Project 2: User Programs

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

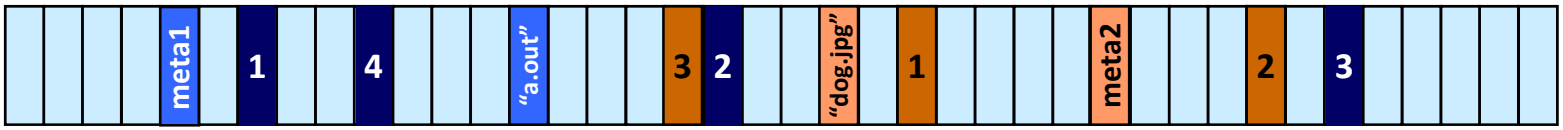
Supporting User Programs

- **What should be done to run user programs?**
  1. Provide file system accesses
  2. Manage "processes"
  3. Setup an address space for each process
  4. Pass arguments
  5. Provide system calls

File Systems (1)

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

- **File systems**
  - <filename, metadata, data> → <a set of sectors>



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
  - A system crash may corrupt the disk
  - 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 fs.dsk 2
```

  - Creates a 2MB disk named "fs.dsk"

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

  - Formats the disk (-f) and exits as soon as the format is done (-q)

```
$ 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"

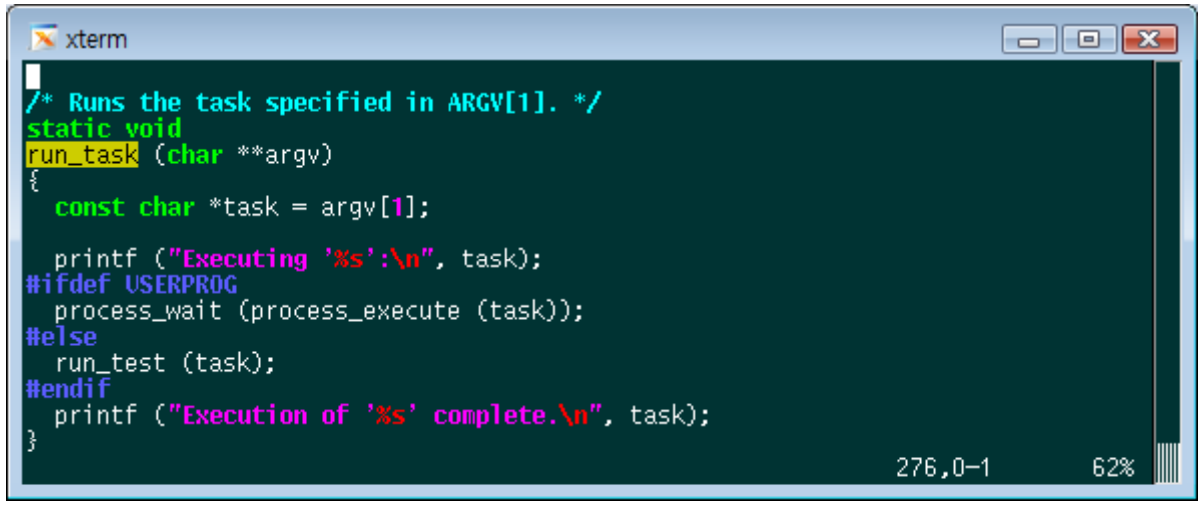
```
$ pintos -fs-disk=2 -p ../../examples/echo -a
  echo -- -f -q run 'echo x'
```

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)

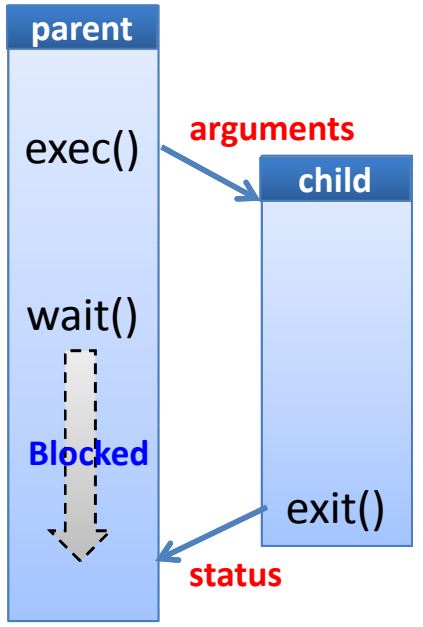
- **Processes in Pintos**
  - Each process has only one thread
  - The initial program is loaded by the Pintos
    - In `run_task() @ threads/init.c`



  - 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

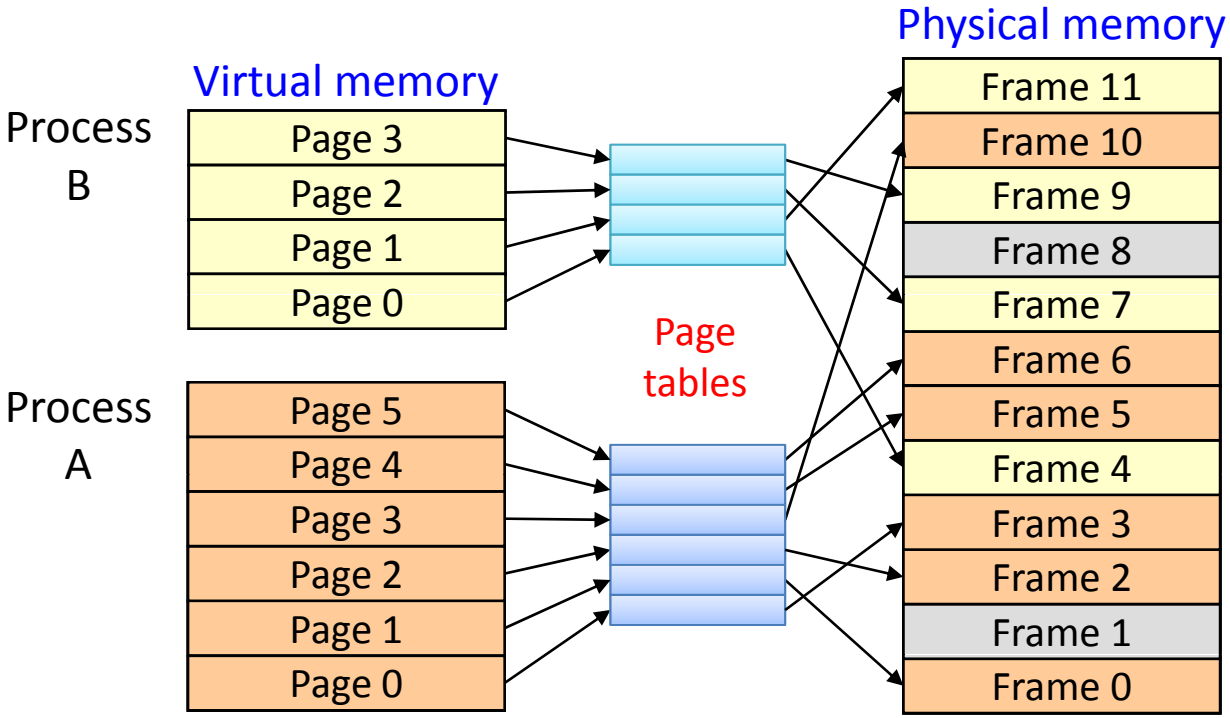


Processes (3)

- **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 (more on later), etc.
  - Basic skeleton codes are available in `userprog/process.c`

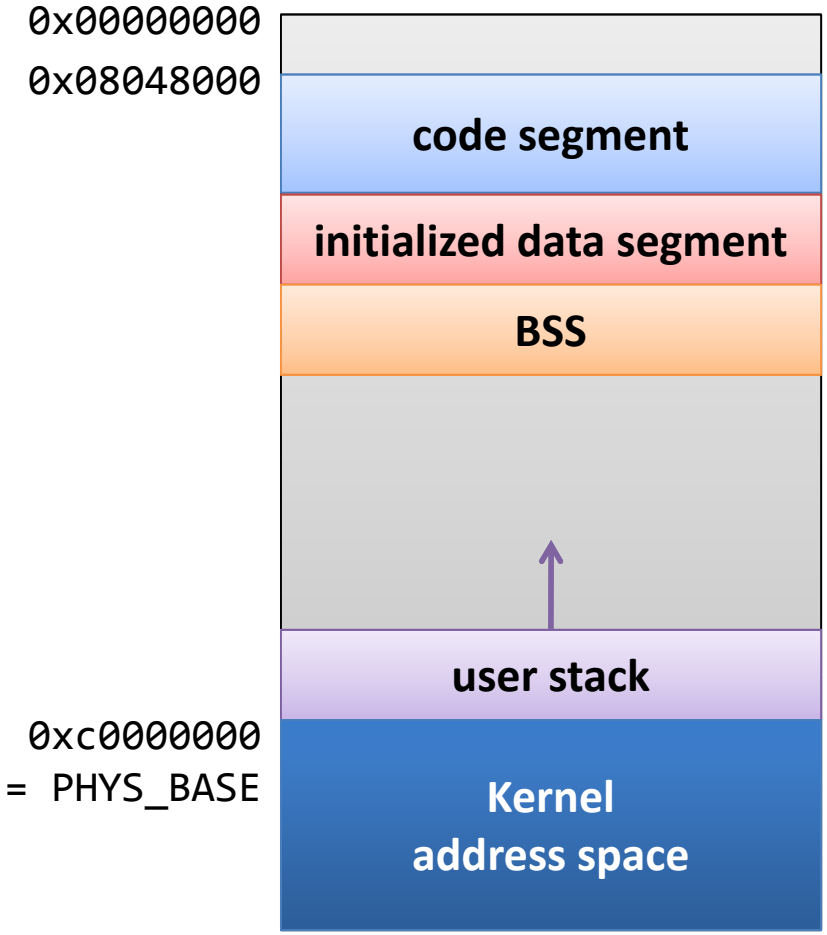
Address Spaces (1)

- **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

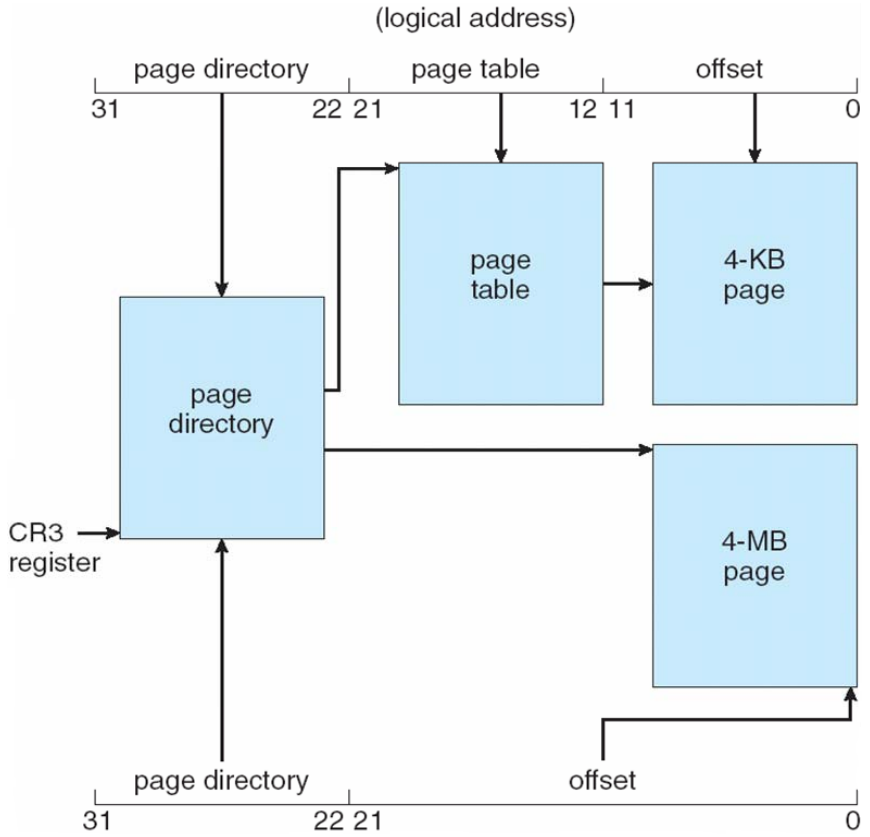


Only the used memory regions have valid mapping entries in page tables

fixed in size until Project 3

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)

- **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.

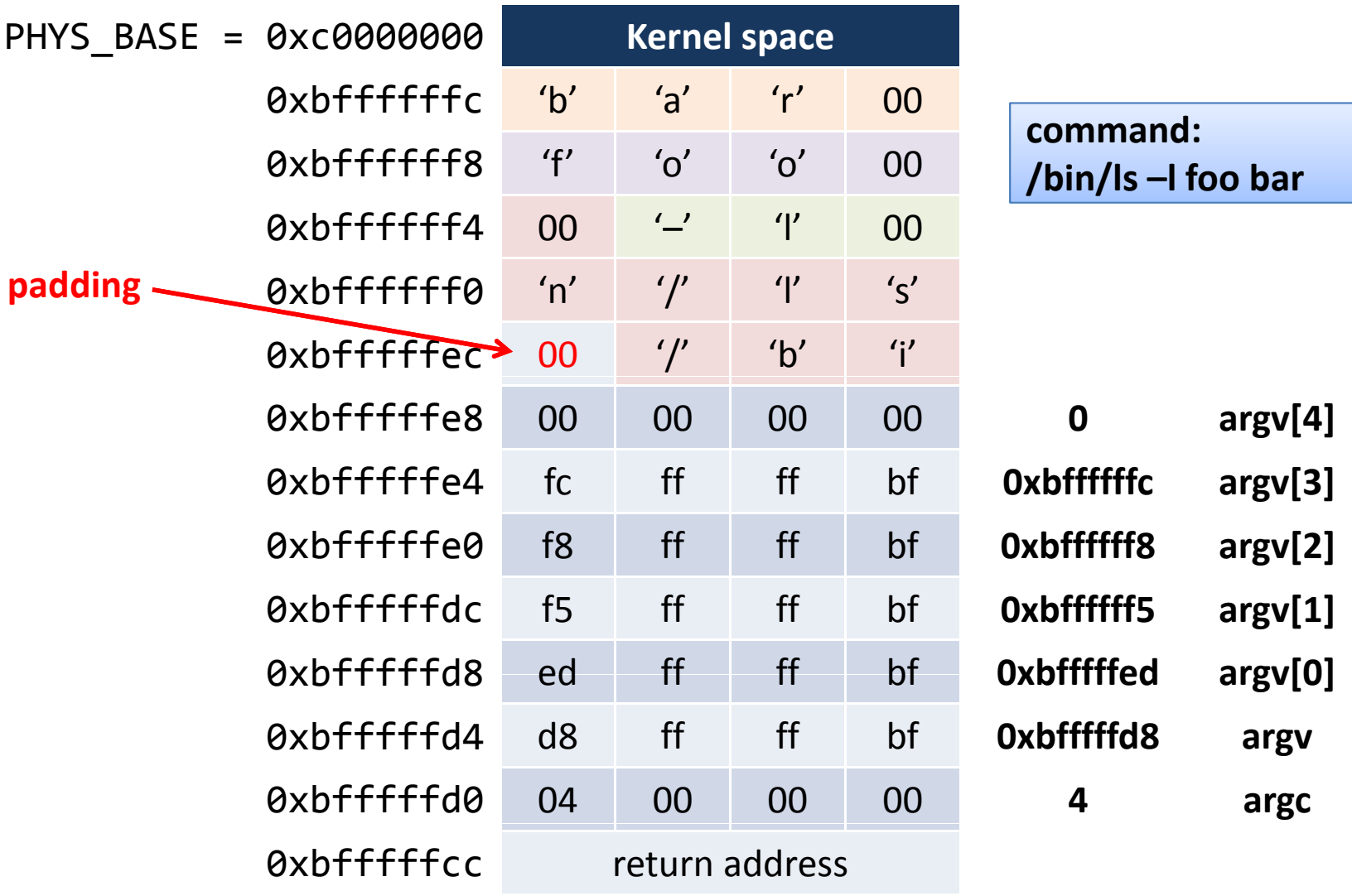
Program Arguments (1)

- **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`
  - Arguments are passed via the user stack

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

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

Program Arguments (2)



System Calls (1)

- System calls look just like library functions

examples/cat.c

```
/* cat.c

   Prints files specified on command line to the console. */

#include <stdio.h>
#include <syscall.h>

int
main (int argc, char *argv[])
{
  bool success = true;
  int i;

  for (i = 1; i < argc; i++)
    {
      int fd = open (argv[i]);
      if (fd < 0)
        {
          printf ("%s: open failed\n", argv[i]);
          success = false;
          continue;
        }
      for (;;)
        {
          char buffer[1024];
          int bytes_read = read (fd, buffer, sizeof buffer);
          if (bytes_read == 0)
            break;
          write (STDOUT_FILENO, buffer, bytes_read);
        }
      close (fd);
    }
  return success ? EXIT_SUCCESS : EXIT_FAILURE;
}
"cat.c" 34 lines --2%--                                  1,1           All
```

System Calls (2)

- However, they are implemented differently

lib/user/syscall.c

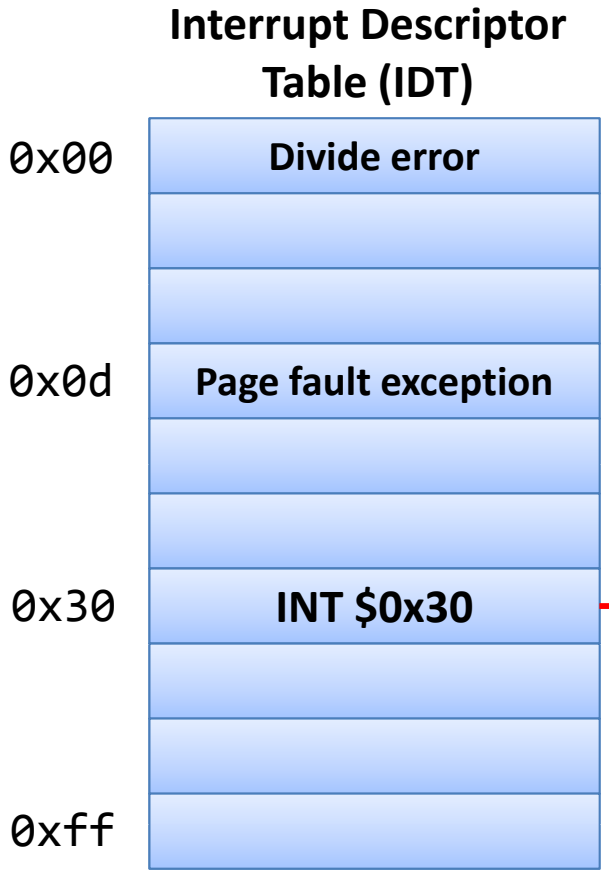
```
/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an 'int'. */
#define syscall1(NUMBER, ARG0)                                          \
        ({                                                              \
          int retval;                                                   \
          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 (3)

- INT $0x30



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 (4)

- **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`
  - Note that you're accessing user address space in the kernel!

System Calls (5)

- **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 (>= 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 (6)

- **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 (7)

- **Accessing user memory: Option 2**
  - Check only that a user pointer < 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 → B, B → C
  - Pintos should not be terminate until the initial process exits

System Calls (9)

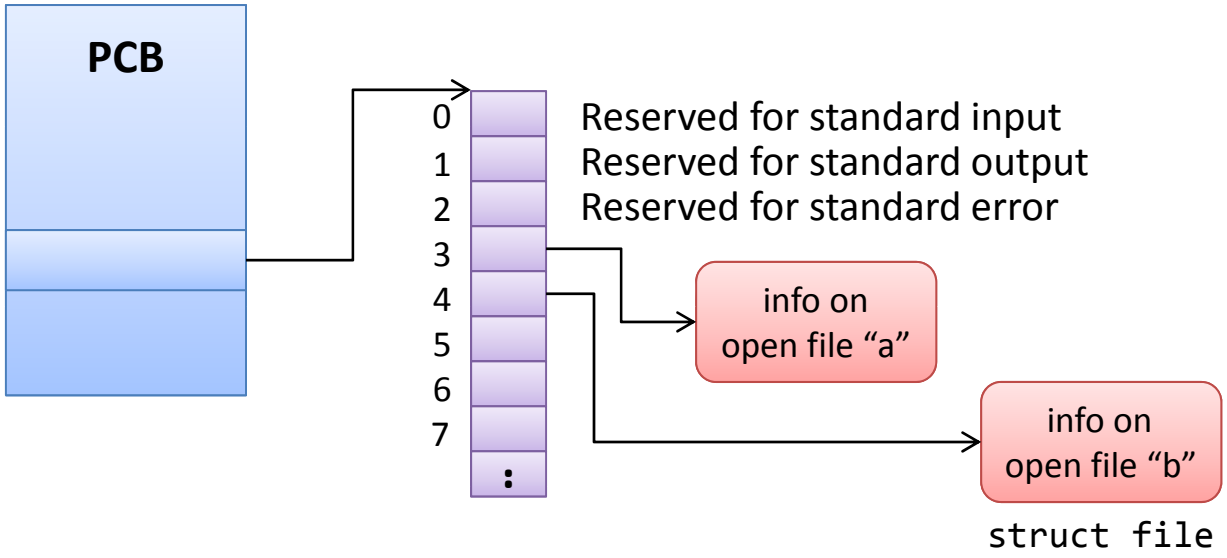
- **System calls related to files**

```
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**
  - November 10, 11:59PM
  - Fill out the design document (`userprog.txt`) and put it in your source tree (`pintos/src/userprog`)
  - Tar and gzip your Pintos source codes

```
$ cd pintos
$ (cd src/userprog; make clean)
$ tar cvzf TeamName.tar.gz ./src
```

  - Send it to the instructor via e-mail (NOT to the GoogleGroups!!)
  - Hand in the printed version of your design document in the following class on November 11.