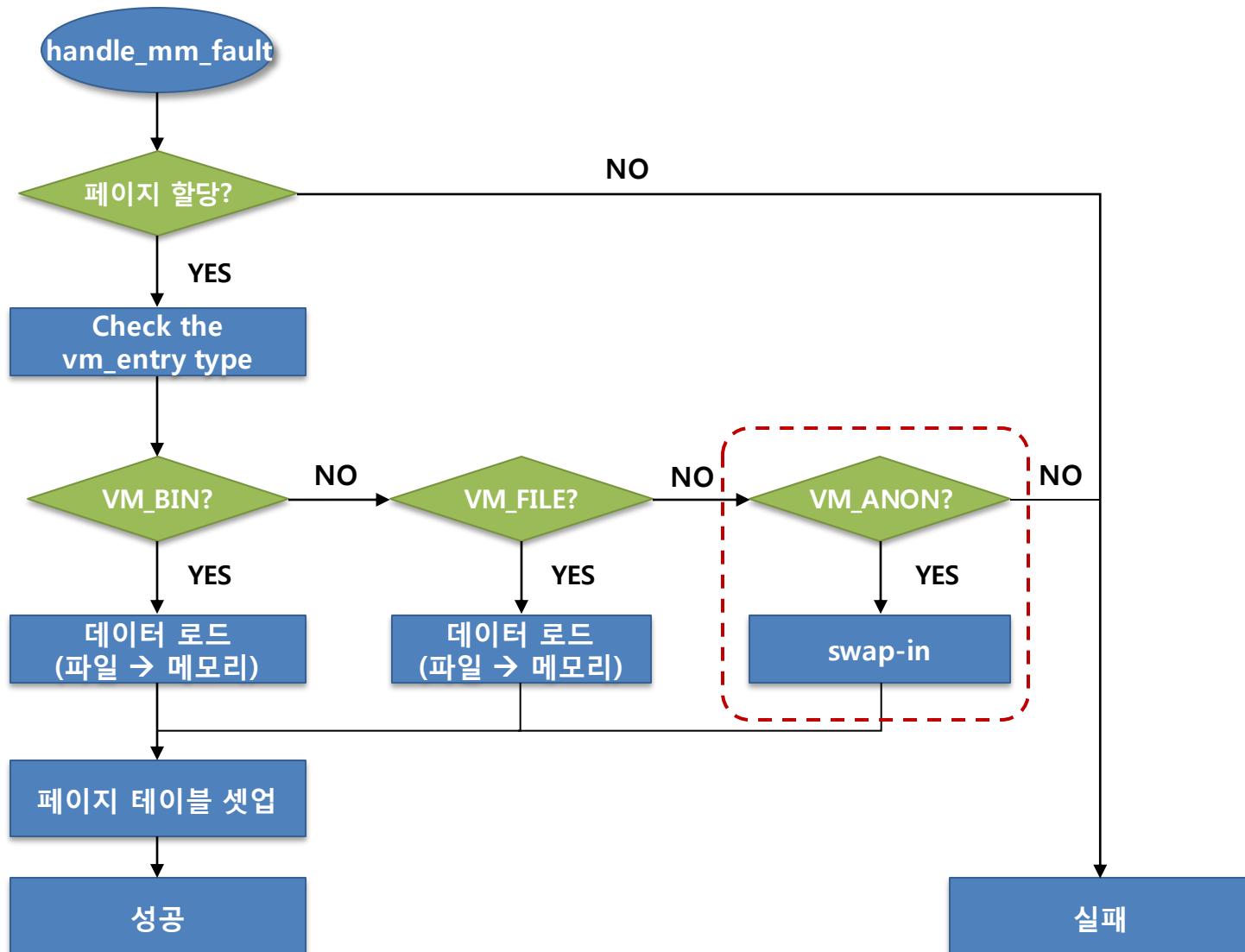
- ▣ pintos/src/userprog/pagedir.c

```
void pagedir_clear_page (uint32_t *pd, void *upage)
```

- ◆ 페이지 테이블에서 upage에 해당하는 주소의 엔트리를 제거



스왑 영역 요구 페이징 구현



handle_mm_fault() 함수 수정

- vm_entry 탑이 VM_ANON시 swap_in할 수 있도록 코드 수정

pintos/src/userprog/process.c

```
bool handle_mm_fault(struct vm_entry *vme) {
    bool success = false;
    viod *kaddr;
    ...
    switch(vme->type) {
        case VM_BIN:
            success = load_file(kaddr, vme);
            break;

        case VM_FILE:
            success = load_file(kaddr, vme);
            break;

        case VM_ANON:
            /* swap_in하는 코드 삽입 */
            break;
    }
    ...
}
```

스와핑 및 페이지 교체 정책 과제 검증

- ▣ 스와핑 및 페이지 교체 정책 과제를 완료 후 코드 동작 확인
 - ◆ 경로 : pintos/src/vm
- \$ make check
- ▣ 실행 결과 109개의 테스트 중 5개에서 fail 발생
 - ◆ 다음과제 완료후 모두 패스 가능
 - ◆ pt-grow-stack ◆ pt-grow-pusha ◆ pt-big-stk-obj ◆ pt-grow-stk-sc
 - ◆ page-merge-stk

```
root@gaya: /home/gaya/VM/Project3/p_3/pintos/src/vm
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
-----)
5 of 109 tests failed.
```

추가 함수

```
void lru_list_init(viod)
/* LRU list를 초기화 */

void add_page_to_lru_list(struct page* page)
/* LRU list의 끝에 유저 페이지 삽입 */

void del_page_from_lru_list(struct page* page)
/* LRU list에 유저 페이지 제거 */

struct page* alloc_page(enum palloc_flags flags)
/* 페이지 할당 */

void free_page(void *kaddr)
/* 물리 주소 kaddr에 해당하는 page해제 */
```



추가 함수

```
void __free_page(struct page* page)
/* LRU list 리스트 내 page 해제 */

static struct list_elem* get_next_lru_clock()
/* LRU list 리스트의 next 리스트를 반환 */

void try_to_free(enum palloc_free_page)
/* Clock 알고리즘을 이용하여 victim page를 선정하고 방출 */
```



추가 함수

```
void swap_init(void)
    /* swap 영역 초기화 */

size_t swap_out(void *kaddr)
    /* 메모리의 내용을 디스크의 스왑 영역으로 방출 */

void swap_in(size_t used_index, void *kaddr)
    /* swap-out된 페이지를 다시 메모리로 적재 */

static struct list_elem* get_next_lru_clock()
    /* LRU list 내에 다음 clock 탐색 */

void try_to_free_pages(enum palloc_flags flags)
    /* 물리 페이지가 부족 할 때 clock 알고리즘을 이용하여 여유메모리 확보 */
```

수정 함수

```
bool handle_mm_fault(struct vm_entry *vme)
    /* 페이지 폴트 발생시 물리페이지를 할당 */

static bool setup_stack(void **esp)
    /* 가상 메모리의 스택부분을 초기화 하는 함수 */

void do_munmap(struct mmap_file* mmap_file)
    /* mmap_file의 vme_list에 연결된 모든 vm_entry들을 제거 */

static void syscall_handler(struct intr_frame *f UNUSED)
    /* 시스템 콜 넘버에 해당하는 시스템 콜을 호출 하는 함수 */

int main(void)
    /* Pintos를 실행하는 함수 */
```



14. Stack

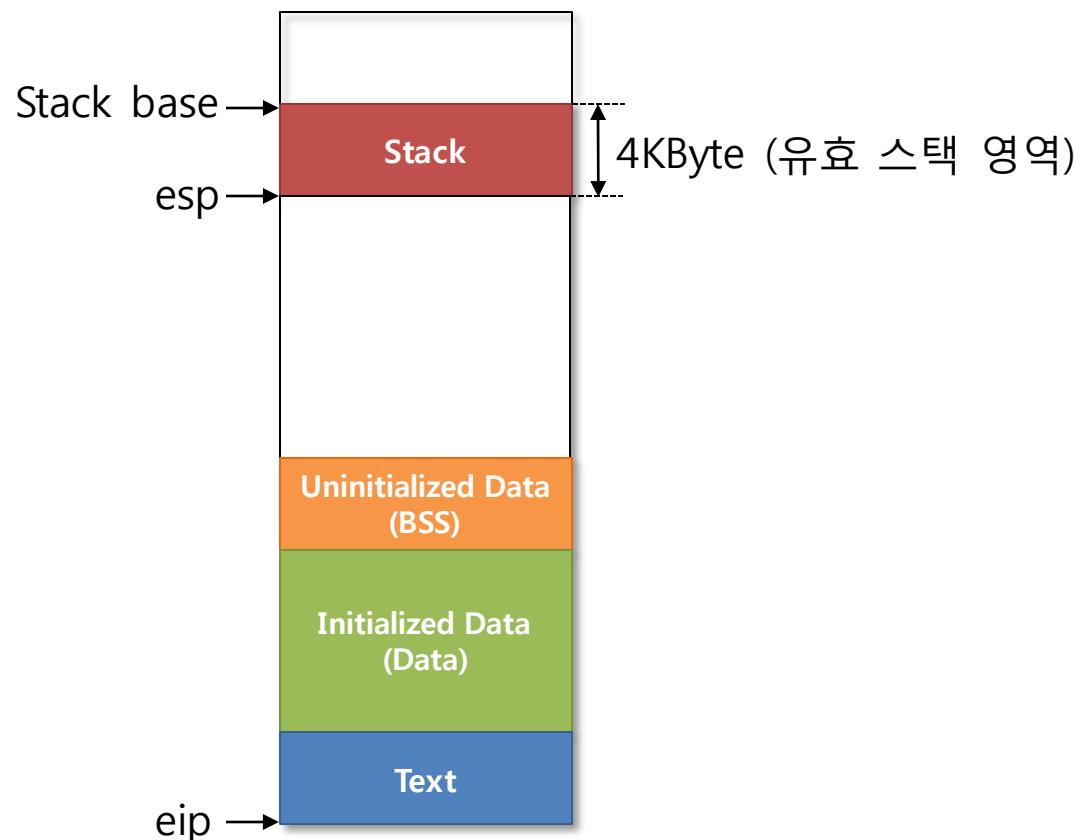
스택 확장 개요

▣ 확장 가능한 스택 구현

- ◆ 현재 스택의 크기는 4KB로 고정
- ◆ 현재 스택의 크기를 초과하는 주소에 접근이 발생했을 때, 유효한 스택 접근인지 세그멘테이션 폴트인지 판별하는 휴리스틱을 적용하도록 코드 수정
 - 유효한 스택 접근으로 판별한 경우, 스택을 확장
 - 예) (접근 주소 > 스택 포인터 – 32) 이면 스택 확장
- ◆ 확장 가능한 스택의 최대 크기는 8MB가 되도록 코드 수정

스택 확장 개요

- ▣ Pintos는 스택의 크기를 4KByte로 고정
- ▣ 스택 포인터(esp)가 Stack 영역을 벗어나면, 세그멘테이션 폴트 발생



스택 확장 개요 (Cont.)

▣ 수정 파일

- ◆ pintos/src/userprog/exception.c

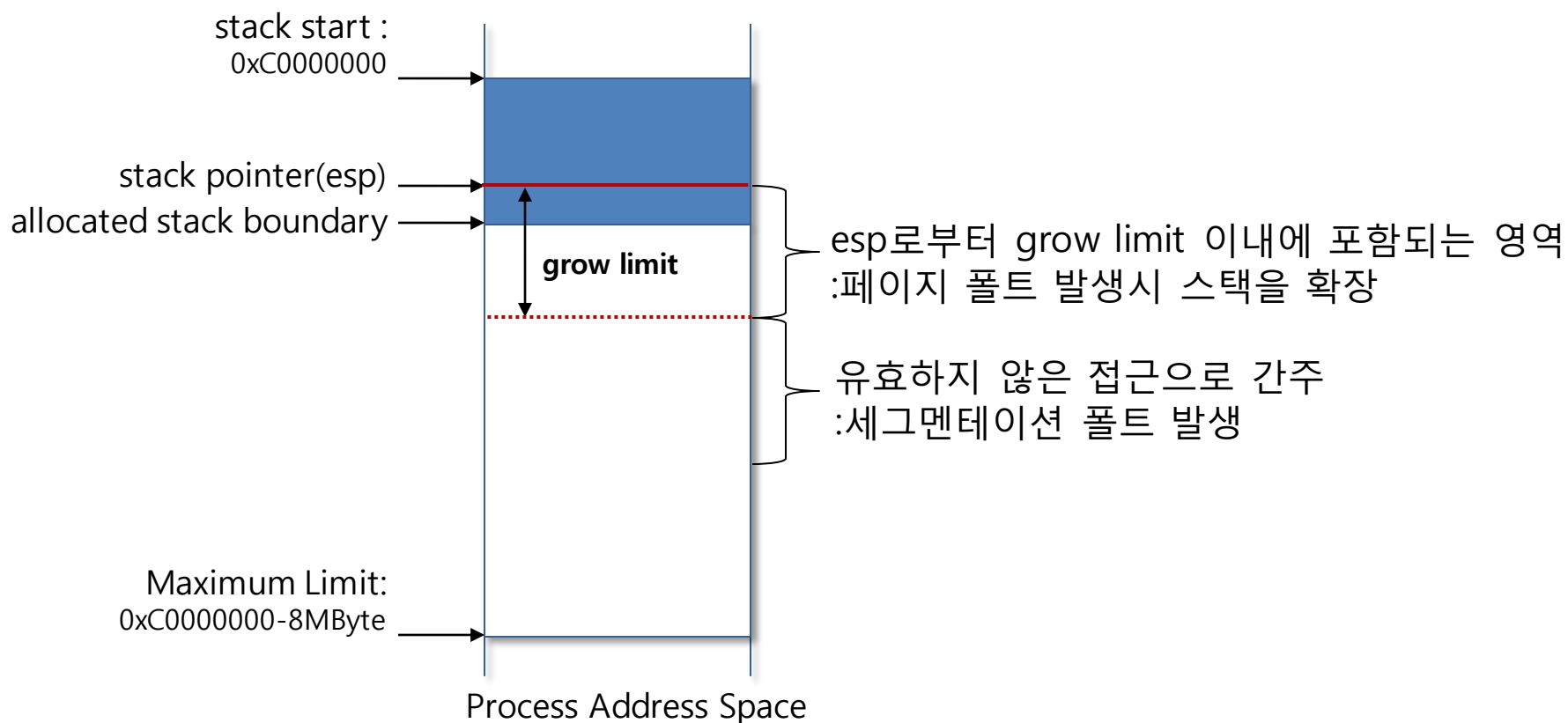
▣ 추가 파일

- ◆ pintos/src/userprog/process.c
- ◆ pintos/src/userprog/syscall.c



확장 가능한 스택의 구현

- ▣ 최대 8MB까지 확장
- ▣ 현재 스택 포인터로부터 grow limit 이내에 포함되는 접근은 유효한 스택 접근으로 간주하여 스택을 확장



스택 확장 과제 관련 코드

▣ 사용되는 구조체 및 함수

- ◆ pintos/src/vm/frame.c

```
struct page *alloc_page(enum palloc_flags flags)  
void free_page(void *addr)
```

스택 확장 과제 구현관련 소스 코드

▣ pintos/src/userprog/process.c

```
bool expand_stack(void *addr)
```

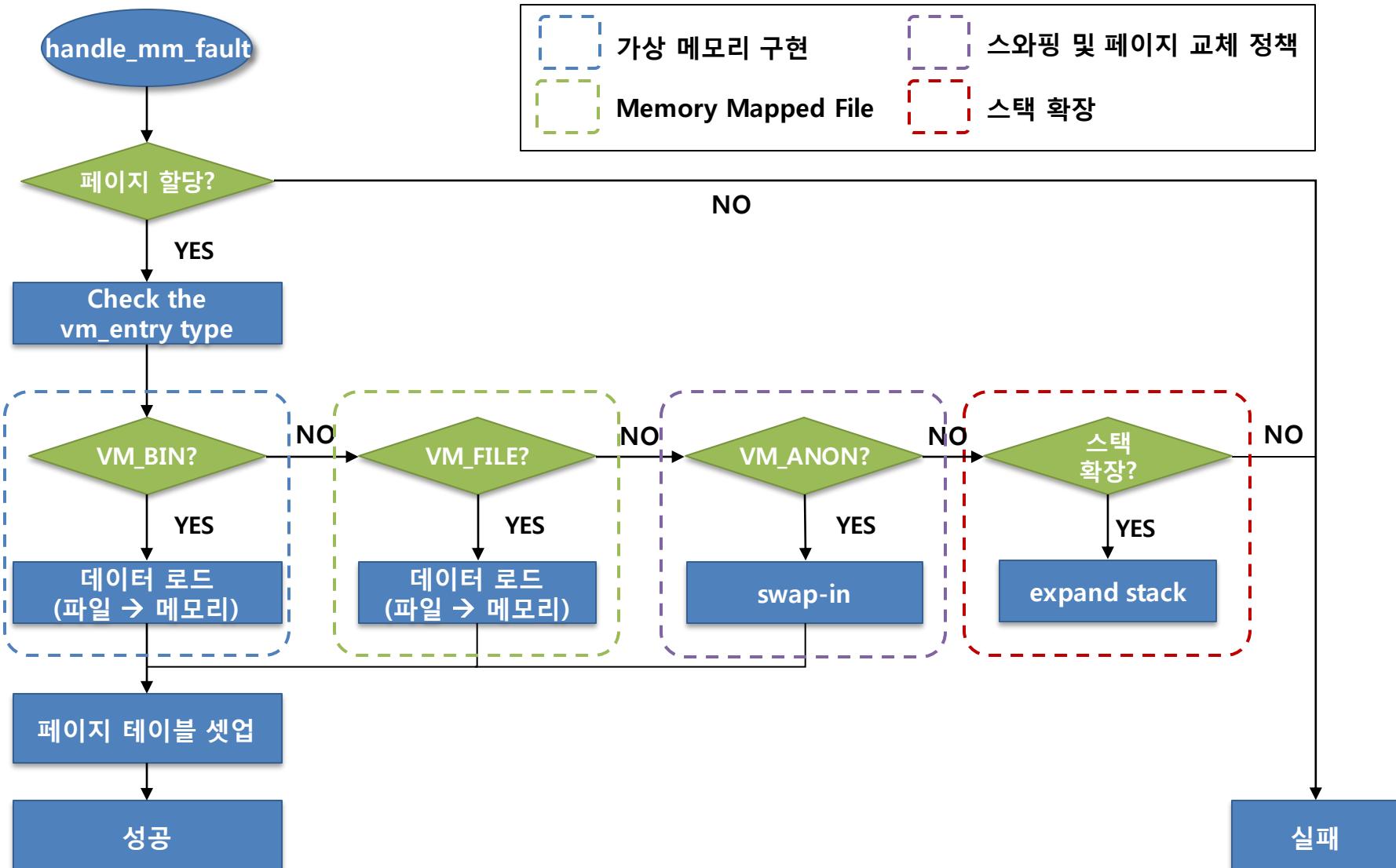
- ◆ addr 주소를 포함하도록 스택을 확장
- ◆ 최대 8MB까지 확장이 가능

```
bool handle_mm_fault(struct vm_entry *vme)
```

- ◆ 가상 메모리 과제에서 구현한 함수
- ◆ 스택 확장을 지원하도록 수정



스택 확장 처리



스택 확장 가능 여부 결정 - page_fault()

- 휴리스틱을 적용하여 스택 확장 여부를 판단
 - ◆ STACK_HEURISTIC 사용
- expand_stack() 호출하여 스택을 확장

pintos/src/userprog/exception.c

```
...
if(not_present)
{
    struct vm_entry *vme = find_vme(fault_addr);
    if(vme)
    {
        ...
    }
    /* fault_addr이 스택영역인지 확인 후 expand_stack()
     * 호출 */
}
...
```

스택 확장 과제 함수 구현

```
bool expand_stack(void* addr)
```

- ◆ addr 주소를 포함하도록 스택을 확장
- ◆ alloc_page()를 통해 메모리 할당
- ◆ vm_entry의 할당 및 초기화
- ◆ install_page() 호출하여 페이지 테이블 설정
- ◆ 성공시 true를 리턴



verify_stack() 함수 구현

```
bool verify_stack(void *sp)
```

- ◆ sp 주소가 포함되어 있는지 확인하는 함수
- ◆ sp 주소가 존재할 시 handle_mm_fault() 함수 호출



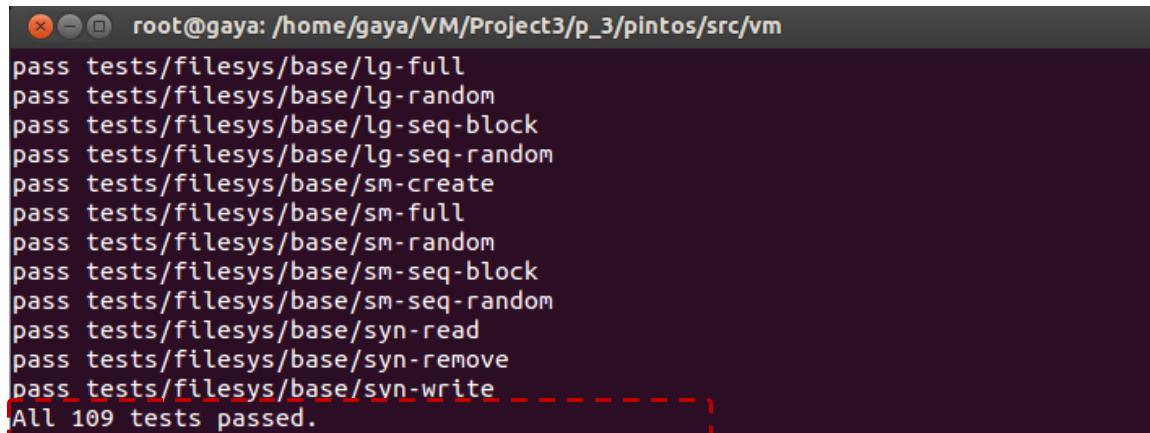
스택 확장 과제 검증

- 스택 확장 과제를 완료 후 코드 동작 확인

- 경로 : pintos/src/vm

```
$ make check
```

- 실행 결과 109개의 테스트 모두 통과



A terminal window titled 'root@gaya: /home/gaya/VM/Project3/p_3/pintos/src/vm' displays the output of a 'make check' command. The output shows 109 tests all passing, with each test result starting with 'pass'. The final message is 'All 109 tests passed.'.

```
root@gaya: /home/gaya/VM/Project3/p_3/pintos/src/vm
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 109 tests passed.
```

추가 및 수정 함수

```
bool expand_stack(void *addr)  
/* addr 주소를 포함하도록 스택을 확장 */
```

```
bool verify_stack(void *sp)  
/* sp 주소가 포함되어 있는지 확인하는 함수 */
```

```
static void page_fault(struct intr_frame *f)  
/* 페이지 폴트 핸들러 */
```

15. Buffer Cache

Buffer cache 개요

▣ 과제목표

- ◆ Buffer cache는 디스크 블록을 캐싱하는 메모리 영역이다. 디스크블록을 메모리영역에 둠으로써 파일의 입출력 응답시간을 줄인다. 현재 pintos에서는 buffer cache가 존재하지 않으므로, 파일 입출력 시 바로 디스크 입출력 동작을 수행한다. 이번 과제에서는 buffer cache를 구현하고, 성능향상을 살펴보기로 한다.

▣ 수정파일

- ◆ pintos/src/filesys/inode.c

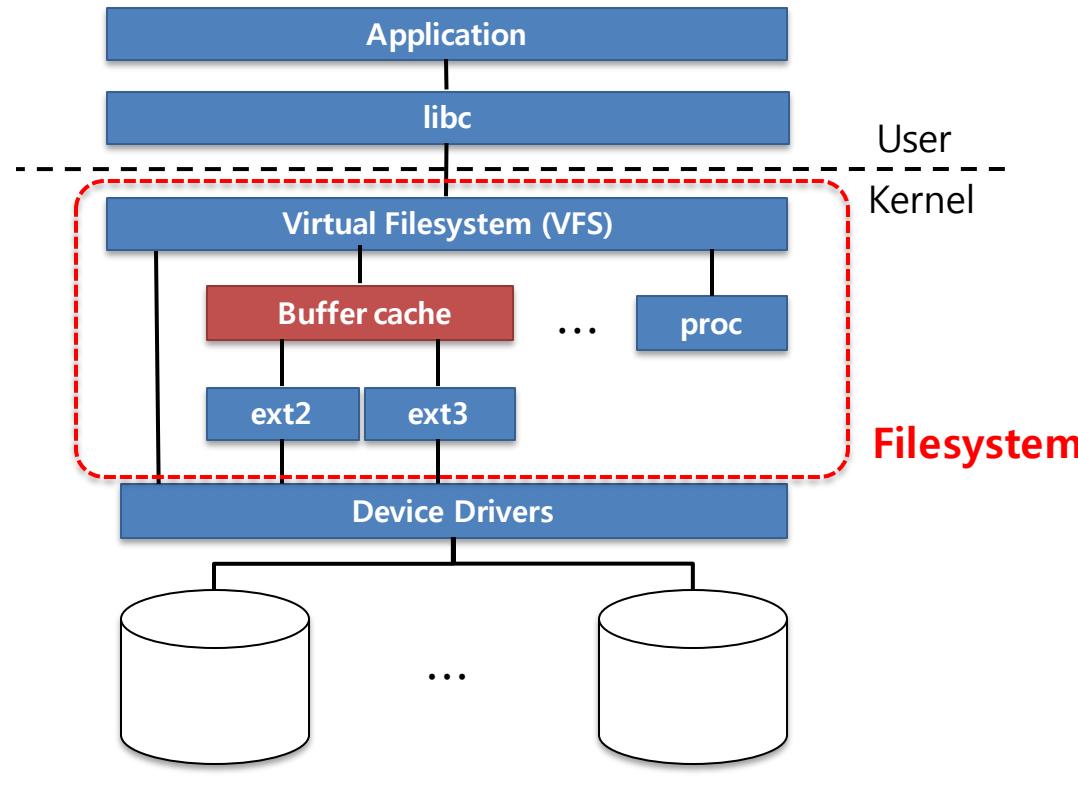
▣ 생성파일

- ◆ pintos/src/filesys/buffer_cache.h
- ◆ pintos/src/filesys/buffer_cache.c



Buffer cache 개요

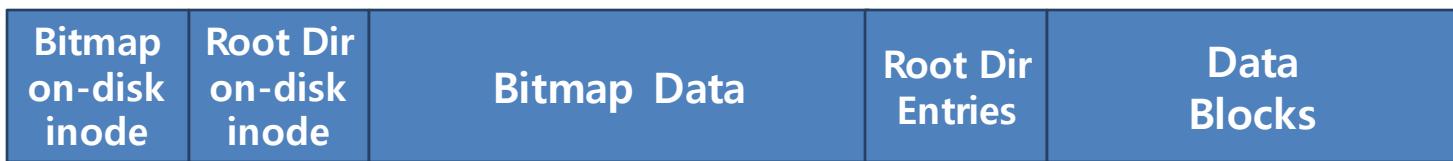
- Filesystem: OS가 디렉터리나 파일을 생성, 접근 및 보관할 수 있도록 하는 계층
 - 블록디바이스를 일정한 크기의 블록(sector)들로 관리
 - 블록의 사용 여부를 bitmap을 통해 관리



Filesystem Block Layout

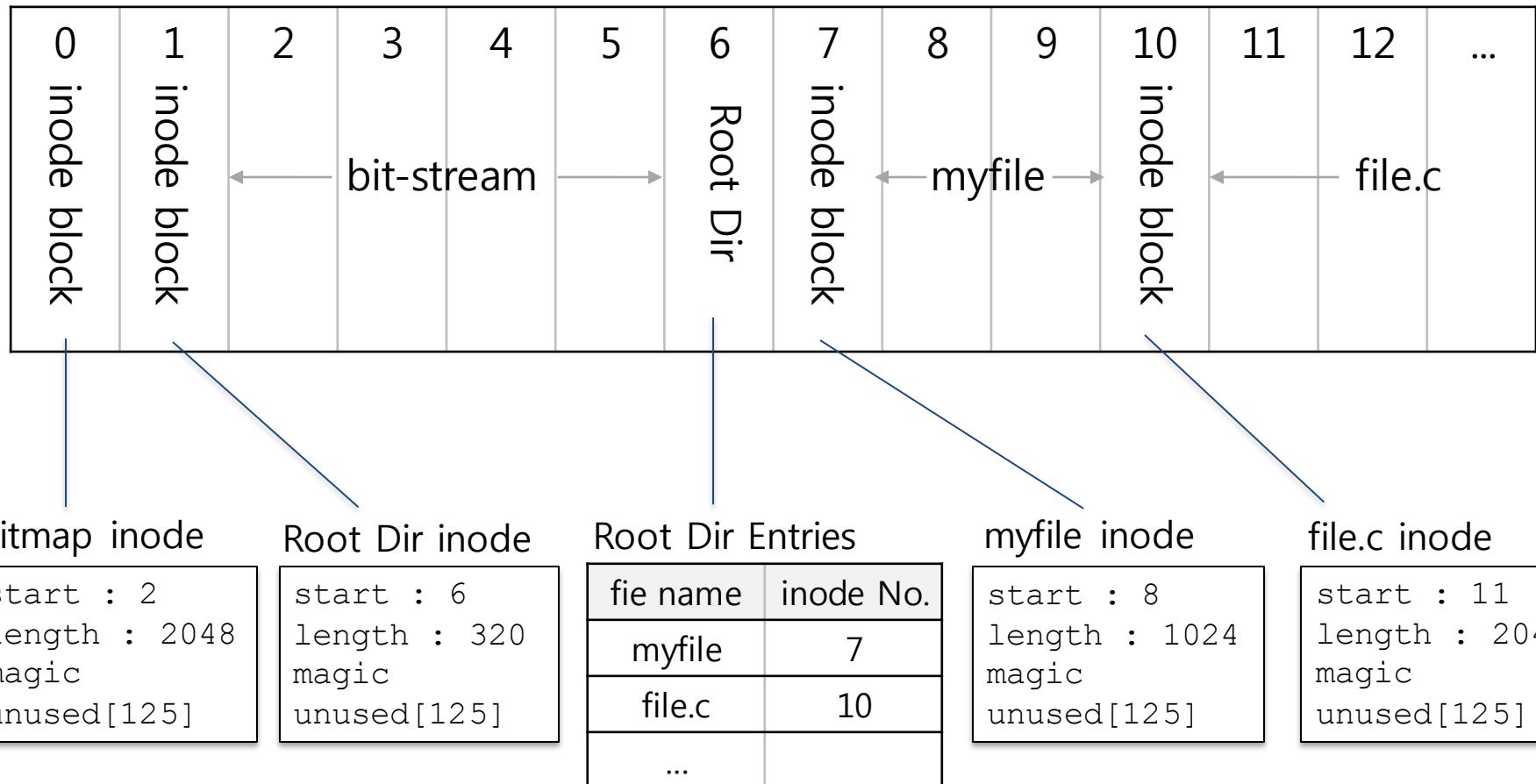
- ▣ Pintos의 8MByte Filesystem 예시
 - ◆ $8\text{MByte}/512\text{Byte} = 16,384 = \text{총 블록의 개수}$

Block number: 0 1 2~5 6 7~16,383



inode_sector	inode_sector	...
name[14]	name[14]	...
in_use = 0	in_use = 0	...

Filesystem Block Layout (Cont.)



파일의 표현

▣ In-memory inode: struct inode

- ◆ sector : inode가 저장된 블록 번호
- ◆ data : disk_inode 데이터
- ◆ removed : 파일의 삭제 여부

pintos/src/filesys/inode.c

```
struct inode {  
    struct list_elem elem; /* Element in inode list. */  
    block_sector_t sector;  
    int open_cnt;           /* Number of openers. */  
    bool removed;  
    int deny_write_cnt;    /* 0: writes ok, >0: deny writes. */  
    struct inode_disk data; /* Inode content. */  
};
```

파일의 표현

▣ On-disk inode (struct inode_disk)

- ◆ start : 파일 데이터의 디스크 블록 시작 주소
- ◆ length : 파일의 크기(byte)
- ◆ unused[125] : 사용되지 않는 영역

pintos/src/filesys/inode.c

```
struct inode_disk
{
    block_sector_t start;          /* First data sector. */
    off_t length;                 /* File size in bytes. */
    unsigned magic;               /* Magic number. */
    uint32_t unused[125];         /* Not used. */
};
```

디렉터리

▣ 디렉터리의 정보를 관리하는 자료구조

- ◆ inode : 디렉터리의 in-memory inode를 포인팅
- ◆ pos : 디렉터리 엔트리 오프셋

pintos/src/filesys/directory.c

```
struct dir
{
    struct inode *inode;      /* Backing store. */
    off_t pos;                /* Current position. */
};
```



디렉터리 엔트리 자료 구조

- ▣ 디렉터리 엔트리(파일 또는 디렉터리)의 정보를 나타냄
 - ◆ inode_sector : inode의 디스크 블록번호를 저장
 - ◆ 파일 이름 길이는 최대 14 character로 한정
 - ◆ in_use : dir_entry의 사용 여부

pintos/src/filesys/directory.c

```
struct dir_entry
{
    block_sector_t inode_sector;
    char name[NAME_MAX + 1]; /* NAME_MAX = 14 */
    bool in_use;
};
```



블록 비트맵

▣ free_map

- ◆ 디스크 블록의 free/used 표현하는 비트맵
- ◆ free_map_file : bitmap을 디스크에 파일 형태로 저장
- ◆ bit_cnt : 파일 시스템 전체의 디스크 블록 개수

pintos/src/kernel(bitmap.c

```
static struct bitmap *free_map;
static struct file *free_map_file;
struct bitmap
{
    size_t bit_cnt;
    elem_type *bits;
};
```

사용중인 파일의 표현

▣ struct file

- ◆ inode : 파일의 in-memory inode 포인터
- ◆ pos : 파일 오프셋
- ◆ deny_write : write 명령 가능 여부

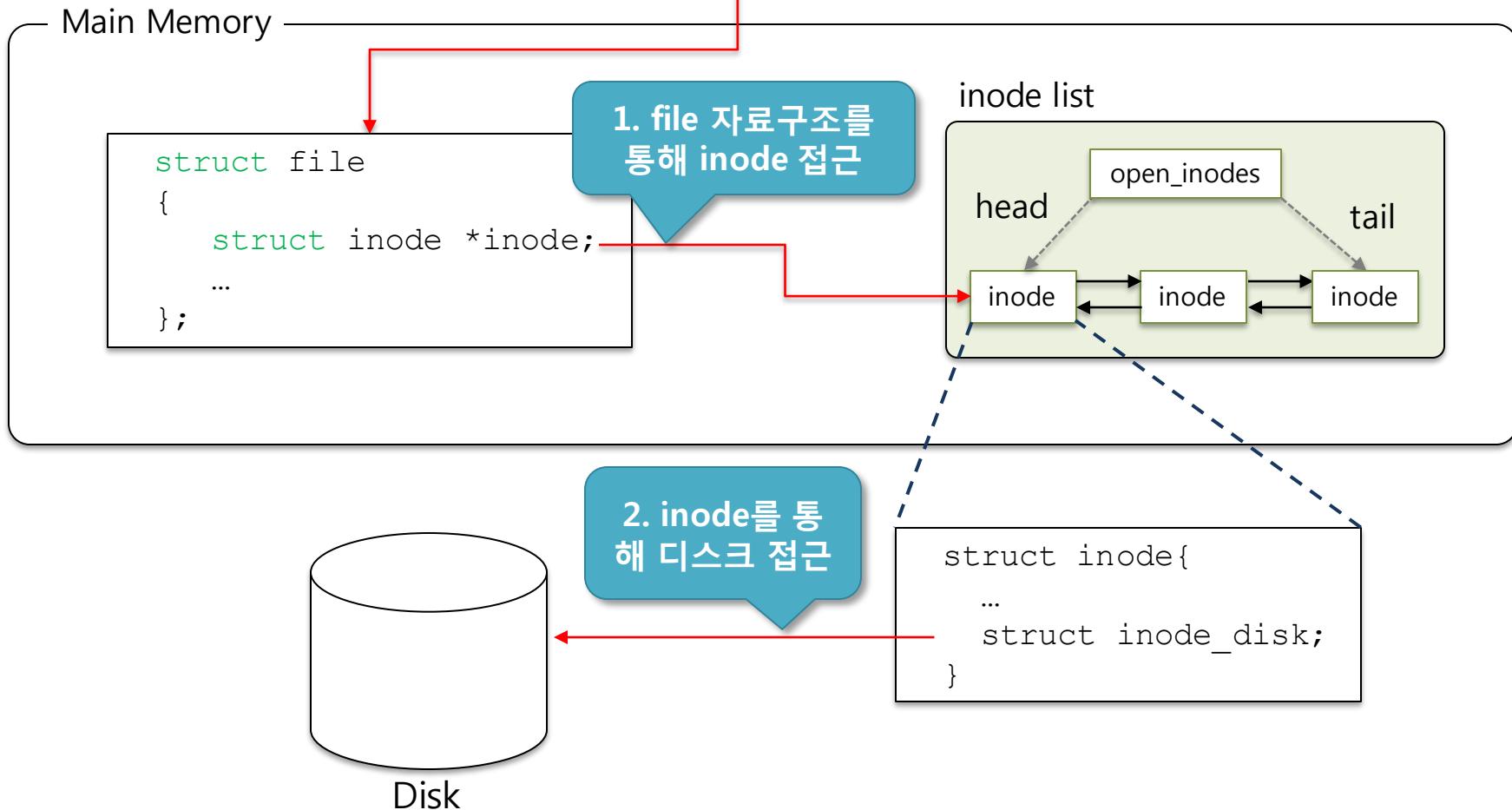
pintos/src/filesys/file.c

```
struct file {  
    struct inode *inode;          /* File's inode. */  
    off_t pos;                  /* Current position. */  
    bool deny_write;  
};
```

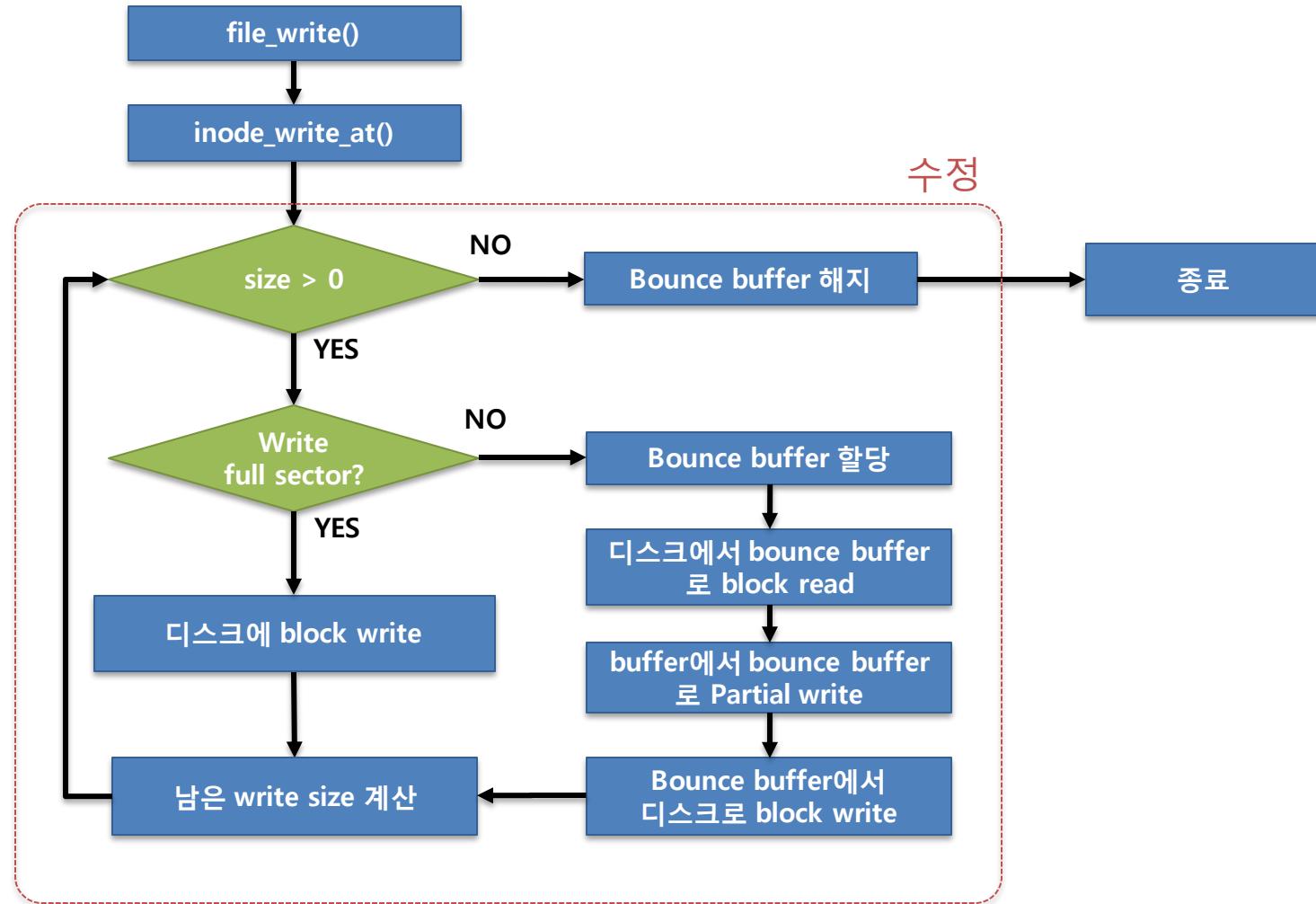
디스크 블록 접근 순서

ex) File read 함수 호출 시

- ◆ `off_t file_read (struct file *file, void *buffer, off_t size)`



Write



Write (Cont.)

▣ file_write

- ◆ inode_write_at()을 호출하여 디스크 블록에 데이터를 기록
 - 기록한 크기 만큼 파일 오프셋 변경

pintos/src/filesys/file.c

```
off_t file_write (struct file *file, const void *buffer,
                  off_t size)
{
    off_t bytes_written = inode_write_at (file->inode, buffer,
size, file->pos);
    file->pos += bytes_written;
    return bytes_written;
}
```

```
struct file{
    struct inode *inode;
    off_t pos;
    ...
}
```

```
→ struct inode{
    struct inode_disk data;
    ...
}
```

Write (Cont.)

▣ inode_write_at

- ◆ buffer가 가리키는 영역의 데이터를 디스크에 기록

pintos/src/filesys/inode.c

```
off_t inode_write_at (struct inode *inode, const void
                      *buffer_, off_t size, off_t offset)
{
    const uint8_t *buffer = buffer_;
    off_t bytes_written = 0;
    uint8_t *bounce = NULL;
    if (inode->deny_write_cnt)
        return 0;
```

Write (Cont.)

- ◆ 디스크 블록 단위로 loop를 돌며 디스크 블록에 기록: `block_write()`
 - `byte_to_sector()`: 데이터를 기록할 디스크 블록 번호를 얻음
 - `sector_ofs`: 데이터를 기록할 디스크 블록 내부의 오프셋

pintos/src/filesys/inode.c – `inode_write_at()` 계속

```
while (size > 0) {  
    block_sector_t sector_idx = byte_to_sector (inode, offset);  
    int sector_ofs = offset % BLOCK_SECTOR_SIZE;  
    ...  
    if (sector_ofs == 0 && chunk_size == BLOCK_SECTOR_SIZE) {  
        /* Write full sector directly to disk. */  
        block_write (fs_device, sector_idx, buffer +  
                    bytes_written);  
    }  
    ...
```

Write (Cont.)

- ◆ Partial Write의 경우, 타겟 블록을 읽어 bounce buffer에 저장
 - memcpy()를 통해, bounce buffer에 partial write 수행
 - bounce buffer의 데이터를 디스크에 기록 : block_write()

pintos/src/filesys/inode.c – inode_write_at() 계속

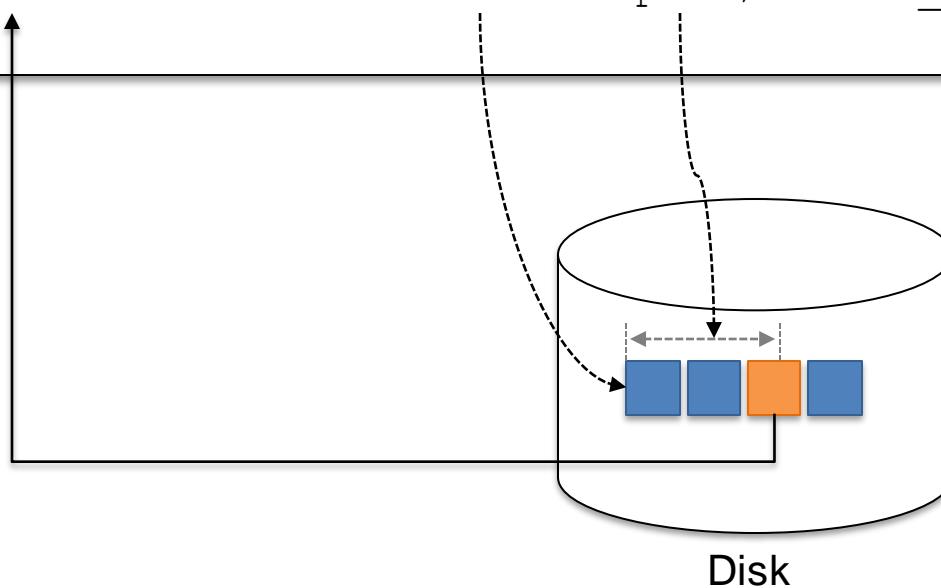
```
...
else {          /* We need a bounce buffer. */
    if (bounce == NULL)
        bounce = malloc (BLOCK_SECTOR_SIZE);
    if (sector_ofs > 0 || chunk_size < sector_left)
        block_read (fs_device, sector_idx, bounce);
    else
        memset (bounce, 0, BLOCK_SECTOR_SIZE);
    memcpy (bounce + sector_ofs, buffer + bytes_written,
            chunk_size);
    block_write (fs_device, sector_idx, bounce);
}
...
```

파일 오프셋을 통한 디스크 블록 주소 확인

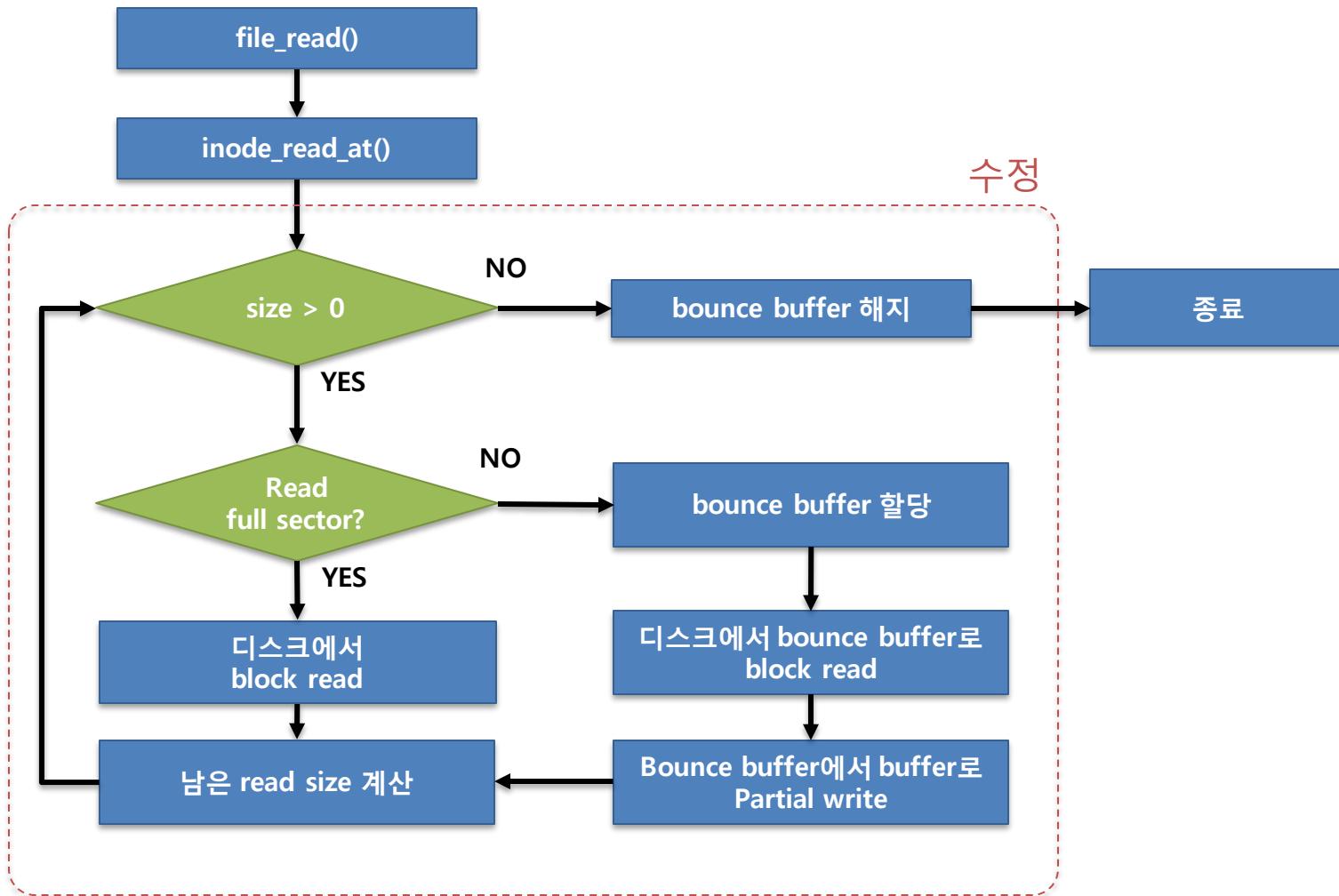
- 파일의 시작블록으로부터 오프셋값을 더하여 디스크 블록번호 반환

pintos/src/filesys/inode.c

```
static block_sector_t byte_to_sector (const struct inode *inode,
                                      off_t pos) {
    ...
    return inode->data.start + pos / BLOCK_SECTOR_SIZE;
}
```



Read



수정

종료

bounce buffer 해지

file_read()

inode_read_at()

size > 0

NO

bounce buffer 해지

종료

YES

Read
full sector?

NO

bounce buffer 할당

디스크에서
block read

디스크에서 bounce buffer로
block read

Bounce buffer에서 buffer로
Partial write

남은 read size 계산

남은 read size 계산

남은 read size 계산

Read (Cont.)

▣ file_read

- ◆ inode_read_at() 을 호출하여 디스크에서 버퍼로 데이터를 읽음
- ◆ 읽은 크기 만큼 파일 오프셋 변경

pintos/src/filesys/file.c

```
off_t file_read (struct file *file, void *buffer, off_t size){  
    off_t bytes_read = inode_read_at (file->inode, buffer, size,  
file->pos);  
    file->pos += bytes_read;  
    return bytes_read;  
}
```

```
struct file{  
    struct inode *inode;  
    off_t pos;  
    ...  
}
```

```
struct inode{  
    struct inode_disk data;  
    ...  
}
```

Read (Cont.)

▣ inode_read_at

- ◆ 디스크 블록 단위로 loop를 돌며 디스크에서 데이터 읽음: block_read()
- ◆ byte_to_sector(): 데이터를 읽을 디스크 블록 번호를 얻음
- ◆ sector_ofs: 데이터를 읽을 디스크 블록 내의 오프셋

pintos/src/filesys/inode.c

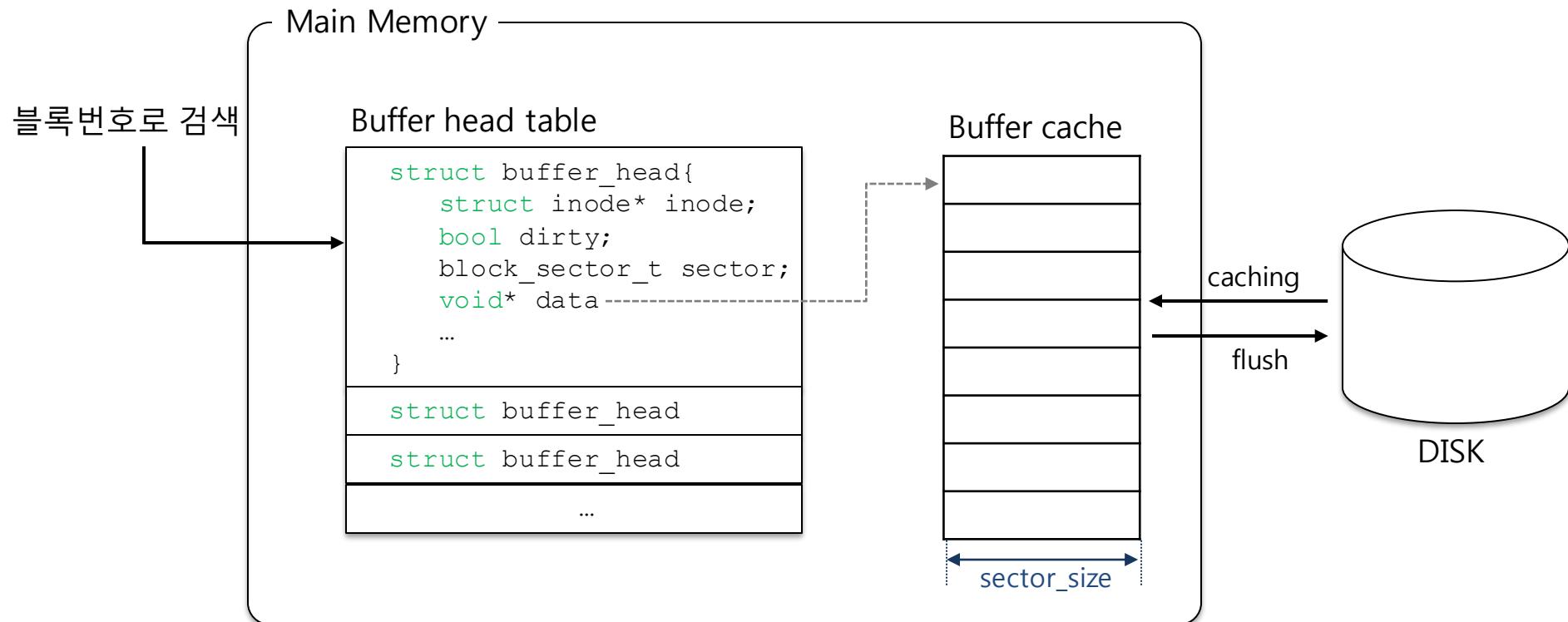
```
off_t inode_read_at (struct inode *inode, void *buffer_,  
                      off_t size, off_t offset) {  
    ...  
    while (size > 0) {  
        block_sector_t sector_idx = byte_to_sector (inode,  
offset);  
        int sector_ofs = offset % BLOCK_SECTOR_SIZE;  
        /* cache read로 대체 필요 */  
        block_read (fs_device, sector_idx, buffer + bytes_read);  
    }  
    ...  
}
```

Buffer cache 구현 세부 요구사항

▣ Buffer cache 관리 위한 인터페이스 구현

- ◆ 최대 64 블럭($64 \times 512\text{Byte} = 32\text{KByte}$)으로 구성
- ◆ Cache search, patch, select victim, flush 기능 구현
 - 교체 알고리즘 구현 ex) LRU, clock 알고리즘
- ◆ Write behind 구현
 - Filesystem 종료 시, 모든 dirty 블록을 디스크로 flush
 - victim entry 선정 시, dirty일 경우 블록을 디스크로 flush
- ◆ inode_write_at(), inode_read_at() 함수 수정

Buffer cache 구조도



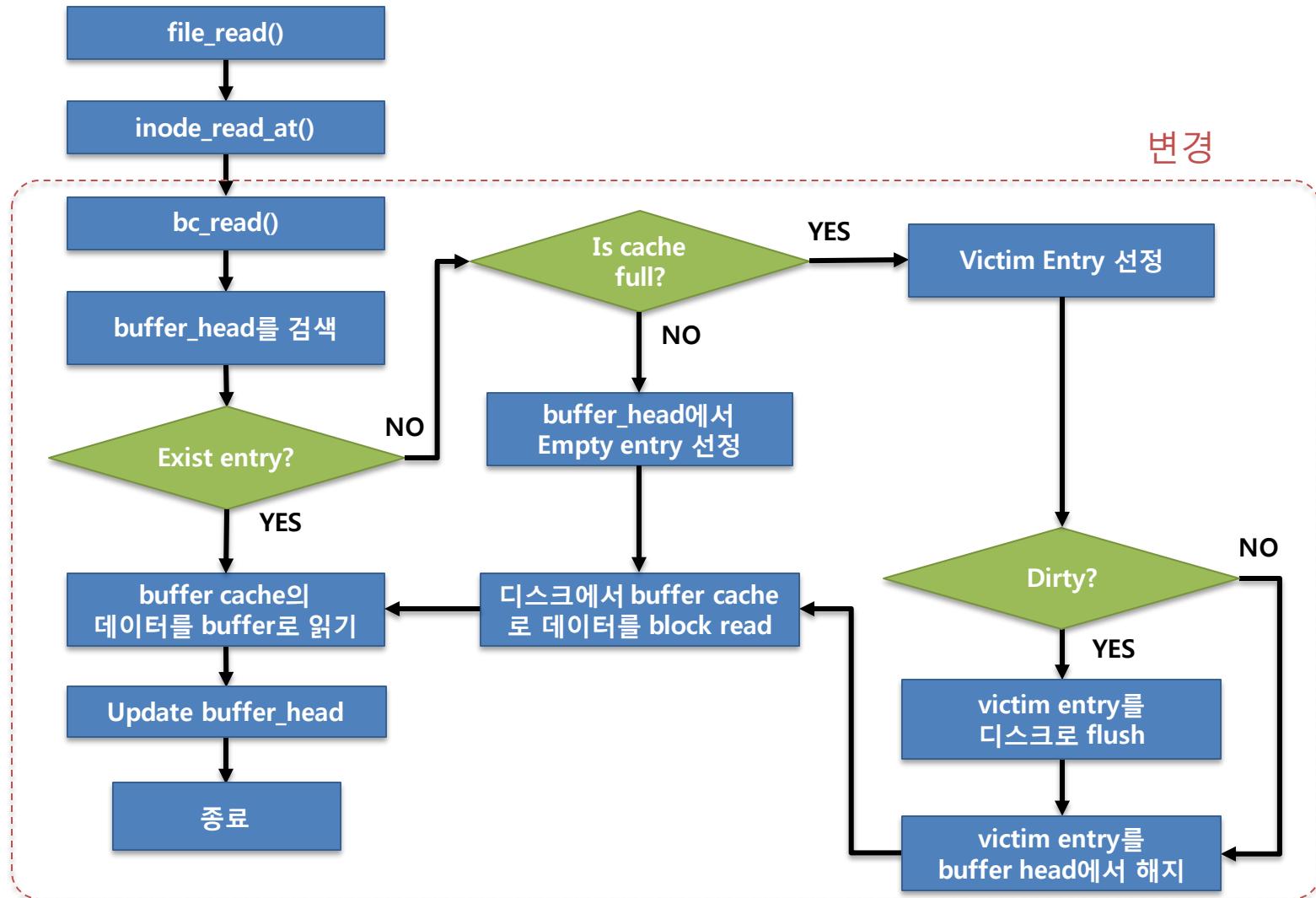
buffer_head 자료구조 추가

- ▣ struct buffer_head: buffer cache의 각 entry를 관리
 - ◆ 구현 위치: pintos/src/filesys/buffer_cache.h
 - ◆ 포함 내용
 - 해당 entry가 dirty인지를 나타내는 flag
 - 해당 entry의 사용 여부를 나타내는 flag
 - 해당 entry의 disk sector 주소
 - clock algorithm을 위한 clock bit
 - lock 변수 (struct lock)
 - buffer cache entry를 가리키기 위한 데이터 포인터

Buffer cache 전역변수 추가

- ▣ 추가 위치: `pintos/src/filesys/buffer_cache.c`
 - ◆ `BUFFER_CACHE_ENTRY_NB` : buffer cache entry의 개수 (32kb)
 - ◆ `*p_buffer_cache` : buffer cache 메모리 영역을 가리킴
 - ◆ `buffer_head[]`: buffer head 배열
 - ◆ `clock_hand` : victim entry 선정 시 clock 알고리즘을 위한 변수

Buffer cache를 추가한 Read



변경

Buffer cache read 구현

bc_read

- ◆ Buffer cache에서 데이터를 읽어 유저 buffer에 저장
- ◆ 읽을 데이터가 buffer cache에 없으면, 디스크에서 읽어 buffer cache에 캐싱
- ◆ memcpy()를 이용해 buffer cache 데이터를 유저 buffer에 복사
- ◆ buffer_head 갱신

pintos/src/filesys/buffer_cache.c

```
bool bc_read (block_sector_t sector_idx, void *buffer, off_t
              bytes_read, int chunk_size, int sector_ofs)
{
    /* sector_idx를 buffer_head에서 검색 (bc_lookup 함수 이용) */
    /* 검색 결과가 없을 경우, 디스크 블록을 캐싱 할 buffer entry의
       buffer_head를 구함 (bc_select_victim 함수 이용) */
    /* block_read 함수를 이용해, 디스크 블록 데이터를 buffer cache
       로 read */
    /* memcpy 함수를 통해, buffer에 디스크 블록 데이터를 복사 */
    /* buffer_head의 clock bit을 setting */
}
```

디스크 read를 buffer cache read로 수정

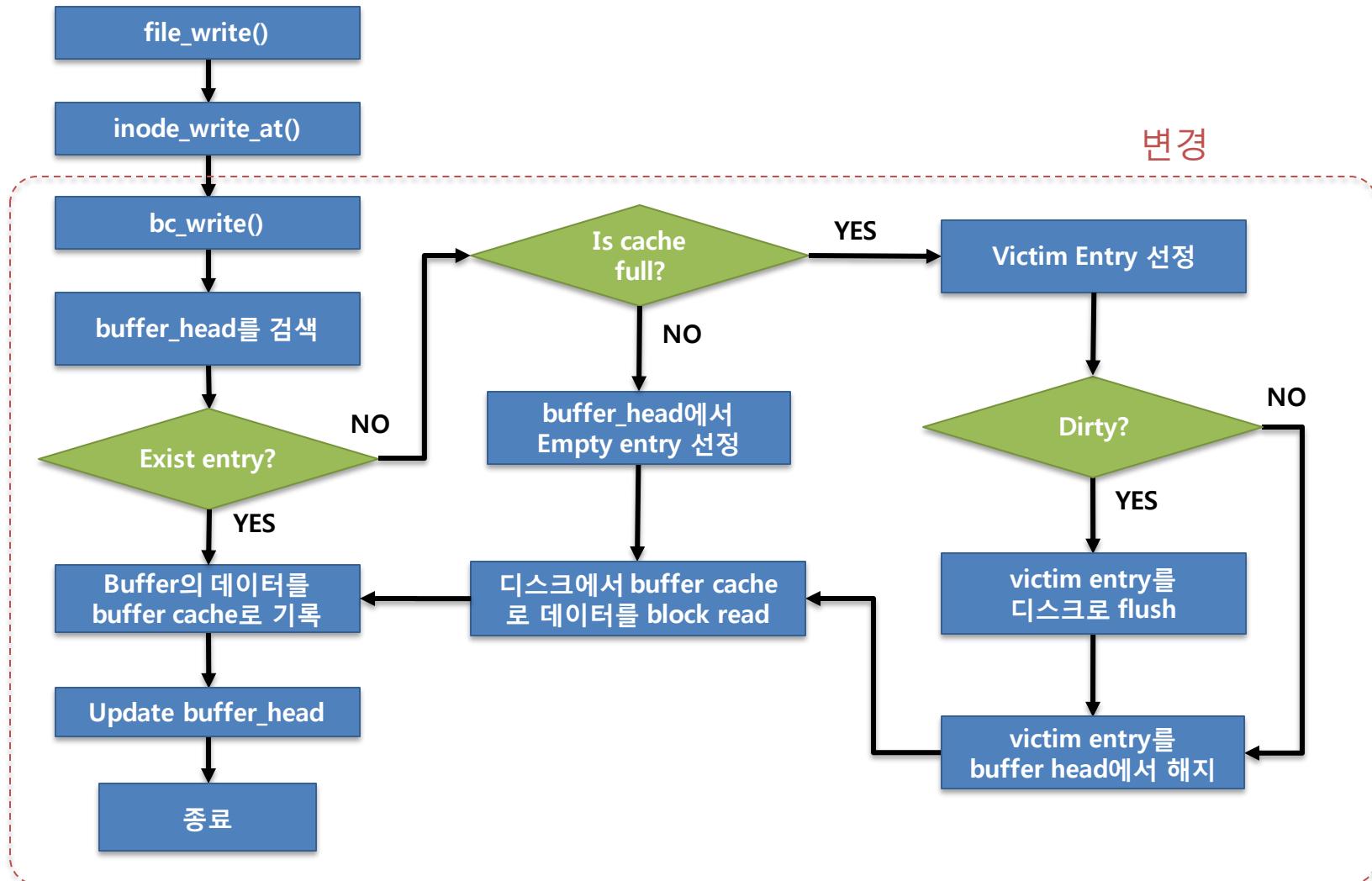
- 파일 read 시, buffer cache에서 데이터를 읽어오도록 수정

pintos/src/filesys/inode.c

```
off_t inode_read_at (struct inode *inode, void *buffer_,  
                      off_t size, off_t offset) {  
    ...  
    while( size > 0 ) {  
        block_sector_t sector_idx = byte_to_sector (inode,  
offset);  
        int sector_ofs = offset % BLOCK_SECTOR_SIZE;  
        ...  
        block_read (fs_device, sector_idx, buffer + bytes_read);  
        ...  
    }  
    ...  
}
```

수정

Buffer cache를 추가한 Write



Buffer cache Write 함수 구현

bc_write

- ◆ buffer의 데이터를 buffer cache에 기록
- ◆ buffer cache 에 빈 entry가 없으면 victim entry 선정하여 디스크에 기록
- ◆ memcpy()를 이용해 buffer의 데이터를 buffer cache에 복사
- ◆ buffer_head를 갱신

pintos/src/filesys/buffer_cache.c

```
bool bc_write (block_sector_t sector_idx, void *buffer, off_t
               bytes_written, int chunk_size, int sector_ofs)
{
    bool success = false;

    /* sector_idx를 buffer_head에서 검색하여 buffer에 복사(구현) */
    /* update buffer head (구현) */

    return success;
}
```

디스크 write를 buffer cache write로 수정

- 파일 write 시, 디스크가 아닌 buffer cache에 데이터를 쓰도록 수정

pintos/src/filesys/inode.c

```
off_t inode_write_at (struct inode *inode, void *buffer_,  
                      off_t size, off_t offset)  
{  
    ...  
    while (size > 0) {  
        ...  
        if (sector_ofs == 0 && chunk_size == BLOCK_SECTOR_SIZE) {  
            /* Write full sector directly to disk. */  
            block_write (fs_device, sector_idx, buffer +  
                         bytes_written);  
        }  
        ...  
    }  
}
```

수정

Buffer cache 초기화 함수 구현

bc_init

- ◆ Buffer cache 영역 할당 및 buffer_head 자료구조 초기화
- ◆ p_buffer_cache 로 buffer cache 영역 포인팅
- ◆ buffer_head 자료구조 초기화

pintos/src/filesys/buffer_cache.c

```
void bc_init(void) {
    /* Allocation buffer cache in Memory */
    /* p_buffer_cache가 buffer cache 영역 포인팅 */
    /* 전역변수 buffer_head 자료구조 초기화 */
}
```

Buffer cache 종료 함수 구현

bc_term

- ◆ Buffer cache에 캐싱 된 데이터를 디스크 블록으로 flush
- ◆ Buffer cache 영역을 메모리에서 할당 해지

pintos/src/filesys/buffer_cache.c

```
void bc_term(void)
{
    /* bc_flush_all_entries 함수를 호출하여 모든 buffer cache
       entry를 디스크로 flush */
    /* buffer cache 영역 할당 해제 */
}
```



Victim Selection 함수 구현

bc_select_victim

- ◆ Clock 알고리즘을 통해, Buffer cache에서 victim entry 선택
- ◆ Victim entry가 dirty일 경우 데이터를 디스크로 flush

pintos/src/filesys/buffer_cache.c

```
struct buffer_head* bc_select_victim (void) {
    /* clock 알고리즘을 사용하여 victim entry를 선택 */
    /* buffer_head 전역변수를 순회하며 clock_bit 변수를 검사 */
    /* 선택된 victim entry가 dirty일 경우, 디스크로 flush */
    /* victim entry에 해당하는 buffer_head 값 update */
    /* victim entry를 return */
}
```

Entry Lookup 함수 구현

bc_lookup

- ◆ buffer_head를 순회하며 디스크 블록의 캐싱 여부 검사
- ◆ 캐싱 되어있다면, buffer cache entry 반환, 없으면 NULL값 반환
- ◆ sector: 검색할 디스크 블록 번호

pintos/src/filesys/buffer_cache.c

```
struct buffer_head* bc_lookup (block_sector_t sector) {
    /* buffer_head를 순회하며, 전달받은 sector 값과 동일한
       sector 값을 갖는 buffer cache entry가 있는지 확인 */
    /* 성공 : 찾은 buffer_head 반환, 실패 : NULL */
}
```

Entry Flush 함수 구현

bc_flush_entry

- ◆ buffer cache 데이터를 디스크로 flush : block_write()

pintos/src/filesys/buffer_cache.c

```
void bc_flush_entry (struct buffer_head *p_flush_entry)
{
    /* block_write을 호출하여, 인자로 전달받은 buffer cache entry
       의 데이터를 디스크로 flush */
    /* buffer_head의 dirty 값 update */
}
```



모든 Entry Flush 함수 구현

bc_flush_all_entries

- ◆ buffer_head를 순회하며 dirty인 entry의 데이터를 디스크로 flush

pintos/src/filesys/buffer_cache.c

```
void bc_flush_all_entries (void){  
    /* 전역변수 buffer_head를 순회하며, dirty인 entry는  
    block_write 함수를 호출하여 디스크로 flush */  
    /* 디스크로 flush한 후, buffer_head의 dirty 값 update */  
}
```

결과

경로 : pintos/src/filesystem

\$make

\$make check

```
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
```

추가 함수

```
bool bc_read (block_sector_t, void*, off_t, int, int);  
/* Buffer cache에서 요청 받은 buffer frame을 읽어옴 */  
  
bool bc_write (block_sector_t, void*, off_t, int, int);  
/* Buffer cache의 buffer frame에 요청 받은 data를 기록 */  
  
struct buffer_head* bc_lookup (block_sector_t);  
/* 버퍼캐시를 순회하며 target sector가 있는지 검색 */  
  
struct buffer_head *bc_select_victim (void);  
/* 버퍼캐시에서 victim을 선정하여 entry head 포인터를 반환 */  
  
void bc_flush_entry (struct buffer_head *);  
/* 인자로 주어진 entry의 dirty비트를 false로 세팅하면서 해당 내역을 disk로 flush */
```



추가 함수 및 자료구조

```
void bc_flush_all_entries (void);  
/* 버퍼캐시를 순회하면서 dirty비트가 true인 entry를 모두 디스크로 flush */  
  
void bc_init (void);  
/* Buffer cache를 초기화하는 함수 */  
  
void bc_term (void);  
/* 모든 dirty entry flush 및 buffer cache 해지 */
```

▣ 추가 자료구조

```
struct buffer_head  
/* buffer cache의 각 entry를 관리 */
```



16. Extensible file

Extensible File 개요

▣ 과제 목표

- ◆ 현재 pintos에서는 파일 생성 시 파일의 크기가 결정되고, 추후 변경이 불가능하다. 이번 과제에서는 파일에 쓰기 동작을 수행할 때에 디스크 블록을 할당 받아 사용하도록 구현한다. 이를 통해 파일 크기가 생성 시에 고정되지 않고, 확장 가능하도록 한다.

▣ 수정 파일

- ◆ pintos/src/filesys/inode.c
- ◆ pintos/src/filesys/inode.h



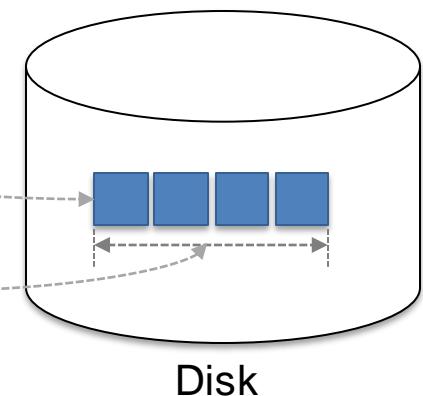
디스크 블록 주소 표현 방식

▣ 파일의 표현

- ◆ start: 시작 블록번호
- ◆ length : 할당 된 블록길이(byte)
 - 파일 생성 시, 디스크 상에 연속된 블록을 할당 받음

pintos/src/filesys/inode.c

```
struct inode_disk
{
    block_sector_t start; /* First data sector */
    off_t length; /* File size in bytes */
    unsigned magic; /* Magic number */
    uint32_t unused[125]; /* Not Used */
}
```



Disk

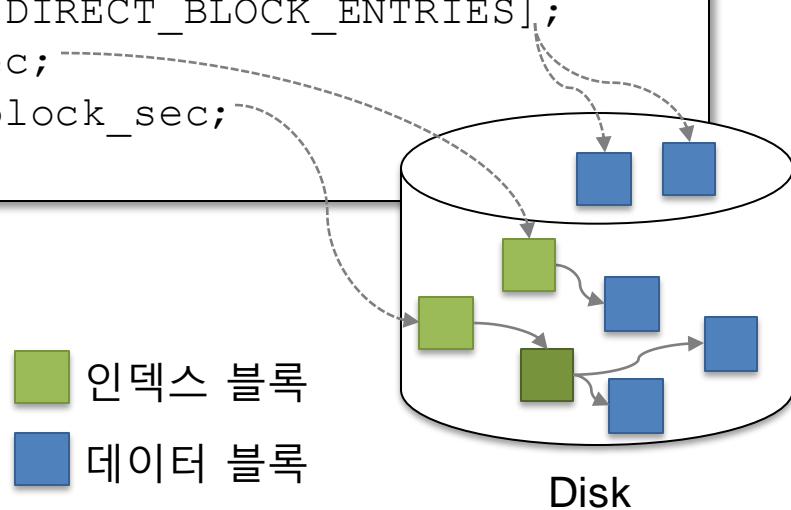
블록번호 표현 방법 변경

파일 구조 변경

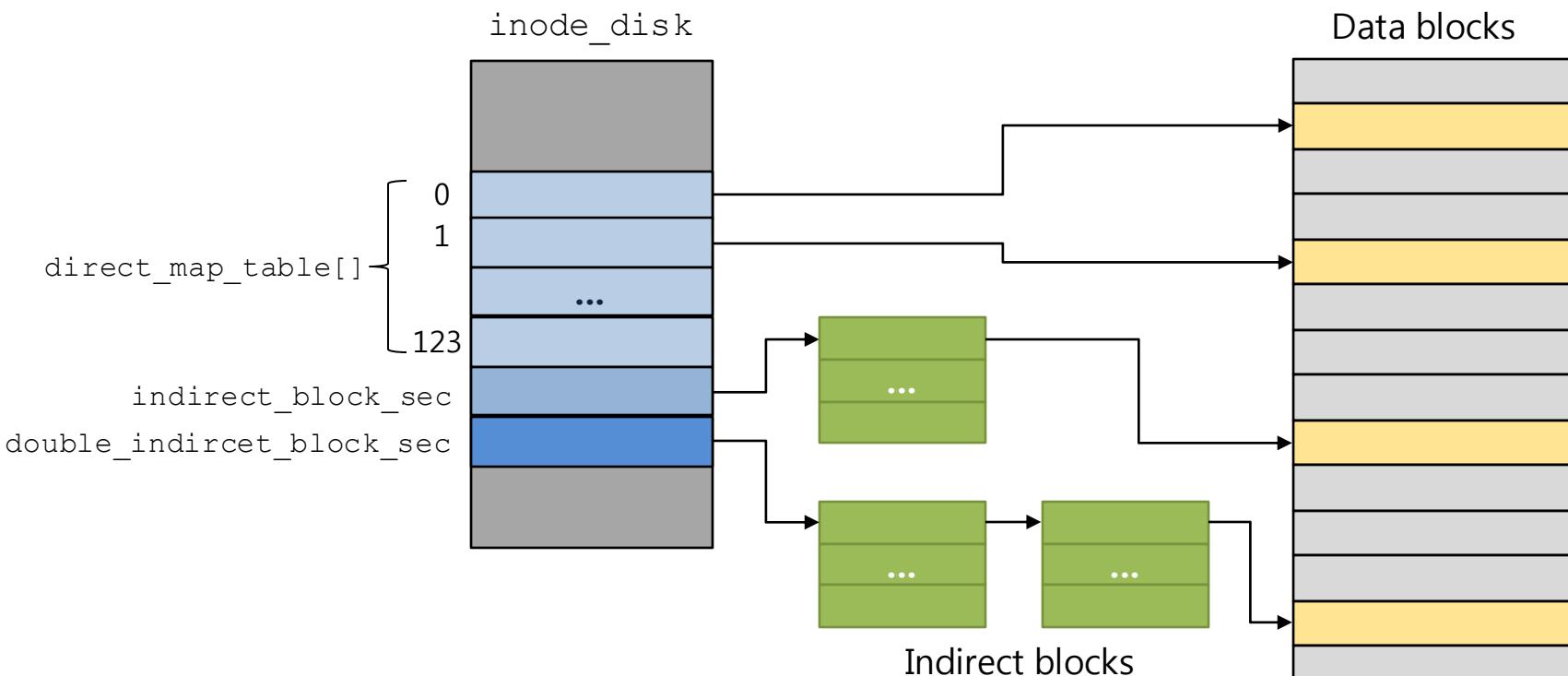
- ◆ 블록 위치를 direct, indirect, double indirect 방식으로 표현
 - inode_disk 자료구조의 크기가 1 블록 크기가 되도록, direct 방식으로 표현할 블록의 수를 결정

pintos/src/filesys/inode.c

```
struct inode_disk{  
    off_t length;           /* File size in bytes */  
    unsigned magic;         /* Magic number */  
    block_sector_t direct_map_table[DIRECT_BLOCK_ENTRIES];  
    block_sector_t indirect_block_sec;  
    block_sector_t double_indirect_block_sec;  
}
```



블록번호 표현 방법 변경 (Cont.)



pintos/src/filesys/inode.c

```
struct inode_disk{
    off_t length;          /* File size in bytes */
    unsigned magic;         /* Magic number */
    block_sector_t direct_map_table[DIRECT_BLOCK_ENTRIES];
    block_sector_t indirect_block_sec;
    block_sector_t double_indirect_block_sec;
}
```

전역 변수 추가

▣ INDIRECT_BLOCK_ENTRIES

- ◆ 추가 위치: `pintos/src/filesys/inode.c`
- ◆ 하나의 인덱스 블록이 저장할 수 있는 다음 인덱스 블록(또는 데이터 블록) 번호의 개수로 값을 할당

▣ DIRECT_BLOCK_ENTRIES

- ◆ 추가 위치: `pintos/src/filesys/inode.c`
- ◆ inode에 direct 방식으로 저장할 블록번호의 개수
- ◆ `inode_disk` 자료구조의 크기가 1 블록 크기(512Byte)가 되도록 선언



enum direct_t 변수 추가

- ▣ inode가 디스크 블록의 번호를 가리키는 방식들을 열거
 - ◆ 추가 위치: pintos/src/filesys/inode.c
 - ◆ NORMAL_DIRECT: inode에 디스크 블록 번호를 저장
 - ◆ INDIRECT: 1개의 인덱스 블록을 통해 디스크 블록 번호에 접근
 - ◆ DOUBLE_INDIRECT: 2개의 인덱스 블록을 통해 디스크 블록 번호에 접근
 - ◆ OUT_LIMIT: 잘못된 파일 오프셋 값일 경우

sector_location 자료구조 추가

- ▣ 블록 주소 접근 방식과, 인덱스 블록내의 오프셋 값을 저장
 - ◆ 추가 위치: `pintos/src/filesys/inode.c`
 - ◆ directness: 디스크 블록 접근 방법(Direct, Indirect, or Double indirect)
 - ◆ index1: 첫 번째 index block에서 접근할 entry의 offset
 - ◆ index2: 두 번째 index block에서 접근할 entry의 offset

inode_indirect_block 자료구조 추가

- ▣ 인덱스 블록을 표현하는 자료구조

- ◆ 추가 위치: `pintos/src/filesys/inode.c`
- ◆ `block_sector_t` 자료형의 배열인 `map_table` 변수를 가짐
 - `INDIRECT_BLOCK_ENTRIES`를 사용하여 선언



파일 표현 방식 수정

- inode_disk를 수정하여, 인덱스 방식으로 블록 번호를 표현

pintos/src/filesys/inode.c

```
struct inode_disk
{
    block_sector_t start; /* First data sector */
    off_t length;          /* File size in bytes */
    unsigned magic;         /* Magic number */
    uint32_t unused[125];  /* Not Used */
}
```

제거

제거

- inode_disk에 추가할 변수

- direct_map_table: Direct 방식으로 접근할 디스크 블록의 번호들이 저장된 배열
- indirect_block_sec: Indirect 방식으로 접근할 인덱스 블록의 번호
- double_indirect_block_sec: Double indirect 방식으로 접근할 경우, 1차 인덱스 블록의 번호

struct inode 자료구조 수정

- inode 자료구조에서 data 변수 제거
- inode 접근 시 획득하는 세마포어 락(extend_lock) 추가

pintos/src/filesys/inode.c

```
struct inode {  
    struct list_elem elem;          /*Element in inode list.*/
    block_sector_t sector;          /*Sector number of disk location.*/
    int open_cnt;                  /*Number of openers.*/
    bool removed;                  /*True if deleted, false otherwise.*/
    int deny_write_cnt;            /*0: writes ok, >0: deny writes.*/
    struct lock extend_lock;        /*Semaphore lock */
    struct inode_disk data;        /*Inode content. */ 제거  
}
```

On-disk inode 획득 함수 구현

- inode를 버퍼캐시로 부터 읽어 전달

pintos/src/filesys/inode.c

```
static bool get_disk_inode(const struct inode *inode, struct
inode_disk *inode_disk)
{
    /* inode->sector에 해당하는 on-disk inode를 buffer cache에서
       읽어 inode_disk에 저장 (bc_read() 함수 사용) */
    /* true 반환 */
}
```

- ◆ inode->sector: inode가 저장된 디스크 블록 번호
- ◆ bc_read(): buffer cache로 부터, 디스크 블록 번호에 해당하는 데이터를 읽어오는 함수

인덱스 블록 내 오프셋 계산함수 구현

▣ locate_byte

- ◆ 디스크 블록 접근 방법(Direct, Indirect, or Double indirect)을 확인
- ◆ 1차 인덱스 블록 내의 오프셋, 2차 인덱스 블록 내의 오프셋을 확인

pintos/src/filesys/inode.c

```
static void locate_byte (off_t pos, struct sector_location
*sec_loc)
{
    off_t pos_sector = pos / BLOCK_SECTOR_SIZE;

    /* Direct 방식일 경우 */
    if (pos_sector < DIRECT_BLOCK_ENTRIES) {
        //sec_loc 자료구조의 변수 값 업데이트 (구현)
    }
    ...
}
```

인덱스 블록 내 오프셋 계산함수 구현 (Cont.)

pintos/src/filesys/inode.c – locate_byte() 계속

```
...
/* Indirect 방식일 경우 */
else if(pos_sector<(off_t)(DIRECT_BLOCK_ENTRIES +
    INDIRECT_BLOCK_ENTRIES)
{
    //sec_loc 자료구조의 변수 값 업데이트(구현)
}
/* Double Indirect 방식일 경우 */
else if (pos_sector <(off_t)(DIRECT_BLOCK_ENTRIES +
    INDIRECT_BLOCK_ENTRIES * (INDIRECT_BLOCK_ENTRIES + 1)))
{
    //sec_loc 자료구조의 변수 값 업데이트(구현)
}
/* 잘못된 파일 오프셋일 경우 */
else
    sec_loc->directness = OUT_LIMIT;
}
```

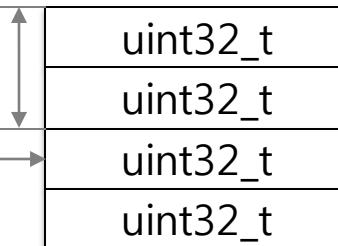
오프셋 계산 함수 구현

▣ 오프셋을 byte 단위로 변환

예시)

offset = 2

return 8byte



인덱스 블록

pintos/src/filesys/inode.c

```
static inline off_t map_table_offset (int index)
{
    /* byte 단위로 변환한 오프셋 값 return */
}
```

inode 업데이트 함수 구현

▣ register_sector

- ◆ 새로 할당 받은 디스크 블록의 번호를 inode_disk에 업데이트

pintos/src/filesys/inode.c

```
static bool register_sector (struct inode_disk *inode_disk,
                            block_sector_t new_sector, struct sector_location sec_loc)
{
    ...
    switch (sec_loc.directness)
    {
        case NORMAL_DIRECT:
            /* inode_disk에 새로 할당받은 디스크 번호 업데이트 */
            break;
        ...
    }
}
```

- ◆ inode_disk: 업데이트할 inode
- ◆ new_sector: 새로 할당 받은 디스크 블록 번호
- ◆ sec_loc: 접근할 인덱스 블록내의 오프셋 값을 나타내는 자료구조

inode 업데이트 함수 구현 (Cont.)

pintos/src/filesys/inode.c – register_sector() 계속

```
    case INDIRECT:
        new_block = malloc (BLOCK_SECTOR_SIZE);
        if (new_block == NULL)
            return false;
        /* 인덱스 블록에 새로 할당 받은 블록 번호 저장 */
        /* 인덱스 블록을 buffer cache에 기록 */
        break;
    case DOUBLE INDIRECT:
        new_block = malloc (BLOCK_SECTOR_SIZE);
        if (new_block == NULL)
            return false;
        /* 2차 인덱스 블록에 새로 할당 받은 블록 주소 저장 후,
         각 인덱스 블록을 buffer cache에 기록 */
        break;
    default:
        return false;
    }
    free(new_block);
    return true;
}
```

파일 오프셋으로 블록 번호 검색 구현

▣ byte_to_sector

- ◆ 파일 오프셋으로 on-disk inode를 검색하여 디스크 블록 번호를 반환

pintos/src/filesys/inode.c

```
static block_sector_t byte_to_sector (const struct inode_disk
*inode_disk, off_t pos) {
    block_sector_t result_sec; // 반환할 디스크 블록 번호

    if (pos < inode_disk->length) {
        struct inode_indirect_block *ind_block;
        struct sector_location sec_loc;
        locate_byte(pos, &sec_loc); // 인덱스 블록 offset 계산

        switch (sec_loc.directness) {
            /* Direct 방식일 경우 */
            case NORMAL_DIRECT:
                /* on-disk inode의 direct_map_table에서
                   디스크 블록 번호를 얻음 */
                break;
        }
    }
}
```



파일 오프셋으로 블록 번호 검색 구현 (Cont.)

pintos/src/filesys/inode.c – byte_to_sector() 계속

```
...
/* Indirect 방식일 경우 */
case INDIRECT:
    ind_block = (struct inode_indirect_block *) malloc
(BLOCK_SECTOR_SIZE);

    if (ind_block) {
        /* buffer cache에서 인덱스 블록을 읽어 옴 */
        /* 인덱스 블록에서 디스크 블록 번호 확인 */
    }
    else
        result_sec = 0;

    free (ind_block);
break;
...
```



파일 오프셋으로 블록 번호 검색 구현 (Cont.)

pintos/src/filesys/inode.c – byte_to_sector() 계속

```
/* Double indirect 방식일 경우 */
case DOUBLE_INDIRECT:
    ind_block = (struct inode_indirect_block *) malloc
(BLOCK_SECTOR_SIZE);
    if (ind_block) {
        /* 1차 인덱스 블록을 buffer cache에서 읽음 */
        /* 2차 인덱스 블록을 buffer cache에서 읽음 */
        /* 2차 인덱스 블록에서 디스크 블록 번호 확인 */
    }
    ...
}
...
return result_sec;
}
```



파일 크기 업데이트 함수 구현

▣ inode_update_file_length

- ◆ 파일 오프셋이 기존 파일 크기 보다 클 경우, 새로운 디스크 블록 할당 및 inode 업데이트

pintos/src/filesys/inode.c

```
bool inode_update_file_length(struct inode_disk* inode_disk,
off_t start_pos, off_t end_pos) {
    ...
    /* 블록 단위로 loop을 수행하며 새로운 디스크 블록 할당 */
    while(size > 0) {
        /* 디스크 블록 내 오프셋 계산 */
        int sector_ofs = offset % BLOCK_SECTOR_SIZE;
        if(sector_ofs > 0) {
            /* 블록 오프셋이 0보다 클 경우, 이미 할당된 블록 */
        }
        ...
    }
}
```

- ◆ start_pos: 증가시켜야 하는 파일 영역의 시작 오프셋 (byte 단위)
- ◆ end_pos: 증가시켜야 하는 파일 영역의 마지막 오프셋 (byte 단위)

파일 크기 업데이트 함수 구현 (Cont.)

pintos/src/filesys/inode.c – inode_update_file_length() 계속

```
...
else{
    /* 새로운 디스크 블록을 할당 */
    if(free_map_allocate(1, &sector_idx)){
        /* inode_disk에 새로 할당 받은 디스크 블록 번호 업데이트
 */
    }
    else{
        free(zeroes);
        return false;
    }
    /* 새로운 디스크 블록을 0으로 초기화 */
    bc_write(sector_idx, zeroes, 0, BLOCK_SECTOR_SIZE, 0);
}
/* Advance. */
size -= chunk_size;
offset += chunk_size;
}
free(zeroes);
return true;
}
```



디스크 블록 할당 해지 함수 구현

▣ free_inode_sectors

- ◆ 파일에 할당된 모든 디스크 블록의 할당 해지

pintos/src/filesys/inode.c

```
static void free_inode_sectors (struct inode_disk *inode_disk) {
    ...
    /* Double indirect 방식으로 할당된 블록 해지 */
    if (inode_disk->double_indirect_block_sec > 0) {

        /* 1차 인덱스 블록을 buffer cache에서 읽음 */
        i = 0;
        /* 1차 인덱스 블록을 통해 2차 인덱스 블록을 차례로 접근 */
        while (ind_block_1->map_table[i] > 0) {
            ...
        }
    }
}
```



디스크 블록 할당 해지 함수 구현 (Cont.)

pintos/src/filesys/inode.c – free_inode_sectors() 계속

```
...
/* 2차 인덱스 블록을 buffer cache에서 읽음 */
j = 0;
/* 2차 인덱스 블록에 저장된 디스크 블록 번호를 접근 */
while (ind_block_2->map_table[j] > 0) {
    /* free_map 업데이트를 통해 디스크 블록 할당 해지 */
    j++;
}
/* 2차 인덱스 블록 할당 해지 */
i++;
}
/* 1차 인덱스 블록 할당 해지 */
...
}
```



디스크 블록 할당 해지 함수 구현 (Cont.)

pintos/src/filesys/inode.c – free_inode_sectors() 계속

```
/* Indirect 방식으로 할당된 디스크 블록 해지 */
if(inode_disk->indirect_block_sec > 0) {
    ...
    /* 인덱스 블록을 buffer cache에서 읽음 */
    /* 인덱스 블록에 저장된 디스크 블록 번호를 접근 */
    while(ind_block->map_table[i] > 0) {
        /* free_map 업데이트를 통해 디스크 블록 할당 해지 */
        i++;
    }
    ...
    /* Direct 방식으로 할당된 디스크 블록 해지 */
    while (inode_disk->direct_map_table[i] > 0) {
        /* free_map 업데이트를 통해 디스크 블록 할당 해지 */
        i++;
    }
}
```



inode_create() 함수 수정

pintos/src/filesys/inode.c

```
bool inode_create (block_sector_t sector, off_t length) {
    ...
    disk_inode = calloc (1, sizeof *disk_inode);
    if (disk_inode != NULL) {
        disk_inode->length = length;
        disk_inode->magic = INODE_MAGIC;
        if (length > 0)
            /* length 만큼의 디스크 블록을 inode_update_file_length()를
               호출하여 할당 */
        }
        /* on-disk inode를 bc_write()를 통해 buffer cache에 기록 */
        /* 할당받은 disk_inode 변수 해제 */
        /* success 변수 update */
    }
    return success;
}
```

추가 (기존코드 제거)

- ◆ sector: inode를 저장할 디스크 블록 번호
- ◆ length: 파일의 크기 (byte 단위)

inode_open() 함수 수정

- inode 자료구조 초기화 시, lock 변수 초기화 부분 추가
- inode 자료구조의 data 변수에 inode를 읽어오는 부분 제거

pintos/src/filesys/inode.c

```
struct inode* inode_open(block_sector_t sector) {  
    ...  
    /* inode의 extend_lock 변수 초기화 (lock_init() 사용) */ 추가  
    // block_read(fs_device, inode->sector, &inode->data); 제거  
    return success;  
}
```

inode_read_at() 함수 수정

- byte_to_sector() 호출 시, 인자로 in-memory inode가 아닌 on-disk inode를 전달 하도록 수정

pintos/src/filesys/inode.c

```
off_t inode_read_at (struct inode *inode, void *buffer_, off_t
                      size, off_t offset) {
    ...
    /* inode_disk 자료형의 disk_inode 변수를 동적 할당 */
    /* on-disk inode를 buffer cache에서 읽어옴 (get_disk_inode()
       이용) */
    while (size > 0) {
        ...
        block_sector_t sector_idx=byte_to_sector(inode, offset);
        ...
    }
    ...
    return bytes_read;
}
```

추가

수정

inode_write_at() 함수 수정

▣ 유저 데이터를 buffer cache에 쓰는 함수

- ◆ byte_to_sector() 호출 시, 인자로 in-memory inode가 아닌 on-disk inode를 전달 하도록 수정
- ◆ 쓰기 연산으로 인해 파일 길이가 증가할 시, on-disk 자료구조의 length 변수를 업데이트하는 부분 추가

pintos/src/filesys/inode.c

```
off_t inode_write_at (struct inode *inode, const void *buffer_,  
                      off_t size, off_t offset) {  
    ...  
    /* inode_disk 자료형의 disk_inode 변수를 동적 할당 */  
    if(disk_inode == NULL)  
        return 0;  
    /* on-disk inode를 buffer cache에서 읽어옴 (get_disk_inode()  
     이용) */  
    ...  
    추가  
}
```

inode_write_at() 함수 수정 (Cont.)

pintos/src/filesys/inode.c - inode_write_at() 계속

```
...
/* inode의 lock 획득 */

int old_length = disk_inode->length;
int write_end = offset + size - 1;

if (write_end > old_length - 1) {
    /* 파일길이가 증가하였을 경우, on-disk inode 업데이트 */
}

/* inode의 lock 해제 */

while (size > 0) {
    ...
}
```

추가



inode_write_at() 함수 수정 (Cont.)

▣ byte_to_sector() 함수 호출 인자 변경

- ◆ in-memory inode 대신 on-disk inode 자료구조를 전달 받도록 수정
- ◆ 함수 종료 전, 수정한 disk_inode 자료구조를 buffer cache에 기록하는 부분 추가

pintos/src/filesys/inode.c - inode_write_at() 계속

```
...
while (size > 0) {
    ...
    block_sector_t sector_idx=byte_to_sector(inode, offset); 수정
    ...
}

/* 수정한 disk_inode 자료구조를 buffer cache에 기록
   (bc_write() 함수 이용) */
/* free (bounce); 제거 */

추가
return bytes_written;
}
```

inode_close() 함수 수정

- in-memory inode, on-disk inode 할당 해지
 - 할당 받았던 디스크 블록을 반환하도록 free_inode_sectors() 함수 호출 부분 추가

pintos/src/filesys/inode.c

```
void inode_close (struct inode *inode) {
    ...
    /* Release resources if this was the last opener. */
    if (--inode->open_cnt == 0) {
        ...
        /* Deallocate blocks if removed. */
        if (inode->removed) {
            /* inode의 on-disk inode 획득(get_disk_inode() 이용) */
            /* 디스크 블록 반환(free_inode_sectors() 이용) */
            /* on-disk inode 반환 (free_map_release() 이용) */
            /* disk_inode 변수 할당 해제 (free() 이용) */
        }
    }
    ...
}
```

구현 (기존 코드 제거)

결과

경로 : pintos/src/filesystem

\$make

\$make check

```
pass tests/filesys/extended/grow-create
pass tests/filesys/extended/grow-dir-lg
pass tests/filesys/extended/grow-file-size
pass tests/filesys/extended/grow-root-lg
pass tests/filesys/extended/grow-root-sm
pass tests/filesys/extended/grow-seq-lg
pass tests/filesys/extended/grow-seq-sm
pass tests/filesys/extended/grow-sparse
pass tests/filesys/extended/grow-tell
pass tests/filesys/extended/grow-two-files
pass tests/filesys/extended/syn-rw
```

수정 함수

```
static block_sector_t byte_to_sector(const struct  
    inode_disk *inode_disk, off_t pos)  
/* 파일 오프셋으로 inode를 검색하여 디스크 블록의 번호를 반환 */  
  
bool inode_create(block_sector_t sector, off_t length)  
/* 파일 생성 시, 파일 크기만큼 디스크 블록을 할당 */  
  
struct inode* inode_open(block_sector_t sector)  
/* inode 자료구조를 생성 */  
  
off_t inode_read_at(struct inode *inode, void *buffer_,  
    off_t size, off_t offset)  
/* 디스크에서 요청 받은 데이터를 읽어 유저 buffer에 저장 */
```



수정 함수 (Cont.)

```
off_t inode_write_at(struct inode *inode, const void
                      *buffer_, off_t size, off_t offset)
/* 유저 buffer의 데이터를 디스크에 저장 */

void inode_close(struct inode *inode)
/* inode list 자료구조에서 제거. 할당 받은 디스크 블록 할당 해지 */
```



추가 함수

```
static bool get_disk_inode(const struct inode *inode, struct  
                           inode_disk * inode_disk)  
/* in-memory inode의 on-disk inode를 인자에 저장하여 반환 */  
  
static void locate_byte(off_t pos, struct sector_location,  
                       *sec_loc)  
/* 파일 오프셋을 통해 인덱스 블록의 오프셋 값을 계산 */  
  
static inline off_t map_table_offset(int index)  
/* 인덱스 블록 내의 byte 오프셋 값을 계산 */  
  
static bool register_sector(struct inode_disk* inode_disk,  
                           block_sector_t new_sector, struct sector_location sec_loc)  
/* 새로 할당 받은 디스크 블록의 물리 주소를 inode 자료구조에 갱신 */
```

추가 함수 및 수정 자료구조

```
bool inode_update_file_length(struct inode_disk* inode_disk,
                               off_t start_pos, off_t end_pos)

/* 파일 크기 증가 시, 새로운 디스크 블록 할당 및 inode 업데이트 */

static void free_inode_sectors (struct inode_disk
                                *inode_disk)

/* 파일에 할당된 모든 디스크 블록의 할당 해지 */
```

▣ 수정 자료구조

```
struct inode_disk
    /* On-disk inode */

struct inode
    /* In-memory inode */
```



17. Subdirectory

Subdirectories 개요

▣ 과제목표

- ◆ 현재 pintos에서는 root 디렉터리만 존재하는 단일계층으로 root 디렉터리에만 파일을 생성한다. 본 과제에서는 root 디렉터리내에 파일과 디렉터리를 생성할 수 있도록 계층구조를 구현한다.

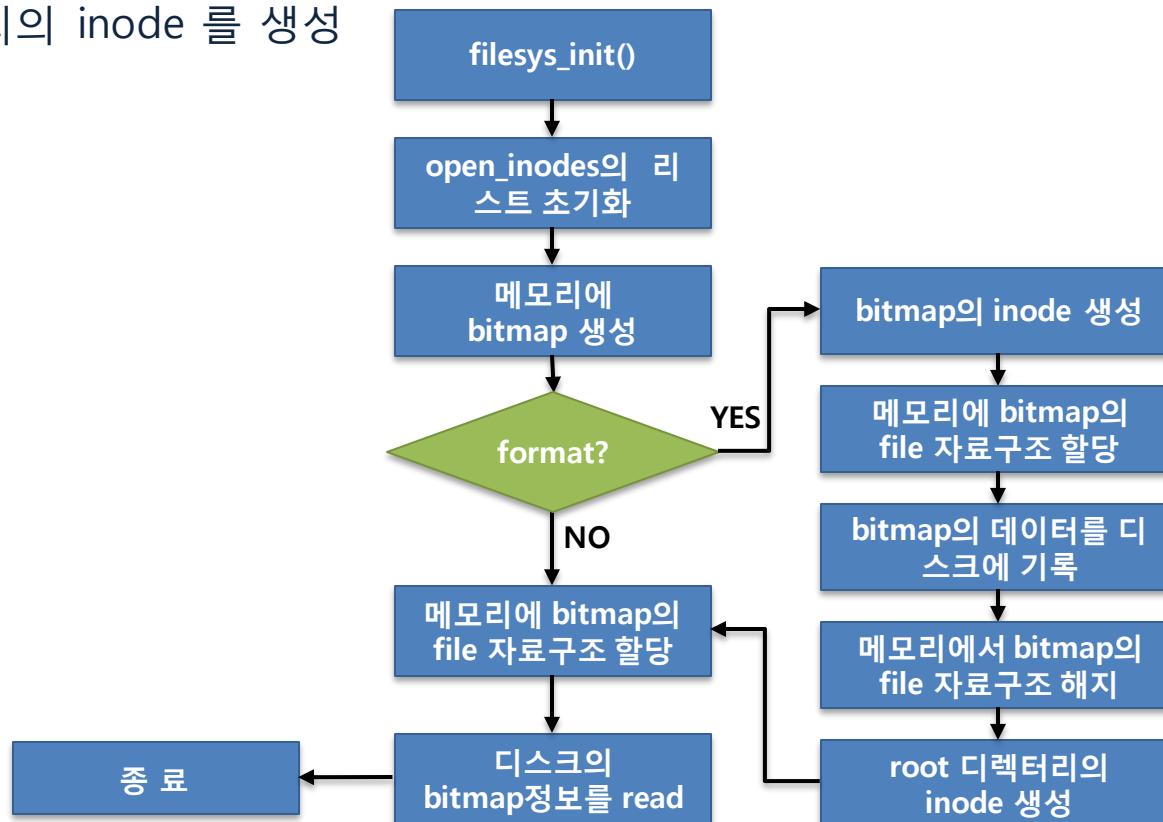
▣ 수정 파일

- ◆ pintos/src/filesys/inode.c
- ◆ pintos/src/filesys/filesys.*
- ◆ pintos/src/filesys/file.*
- ◆ pintos/src/filesys/directory.*
- ◆ pintos/src/userprog/syscall.c

파일 시스템 초기화 (포맷)

▣ 디스크에 filesystem layout 생성

- ◆ Bitmap 생성 및 초기화
- ◆ Bitmap의 inode 생성 및 데이터를 디스크에 기록.
- ◆ Root 디렉터리의 inode 를 생성



파일 시스템 초기화 (Cont.)

▣ Filesystem의 bitmap 생성 및 디스크에 기록

- ◆ fs_device : filesystem device를 포인팅
- ◆ free_map_init() : bitmap 생성 및 초기화
- ◆ do_format()
 - bitmap의 inode 생성 및 디스크에 기록
 - Root 디렉터리의 inode 생성

pintos/src/filesys/filesys.c

```
void filesys_init (bool format) {
    fs_device = block_get_role (BLOCK_FILESYS);
    ...
    inode_init ();
    free_map_init ();
    if (format)
        do_format ();
    free_map_open ();
}
```

파일 시스템 초기화 (Cont.)

▣ block type에 해당하는 블럭 포인터 반환

- ◆ `block_by_role[]` : block 자료구조의 메모리 주소를 가리키는 배열

pintos/src/devices/block.c

```
struct block * block_get_role (enum block_type role)
{
    ASSERT (role < BLOCK_ROLE_CNT);
    return block_by_role[role];
}
```



파일 시스템 초기화 (Cont.)

▣ In-memory inode를 관리하는 list 초기화

- ◆ open_inodes : in-memory inode 전역변수 (Double linked list)

pintos/src/filesys/inode.c

```
static struct list open_inodes;  
  
void inode_init (void)  
{  
    list_init (&open_inodes);  
}
```

```
void list_init (struct list *list)  
{  
    ASSERT (list != NULL);  
    list->head.prev = NULL;  
    list->head.next = &list->tail;  
    list->tail.prev = &list->head;  
    list->tail.next = NULL;  
}
```

파일 시스템 초기화 (Cont.)

▣ In-memory 블록 bitmap 생성 및 초기화

- ◆ `bitmap_create()` : filesystem 크기의 bitmap 생성 후, 각 bit의 값을 false로 초기화
- ◆ `bitmap_mark()`: 전달받은 디스크 블록번호의 bit를 true로 설정

pintos/src/filesys/free-map.c

```
void free_map_init (void) {
    free_map = bitmap_create (block_size (fs_device));
    ...
    /* FREE_MAP_SECTOR = 0, ROOT_DIR_SECTOR = 1 */
    bitmap_mark (free_map, FREE_MAP_SECTOR);
    bitmap_mark (free_map, ROOT_DIR_SECTOR);
}
```

파일 시스템 초기화 (Cont.)

- ▣ 디스크에 free 블록 비트맵 기록 및 root 디렉터리 초기화
 - ◆ free_map_create(): bitmap의 inode를 생성
 - bitmap의 데이터를 디스크에 기록
 - ◆ dir_create() : root 디렉터리의 inode를 생성
 - Root 디렉터리의 파일 개수는 16개로 제한

pintos/src/filesys/filesys.c

```
static void do_format (void) {  
    free_map_create ();  
    if (!dir_create (ROOT_DIR_SECTOR, 16))  
        PANIC ("root directory creation failed");  
    free_map_close ();  
}
```

파일 시스템 초기화 (Cont.)

▣ bitmap의 inode를 생성 및 디스크에 기록

- ◆ `inode_create()` : bitmap의 inode를 0번블록에 생성
- ◆ `bitmap_write()` : bitmap의 데이터를 디스크에 기록

pintos/src/filesys/free_map.c

```
void free_map_create (void) {
    /* Create inode. */
    if (!inode_create (FREE_MAP_SECTOR, bitmap_file_size
(free_map)))
        PANIC ("free map creation failed");
    free_map_file = file_open (inode_open (FREE_MAP_SECTOR));
    ...
    /* Write bitmap to file. */
    if (!bitmap_write (free_map, free_map_file))
}
```

파일 시스템 초기화 (Cont.)

▣ 1번 디스크 블록에 root 디렉터리의 inode를 생성

- ◆ sector : inode의 위치 (디스크 블록번호)
- ◆ entry_cnt : root 디렉터리 엔트리의 최대 개수

pintos/src/filesys/directory.c

```
bool dir_create (block_sector_t sector, size_t entry_cnt)
{
    return inode_create (sector, entry_cnt * sizeof (struct
dir_entry));
}
```



파일 시스템 초기화 (Cont.)

- 루트디렉토리 데이터를 위한 디스크 블록 할당, inode에 시작 블록 번호 저장

pintos/src/filesys/inode.c

```
bool inode_create (block_sector_t sector, off_t length) {
    ...
    size_t sectors = bytes_to_sectors(length);
    if (free_map_allocate (sectors, &disk_inode->start)) {
        block_write(fs_device, sector, disk_inode);
    }
    ...
    return success;
}
```

- bytes_to_sectors(): 인자로 전달 받은 byte길이에 해당하는 블록 개수 반환
- free_map_allocate(): 첫 번째 인자로 전달 받은 크기의 데이터를 저장할 연속된 블록을 찾고, 할당 받은 디스크 블록의 시작 번호를 두 번째 인자에 저장

파일 시스템 초기화 (Cont.)

- Bitmap 기록용 파일의 닫기: in-memory inode를 해지 및 open_inodes list에서 제거

pintos/src/filesys/free-map.c

```
void free_map_close (void)
{
    file_close (free_map_file);
}
```

- ◆ file_close()
 - 인자로 받은 file의 deny_write 변수를 false로 설정
 - open_inodes list에서 해당 file의 inode 제거 및 할당 해지
 - file 자료구조 메모리 할당 해지

파일 시스템 초기화 (Cont.)

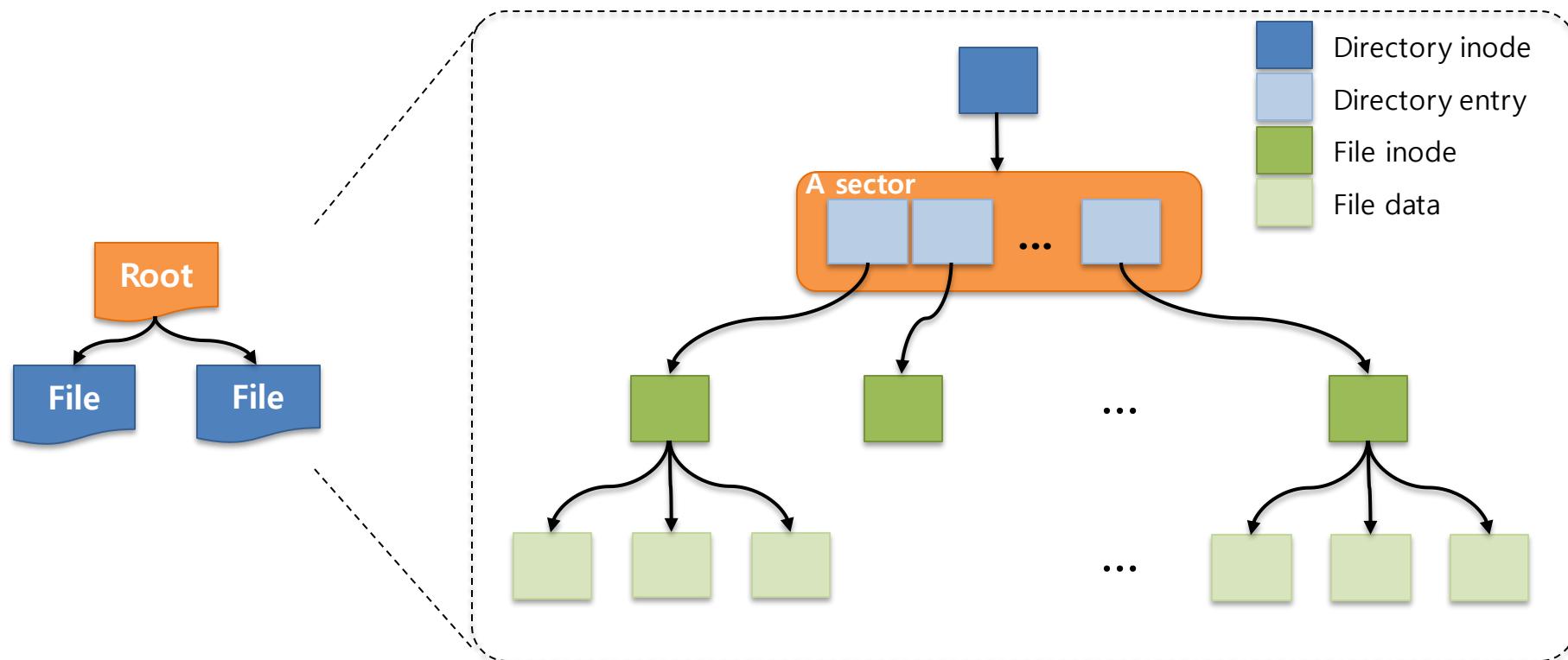
- 디스크에 저장되어 있는 bitmap 정보를 읽어들임

pintos/src/filesys/free-map.c

```
void free_map_open (void) {
    free_map_file = file_open (inode_open (FREE_MAP_SECTOR));
    if (free_map_file == NULL)
        PANIC ("can't open free map");
    if (!bitmap_read (free_map, free_map_file))
        PANIC ("can't read free map");
}
```

- file_open() : 메모리에 file 자료구조 할당 후, 타겟 파일의 inode 포인팅 및 초기화
- bitmap_read() : 디스크에 기록된 bitmap의 데이터를 읽기

디렉터리 구조



파일 생성

- Root 디렉터리에 파일을 생성
- 디렉토리 계층 구조를 구현
 - '..', '.' 기능의 file 구현
 - 절대, 상대 경로 기능 구현 ('/' 구분자에 의해 구분)

pintos/src/filesys/filesys.c

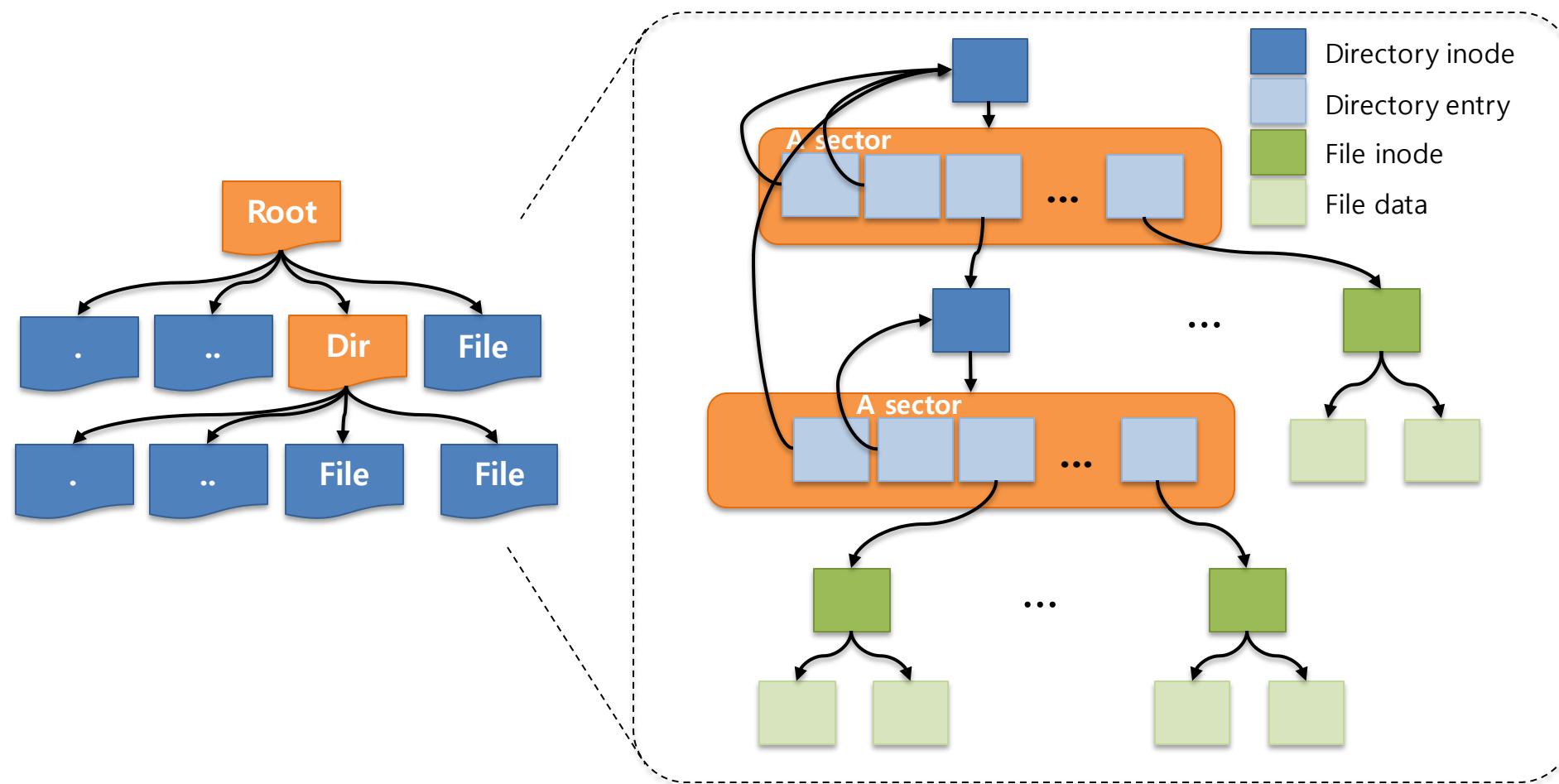
```
bool filesys_create (const char *name, off_t
                      initial_size)
{
    block_sector_t inode_sector = 0;
    struct dir *dir = dir_open_root(); 수정
    ...
}
```



Requirement

- ▣ 디렉토리 계층 구조 구현 (현재 single directory)
 - ◆ 디렉토리 엔트리구조를 파일과 다른 디렉토리들을 포인팅하도록 수정
 - ◆ '.', '..' 기능 구현
- ▣ 스레드의 현재 작업디렉터리 정보 추가
 - ◆ 절대, 상대 경로 기능 구현 ('/' 구분자에 의해 디렉토리 구분)
- ▣ create, open, remove 시스템콜 수정
- ▣ 디렉터리 관련 시스템콜 추가

Subdirectories 계층 구조 구현



On-disk inode(inode_disk) 자료구조 수정

- ▣ 현재 핀토스는 루트디렉토리만 존재함.
- ▣ 디렉터리와 파일을 구별하기 위한 변수 추가
 - ◆ struct On-disk inode
 - 자료구조 위치: pintos/src/filesys/directory.c
 - 파일 (=0), 디렉터리 (=1)로 구분

inode 초기화 함수 수정

▣ inode_create

- ◆ 인터페이스에 파일, 디렉터리를 구분하는 인자를 추가

pintos/src/filesys/inode.c

```
bool inode_create (block_sector_t sector, off_t length,
                   uint32_t is_dir) {추가  
...  
disk_inode->length = length;  
disk_inode->magic = INODE_MAGIC;  
/* inode 생성 시, struct inode_disk에 추가한 파일, 디렉터리 구분  
을 위한 필드를 is_dir인자 값으로 설정 */  
...  
return success;  
}
```

inode 초기화 함수 수정 (Cont.)

- ▣ 다음 함수에서 `inode_create()` 함수 호출 시, 변경된 인터페이스에 맞게 `is_dir` 값을 설정
 - ◆ `dir_create()` (위치: `pintos/src/filesys/directory.c`)
 - 디렉터리의 on-disk inode 생성시 `is_dir`값을 1로 설정
 - ◆ `filesys_create()` (위치: `pintos/src/filesys/filesys.c`)
 - 파일의 on-disk inode 생성시 `is_dir`값을 0으로 설정
 - ◆ `free_map_create()` (위치: `pintos/src/filesys/free-amp.c`)
 - free-map의 on-disk inode 생성시 `is_dir`값을 0으로 설정



thread 자료구조 수정

- ▣ thread가 작업중인 디렉터리의 정보를 저장하도록 변수 추가
 - ◆ struct thread
 - 자료구조 위치: pintos/src/threads/thread.h
 - 디렉터리의 정보를 나타내는 struct dir 사용



thread의 현재 작업 디렉터리 초기화

- thread 자료구조 초기화 시 추가한 필드를 초기화

pintos/src/threads/thread.c

```
void thread_init (void)
{
    ASSERT (intr_get_level () == INTR_OFF);
    lock_init (&tid_lock);
    list_init (&ready_list);
    list_init (&all_list);
    ...
    /* struct thread에서 추가한 필드를 NULL로 초기화 */
}
```



thread의 현재 작업 디렉터리 설정

- Filesystem 초기화 후, 스레드의 현재 작업 디렉토리를 root 디렉터리로 설정
 - dir_open_root() : root 디렉터리의 정보를 반환

pintos/src/filesys/filesys.c

```
void filesys_init (bool format) {  
    ...  
    if (format)  
        do_format ();  
    free_map_open ();  
    /* struct thread에서 추가한 필드를 root 디렉터리로 설정 */  
}
```

thread 생성 함수 수정

- 스레드 생성시, 자식 스레드의 작업 디렉터리를 부모 스레드의 작업 디렉터리의 위치로 저장

pintos/src/threads/thread.c

```
tid_t thread_create (const char *name, int priority,
                      thread_func *function, void *aux) {
    ...
    init_thread (t, name, priority);
    tid = t->tid = allocate_tid ();
    if (thread_current ()->cur_dir != NULL) { 추가
        /* 자식 스레드의 작업 디렉터리를 부모 스레드의 작업 디렉터리로 디
         *렉터리를 다시 오픈하여 설정 */
    }
    ...
}
```

프로세스 종료 함수 수정

- ▣ 프로세스의 리소스를 해지 시, 스레드의 cur_dir을 해지

pintos/src/userprog/process.c

```
void process_exit (void)
{
    struct thread *cur = thread_current ();
    ...
    sema_up (&cur -> cp -> sema_exit);
    /* 스래드의 현재작업 디렉터리의 디렉터리 정보를 메모리에서 해지*/
    pd = cur->pagedir;
    ...
}
```

inode_is_dir 함수 구현

▣ inode_is_dir

- ◆ inode_disk 자료구조를 메모리에 할당
- ◆ In-memory inode의 on-disk inode를 읽어 inode_disk에 저장
- ◆ On-disk inode의 is_dir을 result에 저장하여 반환

pintos/src/filesys/inode.c

```
bool inode_is_dir (const struct inode *inode) {  
    bool result;  
  
    /* inode_disk 자료구조를 메모리에 할당 */  
    /* in-memory inode의 on-disk inode를 읽어 inode_disk에 저장 */  
    /* on-disk inode의 is_dir을 result에 저장하여 반환 */  
    return result;  
}
```

isdir 시스템콜 구현

▣ sys_isdir

- ◆ fd 리스트에서 fd에 대한 file 정보를 얻어옴
- ◆ fd의 in-memory inode가 디렉터리 인지 판단하여 성공여부 반환



경로 분석 함수 구현

▣ parse_path

- ◆ return : path_name을 분석하여 작업하는 디렉터리의 정보 포인터를 반환
- ◆ *file_name : path_name을 분석하여 파일, 디렉터리의 이름을 포인팅
- ◆ path_name의 시작이 '/'의 여부에 따라 절대, 상대경로 구분하여 디렉터리 정보를 dir에 저장
- ◆ strtok_r() 함수를 이용하여 path_name의 디렉터리 정보와 파일 이름 저장
- ◆ file_name에 파일 이름 저장
- ◆ dir로 오픈된 디렉터리를 포인팅



경로 분석 함수 구현 (Cont.)

pintos/src/filesys/filesys.c

```
struct dir* parse_path (char *path_name, char *file_name) {  
    struct dir *dir;  
  
    if (path_name == NULL || file_name == NULL)  
        goto fail;  
  
    if (strlen(path_name) == 0)  
        return NULL;  
  
    /* PATH_NAME의 절대/상대경로에 따른 디렉터리 정보 저장 (구현) */  
    char *token, *nextToken, *savePtr;  
  
    token = strtok_r (path_name, "/", &savePtr);  
    nextToken = strtok_r (NULL, "/", &savePtr);  
  
    ...
```



경로 분석 함수 구현 (Cont.)

pintos/src/filesys/filesys.c – parse_path() 계속

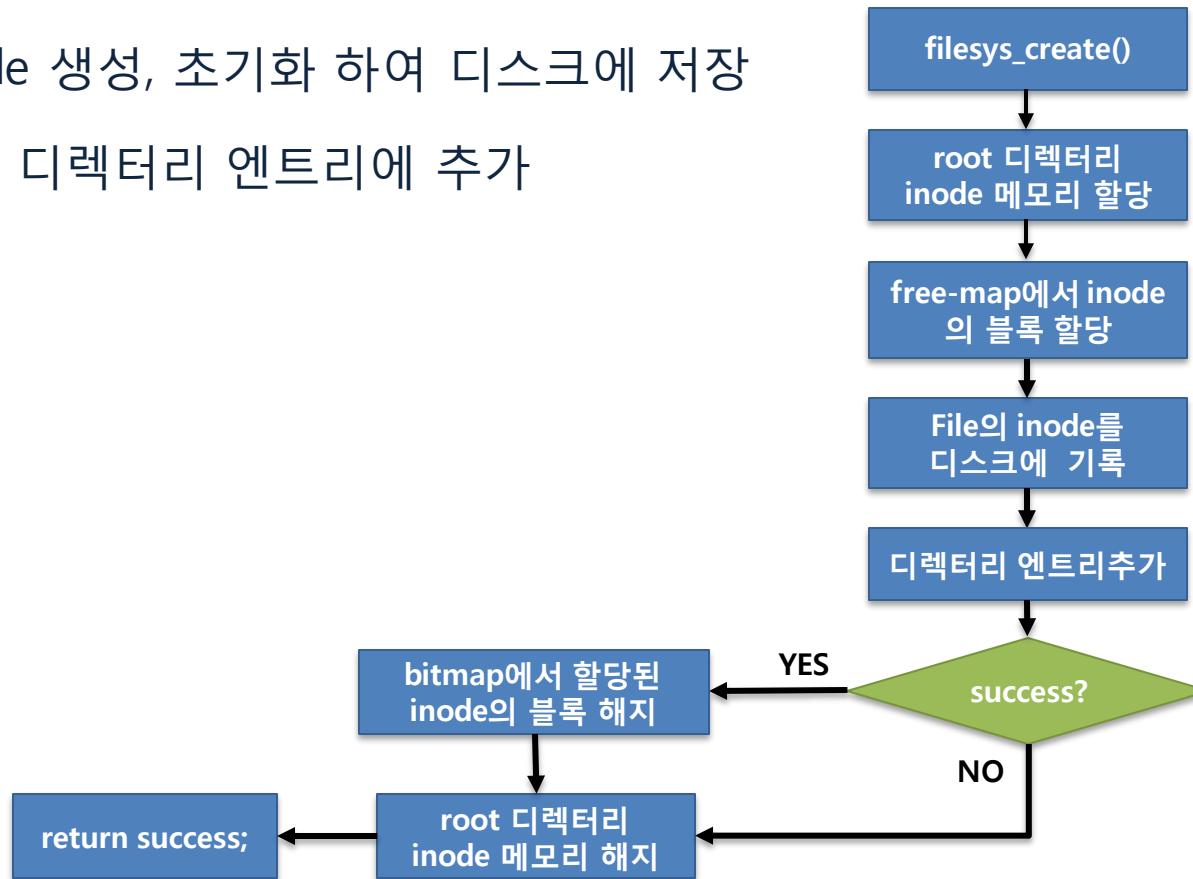
```
while (token != NULL && nextToken != NULL) {  
    /* dir에서 token이름의 파일을 검색하여 inode의 정보를 저장*/  
    /* inode가 파일일 경우 NULL 반환 */  
    /* dir의 디렉터리 정보를 메모리에서 해지 */  
    /* inode의 디렉터리 정보를 dir에 저장 */  
    /* token에 검색할 경로 이름 저장 */  
}  
  
/* token의 파일 이름을 file_name에 저장  
/* dir 정보 반환 */  
}
```



File create

▣ filesys_create()

- ◆ System call 'create()'이 호출
- ◆ inode 생성, 초기화 하여 디스크에 저장
- ◆ root 디렉터리 엔트리에 추가



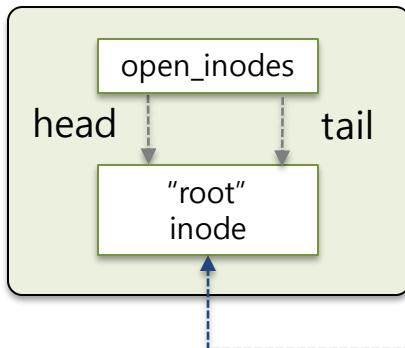
File create (Cont.)

ex) 'testfile' create

```
bool filesys_create ("testfile")
```

Main Memory

inode list



3. root 디렉터리
entries read

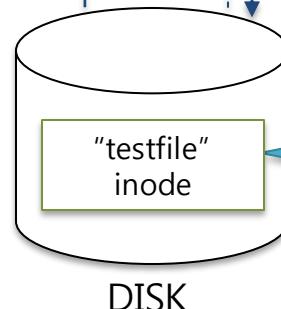
root directory entries

inode = 7	...
testfile	...
in_use = 1	...

4. 'testfile'의
디렉터리 엔트리 추가

1. root 디렉터리
inode read

5. root 디렉터리
entries write



2. 디스크에 'testfile'의
inode 기록

File create (Cont.)

▣ filesys_create

- ◆ inode 생성 및 초기화, root 디렉터리 엔트리에 파일명과 inode 블록 번호 추가

pintos/src/filesys/filesys.c

```
bool filesys_create (const char *name, off_t initial_size) {  
    block_sector_t inode_sector = 0;  
    struct dir *dir = dir_open_root();  
    bool success = (dir != NULL  
        && free_map_allocate (1, &inode_sector))  
    ...
```

- ◆ dir_open_root() : 메모리에 root 디렉터리 inode 자료구조 생성
- ◆ free_map_allocate() : free 블럭의 할당 및 할당된 주소의 저장
 - 할당받은 블록의 시작 번호를 두 번째 인자에 저장

File create (Cont.)

pintos/src/filesys/filesys.c – filesys_create() 계속

```
    && inode_create (inode_sector, initial_size)  
    && dir_add (dir, name, inode_sector));  
  
...  
  
return success;  
}
```

- ◆ inode_create() : 데이터 블럭을 할당 받고, inode에 위치 저장, 해당 inode 디스크 저장
- ◆ dir_add(): 디렉터리 내에 파일 이름의 존재여부 검사 후, 디렉터리 엔트리 추가

File create (Cont.)

- ▣ 메모리에 root 디렉터리 자료구조 할당

pintos/src/filesys/directory.c

```
struct dir * dir_open_root (void)
{
    return dir_open (inode_open (ROOT_DIR_SECTOR));
}
```

- ◆ inode_open() : 파일에 해당하는 inode에 대한 자료구조 포인터 return
 - open되지 않았을 경우, 먼저 해당 inode를 open함
- ◆ dir_open() : dir 자료구조의 메모리 할당 및 초기화

File create (Cont.)

- inode_open: 섹터주소에 존재하는 inode의 메모리 위치를 return

pintos/src/filesys/inode.c

```
struct inode * inode_open (block_sector_t sector) {  
    ...  
    /* open_inodes 리스트에 inode가 존재하는지 검사 */  
    for (e = list_begin (&open_inodes); e != list_end  
&open_inodes; e = list_next (e)) {  
        inode = list_entry (e, struct inode, elem);  
        if (inode->sector == sector) {  
            inode_reopen (inode);  
            return inode;  
        }  
    }  
    ...
```

File create (Cont.)

pintos/src/filesys/inode.c – inode_open() 계속

```
...
/* inode 자료구조 메모리 할당 */
inode = malloc (sizeof *inode);
...

/* inode 자료구조 초기화 */
list_push_front (&open_inodes, &inode->elem);
inode->sector = sector;
inode->open_cnt = 1;
inode->deny_write_cnt = 0;
inode->removed = false;
block_read (fs_device, inode->sector, &inode->data);

return inode;
}
```

File create (Cont.)

- ▣ dir 자료구조를 메모리에 할당 후, inode 포인팅과 오프셋 설정

pintos/src/filesys/directory.c

```
struct dir * dir_open (struct inode *inode) {  
    struct dir *dir = calloc (1, sizeof *dir);  
  
    if (inode != NULL && dir != NULL) {  
        dir->inode = inode;  
        dir->pos = 0;  
        return dir;  
    }  
    ...  
}
```

File create (Cont.)

pintos/src/filesys/free-map.c

```
bool free_map_allocate (size_t cnt, block_sector_t *sectorp) {
    block_sector_t sector = bitmap_scan_and_flip (free_map, 0,
cnt, false);

    ...
    if (sector != BITMAP_ERROR)
        *sectorp = sector;
    return sector != BITMAP_ERROR;
}
```

- ◆ free-map에서 할당할 블록을 first-fit 방식으로 검색
- ◆ cnt : 할당하고자 하는 블록 개수
- ◆ sectorp : 할당 받은 블록의 시작 번호
- ◆ bitmap_scan_and_flip() : bitmap에서 할당할 연속된 블록을 찾고, 할당한 블록의 bitmap을 true로 설정

File create (Cont.)

- ▣ 디렉터리 내에 파일 이름의 존재여부 검사 후, 디렉터리 엔트리에 추가
 - ◆ 디렉터리 엔트리 갱신 및 디스크에 기록

pintos/src/filesys/directory.c

```
bool dir_add (struct dir *dir, const char *name,
block_sector_t inode_sector) {
    struct dir_entry e;
    off_t ofs;
    bool success = false;

    ...
    /* Check that NAME is not in use. */
    if (lookup (dir, name, NULL, NULL))
        goto done;

    /* 사용 중이지 않은 디렉터리 엔트리 검색 */
    for (ofs = 0; inode_read_at (dir->inode, &e, sizeof e,
ofs) == sizeof e; ofs += sizeof e)
        if (!e.in_use)
            break;

    ...
}
```

File create (Cont.)

pintos/src/filesys/directory.c – dir_add() 계속

```
...
/* Write slot. */

e.in_use = true;
strlcpy (e.name, name, sizeof e.name);
e.inode_sector = inode_sector;
success = inode_write_at (dir->inode, &e, sizeof e, ofs)
== sizeof e;

done:

return success;

}
```



File create (Cont.)

▣ lookup

- ◆ 디렉토리에 주어진 파일명이 존재하는지 검색
- ◆ 검색된 디렉토리 엔트리의 주소를 반환(인자로)

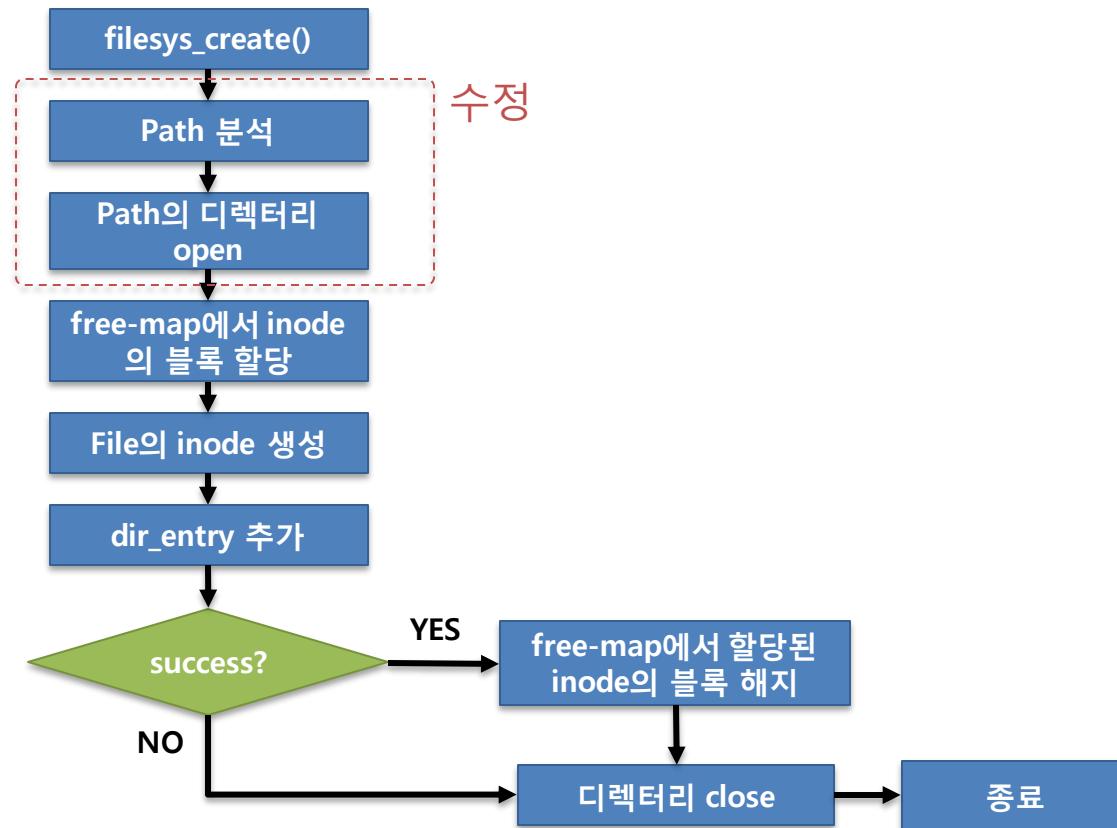
pintos/src/filesys/directory.c

```
static bool lookup (const struct dir *dir, const char *name,
                   struct dir_entry *ep, off_t *ofsp) {

    struct dir_entry e;
    size_t ofs;
    ...
    for (ofs = 0; inode_read_at (dir->inode, &e, sizeof e, ofs) ==  
         sizeof e; ofs += sizeof e)
        if (e.in_use && !strcmp (name, e.name)) {
            if (ep != NULL)
                *ep = e;
            if (ofsp != NULL)
                *ofsp = ofs;
            return true;
        }
    return false;
}
```

File create 알고리즘 수정

- 절대 경로 및 상대 경로 분석과정 추가



filesys_create 함수 수정

▣ filesys_create

- ◆ root 디렉터리에 파일 생성을 name 경로에 파일 생성하도록 변경
 - 생성하고자 하는 파일경로인 name을 cp_name에 복사
 - 파일 생성시 절대, 상대경로를 분석하여 디렉토리에 파일을 생성하도록 수정
 - parse_path()를 추가하여 cp_name의 경로 분석
 - return : 파일을 생성하고자 하는 디렉터리의 위치
 - path_name : 생성하고자 하는 파일의 경로
 - file_name : 생성하고자 하는 파일의 이름 저장
 - ◆ On-disk inode 생성시 is_dir 값을 파일(=0)로 수정
 - ◆ 디렉터리 엔트리에 file_name의 엔트리를 추가하도록 수정

filesys_create 함수 수정 (Cont.)

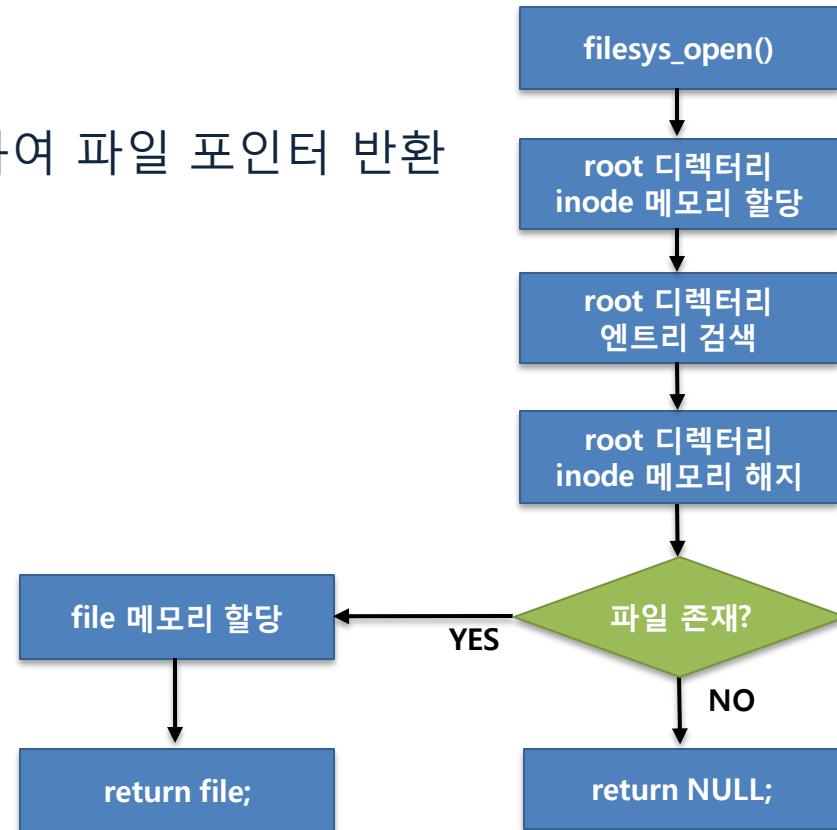
pintos/src/filesys/filesys.c

```
bool filesys_create (const char *name, off_t initial_size) {  
    block_sector_t inode_sector = 0;  
    /* name의 파일경로를 cp_name에 복사 */  
    /* cp_name의 경로 분석 */  
    struct dir *dir = dir_open_root(); 수정  
    bool success = (dir != NULL  
                    && free_map_allocate (1, &inode_sector))  
    /* inode의 is_dir값 설정 */ 수정  
        && inode_create (inode_sector, initial_size)  
    /* 추가되는 디렉터리 엔트리의 이름을 file_name으로 수정 */  
        && dir_add (dir, name, inode_sector));  
    ...  
    return success;  
}
```

File open

▣ filesys_open()

- ◆ System call 'open()'이 호출
- ◆ inode list에 inode 추가
- ◆ file 자료구조 생성 및 초기화하여 파일 포인터 반환



File open (Cont.)

ex) 'testfile' open

```
struct file * filesys_open ("testfile")
```

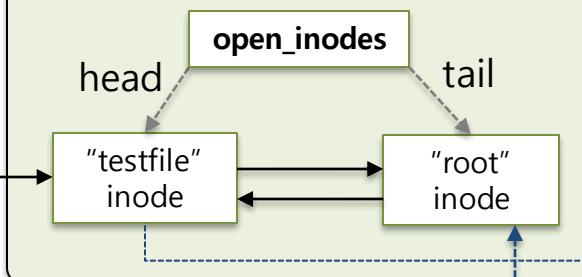
Main Memory

4. file 자료구조 생성

```
struct file {  
    struct inode*  
    ...  
};
```

5. inode 주소 저장
& file 주소값 반환

inode list

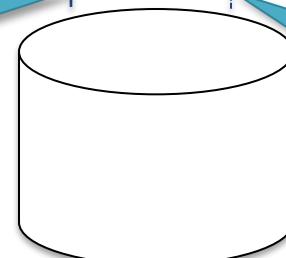


root directory entries

inode = 7	...
testfile	...
in_use = 1	...

3. 'testfile'의 inode를
inode list에 삽입

1. root 디렉터리
inode read



2. root 디렉터리
entries read

File open (Cont.)

▣ filesys_open

- ◆ file 자료구조 할당 및 초기화하여 file 포인터 반환

pintos/src/filesys/filesys.c

```
struct file * filesys_open (const char *name) {
    struct dir *dir = dir_open_root ();
    struct inode *inode = NULL;
    if (dir != NULL)
        dir_lookup (dir, name, &inode);
    dir_close (dir);
    return file_open (inode);
}
```

- ◆ dir_open_root() : root 디렉터리 inode를 open_inodes 리스트에 추가
- ◆ dir_lookup() : 디렉터리 엔트리를 검색, 파일의 inode 를 open_inodes 리스트에 추가
- ◆ file_open() : 메모리에 file 자료구조 할당 및 초기화

File open (Cont.)

▣ dir_lookup

- ◆ 디렉터리 엔트리에서 파일을 검색하여, inode를 open하고 성공 여부 반환

pintos/src/filesys/directory.c

```
bool dir_lookup (const struct dir *dir, const char *name,
                 struct inode **inode) {
    struct dir_entry e;
    ...
    if (lookup (dir, name, &e, NULL))
        *inode = inode_open (e.inode_sector);
    else
        *inode = NULL;
    return *inode != NULL;
}
```

- ◆ lookup() : 디렉터리 엔트리를 디스크에서 읽어 파일 이름을 검색 후, 해당 엔트리를 3번째 인자에 저장

File open (Cont.)

▣ file_open

- ◆ 메모리에 file 자료구조 할당 및 초기화, 파일 포인터 반환

pintos/src/filesys/file.c

```
struct file * file_open (struct inode *inode) {  
    /* file 자료구조 메모리 할당 */  
    struct file *file = calloc (1, sizeof *file);  
    /* file 자료구조 초기화 */  
    if (inode != NULL && file != NULL) {  
        file->inode = inode;  
        file->pos = 0;  
        file->deny_write = false;  
        return file;  
    ...  
}
```

open 시스템 콜 수정

▣ filesys_open

- ◆ root 디렉터리에 파일 생성을 name 경로에 파일 생성하도록 변경
 - 생성하고자 하는 파일경로인 name을 cp_name에 복사
 - 파일 생성시 절대, 상대경로를 분석하여 디렉토리에 파일을 생성하도록 수정
 - parse_path()를 추가하여 cp_name의 경로 분석
 - return : 파일을 생성하고자 하는 디렉터리의 위치
 - path_name : 생성하고자 하는 파일의 경로
 - file_name : 생성하고자 하는 파일의 이름 저장
 - ◆ 디렉터리 엔트리에서 file_name을 검색하도록 수정

open 시스템 콜 수정

pintos/src/filesys/filesys.c

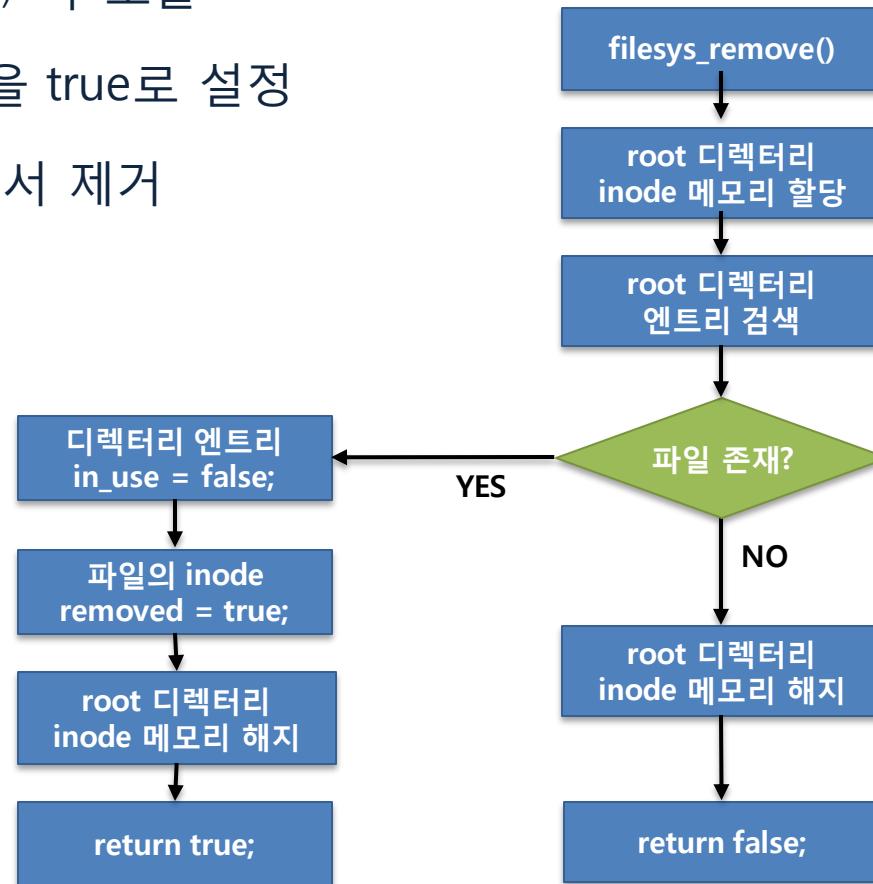
```
struct file* filesys_open (const char *name) {  
    struct dir *dir = dir_open_root(); } 수정  
...  
if (dir != NULL) 수정  
    dir_lookup (dir, name, &inode);  
dir_close (dir);  
...  
return file_open (inode);  
}
```



File remove

▣ filesys_remove()

- ◆ System call 'remove()' 가 호출
- ◆ inode의 removed 값을 true로 설정
- ◆ root 디렉터리 entry에서 제거



File remove (Cont.)

ex) 'testfile' remove

```
bool filesys_remove ("testfile")
```

Main Memory

inode list

4. removed
값 true로 set

```
head → open_inodes  
"testfile" inode {  
    bool removed;  
    ...  
}
```

```
"root" inode{...}
```

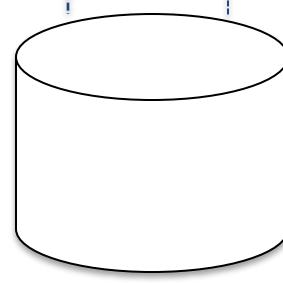
root directory entries

inode = 7	...
testfile	...
in_use = 0	...

3. 'testfile' entry의
in_use 값 false로 set

1. root 디렉터리
inode read

2. root 디렉터리
entries read



DISK

File remove (Cont.)

▣ filesys_remove

- ◆ 디렉터리 엔트리에서 타겟 파일 엔트리 제거, in-memory inode의 removed 값을 true로 설정

pintos/src/filesys/filesys.c

```
bool filesys_remove (const char *name) {  
    struct dir *dir = dir_open_root ();  
    bool success = dir != NULL && dir_remove (dir, name);  
    dir_close (dir);  
    return success;  
}
```

File remove (Cont.)

▣ dir_remove

- ◆ 디렉터리 엔트리에서 타겟 파일 엔트리 제거
- ◆ In-memory inode의 removed 변수를 true로 설정
- ◆ 업데이트 된 디렉터리 엔트리를 디스크에 기록

pintos/src/filesys/directory.c

```
bool dir_remove (struct dir *dir, const char *name) {  
    ...  
    /* Find directory entry. */  
    if (!lookup (dir, name, &e, &ofs))  
        goto done;  
    ...  
    /* Erase directory entry. */  
    e.in_use = false;  
    if (inode_write_at (dir->inode, &e, sizeof e, ofs) !=  
        sizeof e)  
        goto done;  
    ...
```

File remove (Cont.)

pintos/src/filesys/directory.c – dir_remove() 계속

```
...
/* Open inode. */
inode = inode_open (e.inode_sector);
...
/* Remove inode. */
inode_remove (inode);
success = true;
done:
    inode_close (inode);
    return success;
}
```

- ◆ inode_open() : in-memory inode를 open_inodes list에 추가
- ◆ inode_remove() : in-memory inode의 removed값을 true로 설정

remove 시스템 콜 수정

▣ filesys_remove

- ◆ root 디렉터리에 파일 생성을 name 경로에 파일 생성하도록 변경
 - 생성하고자 하는 파일경로인 name을 cp_name에 복사
 - 파일 생성시 절대, 상대경로를 분석하여 디렉토리에 파일을 생성하도록 수정
 - parse_path()를 추가하여 cp_name의 경로 분석
 - return : 파일을 생성하고자 하는 디렉터리의 위치
 - path_name : 생성하고자 하는 파일의 경로
 - file_name : 생성하고자 하는 파일의 이름 저장
 - ◆ 디렉터리 엔트리에서 file_name의 in-memory inode가 파일/디렉터리인지 판단
 - inode가 디렉터리일 경우 디렉터리내 파일 존재 여부 검사
 - 디렉터리내 파일이 존재하지 않을 경우, 디렉터리에서 file_name의 엔트리 삭제
 - inode가 파일일 경우 디렉터리 엔트리에서 file_name 엔트리 삭제



chdir 시스템 콜 구현

▣ sys_chdir

- ◆ dir의 디렉터리 정보를 얻어옴
- ◆ 스레드의 현재 작업 디렉터리의 정보를 메모리에서 해지 후, dir로 현재 작업 디렉터리 변경
 - struct thread -> cur_dir : 스레드의 현재 작업 디렉터리

pintos/src/userprog/syscall.c

```
bool sys_chdir (const char *dir) {  
    /* dir 경로를 분석하여 디렉터리를 반환 */  
    /* 스레드의 현재 작업 디렉터리를 변경 */  
}
```



mkdir 시스템 콜 구현

- ▣ dir 경로에 디렉터리 생성

pintos/src/userprog/syscall.c

```
bool sys_mkdir (const char *dir)
{
    return filesys_create_dir(dir);
}
```



filesys_create_dir 함수 구현

▣ filesys_create_dir

- ◆ Root 디렉터리에 파일 생성을 name 경로에 파일 생성하도록 변경
 - 생성하고자 하는 파일경로인 name을 cp_name에 복사
 - 파일 생성시 절대, 상대경로를 분석하여 디렉토리에 파일을 생성하도록 수정
 - parse_path()를 추가하여 cp_name의 경로 분석
 - return : 파일을 생성하고자 하는 디렉터리의 위치
 - path_name : 생성하고자 하는 파일의 경로
 - file_name : 생성하고자 하는 파일의 이름 저장
 - ◆ bitmap에서 inode sector번호 할당
 - ◆ 할당받은 sector에 file_name의 디렉터리 생성
 - ◆ 디렉터리 엔트리에 file_name의 엔트리 추가
 - ◆ 디렉터리 엔트리에 '.', '..' 파일의 엔트리 추가

filesys_create_dir 함수 구현 (Cont.)

pintos/src/filesys/filesys.c

```
bool filesys_create_dir (const char *name) {  
    /* name 경로 분석 */  
    /* bitmap에서 inode sector번호 할당 */  
    /* 할당받은 sector에 file_name의 디렉터리 생성 */  
    /* 디렉터리 엔트리에 file_name의 엔트리 추가 */  
    /* 디렉터리 엔트리에 '.', '..' 파일의 엔트리 추가 */  
}
```



readdir 시스템 콜 구현

▣ filesys_create_dir

- ◆ fd 리스트에서 fd에 대한 file 정보를 얻어옴
- ◆ fd의 in-memory inode가 디렉터리 인지 판단
 - inode가 파일일 경우 false 반환
- ◆ 디렉터리의 엔트리를 읽어 name변수에 파일이름 저장
 - `bool dir_readdir (struct dir *dir, char name[NAME_MAX + 1])`
 - dir->pos 엔트리를 읽어 name에 파일이름 저장
 - '.', '..'파일 외의 파일을 name 변수에 저장



readdir 시스템 콜 구현

pintos/src/userprog/syscall.c

```
bool sys_readdir (int fd, char *name) {  
    /* fd 리스트에서 fd에 대한 file정보를 얻어옴 */  
    /* fd의 file->inode가 디렉터리인지 검사 */  
    /* p_file을 dir 자료구조로 포인팅 */  
    /* 디렉터리의 엔트에서 "..",.." 이름을 제외한 파일이름을 name에 저장*/  
}
```



inumber 시스템 콜 구현

▣ sys_inumber

- ◆ fd 리스트에서 fd에 대한 file 정보를 얻어옴
- ◆ fd의 on-disk inode 블록 주소를 반환
 - inode_get_inumber() : in-memory inode의 sector 값 반환



파일 시스템 포맷 함수 수정

- 파일 시스템 포맷 시, Root 디렉터리 엔트리에 '.', '..' 파일 추가
 - do_format
 - 자료구조 위치: pintos/src/filesys/filesys.c

디렉터리 삭제 함수 수정

- ▣ '.', '..' 파일을 삭제 시 false를 리턴하도록 수정
 - ◆ dir_remove
 - ◆ 자료구조 위치: pintos/src/filesys/directory.c



결과

경로 : pintos/src/filesystem

\$make

\$make check

```
pass tests/filesys/extended/dir-empty-name
pass tests/filesys/extended/dir-mk-tree
pass tests/filesys/extended/dir-mkdir
pass tests/filesys/extended/dir-open
pass tests/filesys/extended/dir-over-file
pass tests/filesys/extended/dir-rm-cwd
pass tests/filesys/extended/dir-rm-parent
pass tests/filesys/extended/dir-rm-root
pass tests/filesys/extended/dir-rmtree
pass tests/filesys/extended/dir-rmdir
pass tests/filesys/extended/dir-under-file
pass tests/filesys/extended/dir-vine
```

추가 및 수정 함수

```
struct dir* parse_path (char *path_name, char *file_name)
/* 절대, 상대경로를 분석하여 파일 이름과 디렉터리 정보를 저장 */

bool inode_is_dir (const struct inode *inode)
/* in-memory inode의 on-disk inode일 읽어 파일(=0), 디렉터리(=1) 구분 */

bool filesys_create_dir (const char *name)
/* name 경로에 디렉터리 생성 */

void filesys_init(bool format)
/* 파일 시스템의 bitmap 생성 및 디스크에 기록 */

bool filesys_create(const char *name, off_t initial_size)
/* inode 생성 및 초기화, root 디렉터리 엔트리에 파일 추가 */
```

수정 함수

```
struct file* filesys_open(const char *name)
    /* file 자료구조 할당 및 초기화하여 file 포인터 반환 */

bool filesys_remove(const char *name)
    /* 디렉터리 엔트리에서 타겟 파일 엔트리 제거 */

bool dir_create(block_sector_t sector, size_t entry_cnt)
    /* 디렉터리의 inode를 생성 */

bool dir_remove(struct dir *dir, const char *name)
    /* 디렉터리 엔트리에서 타겟 파일 엔트리 제거 */

void free_map_create(void)
    /* bitmap의 inode를 생성 및 디스크에 기록 */
```

수정 함수(Cont.)

```
void thread_init(void)  
/* thread 자료구조 초기화 */  
  
tid_t thread_create (const char *name, int priority,  
                     thread_func *function, void *aux)  
/* thread 생성 */  
  
void process_exit (void)  
/* 프로세스의 자원 해지 */  
  
void do_format (bool format)  
/* bitmap의 inode 생성 및 디스크에 기록, Root 디렉터리의 inode 생성 */  
  
bool inode_create(block_sector_t sector, off_t length)  
/* 파일 생성 시, 파일 크기만큼 디스크 블록을 할당 */
```

수정 자료구조

```
struct inode_disk  
/* On-disk inode */  
  
struct thread  
/* thread 자료구조 */
```

추가 시스템 콜

- 선언 및 구현 위치: pintos/src/userprog/syscall.c

```
bool sys_chdir (const char *dir);
```

- Arg : 디렉토리 경로
- Return : true 성공, false 실패
- 프로세스의 현재 작업 디렉토리를 dir로 변경 (상대, 절대경로 허용)

```
bool sys_mkdir (const char *dir);
```

- Arg : 디렉토리 이름
- Return : true, 성공, false 실패
 - 이미 동일한 이름의 디렉토리 존재시 fail
 - dir의 경로 내의 디렉토리가 존재하지 않는 경우 fail
- dir이름의 디렉토리 생성 (상대, 절대경로 허용)

추가 시스템 콜 (Cont.)

- 선언 및 구현 위치: `pintos/src/userprog/syscall.c`

```
bool sys_readdir (int fd, char *name);
```

- Arg : 디렉터리의 file descriptor, 읽어들인 디렉토리 엔트리의 파일 이름
- Return : true 엔트리 존재, false 엔트리가 존재하지 않음
- fd로부터 하나의 디렉토리 엔트리를 읽어 name에 파일 이름 저장

```
bool sys_isdir (int fd);
```

- Arg : file descriptor
- Return : true 디렉토리, false 파일
- Inode의 디렉토리 여부 판단



추가 시스템 콜 (Cont.)

- 선언 및 구현 위치: `pintos/src/userprog/syscall.c`

```
int sys_inumber (int fd);
```

- Arg : file descriptor
- Return : inode sector number
- `fd`와 관련된 파일 또는 디렉터리의 inode number를 반환

