Project 2: User Programs and System Calls

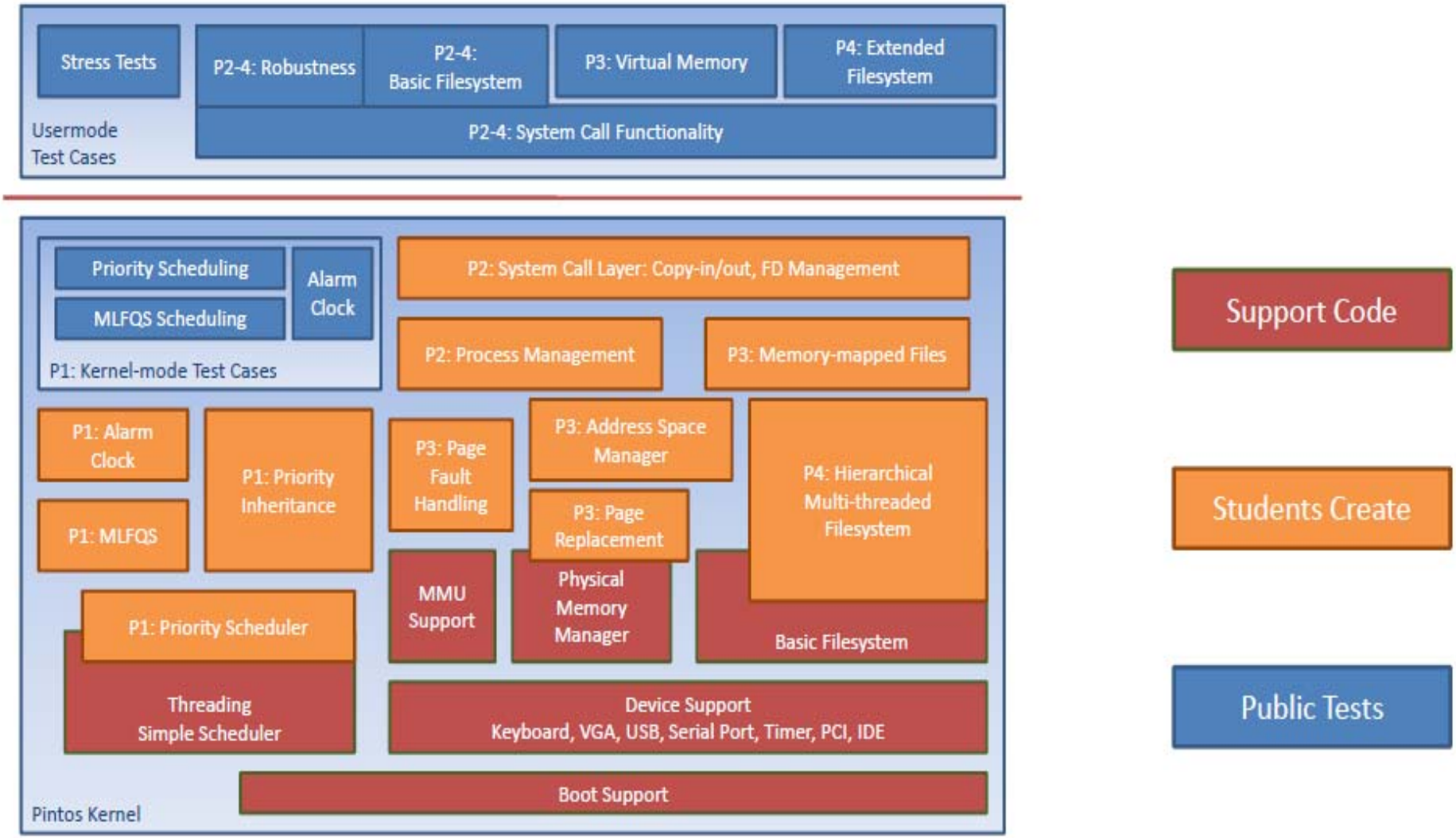
Dennis Ritchie: 1941-2011

**"UNIX is basically a simple operating system,
but you have to be a genius to understand the simplicity"**

-- Dennis Ritchie

http://www.nytimes.com/2011/10/14/technology/dennis-ritchie-programming-trailblazer-dies-at-70.html

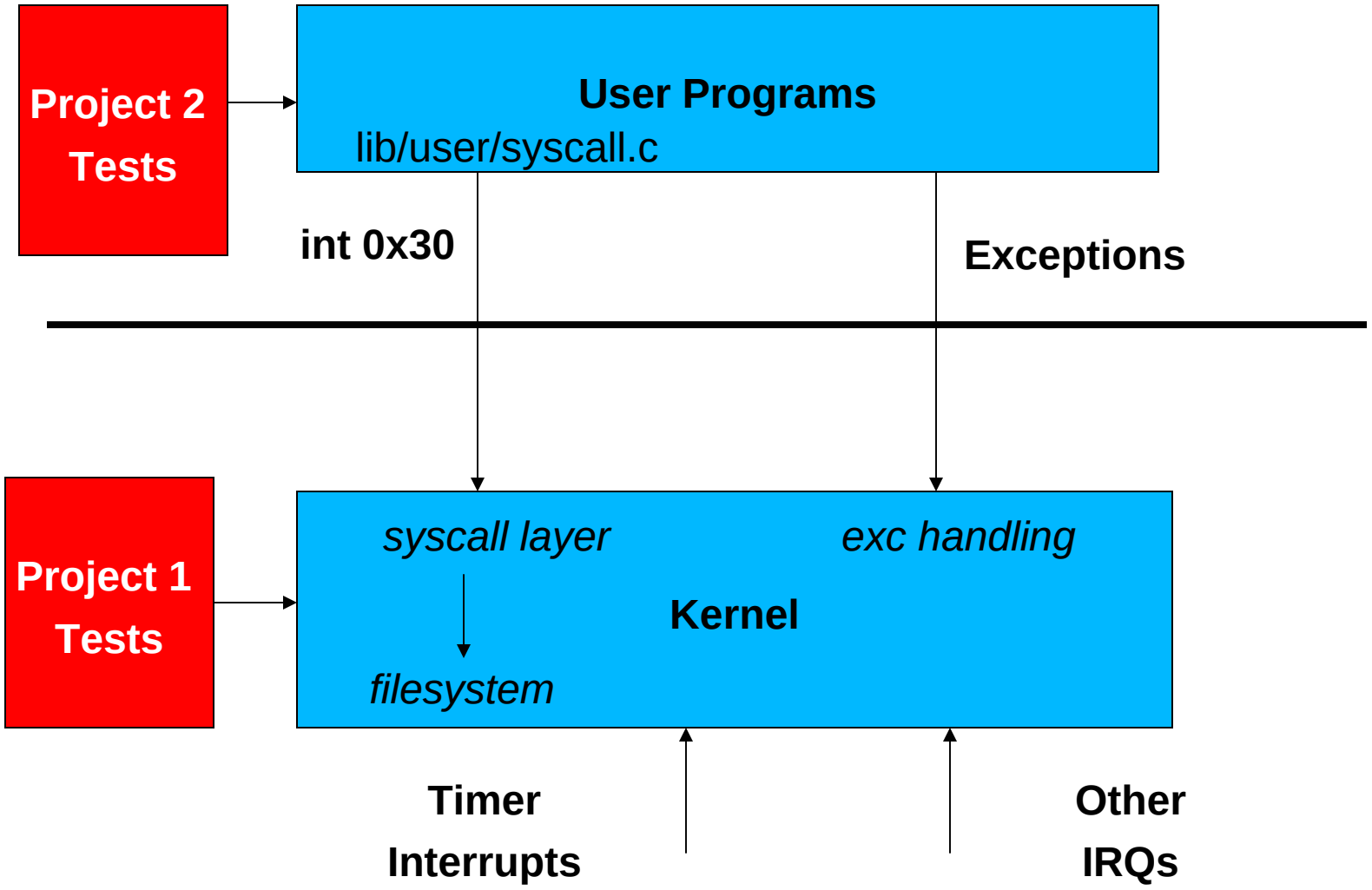
P2: Project 2 – System Calls



Till now…

- All code part of Pintos Kernel
- Code compiled directly with the kernel
  - This required that the tests call some functions whose interface should remain unmodified
- From now on, run user programs on top of kernel
  - Freedom to modify the kernel to make the user programs work

Project 1 and Project 2



When does a process need to access Operating System functionality?

- Here are several examples:
  - Reading a file. The OS must perform the file system operations required to read the data off of disk.
  - Creating a child process. The OS must set stuff up for the child process.
  - Sending a packet out onto the network. The OS typically handles the network interface.

Why have the OS do these things?

- Why doesn't the process just do them directly?
  - **Convenience.** Implement the functionality once in the OS and encapsulate it behind an interface that everyone uses. So, processes just deal with the simple interface, and developers don't have to write complicated low-level code to deal with devices.
  - **Portability.** OS exports a common interface typically available on many hardware platforms. Applications do not contain hardware-specific code.
  - **Protection.** If applications have direct access to disks or network protocol stacks, they can corrupt data from other applications, either maliciously or because of bugs. Having the OS do it eliminates security problems between applications. Applications still have to trust the OS.

How do processes invoke OS functionality?

- By making a system call.
  - Conceptually, processes call a subroutine that goes off and performs the required functionality. But OS must execute in supervisor mode, which allows it to do things like manipulate the disk directly.
  - To switch from normal user mode to supervisor mode, most machines provide a system call instruction.
    - This instruction causes an exception to take place.
    - The hardware switches from user mode to supervisor mode and invokes the exception handler inside the operating system.
  - There is typically some kind of convention that the process uses to interact with the OS.

Let's do an example - Open() system call

- System calls typically start out with a normal subroutine call.

```
int handle = open("sample.txt");
```

- Open() executes a syscall instruction, which generates a system call exception 0x30.

```
syscall1 (SYS_OPEN, file);
```

- By convention, the Open subroutine puts a number on the stack to indicate which routine (SYS_OPEN = 6) should be invoked.
  - Inside the exception handler the OS looks at the stack to figure out what system call it should perform.
- The Open system call also takes a parameter. By convention, the compiler also puts this (e.g., a pointer to the filename) on the stack.
  - More conventions: return values are put into the %EAX register.
- Inside the exception handler, the OS figures out what action to take, performs the action, then returns back to the user program.

SYS_OPEN is defined in lib/syscall-nr.h

```
#ifndef __LIB_SYSCALL_NR_H
#define __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. */
…
```

Thus, SYS_OPEN = 6.

Open( ) system call details

In pintos/src/lib/user/syscall.c:

```
int
open (const char *file)
{
  return syscall1 (SYS_OPEN, file);
}
.. (and above)..

/* 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;                                                            \
    })
```

Initialize syscall handler

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

Add sys_open function to syscall.c

```
+sys_open (const char *ufile)
+{
+  char *kfile = copy_in_string (ufile);
+  struct file_descriptor *fd;
+  int handle = -1;
+
+  fd = malloc (sizeof *fd);
+  if (fd != NULL)
+    {
+      lock_acquire (&fs_lock);
+      fd->file = filesys_open (kfile);
+      if (fd->file != NULL)
+        {
+          struct thread *cur = thread_current ();
+          handle = fd->handle = cur->next_handle++;
+          list_push_front (&cur->fds, &fd->elem);
+        }
+      else
+        free (fd);
+      lock_release (&fs_lock);
+    }
+
+  palloc_free_page (kfile);
+  return handle;
+}
```

Add sys_open to syscall_handler

```
+static void
+syscall_handler (struct intr_frame *f)
+{
+  typedef int syscall_function (int, int, int);
+
+  /* A system call. */
+  struct syscall
+    {
+      size_t arg_cnt;           /* Number of arguments. */
+      syscall_function *func;   /* Implementation. */
+    };
+
+  /* Table of system calls. */
+  static const struct syscall syscall_table[] =
+    {
+      {0, (syscall_function *) sys_halt},
+      {1, (syscall_function *) sys_exit},
+      {1, (syscall_function *) sys_exec},
+      {1, (syscall_function *) sys_wait},
+      {2, (syscall_function *) sys_create},
+      {1, (syscall_function *) sys_remove},
+      {1, (syscall_function *) sys_open},
+      …
+    }
```

Add sys_open to syscall_handler

```
+  const struct syscall *sc;
+  unsigned call_nr;
+  int args[3];
+  /* Get the system call. */
+  copy_in (&call_nr, f->esp, sizeof call_nr);
+  if (call_nr >= sizeof syscall_table / sizeof *syscall_table)
+    thread_exit ();
+  sc = syscall_table + call_nr;

+  /* Get the system call arguments. */
+  ASSERT (sc->arg_cnt <= sizeof args / sizeof *args);
+  memset (args, 0, sizeof args);
+  copy_in (args, (uint32_t *) f->esp + 1, sizeof *args * sc->arg_cnt);
+
+  /* Execute the system call,
+     and set the return value. */
+  f->eax = sc->func (args[0], args[1], args[2]);
+}
```

Interrupts vs. Exceptions

- The difference between interrupts and exceptions is that
  - **interrupts** are generated by external events (the disk IO completes, a new character is typed at the console, etc.), and
  - **exceptions** are generated by a running program.

Using the File system

- Interfacing with the file system
- No need to modify the file system
- Certain limitations
  - No internal synchronization
  - File size fixed
  - File data allocated as a single extent
  - No subdirectories
  - File names limited to 14 chars
  - System crash might corrupt the file system
- Files to take a look at: ‘filesys.h’ & ‘file.h’

Some commands

- Creating a simulated disk
  - pintos-mkdisk fs.dsk 2
- Formatting the disk
  - pintos -f -q
  - This will only work after your disk is created and kernel is built!
- Copying the program "echo" onto the disk
  - pintos -p ../../examples/echo -a echo -- -q
- Running the program
  - pintos -q run ’echo x’
- All in a single command:

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

- $ **make check** – builds the disk automatically
  - You can just copy & paste the commands that make check does!

Various directories

- Few user programs:
  - src/examples
- Relevant files:
  - userprog/
- Other files:
  - threads/

Project 2 Requirements

- Process Termination Messages
- Argument Passing
- System Calls
- Deny writes to executables

1. Process Termination

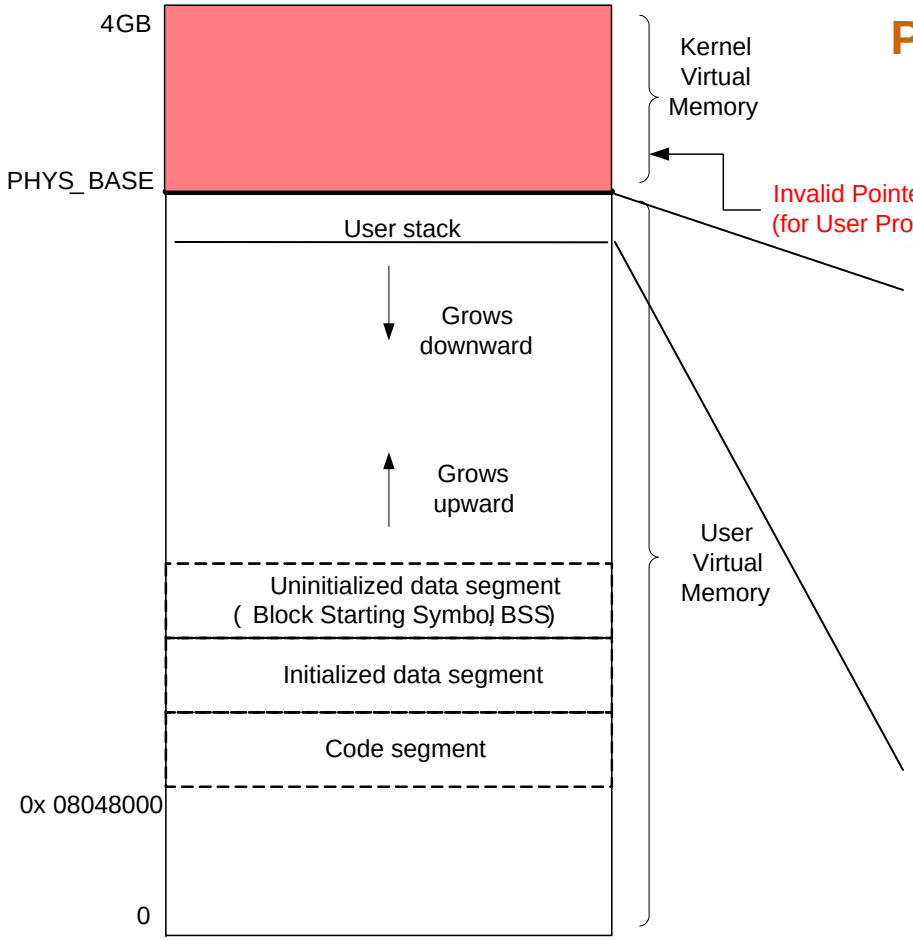
- When a Process Terminates
  - printf ("%s: exit(%d)\n",...);
- Name: Full name passed to process_execute() in process.c
- Exit Code
- Do not print any other message!

2. Argument Passing

- No support currently for argument passing
- Change *esp = PHYS_BASE to *esp = PHYS_BASE – 12 in setup_stack() to get started
- Change process_execute() to process multiple arguments
- Can limit the arguments to fit in a page(4 kb)
- String Parsing: strtok_r() in lib/string.h

```
pgm.c
main(int argc,
        char *argv[]) {
        ...
   }
$ pintos run ‘pgm alpha beta’
argc = 3
argv[0] = “pgm”
argv[1] = “alpha”
argv[2] = “beta”
```

Memory Layout



PHYS_BASE = 3GB

Invalid Pointer Area
(for User Programs)

Address	Name	Data	Type
0xbffffffc	argv[3][...]	'bar\0'	char[4]
0xbffffff8	argv[2][...]	'foo\0'	char[4]
0xbffffff5	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]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

Setting up the Stack

How to setup the stack for the program - /bin/ls –l foo bar

Address	Name	Data	Type
0xbffffffc	argv[3][...]	'bar\0'	char[4]
0xbffffff8	argv[2][...]	'foo\0'	char[4]
0xbffffff5	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]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

3. System Calls – already discussed some; e.g. open( ) system call

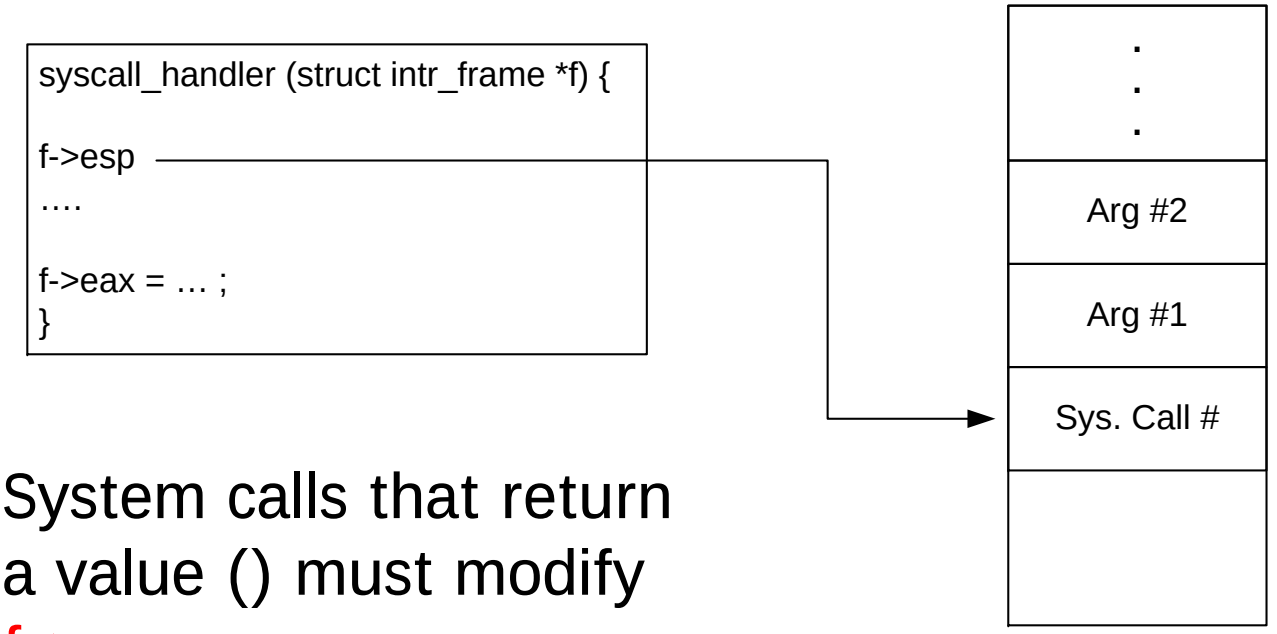
- No Support for system calls currently
- Implement the system call handler in userprog/syscall.c
- System call numbers defined in lib/syscall-nr.h
- Process Control: exit, exec, wait
- File system: create, remove, open, filesize, read, write, seek, tell, close
- Others: halt

System Call Details

- Types of Interrupts – External and Internal
- System calls – Internal Interrupts or Software Exceptions
- 80x86 – ‘int’ instruction to invoke system calls
- Pintos – ‘int $0x30’ to invoke system call

Continued…

- A system call has:
  - System call number
  - (possibly) arguments
- When syscall_handler() gets control:



  - System calls that return a value () must modify f->eax

System calls – File system

- Decide on how to implement the file descriptors – keep it simple -- O(n) data structures will entail no deduction
- Access granularity is the entire file system – add one global lock
- write() – fd 1 writes to console – use putbuf() to write entire buffer to console
- read() – fd 0 reads from console – use input_getc() to get input from keyboard
- Implement the rest of the system calls

System calls – Process Control

- wait(pid) – Waits for process pid to die and returns the status pid returned from exit
- Returns -1 if
  - pid was terminated by the kernel
  - pid does not refer to child of the calling thread
  - wait() has already been called for the given pid
- exec(cmd) – runs the executable whose name is given in command line
  - returns -1 if the program cannot be loaded
- exit(status) – terminates the current user program, returns status
  - status of 0 indicates success, non zero otherwise

Process Control – continued…

- Parent may or may not wait for its child
- Parent may call wait() after child terminates!
- Implement process_wait() in process.c
- Then, implement wait() in terms of process_wait()
- Cond variables and/or semaphores will help
  - Think about what semaphores may be used for and how they must be initialized

```
main() {
int i; pid_t p;
p = exec("pgm a b");
i = wait (p);
... /* i must be 5 */
}
```

```
main() {
int status;
... status = 5;
exit(status);
}
```

pgm.c

Memory Access

- Invalid pointers must be rejected. Why?
  - Kernel has access to all of physical memory including that of other processes
  - Kernel like user process would fault when it tries to access unmapped addresses
- User process cannot access kernel virtual memory
- User Process after it has entered the kernel can access kernel virtual memory and user virtual memory
- How to handle invalid memory access?

Memory Access – contd…

- Two methods to handle invalid memory access
  - Verify the validity of user provided pointer and then dereference it
    - Look at functions in userprog/pagedir.c, threads/vaddr.h
    - Strongly recommended!
  - Check if user pointer is below PHYS_BASE and dereference it
    - Could cause page fault
    - Handle the page fault by modifying the page_fault() code in userprog/exception.c
  - Make sure that resources are not leaked

Some Issues to look at…

- Check the validity of the system call parameters
- Every single location should be checked for validity before accessing it; e.g. not only for f->esp, but also the locations f->esp +1, f->esp+2 and f->esp+3 should be checked
- Read system call parameters into kernel memory (except for long buffers) – write a copy_in function for this purpose.

Denying writes to Executables

- Use file_deny_write() to prevent writes to an open file
- Use file_allow_write() to re-enable write
- Closing a file will re-enable writes

Suggested Order of Implementation

- Change *esp = PHYS_BASE to *esp = PHYS_BASE – 12 to get started
- Implement the system call infrastructure
- Change process_wait() to a infinite loop to prevent pintos getting powered off before the process gets executed
- Implement exit system call
- Implement write system call
- Start making other changes

Pintos Project 2 Sample Test

- Example Open System Call
- Test: tests/userprog/open-normal.c

```
/* Open a file. */
#include <syscall.h>
#include "tests/lib.h"
#include "tests/main.h"

void
test_main (void)
{
  int handle = open ("sample.txt");
  if (handle < 2)
    fail ("open() returned %d", handle);
}
```

userprog/ - make check

```
gcc -c ../../tests/userprog/open-normal.c -o tests/userprog/open-normal.o
 -g -msoft-float -O -fno-stack-protector -nostdinc -I../.. -I../../lib
 -I../../lib/user -I. -Wall -W -Wstrict-prototypes -Wmissing-prototypes
 -Wsystem-headers  -MMD -MF tests/userprog/open-normal.d
gcc  -Wl,--build-id=none -nostdlib -static -Wl,-T,../../lib/user/user.lds
 tests/userprog/open-normal.o tests/main.o tests/lib.o lib/user/entry.o
 libc.a -o tests/userprog/open-normal
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/open-normal
 -a open-normal -p ../../tests/userprog/sample.txt -a sample.txt -- -q
 -f run open-normal < /dev/null 2> tests/userprog/open-normal.errors
 > tests/userprog/open-normal.output
perl -I../.. ../../tests/userprog/open-normal.ck
 tests/userprog/open-normal tests/userprog/open-normal.result
pass tests/userprog/open-normal
----------------------------------------------------------------------
ar r libc.a lib/debug.o lib/random.o lib/stdio.o lib/stdlib.o lib/string.o
 lib/arithmetic.o lib/ustar.o lib/user/debug.o lib/user/syscall.o
 lib/user/console.o
ranlib libc.a
```

$ objdump -D open-normal

```
080480a0 <test_main>:
 80480a0:       55                      push   %ebp
 80480a1:       89 e5                   mov    %esp,%ebp
 80480a3:       83 ec 18                sub    $0x18,%esp
 80480a6:       c7 04 24 6a a7 04 08    movl   $0x804a76a,(%esp)
 80480ad:       e8 1f 21 00 00          call   804a1d1 <open>
 80480b2:       83 f8 01                cmp    $0x1,%eax
..
0804a1d1 <open>:
 804a1d1:       55                      push   %ebp
 804a1d2:       89 e5                   mov    %esp,%ebp
 804a1d4:       ff 75 08                pushl  0x8(%ebp)
 804a1d7:       6a 06                   push   $0x6
 804a1d9:       cd 30                   int    $0x30
 804a1db:       83 c4 08                add    $0x8,%esp
 804a1de:       5d                      pop    %ebp
 804a1df:       c3                      ret
..
0804a76a <.rodata.str1.1>:
 804a76a:       73 61 6d 70 6c 65 2e 74 78 74 00
                 s  a  m  p  l  e  .  t  x  t
```

src/userprog/syscall.c

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

src/userprog/syscall.c

```
/* System call handler. */
static void
syscall_handler (struct intr_frame *f)
{
  typedef int syscall_function (int, int, int);
  /* A system call. */
  struct syscall
    {
      size_t arg_cnt;           /* Number of arguments. */
      syscall_function *func;   /* Implementation. */
    };
  /* Table of system calls. */
  static const struct syscall syscall_table[] =
    {
      {0, (syscall_function *) sys_halt},
      {1, (syscall_function *) sys_exit},
      {1, (syscall_function *) sys_exec},
      {1, (syscall_function *) sys_wait},
      {2, (syscall_function *) sys_create},
      {1, (syscall_function *) sys_remove},
      {1, (syscall_function *) sys_open},  <-- call number 6
```

src/userprog/syscall.c

```
 ...
  const struct syscall *sc;
  unsigned call_nr;
  int args[3];

  /* Get the system call. */
  copy_in (&call_nr, f->esp, sizeof call_nr);
  if (call_nr >= sizeof syscall_table / sizeof *syscall_table)
    thread_exit ();
  sc = syscall_table + call_nr;

  /* Get the system call arguments. */
  ASSERT (sc->arg_cnt <= sizeof args / sizeof *args);
  memset (args, 0, sizeof args);
  copy_in (args, (uint32_t *) f->esp + 1, sizeof *args * sc->arg_cnt);

  /* Execute the system call,
     and set the return value. */
  f->eax = sc->func (args[0], args[1], args[2]);
}
```

src/userprog/syscall.c

```
/* Open system call. */
static int
sys_open (const char *ufile)
{
  char *kfile = copy_in_string (ufile);
  struct file_descriptor *fd;
  int handle = -1;

  fd = malloc (sizeof *fd);
  if (fd != NULL)
    {
      lock_acquire (&fs_lock);
      fd->file = filesys_open (kfile);
      if (fd->file != NULL)
...
```