

# LAB2: USER PROGRAM

Operating Systems 2015 Spring by Euseong Seo

# Overview

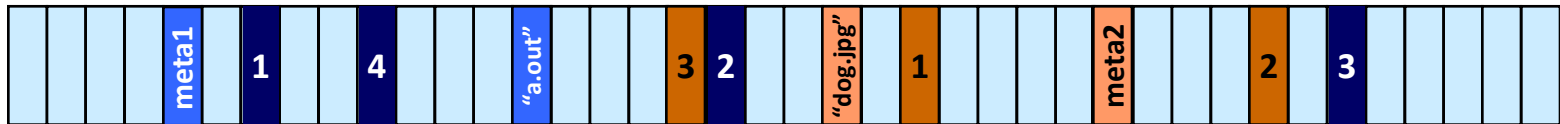
- Supporting User Programs
  1. Provide file system accesses
  2. Process wait / exit
  3. Pass arguments
  4. Provide system calls

# File Systems (1)

- Block device abstraction
  - A set of sectors (sector = 512 bytes)



- File systems
  - $\langle \text{filename, metadata, data} \rangle \rightarrow \langle \text{a set of sectors} \rangle$



# File Systems (2)

- Pintos file system
  - ▣ No internal synchronization
  - ▣ File size is fixed at creation time
  - ▣ File data is allocated as a single extent (i.e., in a contiguous range of sectors on disk)
  - ▣ No subdirectories
  - ▣ File names are limited to 14 characters
  - ▣ Unix-like semantics: If a file is open when it is removed, it may still be accessed by any threads that it open, until the last one closes it.

# File Systems (3)

## □ Using the Pintos file system

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

- Creates a 2MB disk named “fs.dsk”

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

- Formats the disk (-f)

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

- Put the file “../../examples/echo” to the Pintos file system under the name “echo”

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

- Run the executable file “echo”, passing argument “x”

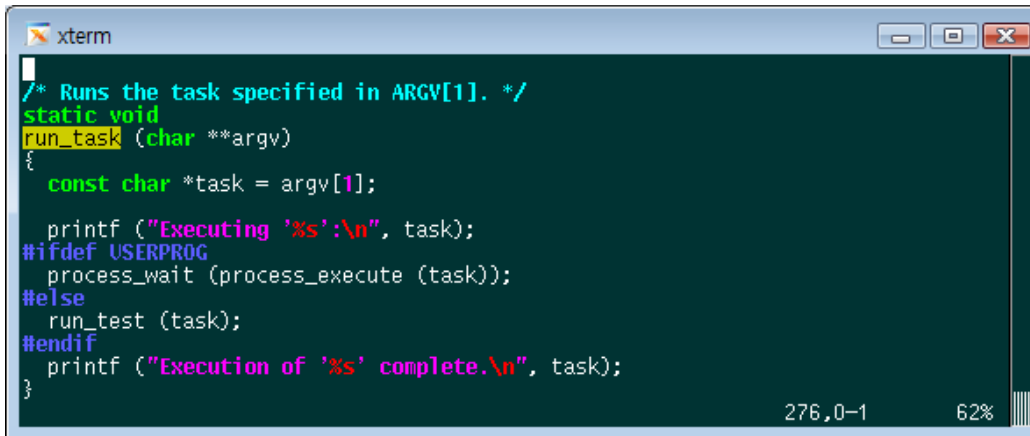
# File Systems (4)

## □ Note:

- User programs are loaded from the file system
  - To run a program, it should be moved to the file system.
  - Executable files are in ELF format
  - Data files can be also stored in the file system
- Refer to the interfaces provided by the file system
  - In `filesys/filesys.h` and `filesys/file.h`
  - You will need them to implement system calls related to file systems
    - e.g., `create()`, `remove()`, `open()`, `filesize()`, `read()`, `write()`, `seek()`, `tell()`, `close()`
- You don't need to modify the file system code (under the `filesys` directory) for this project

# Processes (1)

- Processes in Pintos
  - ▣ Each process has only one thread
  - ▣ The initial program is loaded by the Pintos
    - In `run_task()` @ `threads/init.c`



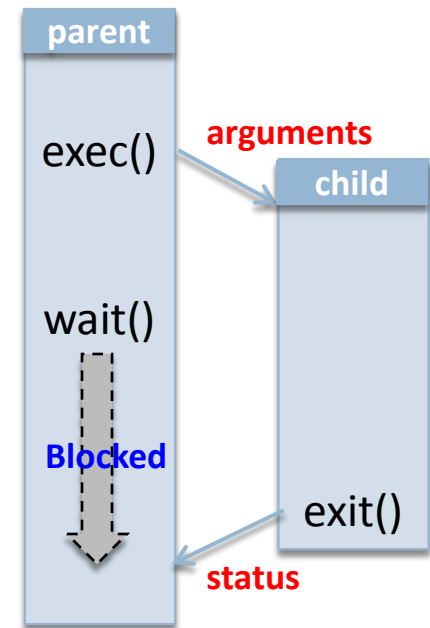
```
xterm
/* Runs the task specified in ARGV[1]. */
static void
run_task (char **argv)
{
    const char *task = argv[1];
    printf ("Executing '%s':\n", task);
#ifdef USERPROG
    process_wait (process_execute (task));
#else
    run_test (task);
#endif
    printf ("Execution of '%s' complete.\n", task);
}
```

276,0-1 62%

- ▣ New processes are created by the `exec()` system call
  - `fork()` + `exec()` in Unix

# Processes (2)

- Managing processes
  - A process is identified by PID (process id)
  - The `exec()` system call returns the child's pid
  - Processes form a tree
    - A process can have many child processes
    - Each process has a unique parent process
  - The `exit()` system call terminates the current process, returning **status** to the kernel
  - The parent process waits for the termination of its child process by calling `wait()`, and **gets the status**



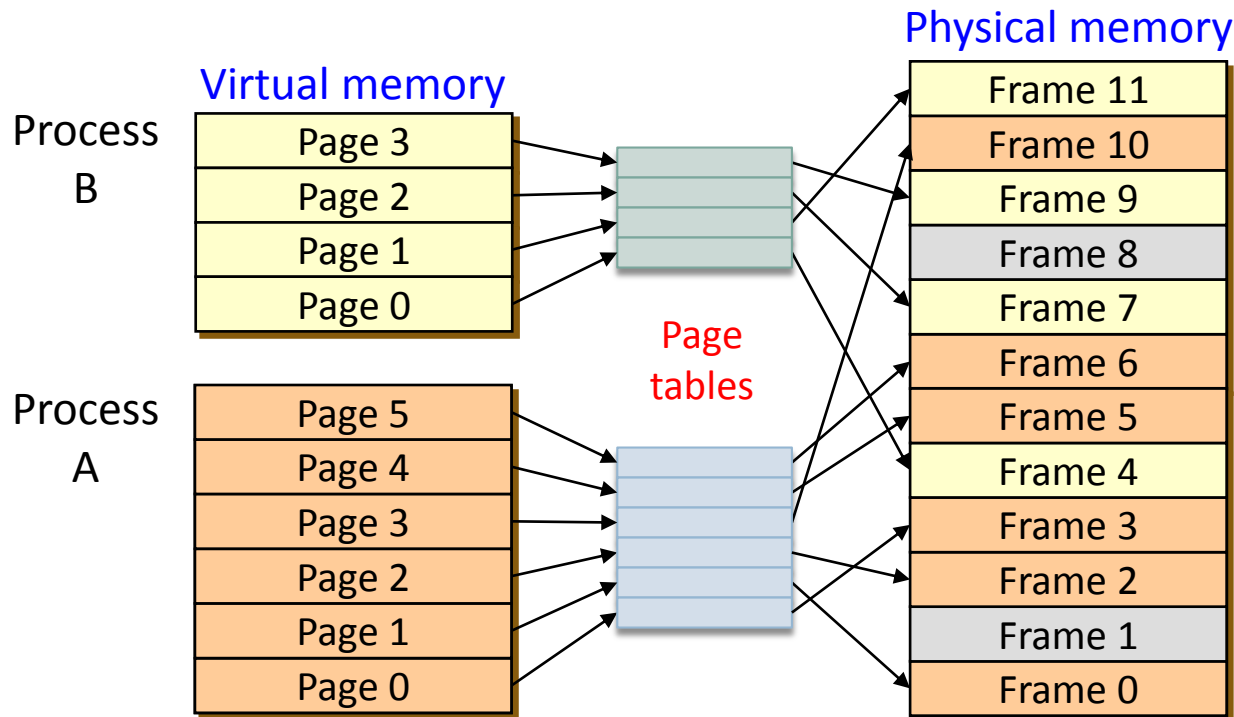


# Processes (3)

- Implementing processes
  - ▣ A process is created, scheduled, and managed using a **kernel thread**
  - ▣ Need to keep track of additional information for processes
    - PID, parent PID, file descriptors, etc.
  - ▣ **Require a way to deliver a status of child processes**
  - ▣ `userprog/process.c`

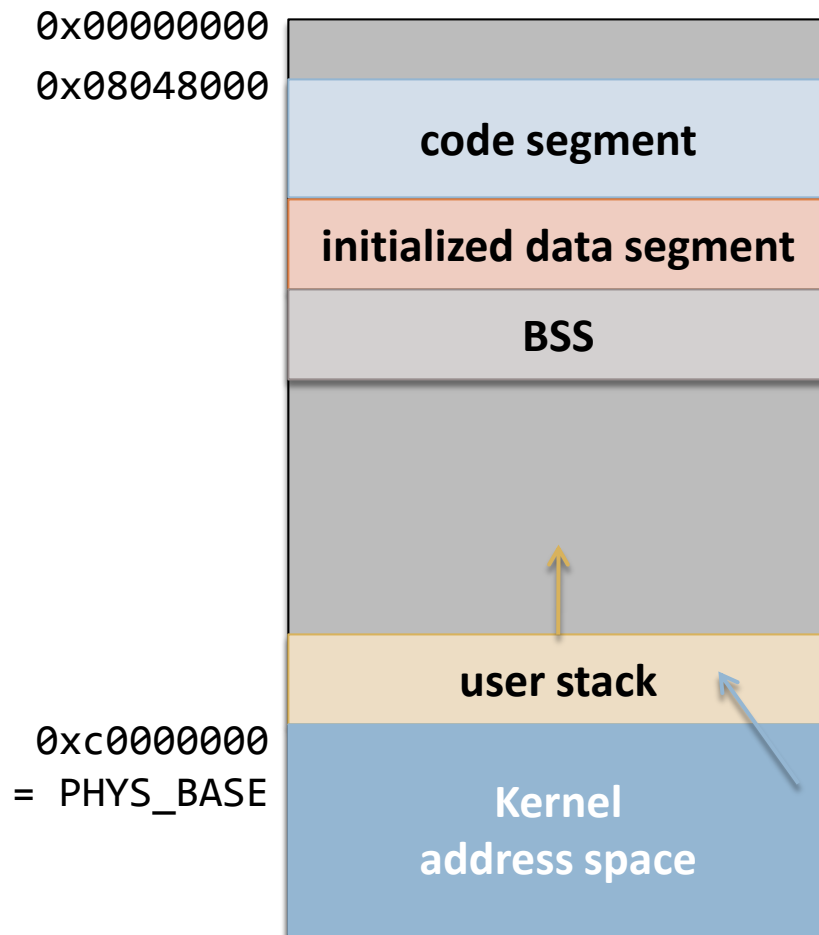
# Address Spaces (1)

- Virtual memory
  - ▣ Gives an illusion that a process has a large memory
  - ▣ **Each process** has a separate virtual address space



# Address Spaces (2)

## □ Typical memory layout in Pintos



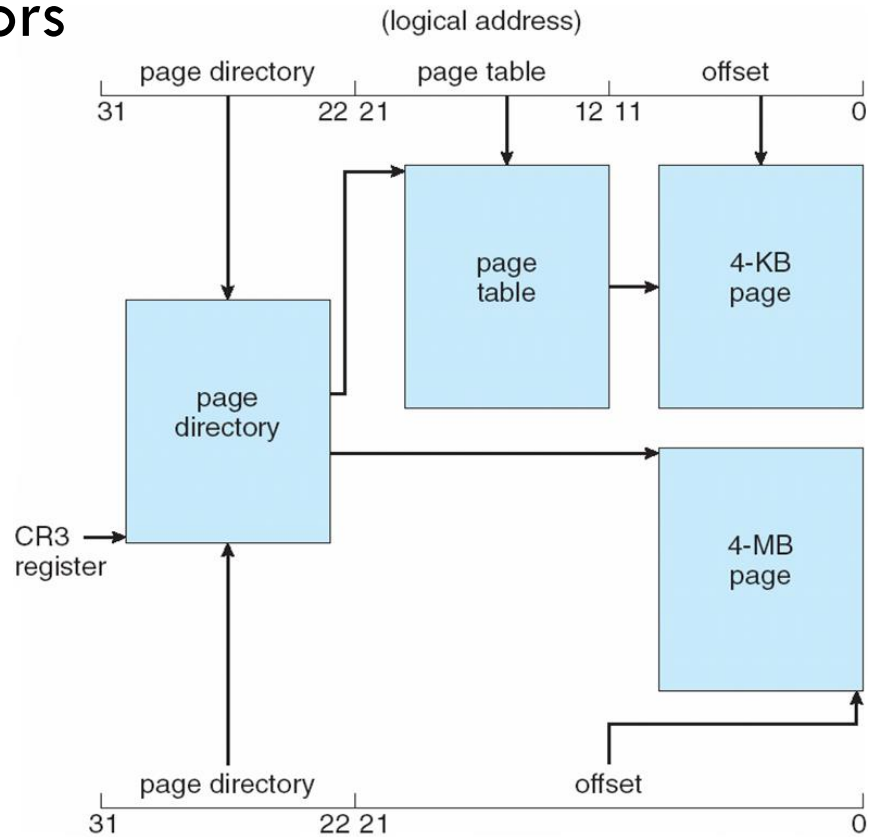
Only the used memory regions have valid mapping entries in page tables

} fixed in size until Project 3  
(1 page size : 4KB)

# Address Spaces (3)

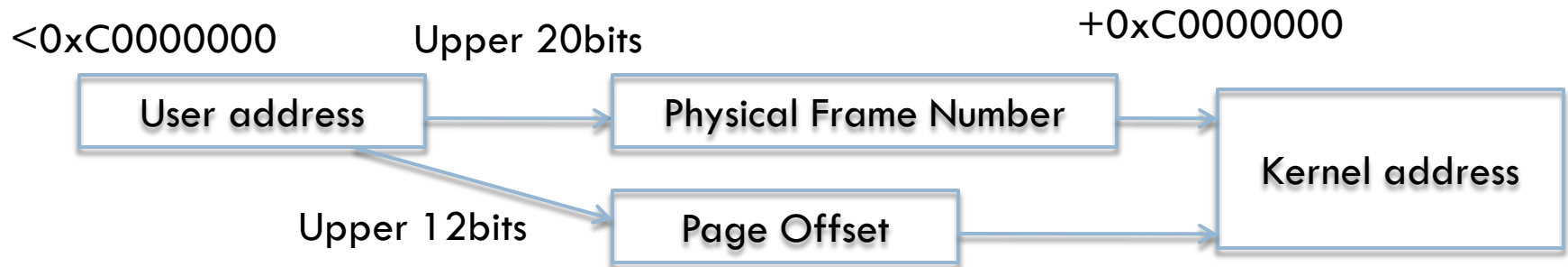
## □ Page tables in x86 processors

- Common page size: 4KB
- 2-level page tables
  - Page directory
  - Page table
- Page tables allocated when a process is created
- The address of the page directory is a part of process context
  - Changed on context switch
  - `(struct thread *) t->pagedir`



# Address Spaces (4)

- `pagedir_get_page` (userprog/pagedir.c)



# Address Spaces (5)

## □ Page faults

- A user program can only access its own user virtual memory (0x00000000 to 0xbfffffff)
- An attempt to access kernel virtual memory causes a page fault (page\_fault() @ userprog/exception.c)
- On page fault, the process will be terminated
- Kernel threads can access both kernel virtual memory and, if a user process is running, the user virtual memory of the running process.
- However, even in the kernel, an attempt to access memory at an unmapped user virtual address will cause a page fault.

# Program Arguments (1)

## □ Program startup

- `_start()` is the entry point for user programs

  - Linked with user programs

  - In `lib/user/entry.c`

- `_start()` will make a procedure call to `main()`

- `main()` requires two parameters: `argc`, `argv`

- Argum

	0xbffffe7c	3
	0xbffffe78	2
	0xbffffe74	1
SP →	0xbffffe70	return address

**x86 Calling Convention for `f(1, 2, 3)`**

# Program Arguments (2)

PHYS\_BASE = 0xc0000000

padding

esp

	Kernel space			
0xbfffffffcc	'b'	'a'	'r'	00
0xbfffffff8	'f'	'o'	'o'	00
0xbfffffff4	00	'-'	'l'	00
0xbffffff0	'n'	'/'	'l'	's'
0xbffffffec	00	'/'	'b'	'i'
0xbffffffe8	00	00	00	00
0xbffffffe4	fc	ff	ff	bf
0xbffffffe0	f8	ff	ff	bf
0xbffffffdc	f5	ff	ff	bf
0xbffffffd8	ed	ff	ff	bf
0xbffffffd4	d8	ff	ff	bf
0xbffffffd0	04	00	00	00
0xbffffffcc	return address			

command:  
/bin/ls -l foo bar

0	argv[4]
0xbffffffc	argv[3]
0xbfffffff8	argv[2]
0xbffffff5	argv[1]
0xbffffffed	argv[0]
0xbffffffd8	argv
4	argc



# Program Arguments(3)

- strtok function
  - #define strtok dont\_use\_strtok\_use\_strtok\_r
  - strtok\_r is safety
- strcpy, strcat ....

# System Calls (1)

- A user program requests a service through **system call**
- Library function has a system call
  - ▣ lib/syscall.c
- All system calls in user space use a interrupt
  - ▣ Push arguments and **syscall number**
  - ▣ Interrupt 0x30
  - ▣ Add stack pointer (the number of argument \* 4+4)

# System Calls (2)

- Interrupt 0x30 is occurred, Interrupt handler (syscall\_handler) should control it by providing appropriate services
  - ▣ Stack pointer is always user address
  - ▣ Handler can distinguish services (open, close, read, write, create, delete ....) and the number of arguments by **syscall number**

# System Calls (3)

```
lib/user/syscall.c
/* Invokes syscall NUMBER, passing argument ARG0, and returns the
return value as an 'int'. */
#define syscall1(NUMBER, ARG0)
{
    int retval;
    asm volatile
    ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp"
     : "=a" (retval)
     : [number] "i" (NUMBER),
       [arg0] "g" (ARG0)
     : "memory");
    retval;
}

int
open (const char *file)
{
    return syscall1 (SYS_OPEN, file);
}
"syscall.c" [Modified] 185 lines —8%—
```

```
lib/syscall-nr.h
/* System call numbers. */
enum
{
    /* Projects 2 and later. */
    SYS_HALT, /* Halt the operating system. */
    SYS_EXIT, /* Terminate this process. */
    SYS_EXEC, /* Start another process. */
    SYS_WAIT, /* Wait for a child process to die. */
    SYS_CREATE, /* Create a file. */
    SYS_REMOVE, /* Delete a file. */
    SYS_OPEN, /* Open a file. */
    SYS_FILESIZE, /* Obtain a file's size. */
    SYS_READ, /* Read from a file. */
    SYS_WRITE, /* Write to a file. */
    SYS_SEEK, /* Change position in a file. */
    SYS_TELL, /* Report current position in a file. */
    SYS_CLOSE, /* Close a file. */
}
3,0-1 12%
```

# System Calls (4)

## □ INT \$0x30

Interrupt Descriptor Table (IDT)

0x00	Divide error
0x0d	Page fault exception
0x30	INT \$0x30
0xff	

```
userprog/syscall.c
#include "userprog/syscall.h"
#include <stdio.h>
#include <syscall-nr.h>
#include "threads/interrupt.h"
#include "threads/thread.h"

static void syscall_handler (struct intr_frame *);

void
syscall_init (void)
{
    intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
}

static void
syscall_handler (struct intr_frame *f UNUSED)
{
    printf ("system call!\n");
    thread_exit ();
}

"syscall.c" 20 lines —5%—          1,1          All
```

# System Calls (5)

- Getting system call number & arguments
  - ▣ System call numbers are defined in `<lib/syscall-nr.h>`
  - ▣ System call number and arguments are put in the caller's **user stack**
  - ▣ The caller's register context is available to the exception handler (`struct intr_frame *f`)
  - ▣ Use the caller's stack pointer to access them
    - `(struct intr_frame *) f->esp`
  - ▣ Save the return value to the caller's EAX register
    - `(struct intr_frame *) f->eax`

# System Calls (6)

- Accessing user memory
  - Can you trust the information provided by user?
    - Stack pointer itself!
    - System call arguments, especially pointers
      - Buffer addresses, string addresses, etc.
  - What's wrong with pointers provided by user?
    - What if it's a NULL pointer?
    - What if it points to the kernel address ( $\geq$  PHYS\_BASE)?
    - What if it points to the unmapped user address?
  - Requires various sanity checks on system call entry
    - If any pointer is invalid, kill the process with -1 exit code
    - Return error if the system call number is wrong
    - Return error if any argument value is not what you want, ...

# System Calls (7)

- Accessing user memory: Option 1
  - Verify the validity of a user-provided pointer, then dereference it
  - How to verify?
    - Traverse the user's page tables
    - The valid address should have the corresponding PTE (page table entry)
    - “Present” flag in the PTE should be set
    - Refer to `userprog/pagedir.c`, `threads/vaddr.h`, and `threads/pte.h`
  - Simple
  - Pessimistic approach



# System Calls (8)

- Accessing user memory: Option 2
  - ▣ Check only that a user pointer  $< \text{PHYS\_BASE}$ , then dereference it
  - ▣ Use `get_user()` and `put_user()` routines to read from or write to user memory (provided in the Pintos documentation)
  - ▣ Detects and handles invalid user pointer in the page fault handler
    - In `page_fault()` @ `userprog/exception.c`
    - For a page fault occurred in the kernel, set `EAX` to `0xffffffff` and copy its former value into `EIP`
  - ▣ Optimistic approach, faster (used in Linux)

# System Calls (8)

## □ System calls related to processes

```
void    exit (int status);  
pid_t  exec (const char *cmd_line);  
int     wait (pid_t pid);
```

- All of a process's resources must be freed on exit()
- The child can exit() before the parent performs wait()
- A process can perform wait() only for its children
- Wait() can be called twice for the same process
  - The second wait() should fail
- Nested waits are possible:  $A \rightarrow B, B \rightarrow C$
- Pintos should not be terminate until the initial process exits

# System Calls (9)

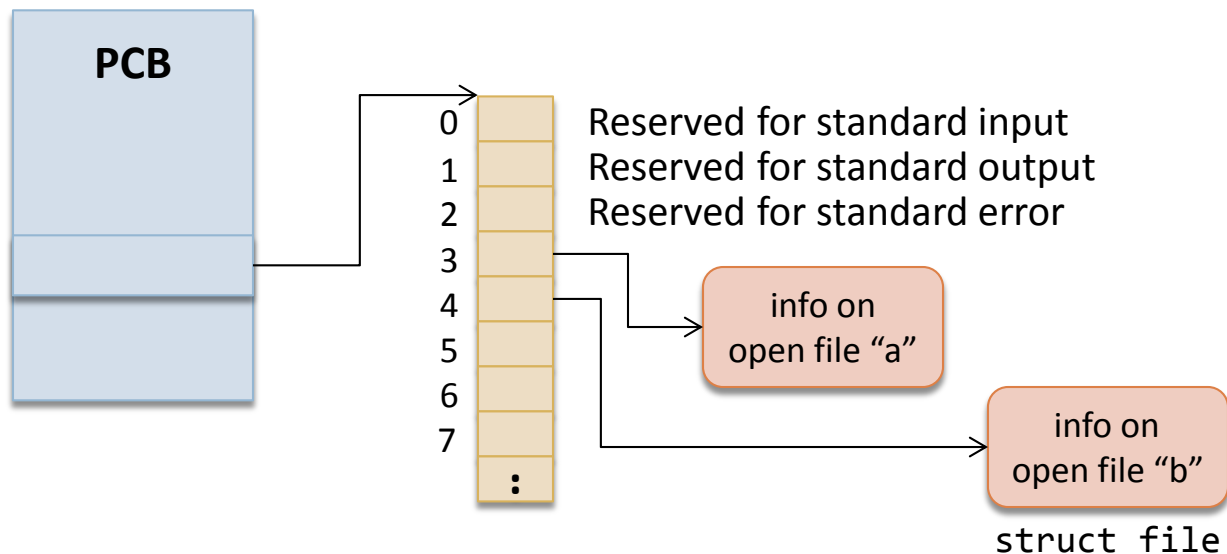
## □ System calls related to files

```
bool  create (const char *file, unsigned initial_size);
bool  remove (const char *file);
int   open  (const char *file);
int   filesize (int fd);
int   read  (int fd, void *buffer, unsigned size);
int   write (int fd, void *buffer, unsigned size);
void  seek  (int fd, unsigned position);
unsigned tell (int fd);
void  close (int fd);
```

- create()/remove()/open() work on file names
- The rest of them work on file descriptors

# System Calls (10)

- File descriptor
  - ▣ An integer (C type `int`)
  - ▣ An index for an entry in a kernel-resident data structure containing the details of all open files (file descriptor tables)



# System Calls (11)

- Implementing system calls related to files
  - ▣ No need to change the code in the `filesys` directory
  - ▣ The existing routines in the `filesys` directory work on the “file” structure (`struct file *`)
  - ▣ Maintain a mapping structure from a file descriptor to the corresponding “file” structure
  - ▣ Deny writes to a running process’s executable file
  - ▣ Ensure only one process at a time is executing the file system code

# Tips

- First things to implement
  - ▣ Argument passing
  - ▣ System call infrastructure
    - Get system call number
    - Get system call arguments
  - ▣ write() system call for file descriptor 1
  - ▣ exit()
  - ▣ process\_wait() & wait()
  - ▣ Build project 2 on top of your project 1 or start fresh
  - ▣ Work in the userprog directory

# Submission

## □ Due

- May 18, 11:59PM

- Fill out the design document and save it with PDF format (GDHong\_2012345678.pdf)

  - NO .doc or .hwp

- Tar and gzip your Pintos source codes \$ cd pintos \$ (cd src/userprog; make clean)

  - \$ tar cvzf GDHong\_2012345678.tar.gz src

- Good luck! 😊