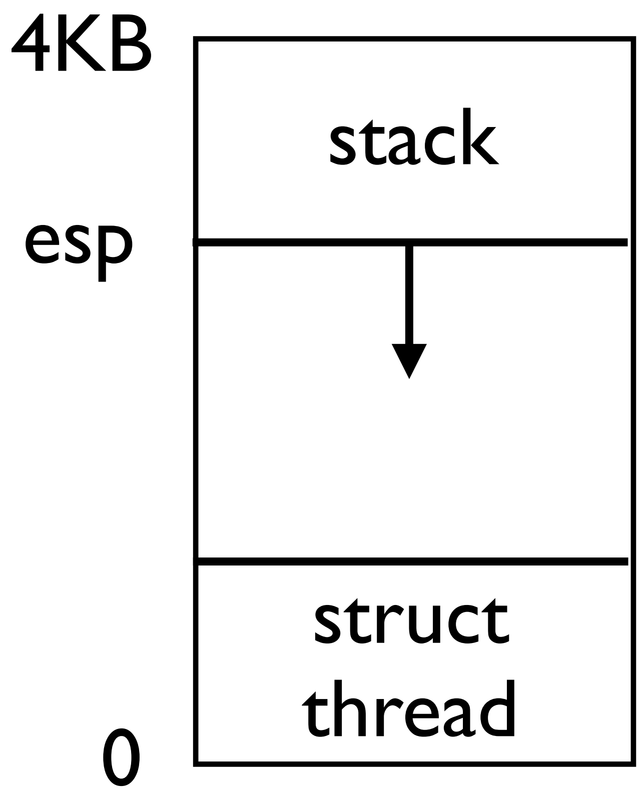
Pintos User Programs

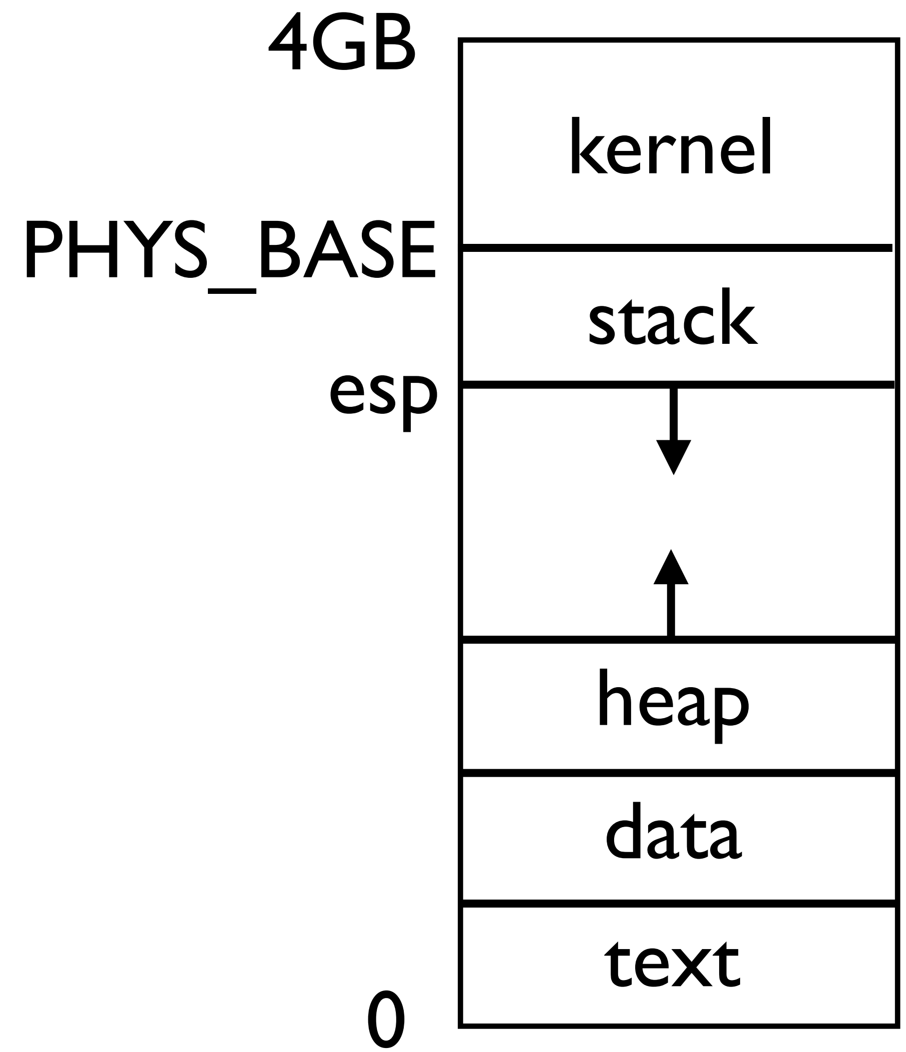
User Processes

- Userprog kernel (`pintos/userprog/`)
  - Loads programs from disk
  - Processes are created, scheduled and managed by a kernel thread
  - Each process maps to a `struct thread`
  - Unlike threads, user memory is not shared

Processes vs Threads



Thread



User process

Running user programs

- Compile programs in `pintos/examples/`

```
$ cd pintos/examples
$ make
```

- Create a filesystem

```
$ cd userprog/build
$ pintos-mkdisk filesys.dsk --filesys-size=2      # create filesys
$ pintos -q -f                                    # format
$ pintos -p ../../examples/echo -a echo -- -q     # copy echo to filesys
```

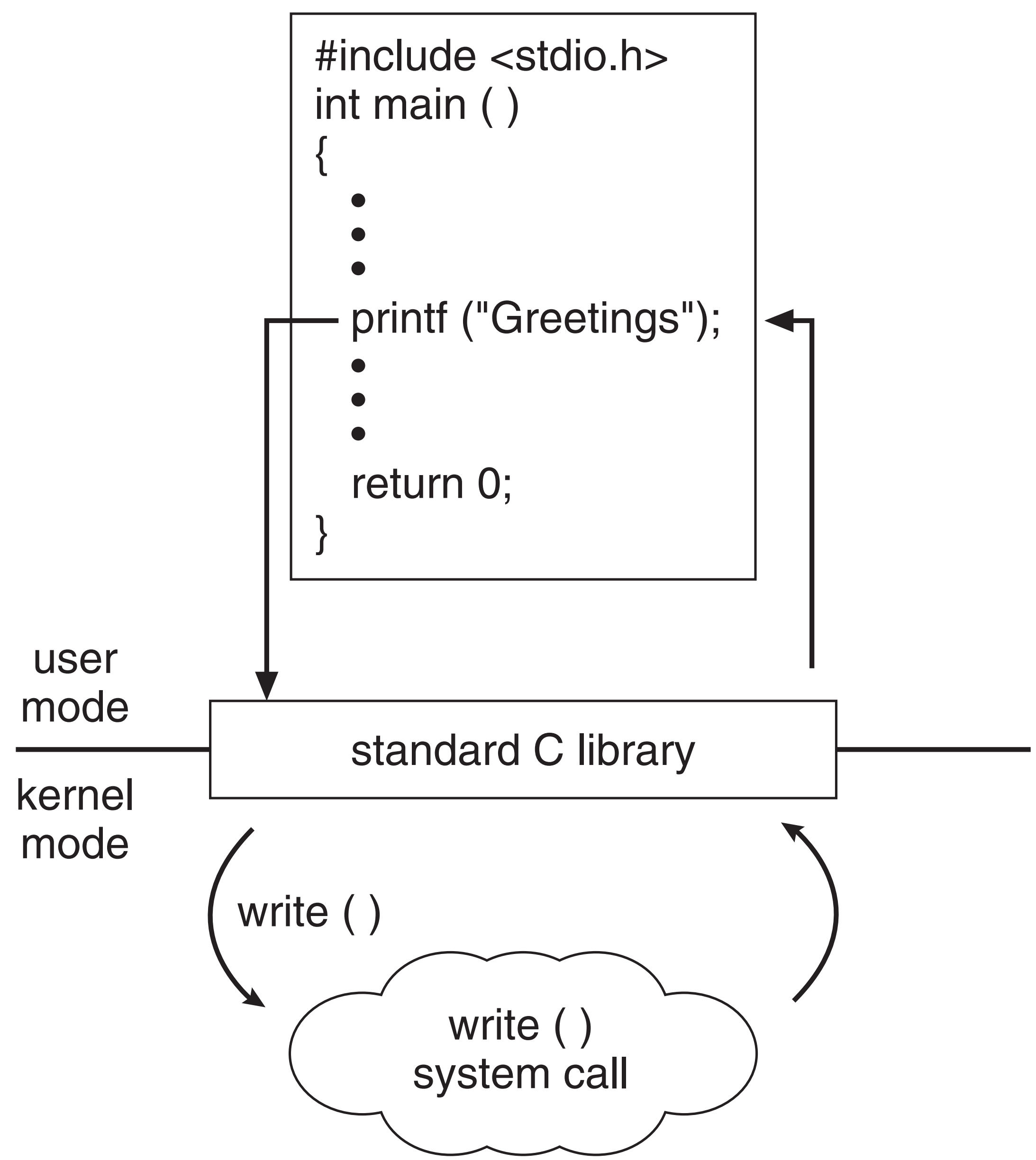
- Run as usual

```
$ pintos run 'echo x'
```

Assignment 2

- Essential features missing
  - System call handler
  - Argument passing
  - Function `process_wait()`

System call handler



System call handler

- Pintos passes arguments on the stack
- In `lib/user/syscall.c`

```
int
write (int fd, const void *buffer, unsigned size)
{
  return syscall3 (SYS_WRITE, fd, buffer, size);
}
```

- Macro `syscall3` pushes the arguments on the stack and generates an interrupt

System call handler

- Interrupt handled in `userprog/syscall.c`

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

- For assignment 2
  - handle `printf()` and `exit()`

Hints

- System call numbers defined in `lib/syscall-nr.h`
- Syscall handler has access to registers
  - Stack pointer is `f->esp`
  - Save the return value to `f->eax`
- Use function `putbuf()` to print to stdout

Argument Passing

- New processes are created by function `process_execute()`
  - Creates a new thread process
  - Loads program from filesystem
  - Executes function `main` passing `argc` and `argv`
- Equivalent to Unix `fork` + `exec`

Argument Passing

- Right now, kernel is not passing the arguments to the executable
- Should be passed on the user stack
- Example: to call `f(1, 2, 3)`

```
                              +----------------+
                 0xbffffe7c |        3       |
                 0xbffffe78 |        2       |
                 0xbffffe74 |        1       |
stack pointer --> 0xbffffe70 | return address |
                              +----------------+
```

Argument Passing

- Example: `/bin/ls -l foo bar`

PHYS_BASE = 0xc0000000
PHYS_BASE - 4 = 0xbffffffc

optional

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 (*) ()

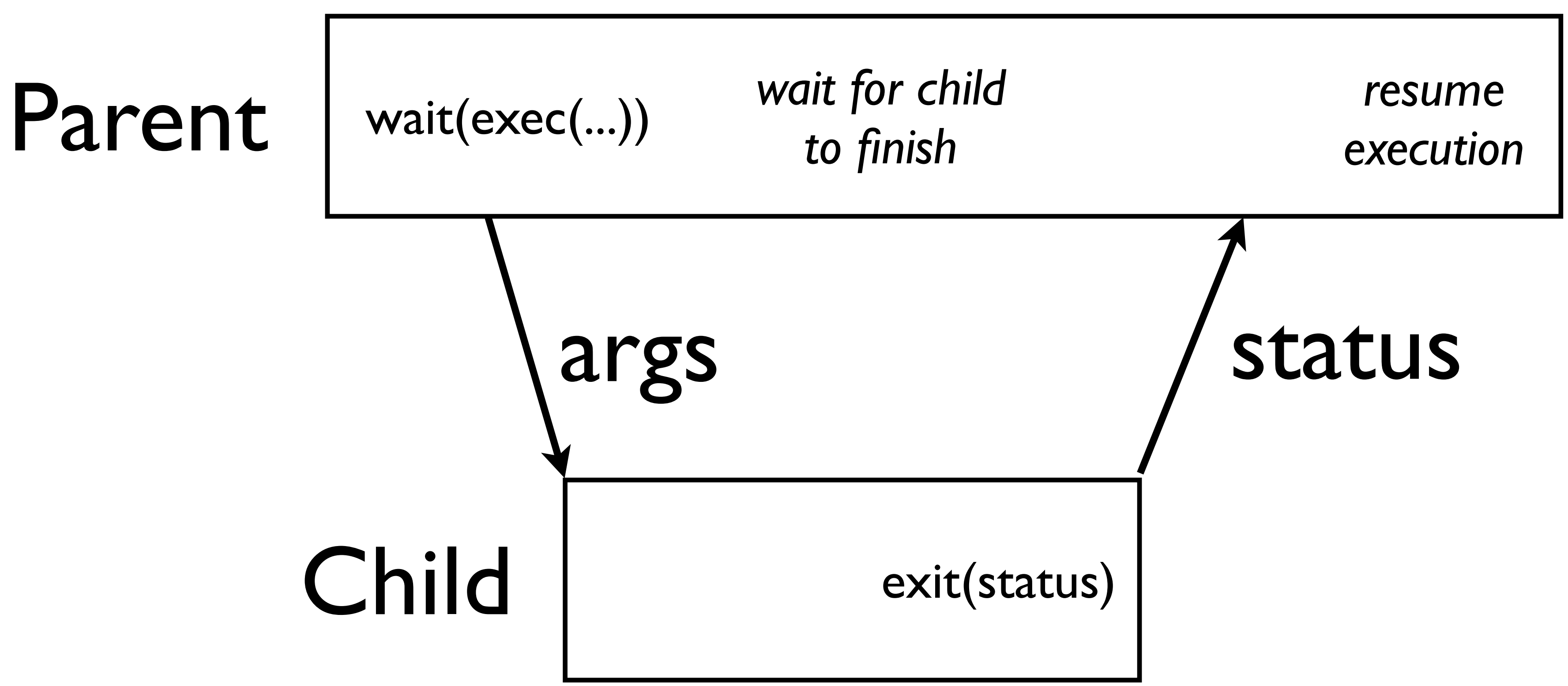
Hints

- Look at functions `process_execute` and `start_process` in `userprog/process.c`
- Use function `strtok_r()` to tokenize the command line
- Remember: the stack grows downwards!

Process wait

- You have to implement function
  `int process_wait(tid_t child)`
- Calling process/thread blocked until child exits

Process wait



Process wait

- Used in pintos when starting a program

```
/* 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);
}
```

Hints

- Can be implemented as follows
  - child keeps track of parent thread
  - use `thread_block` to block the parent when `process_wait` is called
  - when child exits, `thread_unblock` parent thread

Tests

- We expect first 5 tests to pass
  - args-none
  - args-single
  - args-multiple
  - args-many
  - args-dbl-space

Readings

- Chapter 3
- You can skip sections 3.1.5 and 3.3.5