

Operating Systems Lab

Part 0: Installing PintOS



Youjip Won

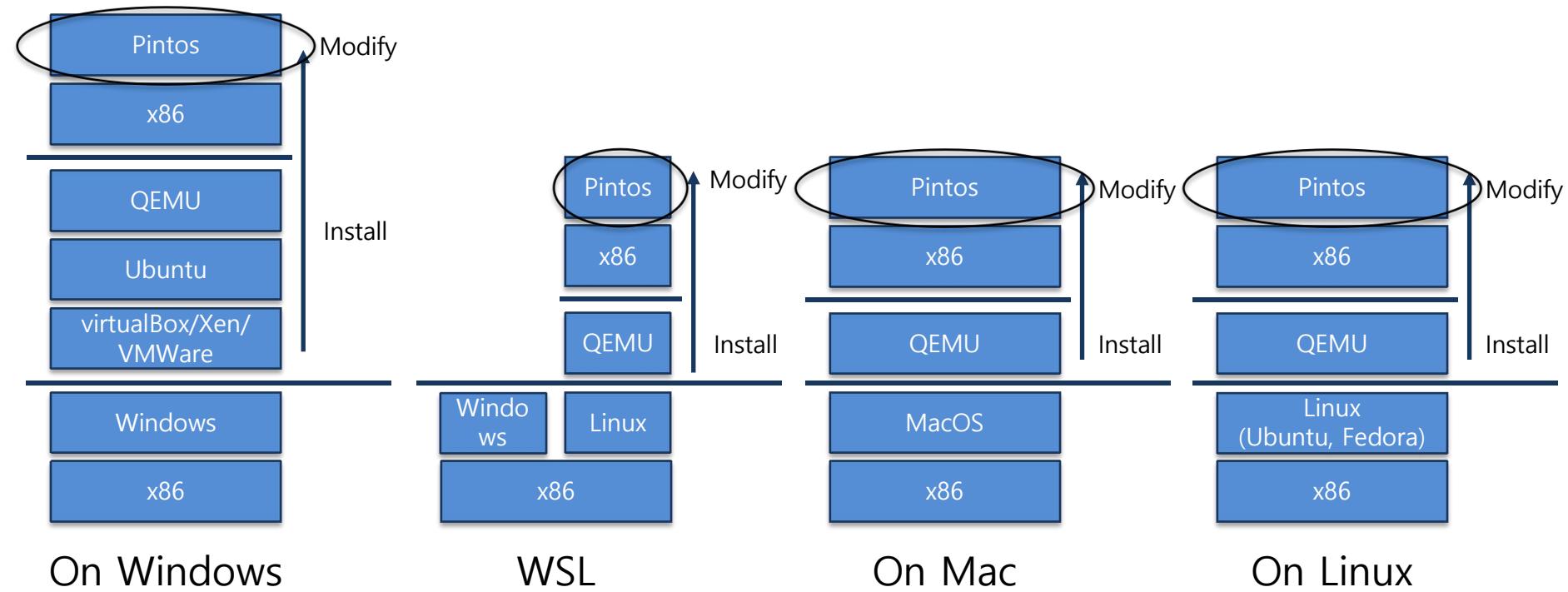
▣ Pintos?

- ◆ Educational operating system for x86 architecture
- ◆ Developed by Ben Pfaff in Stanford Univ, 2004
- ◆ Support kernel threads, loading and running user programs, file system, etc.
- ◆ Uses x86 simulators, such as Bochs or QEMU

▣ Why we use Pintos?

- ◆ It is important to implement a variety of concepts (threads, processes, memory management, and file systems) in the operating system manually
- ◆ Commercial operating systems, such as Linux are very large(1 million lines). More than 80% of 1 million lines are device driver codes to support hardware.
- ◆ Linux compile: takes at least an hour.
- ◆ Pintos : Simple, easy to understand, easy to compile

Execution of Pintos



Install pintos

- ▣ Install pintos on Windows
 - ◆ VM + Linux + QEMU + PintOS
 - ◆ WSL + QEMU + PintOS
- ▣ Install pintos on Linux
 - ◆ QEMU + PintOS
- ▣ Install PintOS on MacOS
 - ◆ QEMU + PintOS

For Windows user: Install Virtual Box

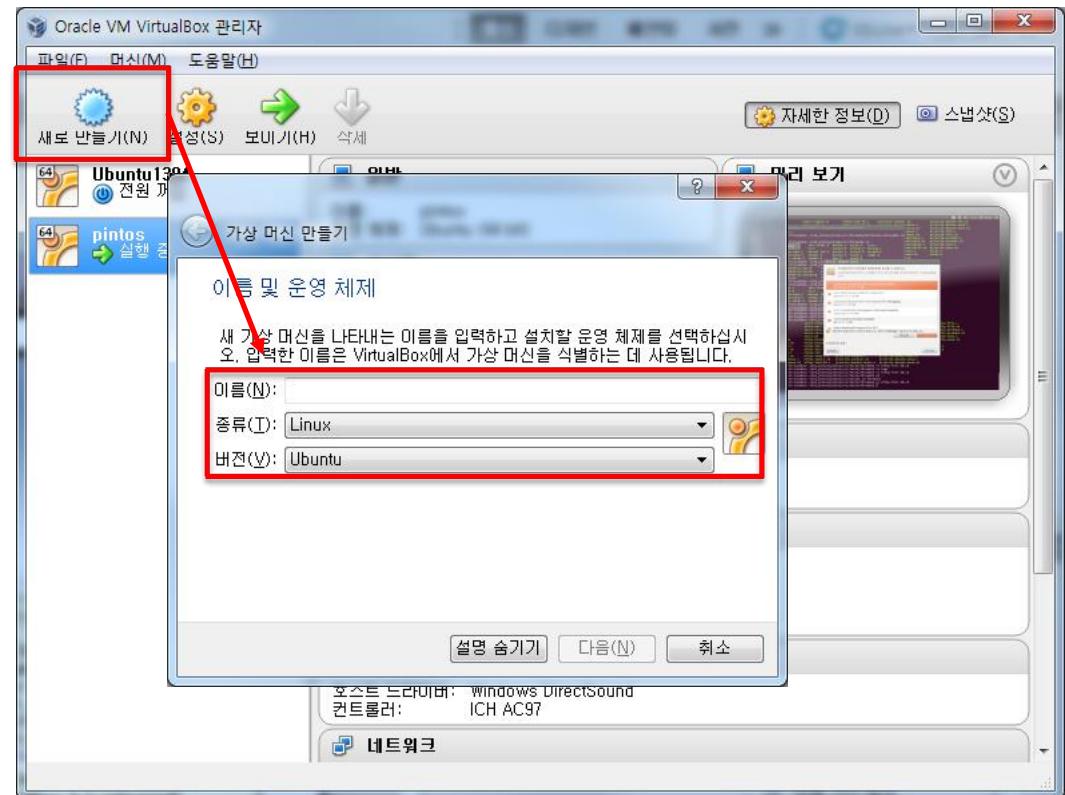
- Download Virtual Box at <http://www.virtualbox.org> and install it.

Here, Download



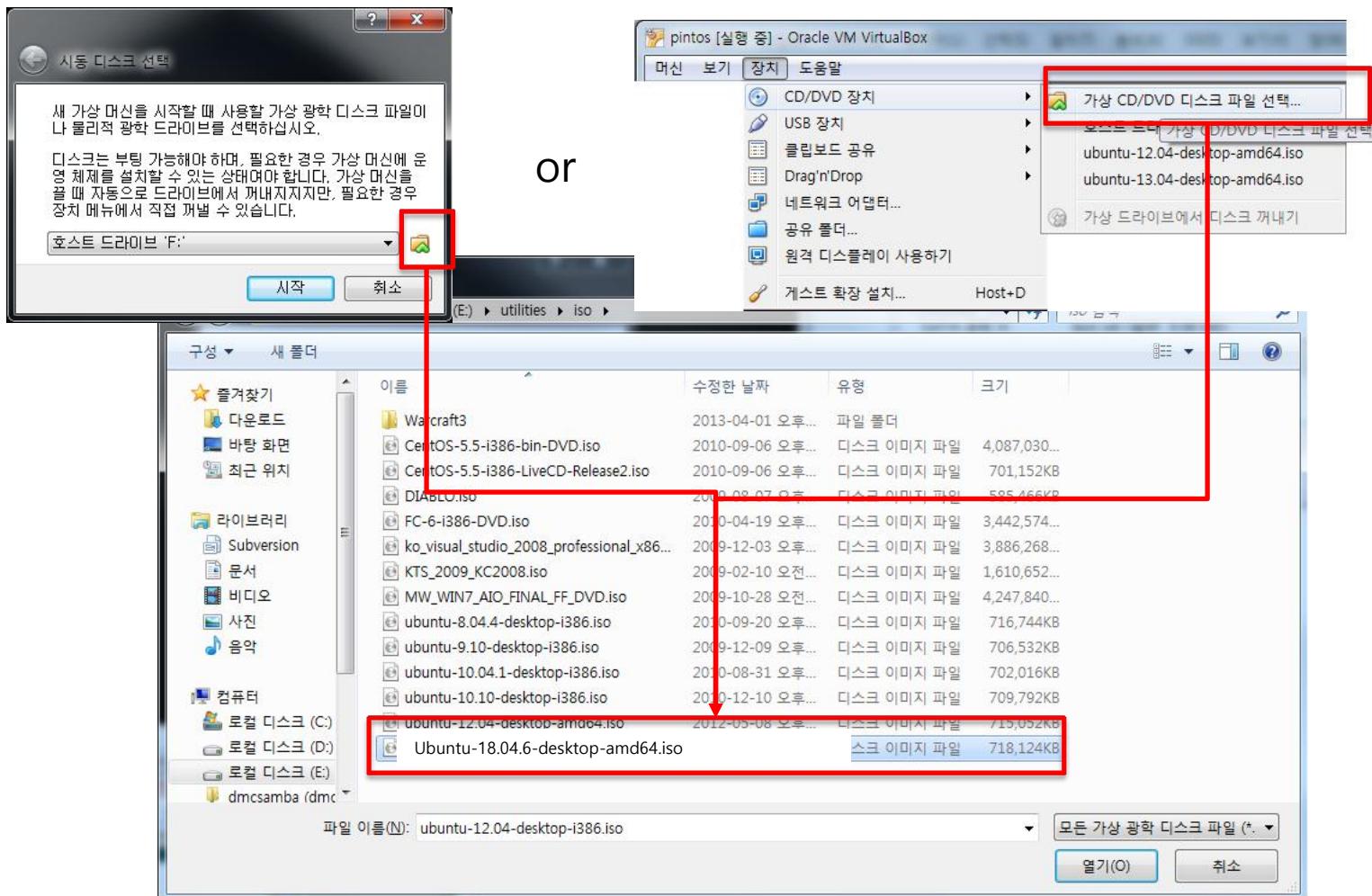
For Windows user: Install Ubuntu

- Create a Linux Virtual Machine on VirtualBox, and then install Linux (Ubuntu 18.04 LTS)
 - ◆ Download ubuntu-18.04.6-desktop-amd64.iso : <https://releases.ubuntu.com/18.04.6/?ga=2.173865891.1436103949.1646090779-879543907.1646090779>
 - ◆ Create the Virtual Machine



For Windows user: Install Ubuntu

- Mount the Ubuntu image file, and install



For Windows user: Ubuntu Installation Complete

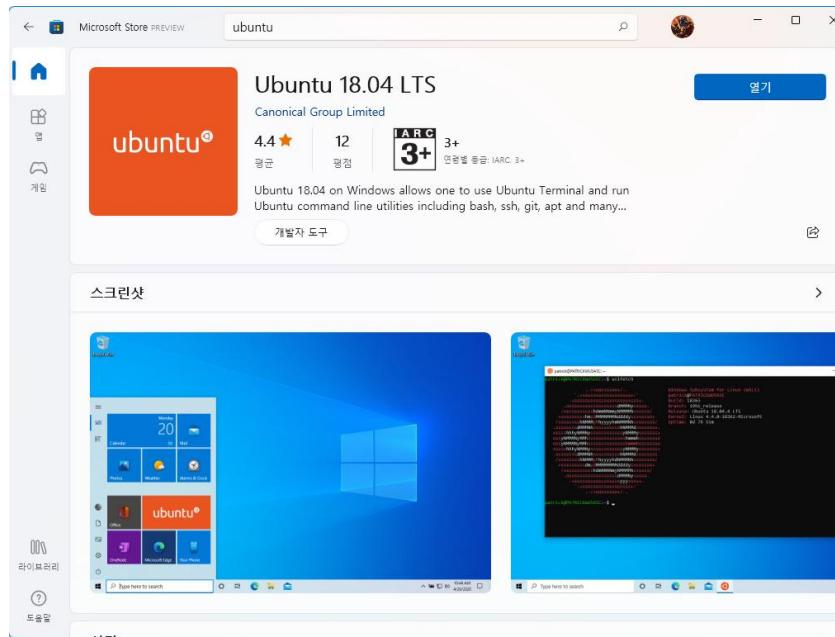
- ❑ Ubuntu Installation Complete and booting

WSL: Install WSL (Recommended)

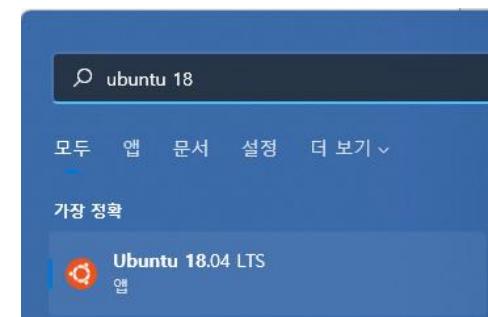
- <https://docs.microsoft.com/en-us/windows/wsl/install>
- Ensure using Windows 10 Version >= 2004 or Windows 11
- Open Powershell or Windows Terminal with administrator
- Run `wsl --install`

WSL: Install Ubuntu 18.04

- Open Microsoft Store (<https://aka.ms/wslstore>)
- Search 'Ubuntu' and find Ubuntu 18.04 LTS

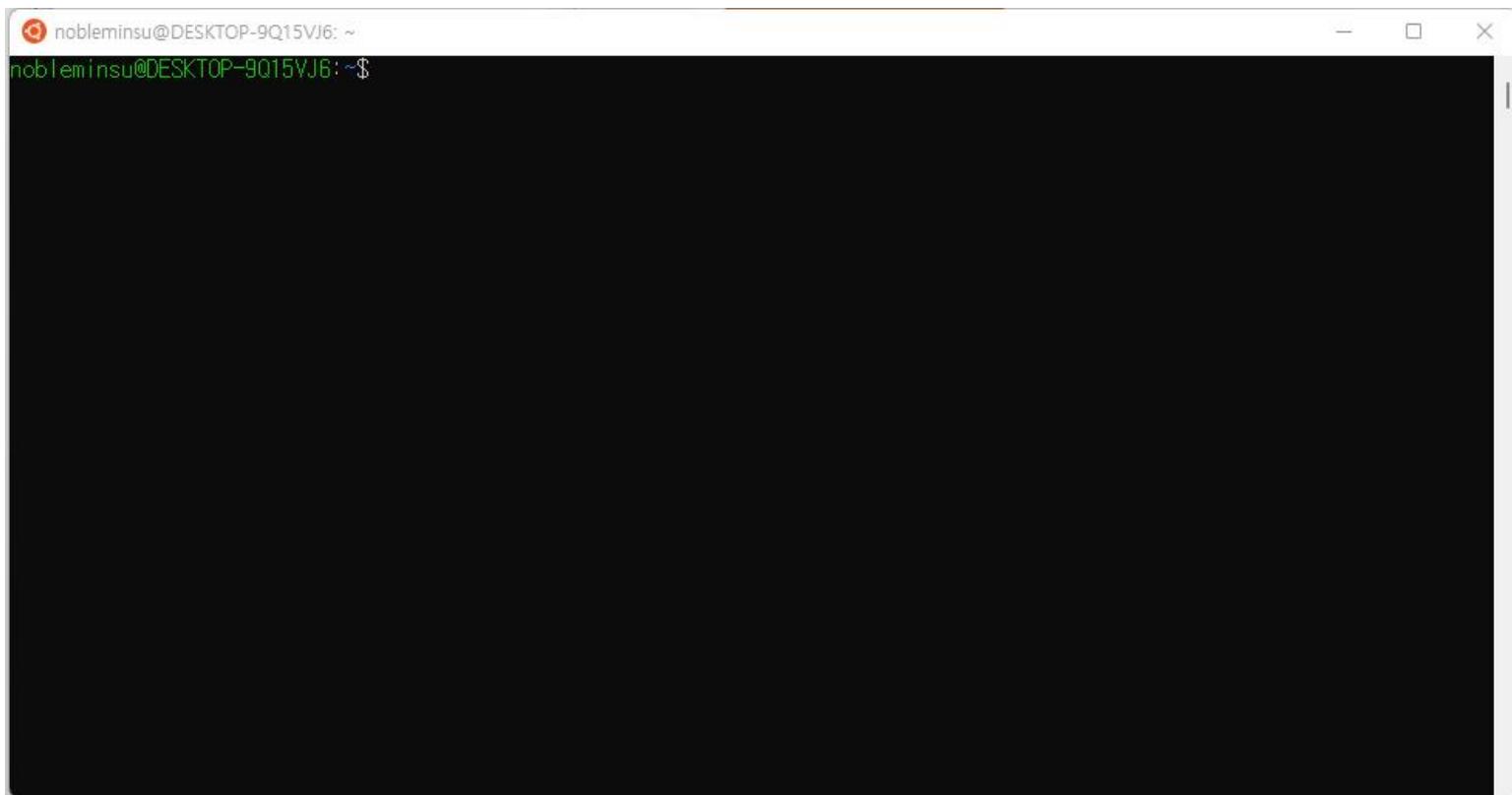


- Install it
- Find the installed program and run it



WSL: Ubuntu Installation Complete

- Set up account and you will see shell screen



For Linux: Install QEMU

- Install QEMU on system

("qemu-system-i386" command is available after below command)

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

- Make link "qemu"

("qemu" command is available after executing the command below)

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

For MacOS: Install QEMU

- Install qemu

```
$ brew install qemu
```

- Make link “qemu”

(“qemu” command is available after below command)

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

Errors

- ❑ Error 1: C compiler cannot create executables

- ◆ Install gcc, g++ and library package

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

- ❑ Error 2: X windows libraries were not found

- ◆ Install X windows library

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

Install PintOS

- Download the source code from the class piazza.

- Unzip and untar

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

- cd to pintos/src/threads/

```
$ make
```

```
$ cd build
```

```
$ pintos --qemu -- -q run alarm-
```

```
root@ubuntu: /home/lee/Downloads/pintos/src/threads/build
(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 4: duration=50, iteration=7, product=140
(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.
```

Install PintOS (MacOS)

- Change the simulator to qemu in Make.vars in src directory.

```
SIMULATOR = --qemu
```

- Trouble shooting I

Can't exec "qemu": No such file or directory at /home/arpith/pintos/src/utils/pintos line 923.

Change the line 623 of perl script (filename: pintos in src/utils)

before: my (@cmd) = ('qemu');

After: my (@cmd) = ('qemu-system-i386');



Install PintOS (MacOS)

▣ Trouble shooting II

```
baekdu:threads yjwon$ make
cd build && /Applications/Xcode.app/Contents/Developer/usr/bin/make all
../../Make.config:37: *** Compiler (i386-elf-gcc) not found. Did you set $PATH
properly? Please refer to the Getting Started section in the documentation for d
etails. ***
/bin/sh: i386-elf-ld: command not found
make[1]: Nothing to be done for `all'.
```

▣ Add the location of i386-elf-ld to your \$PATH.

```
% echo PATH="$PATH:/opt/local/bin" >> ~/.bashrc
% source ~/.bashrc
```

Install PintOS (MacOS)

▣ Trouble shoot III

```
cd build && /Applications/Xcode.app/Contents/Developer/usr/bin/make all
ld: unknown option: -melf_i386
gcc -m32 -E -nostdinc -I../../ -I../../lib -I../../lib/kernel -P ../../threads/kernel.lds.S > threads/kernel.lds.s
gcc -m32 -c ../../threads/start.S -o threads/start.o -Wa,--gstabs -nostdinc -I../../ -I../../lib -I../../lib/kernel -MMD -MF threads/start.d
clang: error: unsupported argument '--gstabs' to option 'Wa,'
make[1]: *** [threads/start.o] Error 1
make: *** [all] Error 2
```

▣ It failed to locate the compiler. We need to enforce the compiler selection.

Install PintOS (MacOS)

- Enforce the compiler selection. Change the compiler setting. Comment out the line 24-30 in src/Make.config as follows.

```
#ifneq (0, $(shell expr `uname -m` : '$(X86)'))  
# CC = $(CCPROG)  
# LD = ld  
# OBJCOPY = objcopy  
#else  
# ifneq (0, $(shell expr `uname -m` : '$(X86_64)'))  
#     CC = $(CCPROG) -m32  
#     LD = ld -melf_i386  
#     OBJCOPY = objcopy  
# else  
#     CC = i386-elf-gcc  
#     LD = i386-elf-ld  
#     OBJCOPY = i386-elf-objcopy  
# endif  
#endif
```

Install PintOS (MacOS)

▣ Trouble shoot IV

In normal situation, PintOS should quit with '-q' option and shell needs to be ready to accept the next command. If the shell hangs, the PintOS failed to shutdown the system properly. There is a problem with the ACPI shutdown module. Please make the following update in src/devices/shutdown.c.

```
printf ("Powering off...\n");
serial_flush ();

//add the following line
++ outw( 0x604, 0x0 | 0x2000 );

/* This is a special power-off sequence supported by Bochs and
QEMU, but not by physical hardware. */
for (p = s; *p != '\0'; p++)
    outb (0x8900, *p);
```

GCC version issue

- Downgrade the gcc version to 4.5 (recommended for pintos)

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

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

Operating Systems Lab: Tools

- GDB, ctags, cscope -



Youjip Won

Contents

1. GDB
2. ctags
3. cscope

- What is GDB?

- ◆ Command line debugging tool made by GNU
- ◆ It provides many functions which can trace the internal behaviors of the computer program while it is executed.

- Compile for GDB

- ◆ \$ gcc **-g** -o gdb_test gdb_test.c

- Run GDB

- ◆ \$ gdb [file]

- Pass argument to GDB

- ◆ \$ gdb [file]
- ◆ (gdb) set args argument

GDB (Cont.)

gdb_test.c

```
1 #include <stdio.h>
2
3 void print() {
4     printf("Hello World!\n");
5 }
6
7 int main(void) {
8
9     int i;
10    for(i = 0; i < 10; i++) print();
11    return 0;
12 }
```

```
Reading symbols from gdb_test...done.
(gdb) break 7
Breakpoint 1 at 0x40053f: file gdb_test.c, line 7.
(gdb) break print
Breakpoint 2 at 0x40052a: file gdb_test.c, line 4.
(gdb) run
Starting program: /home/sundoo/gdb_test

Breakpoint 1, main () at gdb_test.c:10
warning: Source file is more recent than executable.
10                                     for (i=0; i < 10; i++) print();
(gdb) continue
Continuing.

Breakpoint 2, print () at gdb_test.c:4
4                                         printf("Hello World!\n");
(gdb) next
Hello World!
5 }
(gdb) continue
Continuing.

Breakpoint 2, print () at gdb_test.c:4
4                                         printf("Hello World!\n");
(gdb) bt
#0  print () at gdb_test.c:4
#1  0x00000000000400552 in main () at gdb_test.c:10
(gdb) q
A debugging session is active.

Inferior 1 [process 20018] will be killed.

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

GDB commands

- ▣ **continue(c)**: Run program continuously.
- ▣ **step(s)**: Run the code line by line. If cursor is on a function, GDB enters the function.
- ▣ **next(n)**: Run the code line by line. If cursor is on a function, GDB runs the function and move to next line.
- ▣ **finish**: Run the function to the end and stop.
- ▣ **return [value]**: Cancel the running function and return with [value].
- ▣ **list**: print the source of the running part.
 - ◆ It prints main function before program run.
- ▣ **list [number]**: Print [number] line.
- ▣ **list [function]**: Print source code of [function]
- ▣ **set listsize [n]**: Set line size of list. Default is 10

GDB commands (Cont.)

- ▣ Commands for print: check the state of program
 - ◆ **whatis [var]**: Print type of variable.
 - ◆ **print(p) [var]**: Print value of variable.
 - print a->member
 - print add(1,2)
 - print /x value // you can set the print type using x, u, o, c

GDB commands (Cont.)

- ▣ Commands for break: Pause the process at the location you want
 - ◆ **break(b) [number]**: Pause the process at [number] line.
 - ◆ **break(b) [function]**: Pause the process at the [function].
 - ◆ **break(b) [file:function]**: Pause the process at the [function] in [file].
 - ◆ **break(b) [file:number]**: Pause the process at [number] line in [file].
 - ◆ **info break**: Print state of break point
 - ◆ **delete [number]**: Delete break point of [number]. If any number is not set, delete all break point.
- ▣ Command for call stack(history of calling)
 - ◆ **backtrace(bt)**: Print all called functions until this function is called.
 - ◆ **backtrace(bt) [number]**: Print [number] lines of bt command result.

GDB commands (Cont.)

❑ Stack frame

- ◆ Call stack consists of stack frames(or frames).
- ◆ Stack frame contains arguments, local variables and return address of function.
- ◆ commands
 - **frame [number]**: Select a frame of [number], print name of the selected function.
 - **select-frame [number]**: Select a frame of [number], don't print name of the selected function.
 - **info frame**: Print the stored data of the selected frame.

Sample Script - gdb (1)

- Run pintos with gdb (–qemu can be omitted if you use bochs)

```
% cd pintos/src/threads  
% make  
% cd build  
% pintos --gdb --qemu -- run alarm-multiple
```

Now pintos is waiting connection from gdb

- Run gdb and connect to pintos

Open another terminal and follow next steps in the terminal

```
% cd pintos/src/threads/build  
% gdb kernel.o
```

Now gdb shell is opened

```
(gdb) target remote localhost:1234
```

Now gdb is connected to pintos, and pintos is waiting execution order of the gdb

Sample Script - gdb (2)

- ❑ Debug `main()` function in pintos

In the gdb terminal,

```
(gdb) break main  
(gdb) info breakpoints  
(gdb) continue
```

Pintos will be executed, stop at start of main function, and gdb shell will be available

```
(gdb) delete 1  
(gdb) list  
(gdb) list thread_init  
(gdb) next  
(gdb) next
```

Now gdb is on code calling `read_command_line()`.

Let's go to inside of `read_command_line()`

```
(gdb) step
```

Now gdb came inside of `read_command_line()`

```
(gdb) next
```

Now a variable `argc` has been set. Let's figure value in `argc` out

```
(gdb) print argc
```

It will print "\$1=2". So value in `argc` is 2

Sample Script - gdb (3)

❑ Debug `main()` function in pintos (Continue...)

Now, gdb is on code “`argv[i] = p`”.

Let's figure variable `p` out.

```
(gdb) print p
```

It will print `$2 = 0xc0007d3e "run"`.

It means that `p` is pointer including `0xc0007d3e` where have string “run”

Now, there is no stuff to debug in `read_command_line()`.

Let's go back to outside.

```
(gdb) finish
```

Now gdb is on return point of `read_command_line()` at `main()`.

We don't need to execute each code with next command to return function

ctags

- ▣ The tool for making tag file which points location of function, variable, string, etc. of a source file.
- ▣ “tags” file: dictionary that contain all variables and function names and the associated locations.
- ▣ Create a tag file.

ctags [option] [file(s)] / etags [option] [file(s)]

- -R: Scan all subdirectory recursively.
- --exclude=[pattern]: Exclude files and directories which have 'pattern' in name from creating tag file.
- -f [file]: Create tag file with 'file'. If 'file' is '-', tag file has a name with 'tags(case of ctags)', 'TAGS(case of etags)' defaultly.
- --list-languages: Print language list supporting tag file create.
- -x: Print tags as table to stdout without creating tag file.

ctags command in vim

- Keyboard command Action
- Ctrl-] Jump to the tag underneath the cursor
- :ts <tag> <RET> Search for a particular tag
- :tn Go to the next definition for the last tag
- :tp Go to the previous definition for the last tag
- :ts List all of the definitions of the last tag
- Ctrl-t Jump back up in the tag stack

Sample Script

- ▣ Install ctags (in Linux)

```
%sudo apt-get install ctags
```

- ▣ Create tag file

```
%cd pintos/src  
%ctags -R  
%ls tags
```

- ▣ vim and ctag

```
% cd pintos/src  
%vim tags

- ◆ Set tag file from the command mode  
set tags=./tags

```

- ▣ Basic command

- ◆ In command mode, type "tj main" → list the path, filename and line number of `main()` functions
- ◆ Move to `main()` pintos (line 35, threads/init.c)
- ◆ Locate the cursor at function `bss_init` at `main()` and type `Ctrl-]`.
- ◆ Locate the cursor at `memset` and type `Ctrl-]`
- ◆ `tp` from command mode: goes to previous location.

cscope

- The tool for accessing object like ctags, but with more powerful features.
- Find the functions that "call" a given function or that "is called" by a given function.
 - ◆ Find all functions that call malloc()
 - ◆ Find all functions that are called by malloc().

```
cs find <Command Character> <String>
```

Command	Description
s	Find C symbol String
g	Find definition String
d	Find functions called by function String
c	Find functions calling function String
t	Find text String
e	Find egrep pattern String
f	Find file String
i	Find files #including file String

Sample Script

- ▣ Install cscope
 - % sudo apt-get install cscope
- ▣ Create tag file
 - % cd pintos/src
 - % find ./ -name "*.[chS]" > cscope.files
 - % cscope -i cscope.files
 - % ls cscope.out
- ▣ Use cscope in vim
 - ◆ Open the cscope.out file from vim and register the cscope.out for source code navigation.
 - % cd pintos/src
 - % vim cscope.out
 - ◆ Register scope file from the command mode of vim
 - cs add ./cscope.out
- ▣ Basic command
 - ◆ In command mode, type "cs find c memset" → list the path, filename and line number of functions calling memset()
 - ◆ In command mode, type "cs find g main" → list the path, filename and line number of main() functions
 - ◆ In command mode, type "cs find e shut*" → list the path, filename and line number of code including string of "shut*"

Operating Systems Lab: Context Switch in Pintos



Youjip Won

Overview

- Process structure
- Process state
- Process context
- schedule()
- switch thread

Process structure: struct thread

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. */

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

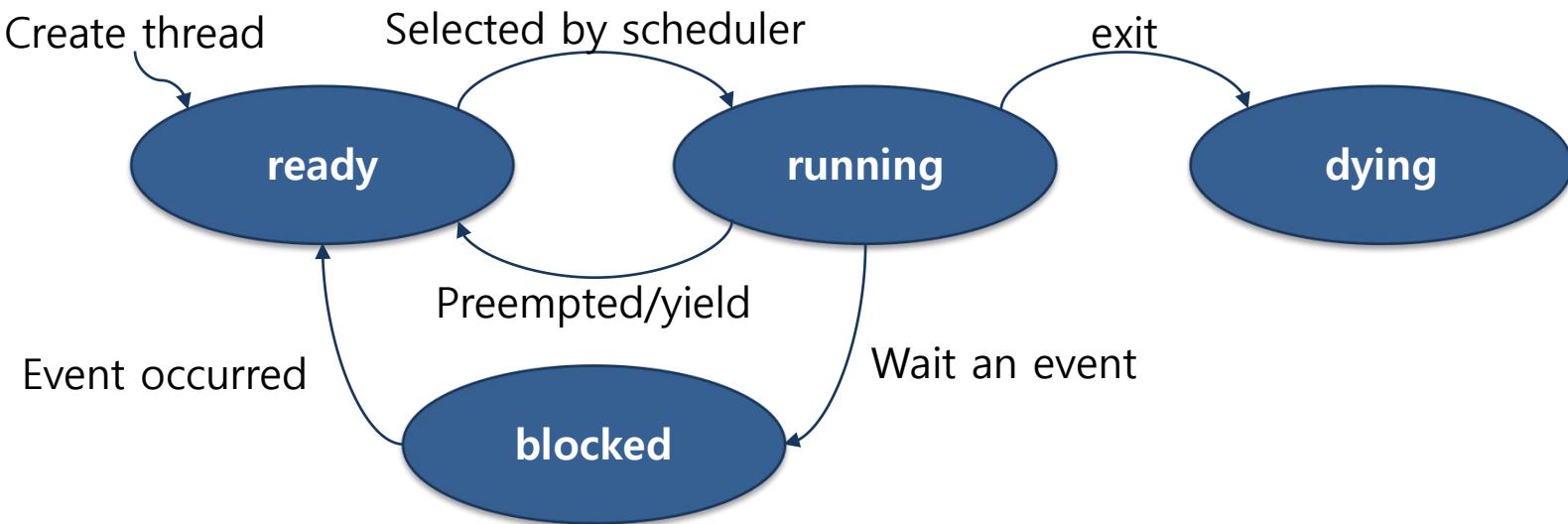
    /* Owned by thread.c. */
    unsigned magic;                /* Detects stack overflow. */
};
```

```
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. */
};
```

Process state

Process state

```
/* States in a thread's life cycle. */
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. */
};
```



Creating a thread

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

- Creates a new kernel thread named NAME with the given PRIORITY, which executes FUNCTION passing AUX as the argument, and adds it to the ready queue. Returns the thread identifier for the new thread, or TID_ERROR if creation fails.

Creating a thread

Allocating a struct thread object

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
    ASSERT (t != NULL);
    ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
    ASSERT (name != NULL);

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);
}
```

Process list

pintos/src/threads/thread.c

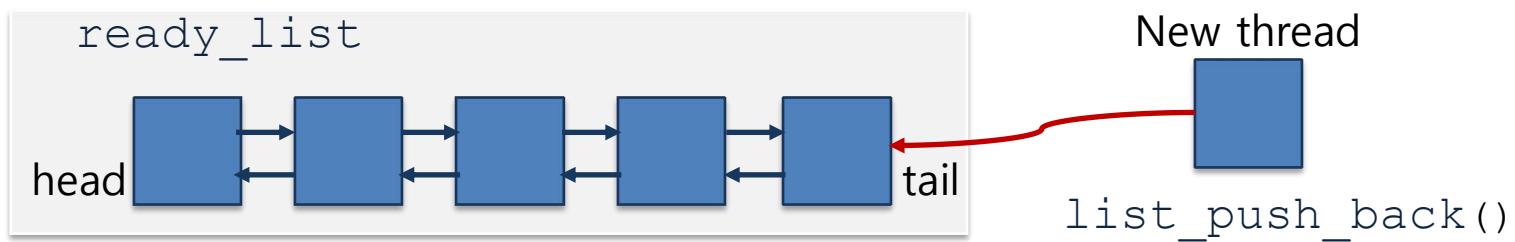
```
/* List of processes in THREAD_READY state, that is, processes that
   are ready to run but not actually running. */
static struct list ready_list;

/* 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: a set of threads that are ready for execution
- ▣ all_list: A set of all threads in the system.

Creating a thread

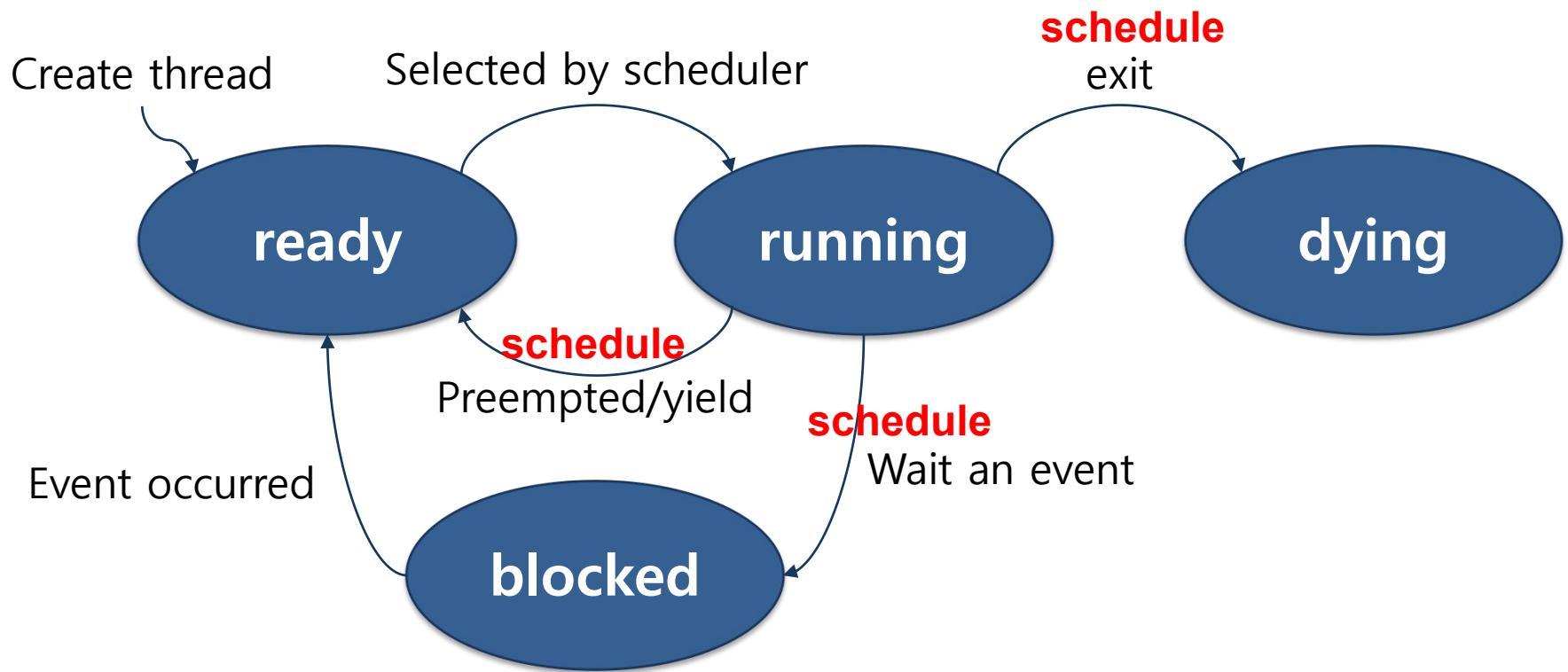
- ▣ Newly created thread is inserted at the end of the list `all_list` (`list_push_back()`)
 - ◆ `thread_create()`
 - ◆ `init_thread`
- ▣ After the thread is ready, it is inserted at the `ready_list`.
 - ◆ `thread_unblock`



schedule()

- ❑ Schedule a new process.
 - ◆ Get the currently running process.
 - ◆ Get the next process to run.
 - ◆ Switch context from current to next.
- ❑ Who calls schedule() ?
 - ◆ exit, block, yield
 - ◆ *Or pre-empted, (when the time quantum expires...)*
- ❑ Before calling schedule
 - ◆ Disable interrupt.
 - ◆ Change the state of the running thread from running to something else.

schedule()



```
void
thread_block (void)
{
    ASSERT (!intr_context ());
    ASSERT (intr_get_level () == INTR_OFF);

    thread_current ()->status = THREAD_BLOCKED;
    schedule ();
}

void
thread_exit (void)
{
    ASSERT (!intr_context ());

    #ifdef USERPROG
    process_exit ();
    #endif

    /* Remove thread from all threads list, set our status to dying,
       and schedule another process. That process will destroy us
       when it calls thread_schedule_tail(). */
    intr_disable ();
    list_remove (&thread_current ()->allelem);
    thread_current ()->status = THREAD_DYING;
    schedule ();
    NOT_REACHED ();
}
```

```
/* Yields the CPU. The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim. */
void
thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    ASSERT (!intr_context ());

    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);
}
```

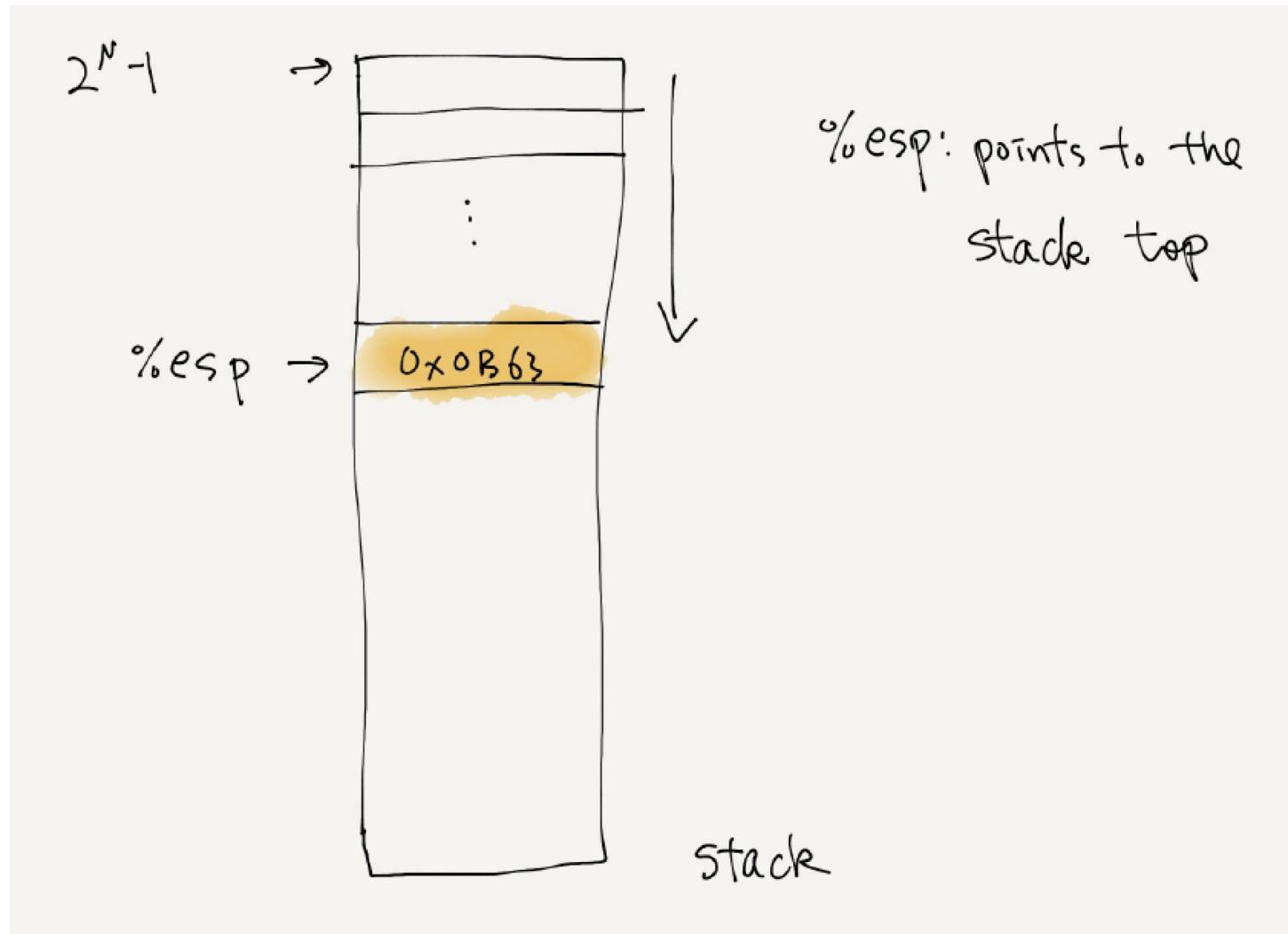
schedule (void)

```
static void
schedule (void)
{
    struct thread *cur = running_thread ();
    struct thread *next = next_thread_to_run ();
    struct thread *prev = NULL;

    ASSERT (intr_get_level () == INTR_OFF);
    ASSERT (cur->status != THREAD_RUNNING);
    ASSERT (is_thread (next));

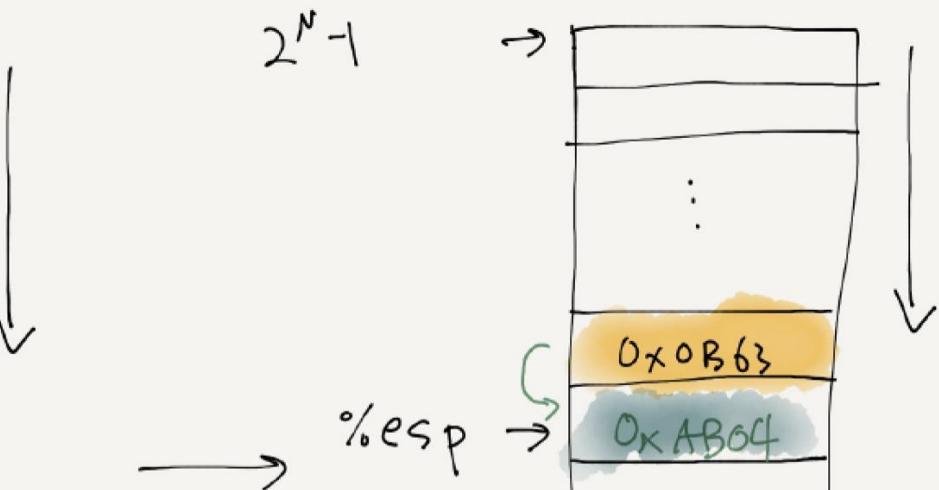
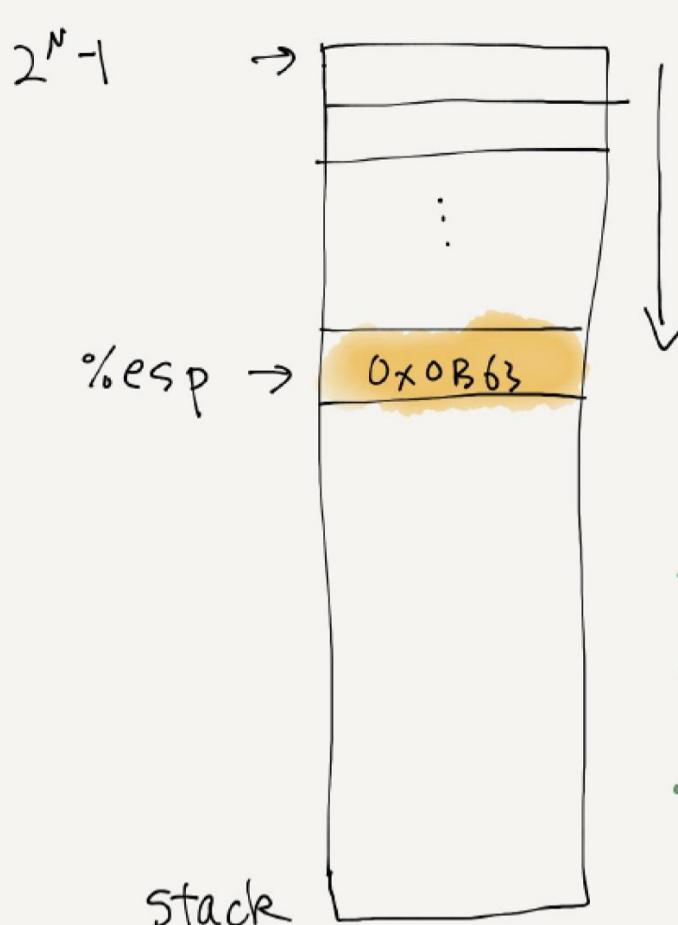
    if (cur != next)
        prev = switch_threads (cur, next);
    thread_schedule_tail (prev);
}
```

Stack



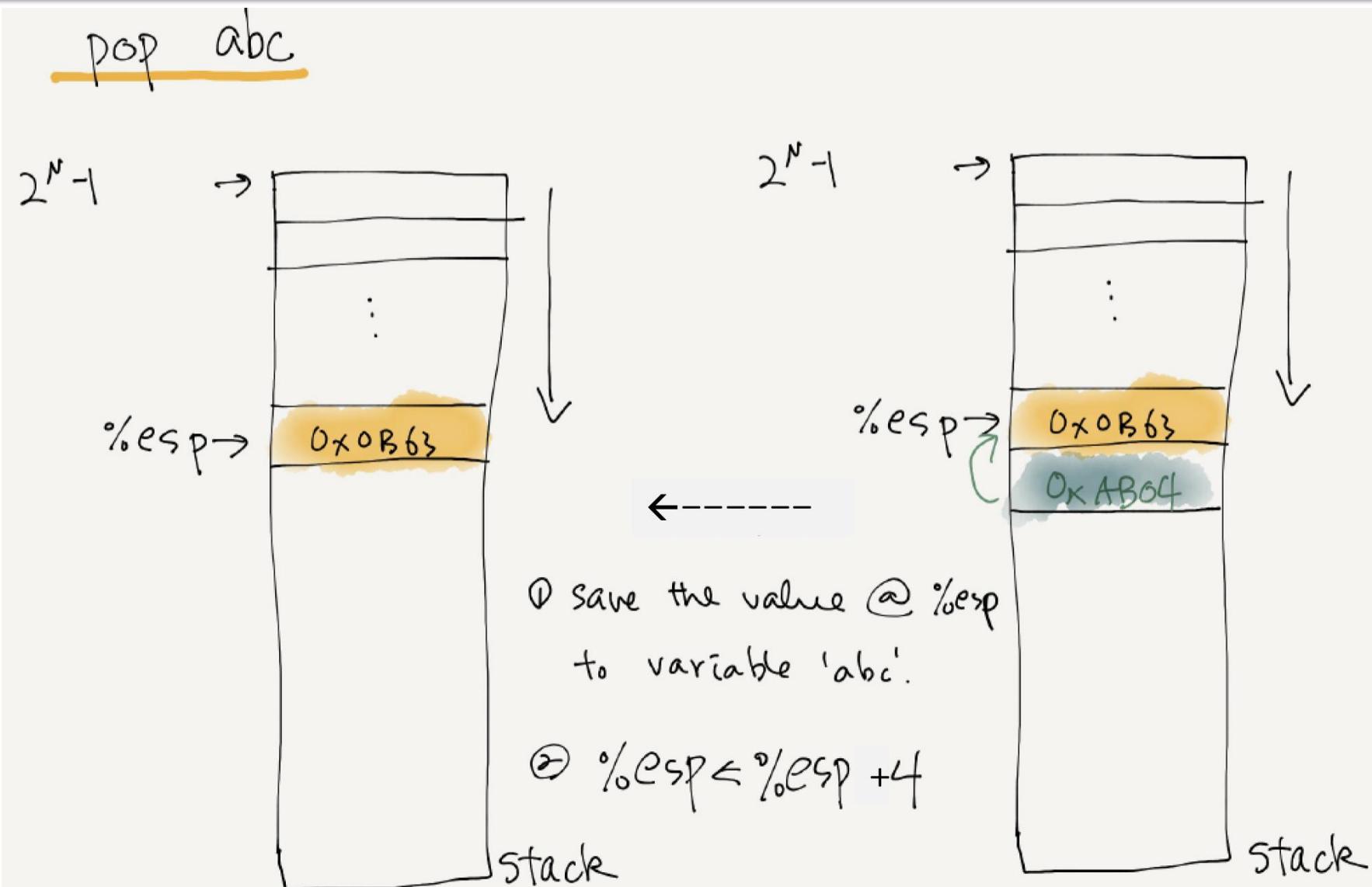
Process context

push abc



- get value from abc
- $\%esp \leftarrow \%esp - 4$
- store the value at the addr. pointed by `%esp`.

Process context



User stack vs. kernel stack

Executing in User Mode

- user stack

executing in the kernel

- use kernel stack

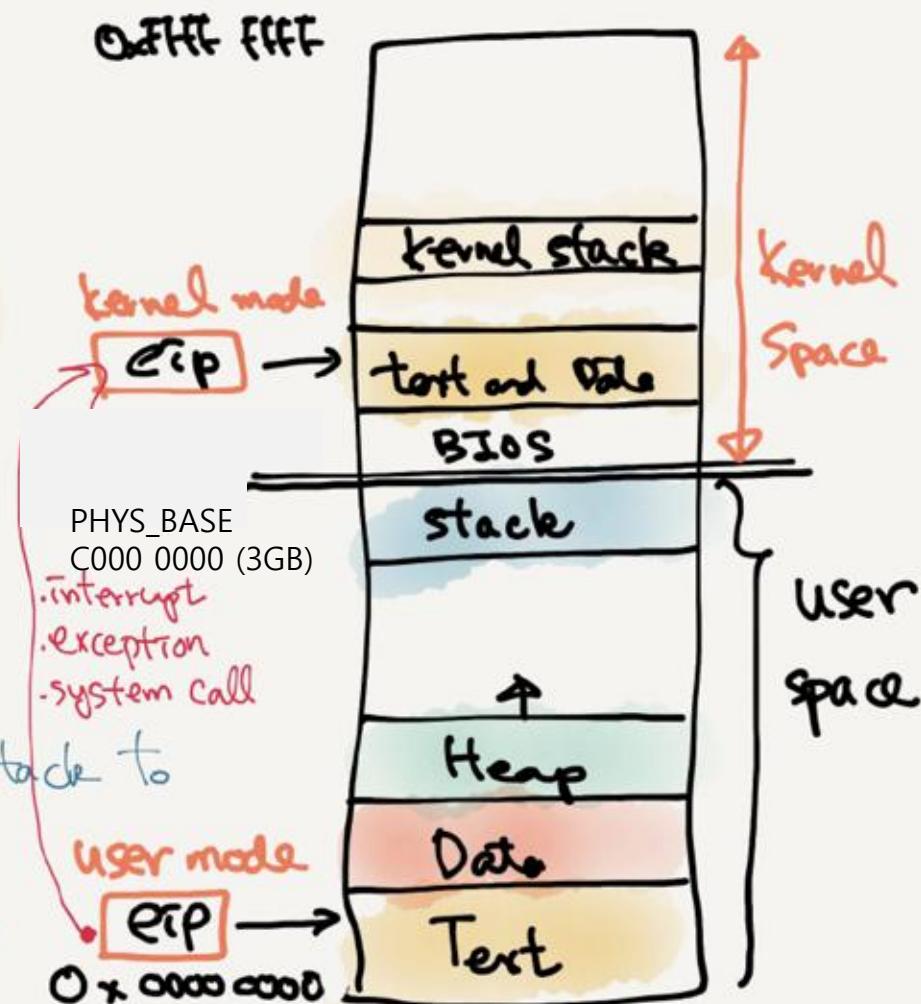
- System Call

- entering the kernel

① **switch** from user stack to

kernel stack

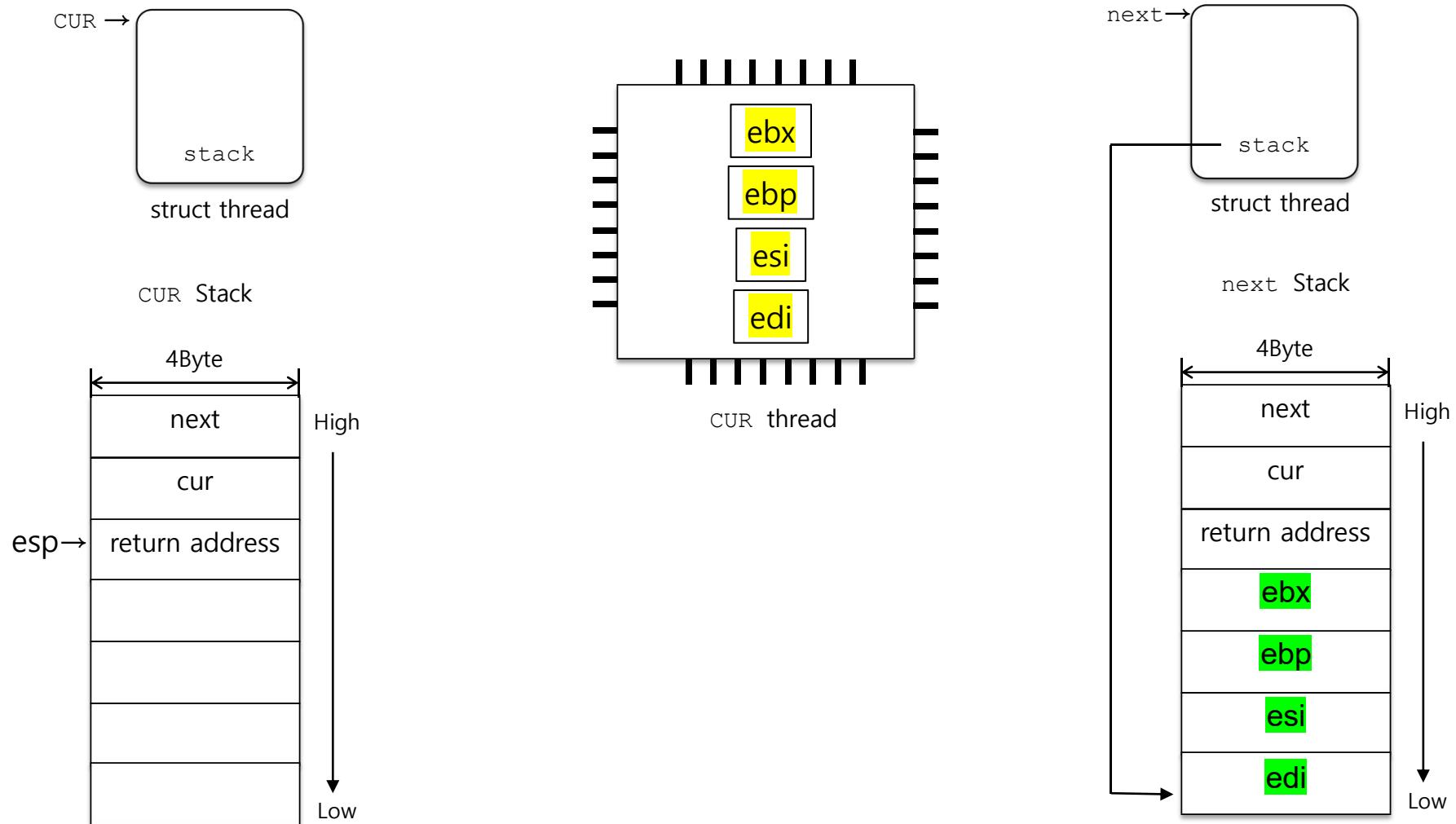
- ② **raise privilege level**



```
switch_threads(struct thread *cur,  
               struct thread *next)
```

- ▣ Save the registers on the kernel stack of cur.
- ▣ Save the location of the current stack top at the current `struct thread`'s `stack` member.
- ▣ Restore the new thread's stack top (kernel stack) into CPU's stack pointer (`esp`).
- ▣ Restore registers from the stack (kernel stack).

Call switch_threads



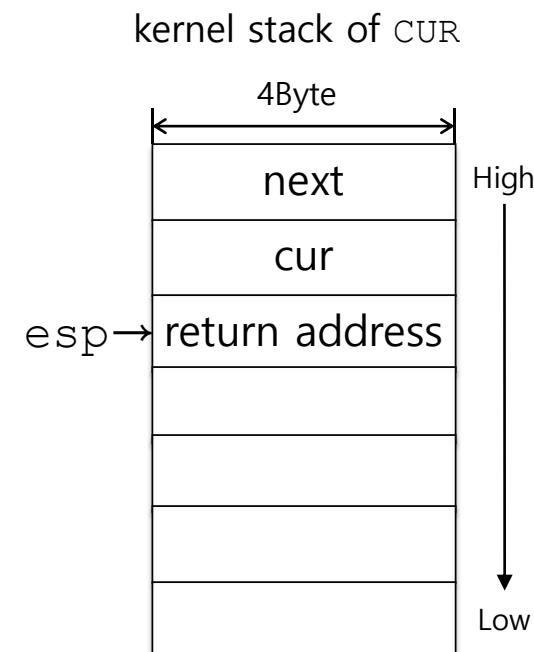
```
switch_threads(struct thread *cur,  
                struct thread *next)
```

- Two tasks

- Switch the CPU context from CUR thread to NEXT thread.
- Return cur (as prev below).

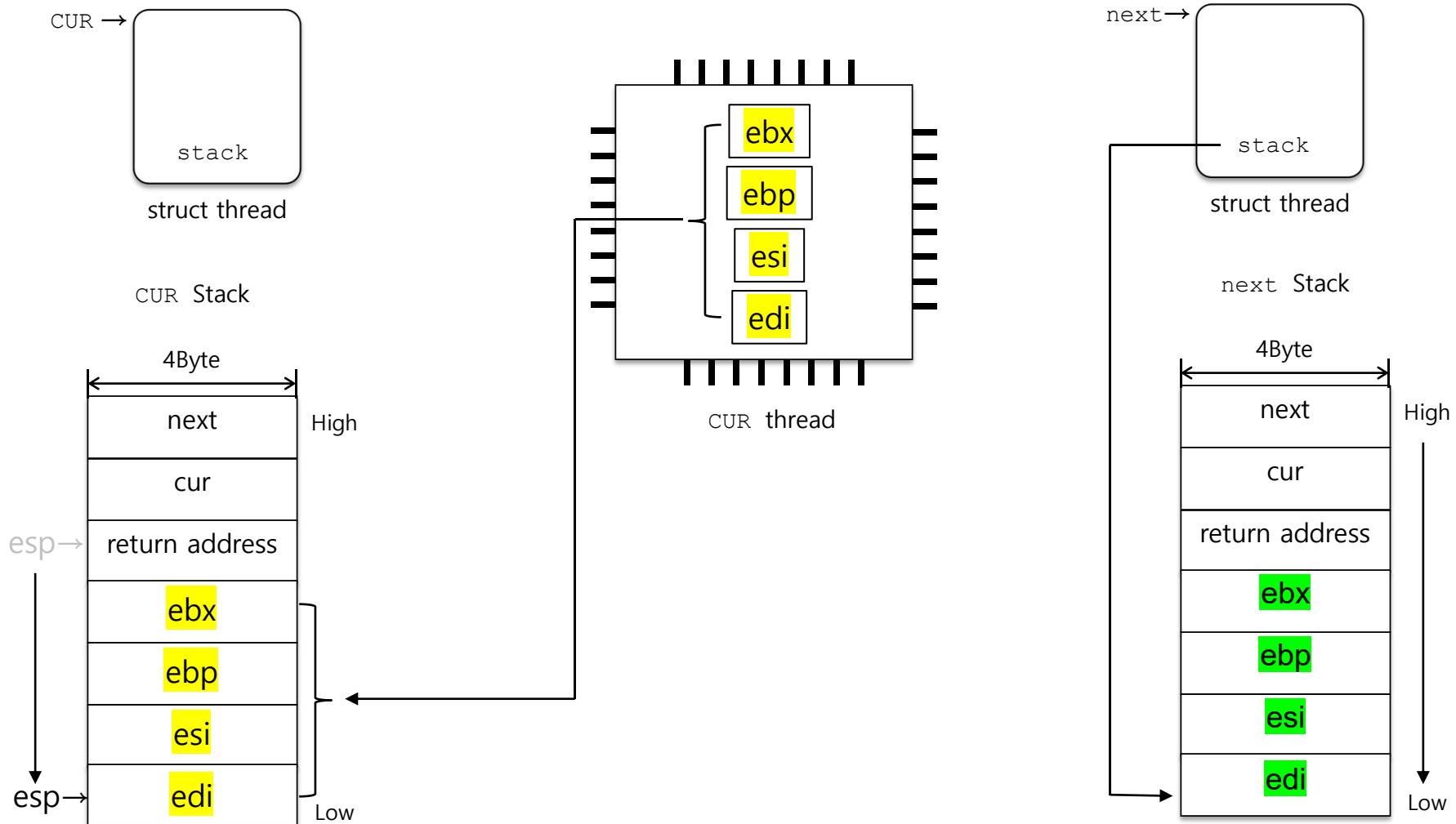
thread/thread.c

```
static void  
schedule (void)  
{  
    ...  
  
    prev = switch_threads(cur, next);  
  
    ...  
}
```



Just before jump into
switch_thread

Store current context

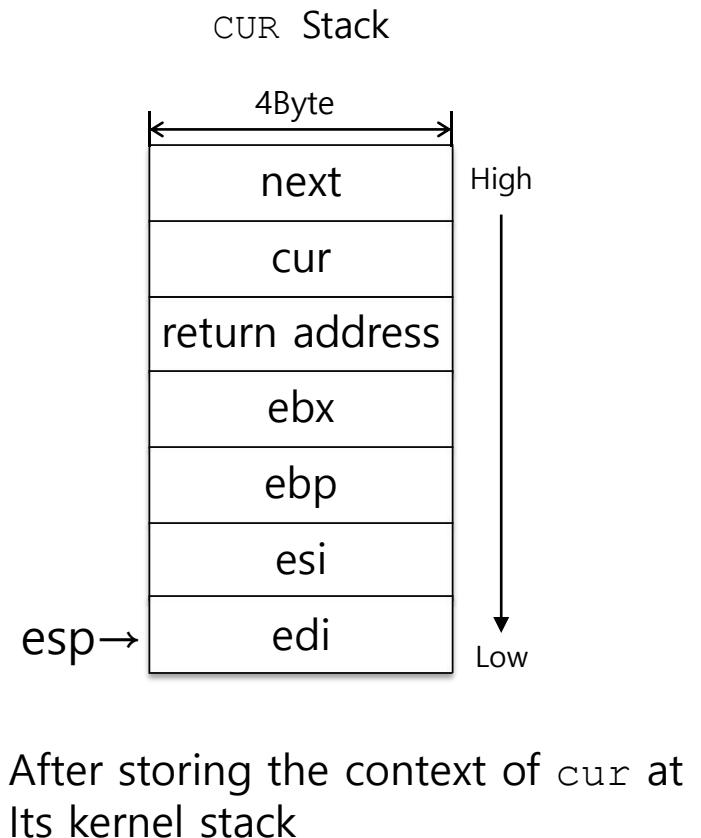


switch_threads: I

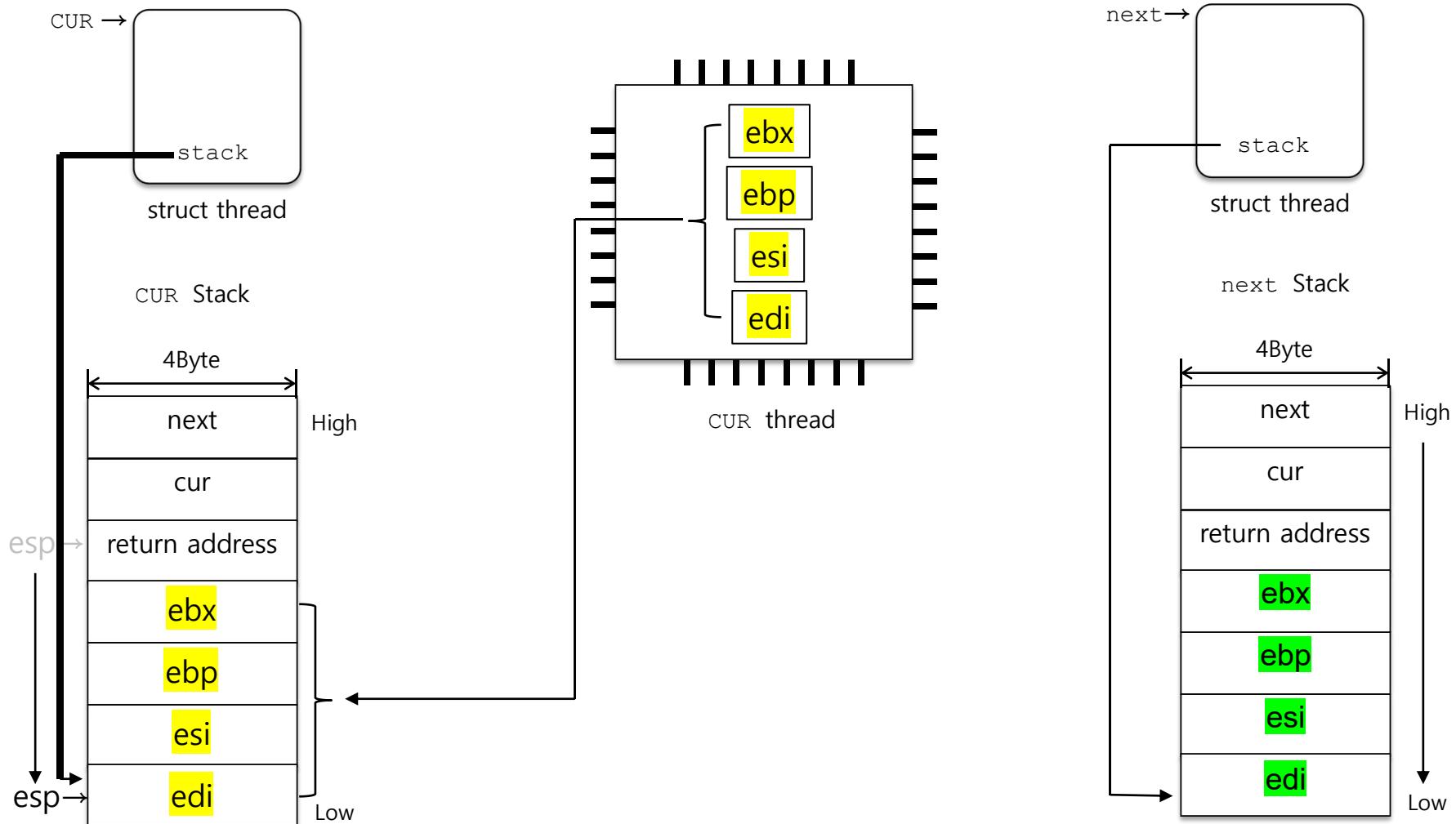
- Save current thread's registers at its kernel stack.

thread/switch.S

```
switch threads:  
    pushl %ebx  
    pushl %ebp  
    pushl %esi  
    pushl %edi  
  
.globl thread_stack_ofs  
    mov thread_stack_ofs, %edx  
  
    movl SWITCH_CUR(%esp), %eax  
    movl %esp, (%eax,%edx,1)  
  
    movl SWITCH_NEXT(%esp), %ecx  
    movl (%ecx,%edx,1), %esp  
  
    popl %edi  
    popl %esi  
    popl %ebp  
    popl %ebx  
    ret
```



Store current context



switch_threads

1. Get the offset of stack pointer in thread structure.

thread/thread.c

```
uint32_t thread_stack_ofs = offsetof (struct thread, stack)
```

lib/stddef.h

```
#define offsetof(TYPE, MEMBER) ((size_t)&((TYPE*)0)->MEMBER)
```

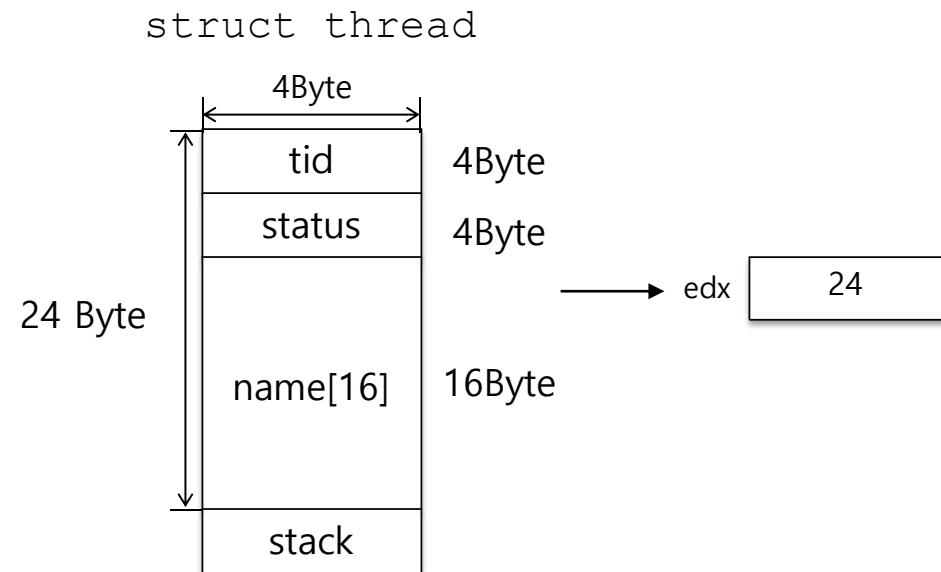
2. Load the offset to edx.

```
.globl thread_stack_ofs
mov thread_stack_ofs, %edx

movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)

movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

popl %edi
popl %esi
popl %ebp
popl %ebx
ret
```



```
switch_threads(struct thread *cur,  
                struct thread *next)
```

3. Save the location of the current struct thread to eax. (return value)

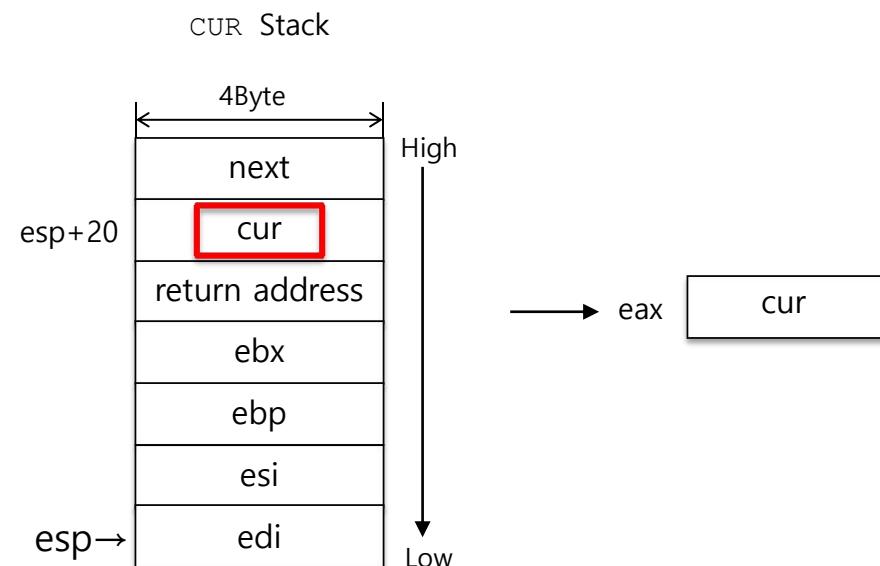
thread/switch.S

```
switch_threads:  
    pushl %ebx  
    pushl %ebp  
    pushl %esi  
    pushl %edi  
  
.globl thread_stack_ofs  
    mov thread_stack_ofs, %edx  
  
    movl SWITCH_CUR(%esp), %eax  
    movl %esp, (%eax,%edx,1)  
  
    movl SWITCH_NEXT(%esp), %ecx  
    movl (%ecx,%edx,1), %esp  
  
    popl %edi  
    popl %esi  
    popl %ebp  
    popl %ebx  
    ret
```

thread/switch.h

```
#define SWITCH_CUR 20
```

- $\text{SWITCH_CUR}(\%esp) = \%esp + 20$



switch_threads

- Save current thread's stack top address at the struct thread (stack field).

$$(\%eax, \%edx, 1) = \%eax + \%edx * 1$$

%eax: location of the struct thread of CUR

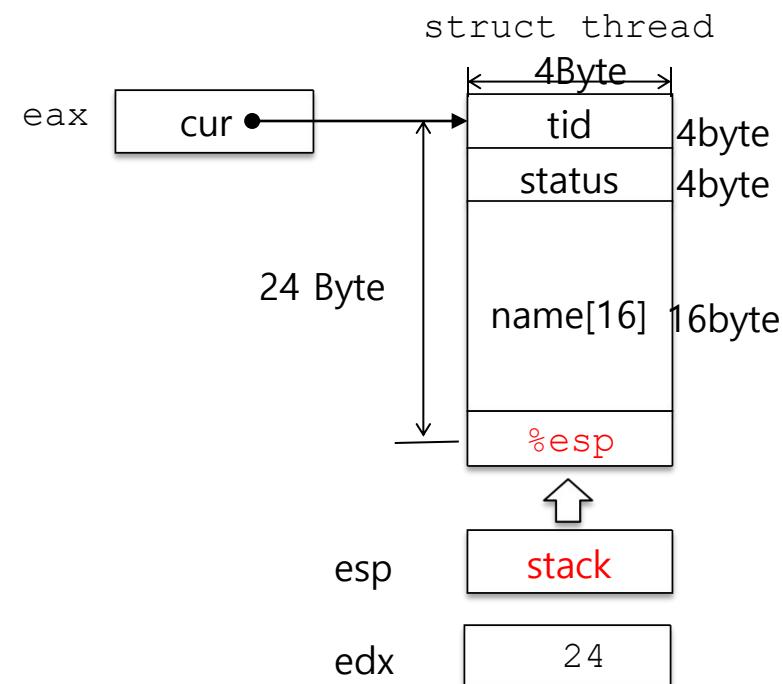
thread/switch.S

```
.globl thread_stack_ofs
mov thread_stack_ofs, %edx

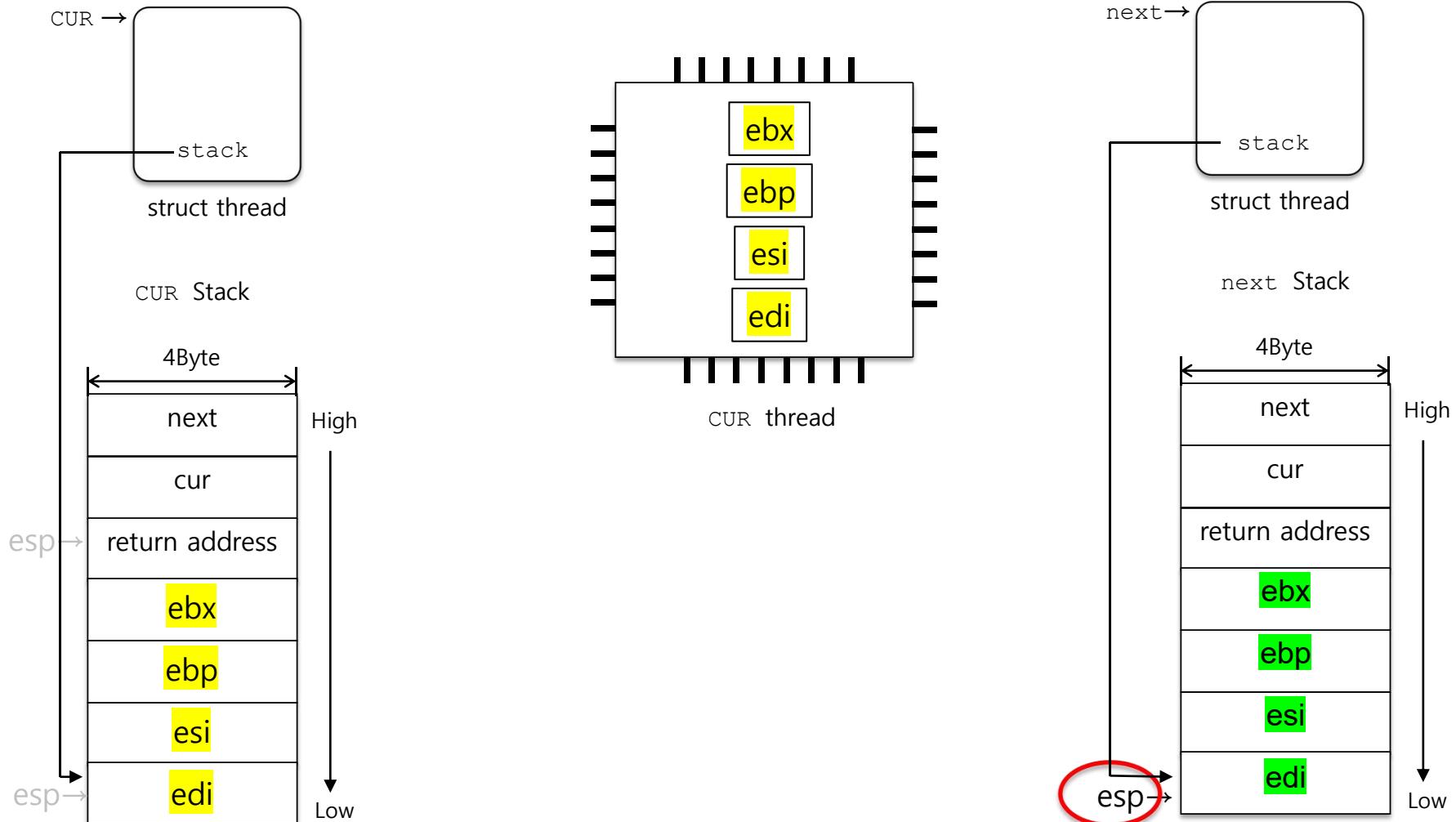
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)

movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

popl %edi
popl %esi
popl %ebp
popl %ebx
ret
```



Switch kernel stack



switch_threads

- Load address of the struct thread of NEXT to ecx.

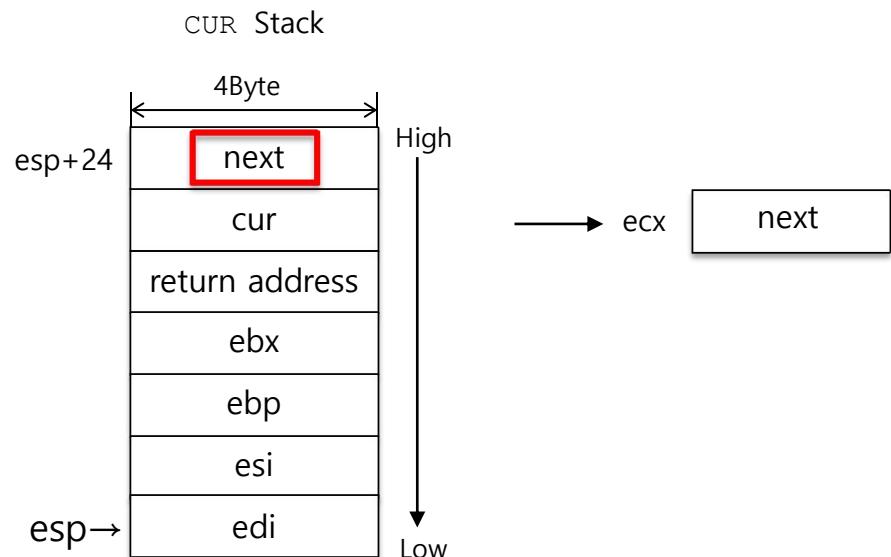
thread/switch.S

```
switch_threads:  
    pushl %ebx  
    pushl %ebp  
    pushl %esi  
    pushl %edi  
  
.globl thread_stack_ofs  
    mov thread_stack_ofs, %edx  
  
    movl SWITCH_CUR(%esp), %eax  
    movl %esp, (%eax,%edx,1)  
  
    movl SWITCH_NEXT(%esp), %ecx  
    movl (%ecx,%eax,1), %esp  
  
    popl %edi  
    popl %esi  
    popl %ebp  
    popl %ebx  
    ret
```

thread/switch.h

```
#define SWITCH_NEXT 24
```

$$\text{SWITCH_NEXT}(\%esp) = \%esp + 24$$



switch_threads

- Stack switch: Load address of the kernel stack of NEXT to esp.

thread/switch.S

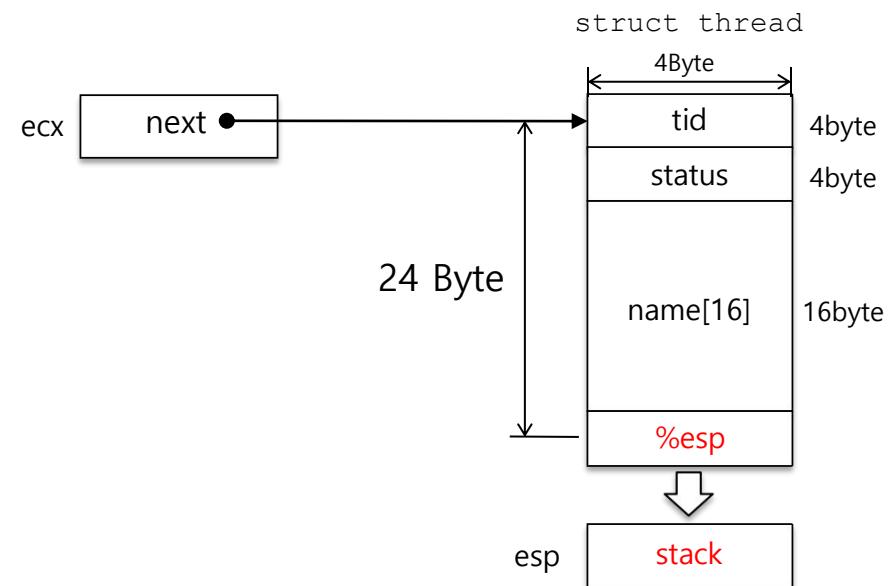
```
.globl thread_stack_ofs
mov thread_stack_ofs, %edx

movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)

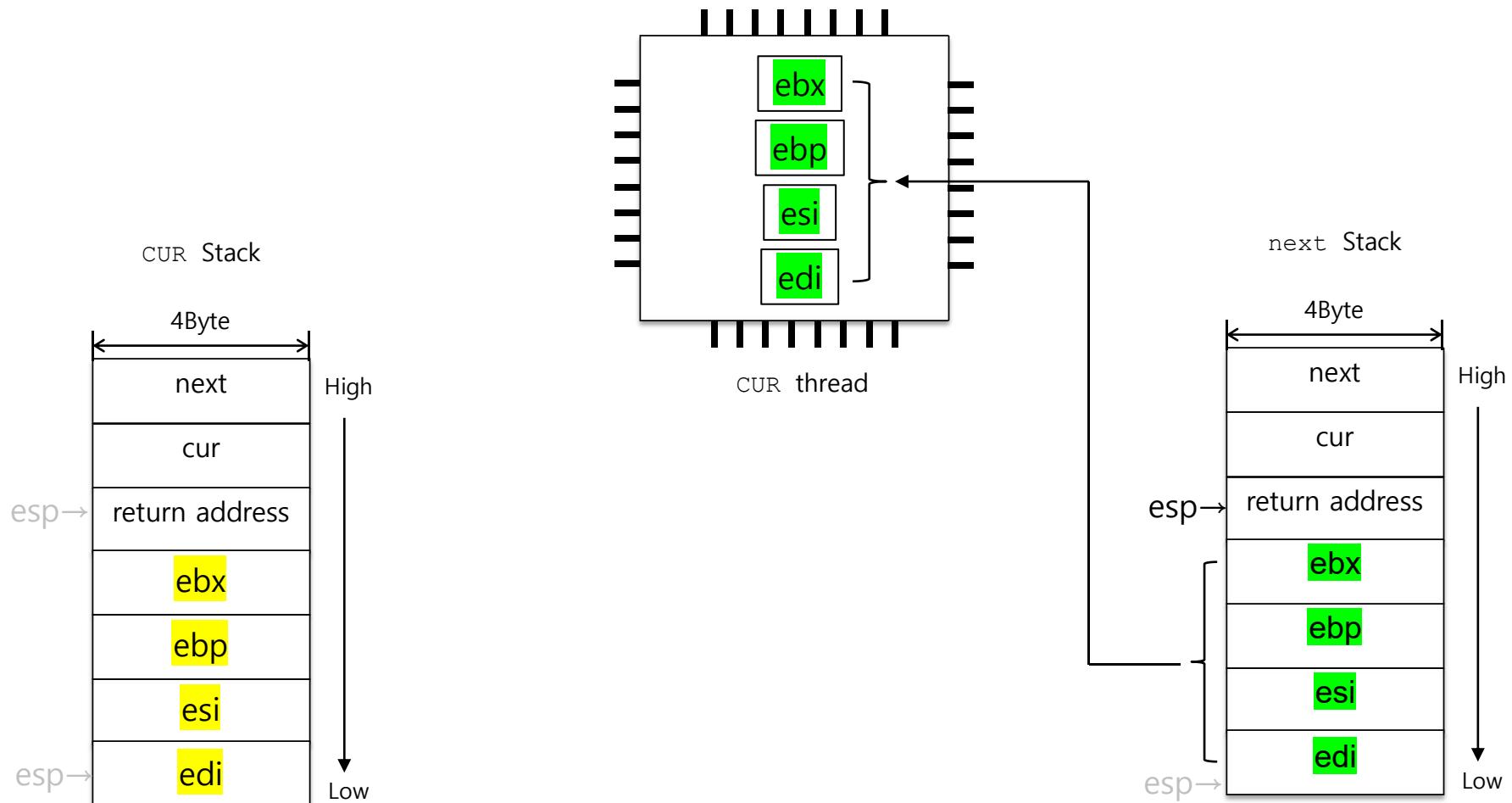
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

popl %edi
popl %esi
popl %ebp
popl %ebx
ret
```

$$(\%ecx, \%edx, 1) = \%ecx + \%edx * 1$$



Restore the new context

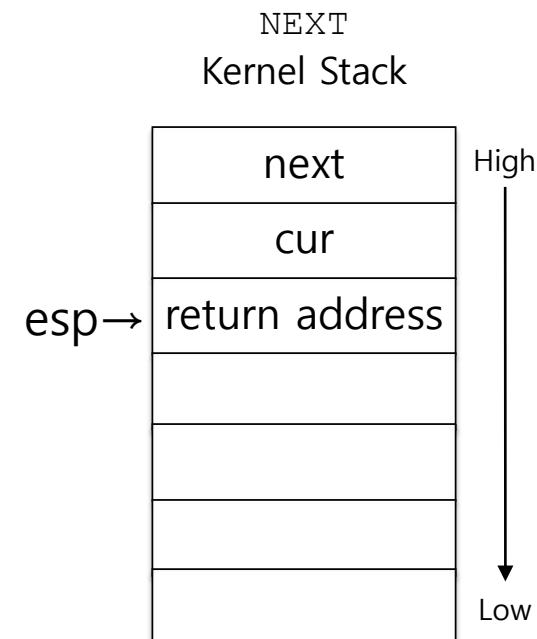


switch_threads

- Restore next thread's registers.
- Return and run instruction of return address.

thread/switch.S

```
switch_threads:  
    pushl %ebx  
    pushl %ebp  
    pushl %esi  
    pushl %edi  
  
.globl thread_stack_ofs  
    mov thread_stack_ofs, %edx  
  
    movl SWITCH_CUR(%esp), %eax (return value)  
    movl %esp, (%eax,%edx,1)  
  
    movl SWITCH_NEXT(%esp), %ecx  
    movl (%ecx,%edx,1), %esp  
  
    popl %edi  
    popl %esi  
    popl %ebp  
    popl %ebx  
    ret
```



thread_schedule_tail()

- ❑ Mark the new thread as running.
- ❑ If the previous thread was in the dying state, then it also frees the page.

Change the state of new current

```
void thread_schedule_tail (struct thread *prev) {  
    struct thread *cur = running_thread ();  
  
    ASSERT (intr_get_level () == INTR_OFF);  
  
    cur->status = THREAD_RUNNING;  
    _____________________________________  
    thread_ticks = 0;  
  
#ifdef USERPROG  
process_activate ();  
#endif  
  
if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)  
{  
    ASSERT (prev != cur);  
    palloc_free_page (prev);  
}  
}
```

Summary

- ❑ schedule()
 - ◆ Called in exit, yield and block.
 - ◆ Get the new process to the CPU.
- ❑ Context switch
 - ◆ Save the context of the currently running thread to the stack.
 - ◆ Save the current stack top at the currently running struct thread.
 - ◆ Restore the stack top of the next thread to esp register.
 - ◆ Restore the context from the stack of the next thread to run.
- ❑ Change the state of the next process to running and frees the memory from the dying process.

Operating Systems Lab

Part 1: Threads



Youjip Won

Overview

- Three topics

- ◆ Alarm clock
- ◆ Priority scheduling
- ◆ Advanced scheduler

Alarm clock

Overview

❑ Main goal

```
timer_alarm(int ticks)
```

system call that wakes up a process in ticks amount of time.

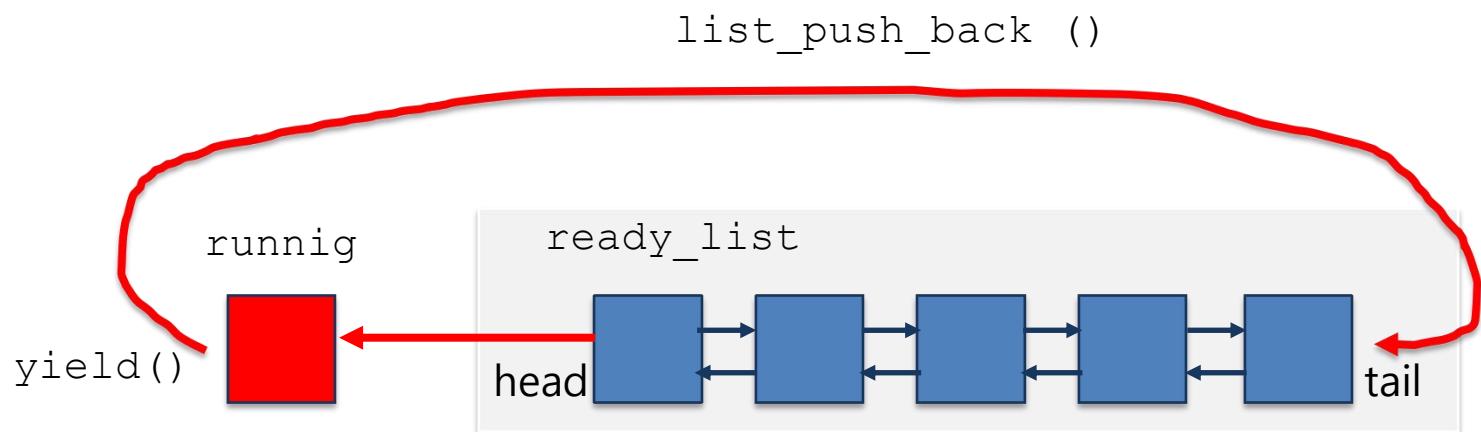
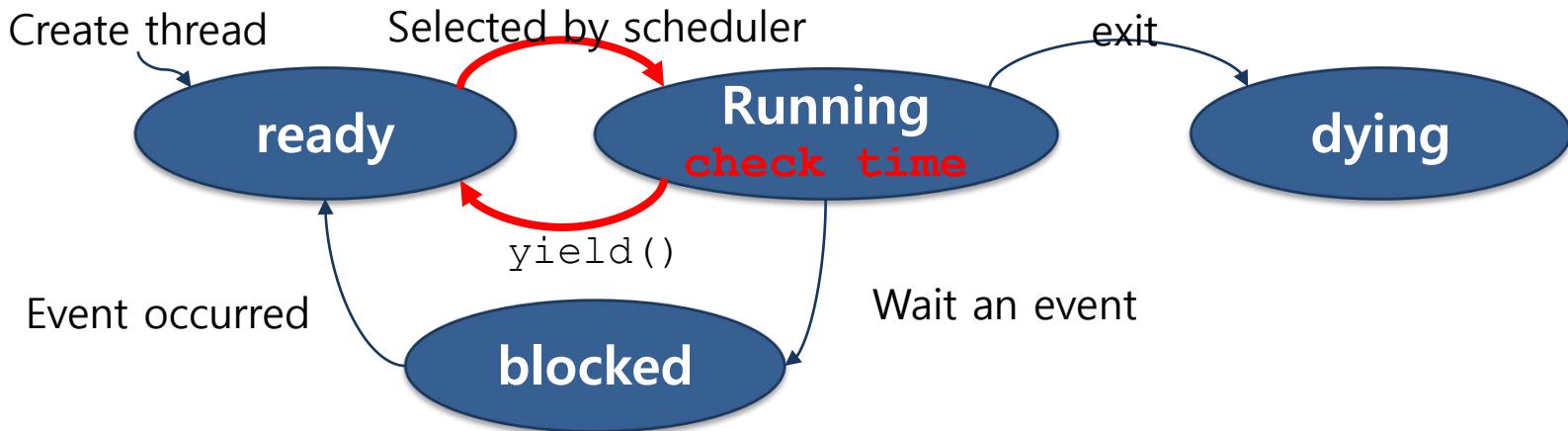
- ◆ Pintos uses busy waiting for alarm.
- ◆ Modify PintOS to use sleep/wakeup for alarm.

❑ Files to modify

- ◆ threads/thread.*
- ◆ devices/timer.*

timer_sleep() in current Pintos

- Keeps consuming CPU cycle



Sleep in original pintos

- Loop-based waiting up to the given tick.
- The thread that called this function is inserted to `ready_list` after the given tick.

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()` : yield the cpu and insert thread to `ready_list`.
- ◆ `timer_ticks()` : return the value of the current tick.
- ◆ `timer_elased()` : return how many ticks have passed since the `start`.

thread_yield()

- Yield the cpu and insert the thread to ready_list.

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 (); schedule ();
    intr_set_level (old_level);
}
```

functions in thread_yield()

□ Description of thread_yield()

thread_current()

- ◆ Return the current thread.

intr_disable()

- ◆ Disable the interrupt and return previous interrupt state.

intr_set_level(old_level)

- ◆ Set a state of interrupt to the state passed to parameter and return previous interrupt state.

list_push_back(&ready_list, &cur->elem)

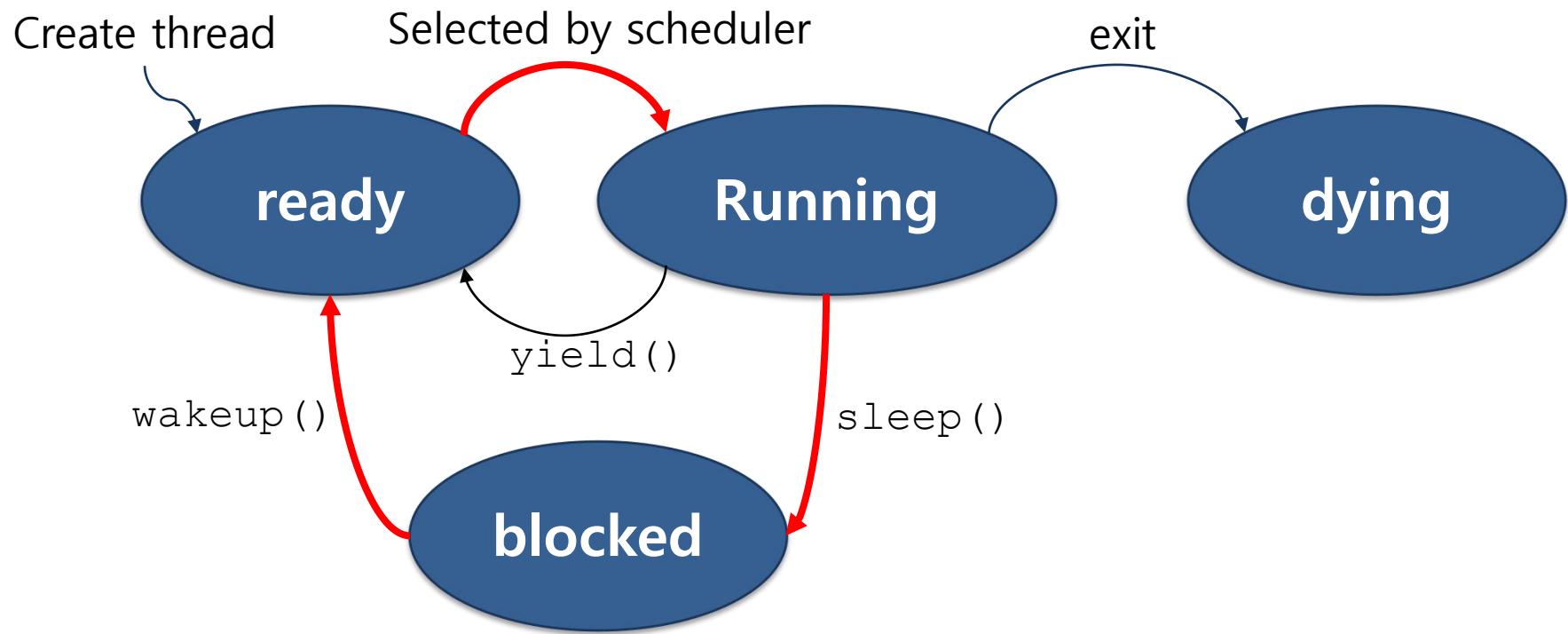
- ◆ Insert the given entry to the last of list.

schedule()

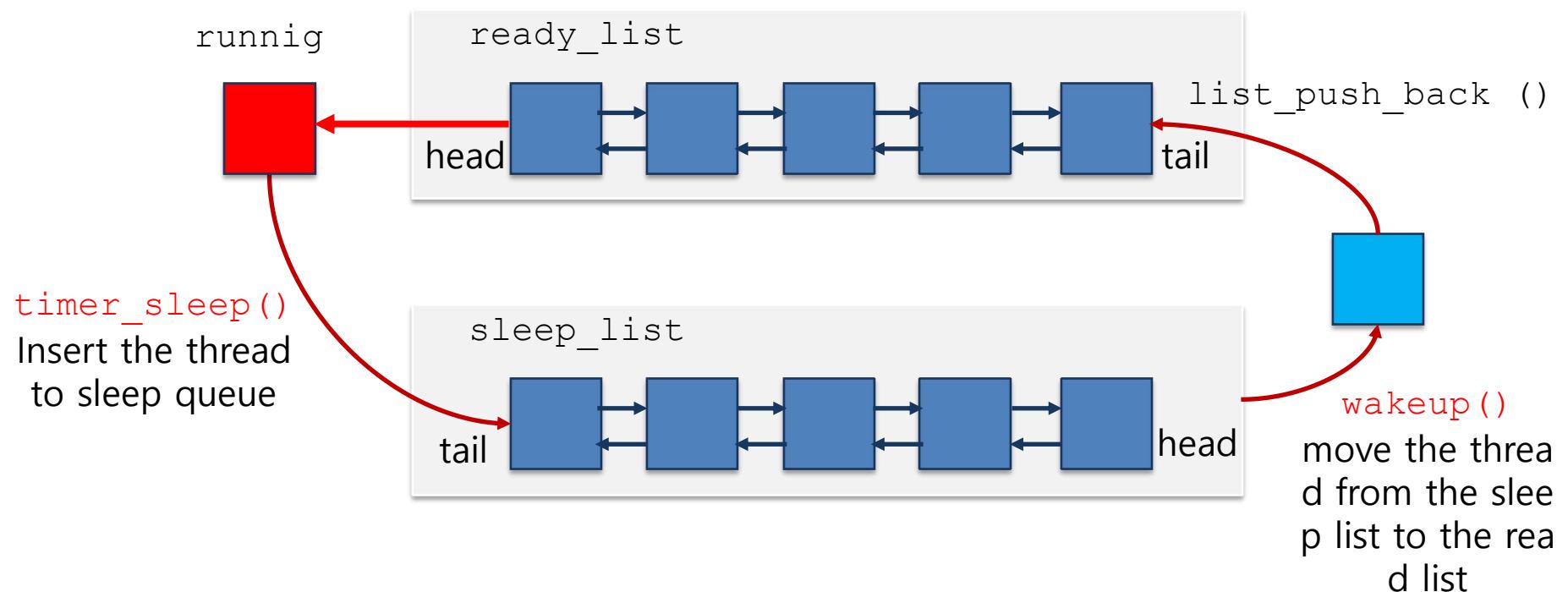
- ◆ Do context switch

Design: use 'blocked' state for new `timer_sleep()`

- Save CPU cycle and power consumption.



Design: Sleep/wakeup-based alarm clock



Implementation of Alarm Clock

- Define Sleep Queue.

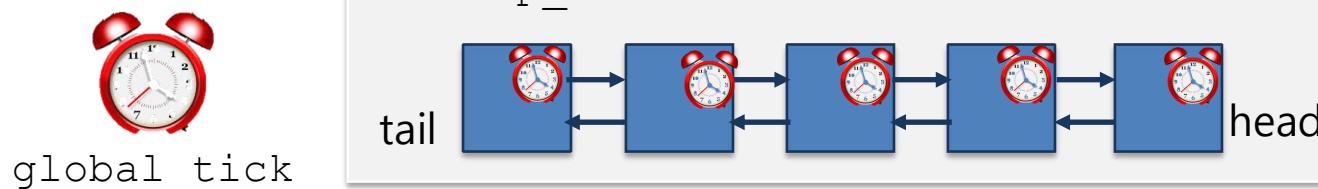
```
static struct list sleep_list;
```

- and initialize it.
- Point to think:

where to declare the list and when to initialize it.

Global tick vs. local tick

- ❑ Kernel (timer interrupt handler) needs to check which threads to wake up.
- ❑ local tick
 - ◆ Each thread needs to maintain the time to wakeup.
 - ◆ Modify the `thread` structure: store the time to wake up.
- ❑ “tick”, the global variable
 - ◆ the minimum value of local `tick` of the threads
 - ◆ Save the time to scan the sleep list
 - ◆ Don’t forget to initialize it.



Modify thread structure

- Add new field for local tick, e.g. `wakeup_tick`
- Use `int64` type.

`pintos/src/thread/thread.h`

```
struct thread
{
    ...
    /* tick till wake up */
    ...
}
```

Implementation of Alarm Clock

- Thread: Move thread (itself) to the sleep queue.
 - When `timer_sleep()` is called, check the tick.
 - If there is time left till the wakeup, remove the caller thread from `ready_list` and insert it to sleep queue.

Sample implementation: `pintos/src/devices/timer.c`

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

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

    if (timer_elapsed (start) < ticks)
        thread_sleep (start + ticks); //implement by yourself
}
```

Implementation of Alarm Clock

- Value of 'start' may become invalid at (2).
- Let's forget it for now.
 - Challenge: Think about how to fix it.

pintos/src/devices/timer.c

```
void timer_sleep (int64_t ticks)
{
    int64_t start = timer_ticks (); ----- (1)

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

    if (timer_elapsed (start) < ticks).----- (2)
        thread_sleep (start + ticks);
}
```

thread_sleep()

- Change the state of the caller thread to ‘blocked’ and put it to the sleep queue

.

pintos/src/threads/thread.c

```
void thread_sleep(int64_t ticks) {  
  
    /* if the current thread is not idle thread,  
       change the state of the caller thread to BLOCKED,  
       store the local tick to wake up,  
       update the global tick if necessary,  
       and call schedule() */  
    /* When you manipulate thread list, disable interrupt! */  
}
```

Implementation of Alarm Clock

- In the timer interrupt,

- ◆ Timer interrupt is heart of everything!.
- ◆ Determine which threads to wake up everytime when timer interrupt occurs.
- ◆ For the threads to wake up, remove them from the sleep queue and insert it to the `ready_list`. (Don't forget to change the state of the thread from sleep to ready!!!)

timer_interrupt()

pintos/src/devices/timer.c

```
static void timer_interrupt (struct intr_frame *args UNUSED)
{
    ticks++;
    thread_tick (); // update the cpu usage for running process

    /* code to add:
       check sleep list and the global tick.
       find any threads to wake up,
       move them to the ready list if necessary.
       update the global tick.
    */
}
```

Summary

❑ Functions to modify

- ◆ `thread_init()`
 - Add the code to initialize the sleep queue data structure.
- ◆ `timer_sleep()`
 - Call the function that insert thread to the sleep queue.
- ◆ `timer_interrupt()`
 - At every tick, check whether some thread must wake up from sleep queue and call wake up function.

Design tip for modularization

▫ Functions to add

1. The function that sets thread state to blocked and wait after insert it to sleep queue.
2. The function that find the thread to wake up from sleep queue and wake up it .
3. The function that save the minimum value of tick that threads have.
4. The function that return the minimum value of tick.

Result

```
$ 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.
```

- ◆ After removing the busy waiting (using sleep queue)

```
(alarm-multiple) thread 2: duration=50, iteration=6, 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.
```

- ◆ The idle tick was zero because it occupied the CPU even in the sleep state, but the idle tick increased after removing the busy waiting.

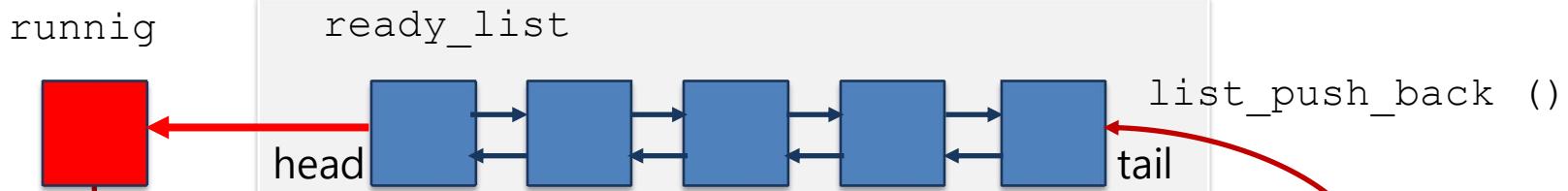
Priority Scheduling

Outline

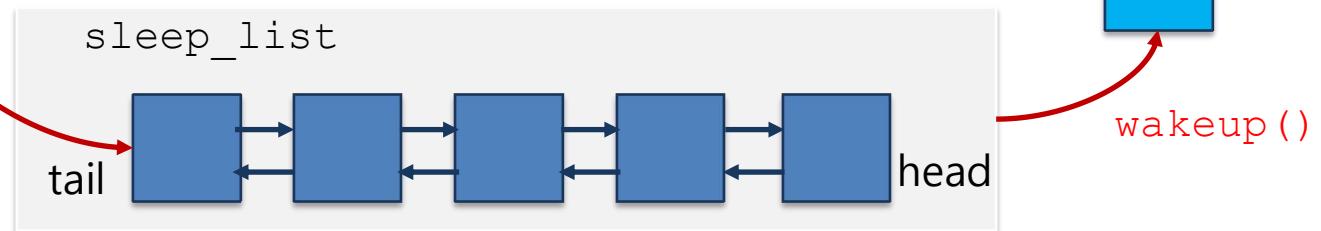
- ▣ Main goal
 - ◆ Pintos uses FIFO scheduling.
 - ◆ Modify PintOS scheduler for priority scheduling
 - Sort the ready list by the thread priority.
 - Sort the wait list for synchronization primitives(semaphore, condition variable).
 - Implement the preemption.
 - Preemption point: when the thread is put into the ready list (not everytime when the timer interrupt is called).
- ▣ Files to modify
 - ◆ threads/thread.*
 - ◆ threads/synch.*

Design: Sleep/wakeup-based alarm clock

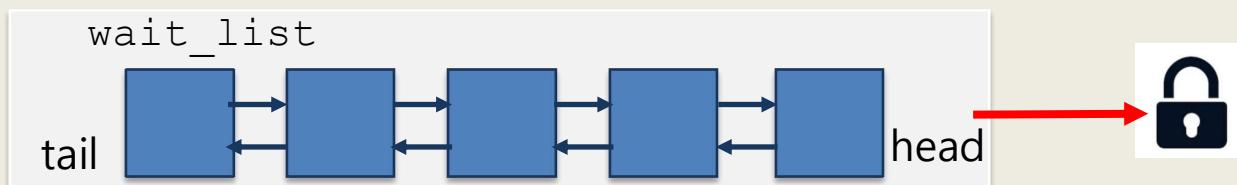
Run the thread with highest priority



`timer_sleep()`



`wakeup()`



get the thread with highest priority

Three things to consider

- When selecting a thread to run in the ready list, select the one with the highest priority.
- Preemption
 - ◆ When inserting the new thread to the ready list, compare the priority with the running thread.
 - ◆ Schedule the newly inserted thread if it has the higher priority with the currently running thread.
- Lock: semaphore, condition variable,
 - ◆ When selecting a thread from the set of threads waiting for a lock (or condition variable), select the one with the highest priority.

Priority in pintos

- ▣ Priority ranges from `PRI_MIN` (=0) to `PRI_MAX` (=63).
 - ◆ The larger the number, the higher priority.
 - ◆ Default is `PRI_DEFAULT` (=31)
- ▣ PintOS sets the initial priority when the thread is created by `thread_create()`
- ▣ Existing functions
 - ◆ `void thread_set_priority (int new_priority)`
 - Change priority of the current thread to `new_priority`
 - ◆ `int thread_get_priority (void)`
 - Return priority of the current thread.

Implementation of Priority Scheduling

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

- Point of updates

- ◆ Insert thread in `ready_list` in the order of priority. (note that it is not scalable)
- ◆ When the thread is added to the `ready_list`, compare priority of new thread and priority of the current thread.
- ◆ If the priority of the new thread is higher, call `schedule()` (the current thread yields CPU).

thread_create()

- When inserting a thread to `ready_list`, compare the priority with the currently running thread.
- If the newly arriving thread has higher priority, preempt the currently running thread and execute the new one.

pintos/src/threads/thread.c

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

    /* compare the priorities of the currently running thread and the newly inserted one. Yield the CPU if the newly arriving thread has higher priority*/
    return tid;
}
```

Others to modify

- ▣ `void thread_unblock(struct thread *t)`
 - ◆ When the thread is unblocked, it is inserted to `ready_list` in the priority order.
- ▣ `void thread_yield(void)`
 - ◆ The current thread yields CPU and it is inserted to `ready_list` in priority order.
- ▣ `void thread_set_priority(int new_priority)`
 - ◆ Set priority of the current thread.
 - ◆ Reorder the `ready_list`

Hint: thread_unblock (happy holiday~!^^\^)

- `thread_unblock()`
 - When unblocking a thread, use `list_inert_ordered` instead of `list_push_back`.

`pintos/src/threads/thread.c`

```
void thread_unblock (struct thread *t)
{
    ...
    //list_push_back (&ready_list, &t->elem); delete
    list_insert_ordered(& ready_list, & t-> elem, add
                        cmp_priority, NULL);
    t->status = THREAD_READY;
    intr_set_level (old_level);
}
```

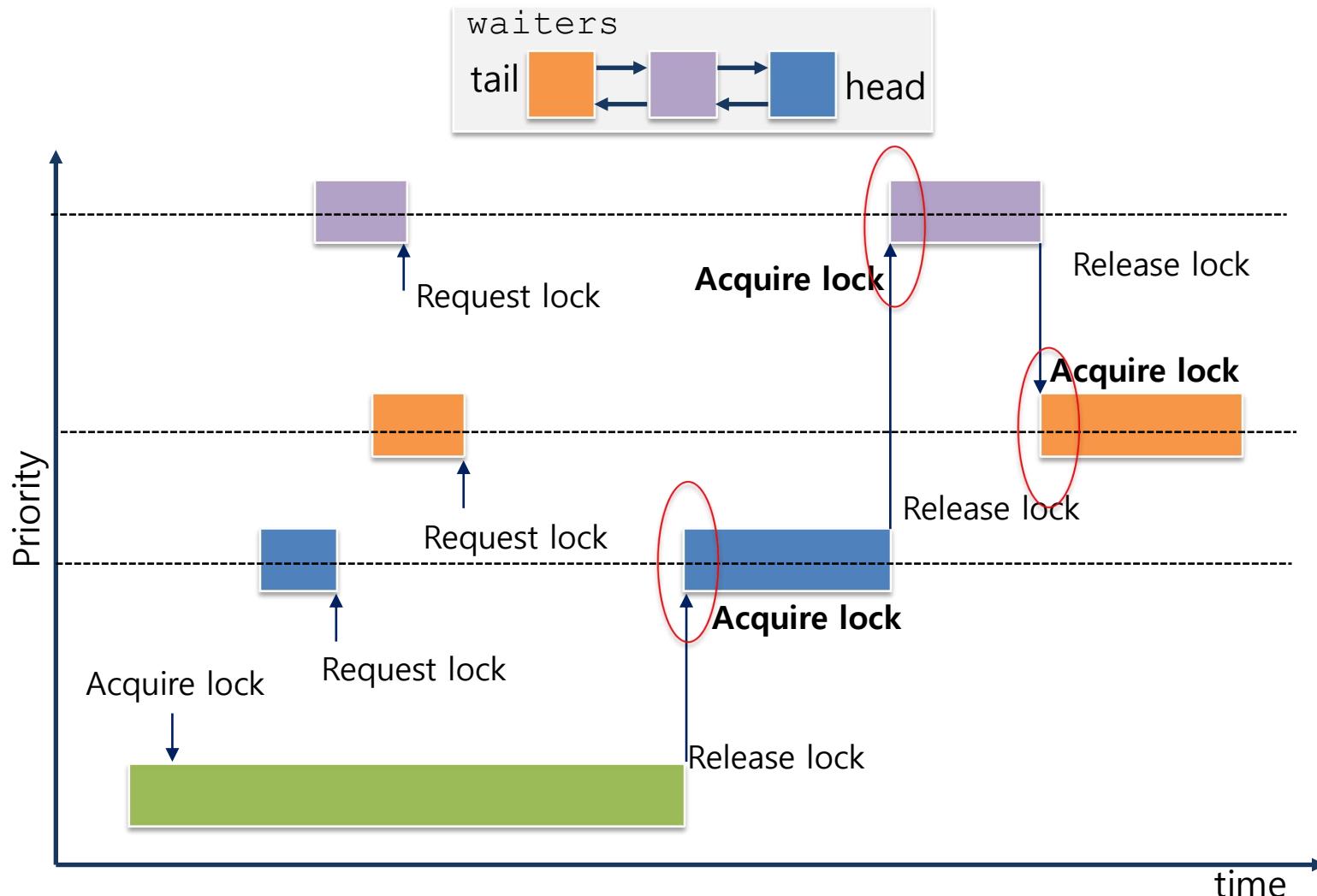
Change the synchronization primitives

- ❑ Lock
- ❑ Semaphore
- ❑ Condition variables

Wake up the waiting thread with respect to the thread's priority.

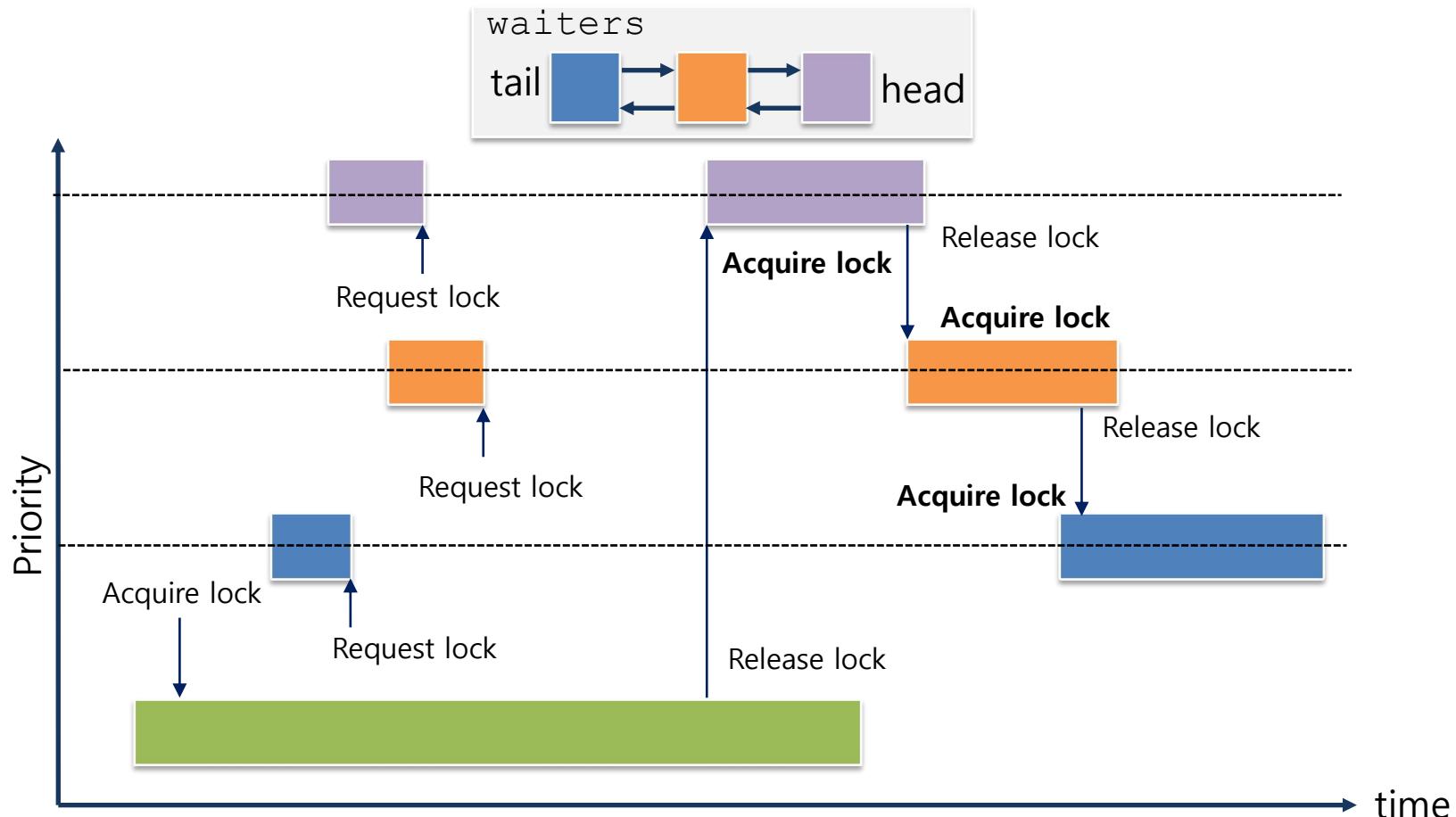
FIFO lock/unlock in priority-less Pintos

- Lock is acquired by FIFO order in `waiters` list, ignoring priority.



Priority-based lock/unlock

- When threads try to acquire semaphore, sort `waiters` list in order of priority.
 - Modify `sema_down()` / `cond_wait()`



Semaphore in pintos

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)`

- ♦ Initialize semaphore to the given value

`void sema_down(struct semaphore *sema)`

- ♦ Request the semaphore. If it acquired the semaphore, decrease the value by 1

`void sema_up(struct semaphore *sema)`

- ♦ Release the semaphore and increase the value by 1

Lock in pintos

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)`

- ◆ Initialize the lock data structure.

`void lock_acquire (struct lock *lock)`

- ◆ Request the lock.

`void lock_release (struct lock *lock)`

- ◆ Release the lock.

Condition variable in pintos

pintos/src/threads/synch.h

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

`void cond_init(struct condition *cond)`

- ◆ Initialize the condition variable data structure.

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

- ◆ Wait for signal by the condition variable.

`void cond_signal(struct condition *cond,`

`struct lock *lock UNUSED)`

- ◆ Send a signal to thread of the highest priority waiting in the condition variable.

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

)

- ◆ Send a signal to all threads waiting in the condition variable.

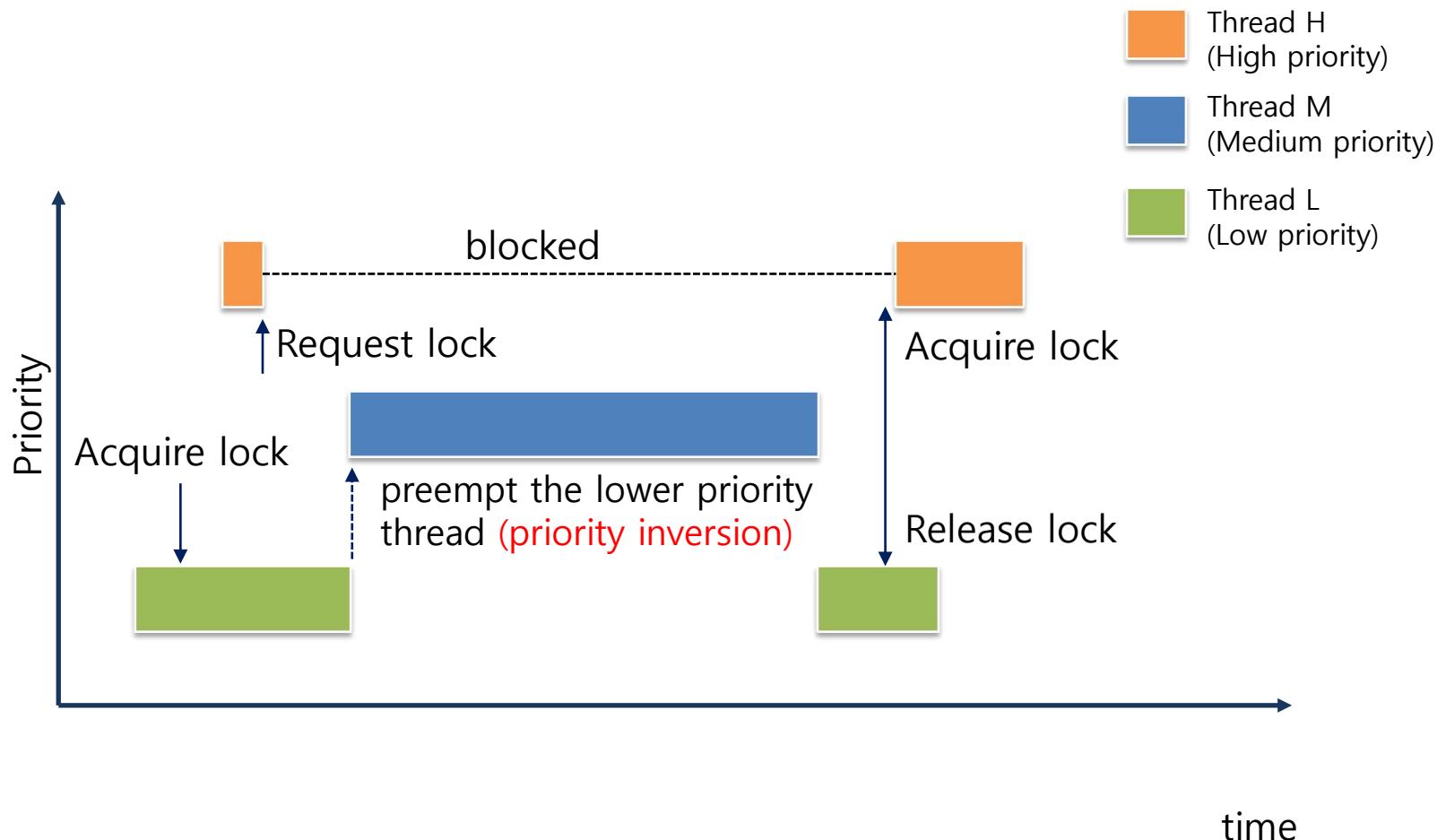
Implementation of Priority Scheduling-Synchronization

- Functions to modify.

- ◆ Modify to insert thread at `waiters` list in order of priority
 - `void sema_down(struct semaphore *sema)`
 - `void cond_wait(struct condition *cond, struct lock *lock)`
- ◆ Sort the `waiters` list in order of priority
 - It is to consider the case of changing priority of threads in `waiters` list.
 - `void sema_up(struct semaphore *sema)`
 - `void cond_signal(struct condition *cond,`
`struct lock *lock UNUSED)`

Priority Inversion

- The situation where thread of the higher priority waits thread of the lower priority.



In 1997, Pathfinder on Mars has stopped. OS has crashed due to the priority inversion.



- The Mars Pathfinder Mission Status Reports — First Week
- [The Mars Pathfinder Mission Status Reports — Second Week](#)
- The Mars Pathfinder Mission Status Reports — Third Week
- What really happened on Mars?
- [A Conversation with Glenn Reeves](#)

How did NASA remotely fix the code on the Mars Pathfinder?



In 1997, NASA remotely fixed a bug that caused priority inversion on their Mars Pathfinder. How did they go about doing this? What kind of communication protocols are used? How do they update the source for an operating system, compile it, and run it from a remote location? This might be simpler than I thought, but to me this seems like quite the feat!

asked :

viewed :

active :



Story of the bugfix here: http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

Linked



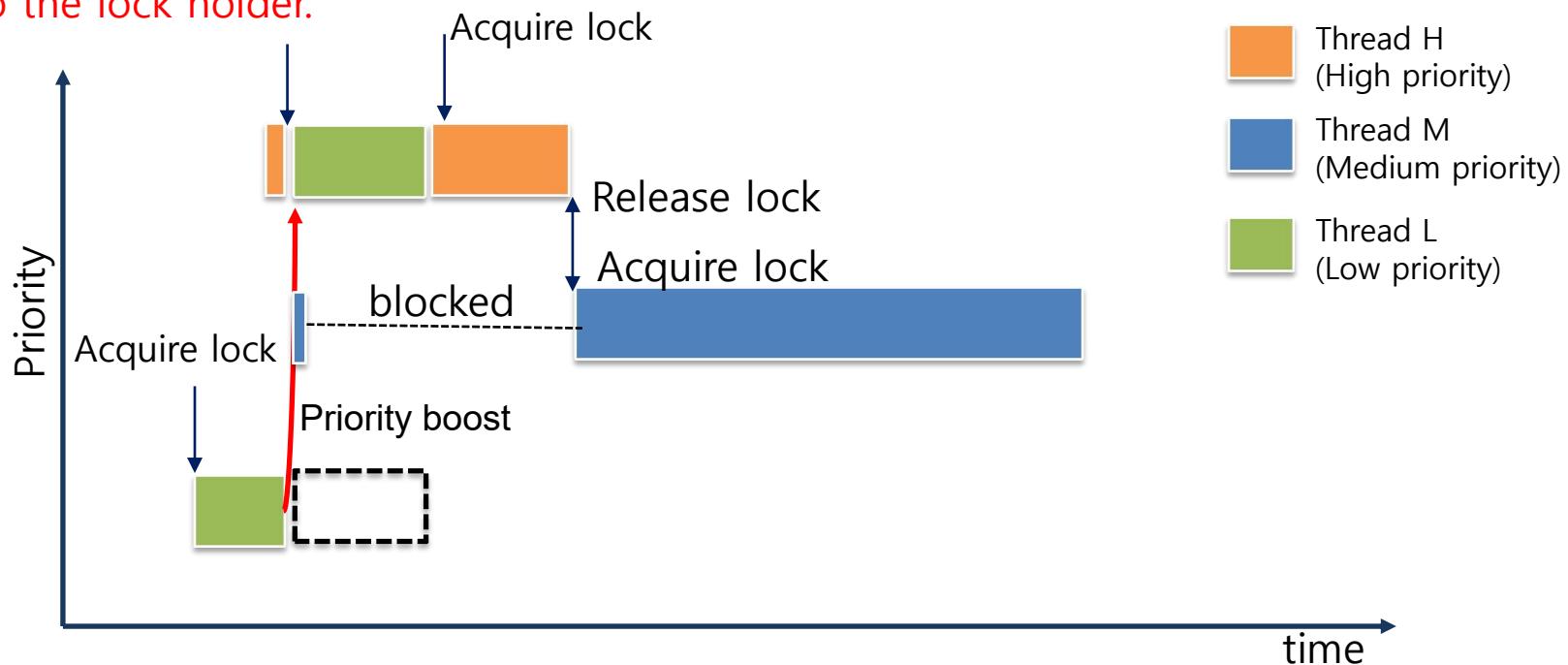
5 The author said to email him and he would provide details, but this was almost 20 years ago. Curious to see if anyone else knows how this worked.

2 E
S

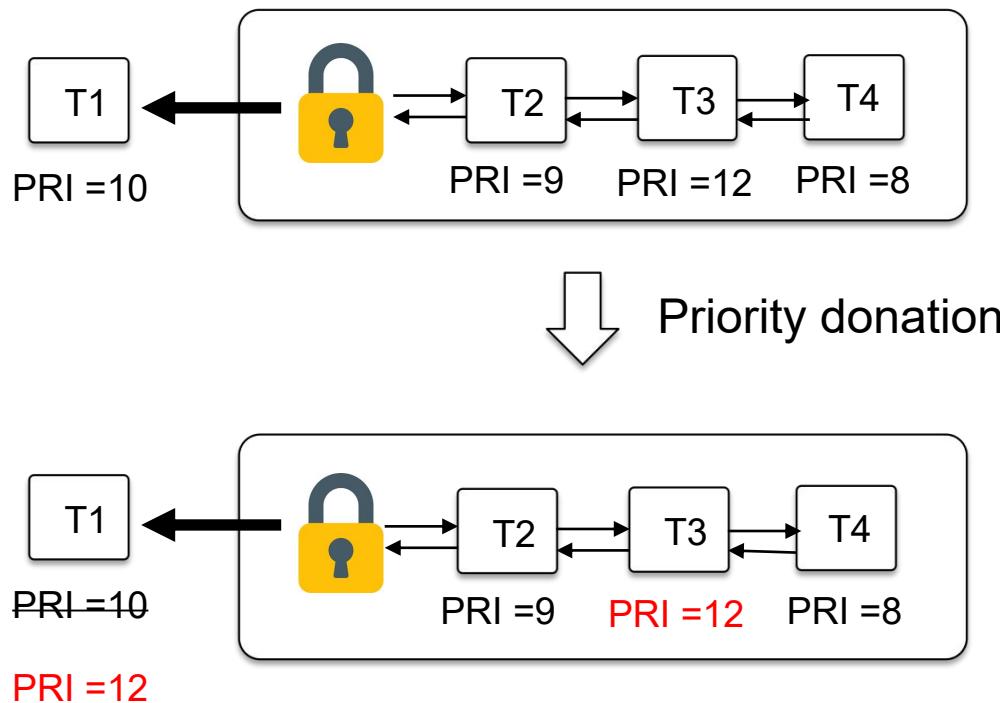
Priority Donation

- Inherit its priority to the lock holder.

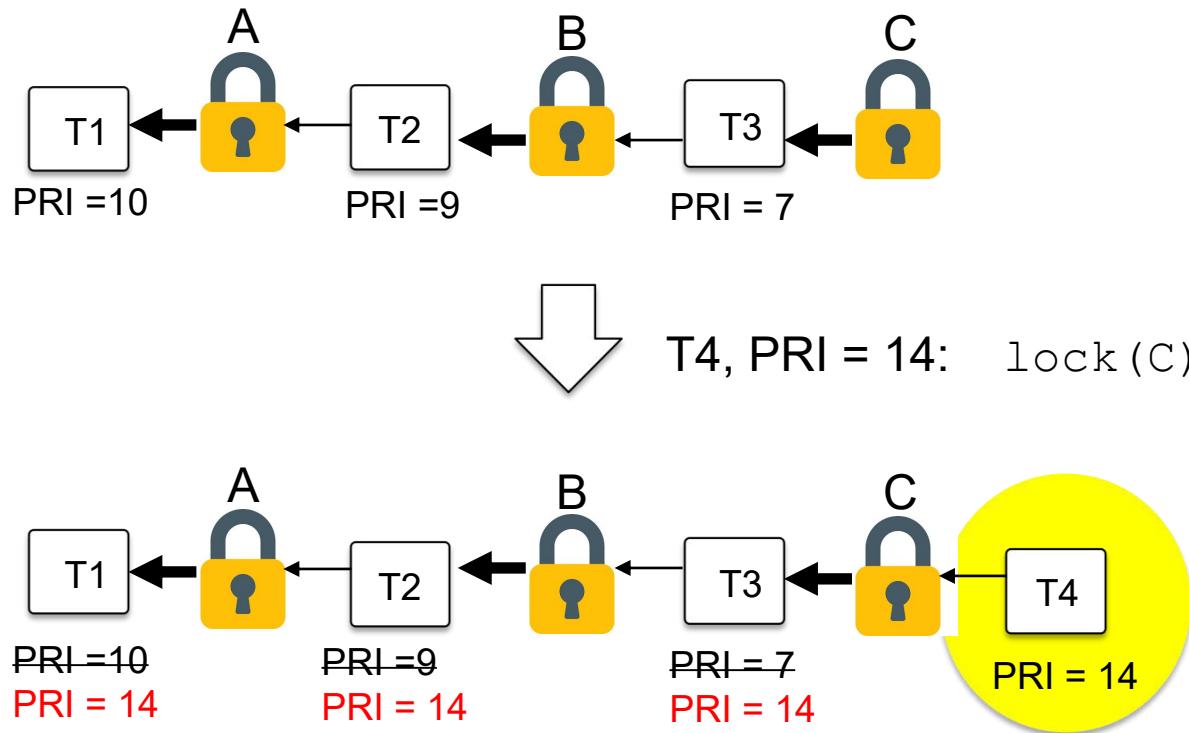
Request lock and inherit its priority to the lock holder.



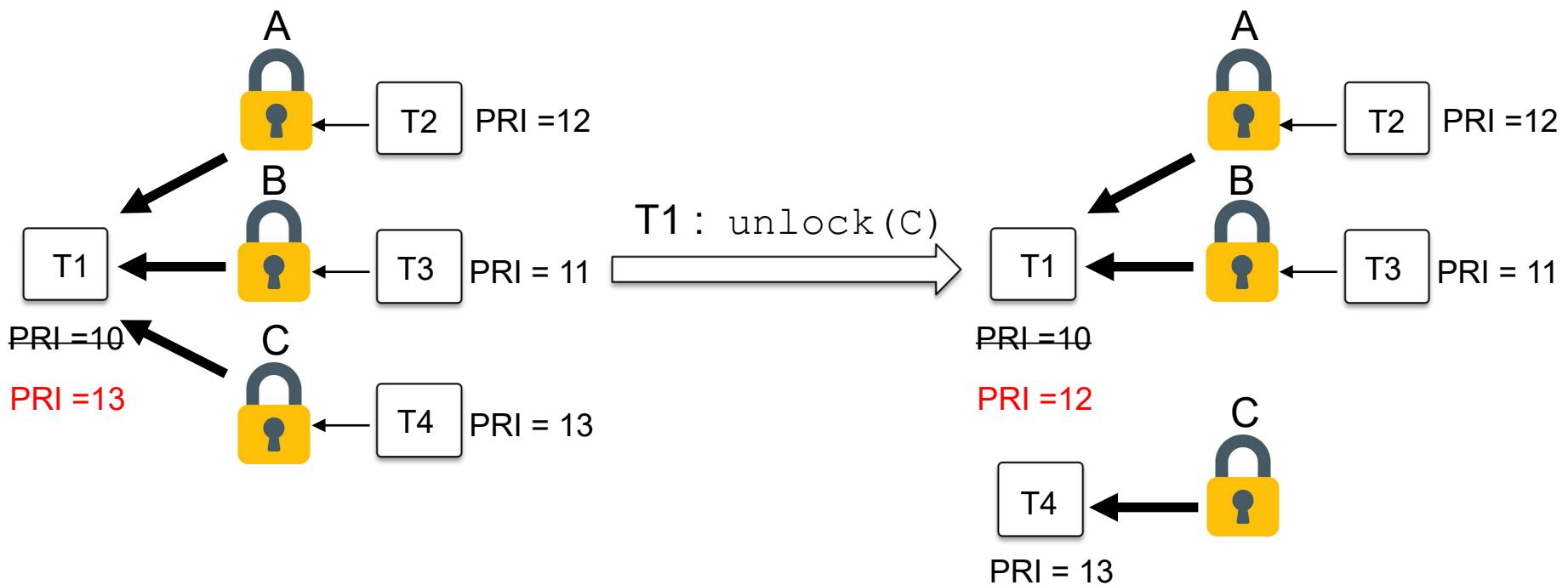
Priority Donation



Nested Donation

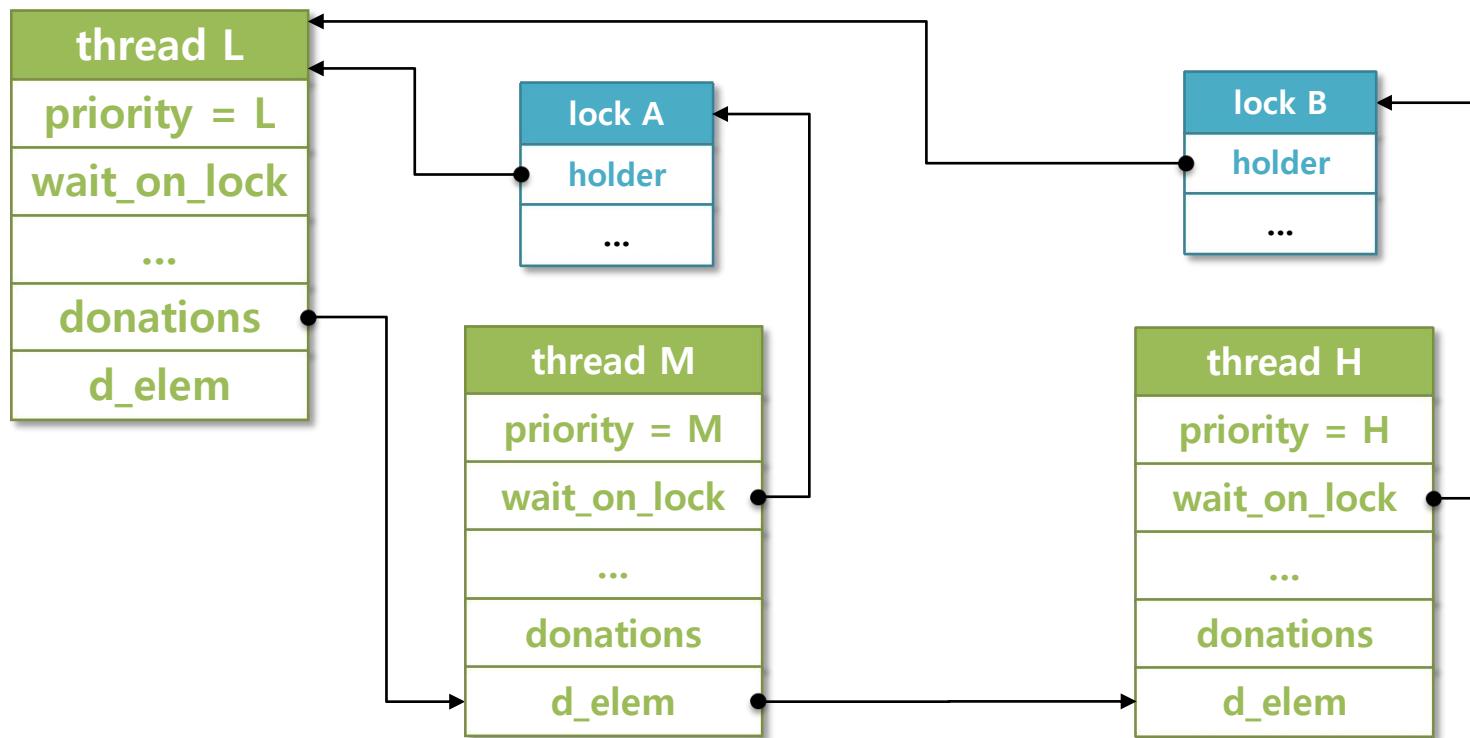


Multiple Donation



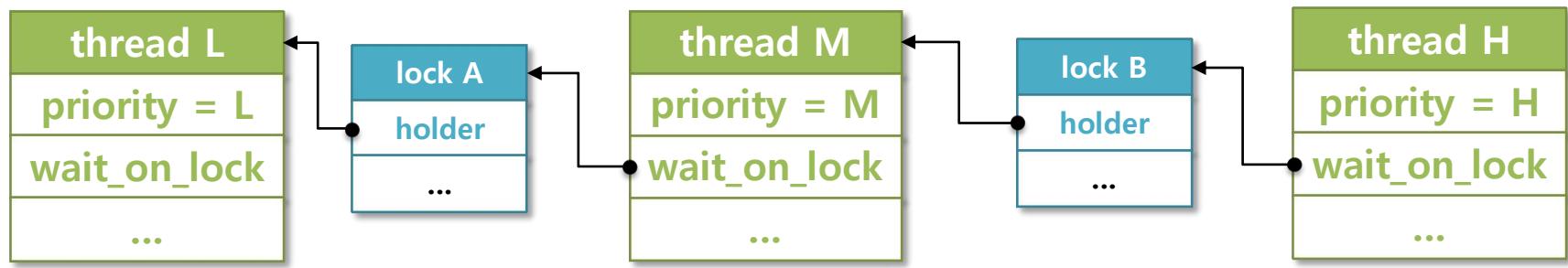
Data Structure for Multiple Donation

- Donations: list of Donors



Data Structure for nested donation

- wait_on_lock: lock that it waits for



Implementation of Priority Donation

Functions to modify

- ◆ `static void init_thread(struct thread *t,
 const char *name, int priority)`
 - Initializes data structure for priority donation.
- ◆ `void lock_acquire(struct lock *lock)`
 - If the lock is not available, store address of the lock.
 - Store the current priority and maintain donated threads on list (multiple donation).
 - Donate priority.
- ◆ `void lock_release(struct lock *lock)`
 - When the lock is released, remove the thread that holds the lock on donation list and set priority properly.
- ◆ `void thread_set_priority(int new_priority)`
 - Set priority considering the donation.

Result

```
$ 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.
```

Operating Systems Lab

Part 1: Threads



Youjip Won

4.4BSD like scheduler

Outline of Advanced Scheduler

❑ Main Goal

- ◆ Implement 4.4 BSD scheduler MLFQ like scheduler without the queues.
 - Give priority to the processes with interactive nature.
 - Priority based scheduler
- ◆ Use “equation”.

❑ Files to modify

- ◆ threads/thread.*
- ◆ devices/timer.c

❑ Nice value

- ◆ Represents the ‘niceness’ of a thread.
- ◆ If a thread is nicer, it is willing to give up some of its CPU time.

❑ Value (from -20 to 20)

- ◆ Nice (0) : not influence on priority. (initial value)
- ◆ Nice (positive) : decrease priority.
- ◆ Nice (negative) : increase priority.

❑ Function

- ◆ `thread_get_nice()`
- ◆ `thread_set_nice(int new_nice)`

□ Priority

- ◆ From 0 (PRI_MIN) to 63 (PRI_MAX)
- ◆ The larger the number, the higher the priority.
- ◆ It initialized when thread is created (default: 31)

```
priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)
```

▣ Philosophy

- ◆ If the thread is nicer, lower the priority.
- ◆ If the thread have been using lots of CPU recently, lower the priority.
- ◆ For all threads, priority is recalculated once in every fourth clock tick.
- ◆ The result is truncated to its nearest integer.

CPU usage

□ Update the cpu usage

recent_cpu

- ◆ Increase the recent_cpu of the currently running process by 1 in every timer interrupt.
- ◆ Decay recent_cpu by decay factor in every second.

recent_cpu = decay * recent_cpu

- ◆ Adjust recent_cpu by nice in every second.

recent_cpu = recent_cpu + nice

- ◆ Putting them together,

```
recent_cpu = decay * recent_cpu + nice
```

Decay factor

- In SVR3

$$\text{decay} = \frac{1}{2}$$

- In BSD4.4

- ◆ In heavy load, decay is nearly 1.
- ◆ In light load, decay is 0.

```
decay = (2*load_average) / (2*load_average + 1)
```

load_average

load_average

```
load_avg = (59/60)*load_avg + (1/60)*ready_threads
```

- ◆ At booting, `load_avg` is initially set to 0.
- ◆ `ready_threads`: the number of threads in `ready_list` and `threads in executing` at the time of update. (except idle thread)

In summary,

In every fourth tick, recompute the priority of all threads

```
priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)
```

In every clock tick, increase the running thread's recent_cpu by one.

In every second, update every thread's recent_cpu

```
recent_cpu = decay * recent_cpu + nice, where
```

```
decay = (2*load_average) / (2*load_average + 1)
```

and

```
load_avg = (59/60)*load_avg + (1/60)*ready_threads
```

```

Priority = PRI_MAX - (recent_cpu/4) - (nice*2)
recent_cpu = (2*load_avg)/(2*load_avg+1)*recent_cpu + nice
load_avg = (59/60)*load_avg + (1/60)*ready_threads

```

nice = 0,
load_avg = 0

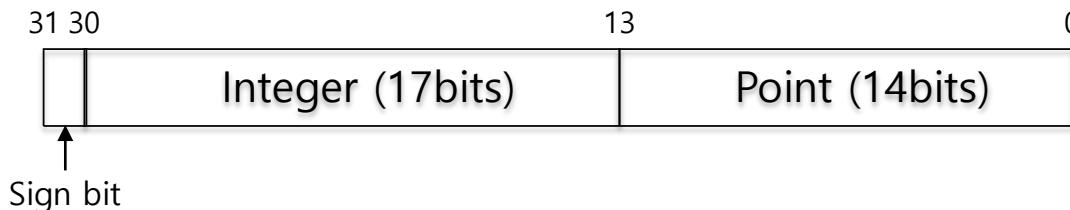
Priority
=PRI_MAX(63)-(4/4)-(0*2)
=62

Sec	Tick	P1		P2		P3	
		Priority	recent_cpu	Priority	recent_cpu	Priority	recent_cpu
0	0	63	0	63	0	63	0
	1	63	1	63	0	63	0
	2	63	2	63	0	63	0
	3	63	3	63	0	63	0
	4	62	4	63	0	63	0
	5	62	4	63	1	63	0
	6	62	4	63	2	63	0
	7	62	4	63	3	63	0
1	8	63	0	62	4	63	0
	9	63	0	62	4	63	1
	10	63	0	62	4	63	2
	11	63	0	62	4	63	3
	12	63	0	63	0	62	4
	13	63	1	63	0	62	4
	14	63	2	63	0	62	4
	15	63	3	63	0	62	4

 Running Priority recalculate Recent CPU recalculate

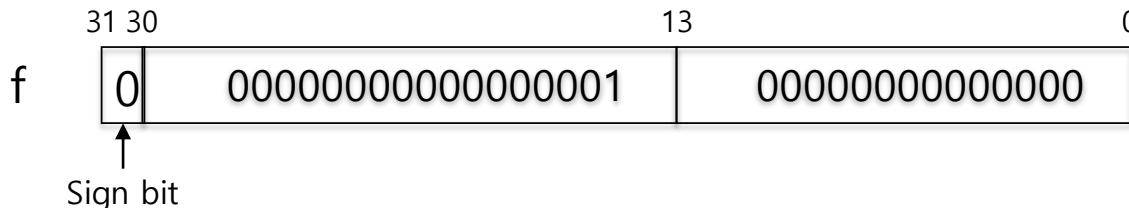
Implement fixed point arithmetic

- ▣ Inside kernel, you can do only integer arithmetic.
 - ◆ Kernel does not save floating point register when switching the context.
- ▣ We need to implement fixed point arithmetic using integer arithmetic.
 - ◆ priority, nice, ready_threads value is integer, and recent_cpu, load_avg value is real number.
 - ◆ Implement the fixed-point arithmetic using 17.14 fixed-point number representation.
 - Decimal point is 14 right-most bits.
 - Integer is 17 next bits to the left.
 - Last of left 1bit is sign bit.



Operations to implement

- o $n : \text{integer}$ $x, y : \text{fixed-point numbers}$ $f : 1 \text{ in } 17.14 \text{ format}$



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 :	$((\text{int64_t}) x) * y / f$
Multiply x by n :	$x * n$
Divide x by y :	$((\text{int64_t}) x) * f / y$
Divide x by n :	x / n

Examples

- Convert n to fixed point: $n * f$: shift n by 14 bits to the left.
- Convert x to integer: shift x by 14 to the right.

Implementation

- Each thread maintains
 - ◆ nice and recent_cpu
 - ◆ Add nice, recent_cpu to thread structure.
- Functions to be added
 - ◆ The function that calculate priority using recent_cpu and nice.
 - ◆ The function that calculate recent_cpu.
 - ◆ The function that calculate load_avg.
 - ◆ The function that increase recent_cpu by 1.
 - ◆ The function that recalculate priority and recent_cpu of all threads.
- Multiple ready queues vs. single ready queue
 - ◆ You can use single in implementing BSD scheduler.
 - ◆ **If you use multiple ready queues, you will get 10 extra mark for 100 mark.**

Functions to modify

- ◆ `static void init_thread(struct thread *t,
const char *name, int priority)`
 - Initialize `nice`, `recent_cpu`
- ◆ `void thread_set_priority(int new_priority)`
 - Disable the priority setting when using the advanced scheduler.
- ◆ `static void timer_interrupt(struct intr_frame *args UNUSED
)`
 - Recalculate `load_avg`, `recent_cpu` of all threads, priority every 1 sec.
 - Recalculate priority of all threads every 4th tick.

Functions to modify

- ◆ `void lock_acquire(struct lock *lock)`
 - Forbid the priority donation when using the advanced scheduler.

- ◆ `void lock_release(struct lock *lock)`
 - Forbid the priority donation when using the advanced scheduler.

Functions to modify

- ◆ `void thread_set_nice(int nice UNUSED)`
 - Set nice value of the current thread.
- ◆ `int thread_get_nice(void)`
 - Return nice value of the current thread.
- ◆ `int thread_get_load_avg(void)`
 - Return load_avg multiplied by 100
 - `timer_ticks() % TIMER_FREQ == 0`
- ◆ `int thread_get_recent_cpu(void)`
 - Return recent_cpu multiplied by 100

Result

```
$ 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.
```

Operating Systems Lab

Part 2: User Programs



Youjip Won

Overview

- Objective

Execute a user program in Pintos.

- Background

- Topics

- ◆ Parameter Passing
- ◆ System call infrastructure
- ◆ File manipulation

Background

To run a program

- Read the executable file from the disk.
 - ◆ Filesystem issue
- Allocate memory for the program to run.
 - ◆ Virtual memory allocation
- Pass the parameters to the program.
 - ◆ Set up user stack.
- Context switch to the user program
 - ◆ OS should wait for the program to exit.

Pintos filesystem

- Create virtual disk: in userprog/build

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

- ◆ filesys.dsk: partition name
- ◆ Filesystem size: 2MByte

- Format the disk

```
pintos -f -q
```

- Copy the file to the pintos filesystem

- ◆ -p: put, -g: get, -a: target filename

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

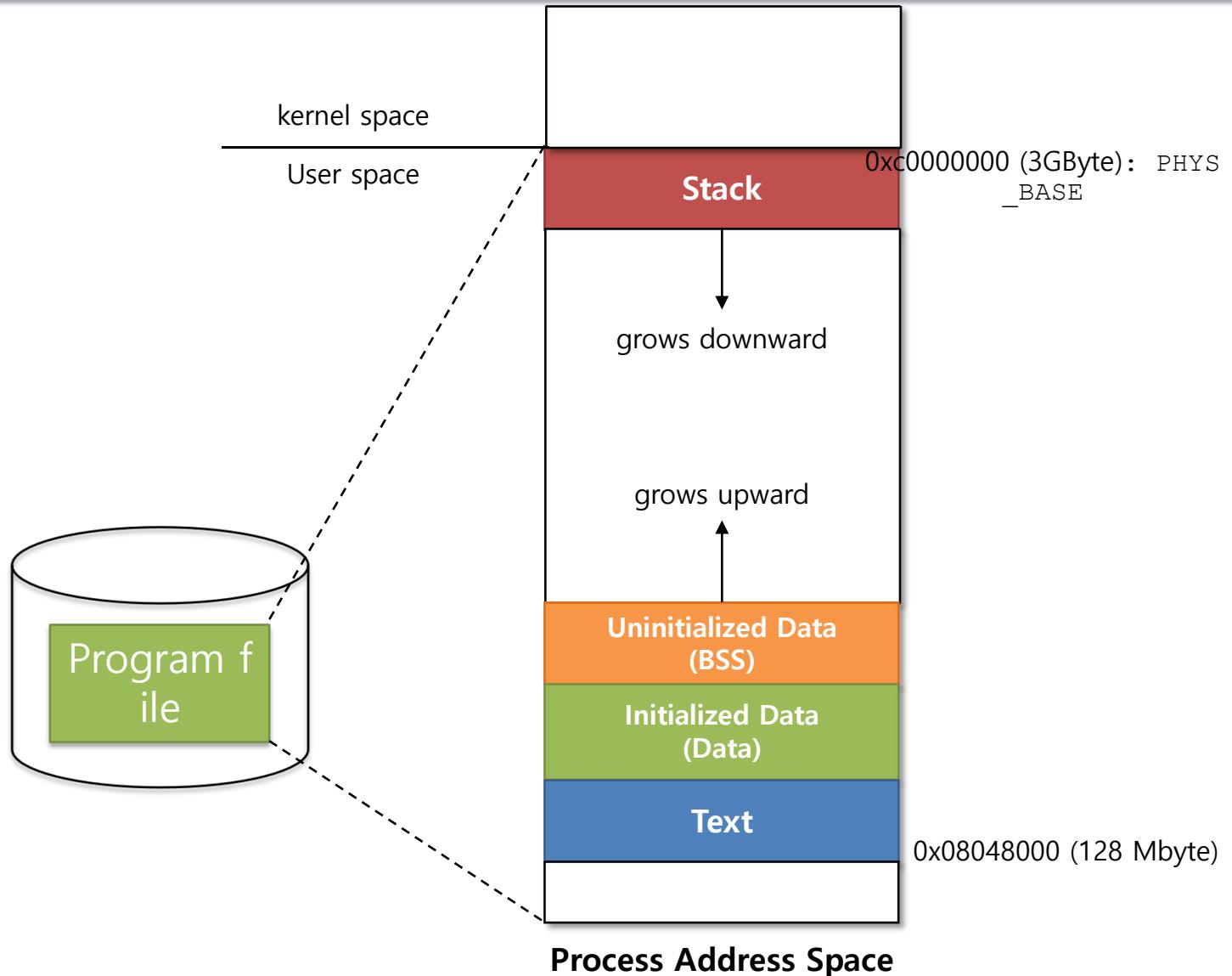
- Run the program

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

- Merge the last three lines into one

```
pintos -p ../../examples/echo -a echo -- -f -q run 'echo x'
```

Pintos VM layout



Running a program in pintos

Calling “process_execute”

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

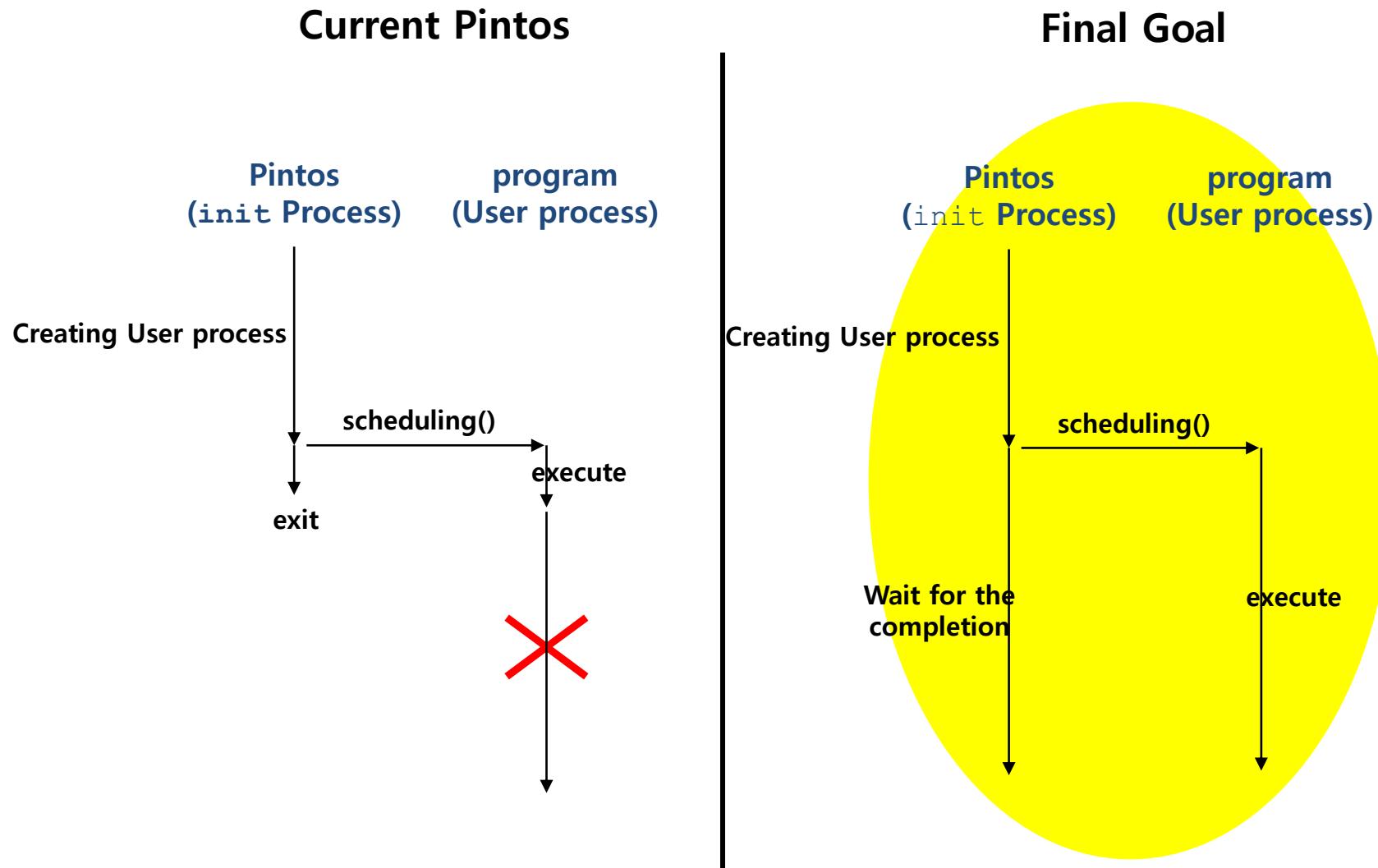
Create thread and start running a program

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

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

The OS quits without waiting for the process to finish!!!

Executing a program

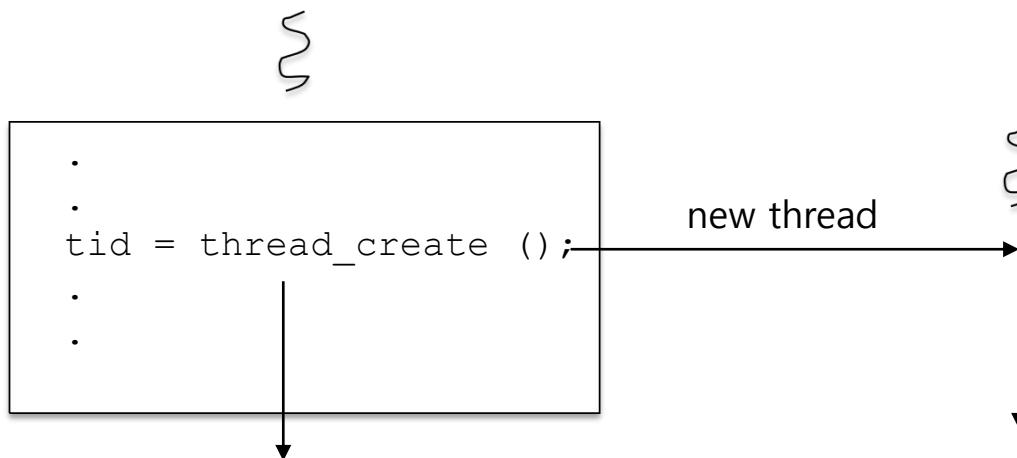


Executing a program

- Execute “file_name”.

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;
}
```



Creating a thread

- `thread_create()`
 - ◆ Create "struct thread" and initialize it.
 - ◆ Allocate the kernel stack.
 - ◆ Register the function to run: `start_process`.
 - ◆ Add it to ready list.

Creating a thread

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

```
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); /* allocating one page*/
    init_thread (t, name, priority); /* initialize thread structure*/
    tid = t->tid = allocate_tid (); /* allocate tid */
    /* Stack frame for kernel_thread(). */
    kf = alloc_frame (t, sizeof *kf);/* allocate stack */
    kf->eip = NULL;
    kf->function = function; /* function to run*/
    kf->aux = aux;           /* parameters for the function to run */
    ...
    /* Add to run queue. */
    thread_unblock (t);
    return tid;
}
```

Starting a process

- ◆ `load()`: load the program of name '`file_name`'
- ◆ If it successfully loads the program, run it. Otherwise, `exit()`.
- ◆ `thread_exit()` : quit the thread.

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    char *file_name = file_name_;
    ...
/* if_.esp: address of the top of the user stack */
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");
}
```

start_process

```
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_exi
t" : : "g" (&if_) : "memory");
}
```

```
bool load (const char *file_name, void (*
*eip) (void), void **esp)
{
    ...
    struct file *file = NULL;
    ...
    file = filesys_open (file_name);
    ...
    /* Set up stack. */
    if (!setup_stack (esp))
        ...
    success = true;
    return success;
}
```

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

Loading a program.

- ▣ Load a ELF file.
 - ◆ Create page table (2 level paging).
 - ◆ Open the file, read the ELF header.
 - ◆ Parse the file, load the 'data' to the data segment.
 - ◆ Create user stack and initialize it.

```

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 (); /* create page directory */
    process_activate (); /* set cr3 register*/
    file = filesys_open (file_name); /* Open the file*/
/* parse the ELF file and get the ELF header*/
    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)
/* load segment information */
    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)
    ...
/* load the executable file */
    if (!load_segment (file, file_page, (void *) mem_page,
                      read_bytes, zero_bytes, writable))
    ...
    if (!setup_stack (esp)) /* initializing user stack*/
        *eip = (void (*) (void)) ehdr.e_entry; /*initialize entry point*/
}

```

Passing the arguments and creating a thread

Overview

- ▣ For “echo x y z”
 - ◆ Original:
 - Thread name: “echo x y z”
 - Find program with file name “echo x y z”
 - Arguments “echo”, “x”, “y”, and “z” are not passed
 - ◆ After modification
 - Thread name: “echo”
 - Find program with file name “echo”
 - Push the arguments to user stack.
- ▣ Files to modify
 - ◆ pintos/src/userprog/process.*

Parse the arguments and push them to the user stack

- ❑ pintos/src/userprog/process.c

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

- ◆ Parse the string of file_name
- ◆ Forward first token as name of new process to thread_create() function

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

- ◆ Parse file_name
- ◆ Save tokens on user stack of new process.

Tokenizing

```
char *strtok_r (char *s, const char *delimiters,  
                char **save_ptr) /* string.h */
```

- ◆ Receive a string (s) and delimiters and parse them by delimiters

ex) Parsing a string by the first space

```
char s[] = "String to tokenize.";  
char *token, *save_ptr;  
for (token = strtok_r (s, " ", &save_ptr); token != NULL;  
     token = strtok_r (NULL, " ", &save_ptr))  
printf ("%s\n", token);
```

Result

```
'String'  
'to'  
'tokenize.'
```

Program Name

Thread name

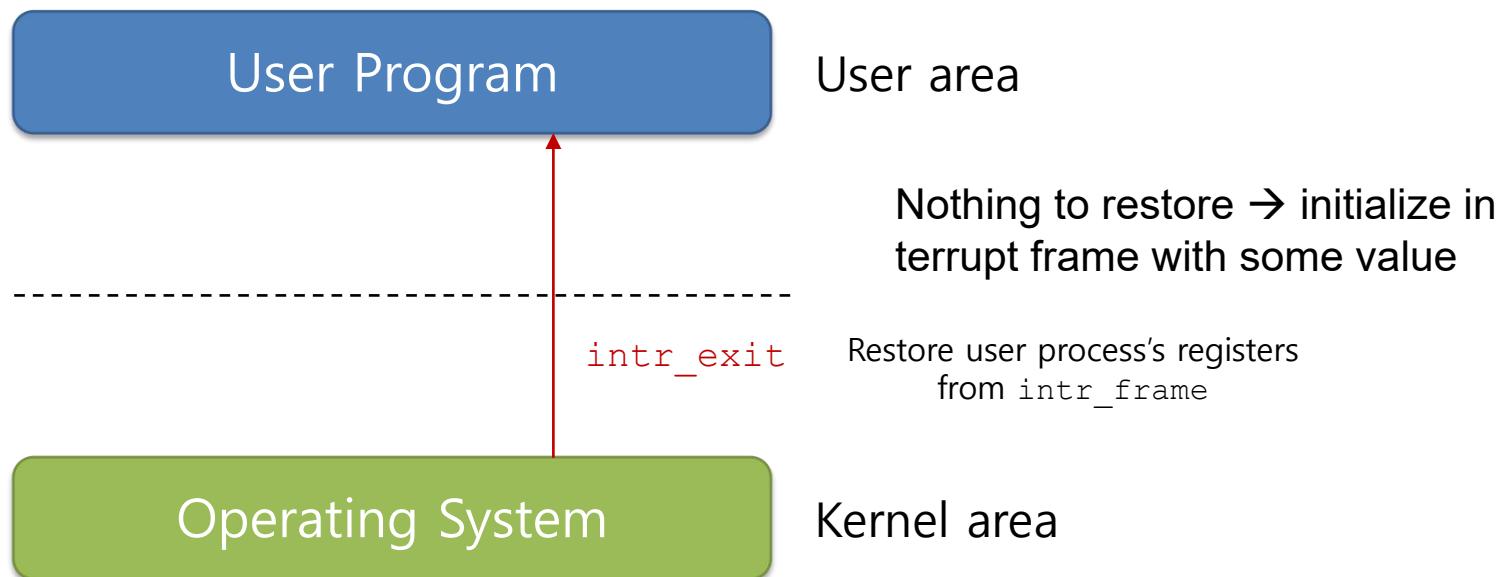
- Before: Entire command line is passed to `thread_create()`
- After modification: Forward only first token of command line to first argument of `thread_create()`
 - “echo x y z” → only use “echo” for name of process

pintos/src/userprog/process.c

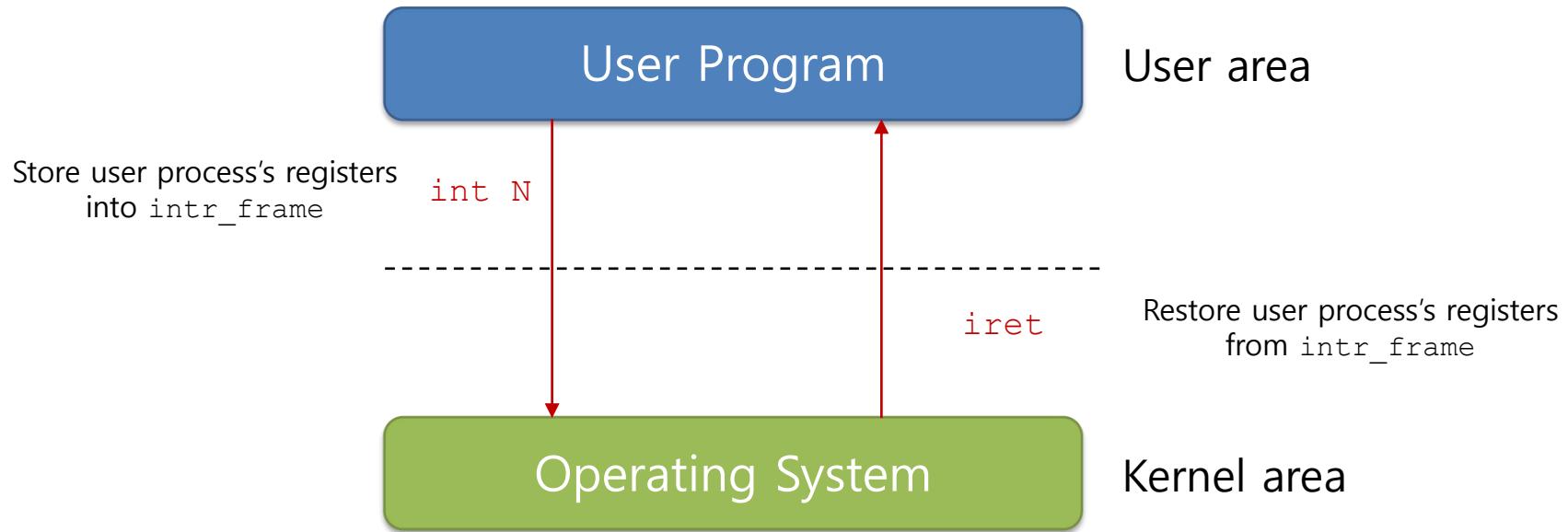
```
tid_t process_execute (const char *file_name)
{
    ...
    /* Parse command line and get program name */
    ...
    /* Create a new thread to execute FILE_NAME. */
    tid = thread_create (file_name, PRI_DEFAULT, start_process,
fn_copy);                                Name of thread
    ...
}
```

start_process

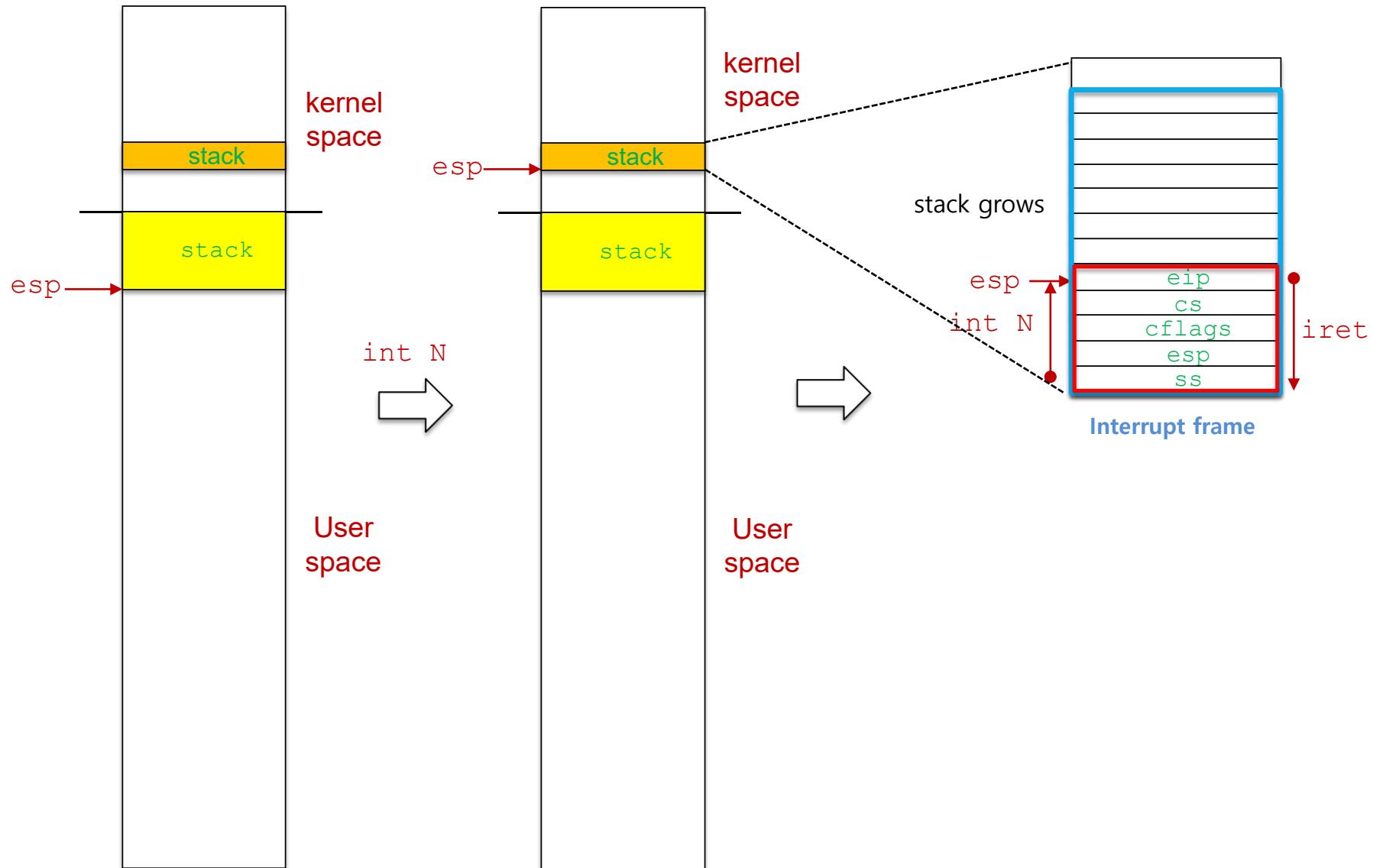
- Allocate interrupt frame.
- Load program and initialize interrupt frame and user stack.
- Setup arguments at the user stack.
- Jump to the user program through `interrupt_exit`.



Getting into and out of kernel



Getting into and out of kernel



struct intr_frame

```
struct intr_frame {  
    /* Pushed by intr_entry in intr-stubs.S.  
     * These are the interrupted task's saved registers. */  
    uint32_t edi;           /* Saved EDI. */  
    uint32_t esi;           /* Saved ESI. */  
    uint32_t ebp;           /* Saved EBP. */  
    uint32_t esp_dummy;     /* Not used. */  
    uint32_t ebx;           /* Saved EBX. */  
    uint32_t edx;           /* Saved EDX. */  
    uint32_t ecx;           /* Saved ECX. */  
    uint32_t eax;           /* Saved EAX. */  
    uint16_t gs, :16;        /* Saved GS segment register. */  
    uint16_t fs, :16;        /* Saved FS segment register. */  
    uint16_t es, :16;        /* Saved ES segment register. */  
    uint16_t ds, :16;        /* Saved DS segment register. */
```

```
    /* Pushed by intrNN_stub in intr-stubs.S. */  
    uint32_t vec_no;         /* Interrupt vector number. */  
  
    /* Sometimes pushed by the CPU,  
     * otherwise for consistency pushed as 0 by intrNN_stub.  
     * The CPU puts it just under `eip', but we move it here. */  
    uint32_t error_code;      /* Error code. */  
  
    /* Pushed by intrNN_stub in intr-stubs.S.  
     * This frame pointer eases interpretation of backtraces. */  
    void *frame_pointer;      /* Saved EBP (frame pointer). */
```

```
    /* Pushed by the CPU.  
     * These are the interrupted task's saved registers. */  
    void (*eip) (void);       /* Next instruction to execute. */  
    uint16_t cs, :16;          /* Code segment for eip. */  
    uint32_t eflags;           /* Saved CPU flags. */  
    void *esp;                 /* Saved stack pointer. */  
    uint16_t ss, :16;          /* Data segment for esp. */
```

```
}
```

- It is in the kernel stack.
- It stores user process' registers.

Stack grows.

Getting into kernel.

int n

- when execute the kernel function, e.g. interrupt handler, system call, the OS saves the registers of currently executing process.
- Where: at the kernel stack of the executing process.
- execution
 1. Set the esp to point to kernel stack
 2. Pushes registers.

Entering the kernel

```
struct intr_frame {  
    /* Pushed by intr_entry in intr-stubs.S.  
     * These are the interrupted task's saved registers. */  
    uint32_t edi;           /* Saved EDI. */  
    uint32_t esi;           /* Saved ESI. */  
    uint32_t ebp;           /* Saved EBP. */  
    uint32_t esp_dummy;     /* Not used. */  
    uint32_t ebx;           /* Saved EBX. */  
    uint32_t edx;           /* Saved EDX. */  
    uint32_t ecx;           /* Saved ECX. */  
    uint32_t eax;           /* Saved EAX. */  
    uint16_t gs, :16;        /* Saved GS segment register. */  
    uint16_t fs, :16;        /* Saved FS segment register. */  
    uint16_t es, :16;        /* Saved ES segment register. */  
    uint16_t ds, :16;        /* Saved DS segment register. */
```

← esp

After interrupt handler, int
r_entry

```
/* Pushed by intrNN_stub in intr-stubs.S. */  
uint32_t vec_no;          /* Interrupt vector number. */  
  
/* Sometimes pushed by the CPU,  
 * otherwise for consistency pushed as 0 by intrNN_stub.  
 * The CPU puts it just under `eip', but we move it here. */  
uint32_t error_code;      /* Error code. */  
  
/* Pushed by intrNN_stub in intr-stubs.S.  
 * This frame pointer eases interpretation of backtraces. */  
void *frame_pointer;      /* Saved EBP (frame pointer). */
```

← esp

After interrupt handler
of intr N

time

```
/* Pushed by the CPU.  
 * These are the interrupted task's saved registers. */  
void (*eip) (void);       /* Next instruction to execute. */  
uint16_t cs, :16;          /* Code segment for eip. */  
uint32_t eflags;           /* Saved CPU flags. */  
void *esp;                 /* Saved stack pointer. */  
uint16_t ss, :16;          /* Data segment for esp. */  
},
```

← esp

After int instruction.

Loading

Load the program

- ◆ Pass the program name to 'load()'.
- ◆ "Load()" find executable file, using name of file and load it onto memory.

pintos/src/userprog/process.c

```
static void start_process (void *file_name_)
{
    char *file_name = file_name_;
    struct intr_frame if_;
    bool success;
    ...
    /* Parse the command line (Use strtok_r()) */

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

program name Function entry point Stack top
 (user stack)

```

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

    /* Initialize interrupt frame and load executable. */
    memset (&if_, 0, sizeof if_);
    if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
    if_.cs = SEL_UCSEG;
    if_.eflags = FLAG_IF | FLAG_MBS;
success = load (file_name, &if_.eip, &if_.esp);

    /* If load failed, quit. */
    palloc_free_page (file_name);
    if (!success)
        thread_exit ();
/*missing parts!!! set up stack */
    /* Start the user process by simulating a return from an
       interrupt, implemented by intr_exit (in
       threads/intr-stubs.S). Because intr_exit takes all of its
       arguments on the stack in the form of a `struct intr_frame',
       we just point the stack pointer (%esp) to our stack frame
       and jump to it. */
asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
    NOT_REACHED ();
}

```

Getting out of the kernel

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

movl %0, %%esp

Set the esp to the top of the interrupt frame.

jmp intr_exit

executes intr_exit

```
.globl intr_exit
.func intr_exit
intr_exit:
    /* Restore caller's registers. */
    popal
    popl %gs
    popl %fs
    popl %es
    popl %ds

    /* Discard `struct intr_frame' vec_no, error_code,
       frame_pointer members. */
    addl $12, %esp

    /* Return to caller. */
    iret
.endfunc
```

Getting out of the kernel

```
struct intr_frame {  
    /* Pushed by intr_entry in intr-stubs.S.  
     * These are the interrupted task's saved registers. */  
    uint32_t edi;           /* Saved EDI. */  
    uint32_t esi;           /* Saved ESI. */  
    uint32_t ebp;           /* Saved EBP. */  
    uint32_t esp_dummy;     /* Not used. */  
    uint32_t ebx;           /* Saved EBX. */  
    uint32_t edx;           /* Saved EDX. */  
    uint32_t ecx;           /* Saved ECX. */  
    uint32_t eax;           /* Saved EAX. */  
    uint16_t gs, :16;        /* Saved GS segment register. */  
    uint16_t fs, :16;        /* Saved FS segment register. */  
    uint16_t es, :16;        /* Saved ES segment register. */  
    uint16_t ds, :16;        /* Saved DS segment register. */
```

← esp

```
/* Pushed by intrNN_stub in intr-stubs.S. */  
uint32_t vec_no;          /* Interrupt vector number. */  
  
/* Sometimes pushed by the CPU,  
 * otherwise for consistency pushed as 0 by intrNN_stub.  
 * The CPU puts it just under `eip', but we move it here. */  
uint32_t error_code;      /* Error code. */  
  
/* Pushed by intrNN_stub in intr-stubs.S.  
 * This frame pointer eases interpretation of backtraces. */  
void *frame_pointer;      /* Saved EBP (frame pointer). */
```

← esp After intr_exit

```
/* Pushed by the CPU.  
 * These are the interrupted task's saved registers. */  
void (*eip) (void);       /* Next instruction to execute. */  
uint16_t cs, :16;          /* Code segment for eip. */  
uint32_t eflags;           /* Saved CPU flags. */  
void *esp;                 /* Saved stack pointer. */  
uint16_t ss, :16;          /* Data segment for esp. */
```

← esp After iret instruction

time

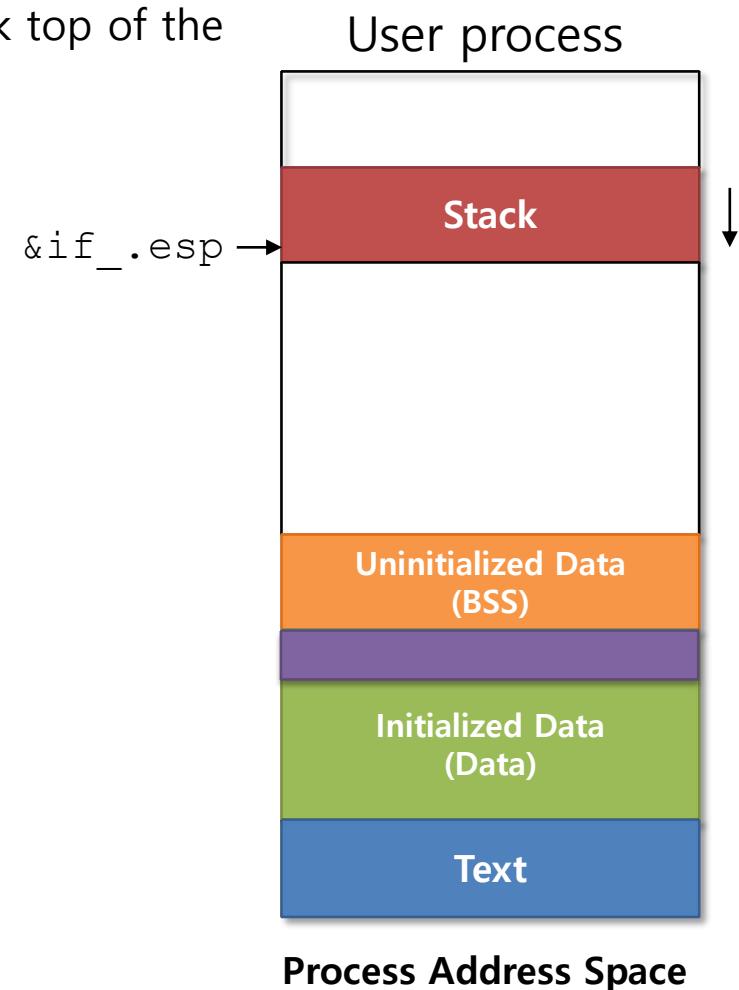
Write a function that sets up a stack.

"esp" field of the interrupt frame contains the stack top of the user stack.

```
sample_function(int argc,  
                char* argv[],  
                void **stackpointer)
```

Current stack top: &if_.esp

Start from &if_.esp - 4



80x86 Calling Convention

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

argc=4

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

1. Push arguments

1. Push character strings from left to write.
2. Place padding if necessary to align it by 4 Byte.
3. Push start address of the character strings.

2. Push argc and argv

1. Push argv
2. Push argc

3. Push the address of the next instruction (return address).

User stack layout in function call

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

Address	Name	Data	Type
0xbfffffffcc	argv[3][...]	'bar\0'	char[4]
0xbfffffff8	argv[2][...]	'foo\0'	char[4]
0xbfffffff5	argv[1][...]	'-l\0'	char[3]
0xbfffffed	argv[0][...]	'/bin/ls\0'	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbfffffc	char *
0xbfffffe0	argv[2]	0xbfffff8	char *
0xbfffffdcc	argv[1]	0xbfffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffc	return address	0 (fake address)	void (*) () → fake address(0)

Why is “return address” here is 0?

Interim Check

- Print the program's stack by using `hex_dump()` (`stdio.h`)
 - Print memory dump in hexadecimal form
 - Check if arguments are correctly pushed on user stack.

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

Intermediate Check (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

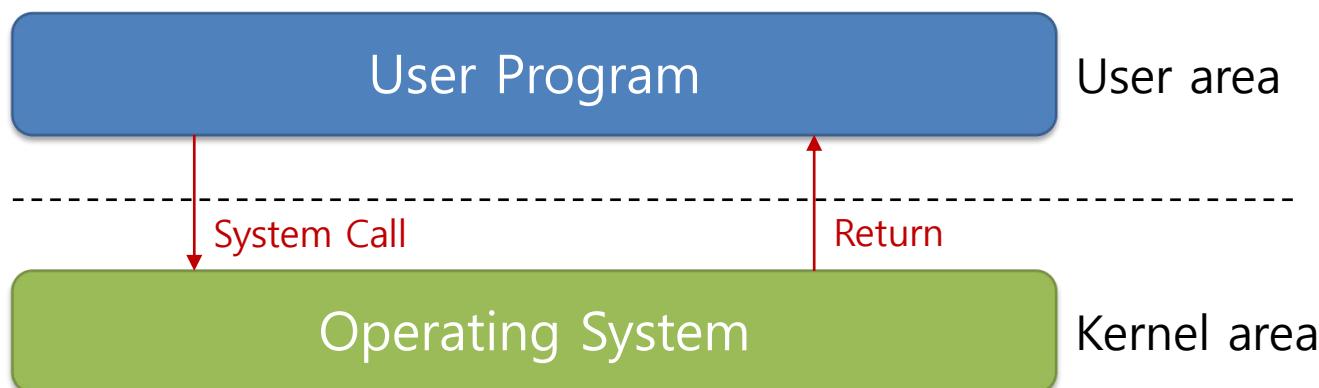
System Calls and Handlers

Overview

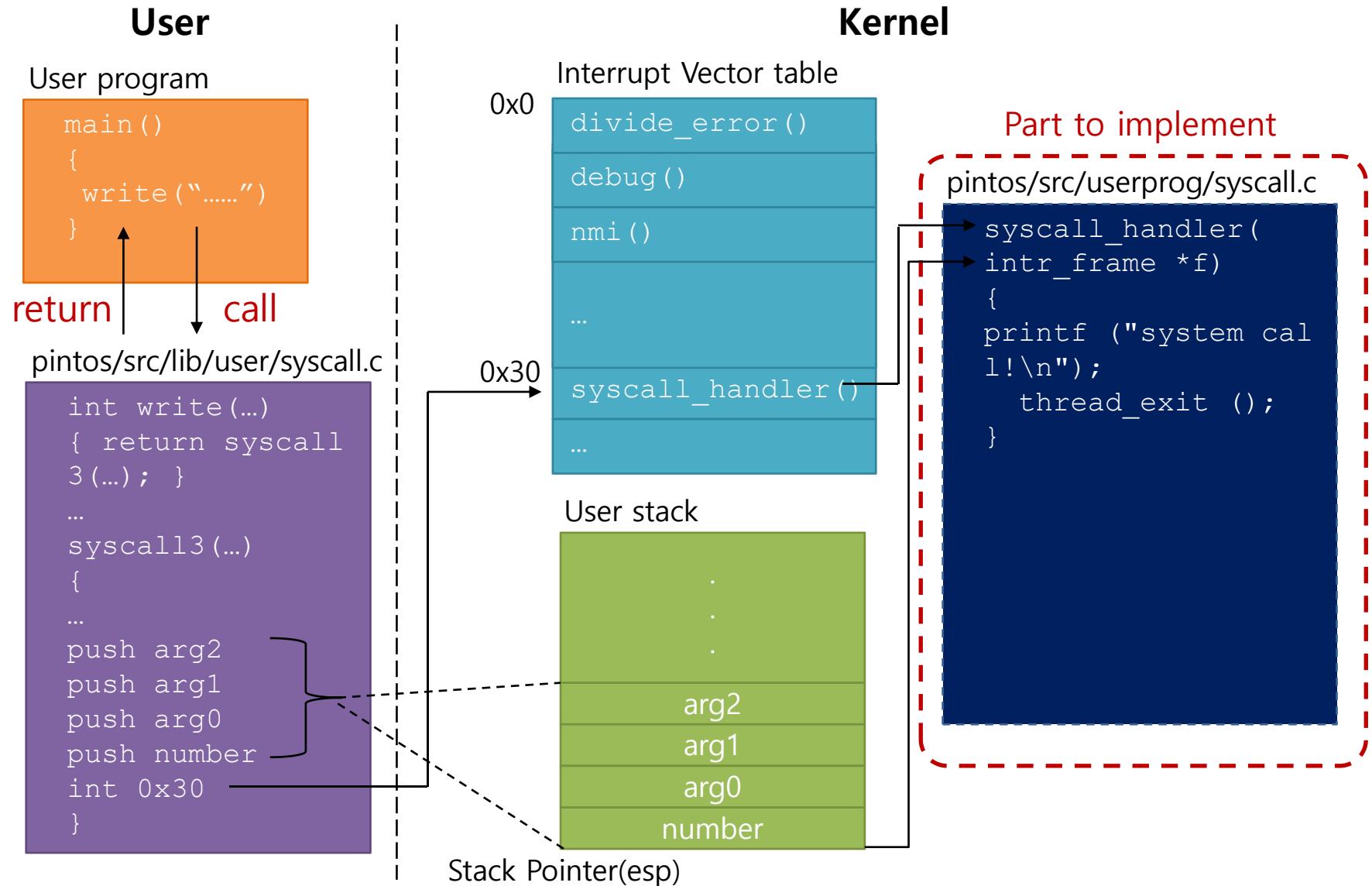
- ▣ Main goal
 - ◆ Original: system call handler table is empty.
 - ◆ After modification:
 - Fill system call handler of pintos out.
 - Add system calls to provide services to users
 - Process related: `halt`, `exit`, `exec`, `wait`
 - File related: `create`, `remove`, `open`, `filesize`, `read`, `write`, `seek`, `tell`, `close`
- ▣ Files to modify
 - ◆ `pintos/src/threads/thread.*`
 - ◆ `pintos/src/userprog/syscall.*`
 - ◆ `pintos/src/userprog/process.*`

System call

- Programming interface for services provided by the operating system
- Allow user mode programs to use kernel features
- System calls run on kernel mode and return to user mode
- Key point of system call is that priority of execution mode is raised to the special mode as hardware interrupts are generated to call system call

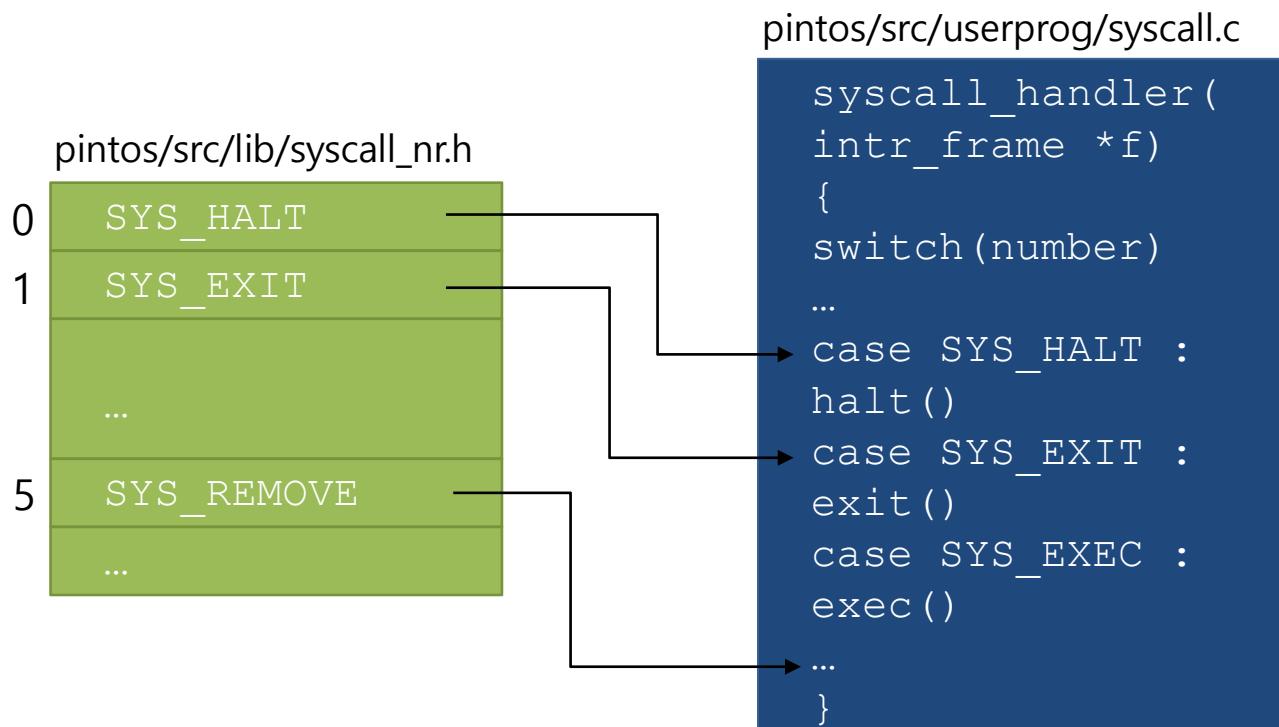


Call process of System call (Pintos)



System call handler

- Call the system call from the system call handler using the system call number.
 - The system call number is defined in `pintos/src/lib/syscall_nr.h`



Requirement for System Call handler

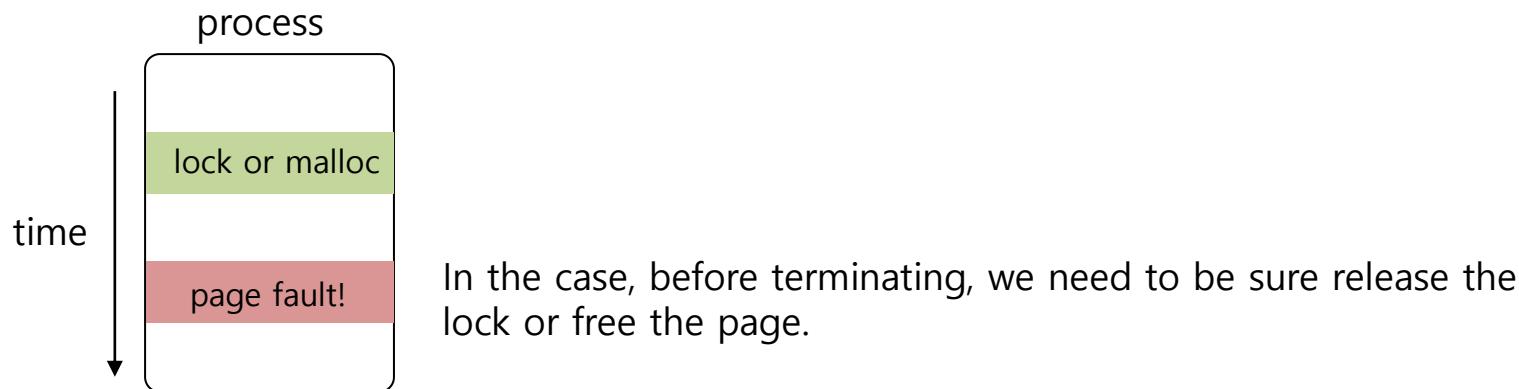
- ▣ Implement system call handler
 - ◆ Make system call handler call system call using system call number
 - ◆ Check validation of the pointers in the parameter list.
 - These pointers must point to user area, not kernel area.
 - If these pointers don't point the valid address, it is page fault
 - ◆ Copy arguments on the user stack to the kernel.
 - ◆ Save return value of system call at eax register.

Address Validation

- ▣ User can pass invalid pointers through the systemcall.
 - ◆ A null pointer / A pointer to unmapped virtual memory
 - ◆ A pointer to kernel virtual memory address space (above **PHYS_BASE**)
- ▣ Kernel need to detect invalidity of pointers and terminating process without harm to the kernel or other running processes.
- ▣ How to detect?
 - ◆ Method 1: Verify the validity of a user-provided pointer.
 - The simplest way to handle user memory access.
 - Use the functions in 'userprog/pagedir.c' and in 'threads/vaddr.h'
 - ◆ Method 2: Check only that a user points below PHYS_BASE.
 - An invalid pointer will cause 'page_fault'. You can handle by modifying the code for `page_fault()`.
 - Normally faster than first one, Because it takes advantage of the MMU.
 - It tends to be used in real kernel.

Accessing User Memory (cont.)

- In either case, make sure not to "leak" resource.



- The first technique is straightforward.
 - Lock or allocate the page only after verifying the validity of pointers.
- The second one is more difficult.
 - Because there's no way to return an error code from a memory access.
 - You can use provided functions to handle these cases. (functions are in next slide.)

Accessing User Memory (cont.)

```
/* Reads a byte at user virtual address UADDR.  
   UADDR must be below PHYS_BASE.  
   Returns the byte value if successful, -1 if a segfault  
   occurred. */  
static int  
get_user (const uint8_t *uaddr)  
{  
    int result;  
    asm ("movl $1f, %0; movzbl %1, %0; 1:"  
        : "=a" (result) : "m" (*uaddr));  
    return result;  
}
```

```
/* Writes BYTE to user address UDST.  
   UDST must be below PHYS_BASE.  
   Returns true if successful, false if a segfault occurred. */  
static bool  
put_user (uint8_t *udst, uint8_t byte)  
{  
    int error_code;  
    asm ("movl $1f, %0; movb %b2, %1; 1:"  
        : "=a" (error_code), "=m" (*udst) : "q" (byte));  
    return error_code != -1;  
}
```

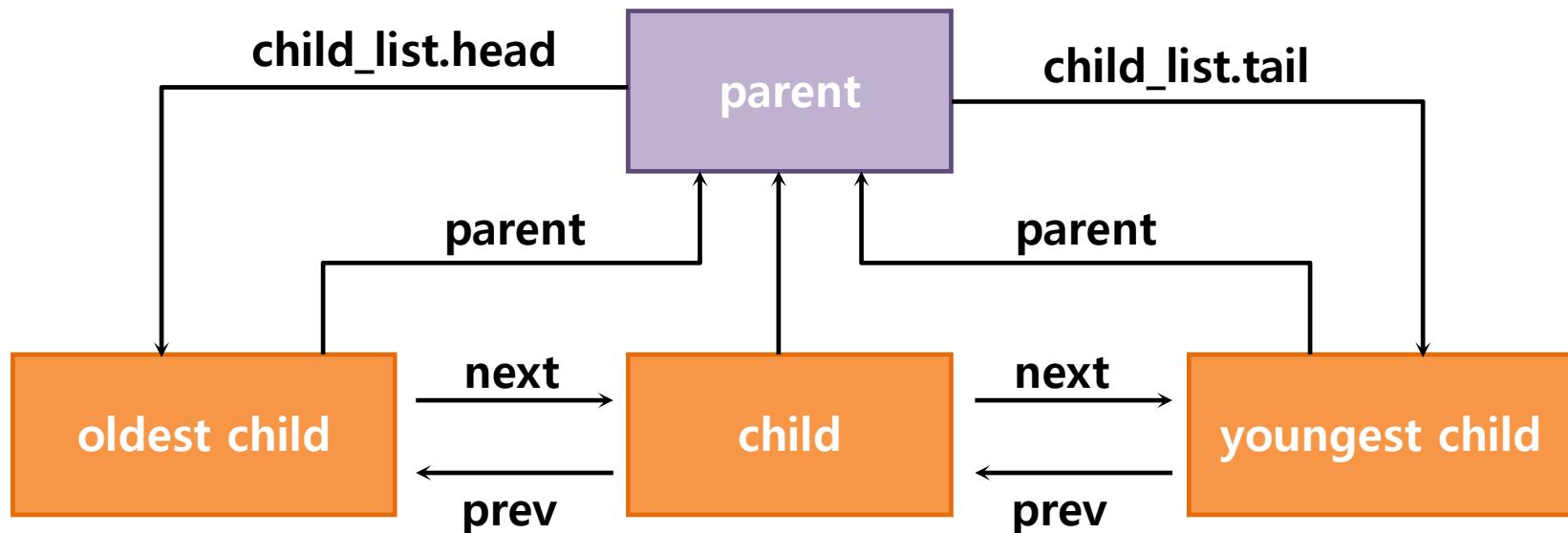
- You also modify the `page_fault ()`: set `eax` to `0xffffffff` and copies its former value into `eip`.

Add system calls: Process related system calls

- ▣ `void halt(void)`
 - Shutdown pintos
 - Use `void shutdown_power_off(void)`
- ▣ `void exit(int status)`
 - Exit process
 - Use `void thread_exit(void)`
 - It should print message "Name of process: exit(status)".
- ▣ `pid_t exec (const char *cmd_line)`
 - Create child process and execute program corresponds to `cmd_line` on it
- ▣ `int wait (pid_t pid)`
 - Wait for termination of child process whose process id is `pid`

Process Hierarchy

- Augment the existing process with the process hierarchy.
- To represent the relationship between parent & child,
 - Pointer to parent process: struct thread*
 - Pointers to the sibling. struct list
 - Pointers to the children: struct list_elem



wait() system call

- ▣ `int wait(pid_t pid)`
 - ◆ Wait for a child process `pid` to exit and retrieve the child's exit status.
 - ◆ If `pid` is alive, wait till it terminates. Returns the status that `pid` passed to `exit`.
 - ◆ If `pid` did not call `exit`, but was terminated by the kernel, return -1.
 - ◆ A parent process can call `wait` for the child process that has terminated.
 - return exit status of the terminated child process.
 - ◆ After the child terminates, the parent should deallocate its process descriptor
 - ◆ `wait` fails and return -1 if
 - `pid` does not refer to a direct child of the calling process.
 - The process that calls `wait` has already called `wait` on `pid`.

Kernel function for wait – process_wait

```
int process_wait (tid_t child_tid UNUSED)
```

- ◆ It is currently empty.

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

- ◆ Insert the infinite loop so that the kernel does not finish. For now...

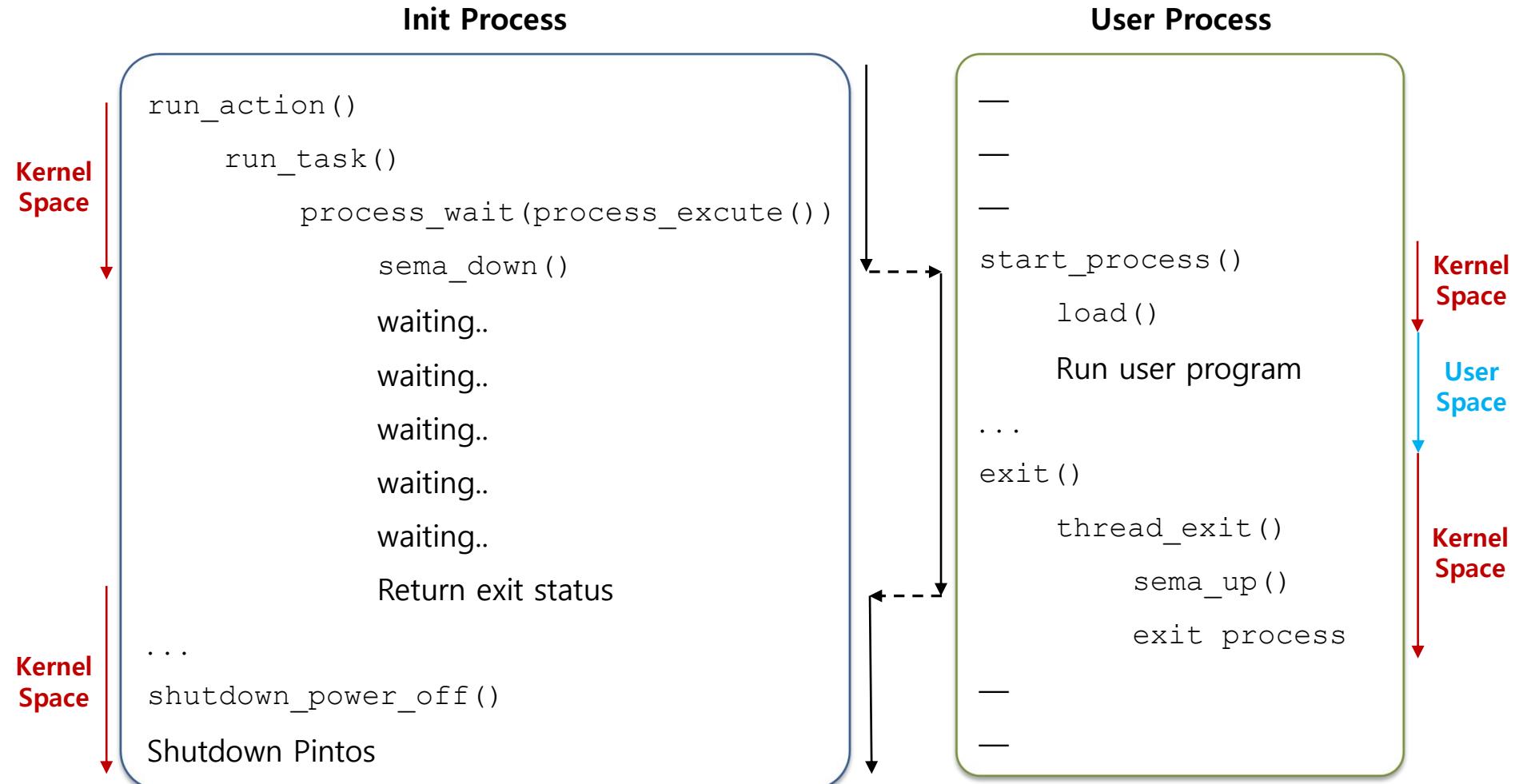
Correct implementation: process_wait()

- ▣ process_wait()
 - ◆ Search the descriptor of the child process by using child_tid.
 - ◆ The caller blocks until the child process exits.
 - ◆ Once child process exits, deallocate the descriptor of child process and returns exit status of the child process.
- ▣ Semaphore
 - ◆ Add a semaphore for “wait” to thread structure.
 - ◆ Semaphore is initialized to 0 when the thread is first created.
 - ◆ In wait(tid), call sema_down for the semaphore of tid.
 - ◆ In exit() of process tid, call sema_up.
 - ◆ Where do we need to place sema_down and sema_up?
- ▣ Exit status
 - ◆ Add a field to denote the exit status to the thread structure.

Flow of parent calling wait and child

Flow of user program execution

→ Flow
→ Scheduling



exec() system call

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

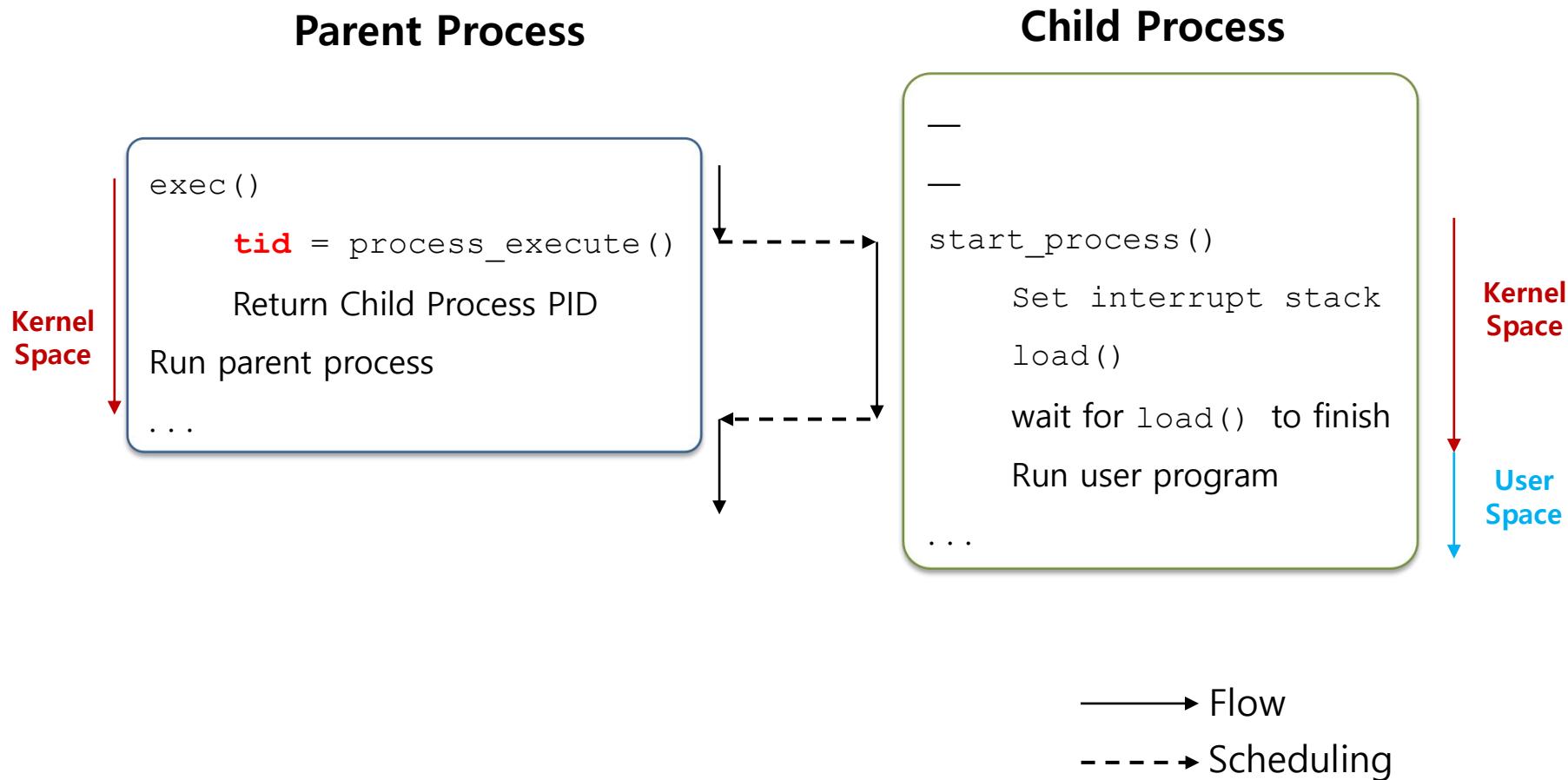
- ◆ Run program which execute cmd_line.
- ◆ Create thread and run. exec() in pintos is equivalent to fork() + exec() in Unix
- .
- ◆ Pass the arguments to program to be executed.
- ◆ Return pid of the new child process.
- ◆ If it fails to load the program or to create a process, return -1.
- ◆ Parent process calling exec should wait until child process is created and loads the executable completely.

Kernel function for exec() : process_execute()

- ▣ Parent should wait until it knows the child process has successfully created and the binary file is successfully loaded.
- ▣ Semaphore
 - ◆ Add a semaphore for “exec()” to thread structure.
 - ◆ Semaphore is initialized by 0 when the thread is first created.
 - ◆ Call `sema_down` to wait for the successful load of the executable file of the child processes.
 - ◆ Call `sema_up` when the executable file is successfully loaded.
 - ◆ **Where do we need to place `sema_down` and `sema_up`?**
- ▣ load status
 - ◆ In the thread structure, we need a field to represent whether the file is successfully loaded or not.

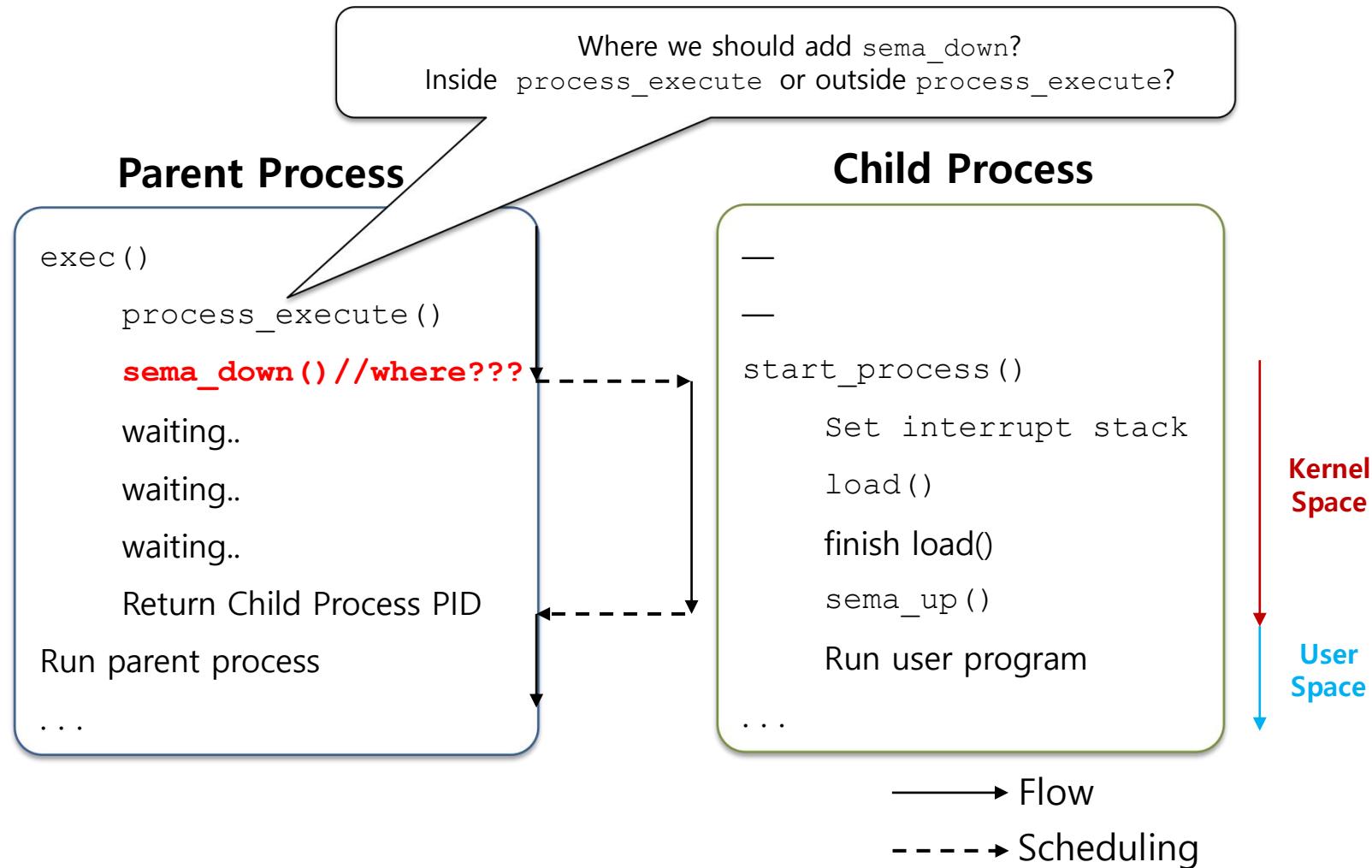
Current flow of the parent calling exec and the child

- exec() return itself only after child is completely loaded.
- tid can have valid value even the load has failed.



Correct Flow of the parent calling exec and the child

- exec() return itself only after child is completely loaded.



exit()

- ❑ Terminate the current user program, returning status to the kernel.
- ❑ If the process' parent waits for it, this is the status that will be returned.

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

Kernel function for exit(): thread_exit

- Exit status
 - ◆ Store the status to the status of process.
- Semaphore
 - ◆ Call sema_up for the current process.

```
/* Deschedules the current thread and destroys it.  Never
   returns to the caller. */
void
thread_exit (void)
{
    ASSERT (!intr_context ());

#ifndef USERPROG
    process_exit ();
#endif

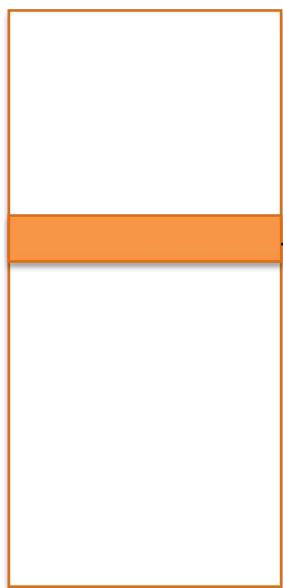
    /* Remove thread from all threads list, set our status to dying,
       and schedule another process.  That process will destroy us
       when it calls thread_schedule_tail(). */
    intr_disable ();
    list_remove (&thread_current ()->allelem);
    thread_current ()->status = THREAD_DYING;
    schedule ();
    NOT_REACHED ();
}
```

File Manipulation

File Descriptor in Unix

Access to File by using File Descriptor

Process Descriptor
Table

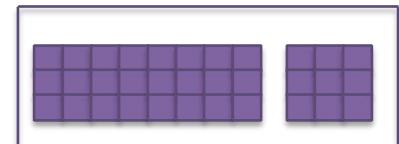


Part to implement

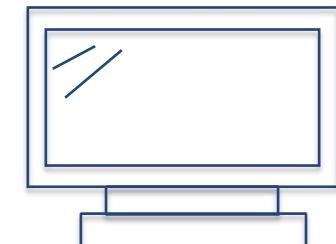
File Descriptor
Table

0 : STDIN
1 : STDOUT
2 : STDERR
3 : File
4 : File
.
.
.

Keyboard
File Object

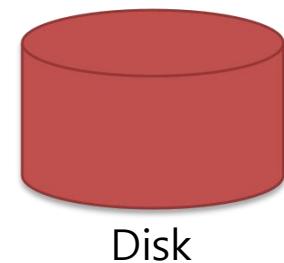


Monitor
File Object



Monitor
File Object

I/O
File Object



File Descriptor Table

▣ Implement File Descriptor Table.

- ◆ Each process has its own file descriptor table (Maximum size: 64 entry).
- ◆ File descriptor table is an array of pointer to struct file.
- ◆ FD is index of the file descriptor table, and it is allocated sequentially.
 - FD 0 and 1 are allocated for `stdin` and `stdout`, respectively.
- ◆ `open()` returns fd.
- ◆ `close()` set 0 at file descriptor entry at index fd.

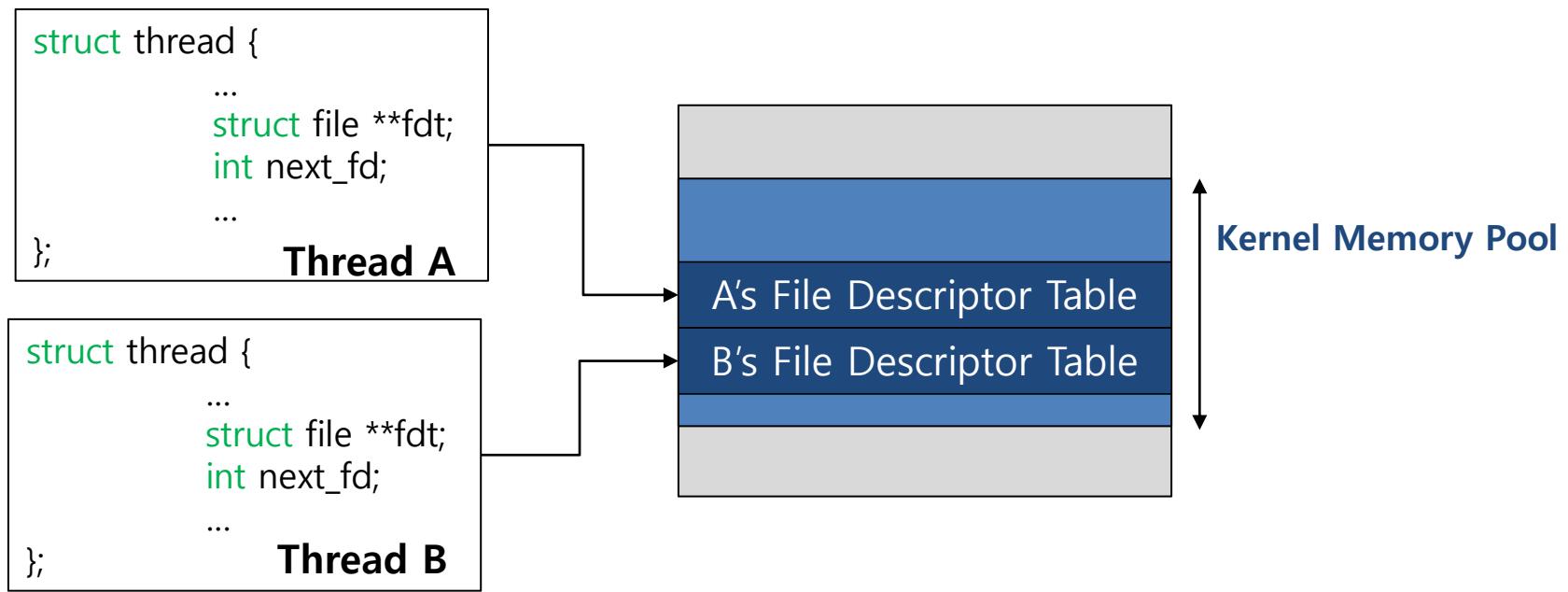
Allocate File Descriptor Table

- Define FDT as a part of thread structure.

```
struct thread {  
    ...  
    struct file fdt[64];  
    int next_fd;  
    ...  
};
```

Thread A

- Allocate FDT at kernel memory area, and add the associated pointer to at the thread structure.



File Descriptor Table

- When the thread is created,
 - ◆ Allocate File Descriptor table.
 - ◆ Initialize pointer to file descriptor table.
 - ◆ Reserve fd0, fd1 for stdin and stdout.
- When thread is terminated,
 - ◆ Close all files.
 - ◆ Deallocate the file descriptor table.
- Use global lock to avoid race condition on file,
 - ◆ Define a global lock on syscall.h (`struct lock filesys_lock`).
 - ◆ Initialize the lock on `syscall_init()` (Use `lock_init()`).
 - ◆ Protect filesystem related code by global lock.

Modify page_fault() for test

- Some tests check whether your kernel handles the bad process properly.
- Pintos needs to kill the process and print the thread name and the exit status -1 when page fault occurs.
- We have to modify page_fault() to satisfy test's requirements.

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;

    /* Call exit(-1) */
    ...
}
```

Add system calls: File related system calls

- ▣ `bool create(const char *file, unsigned initial_size)`
 - ◆ Create file which have size of `initial_size`.
 - ◆ Use `bool filesys_create(const char *name, off_t initial_size)`.
 - ◆ Return true if it is succeeded or false if it is not.
- ▣ `bool remove(const char *file)`
 - ◆ Remove file whose name is `file`.
 - ◆ Use `bool filesys_remove(const char *name)`.
 - ◆ Return true if it is succeeded or false if it is not.
 - ◆ File is removed regardless of whether it is open or closed.
- ▣ `int open(const char *file)`
 - ◆ Open the file corresponds to path in “`file`”.
 - ◆ Return its fd.
 - ◆ Use `struct file *filesys_open(const char *name)`.

Add system calls: File related system calls (Cont.)

- ▣ `int filesize(int fd)`
 - ◆ Return the size, in bytes, of the file open as `fd`.
 - ◆ Use `off_t file_length(struct file *file)`.
- ▣ `int read(int fd, void *buffer, unsigned size)`
 - ◆ Read `size` bytes from the file open as `fd` into `buffer`.
 - ◆ Return the number of bytes actually read (0 at end of file), or -1 if fails.
 - ◆ If `fd` is 0, it reads from keyboard using `input_getc()`, otherwise reads from file using `file_read()` function.
 - `uint8_t input_getc(void)`
 - `off_t file_read(struct file *file, void *buffer, off_t size)`

Add system calls: File related system calls (Cont.)

- ▣ `int write(int fd, const void *buffer, unsigned size)`
 - ◆ Writes size bytes from buffer to the open file fd.
 - ◆ Returns the number of bytes actually written.
 - ◆ If fd is 1, it writes to the console using `putbuf()`, otherwise write to the file using `file_write()` function.
 - `void putbuf(const char *buffer, size_t n)`
 - `off_t file_write(struct file *file, const void *buffer, off_t size)`
- ▣ `void seek(int fd, unsigned position)`
 - ◆ Changes the next byte to be read or written in open file fd to position.
 - ◆ Use `void file_seek(struct file *file, off_t new_pos)`.

Add system calls: File related system calls (Cont.)

- ▣ `unsigned tell(int fd)`
 - ◆ Return the position of the next byte to be read or written in open file `fd`.
 - ◆ Use `off_t file_tell(struct file *file)`.
- ▣ `void close(int fd)`
 - ◆ Close file descriptor `fd`.
 - ◆ Use `void file_close(struct file *file)`.

Denying writes to executable

- ▣ What if the OS tries to execute the file that is being modified?
- ▣ Do not allow the file to be modified when it is opened for execution.
- ▣ Approach
 - ◆ When the file is loaded for execution, call `file_deny_write()`.
 - ◆ When the file finishes execution, call `file_allow_write()`.

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

- ◆ Call `file_deny_write()` when program file is opened.
- ◆ Add a running file structure to thread structure.

```
void process_exit (void)
```

- ◆ Modify current process to close the running file.

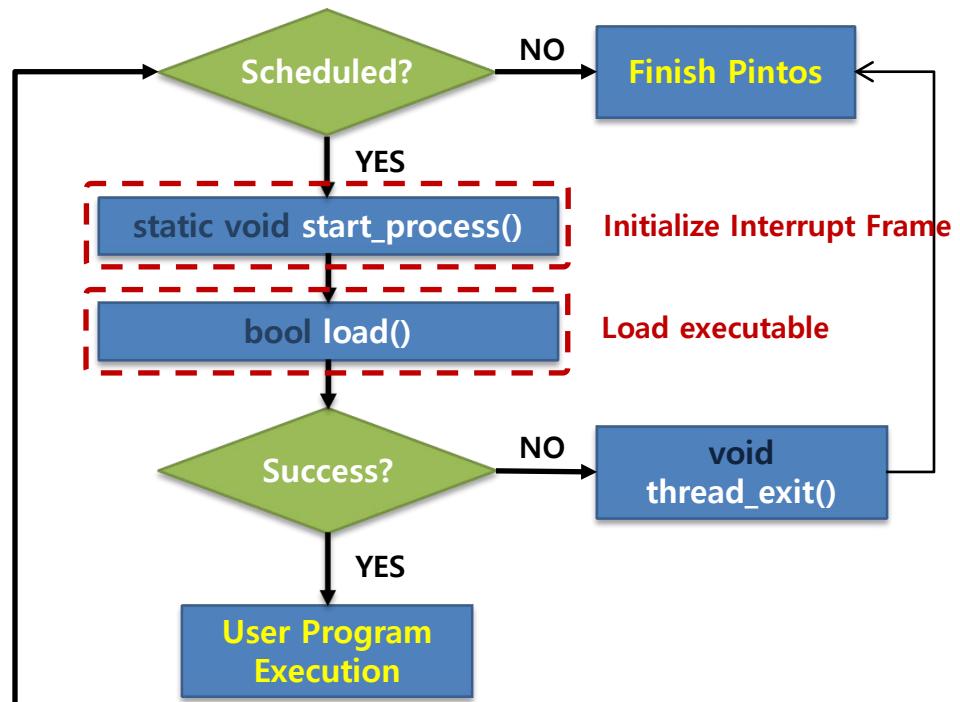
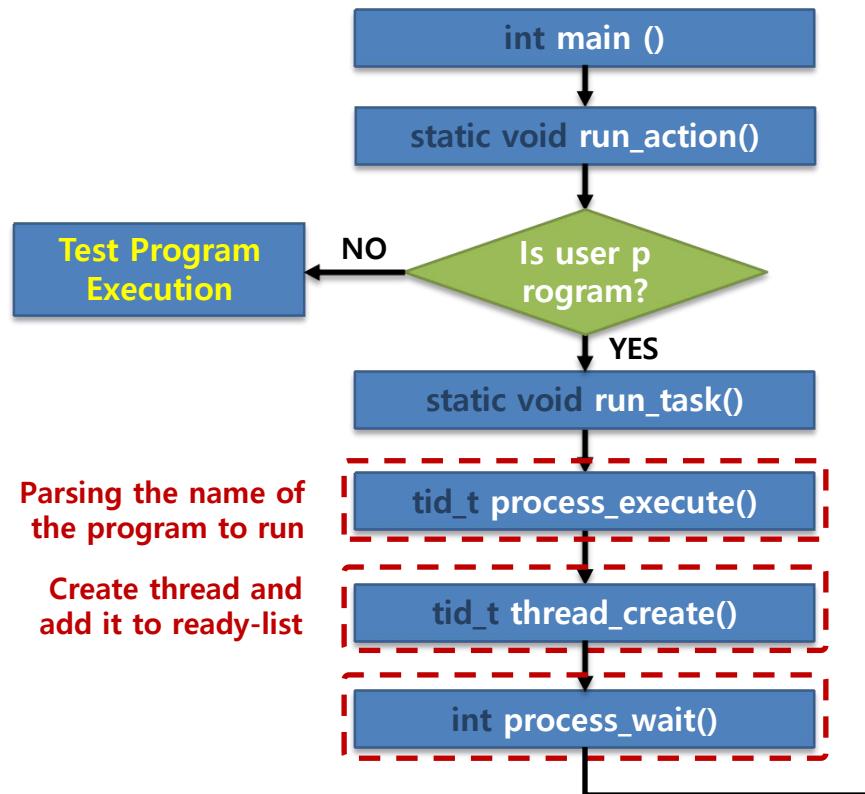
Result

- Check the result of all tests.
 - ◆ Path: pintos/src/userprog

```
$make grade
```

SUMMARY BY TEST SET				
Test Set	Pts	Max	%	Ttl
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%			

Summary



Operating Systems Lab

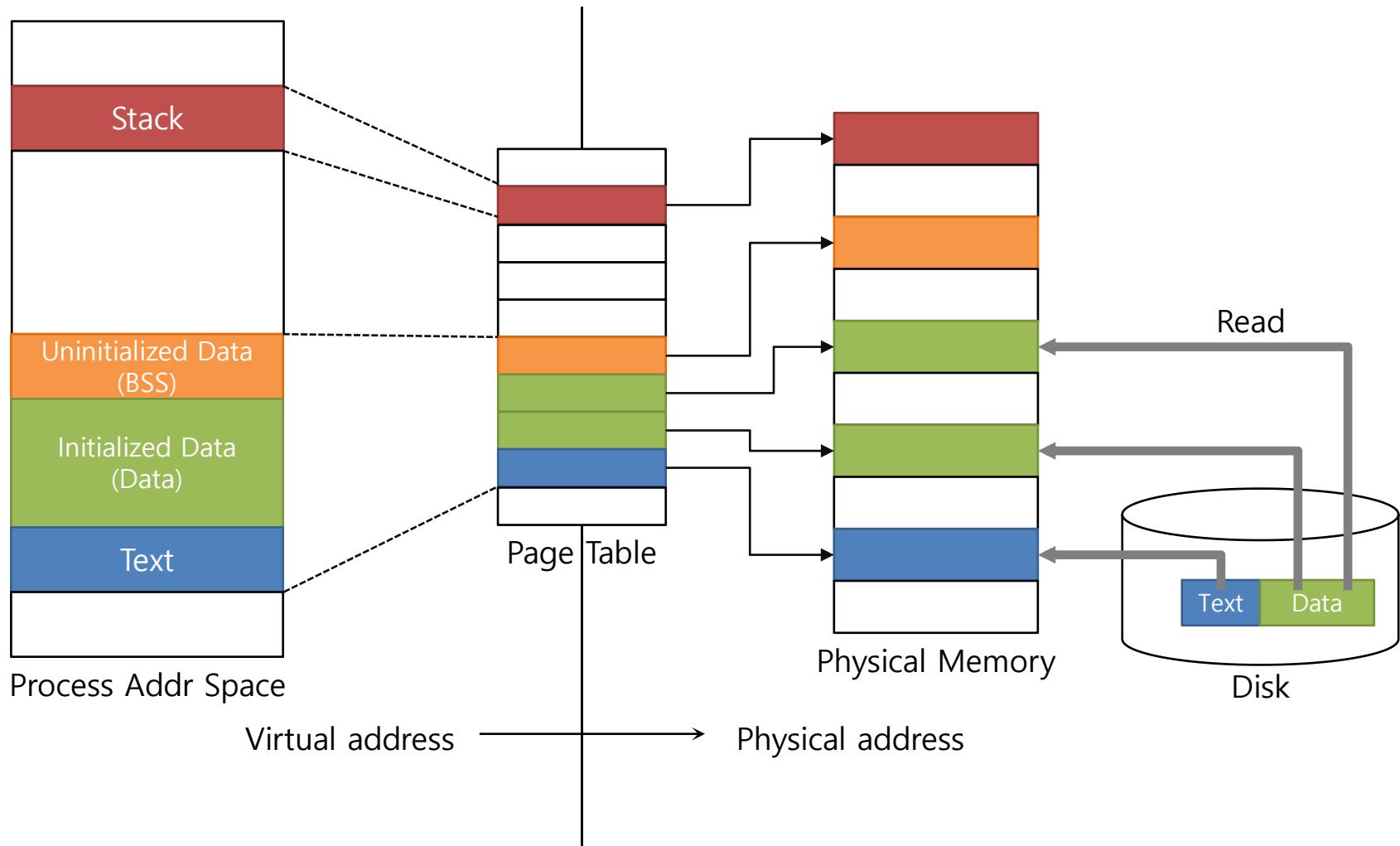
Part 3: Virtual Memory



Youjip Won

Address space of process in Pintos

- Pintos memory layout before project



Virtual Memory

- ❑ Entire executable file is loaded at once at the beginning.
- ❑ Physical addresses of each page in address space are fixed at the beginning of 'fork/exec'.
- ❑ Result

Implement “Virtual Address”.

To Do's

Implement “Virtual Address”.

- ▣ Enable Demand paging/Swapping.
- ▣ Enable Stack Growth.
 - ◆ Dynamic page allocation for page fault on stack
- ▣ Implement Memory mapped file.
 - ◆ Implement `mmap()` and `munmap()`.
 - ◆ For a physical page, differentiate `file_backed` page and anonymous page.
- ▣ Enable Accessing User Memory.

Demand Paging

Basics

- ▣ Virtual page: Virtual Page number (20 bit) + page offset (12bit)
- ▣ Page frame: physical frame number (20 bit) + page offset (12 bit)
- ▣ Page table:
 - ◆ VPN → PFN
 - ◆ It is hardware.
- ▣ Swap space: array of page sized blocks

A page in virtual address space

- Load the page from the disk as requested.
- A page in VM can be either in-memory only or part of a file.
 - ◆ text: part of file
 - ◆ Data: part of file
 - ◆ BSS: in memory
 - ◆ Stack: in memory
 - ◆ Heap: in memory
 - ◆ mmap()ed region: part of file

Page fault in current Pintos

Userprog/exception.c

```
static void
page_fault (struct intr_frame *f)
{
...
/* 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);
}
```

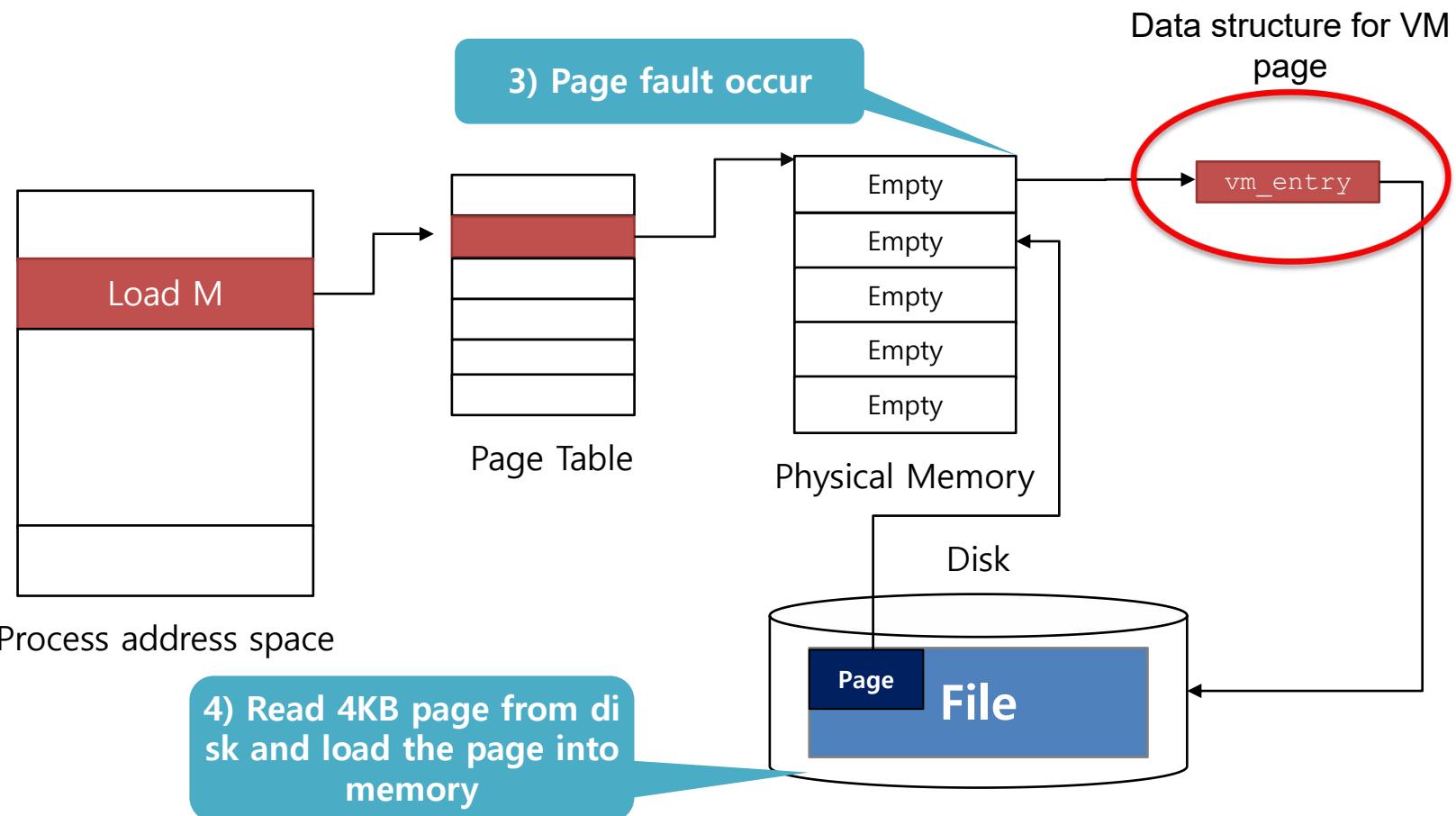
Page fault in Pintos with VM

- ▣ When page fault occurs? Modify the `page_fault` function
 - ◆ Check if the memory reference is valid.
 - locate the content that needs to go into the virtual memory page
 - from the file, from the swap or can simply be all-zero page.
 - ◆ For shared page, the page can be already in the page frame, but not in the page table
 - ◆ Invalid access → kill the process
 - Not valid user address
 - Kernel address
 - Permission error (attempt to write to the read-only page)
 - ◆ Allocate page frame.
 - ◆ Fetch the data from the disk to the page frame.
 - ◆ Update page table.

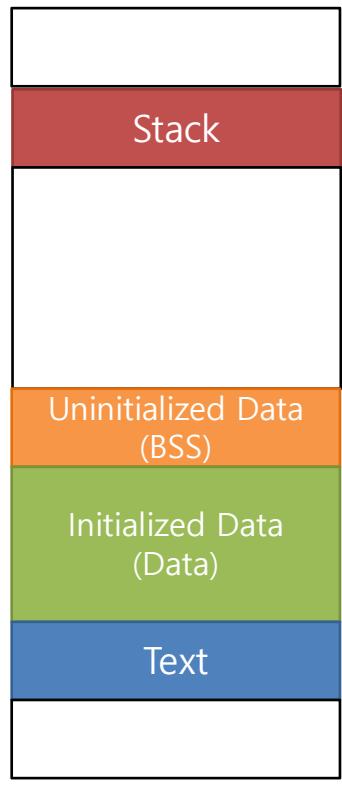
We need additional information for a virtual page

- ❑ Virtual page number
- ❑ Read/write permission
- ❑ Type of virtual page
 - ◆ a page of ELF executable file
 - ◆ a page of general file
 - ◆ a page of swap area
- ❑ Reference to the file object and offset(memory mapped file)
- ❑ Amount of data in the page
- ❑ Location in the swap area
- ❑ In-memory flag: is it in memory?

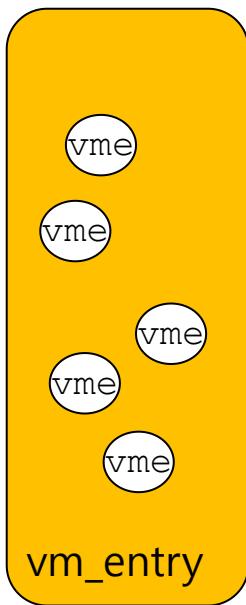
vm_entry



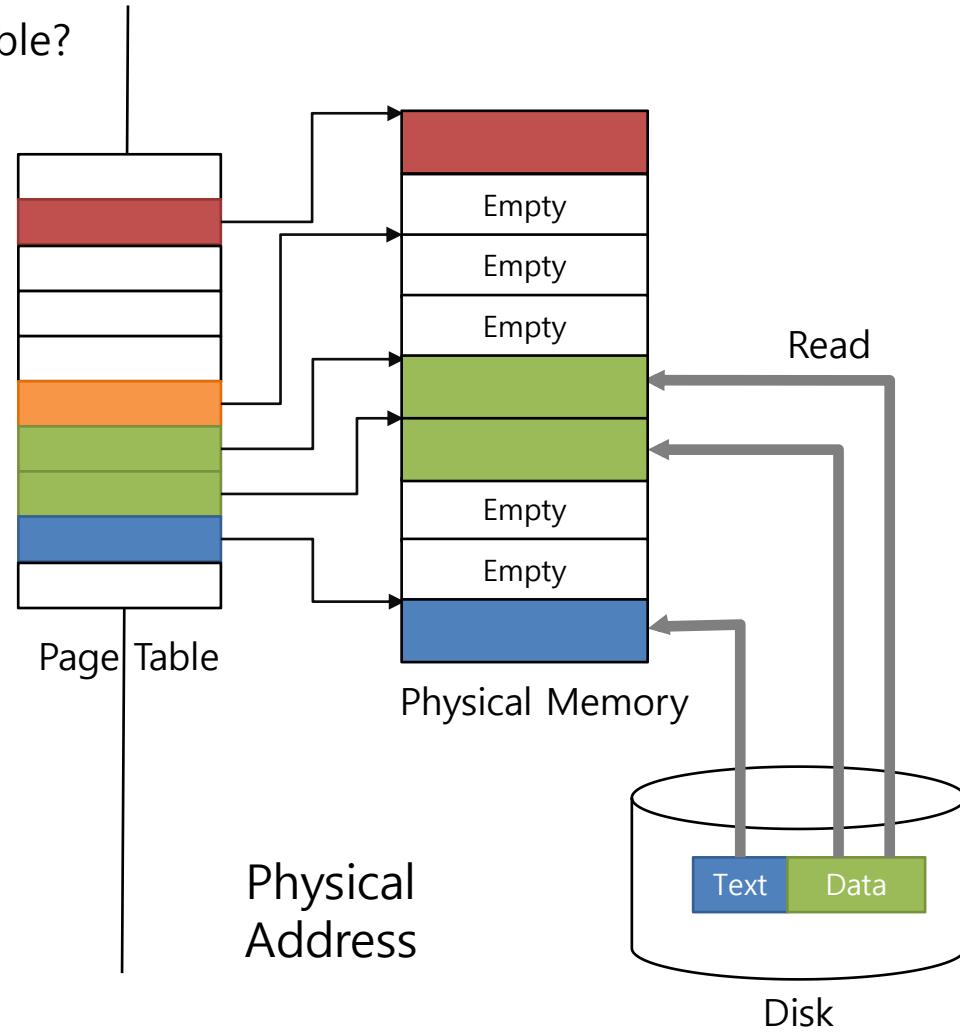
A set of virtual pages for a process: a set of vm_entry



Table/Linked list/Hash table?

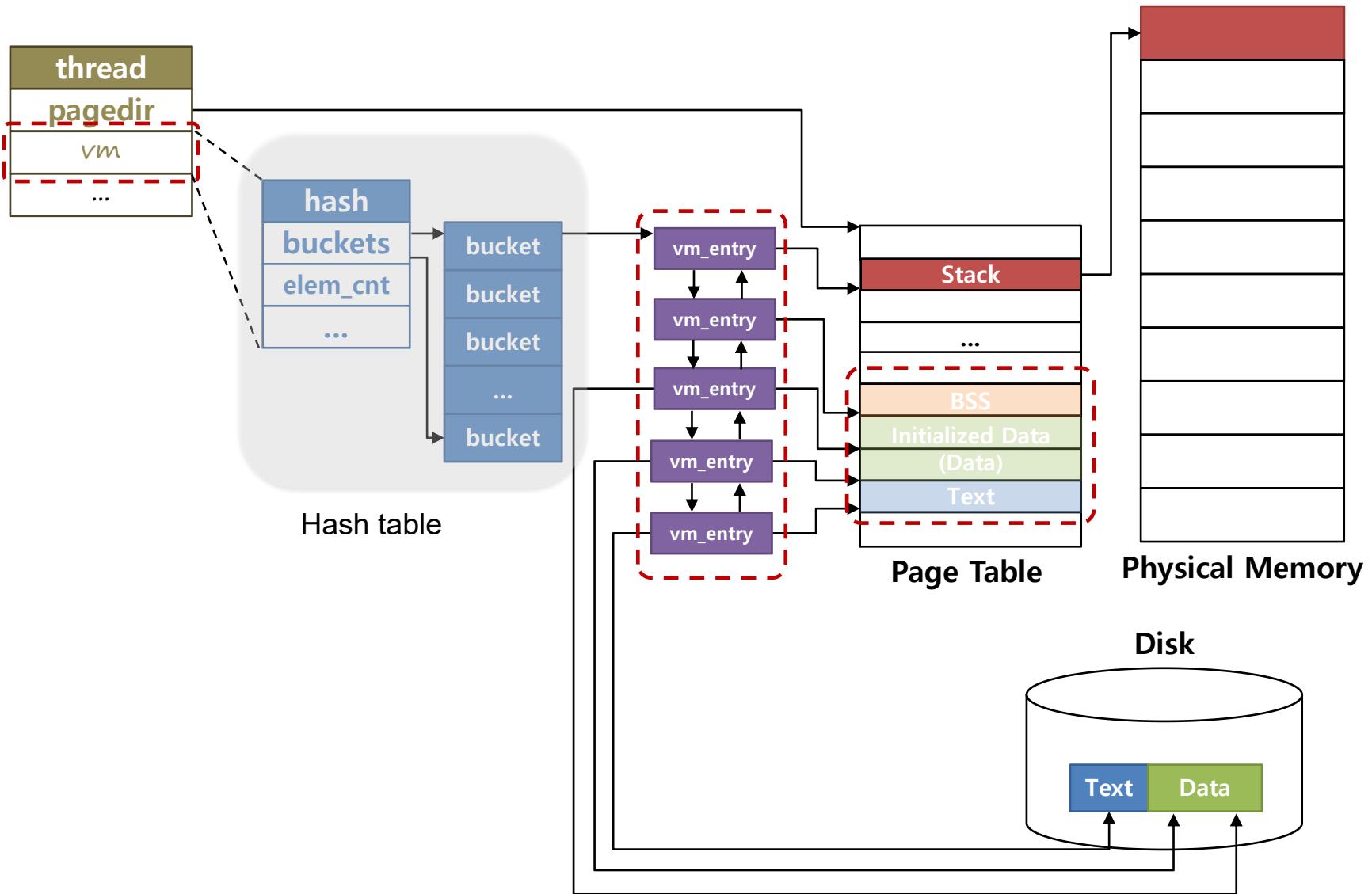


Virtual Address



Physical Address

Address Space in Pintos with VM



vm_entry

pintos/src/vm/page.h

```
struct vm_entry{  
    // fill this out.  
}
```

- ❑ Organize the vm_entry: Hash table(src/lib/kernel/hash.*), linked list, or etc.

Add vm_entry set to thread structure

```
struct thread
```

Since virtual address space is allocated for each process, define the hash table to manage virtual pages.

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. */

}
```

The line `struct hash vm;` is highlighted with a red dashed box, and the comment `/*Hash table to manage virtual address space of thread*/` is shown to its right.

Modify start_process()

pintos/src/userprog/process.c

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

/* Initializing the set of vm_entries, e.g. hash table */

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

Modify exit()

- remove vm_entries when the process exits.

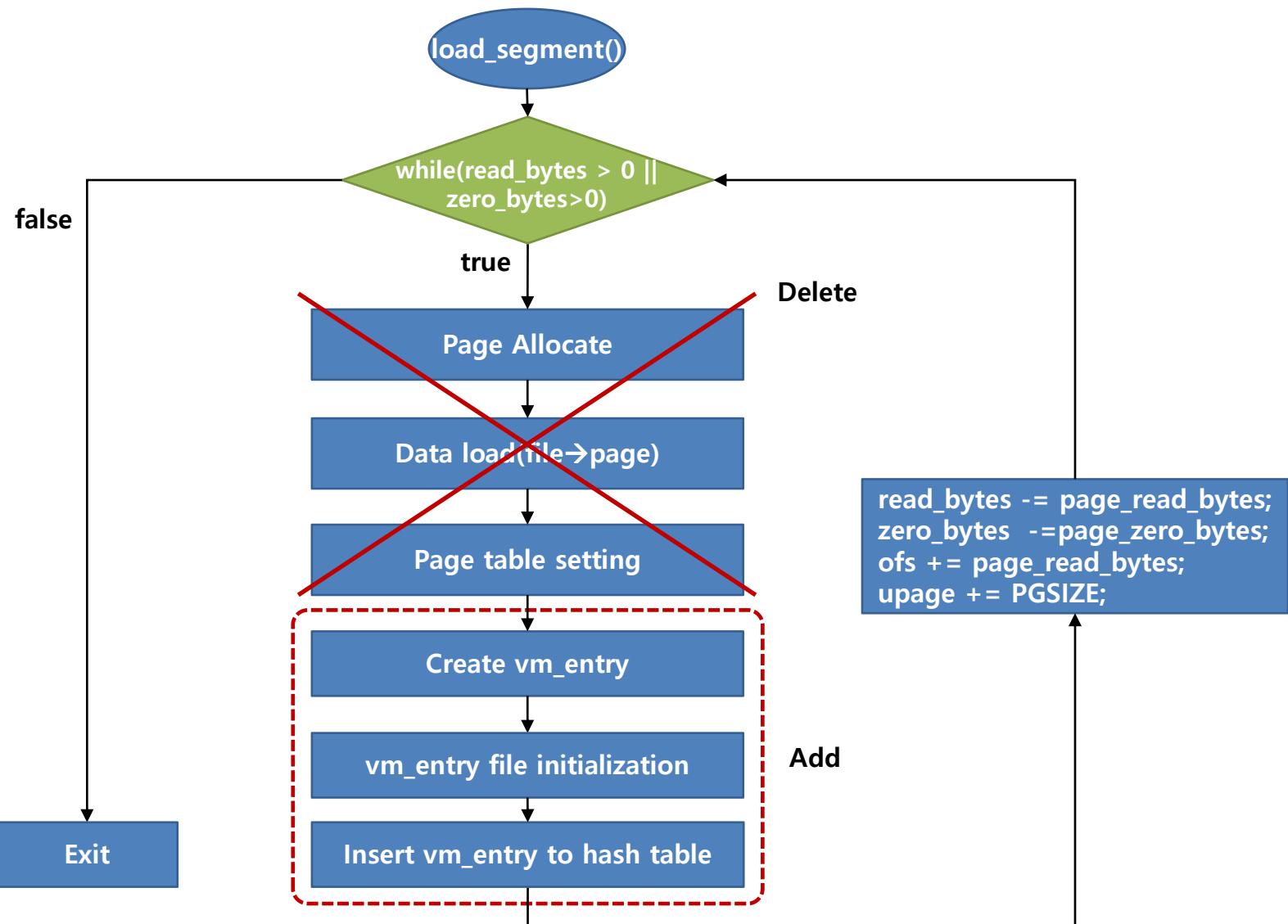
pintos/src/userprog/process.c

```
void process_exit (void) {
    struct thread *cur = thread_current();
    uint32_t *pd;
    ...
    palloc_free_page(cur -> fd);
    /* Add vm_entry delete function */
    pd = cur->pagedir;
    ...
}
```

Address Space Initialization

- ▣ Original Pintos: Allocate physical memory by reading all ELF image.
 - ◆ Read Data and code segment by `load_segment()`.
 - ◆ Allocate physical page of stack by `setup_stack()`.
- ▣ Pintos with VM
 - ◆ Allocate page table: all entries are invalid.(not mapped).
 - ◆ Allocate `vm_entry` for each page instead of allocating of physical memory.
 - ◆ **Modify `load_segment()` .**
 - Add a function that initializes structures related to virtual address space.
 - Remove the following: loading the binary file to virtual address space.
 - Add the followings.
 - allocate `vm_entry` structure.
 - Initialize the field values.
 - insert it to the hash table.

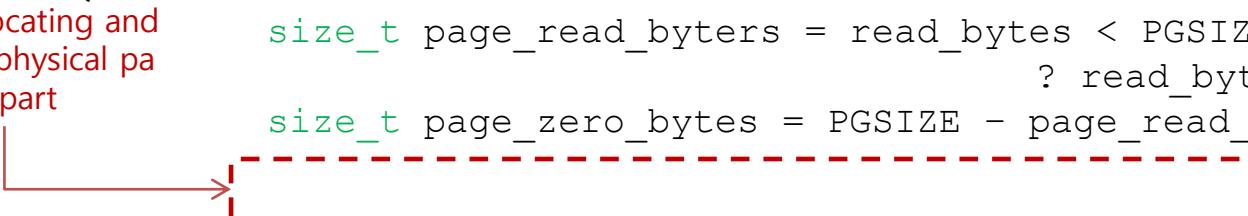
Modify load_segment()



Modify load_segment()

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;

Delete allocating and
mapping physical pa
ge part

        /* Create vm_entry(Use malloc) */
        /* Setting vm_entry members, offset and size of file to rea
d when virtual page is required, zero byte to pad at the end, ... */
        /* Add vm_entry to hash table by insert_vme() */

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

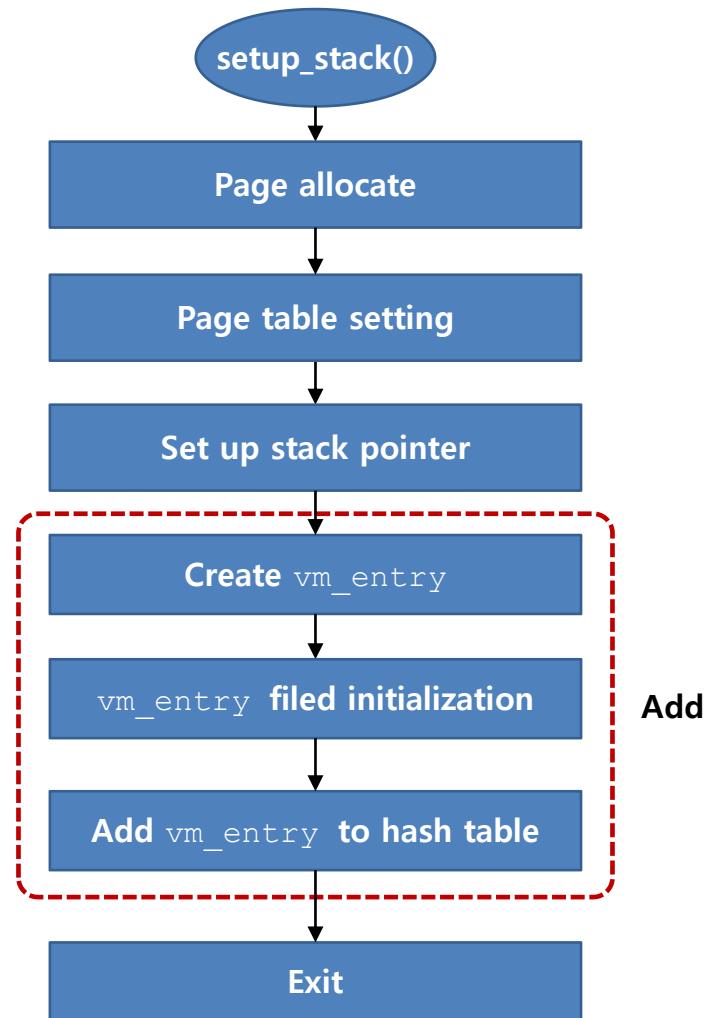
Modify stack initialization function

Original

- ◆ Allocate a single page
- ◆ Page table setting
- ◆ Stack pointer(esp) setting

Add

- ◆ Create `vm_entry` of 4KB stack
- ◆ Initialize created `vm_entry` field value
- ◆ Insert vm hash table



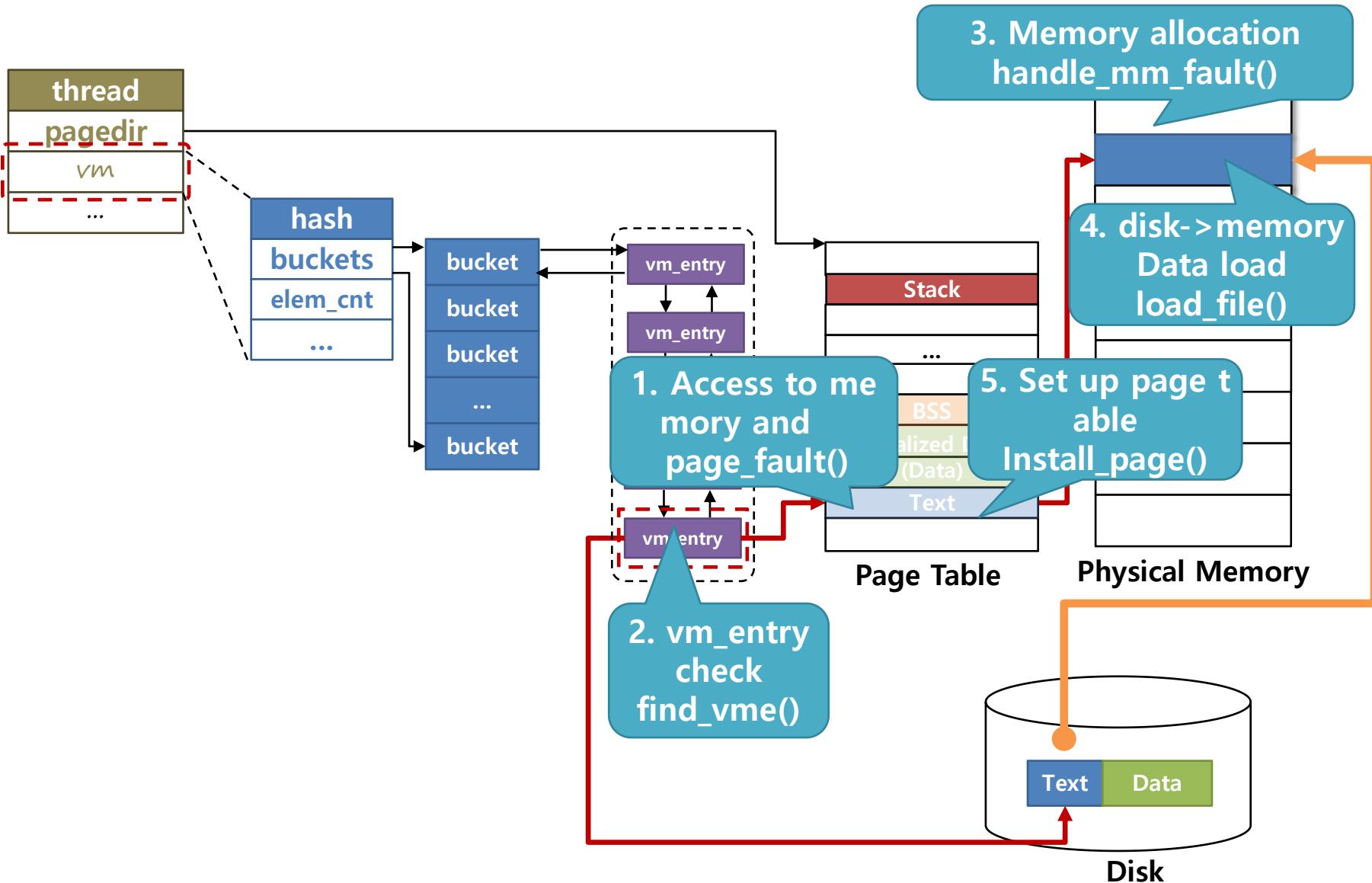
Modify setup_stack()

pintos/src/vm/page.c

```
static bool setup_stack (void **esp)
{
    ...
    if (kpage != NULL)
    {
        ...
    }
    /* Create vm_entry */
    /* Set up vm_entry members */
    /* Using insert_vme(), add vm_entry to hash table */

    ...
}
```

Design: Demand Paging



To do 1: page fault handling

- ▣ `page_fault()` exists in Pintos to manage the page fault.
 - ◆ `pintos/src/userprog/exception.c`
 - `static void page_fault (struct intr_frame *f)`
 - When existing Pintos manage page fault, after checking permission and validation of address, if error occurs, generate "segmentation fault" and `kill(-1)` to terminate.
 - Delete code related to `kill (-1)`.
 - Check Validation of `fault_addr`.
 - Define the new page fault handler and call it.
 - `handle_mm_fault(struct vm_entry *vme)`

page fault management

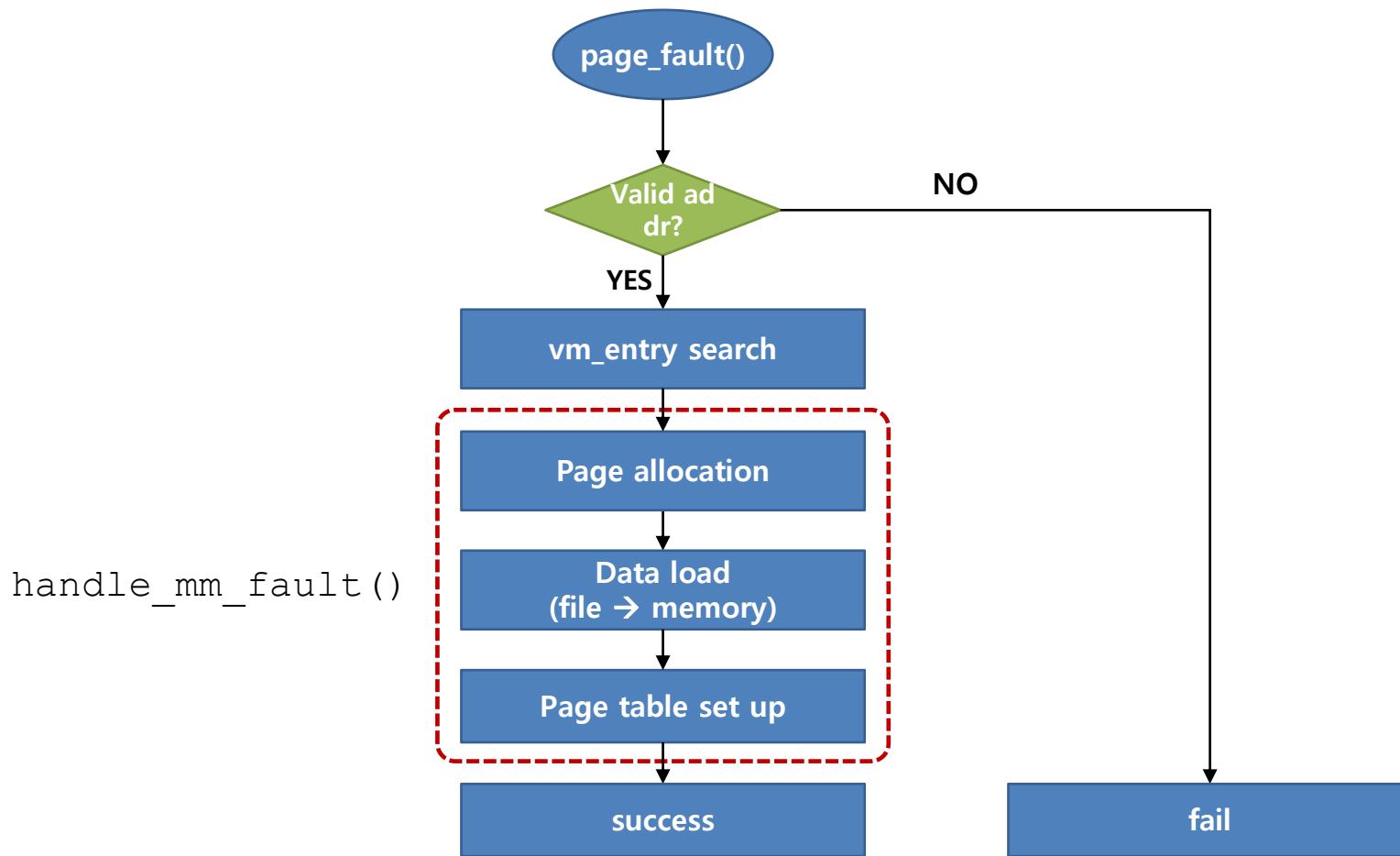
pintos/src/userprog/exception.c

```
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);
}
```

Delete & implement code

page fault management



To do 2: implement page fault handler

□ Page fault handler function(pintos/src/userprog/process.c)

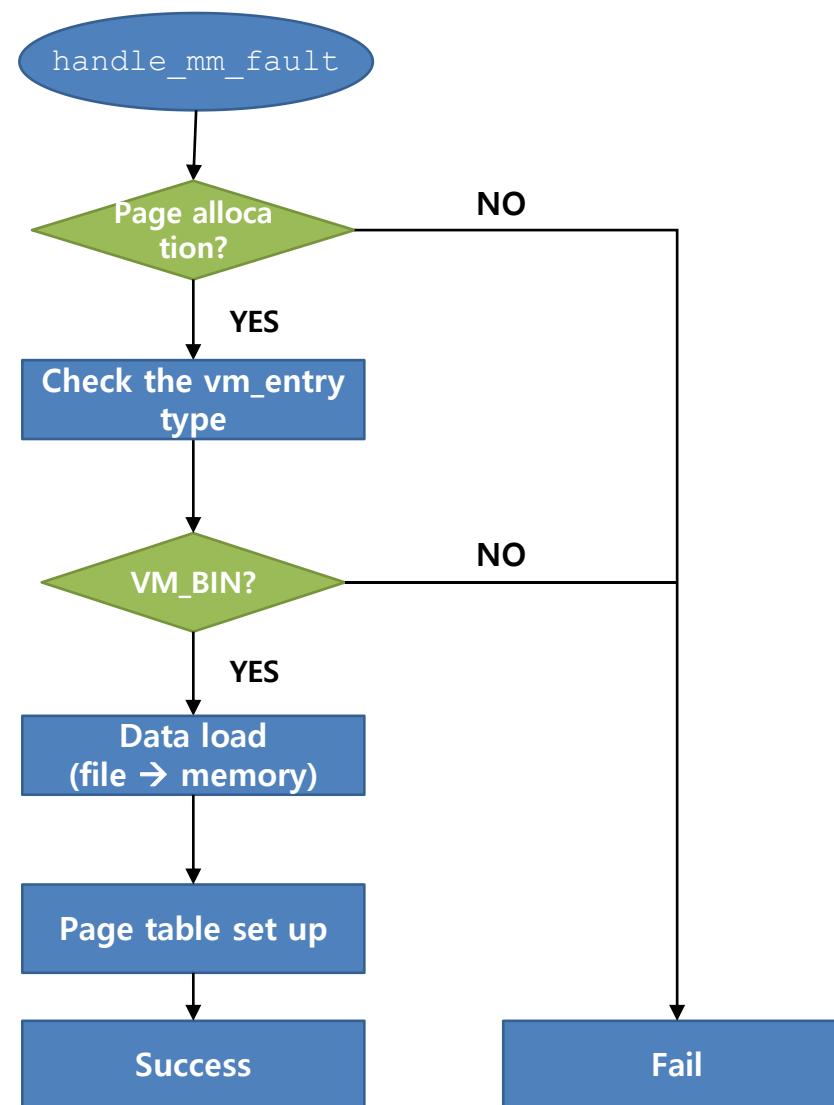
- ◆ `bool handle_mm_fault(struct vm_entry *vme)`
 - `handle_mm_fault` is called to handle page fault.
 - When page fault occurs, allocate physical memory.
 - Load file in the disk to physical memory.
 - Use `load_file (void* kaddr, struct vm_entry *vme)`.
 - Update the associated page table entry after loading into physical memory.
 - Use `static bool install_page(void *upage, void *kpage, bool writab
le)`.

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

page fault handler for loading the ELF file

Later, we will cover anonymous page and the other file backed page.

Here, we only consider the ELF file.



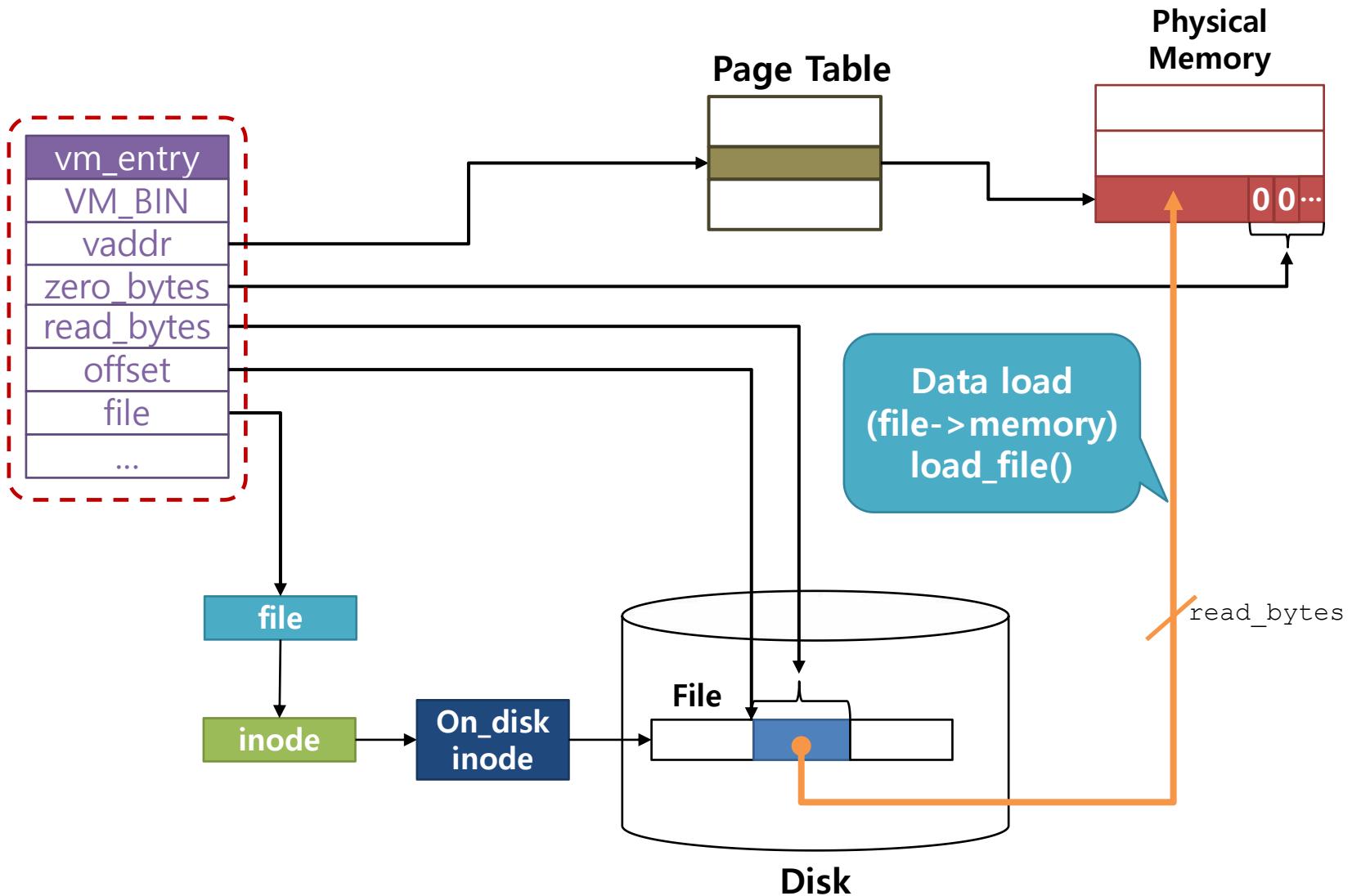
To do 3: load the file to physical memory

- ▣ After physical memory allocation, load the file page from the disk to physical memory(Pintos/src/vm/page.c)
 - ▣ `bool load_file (void* kaddr, struct vm_entry *vme)`
 - ▣ Function to load a page from the disk to physical memory
 - ▣ Implement a function to load a page to kaddr by <file, offset> of vme.
 - ▣ Use `file_read_at()` or `file_read() + file_seek()`.
 - ▣ If fail to write all 4KB, fill the rest with zeros.

```
bool load_file (void *kaddr, struct vm_entry *vme)
{
    /* Using file_read_at()*/
    /* Write physical memory as much as read_bytes by file_read_at*/
    /* Return file_read_at status*/
    /* Pad 0 as much as zero_bytes*/
    /* if file is loaded to memory, return true */

}
```

To do 3: load a file page to physical memory



Functions for demand paging

- ❑ pintos/src/userprog/exception.c

```
static void page_fault (struct intr_frame *f)

/* When page fault occurs, existing code kill(-1) to terminate*/
/* Delete code related to kill(-1) */
/* Modify code to search for vm_entry and allocate page using handle_mm_fault() */
```

- ❑ pintos/src/vm/page.c

```
bool load_file (void* kaddr, struct vm_entry *vme)

/* Load page in disk to physical memory */
/* Implement function to load a page to kaddr by <file, offset> of vme */
/* Use file_read_at() or file_read() + file_seek() */
```

- ❑ pintos/src/userprog/process.c

```
bool handle_mm_fault(struct vm_entry *vme)

/* handle_mm_fault is function to handle page fault */
/* If page fault occurs, allocate physical page */
```

Files to modify

□ Modify Makefile.build

- ◆ Add code to use added page file

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.

...
```

Files to modify (Cont.)

- Modify Makefile.tests
- If not, occurs fail when make check
 - ◆ Test run times may be exceeded depending on the environment.

pintos/tests/Makefile.tests

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

TIMEOUT = 60 /* Change the test run time for Pintos from 60 seconds to 120 seconds */

clean::
    rm -f $(OUTPUTS) $(ERRORS) $(RESULTS)

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

...
```

Additional Functions you may want to implement

```
void vm_init(struct hash* vm)
/* hash table initialization */

void vm_destroy(struct hash *vm)
/* hash table delete */

struct vm_entry* find_vme(void *vaddr)
    /* Search vm_entry corresponding to vaddr in the address space of the
current process */

bool insert_vme(struct hash *vm, struct vm_entry *vme)
    /* Insert vm_entry to hash table*/

bool delete_vme(struct hash *vm, struct vm_entry *vme)
    /* Delete vm_entry from hash table */
```

Functions to add/modify

```
static unsigned vm_hash_func(const struct hash_elem *e, void *aux UNUSED)
/* Calculate where to put the vm_entry into the hash table */

static bool vm_less_func(const struct hash_elem *a, const struct hash_elem *b, void *aux UNUSED)
/* Compare address values of two entered hash_elem */

static void vm_destroy_func(struct hash_elem *e, void *aux UNUSED)
/* Remove memory of vm_entry */
```

Verify virtual memory project

- Confirm code behavior after completing virtual memory task

- ◆ path : pintos/src/vm

```
$ make check
```

- 28 of 109 tests found to fail as a result of execution

- ◆ 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

The screenshot shows a terminal window with the following text:

```
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$
```

A yellow box highlights the line "28 of 109 tests failed."

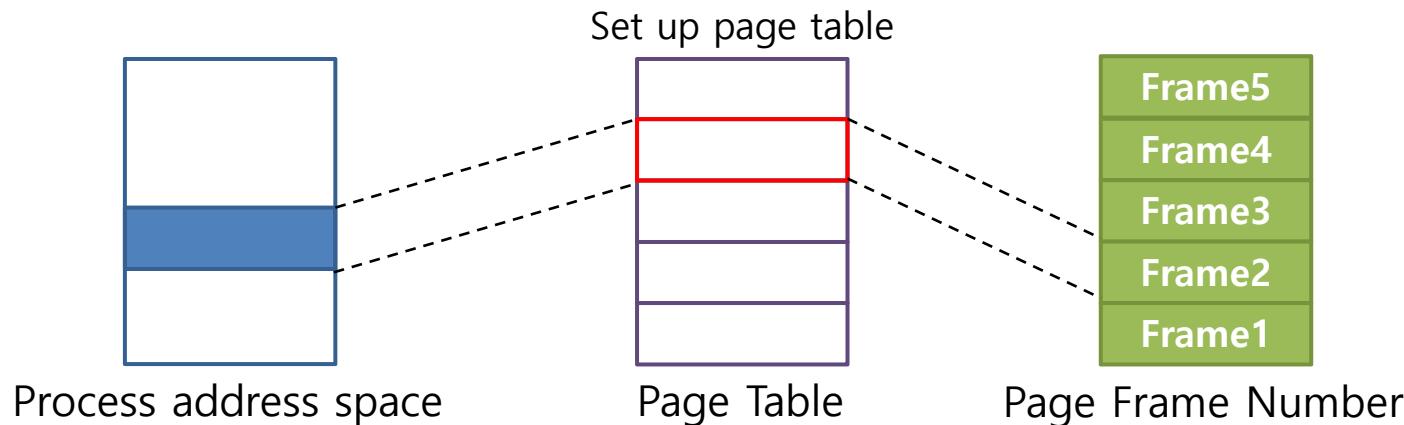
Appendix

Page address mapping function

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

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

- ◆ Map physical page kpage and virtual page upage
- ◆ writable: writable(1), read-only(0)



Physical page allocation and releasing interface

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

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

- ◆ Allocate a 4KB page.
- ◆ Return physical address of page.
- ◆ flags
 - PAL_USER: allocate pages from user memory pool.
 - PAL_KERNEL: allocate pages in kernel memory pool.
 - PAL_ZERO: initialize pages to '0'.

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

- ◆ Use physical address of page as argument.
- ◆ Put page back in free memory pool.

Pintos dynamic memory allocation and releasing interface

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

```
void *malloc(size_t size)
```

- ◆ Allocate the memory chunk of ‘size’ and return start address.
- ◆ Use to allocate memory for dynamic objects such as `vm_entry`.

```
void free(void* p)
```

- ◆ Release the memory space allocated by `malloc()`.
- ◆ Use address allocated memory through `malloc()` as argument.

Operating Systems Lab

Part 3: Virtual Memory

KAIST EE

Youjip Won

Overview of Virtual Memory

- Background of Virtual Memory in Pintos
- Requirements
 - ◆ Paging(swapping)
 - ◆ Growing stack
 - ◆ Memory mapped file
 - ◆ Accessing user memory

Swapping

To Do's

- Implement data structure to represent physical page frame.
- Implement page replacement policy such as LRU, clock, second-chance
- swapping
 - ◆ Store victim pages in swap space when they belong to data segment or stack segment.
 - ◆ swap-out pages are reloaded into memory by demand paging.

Hardware Support

- The dirty bit of page table is set to “1” by hardware when writing to the memory space
- The accessed bit in page table is set to ‘1’ by hardware each time the page is referenced



- When page with dirty bit “1” is selected as victim, the changes must always be stored on disk
- Hardware does not re-zero the accessed bit.

Page Table Manipulation in Pintos (userprog/pagedir.c)

- ◆ `bool pagedir_is_dirty (uint32_t *pd, const void *vpage)`
 - Return dirty bit of pte for vpage in pd
- ◆ `void pagedir_set_dirty (uint32_t *pd, const void *vpage, bool dirty)`
 - Set the dirty bit to dirty in the pte for vpage in pd
- ◆ `bool pagedir_is_accessed (uint32_t *pd, const void *vpage)`
 - Return access bit of pte for vpage in pd
- ◆ `void pagedir_set_accessed (uint32_t *pd, const void *vpage, bool accessed)`
 - Set the access bit to accessed in the pte for vpage in pd

struct page: New data structure required

Select the physical page frame for replacement.

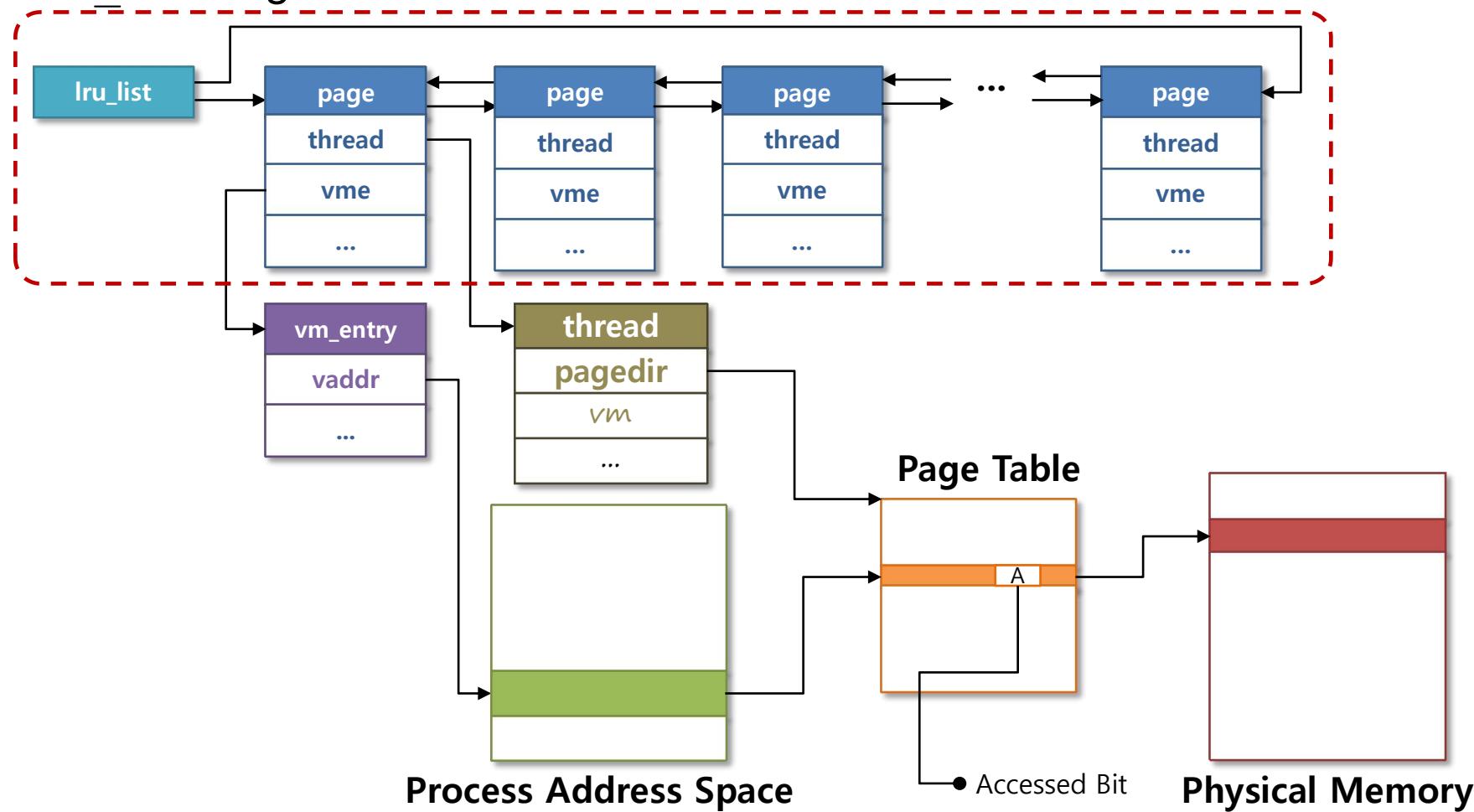
- ▣ Data structure representing each physical page that contains a user page
 - ◆ physical address of page
 - ◆ reference to the virtual page object to which physical page is mapped
 - ◆ Reference to the thread structure to which it belongs
 - ◆ lru: field for list

pintos/src/vm/page.h

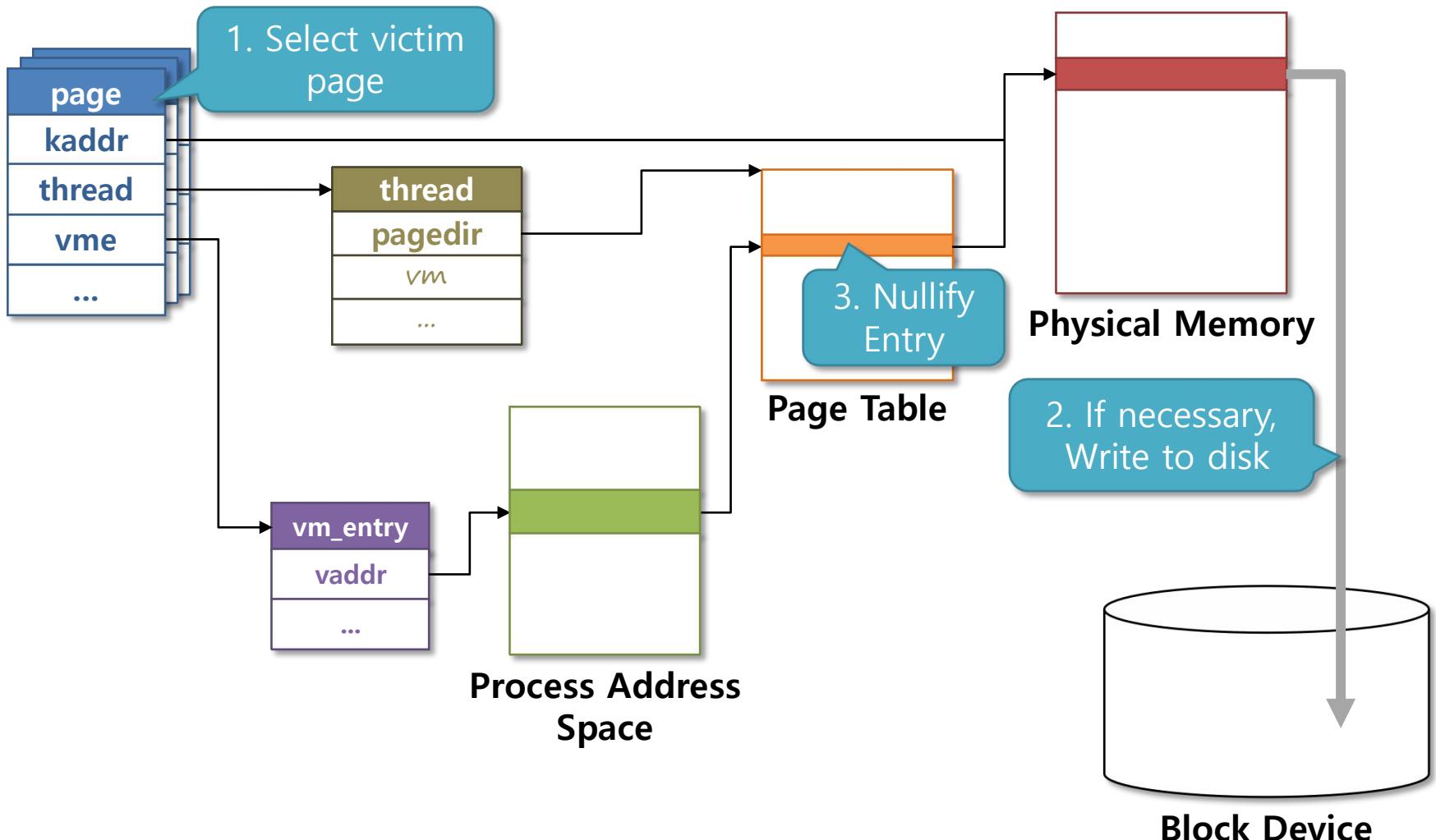
```
struct page {  
    // fill this out  
};
```

A page pool for swapping

- Manage physical pages in use as a list of pages.
- `lru_list`: global variable

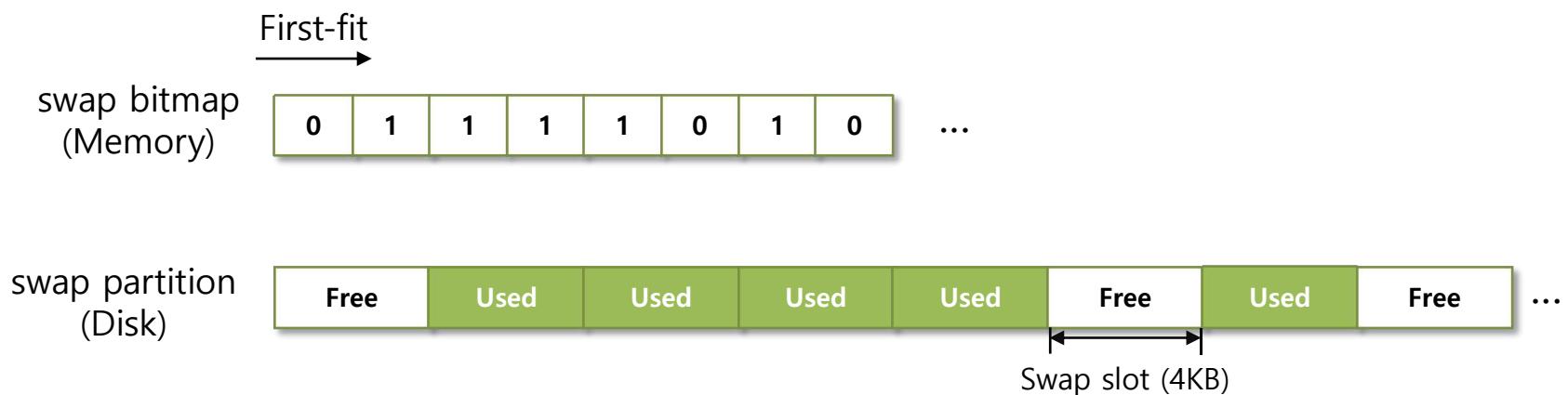


Swap-out



Managing swap partition

- Swap partition is managed per swap slot unit(4 Kbyte).
- Maintaining a swap partition: swap bitmap (global variable in memory)
- Search bitmap for free slot.
- What happens to swap bitmap if the system crashes?



Functions offered by pintos for swap space manipulation

- ▣ Swap partition is provided as block device in pintos.
- ▣ Functions for block device (src/block/block.c)
 - ◆ `struct block *block_get_role (enum block_type role)`
 - Return the block device (`struct block *`) fulfilling the given ROLE.
 - ROLES defined in pintos now (devices/block.h)
 - `BLOCK_KERNEL`: OS Partition
 - `BLOCK_FILESYS`: File system
 - `BLOCK_SCRATCH`: Scratch partition
 - `BLOCK_SWAP`: Swap partition
 - ◆ `void block_read (struct block *block, block_sector_t sector, void *buffer)`
 - Read contents at sector on block and save them at buffer
 - ◆ `void block_write (struct block *block, block_sector_t sector, const void *buffer)`
 - Write contents at buffer at sector on block

Implementation

- ❑ LRU list for physical page frame
 - ◆ List of struct page
 - ◆ List of physical pages allocated to user process
- ❑ functions for allocate/release physical page frame from the list
 - ◆ When there runs out of physical page frame, select a victim and swap it out.
- ❑ Modify page fault handler for swapping.
 - ◆ Before: Physical page is allocated directly when page fault occurs.
When there is no page to allocate, pintos is finished.
 - ◆ After: Physical page is allocated from LRU list when page fault occurs.
When there is no page to allocate, pintos swap in the page.

Functions to write

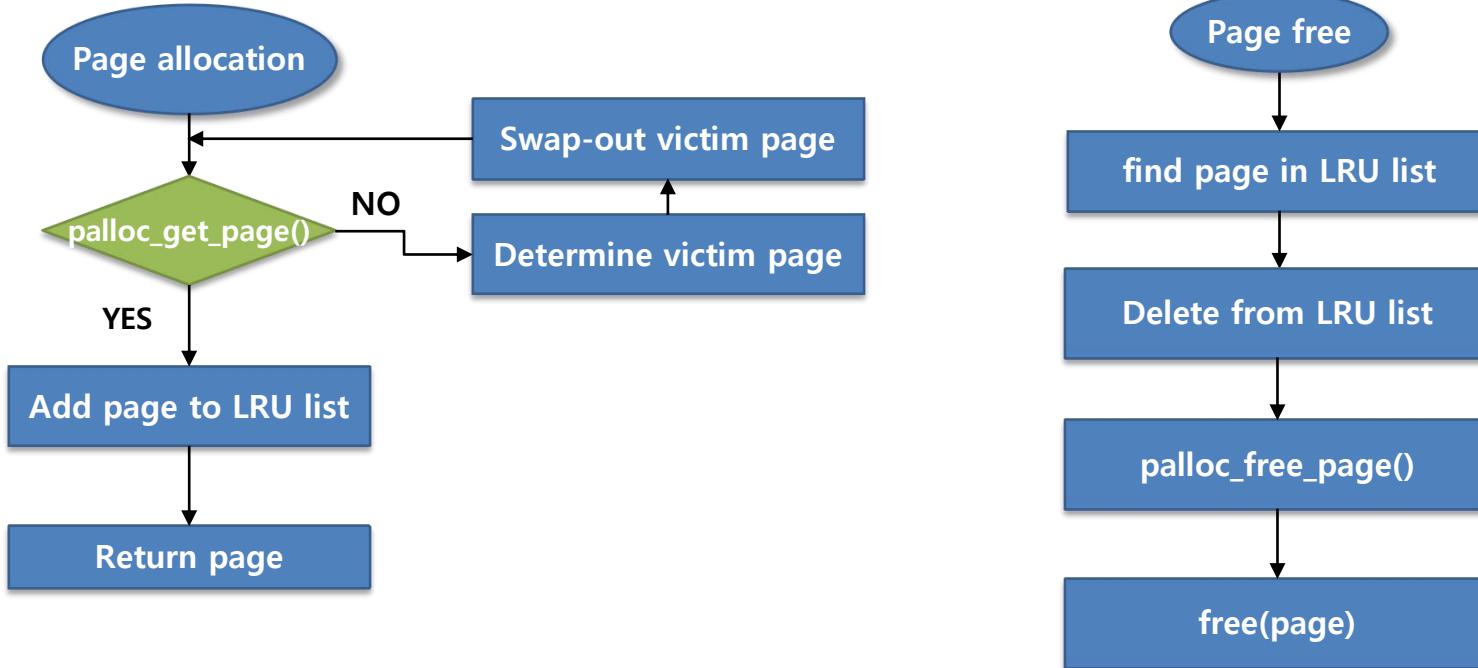
- ▣ Function about LRU list (initializing, insert, remove).
- ▣ Function to allocate a page from LRU list.
- ▣ Function to free page from LRU list.
- ▣ Function to select victim page and swap-out the page.
 - ◆ e.g.: Clock algorithm, Second chance algorithm
- ▣ Function about swapping (initializing, swap in, swap out).

Functions to modify

- ▣ `bool handle_mm_fault(struct vm_entry *vme)`
 - ◆ Modify to allocate physical pages from LRU list when page fault occurs
 - ◆ Modify to swap-in if `vm_entry type is VM_ANON`
- ▣ `static bool setup_stack(void **esp)`
 - ◆ Modify to allocate pages from LRU list when page fault occurs
- ▣ `int main(void)`
 - ◆ Initialize LRU list.

Functions for allocation/free page

- Try to obtain free space when memory cannot be allocated through `palloc_get_page()` within the page allocation function.

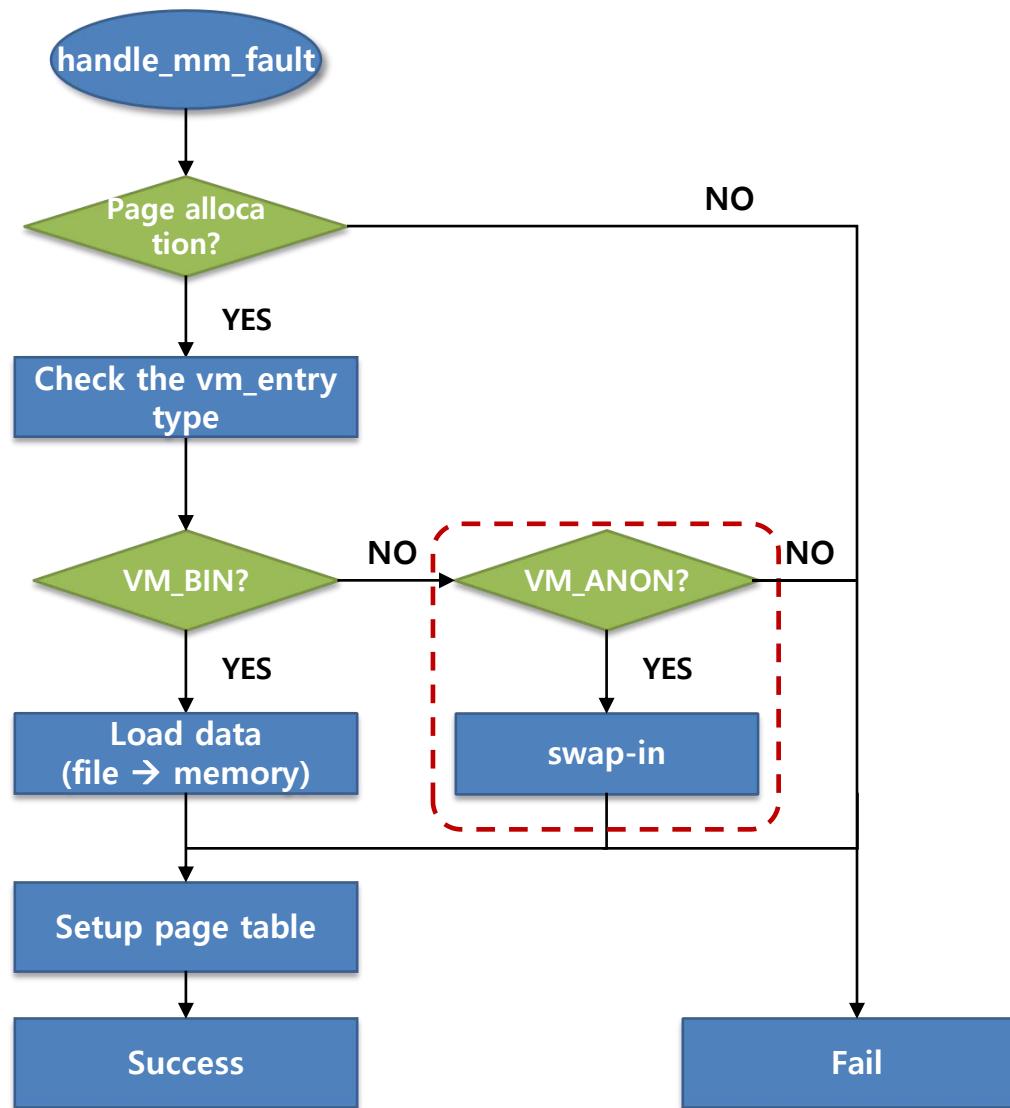


Swap-out

- Type of a page in the physical page frame
 - ◆ VM_BIN
 - If dirty bit is “1”, write to the swap partition and free the page frame.
 - Change type to VM_ANON for demand paging
 - ◆ VM_FILE
 - If dirty bit is “1”, write the page to the file and free the page frame.
 - If dirty bit is “0”, free the page frame.
 - ◆ VM_ANON
 - Write to the swap partition.
- Mark the page “not present” in pd (page directory).

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

Demand paging for anonymous page (stack or heap)



Modify handle_mm_fault()

- If vm_entry type is VM_ANON, modify code to swap in

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_ANON:
            /* insert swap in code */
            break;
    }
    ...
}
```

Growing Stack

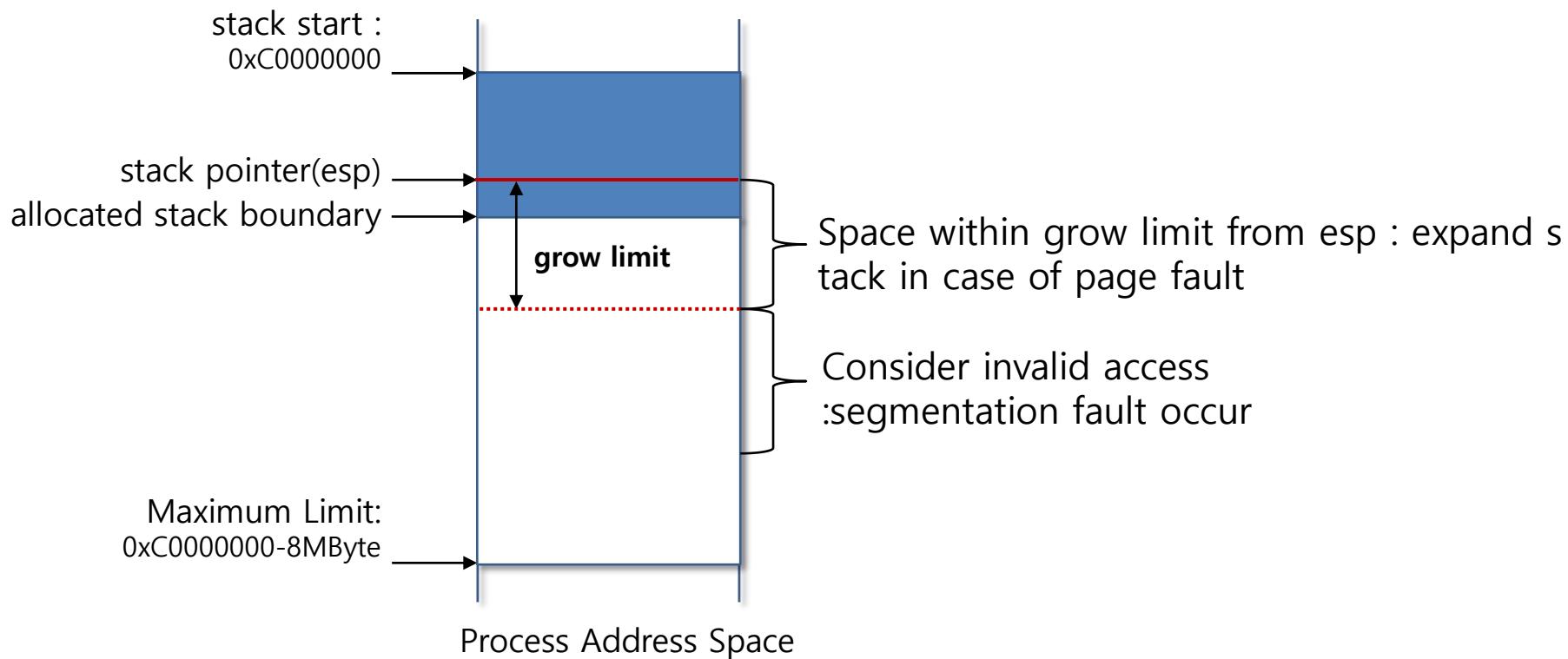
Expandable Stack

❑ Implement expandable stack

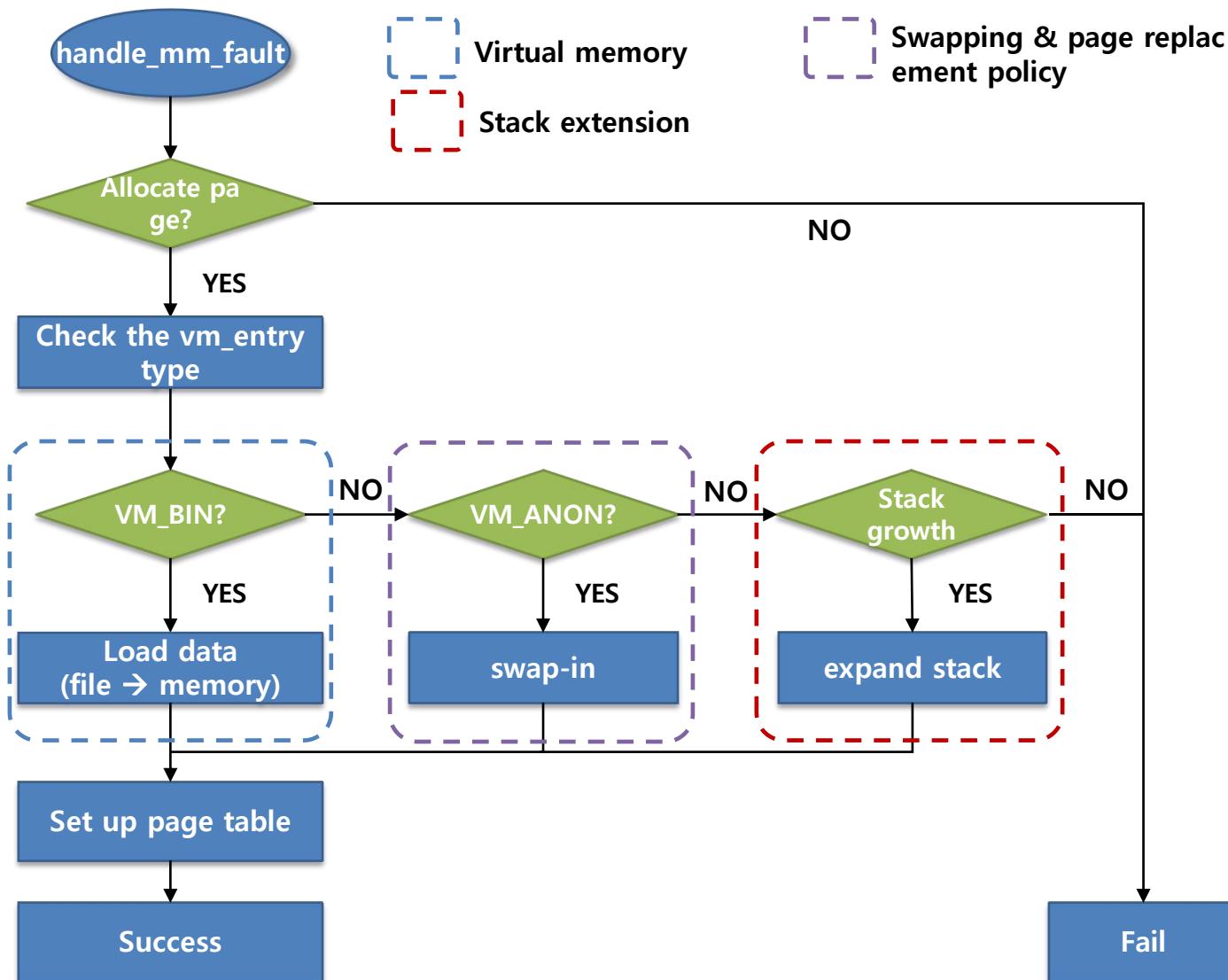
- ◆ In current pintos, stack size is fixed to 4KB.
- ◆ Make the stack expandable.
 - If a process accesses the address that lies outside the stack and that can be handled by expanding the stack, expand the stack.
 - e.g. (access address < stack pointer – 32) Expand stack
- ◆ maximum size of stack is 8MB.

When to expand stack

- Expand the stack when the memory access is within 32 Byte of stack top.
 - “PUSHA” instruction in 80x86 pushes 32 bytes at once.



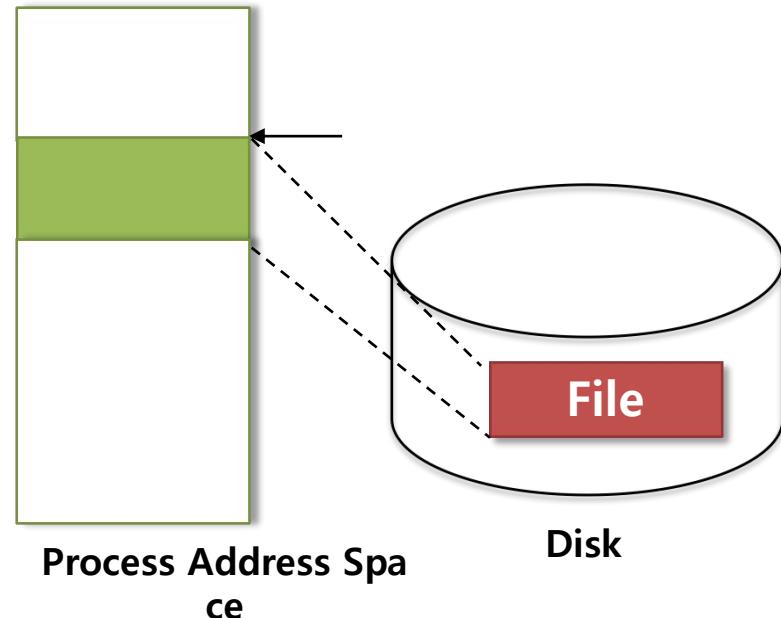
Stack extension mechanism



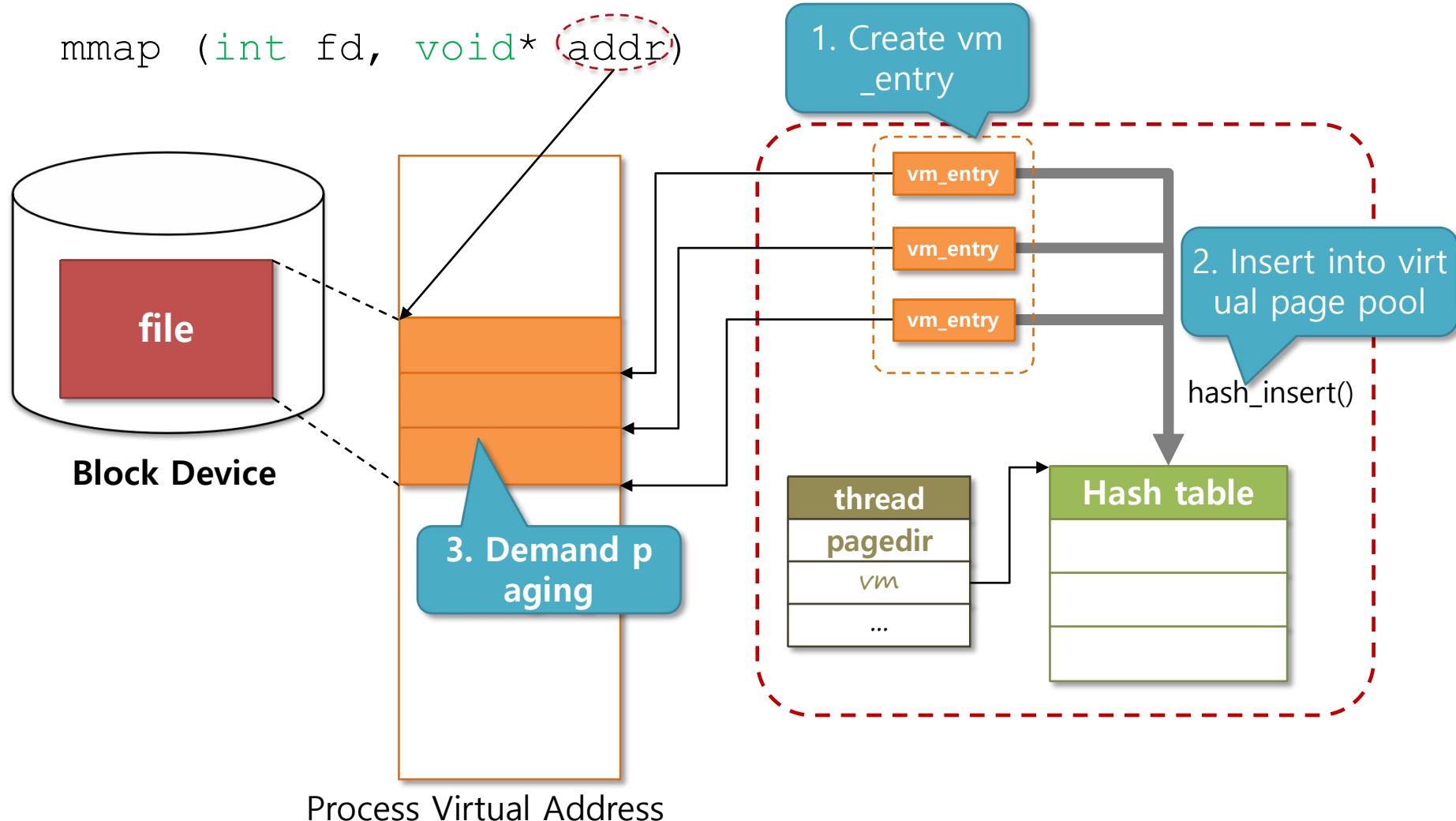
Memory Mapped File

mmap vs. munmap

```
main (int argc, char *argv[]) {  
    int i;  
  
    for (i = 1; i < argc; i++) {  
        int fd;  
        mapid_t map;  
        void *data = (void *) 0x10000000;  
        int size;  
        fd = open (argv[i]);  
        size = filesize (fd);  
        map = mmap (fd, data);  
        write (STDOUT_FILENO, data, size);  
        munmap (map);  
    }  
    return EXIT_SUCCESS;  
}
```



mmap and munmap



mmap() and munmap()

- `int mmap(int fd, void *addr)`
 - ◆ Load file data into memory by demand paging.
 - ◆ `mmap()`'ed page is swapped out to its original location in the file.
 - ◆ For a fragmented page, fill the unused fraction of page with zero.
 - ◆ Return mapping_id: unique id within a process to identify the mapped file.
 - ◆ Fails if
 - File size is 0.
 - Addr is not page aligned.
 - Address is already in use.
 - Addr is 0.
 - STDIN and STDOUT are not mappable..
- `void munmap(mapid_t mapid)`
 - ◆ Unmap the mappings in the `mmap_list` which has not been previously unmapped.

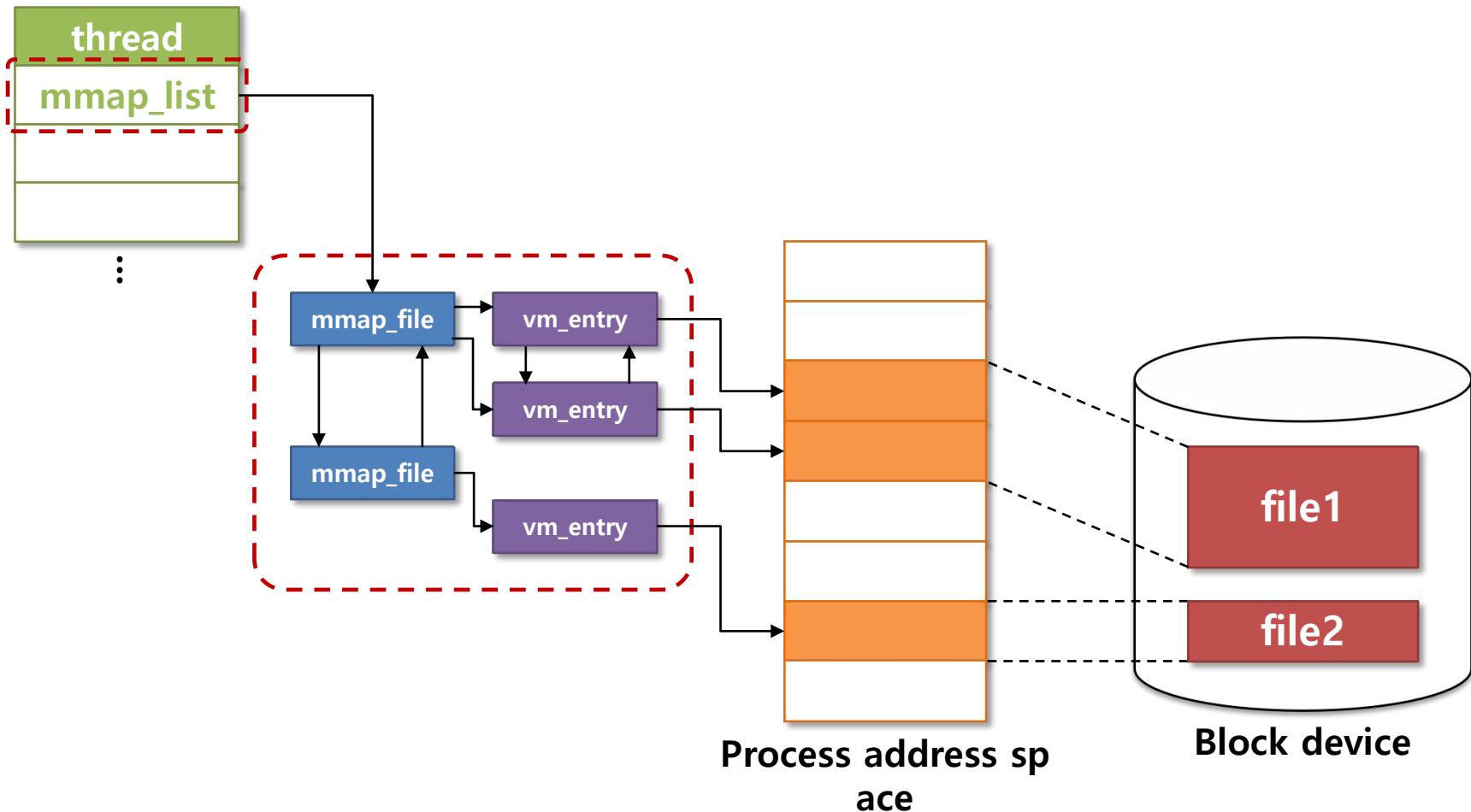
Requirements

- All mappings of a process are implicitly unmapped when the process exits.
- When a mapping is unmapped, the pages are written back to the file.
- Upon munmap, the pages are removed from the process' virtual page list.
- Once created, mapping is valid until it is unmapped regardless of the file is closed or deleted.
- If the two or more processes map the same file, they do not have to see the consistent view.

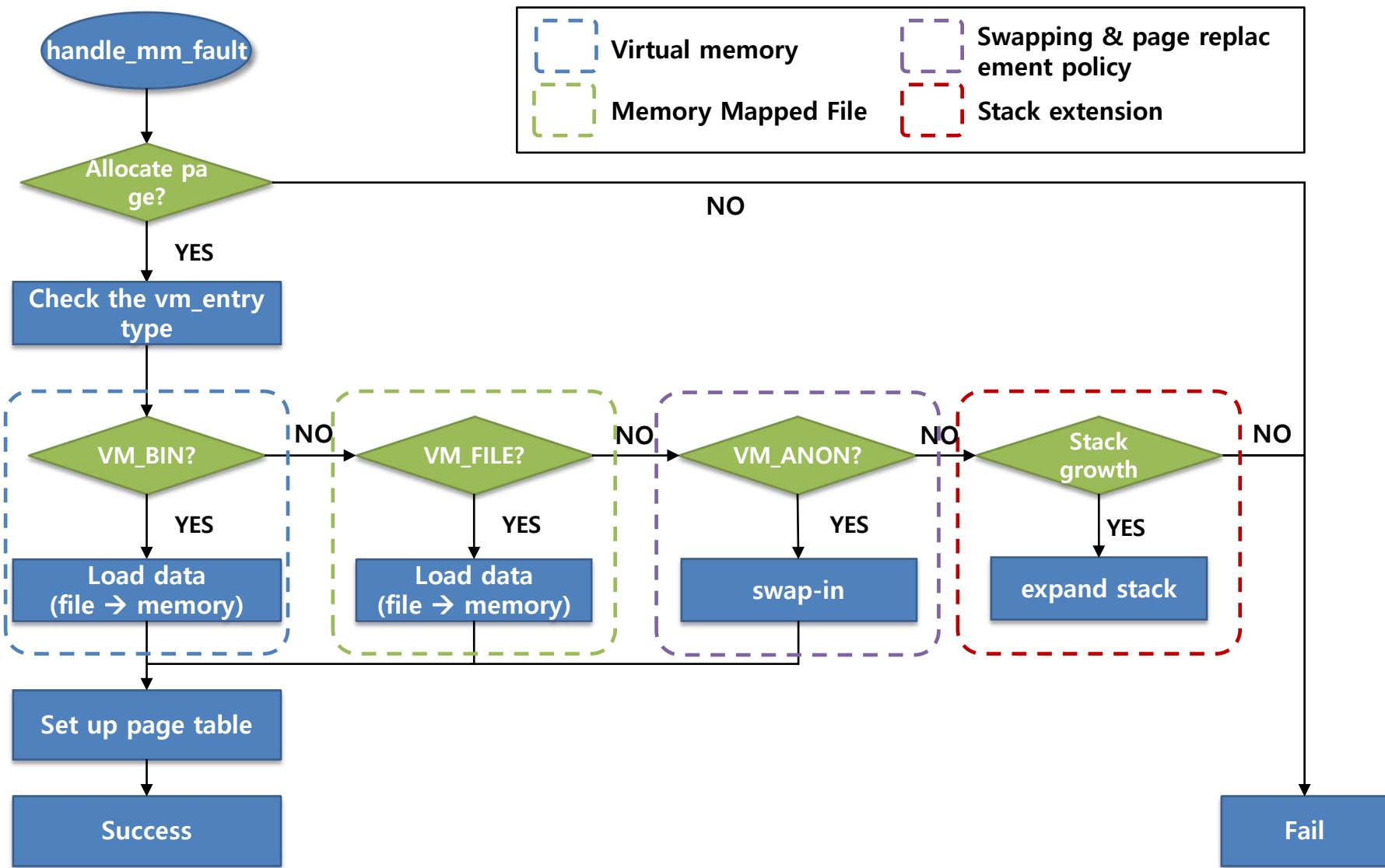
Additional data structure and Functions to modify

- ▣ `struct mmap_file`
 - ◆ Data structure containing information from mapped files
 - ◆ mapping id
 - ◆ mapping file object
 - ◆ `mmap_file` list element
 - ◆ `vm_entry` list.
- ▣ `bool handle_mm_fault(struct vm_entry *vme)`
 - ◆ Load data if `vm_entry` type is `VM_FILE`
- ▣ `void process_exit (void)`
 - ◆ Release all `vm_entry` corresponding to `mapping_list` at the end of process.

Managing mapped files



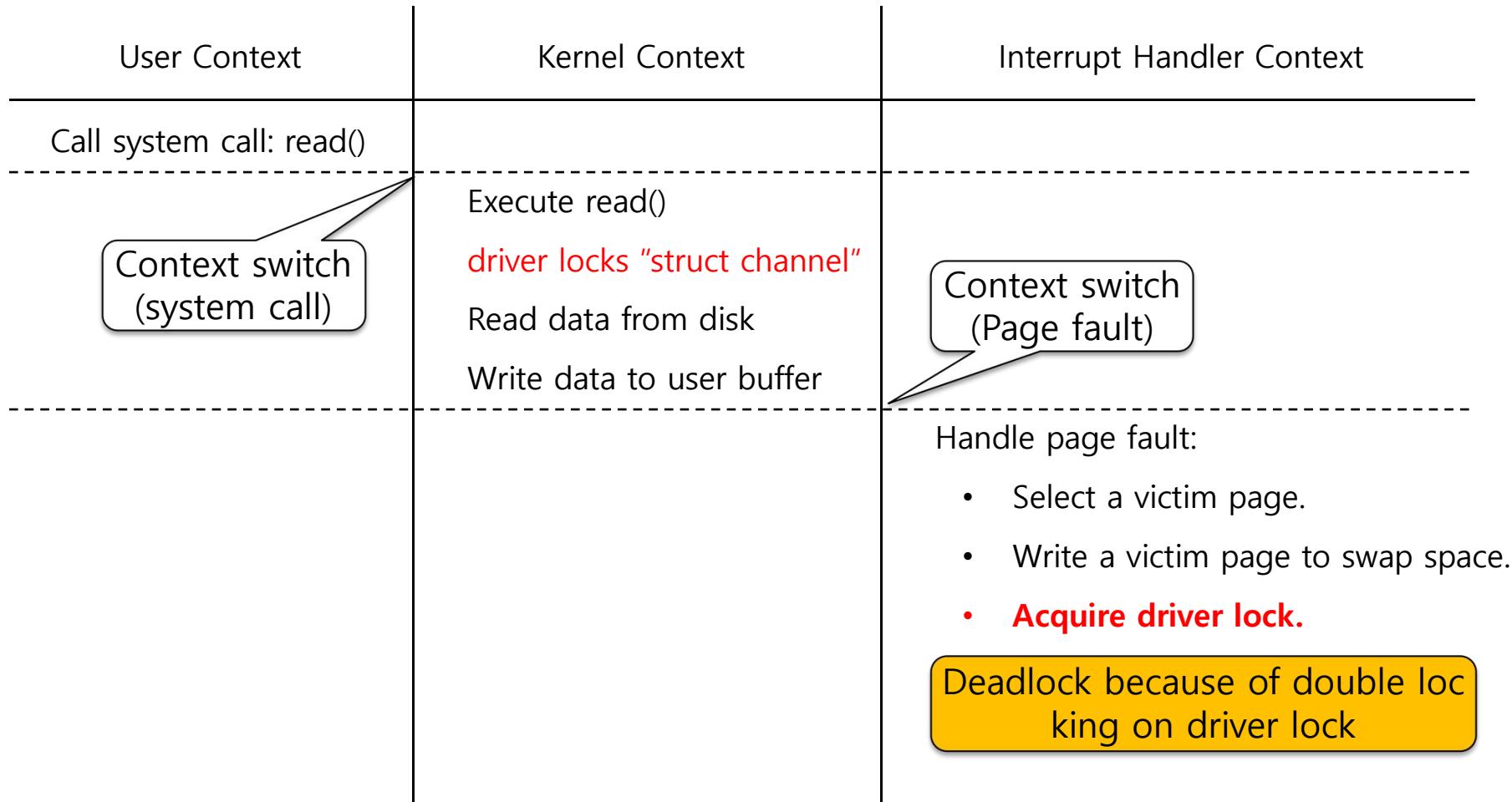
Modify page fault handler for mmap()



Accessing User Address Space

Why should page fault not occur in kernel code?

- A deadlock on kernel resource can occur.



Pinning Page

- Prevent evicting the pages accessed during system call
- Define pinning flag about each physical page.
- On every system call,
 - ◆ Find the virtual page and pin the associated physical page.
 - ◆ After the system call returns and before the system call handler returns, unpin the pages
- On Swapping handler,
 - ◆ Do not select a pinned page as a victim.

Operating System Lab

Part 4: Filesystem

KAIST EE

Youjip Won

Overview of Filesystem

- Background of Filesystem in Pintos
- To Do's in project 4
 - ◆ Buffer Cache
 - ◆ Indexed and Extensible Files
 - ◆ Subdirectories

Background

Basic concepts

- inode

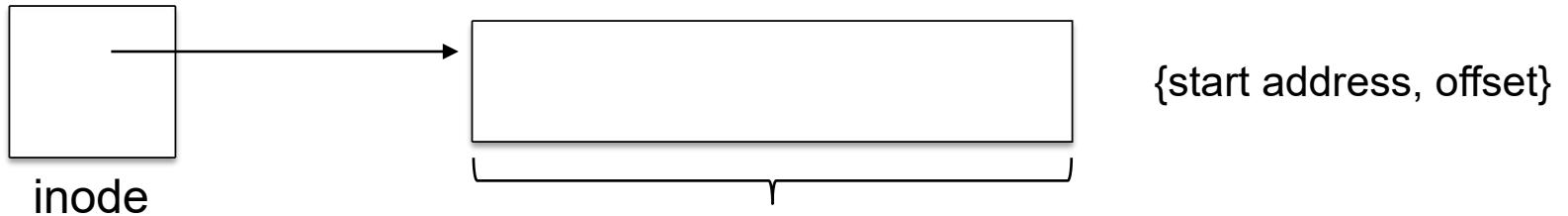
- ◆ Represents a file on the disk
- ◆ File size
- ◆ Pointers to the disk block(s)
- ◆ Attributes: permission, access time, modification time and etc.
- ◆ On disk inode
- ◆ In-memory inode = on-disk inode + on-disk location of the inode

- File object

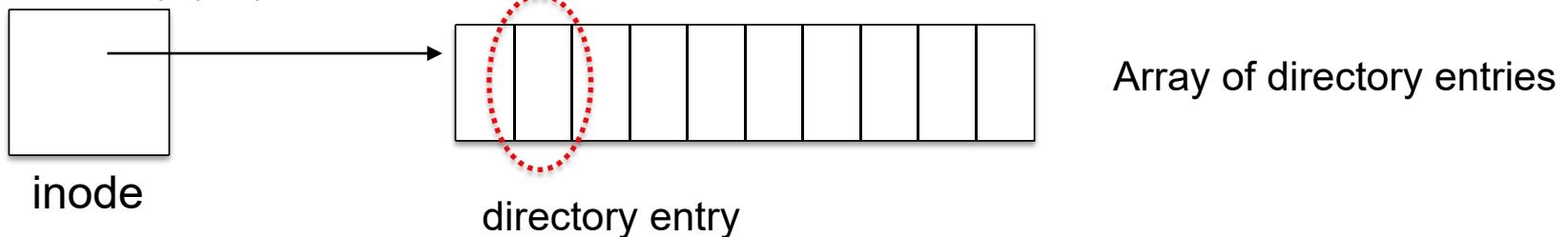
- ◆ Represent an “open” file.
- ◆ Current offset to perform read/write
- ◆ Filesystem type it belongs: EXT4

concepts

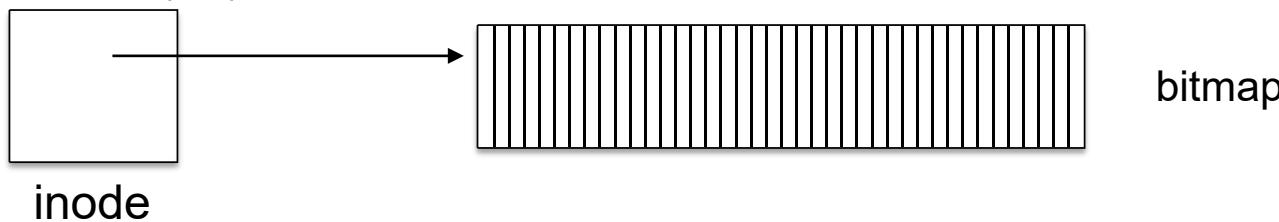
Regular file



Directory (file)



Bitmap (file)

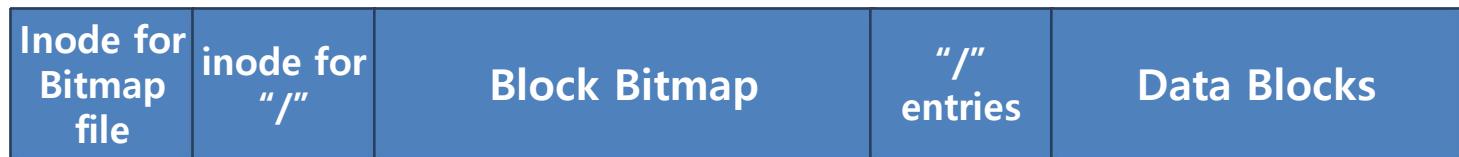


Filesystem layout in pintos

Example of 8MByte Filesystem in Pintos

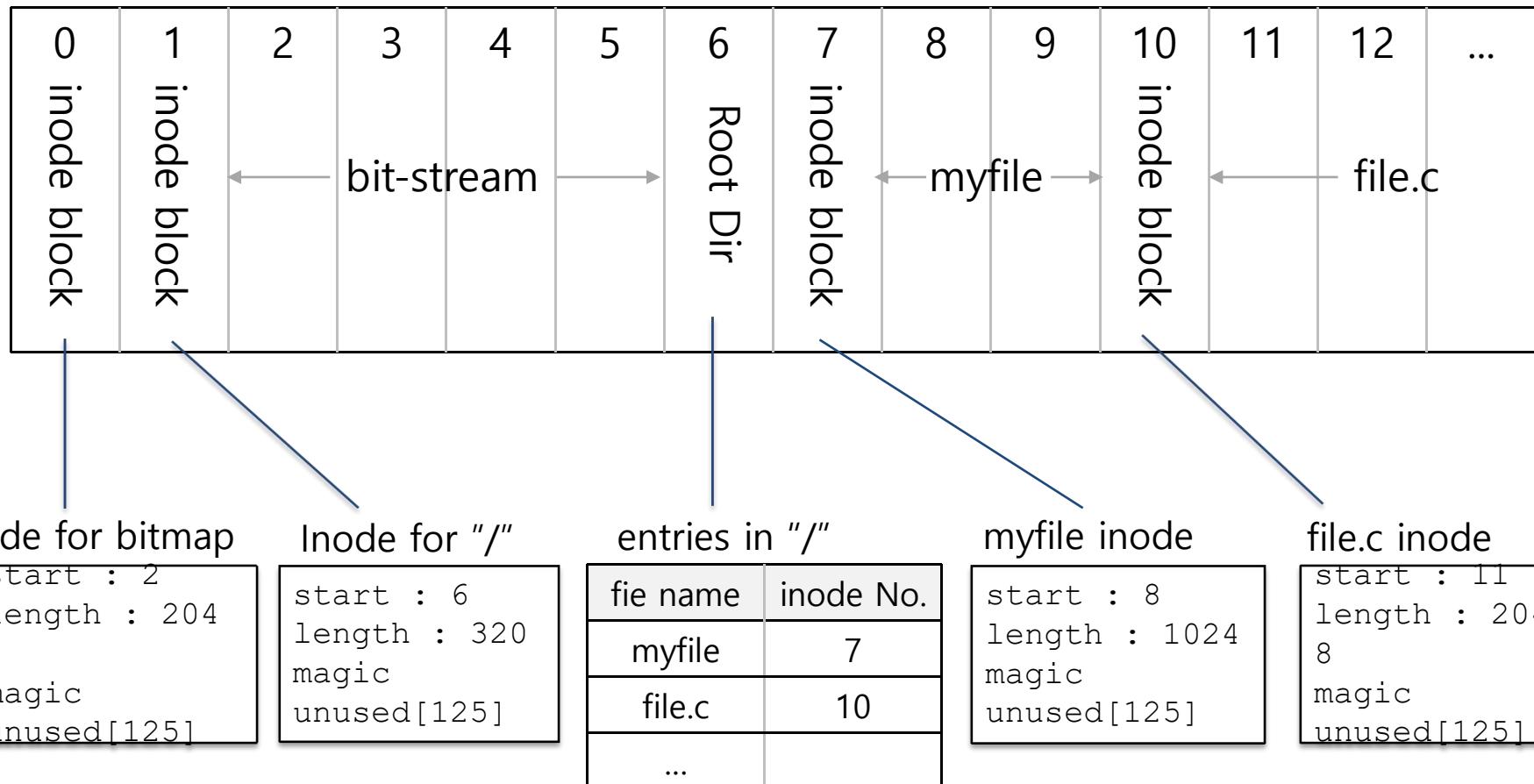
- ◆ Block Size (=Sector size): 512 Byte
- ◆ $8\text{MByte}/512\text{Byte} = 16,384 = \text{Total number of blocks}$
- ◆ Block bitmap: $4*512*8 \text{ bits} = 16,384 \text{ bit}$
- ◆ In PINTOS, block bitmap is represented as a "file".

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



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

Sample Filesystem Layout in Pintos



In-memory inode

- In-memory inode: struct inode

- ◆ sector : block number where inodes are stored
- ◆ data : disk_inode data
- ◆ removed : Whether to delete the file

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. */  
};
```

On-disk inode

▣ On-disk inode (struct inode_disk)

- ◆ start : start address of file data in block address
- ◆ length : size of file(byte)
- ◆ unused[125] : unused area
- ◆ One inode occupies a single sector.

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. */
};
```

Directory object

- Represent an open directory.
 - ◆ inode : the pointer to the associated in-memory inode
 - ◆ pos : position of the next directory entry to read/write

pintos/src/filesys/directory.c

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

Directory entry

- Indicate information in directory entry (file or directory)
 - inode_sector : sector number of the inode (inode size is 512 byte)
 - file name: up to 14 characters
 - in_use : Whether to use dir_entry

pintos/src/filesys/directory.c

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

Block bitmap

▣ free_map

- ◆ bitmap to represent status of the blocks in the filesystem partition
- ◆ free_map_file : bitmap is stored as a file.
- ◆ bit_cnt : Number of disk blocks in entire file system

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;
};
```

File structure

▣ struct file

- ◆ Created when a file is open.
- ◆ inode : pointer to the file's in-memory inode
- ◆ pos : current file offset
- ◆ deny_write: indicate whether a file is writable.

pintos/src/filesys/file.c

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

To Do's

- ▣ Buffer cache
 - ◆ Allocate buffer cache (64 blocks).
 - ◆ Cache data blocks.
 - When read or write data blocks, read and save it in buffer cache
 - ◆ Write dirty data blocks.
 - When dirty data blocks is evicted to reclaim buffer cache entry
 - When filesystem is shut down
- ▣ Indexed and Extensible File
 - ◆ Implement block pointers in inode
 - direct, single-indirect, double-indirect
- ▣ Subdirectories
 - ◆ Implement hierarchical name space for file

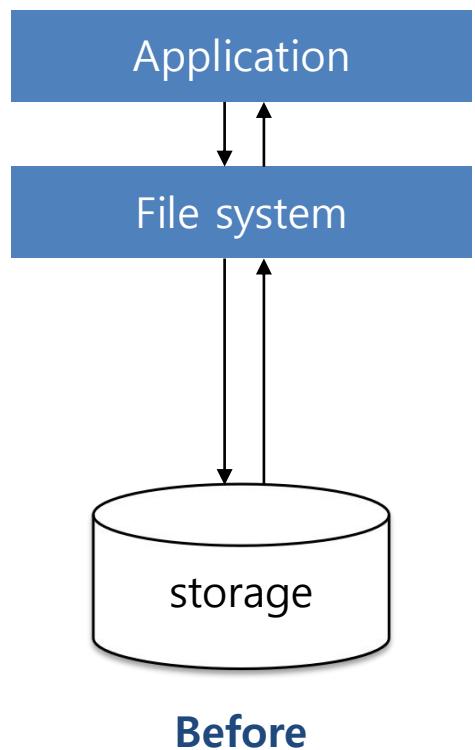
Buffer Cache

Buffer cache

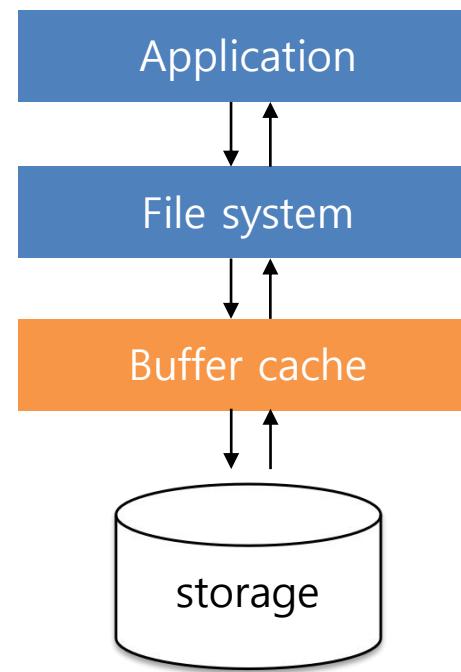
- ▣ Buffer Cache: using the part of the memory as disk.
 - ◆ Consecutive physical pages
 - ◆ Initialized when the system starts or when a filesystem is mounted.
 - ◆ Virtual Memory: using the part of the disk as memory.
- ▣ In current pintos, there is no cache for disk I/O.
- ▣ In reality, most OS's have a cache for disk I/O.
- ▣ Modify the filesystem to cache the file blocks.
 - ◆ Cache the data blocks.
 - ◆ Capacity of cache: 64 blocks
- ▣ File to modify
 - ◆ `pintos/src/filesys/inode.c`

Background - Pintos's read/write

- ¤ Current Pintos accesses storage on user read/write requests.
- ¤ Change user I/O to be performed through buffer cache (memory)



Before



After

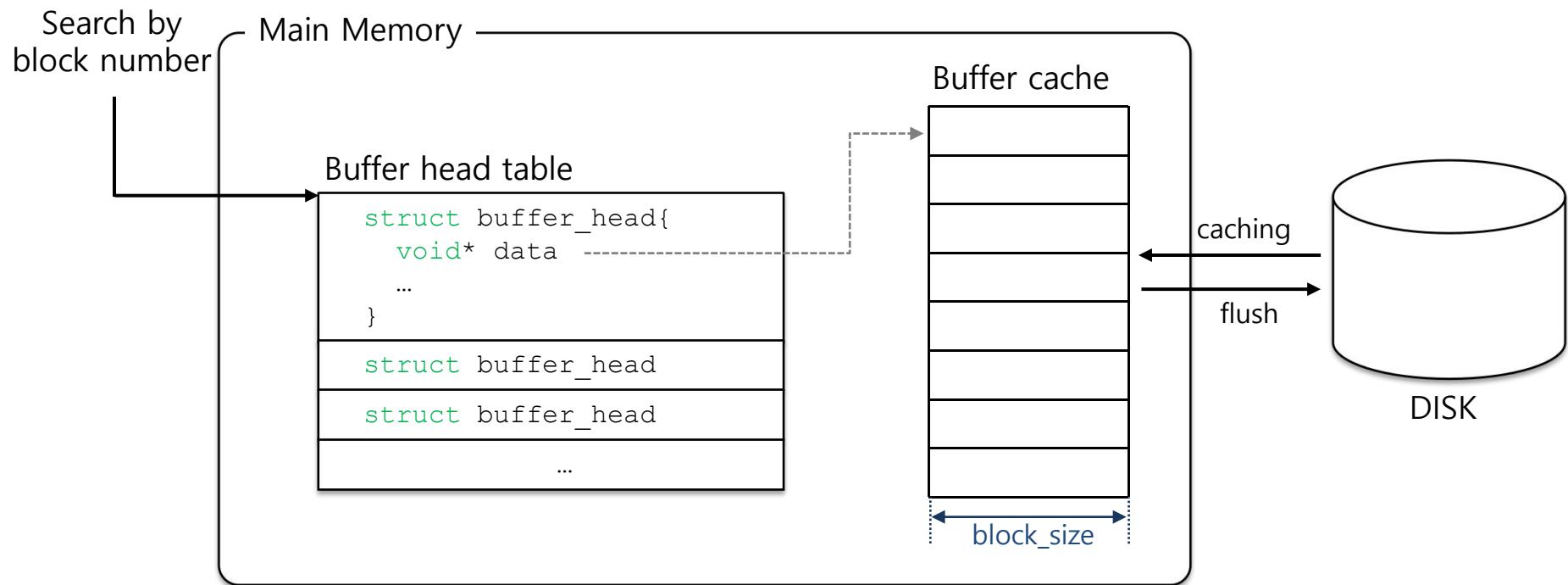
To do's

1. Define structures for buffer cache
2. Allocate and initialize buffer cache
3. When read data from file, read data from buffer cache
4. When write data to file, write data to buffer cache
5. If cache miss, read data from disk and save them in buffer cache
6. If buffer cache is full, evict buffer cache entry and reclaim free one
7. Write dirty buffer cache (sync)

To Do 1: Define structures for buffer cache

- ▣ Metadata for a buffer cache entry (ex: `struct buffer_head`)
 - ◆ The "dirty" flag
 - ◆ The flag indicating whether the entry being used or not
 - ◆ The "access" flag indicating whether the entry is accessed recently or not
 - ◆ The on-disk location
 - ◆ The virtual address of the associated buffer cache entry
- ▣ Maintain all 64 `struct buffer_head` by data structure
 - ◆ Array, List, or Hash table.

Buffer cache diagram



To Do 2: Allocate and initialize buffer cache

- ▣ Allocate memory for buffer cache for 64 block
 - ◆ Block size (=Sector size): 512 Byte
 - ◆ $64 * 512 \text{ Byte} = 32 \text{ Kbyte}$
- ▣ Allocate memory for `struct buffer_head`'s
 - ◆ `sizeof (struct buffer_head) * 64`

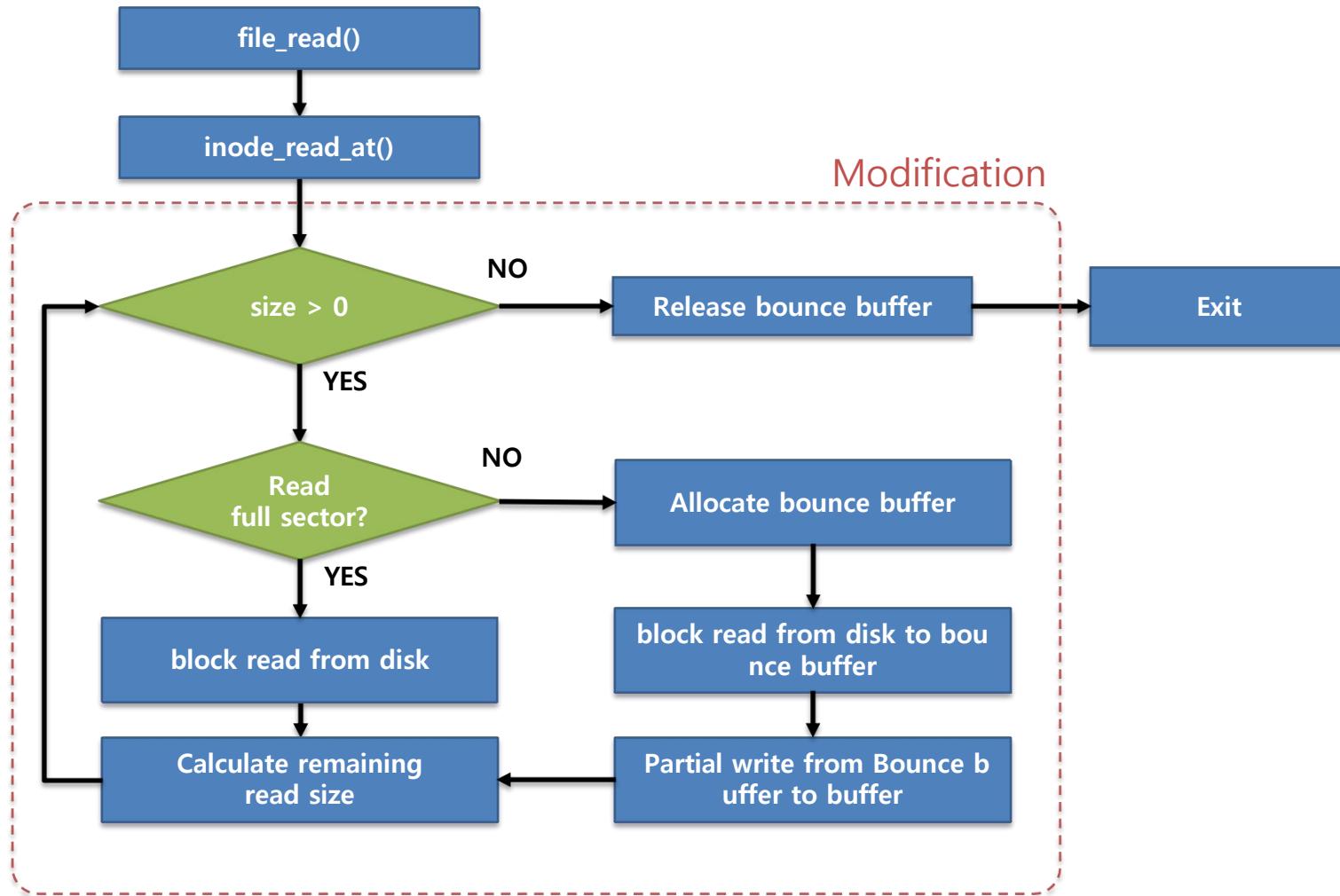
pintos/src/filesys/filesys.c

```
void filesys_init (bool format) {
    fs_device = block_get_role (BLOCK_FILESYS);
    ...
    if (format)
        do_format ();

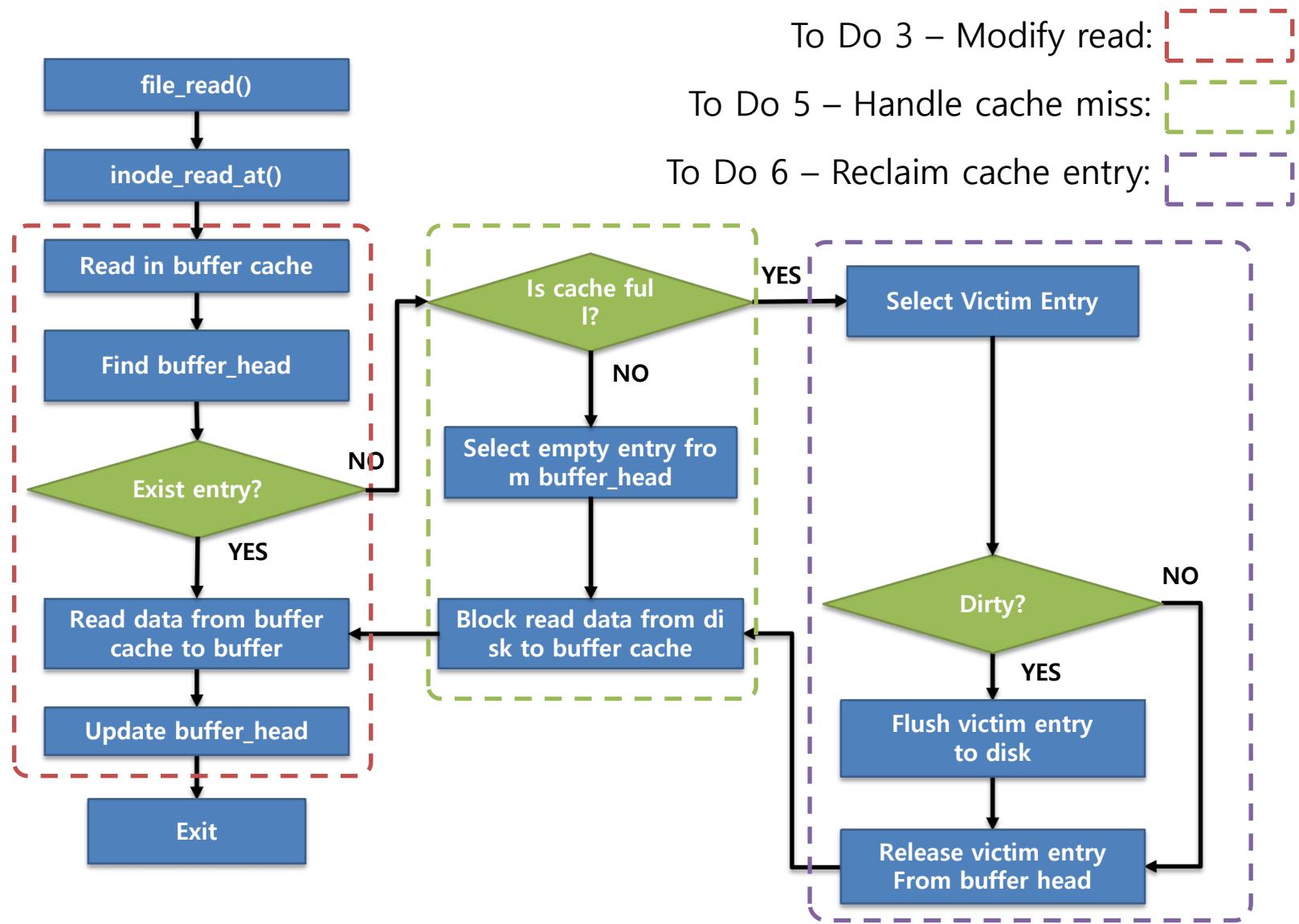
    free_map_open ();

    /* Add code here */
}
```

READ in current pintos



Read with Buffer Cache



Modify disk read to buffer cache read

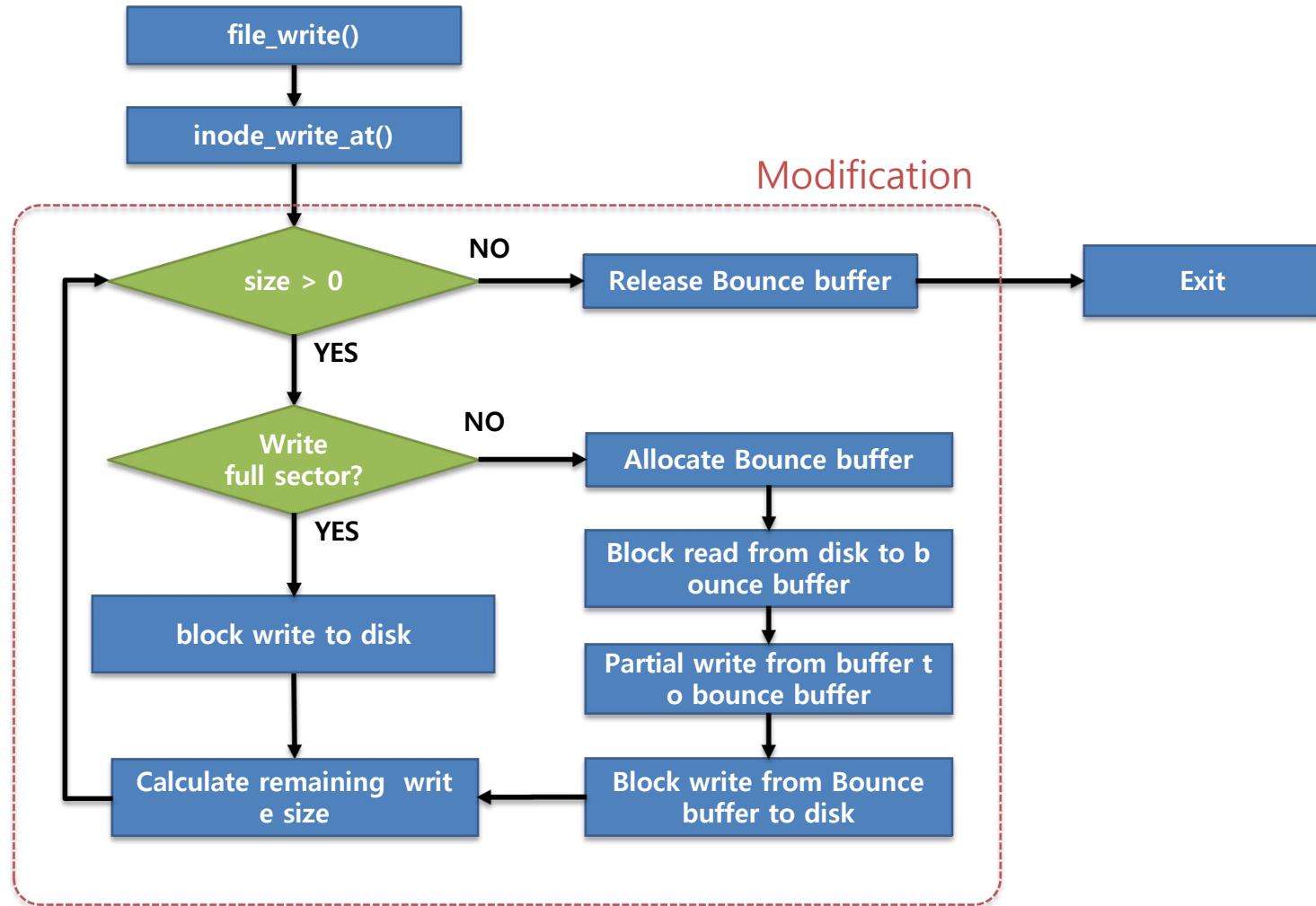
- When reading file, modify the read to read the data from the buffer cache.

pintos/src/filesys/inode.c

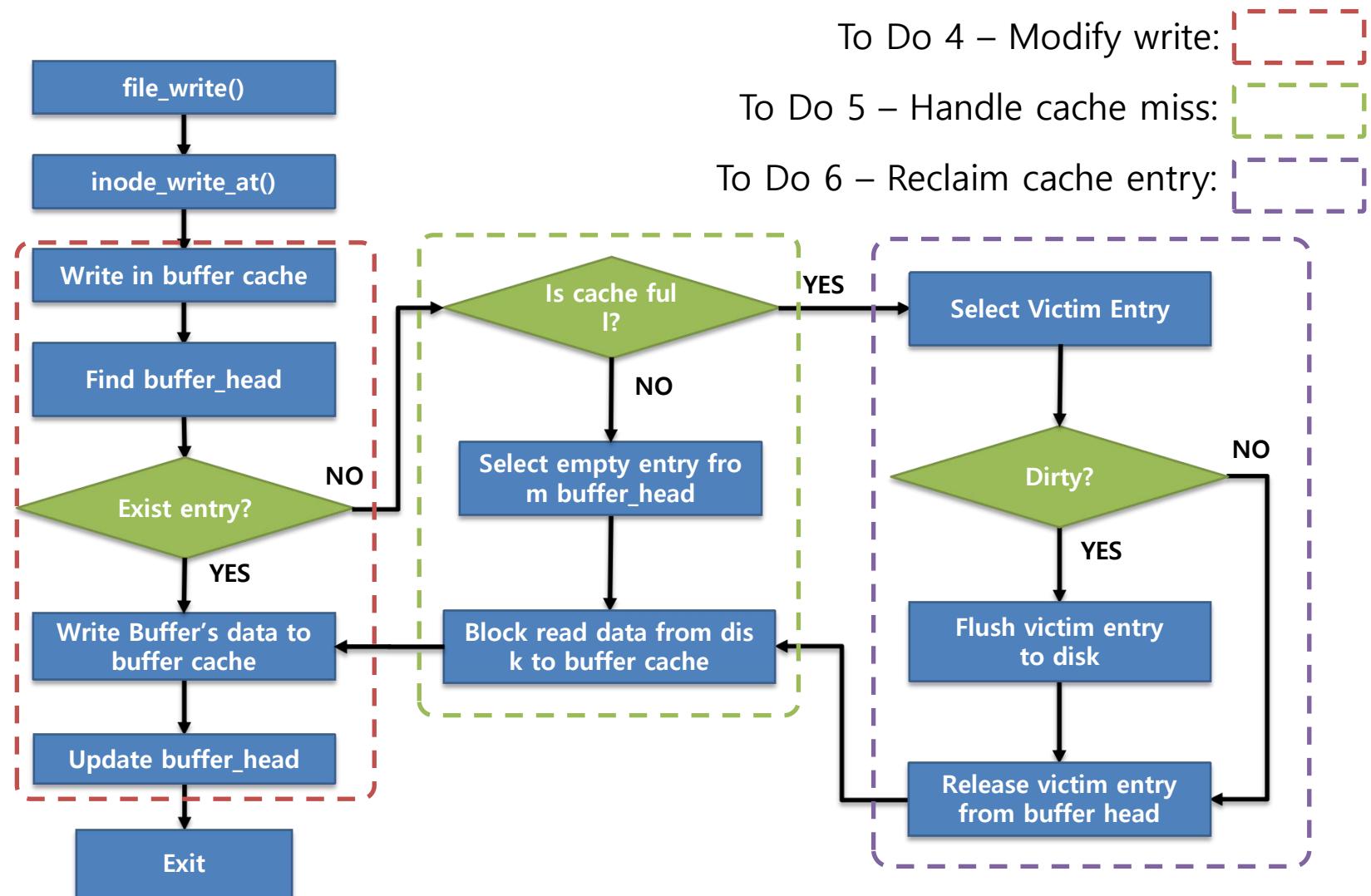
```
off_t inode_read_at (struct inode *inode, void *buffer_,  
                      off_t size, off_t offset) {  
    ...  
    while( size > 0 ) {  
        block_sector_t sector_idx = byte_to_sector (inode, offset);  
        int sector_ofs = offset % BLOCK_SECTOR_SIZE;  
        ...  
        block_read (fs_device, sector_idx, buffer + bytes_read);  
        ...  
    }  
    ...  
}
```

modify

Write in current pintos



Write with Buffer cache



Modify the disk write to write to the buffer cache.

- When writing file, modify it to write data to buffer cache rather than to disk

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);  
        }  
        ...  
    }  
    ...  
}
```

modification

To Do 7: Write dirty buffer cache (sync)

- ▣ Write dirty buffer cache entries,
 - ◆ when the buffer cache entry is evicted.
 - ◆ when filesystem is shut down.

pintos/src/filesys/filesys.c

```
void  
filesys_done (void)  
{  
    /* Add code here */  
    free_map_close ();  
}
```

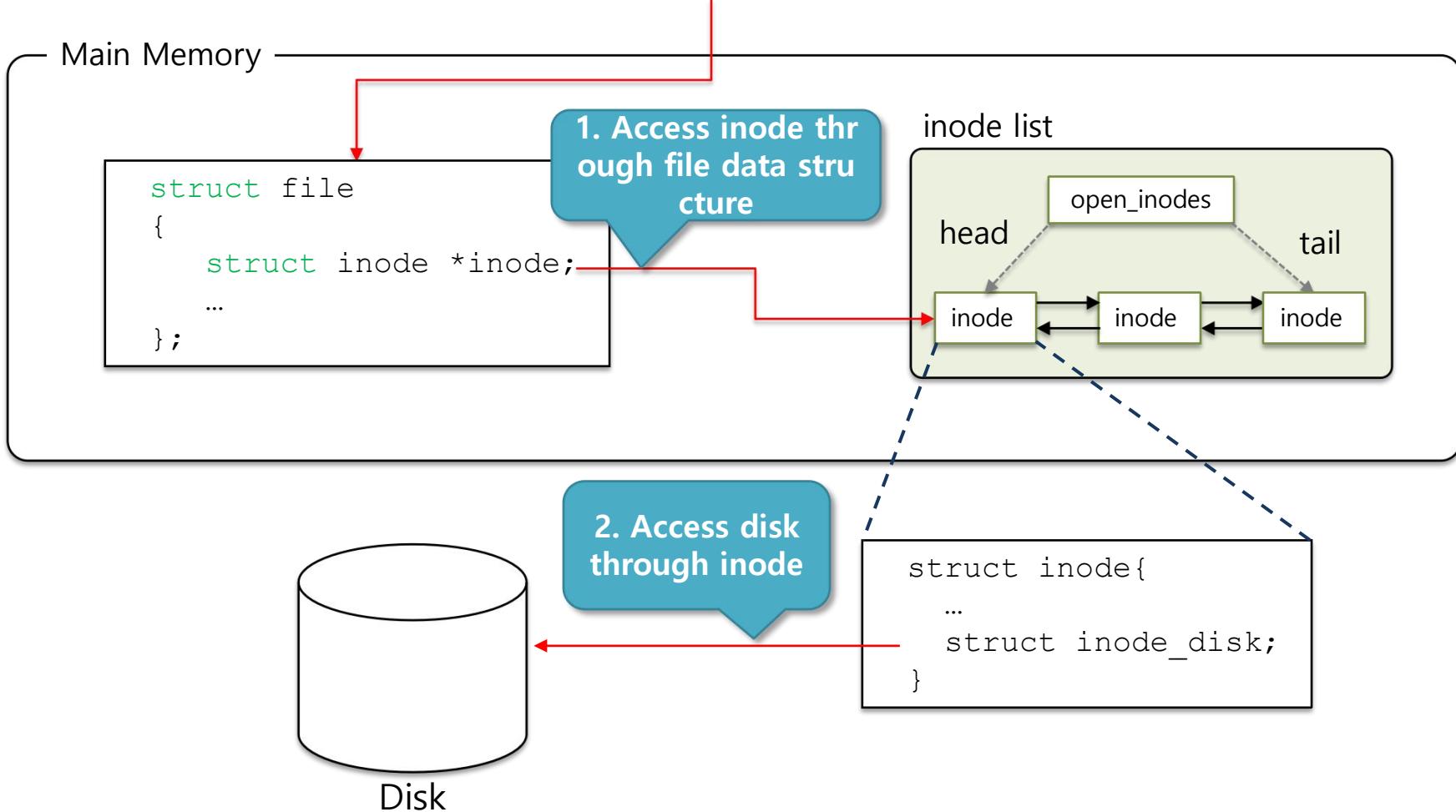
- ◆ Periodically
 - Use timer interrupt.

Read/write in Pintos

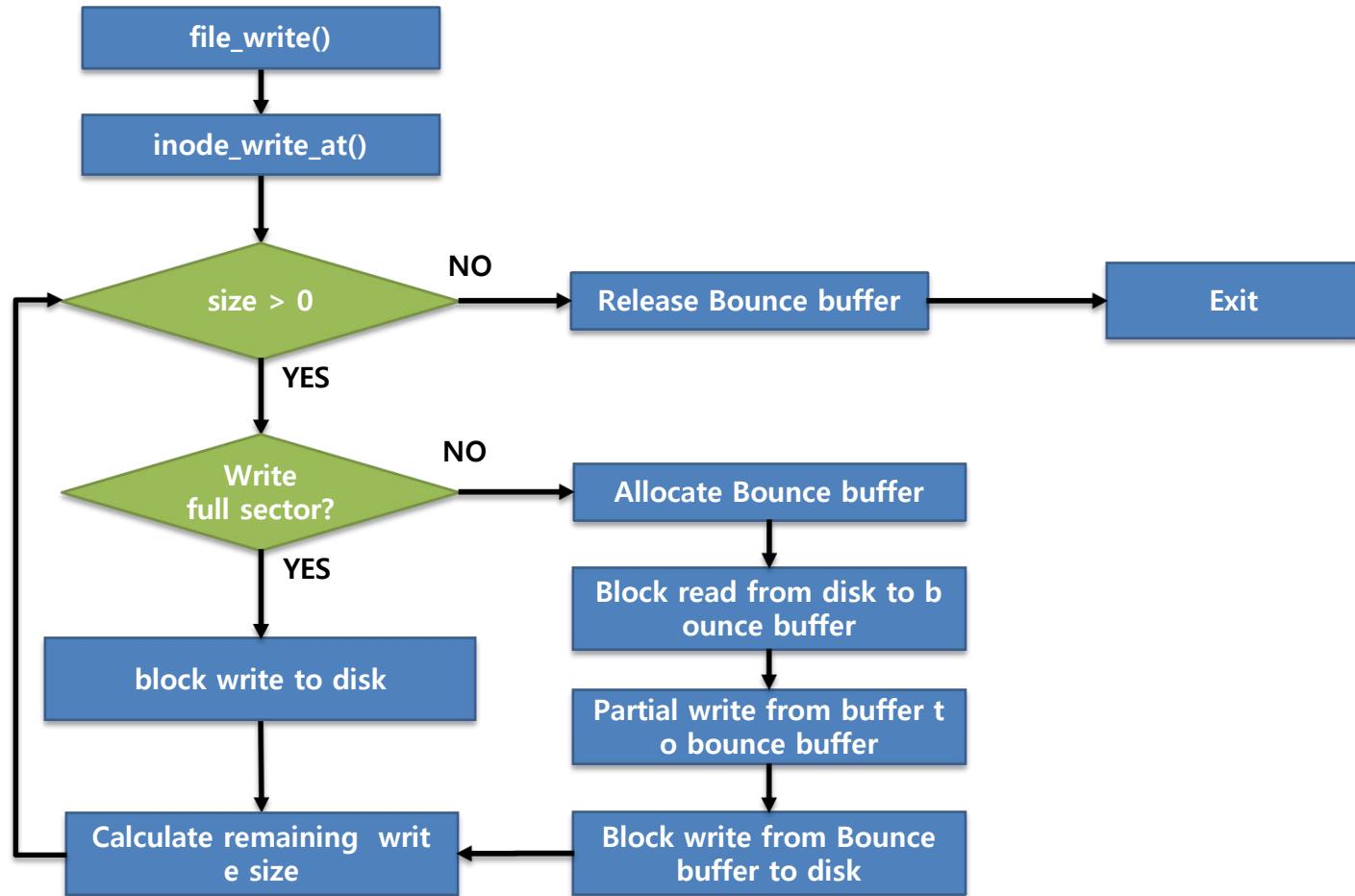
Background - Disk block access sequence in pintos

ex) reading a file

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



Background - Write in current pintos



Background - Write in current pintos (Cont.)

▣ file_write

- ◆ Call inode_write_at() to write data in disk block
 - Change file offset by size recorded

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;
    ...
}
```

Background - Write in current pintos (Cont.)

- inode_write_at

- Record data that buffer points to disk

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;
```

Background - Write in current pintos (Cont.)

- Loop per disk block and write to the disk block: `block_write()`.
 - `byte_to_sector()`: Obtain the disk block number writing data.
 - `sector_ofs`: Offset within the disk block for writing the data.

pintos/src/filesys/inode.c – `inode_write_at()` (Cont.)

```
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);
    }
    ...
}
```

Background - Write in current pintos (Cont.)

- In case of Partial Write, read the target block and save it to bounce buffer.
 - Through `memcpy()`, perform partial write on bounce buffer.
 - Record bounce buffer's data to disk : `block_write()`

pintos/src/filesys/inode.c – inode_write_at()

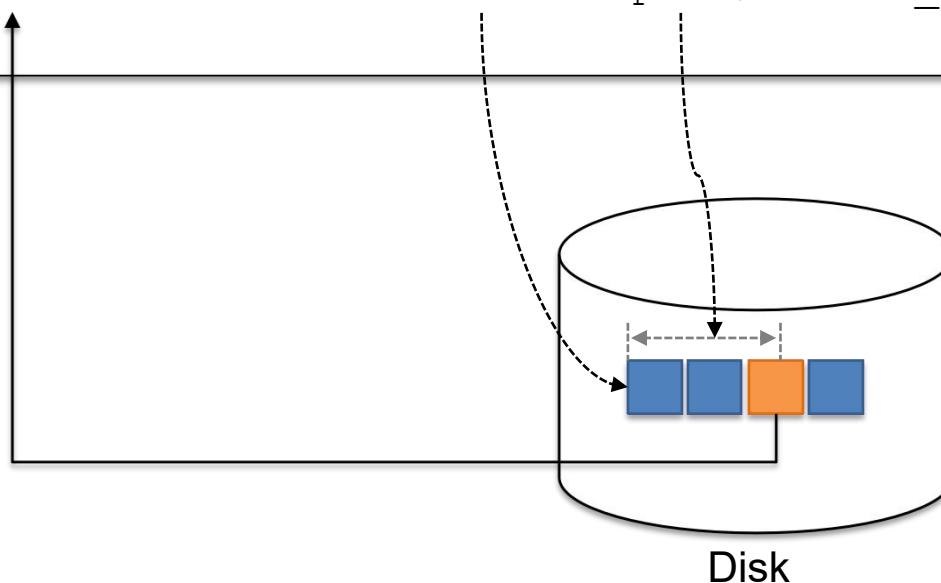
```
...
else {          /* We need a bounce buffer. */
    if (bounce == NULL)
        bounce = malloc (BLOCK_SECTOR_SIZE);
    if (sector_ofs > 0 || chunk_size < sector_left)
        block_read (fs_device, sector_idx, bounce);
    else
        memset (bounce, 0, BLOCK_SECTOR_SIZE);
    memcpy (bounce + sector_ofs, buffer + bytes_written,
            chunk_size);
    block_write (fs_device, sector_idx, bounce);
}
...
```

Background - Get disk block address from file offset

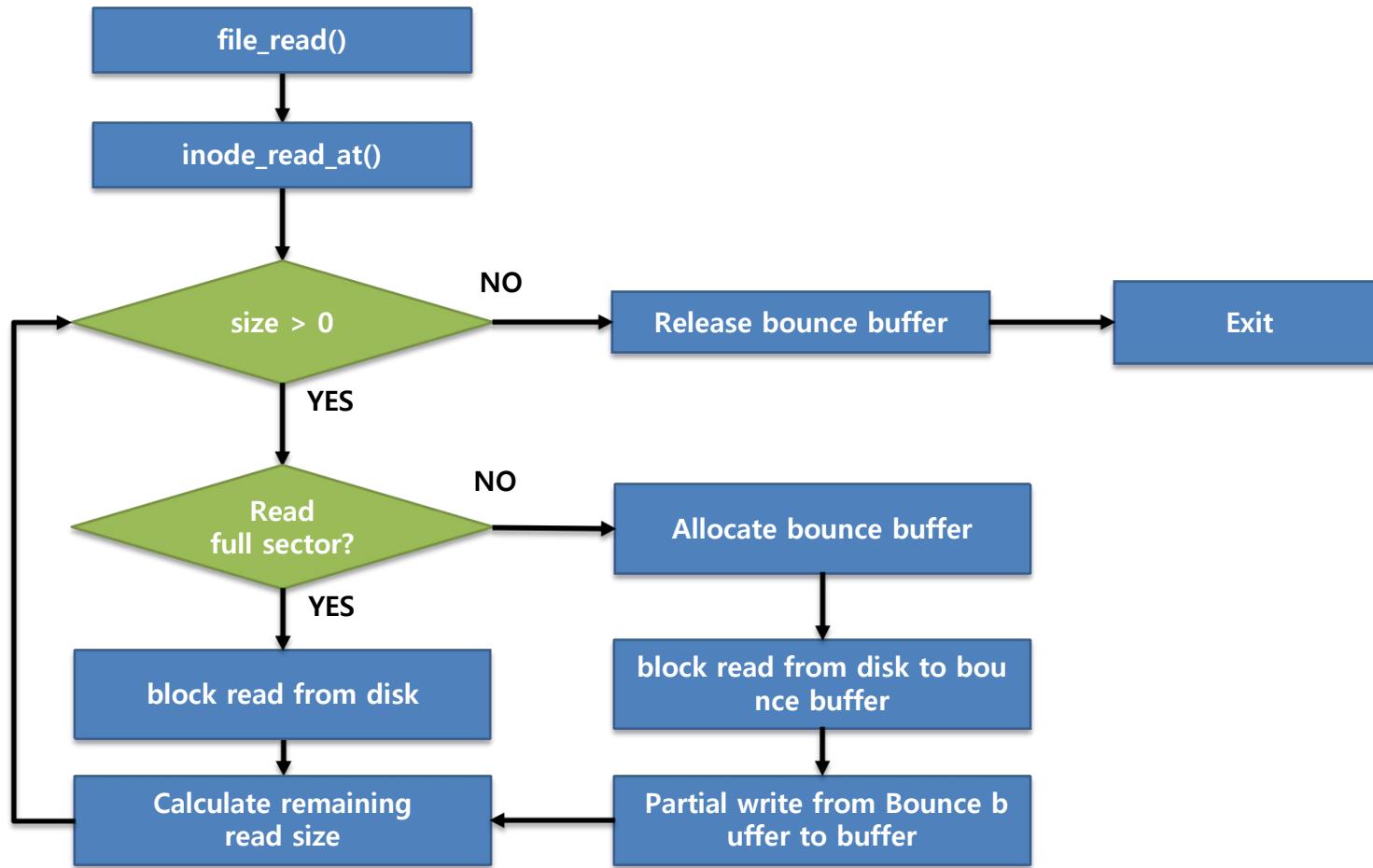
- Return the disk block number by adding offset value from file's starting block.

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;
}
```



Background - read in current pintos



Background - read in current pintos (Cont.)

file_read

- ◆ Call inode_read_at() to read data from disk to buffer
- ◆ Change file offset by read size

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;  
    ...  
}
```

Background - read in current pintos (Cont.)

- inode_read_at

- ◆ Loop per disk block and Read data from disk: block_read()
- ◆ byte_to_sector(): Obtain disk block number to read data
- ◆ sector_ofs : Offset within disk block to read data

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, offse  
t);  
        int sector_ofs = offset % BLOCK_SECTOR_SIZE;  
        /* require replacement with cache read */  
        block_read (fs_device, sector_idx, buffer + bytes_read);  
    }  
    ...  
}
```

Operating System Lab

Part 4: Filesystem

KAIST EE

Youjip Won

Indexed and Extensible File

Extensible File

□ Goal

- ◆ In original pintos, file size is fixed when it is created.
- ◆ In this project, we will modify pintos filesystem to change the file size dynamically. Maximum file size will be 8MB.

□ Files to modify

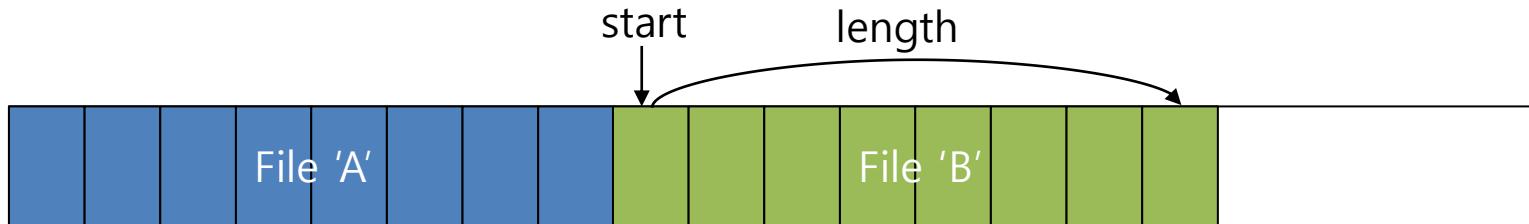
- ◆ `pintos/src/filesys/inode.c`
- ◆ `pintos/src/filesys/inode.h`

How to allocate block when pintos create file

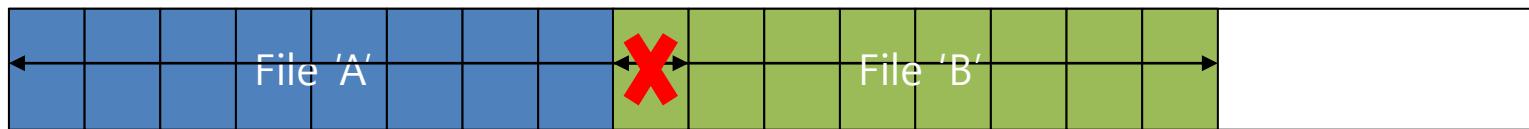
1. Creation of file A (Save start block address and length to inode)



2. Creation of file B (Save start block address and length to inode)



3. The size of file A can't be increased because of conflict with file B

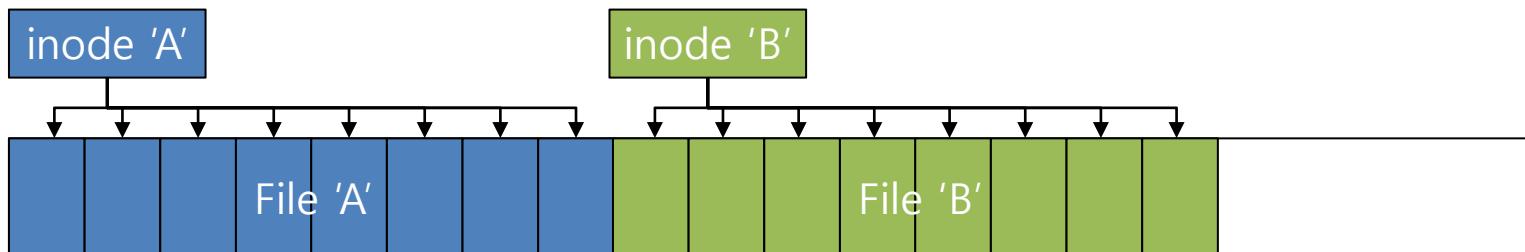


After modification

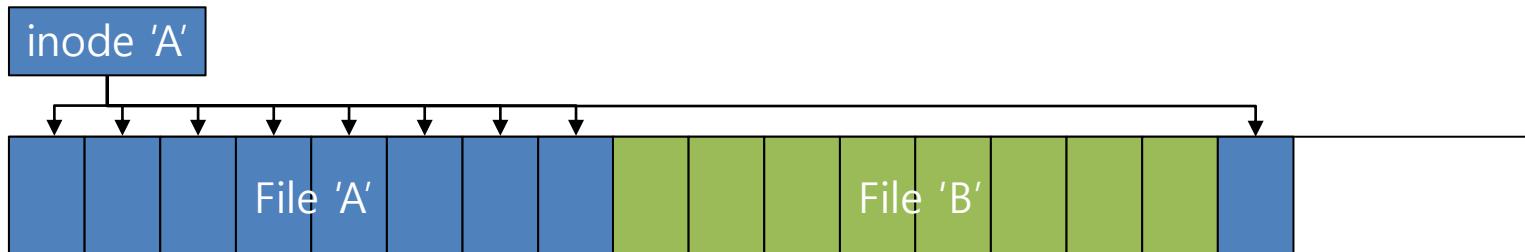
1. Creation of file A (Save block addresses of all each blocks to inode)



2. Creation of file B (Save block addresses of all each blocks to inode)

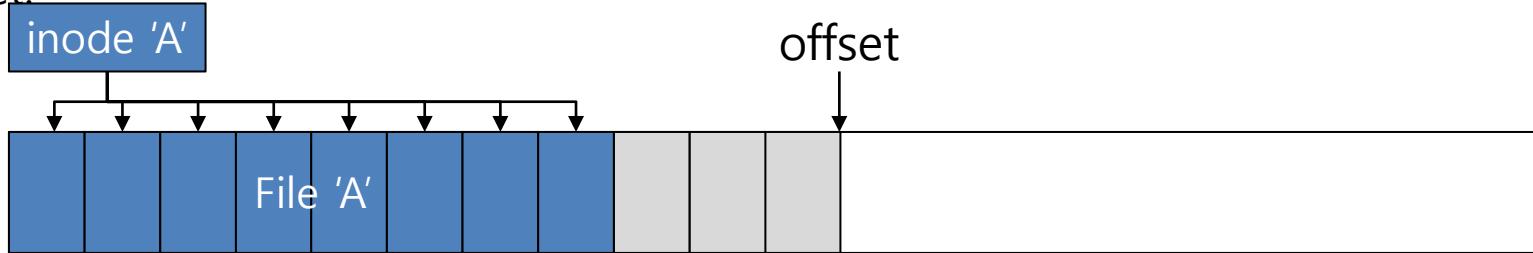


3. Expanding the file A can be done by allocating a new block and by saving its address to inode of file A



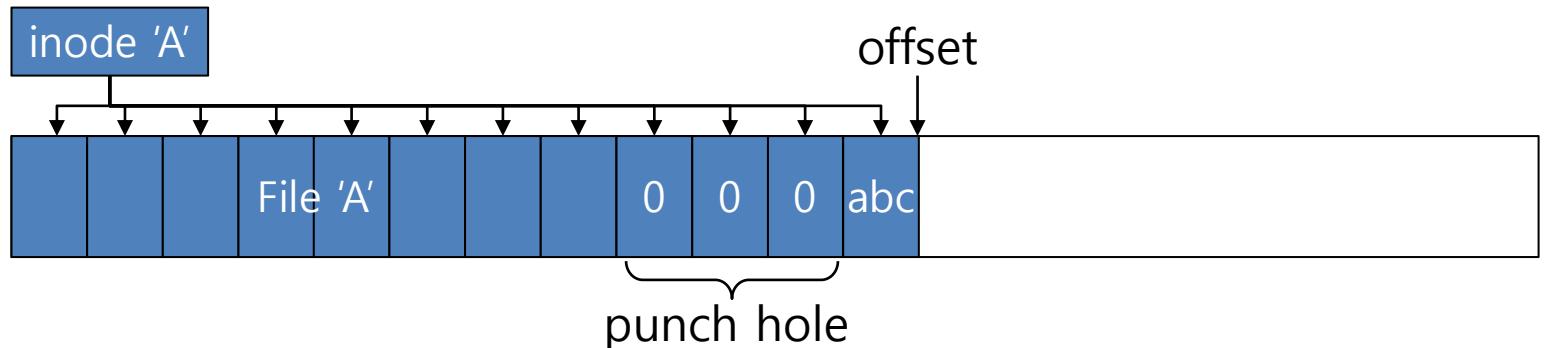
Some details

1. The seek() does not change the file size nor does it allocate the blocks. It just updates offset.



2. When a `write()` is called and offset is larger than the file size, the file size is updated and blocks are allocated (Fill punch hole with zero).

After write ('A', "abc", size):



To Do's

1. Modify on-disk inode structure (`struct inode_disk`).
2. Modify code using on-disk inode.
 - ◆ Changing file offset to block address
 - `static block_sector_t byte_to_sector(const struct inode_disk *inode_disk, off_t pos)`
 - ◆ Creating new inode
 - `bool inode_create(block_sector_t sector, off_t length)`
 - ◆ Deleting an inode
 - `void inode_close(struct inode *inode)`
3. Handle extension of file.
 - ◆ `off_t inode_write_at(struct inode *inode, const void *buffer_, off_t size, off_t offset)`

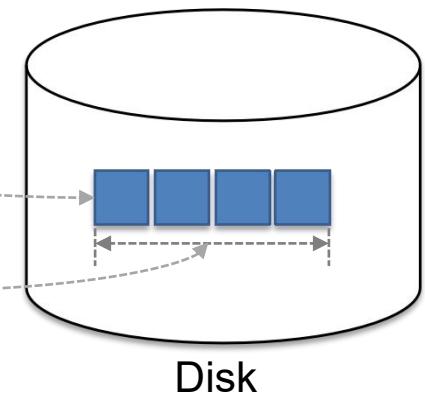
On-disk inode in current pintos

- Fields for pointing blocks in current pintos

- start: Start block address
- length : Size of file (byte)
 - Fixed when the file is created.
 - All blocks should be continuous.

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 */
}
```

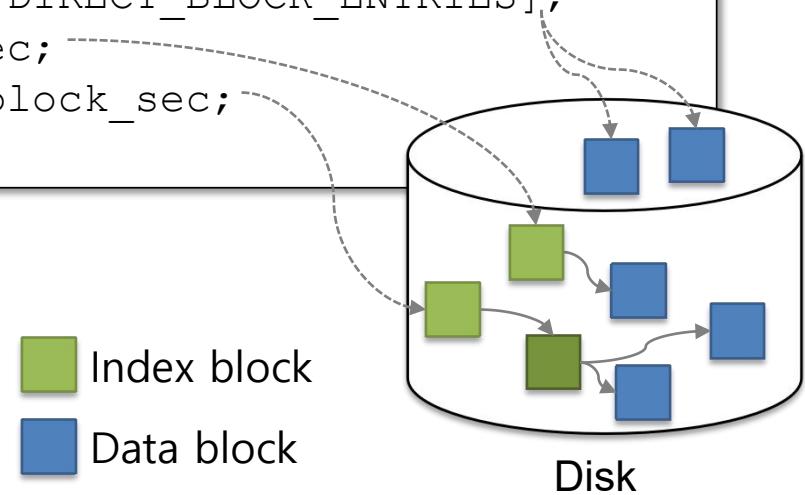


To Do 1 - Modify on-disk inode structure

- Modify `struct inode_disk`.
 - ◆ Add block pointers of direct, indirect, double indirect.
 - ◆ We must maintain the size of "`struct inode_disk`" by adjusting the number of direct block pointers.

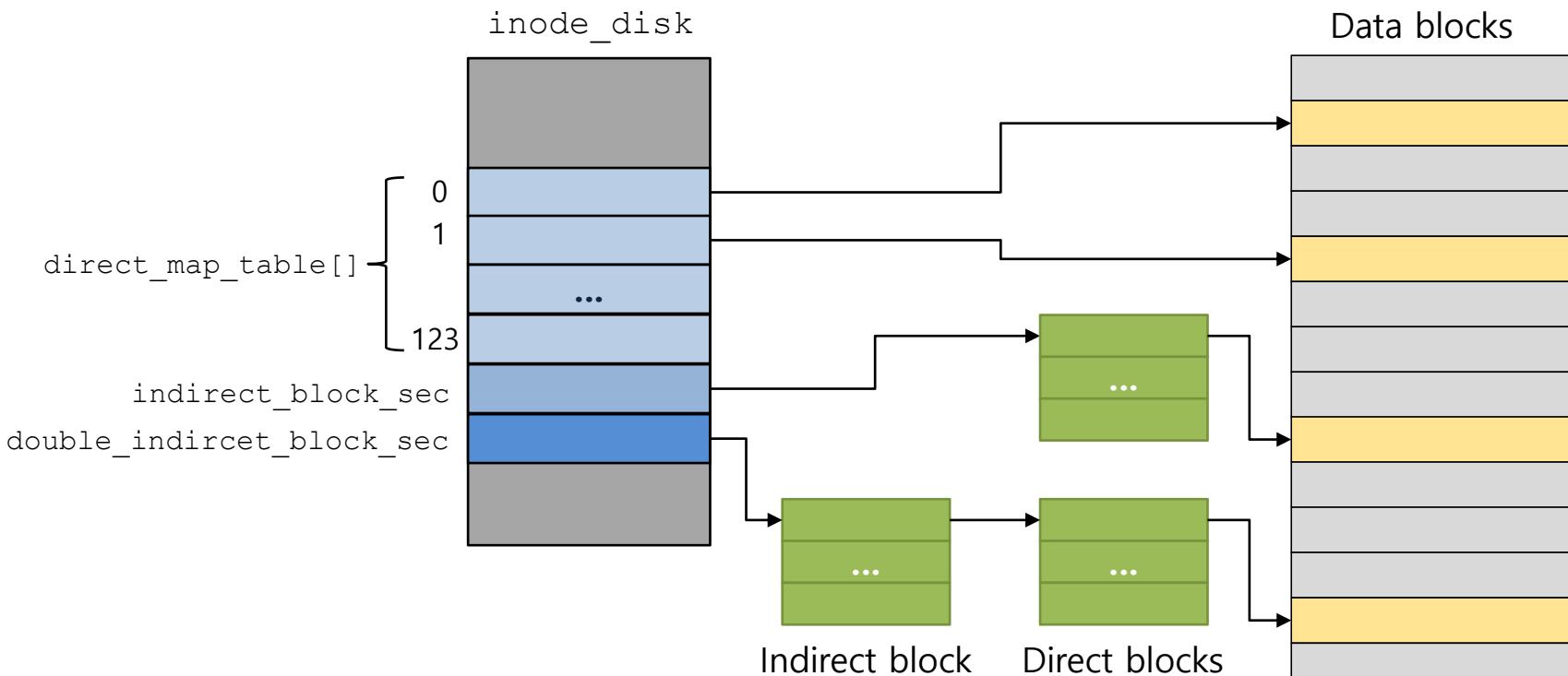
pintos/src/filesys/inode.c

```
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;  
}
```



Index block
Data block

On-disk inode with block indexing



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;
}
```

To Do 2 – compute the sector number from the file offset

- ▣ `static block_sector_t byte_to_sector(const struct inode_disk *inode_disk, off_t pos)`
 - ◆ Return block address associated an offset of inode.
- ▣ In original: just return sum of start and pos.
- ▣ Modify code in red box to use block indexing.

pintos/src/filesys/inode.c

```
static block_sector_t byte_to_sector (const struct inode *inode, off_t pos) {
    ASSERT (inode != NULL);
    if (pos < inode->data.length)
        return inode->data.start + pos / BLOCK_SECTOR_SIZE;
    else
        return -1;
}
```

To Do 2 – Creating an inode

- ▣ `bool inode_create(block_sector_t sector, off_t length)`
 - ◆ Create file which have size of length.
- ▣ In original: Allocate contiguous blocks and save its start address.
- ▣ Modify code to save the block addresses of all blocks allocated.

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 (free_map_allocate(sectors, &disk_inode->start)) {  
            block_write(fs_device, sector, disk_inode);  
            if (sectors > 0) {  
                /* Fill file out by zero */  
            }  
            success = true;  
        }  
        free(disk_inode);  
    }  
    return success;  
}
```

To Do 2 – Deleting an inode

- When it deletes an inode,
 - We have to deallocate all blocks inode have.
 - Add block deallocating code at inode_close.

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) {
            /* Get on-disk inode structure by get_disk_inode() */
            /* Deallocate each blocks by free_inode_sectors() */
            /* Deallocate on-disk inode by free_map_release() */
        }
    }
}
```

Remove original code and add new code

To Do 3 – Handle extension of file

- When the file size changes,
 - Allocate a new block and update data block pointer in inode.
 - Fill the allocated blocks with zero.

pintos/src/filesys/inode.c

```
off_t inode_write_at (struct inode *inode, const void *buffer_,  
                      off_t size, off_t offset) {  
    ...  
    /* Acquire some lock to avoid contention on inode */  
  
    int old_length = disk_inode->length;  
    int write_end = offset + size - 1;  
  
    if(write_end > old_length -1 ) {  
        /* When size of file is updated, Update inode */  
    }  
    /* Release lock */  
  
    while(size > 0) {  
        ...  
    }  
}
```

Subdirectory

Overview of Subdirectories

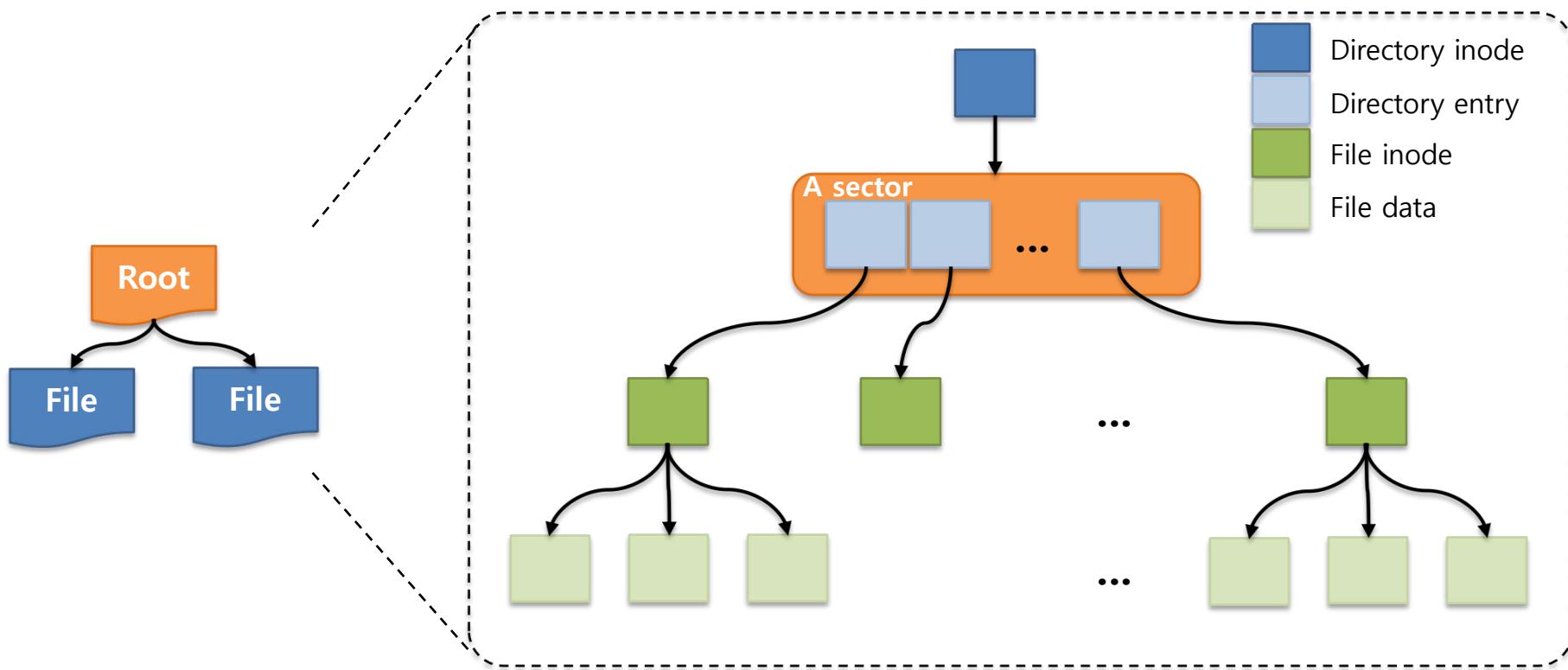
❑ Main Goal

- ◆ Original pintos has only root directory but not the other subdirectories.
- ◆ We will implement subdirectory feature to make filesystem of pintos to have hierarchical structure.

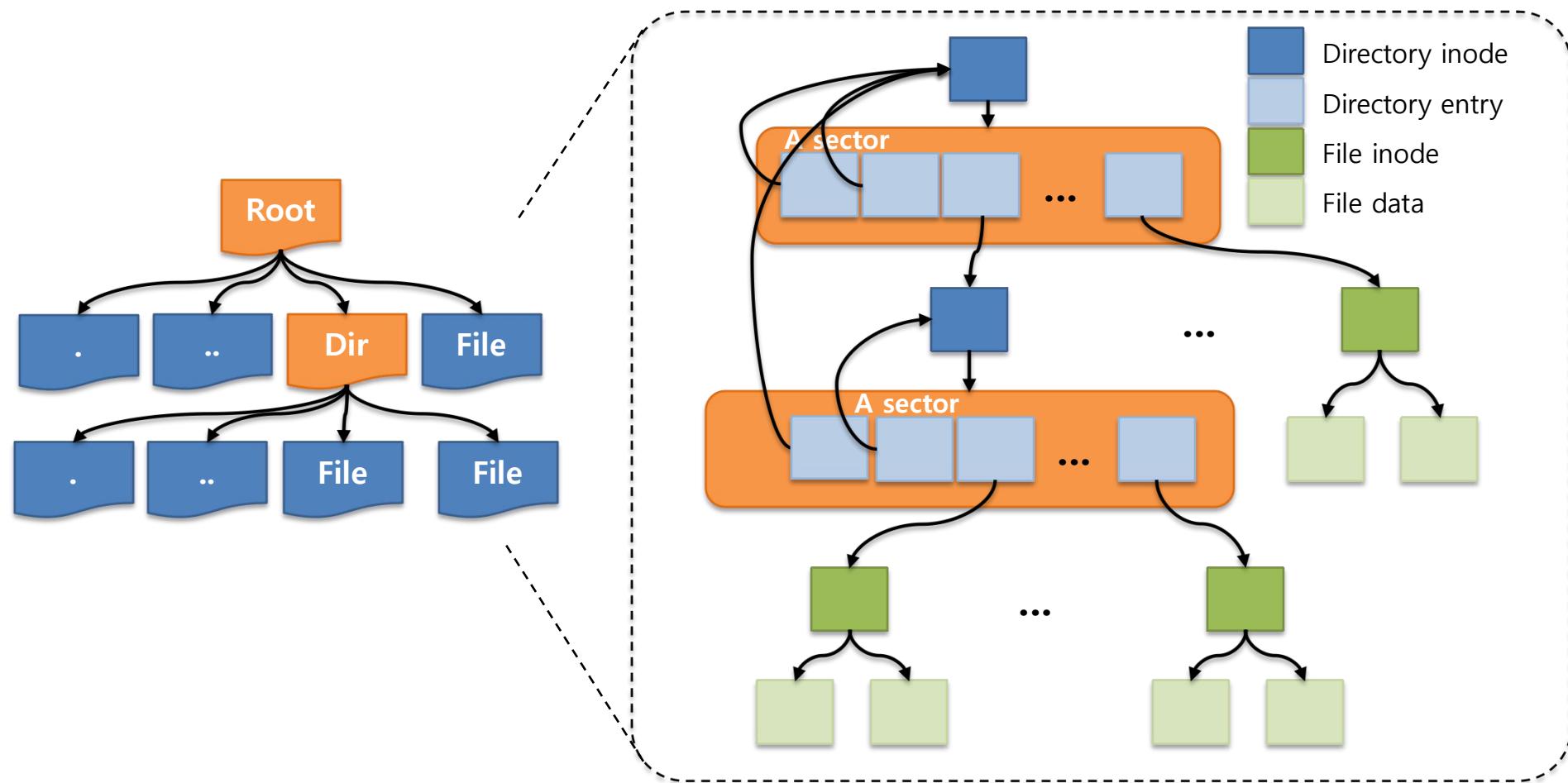
❑ Files to modify

- ◆ pintos/src/filesys/inode.c
- ◆ pintos/src/filesys/filesys.*
- ◆ pintos/src/filesys/file.*
- ◆ pintos/src/filesys/directory.*
- ◆ pintos/src/userprog/syscall.c

Directory structure in original pintos



Hierarchical directory structure



Requirements

- ▣ Implement Hierarchical directory structure.
 - ◆ Make directory entry can point to not only regular file but also directory.
 - ◆ Add directory entry '.' and '..'.
- ▣ Implement "current directory" for a thread.
 - ◆ Distinguish absolute path and relative path (Distinguisher is '/').
- ▣ Modify directory related functions.
 - ◆ `filesys_create()`, `filesys_open()`, `filesys_remove()`
- ▣ Add new directory related system calls.

Now, **there is a concept of path !!!**

To Do's

1. Add flag to indicate whether the inode is for regular file or for directory.
2. Define current directory pointer in `struct thread`.
3. Modify code for directory manipulation.
 1. Creating new file.
 2. Opening a file.
 3. Removing a file.
4. Add system calls for directory manipulation.
5. Add special directory entries.: ".", "..".

To Do 1: Modify On-disk inode

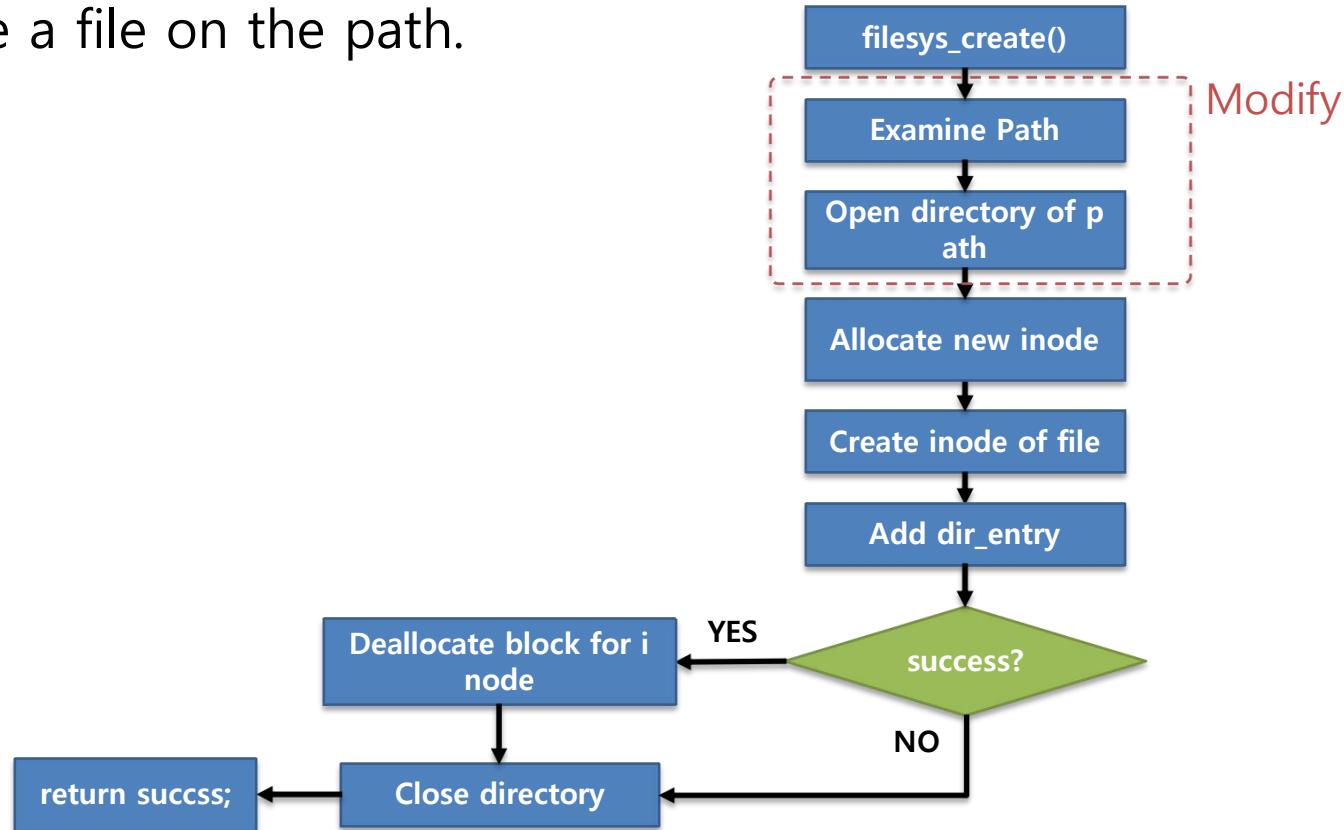
- ▣ Add a flag indicating if it is file or directory.
 - ◆ `struct inode_disk`
 - file: `pintos/src/filesys/directory.c`
 - Regular file (=0), directory (=1)
- ▣ When a file is created, we should set this flag properly.

To Do 2: Define current directory in struct thread.

- ▣ Add current directory pointer in thread.
 - ◆ `struct` `thread`
 - File: `pintos/src/threads/thread.h`
 - Add pointer to `struct` `dir` representing current directory.
- ▣ When a thread is created, the child thread inherits parent's current directory.

To Do 3: Modify algorithm of file creation

- ❑ Distinguish if it is absolute path or relative path.
- ❑ Find directory associated with path.
- ❑ Create a file on the path.



To Do 3: Modify algorithm of File creation (Cont.)

- ▣ `bool filesys_create (const char *name, off_t initial_size)`
 - ◆ Original: Always create a file in the root directory.
 - ◆ After modification: Parse the path and create that file on that directory.
 - Distinguish absolute path and relative path and parse.
 - ◆ Add the code to set `is_dir` flag to 0 if it is regular file.
 - ◆ Add new directory entry to directory of path.

To Do 3: Opening a file

- ▣ `struct file *filesys_open(const char *name)`
 - ◆ Original: Always find the file on the root directory.
 - ◆ After modification: Parse path, find the file on that directory, and open it.
 - Distinguish absolute path and relative path and parse.
 - When the path is absolute: Find from the root directory.
 - When the path is relative: Find from the current directory.

To Do 5: Removing a file

- ▣ `bool filesys_remove(const char *name)`
 - ◆ Original: Always remove file from root directory.
 - ◆ After modification: Remove file from directory specified by path.
 - Distinguish absolute path and relative path and parse.
 - ◆ Check if in-memory of target file is for directory or regular file.
 - If it is directory, check it have files.
 - Remove only when directory is empty.
 - If it is file, just remove it.

To Do 6: Add system calls about directory

- ▣ `bool chdir(const char *dir)`
 - ◆ Change the current working directory of the process to `dir`.
 - ◆ Return true if successful, false on failure.
- ▣ `bool mkdir(const char *dir)`
 - ◆ Creates the directory named `dir`.
 - ◆ Returns true if successful, false on failure.
- ▣ `bool readdir(int fd, char *name)`
 - ◆ Reads a directory entry from file descriptor `fd`, which must represent a directory.
 - ◆ If successful, stores file name in `name` and return true.
 - `..` and `...` should not be returned by `readdir`.
- ▣ `bool isdir(int fd)`
 - ◆ Returns true if `fd` represents a directory, false if it represents an ordinary file.
- ▣ `int inumber(int fd)`
 - ◆ Returns the inode number of the inode associated with `fd`.

To Do 7: Add special entries

- ▣ Special directory entries.
 - ◆ `.`: it represents itself.
 - ◆ `..`: it represents its parent directory.
- ▣ When a directory is created, special entries should be added.
 - ◆ even root directory created during format.
- ▣ If the user tries to remove them, system call should return fails.

Operating System Lab

Part 4: Filesystem

KAIST EE

Youjip Won

Filesystem: Background

Block Device

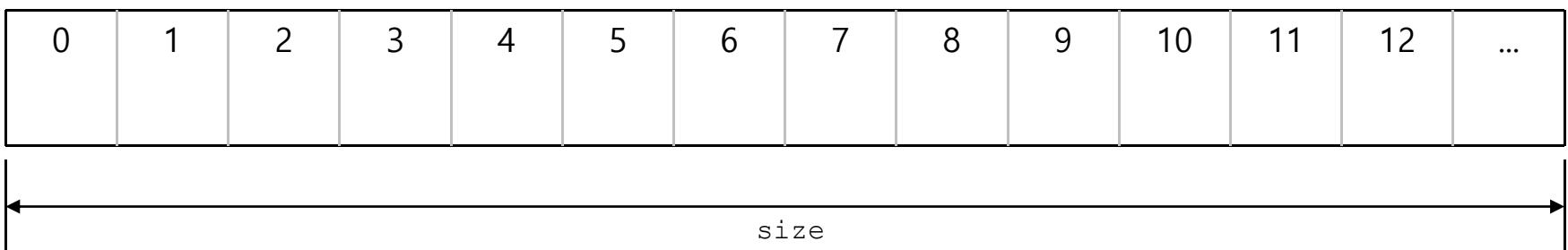
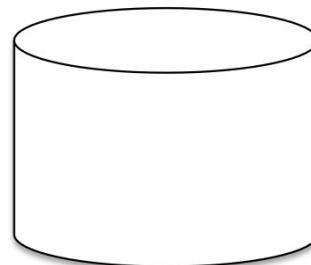


Logical block address (sector number)

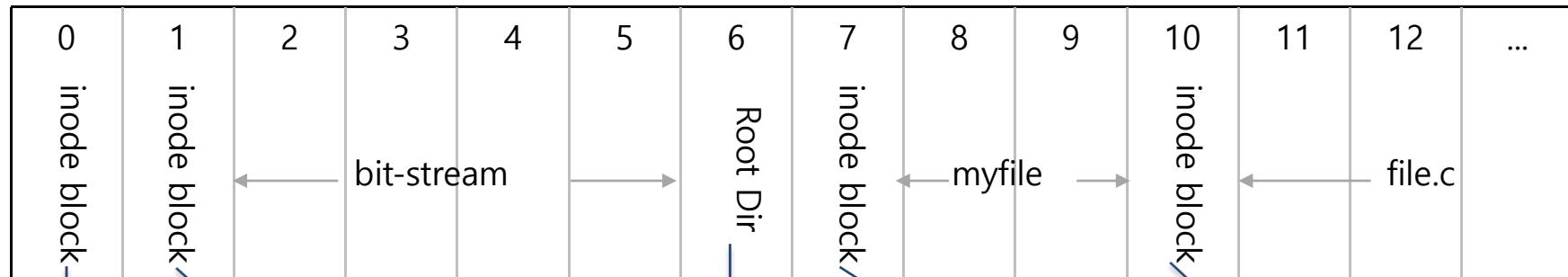
Abstraction for Block Device

```
struct block {
    struct list_elem list_elem;          /* Element in all_blocks. */
    char name[16];                      /* Block device name. */
    enum block_type type;               /* Type of block device. */
    block_sector_t size;                /* Size in sectors. */
    const struct block_operations *ops; /* Driver operations. */
    void *aux;                          /* Extra data owned by driver. */
    unsigned long long read_cnt;        /* Number of sectors read. */
    unsigned long long write_cnt;       /* Number of sectors written. */
};
```

Block device



Filesystem layout



Inode for bitmap

```
start : 2  
length : 2048  
magic  
unused[125]
```

Inode for "/"

```
start : 6  
length : 320  
magic  
unused[125]
```

entries in "/"

file name	inode No.
myfile	7
file.c	10
...	

myfile inode

```
start : 8  
length : 1024  
magic  
unused[125]
```

file.c inode

```
start : 11  
length : 2048  
magic  
unused[125]
```

Bitmap size:

2048 bytes (4 sectors) = 2048×8 bits = 16384 bits

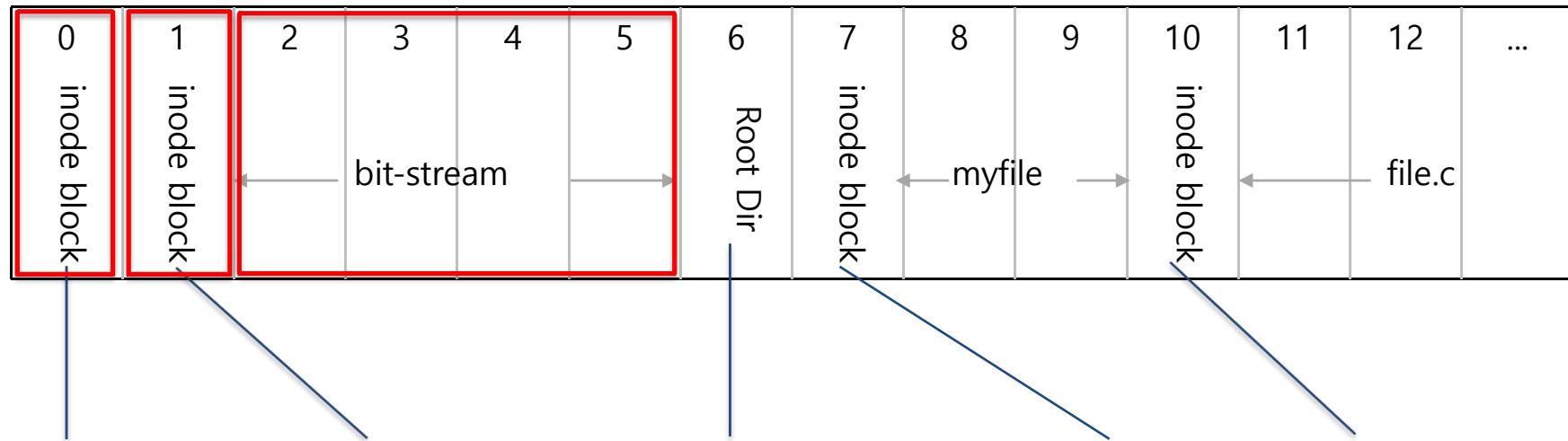
Bitmap for 16384 sectors (8 Mbyte) = 16384×512 Byte

Bitmap for 8 Mbyte filesystem partition

Formatting a filesystem in Pintos

- ❑ Create a filesystem layout on disk.
 - ◆ Create and initialize bitmap.
 - ◆ Create inode of bitmap and write its data on the disk.
 - ◆ Create inode of Root directory.

Formatting a filesystem in Pintos



Inode for bitmap

```
start : 2  
length : 204  
8  
magic  
unused[125]
```

Inode for "/"

```
start : 6  
length : 320  
magic  
unused[125]
```

entries in "/"

file name	inode No.
myfile	7
file.c	10
...	

myfile inode

```
start : 8  
length : 1024  
8  
magic  
unused[125]
```

file.c inode

```
start : 11  
length : 204  
8  
magic  
unused[125]
```

Bitmap size:

2048 bytes (4 sectors) = 2048×8 bits = 16384 bits

Bitmap for 16384 sectors = 16384×512 Byte

Bitmap for 8 Mbyte filesystem partition

Formatting a filesystem in Pintos

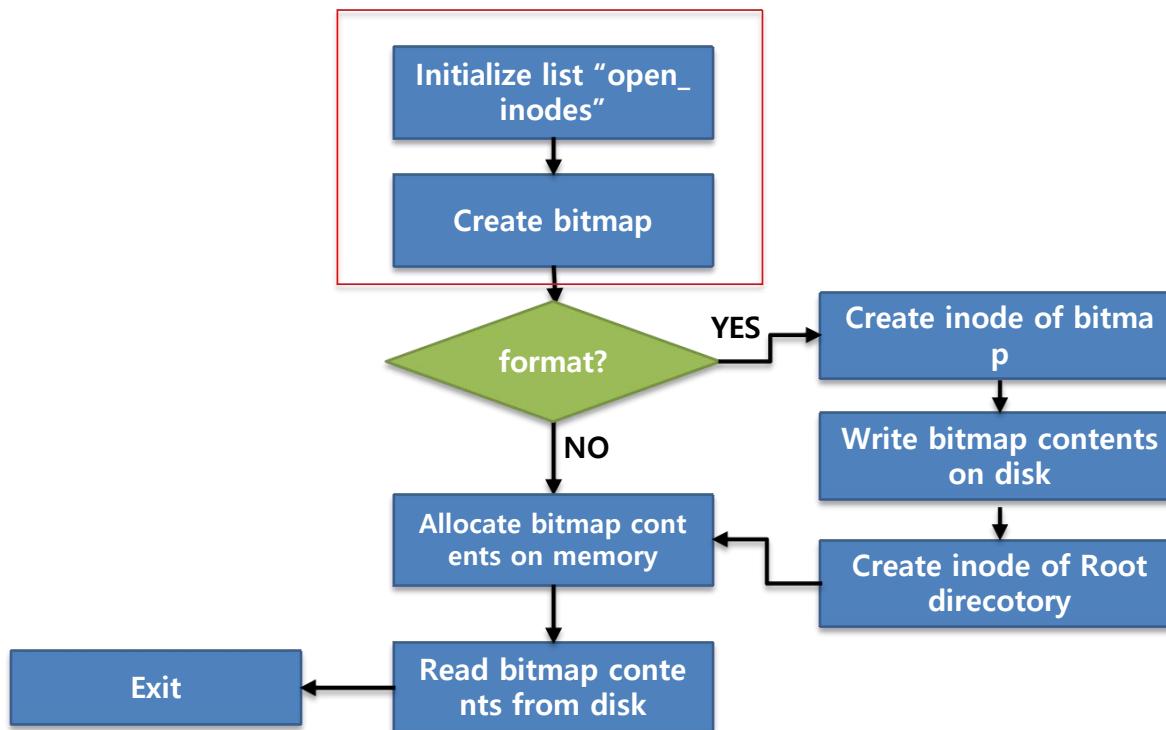
pintos/src/filesys/filesys.c

```
void filesys_init (bool format) {
    fs_device = block_get_role (BLOCK_FILESYS);

    inode_init ();
    free_map_init ();
    if (format)
        do_format ();
    free_map_open ();
}
```

- Create and write bitmap of Filesystem.
 - ◆ inode_init() : Create and initialize the data structure for open files.
 - ◆ free_map_init() : Create and initialize bitmap.
 - ◆ do_format()
 - Create and write the inode of bitmap file.
 - Create the inode of Root directory.

Initializing the filesystem



Initialize the list of open inodes

- Initialize the list of in-memory inodes.
 - open_inodes : global list of in-memory inode (Doubly linked list)
 - What is the data structure that represents the open files in xv6 ?**

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;  
}
```

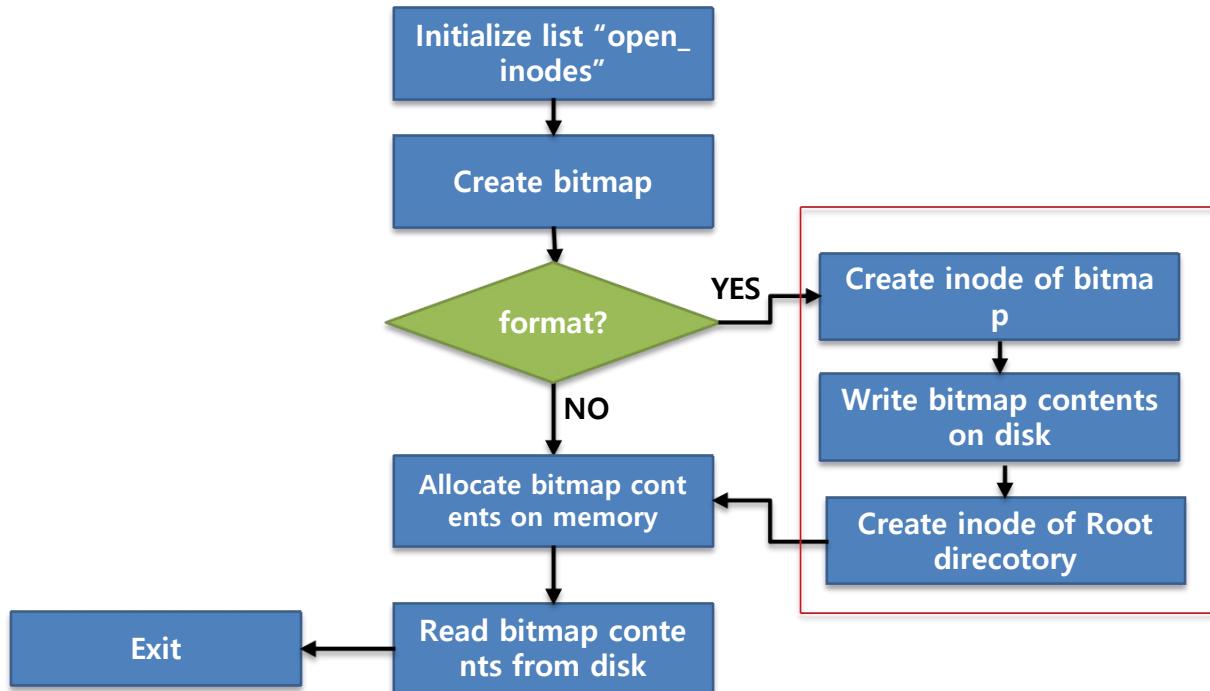
Create and initialize the bitmap

- ❑ Create and initialize the bitmap in-memory.
 - ◆ `bitmap_create()` : Create bitmap and fill it out 0.
 - ◆ `bitmap_mark()`: Flip bit indexed by second parameter.

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);
}
```

Initializing the filesystem



Create the inodes for bitmap and root directory

- ▣ Store the bitmap on the disk and Initialize root directory.
 - ◆ `free_map_create()`: Create inode of bitmap.
 - Write contents of bitmap on disk.
 - ◆ `dir_create()` : Create inode of root directory.
 - The maximum number of files in Root directory is 16.

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 ();  
}
```

Create and save the bitmap file

- Create and write inode of bitmap on disk.

- ◆ `inode_create()` : Create inode of bitmap at block 0.
- ◆ `bitmap_write()` : Write contents of bitmap on disk.

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))
}
```

Create a root directory.

- ❑ Create inode of root directory at block 1.
 - ◆ sector : block number which has the inode for root directory.
 - ◆ entry_cnt : The maximum number of entries in root directory.

pintos/src/filesys/directory.c

```
bool dir_create (block_sector_t sector, size_t entry_cnt)
{
    return inode_create (sector, entry_cnt * sizeof (struct di
r_entry));
}
```

Creating a root directory

- Allocate data blocks for root directory and save its start address at inode.

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(): Translate length of bytes to length of blocks.
- free_map_allocate(): Allocate contiguous blocks and save its start address at second parameter.
- block_write(): write disk_inode to disk.

Synchronizing the bitmap to the disk

- Close inode of bitmap: Deallocate and remove in-memory inode from `open_inodes` list.

pintos/src/filesys/free-map.c

```
void free_map_close (void)
{
    file_close (free_map_file);
}
```

- ◆ `file_close()`
 - Remove inode from `open_inodes` list and deallocate it.
 - Deallocate file structure.

Load the bitmap to memory

- Read bitmap contents on disk.

pintos/src/filesys/free-map.c

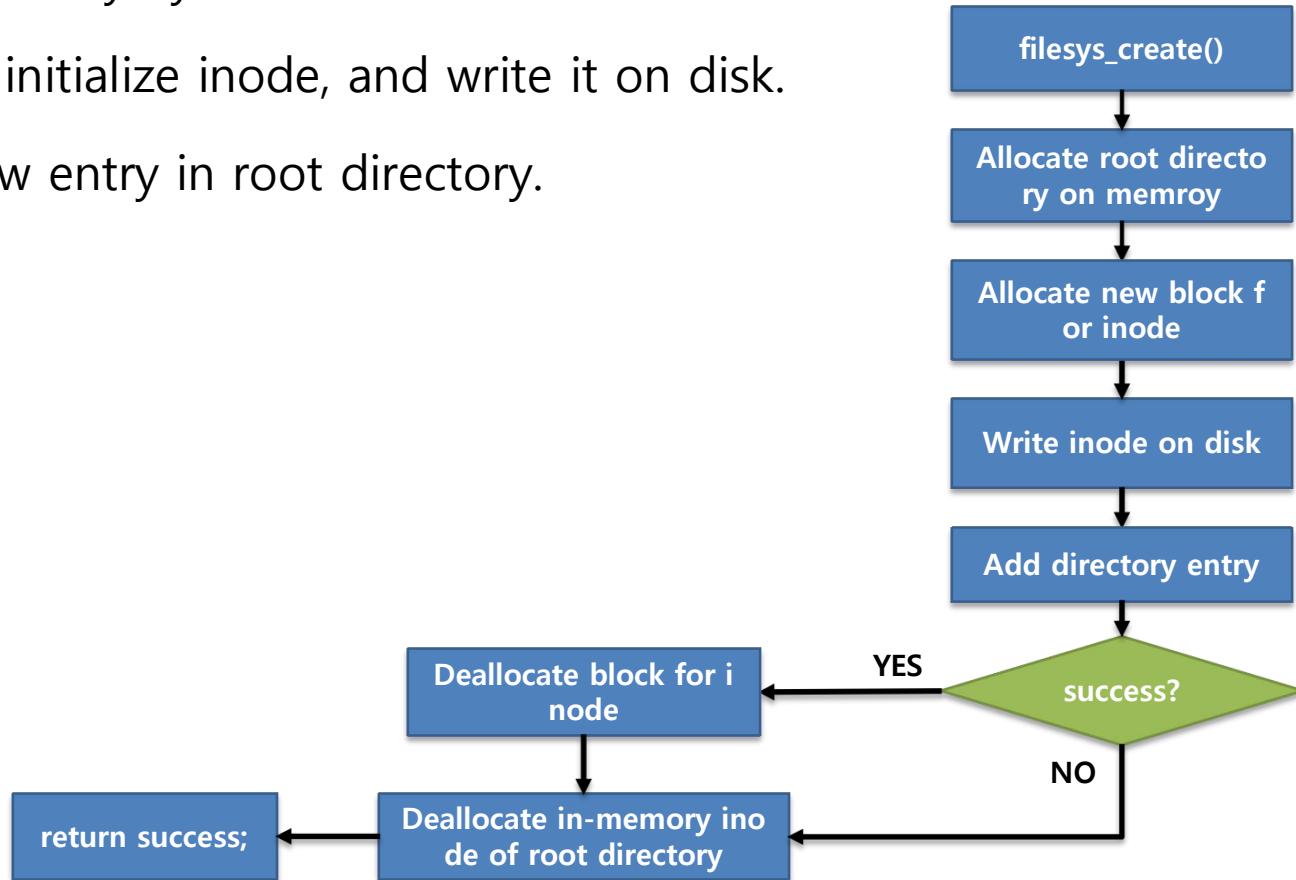
```
void free_map_open (void) {
    free_map_file = file_open (inode_open (FREE_MAP_SECTOR));
    if (free_map_file == NULL)
        PANIC ("can't open free map");
    if (!bitmap_read (free_map, free_map_file))
        PANIC ("can't read free map");
}
```

- file_open() : Allocate and initialize file structure.
- bitmap_read() : Read bitmap contents on disk.

File create

❑ `filesys_create()`

- ◆ It is called by System call ‘`create()`’ .
- ◆ Create, initialize inode, and write it on disk.
- ◆ Add new entry in root directory.



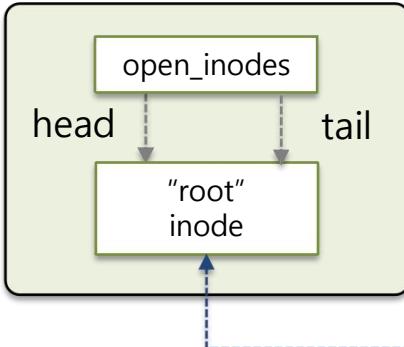
File create

ex) 'testfile' create

```
bool filesys_create ("testfile")
```

Main Memory

inode list



3. Read entries on root directory

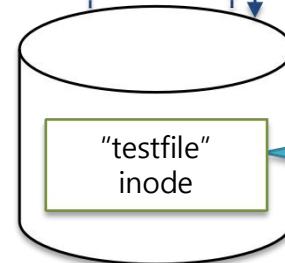
root directory entries

inode = 7	...
testfile	...
in_use = 1	...

4. Add new directory entry for 'testfile'

1. Read inode of root directory

5. Write entries of root directory



2. Write inode of 'testfile' on disk

File create in Pintos

❑ filesys_create

- ◆ Create and initialize inode, and add new directory entry to root directory.

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()` : Allocate dir structure for root directory on memory.
- ◆ `free_map_allocate()` : Allocate 1 sector from the free map and save the start sector at `inode_sector`.

File create in Pintos

pintos/src/filesys/filesys.c – `filesys_create()` (Cont.)

```
    && inode_create (inode_sector, initial_size)  
    && dir_add (dir, name, inode_sector));  
  
...  
  
return success;  
}
```

- ◆ `inode_create()` :
 - initialize an inode with `initial_size` byte
 - write it on disk.
- ◆ `dir_add()`: Add new directory entry to directory.

Opening a file

- inode_open: read on-disk inode at sector and returns its pointer.

pintos/src/filesys/inode.c

```
struct inode * inode_open (block_sector_t sector) {  
    ...  
    /* Firstly, find in-memory inode in open_inodes */  
    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;  
        }  
    }  
    ...
```

Opening a file

pintos/src/filesys/inode.c – inode_open() (Cont.)

```
...
/* Allocate in-memory inode */
inode = malloc (sizeof *inode);
...
/* Initialize in-memory inode */
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;
}
```

Opening a directory

- Allocate `dir` structure and set its field.: `inode` and `pos` (offset).

`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;
    }
    ...
}
```

Allocating the free block

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;
}
```

- ◆ Find cnt consecutive free blocks, scanning free-map.
- ◆ cnt : the number of block to allocate
- ◆ sectorp : start address of blocks allocated
- ◆ bitmap_scan_and_flip() : Find contiguous false bitmap entries and set them true.

Creating a directory

- Add name file to dir.
- Inode of the file is at sector `inode_sector`.

`pintos/src/filesys/directory.c`

```
bool dir_add (struct dir *dir, const char *name, block_secto
r_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;

    /* Find unused directory entry in direcctory */
    for (ofs = 0; inode_read_at (dir->inode, &e, sizeof e, o
fs) == sizeof e; ofs += sizeof e)
        if (!e.in_use)
            break;
    ...
}
```

Background: File create in pintos (Cont.)

pintos/src/filesys/directory.c – dir_add() (Cont.)

```
...
/* 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;
}
```

Directory lookup

▫ lookup

- Check if file name exist in directory or not.
- Return address of `dir_entry` structure by parameter.

pintos/src/filesys/directory.c

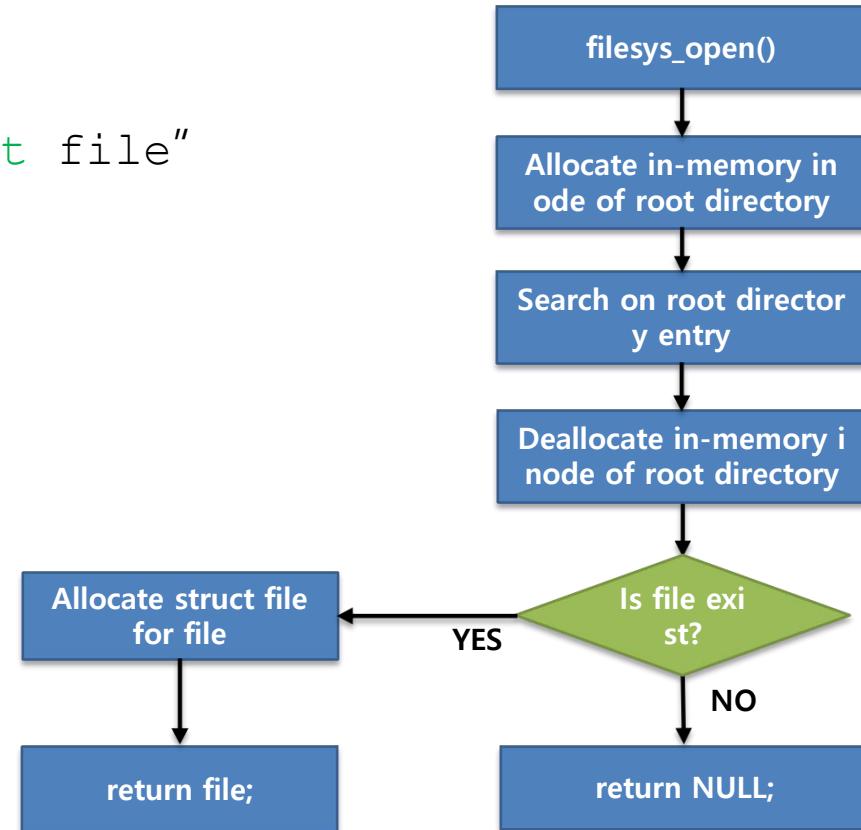
```
static bool lookup (const struct dir *dir, const char *name,
                   struct dir_entry *ep, off_t *ofsp) {

    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;
}
```

Open a file

▣ `struct file *filesystem_open(const char *name)`

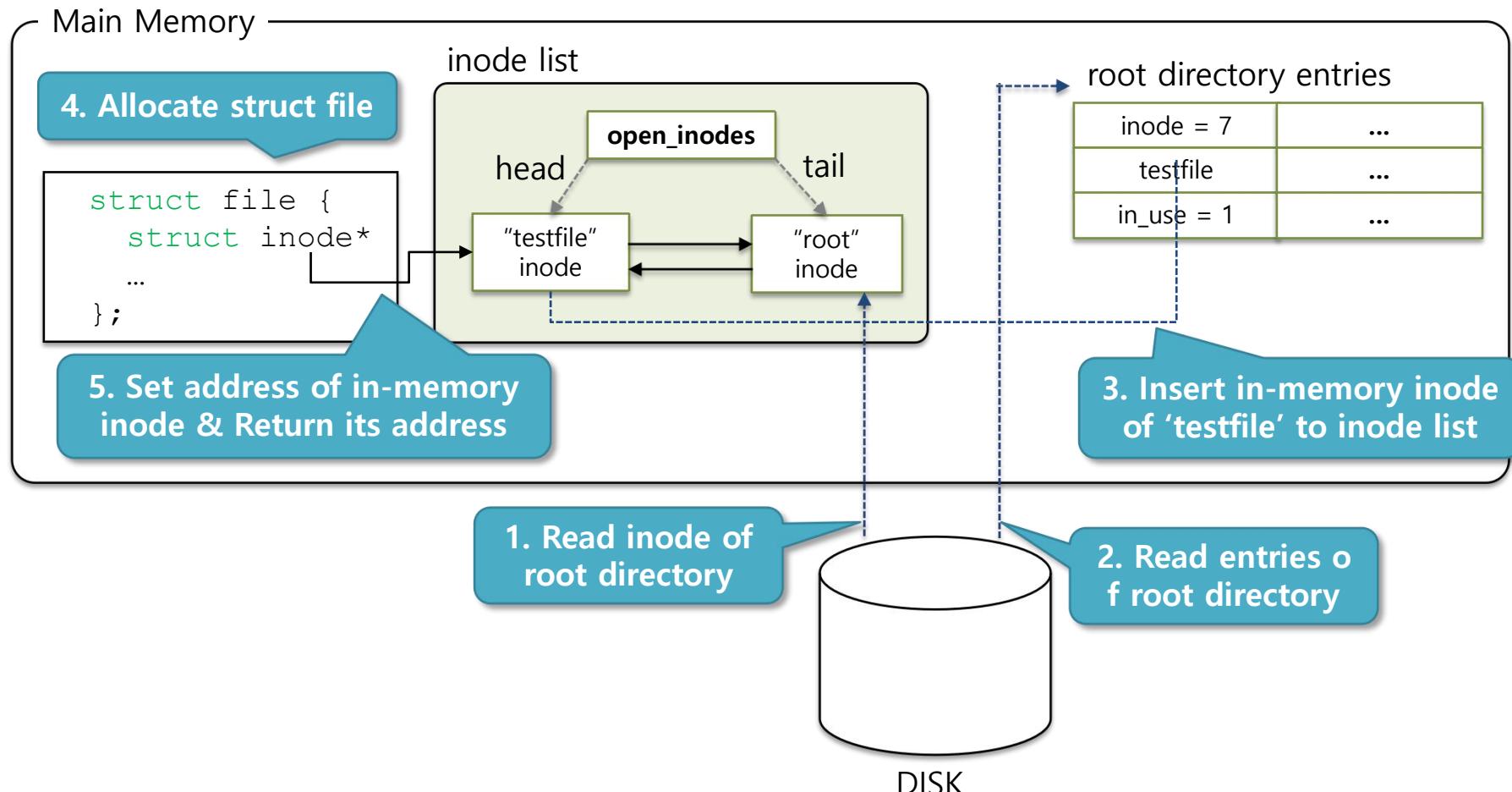
- ◆ It is called by System call 'open()'.
- ◆ Add inode to inode list.
- ◆ Allocate and Initialize "`struct file`" and return its address.



Open a file

ex) 'testfile' open

```
struct file * filesys_open ("testfile")
```



Open a file

- ❑ `filesys_open`

- ◆ Allocate and initialize `struct file`, and return its address.

`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()` : Add in-memory inode of root directory to `open_inodes` list.
 - ◆ `dir_lookup()` : Find directory entry that have name in directory, and open it. (Allocate in-memory inode and add it to `open_inodes`)
 - ◆ `file_open()` : Allocate and initialize `struct file` on memory.

Open a file

▫ dir_lookup

- ◆ Find file in directory, open it, and return success or not.

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 () : Read directory from disk, find file, save its directory entry at 3^r
d parameter.

Open a file

▣ file_open

- ◆ Allocate and initialize struct file on memory, and return its address.

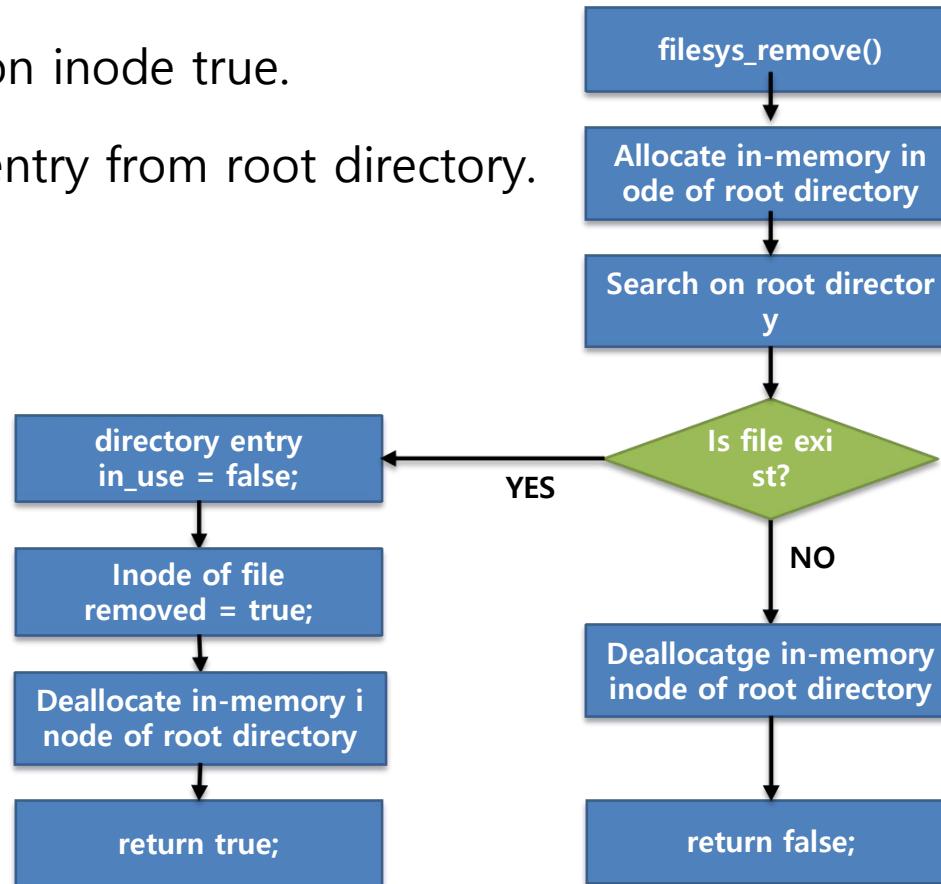
pintos/src/filesys/file.c

```
struct file * file_open (struct inode *inode) {  
    /* Allocate struct file */  
    struct file *file = calloc (1, sizeof *file);  
    /* Initialize struct file */  
    if (inode != NULL && file != NULL) {  
        file->inode = inode;  
        file->pos = 0;  
        file->deny_write = false;  
        return file;  
    ...  
}
```

Remove a file

❑ `filesys_remove()`

- ◆ It is called by System call '`remove()`'.
- ◆ Set flag `removed` on inode true.
- ◆ Remove directory entry from root directory.



Remove a file

ex) 'testfile' remove

```
bool filesys_remove ("testfile")
```

Main Memory

inode list

4. Set removed true

```
"testfile" inode {  
    bool removed;  
    ...  
}
```

head

"root" inode{}

tail

root directory entries

inode = 7

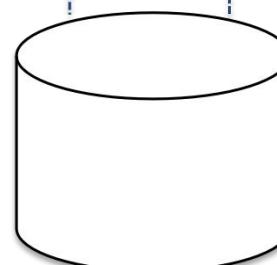
testfile

in_use = 0

3. Set in_use false in
'testfile' entry

1. Read inode of
root directory

2. Read entries o
f root directory



DISK

Remove a file

❑ filesys_remove

- ◆ Remove target file entry from directory.
- ◆ Set removed flag in in-memory inode true.

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;  
}
```

Remove a file

▣ dir_remove

- ◆ Remove entry of target file in directory.
- ◆ Set removed flag in In-memory inode true.
- ◆ Write updated directory entry on disk.

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) != si  
zeof e)  
        goto done;  
    ...
```

Remove a file

pintos/src/filesys/directory.c – dir_remove()

```
...
/* Open inode. */
inode = inode_open (e.inode_sector);
...
/* Remove inode. */
inode_remove (inode);
success = true;
done:
    inode_close (inode);
    return success;
}
```

- ◆ inode_open() : Add in-memory inode to open_inodes list.
- ◆ inode_remove() : Set removed flag in in-memory inode true.

Summary

- Filesystem format
- Create a file
- Create a directory
- Open a file
- Remove a file.

Appendix

Type of block devices

- ❑ In Pintos, there are four types of block devices.
 - ◆ BLOCK_KERNEL: Storage to save kernel binary file.
 - ◆ BLOCK_FILESYS: Storage to be used to '/' filesystem partition.
 - ◆ BLOCK_SCRATCH: SCRATCH filesystem where files are saved to be forwarded to/from virtual machine
 - ◆ BLOCK_SWAP: Storage to be used to swap partition

Initializing the block devices (1)

threads/init.c:76

```
int
main (void)
{
    char **argv;

    /* Clear BSS. */
    bss_init ();
    ...

#ifndef FILESYS
    /* Initialize file system. */
    ide_init ();
    locate_block_devices ();
    filesys_init (format_filesys);
#endif

    printf ("Boot complete.\n");

    /* Run actions specified on kernel command line. */
    run_actions (argv);

    /* Finish up. */
    shutdown ();
    thread_exit ();
}
```

devices/ide.c:101

```
void ide_init (void)
{
    size_t chan_no;

    for (chan_no = 0; chan_no < CHANNEL_CNT; chan_no++)
    {
        struct channel *c = &channels[chan_no];
        int dev_no;

        ...

        /* Read hard disk identity information. */
        for (dev_no = 0; dev_no < 2; dev_no++)
            if (c->devices[dev_no].is_ata)
                identify_ata_device (&c->devices[dev_no]);
    }
}
```

- Find ATA device from each channel.
- Identify and register the device (`identify_ata_device()`) (next slide)

Initializing the block devices (2)

devices/ide.c:101

```
static void
identify_ata_device (struct ata_disk *d)
{
    struct channel *c = d->channel;
    char id[BLOCK_SECTOR_SIZE];
    block_sector_t capacity;
    char *model, *serial;
    char extra_info[128];
    struct block *block;

    ASSERT (d->is_ata);

    /* Send the IDENTIFY DEVICE command, wait for an interrupt
       indicating the device's response is ready, and read the data
       into our buffer. */
    select_device_wait (d);
    issue_pio_command (c, CMD_IDENTIFY_DEVICE);
    sema_down (&c->completion_wait);
    if (!wait_while_busy (d))
    {
        d->is_ata = false;
        return;
    }
    input_sector (c, id);
    ...
}
```

- ▣ Identify block device, using IDENTIFY_DEVICE command.
- ▣ send command to the block device and wait for completion.
- ▣ The command returns the information about block device which is saved in id.

Initializing the block devices (3)

```
...
/* Calculate capacity.
   Read model name and serial number. */
capacity = *(uint32_t *) &id[60 * 2];
model = descramble_ata_string (&id[10 * 2], 20);
serial = descramble_ata_string (&id[27 * 2], 40);
snprintf (extra_info, sizeof extra_info,
          "model \"%s\", serial \"%s\"", model, serial);

/* Disable access to IDE disks over 1 GB, which are likely
   physical IDE disks rather than virtual ones. If we don't
   allow access to those, we're less likely to scribble on
   someone's important data. You can disable this check by
   hand if you really want to do so. */
if (capacity >= 1024 * 1024 * 1024 / BLOCK_SECTOR_SIZE)
{
    printf ("%s: ignoring ", d->name);
    print_human_readable_size (capacity * 512);
    printf ("disk for safety\n");
    d->is_ata = false;
    return;
}

/* Register. */
block = block_register (d->name, BLOCK_RAW, extra_info, capacity,
                      &ide_operations, d);
partition_scan (block);
}
```

- ▣ Print the information about the block device.
- ▣ Prevent attaching block device bigger than 1GB.
- ▣ Register the block device.

Initializing the block devices (4)

devices/ide.c:101

```
struct block *
block_register (const char *name, enum block_type type,
                const char *extra_info, block_sector_t size,
                const struct block_operations *ops, void *aux)
{
    struct block *block = malloc (sizeof *block);
    if (block == NULL)
        PANIC ("Failed to allocate memory for block device descriptor");

    list_push_back (&all_blocks, &block->list_elem);
    strlcpy (block->name, name, sizeof block->name);
    block->type = type;
    block->size = size;
    block->ops = ops;
    block->aux = aux;
    block->read_cnt = 0;
    block->write_cnt = 0;

    printf ("%s: %" PRDSNu " sectors (%s, %s)", block->name, block->size,
            print_human_readable_size ((uint64_t) block->size * BLOCK_SECTOR_SIZE),
            block->extra_info ? extra_info : "");
    if (extra_info != NULL)
        printf ("\n");
    else
        printf ("\n");

    return block;
}
```

- ❑ Create and fill the `struct block` object out.
- ❑ Push the object into `all_blocks` list.

Initializing the block devices (5)

threads/init.c:76

```
int
main (void)
{
    char **argv;

    /* Clear BSS. */
    bss_init ();
    ...

#ifndef FILESYS
    /* Initialize file system. */
    ide_init ();
    locate_block_devices ();
    filesystem_init (format_filesys);
#endif

    printf ("Boot complete.\n");

    /* Run actions specified on kernel command line. */
    run_actions (argv);

    /* Finish up. */
    shutdown ();
    thread_exit ();
}
```

thread/init.c:391

```
static void locate_block_devices (void)
{
    locate_block_device (BLOCK_FILESYS, filesystem_bdev_name);
    locate_block_device (BLOCK_SCRATCH, scratch_bdev_name);
    locate_block_device (BLOCK_SWAP, swap_bdev_name);
}
```

- ❑ Register three block devices; BLOCK_FILESYS, BLOCK_SCRATCH, BLOCK_SWAP.
- ❑ BLOCK_KERNEL don't need to be registered, because kernel is already loaded.

Initializing the block devices (6)

thread/init.c:405

```
static void locate_block_device (enum block_type role, const char *name)
{
    struct block *block = NULL;

    if (name != NULL) {
        block = block_get_by_name (name);
        if (block == NULL)
            PANIC ("No such block device \"%s\"", name);
    } else {
        for (block = block_first (); block != NULL; block = block_next (block))
            if (block_type (block) == role)
                break;
    }

    if (block != NULL) {
        printf ("%s: using %s\n", block_type_name (role), block_name (block));
        block_set_role (role, block);
    }
}
```

devices/block.c:61

```
void
block_set_role (enum block_type role, struct block *block)
{
    ASSERT (role < BLOCK_ROLE_CNT);
    block_by_role[role] = block;
}
```

- ▣ Find struct block object by name or type.
- ▣ Set block_by_role[] array with the address of the struct block object.