CS 5600

Computer Systems

Project 2: User Programs in Pintos

User Programs in Pintos

- Pintos already implements a basic program loader
  - Can parse ELF executables and start them as a process with one thread
- Loaded programs can be executed
- But this system has problems:
  - User processes crash immediately :(
  - System calls have not been implemented

Your Goals

1. Implement argument passing
   - Example: “ls” sort of works
   - … but “ls –l –a” doesn’t work
   - You must pass argv and argc to user programs
2. Implement the Pintos system APIs
   - Process management: exec(), wait(), exit()
   - OS shutdown: halt()
   - File I/O: open(), read(), write(), close()
     - Can be used for writing to the screen (write stdout)
     - … and reading from the keyboard (read stdin)

Formatting the File System

- In this project, you will be running user programs within Pintos
- Thus, you must format a file system to store these user programs on

Total size of the file system, in MB

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

Copy the ‘echo’ program to the Pintos file system

Format the file system

Program Loading

- userprog/process.c contains the code for loading ELF files

```
/* Executable header. This appears at the very beginning of an ELF binary. */
struct Elf32_Ehdr { … }

/* Program header. There are e_phnum of these, starting at file offset e_phoff. */
struct Elf32_Phdr { … }

/* Loads an ELF executable from FILE_NAME into the current thread.
   Stores the executable's entry point into *EIP
   and its initial stack pointer into *ESP.
   Returns true if successful, false otherwise. */
bool load (const char *file_name, void (**eip) (void), void **esp) { … }
```

Setting Up The Stack

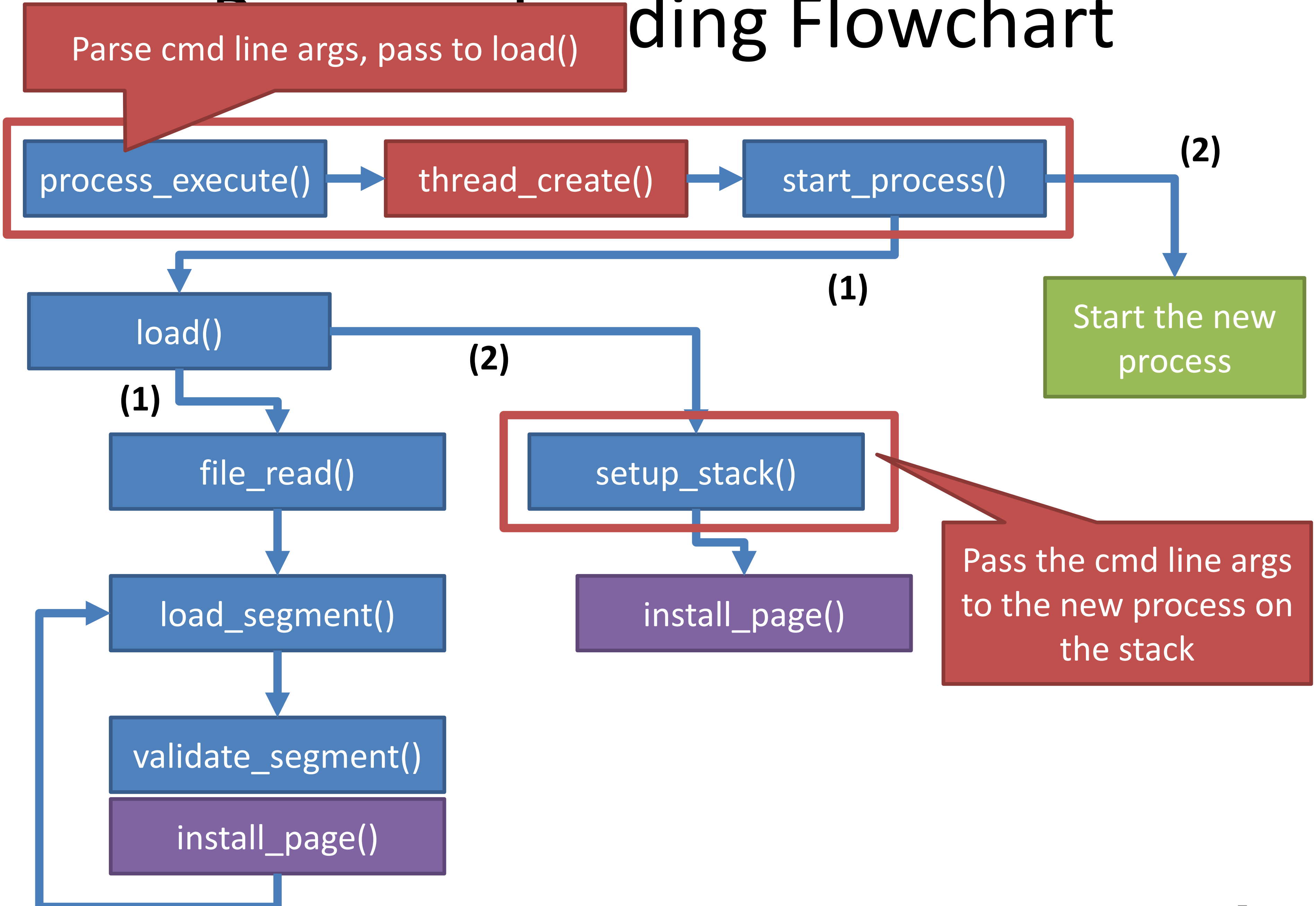
- userprog/process.c

```
/* Create a minimal stack by mapping a zeroed page at the top of user virtual
memory. */
static bool setup_stack (void **esp) {
  uint8_t *kpage;
  bool success = false;

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL) {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success) *esp = PHYS_BASE;
      else palloc_free_page (kpage);
  }
  return success;
}
```

At a minimum, you will need to place argc and *argv on the initial stack, since they are parameters to main()

ding Flowchart



Syscalls in Pintos

- Pintos uses int 0x30 for system calls
- Pintos has code for dispatching syscalls from user programs
  - i.e. user processes will push parameters onto the stack and execute int 0x30
- In the kernel, Pintos will handle int 0x30 by calling syscall_handler() in userprog/syscall.c

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

Syscalls from the user process

- lib/user/syscall.h
  - Defines all the syscalls that user programs can use
- lib/user/syscall.c

```
void halt (void) {
    syscall0 (SYS_HALT);
}

void exit (int status) {
    syscall1 (SYS_EXIT, status);
}
```

```
pid_t exec (const char *file) {
    return (pid_t) syscall1 (SYS_EXEC, file);
}
```

These are syscalls. They are implemented in the kernel, not in userland.

Using int 0x30 to Enter the Kernel

- 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;                                               \
    })
```

On the Kernel Side…

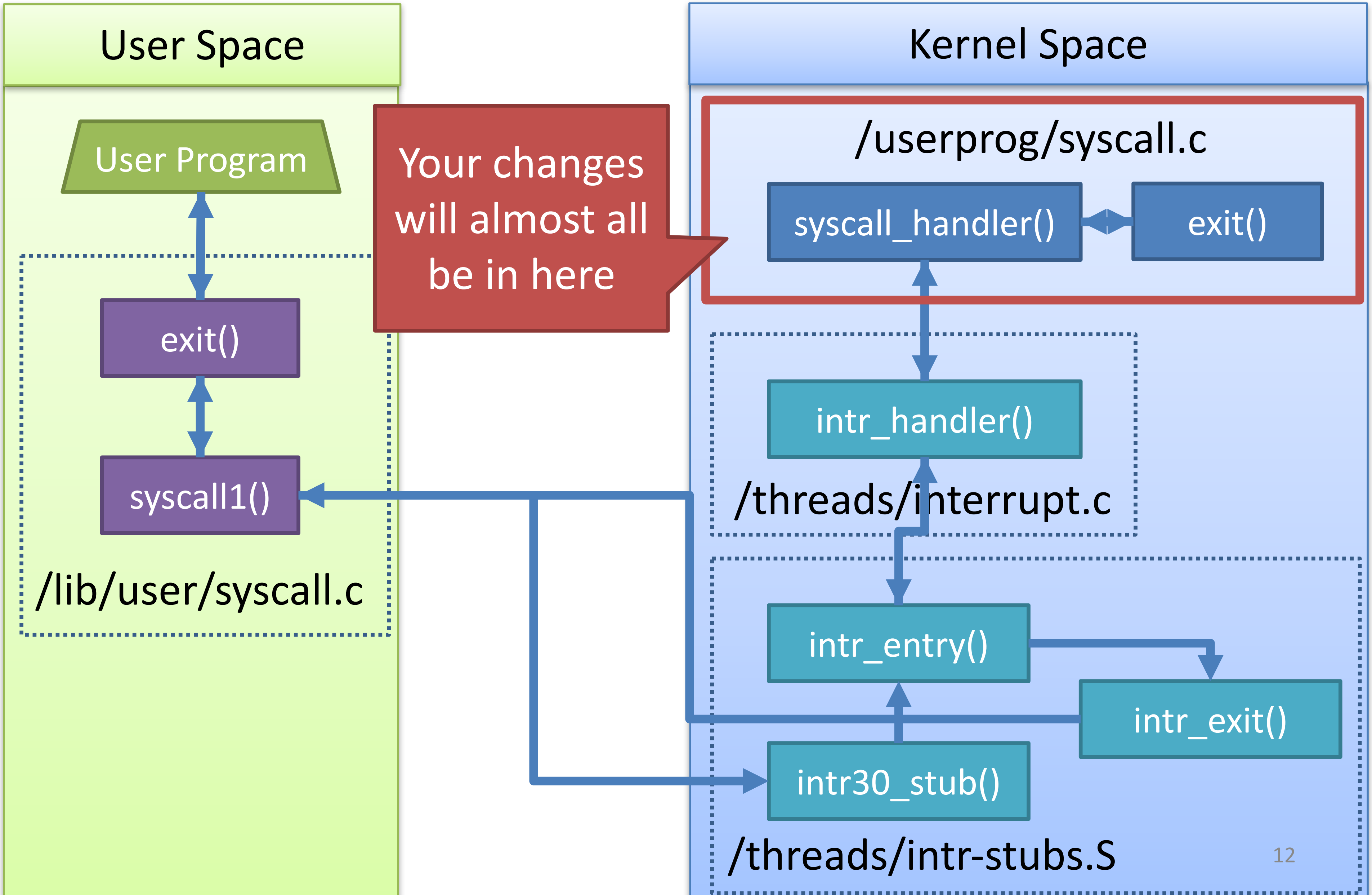
- userprog/syscall.c

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

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

Called during main(), sets syscall_handler() to be run whenever int 0x30 is received

Example Syscall Flowchart (exit)



Other Things Pintos Gives You

- Basic virtual memory management
  - User processes live in virtual memory, cannot access the kernel directly
  - Kernel may access all memory
  - You will enhance this in Project 3
- Trivial filesystem implementation
  - Can store user programs
  - You will enhance this in Project 4

Key Challenges

- Having the kernel read/write memory in user processes
  - Necessary for reading API parameters from the user stack
    - E.g. a string passed via a pointer
  - Need to understand the virtual memory system
- Handling concurrent processes
  - Remember, processes can call exec()
- Handling file descriptors and standard I/O

Modified Files

- threads/thread.c 13
- threads/thread.h 26
- userprog/exception.c 8
- userprog/process.c 247
- userprog/syscall.c 468
- userprog/syscall.h 1
- 6 files changed, 725 insertions(+), 38 deletions(-)

Setting up argv and argc

Implementing syscalls