Project 2 - Final Report (Team 28)

□□□□ 2□ □□□□ □□ □□□ □□, □□ □□, □□□ □□ □□, □□ □□□ □□ □□ □□□ □ □□ Task□ □□□□□ □□. □□ □□ □ □□□□ □ □ □□□ Pintos □□□ □□□ □□□□ □□□□ □□□□□□□□, □□ □□□ □□□ □□□ □□□□□. □□□, □ □□□□□□ □□ □□□ □□□□□ □□□ □ □□ □□□□□ □□□ □□□□□ □□ □□□□□ □□□□ □□□ □□□ □□ □□□, □□ □□□ □□□ □□□ □□ □□□□ □□ □□□.

Process Data Structures and Implementations

Improvements

□□□□□ □□□ □□ □□□ □□ □□□ □□□, □□□□ □□□ □ □□□□. □□, □□□□□ □□□ □□ □□□ □□□□ □□□□□□ □□□ □□□□□, □□ □□□ □ □□□□□ □□□□□ □□□□ □□□ □□□ □□□□□ □ □□ □□. □□□, □□□□□ □□□□ □□□□ □□□□ ELF(Executable and Linkable File format) □□□ □□□□ □□□□□ □□□□ □□□□ □□□, □ □□□□□□ □□□ □□ □□□ □□□□ □□□ □□□□□ □□.

Pintos□ □□ □□□ □ □□□ □□ □□ □□□□□, □□□□□ □□□□□ □□□□ □□□, `process_execute()`, `process_wait()`, `process_exit()` □□ □□□□ □□□ □□ □□□ □□□□ □□ □□□□. □□□, □□ □□□□□□□ □□□ □□□ □ □□□□ □□ □□ □□□□□ □□□□□ □□□□ □□□□ □□.

□□□ □□□□□ □□□□ □□□□ □ □□ □ □□ □□□, Pintos□ □□ □□□ □□□ □□□□□ □□ □□□□ □□□□ □□□□ □□□. `thread.h` □ □□□ □□□ `struct thread`□ □□□ □□□□ □□□ □□.

```
/* From former version of threads/thread.h. */
struct thread
  {
    ...

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    ...
  };
```

`pagedir`□ □□□ □□ □□□ □□□□, □□ □□□ □□□ □□□ □□□□ □□□□ □□□□□. Pintos□ □□ □□□□□, `pagedir`□ `struct thread`□ □□□ □□□□ □□. □□ □□, Pintos□ □□ □□□ □□ □□□□ □□□ □□ □□□ □□□ □□□ □□□□□ □□□□ □□ □. □□ □□ `process_wait()`□ `process_exit()` □□ □□□ □□□ □ □ □□ □□□ □□□□□□, □□□□□□ □□ □□□□ □□ □ □□ □□□ □□□□□ □□□□ □□□□ □□ □□ □□□ □□□ □□□□□ □□□□ `struct process`□ □□□ □□□□□ □□□□□. `struct process`□ □□□□□ PCB(Process Control Block)□□ □□□ □□□□□, □□□ □□□ □□ □□□□ □□ □□□.

Details and Rationales

`struct process`

```
/* From userprog/process.h. */
/* An user process. */
```

```
struct process
  {
    /* Owned by process.c. */
    pid_t pid;                        /* Process identifier. */
    int status;                       /* Exit status. */
    bool success;                     /* Has execution of its child succeeded?
*/
    bool waited;                      /* Is its parent waiting on this? */
    char name[16];                    /* Name. */
    enum process_state state;         /* Process state. */
    uint32_t *pagedir;                /* Page directory. */
    struct semaphore sema;            /* Semaphore to order parent and child.
*/
    struct thread *thread;            /* Thread of this process.*/
    struct thread *parent;            /* Parent thread. */
    struct list children;             /* List of child processes. */
    struct list_elem elem;            /* List element. */

    /* Shared between syscall.c and process.c.*/
    struct list opened;               /* List of opened files. */
  };
```

□□□ □□ □□ `struct process`□ □□□ □□□□□ □□□□ □□ `struct thread`□□ □□□ □□□□□□. □□□ □□□□□ □□□, □ □□□□ □□□□□ □□□□ □□ □□□□ □□□ □□□□, □□□, Pintos□ □□□□ □□ □ □□□ □□□□ □□□□ □□ □□□ □□□□□□ □□ □□ □ □□□□ □□ □□ □□□ □□□□□□ □□□□ □□□ □□□. □□□□□, `process_exit()`□ `process_wait()`□ □□□ □, `struct thread`□ □□□□□ □□ □□ □□ □□ □□ □□□□ `thread_exit()` □□ □, □□ □□□□□ □□□□□ □□ □□ □□□ □□ □□□ □□□□ □□□ □□ □□□ □□□□.

□ □□□ □□ □ □□□ □□□□. `process_exit()`□ □□ □□□□□ □□□□ □□□, □□□□ □□□ □□ □□ □□(exit status)□ □□ □ □□□□ □□□□□ □□. □ □□ □□□□□□ □□□ `struct thread`□ □□ □□□ □□□□□, □□ □□□□ □□ □□□□ □□ □□□□ □□□ □□ □□ □□□ □□□□ □ □□□ □□ □□□□. □□□, □□ □□□ □□□□ □□ □□□□ □□ □□□ □□□ □□ □□ □ □ □□□□□, (□□□ □□□ □□□□ □□□□ □□ □□□□□ □□ □□.) □□ □□□□ □□□□ □□ 4KiB □□□ □□□□ □□□□ □□□□□ □□□ □□ □□□ □□□□□.

□□□□ □□□ □□□, □□ □□□□□ `struct thread`□ □□ □□□ □□ □□□ □□□□ □□ □□□, □□ □□□□□ `struct thread`□ □□ □□□□□ □□ □□□ □□□□ □□□ □□. □ □□□ □□□□ □□ □□□□□ □□□□ □ □, □□ □□□□□ □□ □□□□□ □□ □□□ □□□□ □□□ □□□(atomic)□□ □□□ □□□. □□ □□, □□ □□□□ A□ □□□ □, □□ □□□□□ A□ □□ □□□ □□□□ □□ □□ □□□□□ B□ □□□□ □ □ □□□□□ `struct thread`□ □□□ □□ □□□ □□□□□, `process_wait()`□ □□□□ □□ □□□ □□ □□□. □ □□□□□ □□ □ □□□□ □□ □□□□□ □□ □□□□ □□□□ □□□□ □□□□ □□□ □□ □□□ □□□ □□□□ □□□ □□□□□ □□□ □□ □□□□ □□□□□ □□ □□ □ □□(synchronization method)□ □□□□ □□.

□□ □□ □□□□□ □□□□ □□□, □□□□ □□□ □□□□ □□□□□ □□ □□□ `process_exit()`□ `process_wait()` □□ □□ □□ □ □□□□□ □□□□ □□□□.

`struct process`□ □□□ □□□ □□□□. `pid`□ □□□□□□ □□□ □□□ □□□, `name`□ □□□□□ □□□□ □□□ `make_process()`□ □□□ □□□ □□□ □□□ `tid`□ □□□ □□ □□□□ □□□ □□ □□ □□□□ □□□□ □□. Pintos□ □□□ □□□□□ □□□ □□□□□ □□□ □ □□□□, □□□ □□□ □□ □□□ □□□ □□□□□.

`status`□ □□ □□□ □□□□. `struct process`□ □□ □□□ □□□□ □□□□ □□ □□□ □□□□ □□□, □□ □□□□□ □□ □□□□ □ □□ □□ □□□□.

waited□ □□ □□□□□ □□ □□ □□□□□ □□□□ □□□□ □□□□ □□□□□. □□□ □□□ □□□□ □□□ □□□ □□□□ □□□ □□□□□ □□ □□ □□□ □□□□□, □□ □□ □□ □□□ □□ □□□□□ □□□□□□ □□□ □□ □□□□□, □□ □□□□ □□ □□□□□ □□ □□□□ □□□ □□□□□, □□ □□□□□ □□□ □ □ □□□ □□ □□□ □□□□ □□ □□□□. □□□ struct process□ □□ □□□□□ □□ □□□□□ □□□□ □□ □□ □□□□ □□□□ □□□ □□□ □□, process_wait() □□ □ □□ □□□□□ waited□ □□□□□ □□□□ □□.

state□ □□□□□ □□ □□□ □□□□ □□□□. □□ □□□□ □□□□□ process_exit()□ □□ □□□□□□, □□ □□□□□ □□ □□□□ □□ □□ □□□ struct process□ □□(destory)□□ □□, □□ □□□□□ □□ □□□□□ □□□□ □□□ □□□ □□ □□. □□, □□ □□ □□□ □□ □□ □□□□□ □□□□□ □□□, □□□ process_exit()□ □□ □□□ □□□□□□ □□□□□ □□. □□□, □□□ □□□□□ □□ □ □□ □□□□ □□□ struct process□ □□□□ □□.

pagedir□ □□□□ □□□□ □□□□□ □□□ □□□□□ □□□□ □□□□□. □□□□□ □□□ □□ □□□ □□□□, □□□ □□□□□ □□ □□ □□□ □ □□. □□□ Pintos □□□ struct thread□ □□ □□□ □□□□□□, □□ □□□□ □□□ □□ □□□ □□ □□□ □□ □□□ □□ □□□□ □ □□□□. □□□, □□□ □□□ □□□ □□□□□□ □□□ □□ □□□ □□□□, □□□ □□□ □□□□□ □□□□ □□ □□ □□□□ □□□□ PHYS_BASE □□ □□□ □□□ □□ □□□ □□ □□□□ init_page_dir□ □□□□□□ □□□□ □□.

sema□ process_exec(), process_exit(), process_wait() □□□□□ □□□ □□ □□□ □□□□ □□ □□□□□□. thread□ □□□ □□ □□□□ □□□□ □□□□□, children□ □□ □□□□□ □□□, elem□ □□ □□□□□ □□□□ □□□□ □□ □□□ □ □□□.

parent□ □□ □□□ □□ □□ □□□□□ □□□□ □□□□□, □□ □□ □□□□□ □□□□ □□□ □□□ □□□ □□□□□ □□□□ □□ □□ □□□□ □□ □□□□ struct thread * □□□ □□□□ □□.

opened□ □□ □□□□□ □□ □□□□ □□□ □□□□□. process_exit() □□ opened□ □□□ □□□□ □□□□ file_close() □□ □□□□ □□□ □□. □□□□□ □□ □□ □□ opened□ □□□□ □□□□ □□□ □□ □□ □□□ □□.

process_init()

```
/* From userprog/process.c. */
/* Initialize top level process. This should be called after
thread_init(). */
void
process_init (void)
{
  make_process (NULL, thread_current ());
}
```

Pintos□ □□□ □□ □ □□□□ □□□ □□□□ □□□ □□□ □□□□□ □□□□□ □□□□. □□ □□ □□ □□□□ □□□□□□ □□□□ □□□, Pintos□ □□ □□ □□□□ □□□ □□□□ □□□ □□□□□ □□□□ □□□ □□□ □ □ □□□□. □□□, □□□ □□ □□□ □□ □□ □□ □□□ □□□ □□ □□.

```
/* From userprog/process.c. */
/* Starts a new thread running a user program loaded from
   CMD_LINE.  The new thread may be scheduled (and may even exit)
   before process_execute() returns.  Returns the new process's
   process id, or PID_ERROR if the thread cannot be created.
   This must be executed in user process's context. */
pid_t
process_execute (const char *cmd_line)
```

```
{
  ...
  struct process *this = thread_current ()->process;
  struct process_exec_frame frame;
  bool success = false;
  ...

  ASSERT (this != NULL);

  ...

  /* Tokenize the copy of CMD_LINE,
     and stores each address of tokenized string in ARGV. */
  argv = palloc_get_page (PAL_ZERO);
  if (argv == NULL)
    return PID_ERROR;
  for (token = strtok_r (cmd_line_copy, " ", &pos); token != NULL;
       token = strtok_r (NULL, " ", &pos))
    argv[i++] = token;

  /* Create a new thread to be executed with ARGV. */
  frame.argv = argv;
  frame.parent = this;
  frame.success = &success;

  tid = thread_create (argv[0], PRI_DEFAULT, start_process, &frame);

  ...

  sema_down (&this->sema);
  if (!success)
    return PID_ERROR;

  return (pid_t) tid;
}
```

□□□ `cmd_line`□ □□ □□□□ □□□□□ □□□□□ □□□ □□□□ □□□, `exec()` □□□ □□□ □□□ □□□□ □□□□. □□□ □□□ □□□ □□ □□ □□ □□(argument vector)□ □□□□ □□□ □□□□ □□ □□□ `argv`□ □□□□ □□□ □□□□□. □□□ □□□ □□ □□□ □□ □□□□□□ □□□□ `start_process`□□ □□□ □□ □□□□□ □□□□□ □□□□. □□, □□ □□□□□ □□□ □□ □□□□ □□□□ `thread_create()`□ □□□□ □□□ □□□ □□□ □ □ □□□, □□ □□□□ □□□ □□□ □□□□ □□ □□□ □□□□ □□□ □□□ `argv[0]`□ □□□□□□ □□□□□, `start_process`□ □□□ □□□ □□ □□□ □□□ `fn_copy`□ □□ `frame`□ □□□□ □□ □ □ □□.

□□, □□□ □□□□ □□□□ □□ □□□ □□□□ □□□□ □□ □□ □□□ □□ □□□□□ □□. □□□ □□□ `start_process()`□□ □□□□.

```
/* From userprog/process.c. */
/* Frame needed to execute a process from command line. */
struct process_exec_frame
  {
    char **argv;
    struct process *parent;
    bool *success;
  };
```

`frame`□ `struct process_exec_frame` □□ □□□, `struct process_exec_frame`□ □□□ □□□□□□ □□□□ □ □ □□□ □□□□□ □□□□□□. Pintos□ □□□ □□ □□□ □□□ □□ □ □□□□□ □□□ □□□ 1□□ □□□□□ □□, □□ □□□ □□□□□ □□□□□ □□ □ □□□ □□□ □□ □□□□□ □□ □□□□□ □□□□□□ □□.

`argv`□ □□□ □□□ □□ □□□□. `parent`□ `process_exec()`□ □□□ □□□□□□ □□□□, □□ □□□□□□□□ □□ □□□ □□□ □□□ □□ □□□ □□ □□□ □□ □□□□□□ □□ □□□□□□ □□□ □□□ □□. □□□□□□ `success`□ □□ □□□□□□ □□□ □□ □□ □□□□□□□ □□□□□ □□□, □□ □□□□□□ □□□□□ □□ □□□□□□□□ □□□□□ □□ □□ □□□ □□□ □□□ □□□□□ □□□ □□□□□□.

□□□□□□, □□ □□□□□□ □□□□□□ □□ □□ □□□□□□ □□ □□ □□□ □□ □□□ □□ □□□□□□ □□□. □□ □□ □□□□□□ □□□ □□□□ □□ `PID_ERROR`□ □□□□□, □□ □□ □□ □□□□□□ □□□□□ □□□(process identifier; PID)□ □□□□□.

`start_process()`

```
/* From userprog/process.c. */
/* A thread function that loads a user process and starts it
   running. */
static void
start_process (void *frame_)
{
  char **argv = frame->argv;
  struct process *par = frame->parent;
  void *esp, *cmd_line = pg_round_down (argv[0]);
  struct intr_frame if_;
  ...

  /* Load executable and make process. */
  cur->process = make_process (par, cur);
  load_success = load (argv[0], &if_.eip, &if_.esp);
  *success = load_success && (cur->process != NULL);
  sema_up (&par->sema);

  /* If load or process making failed, quit. */
  if (!(*success))
    {
      palloc_free_page (argv);
      palloc_free_page (cmd_line);
      cur->process = NULL;
      destroy_process (cur->process);
      thread_exit ();
    }

  /* Get ARGC. */
  for (argc = 0; argv[argc] != NULL; argc++);

  /* Push arguments in ARGV in reverse order, calculate padding, and
     store pushed arguments's address on ARGV since original address
     is no longer needed. */
  padding = 0;
  esp = if_.esp;
  for (i = argc - 1; i >= 0; i--)
    {
      len = strlen (argv[i]) + 1;
      esp = push (esp, argv[i], len);
      argv[i] = esp;
      padding = WORD_SIZE - (len + padding) % WORD_SIZE;
    }

  /* Push padding and null pointer indicating the end of argument vector.
*/
  esp = push (esp, NULL, padding);
  esp = push (esp, NULL, sizeof (char *));

  /* Push argument vector. */
  for (i = argc - 1; i >= 0; i--)
    esp = push (esp, &argv[i], sizeof (char *));
  esp = push (esp, &esp, sizeof (char **));

  /* Push ARGC and dummy return address. */
  esp = push (esp, &argc, sizeof (int));
  esp = push (esp, NULL, sizeof (void (*) (void)));
  if_.esp = esp;

  palloc_free_page (argv);
  palloc_free_page (cmd_line);
  ...
}
```

□□ □□□□ □□ □□□□□ □□ □□□□ □□□ □□□□□ □□□□ □□□□. □□ `make_process()`□ `load()`□ □□□□ `struct process`□ □□□□, □□ □□□ □□□□ □□□□, □□ □□□□□ □ □□□ □□ □□□ □□ □□ □□□ □□□□. □□ □□□□□ □ □□ □□□ □□ □□□ `sema_down()`□□ □□□□ □□□□□, `sema_up()`□□ □□ □□□□□ □□□ □□□□□.

□□□□ □□ □□□□□□ □□□ □□ □□□ □□ □□(argument count; ARGC) □□ □□□ □□ □□□□ □□□ □□□□. □□ □□□ □□ □ □ `push()`□ □□□ Pintos □□□ □□□ □□ □□□ □□□ □□□□ □□□□ □□□□□.

```
/* From userprog/process.c. */
/* Pushes SIZE bytes of data from SRC at the top of the stack specified
   with TOP. TOP must be a pointer to somewhere in user virtual address
   space. Or PHYS_BASE if the stack is empty. Returns an address that
refers
   to new top of the stack. If SRC is a null pointer, then pushes SIZE
bytes
   of zeros on the stack. */
static void *
push (void *top, void *src, int size)
{
  ASSERT (is_user_vaddr (top) || top == PHYS_BASE);

  char *new = (char *) top - size;
```

```
    if (src == NULL)
      memset (new, 0, size);
    else
      memcpy ((void *) new, src, size);

    return (void *) new;
  }
```

push()□ □□ top□ □□□□ □□□ □□□□ src□ □□□□ □□□□ size□□ □□□ □, □□□ □□□ □□□□ □□□□ □□□□ □□□□ □□ □□. □□□ □□□□□ □□□ PHYS_BASE□□ □□□□□, □□□ □□□ □□□□ □□□ PHYS_BASE □□□□ □□. □□ src□ □ □□□□□, □□ □□□□ □□□□ □□□ □□ 0□ size □□□□□ □□□ □□□.

□□ □□ □□□ □□ □□□ □□ □□□□ □□ □ □□ □□ □□□ □□ □□ □□(argv)□ □□□□ □□□(cmd_line)□ □□ □□□□. □□ □□□□ □□□□ □□□□ □□□□ □□ □□ □□□□□□ □□□□ □□, □□□ □□ □□□ □□□□ □□□□ □□□□ □□□ □□□ □ □□□□ □□□□ □□ □□□□ pg_round_down()□ □□□ □□ □□□□ □ □□ □□□ □□ □□□ □□□□□.

process_wait()

```
/* Waits for process CHILD_PID to die and returns its exit status.
   If it was terminated by the kernel (i.e. killed due to an
   exception), returns -1.  If CHILD_PID is invalid or if it was not a
   child of the calling process, or if process_wait() has already
   been successfully called for the given PID, returns -1
   immediately, without waiting. It must be called in user process
   context. */
int
process_wait (pid_t child_pid)
{
  int status;
  struct process *child = NULL, *this = thread_current ()->process;
  struct list_elem *e;

  ASSERT (this != NULL);

  /* Find child to wait. */
  for (e = list_begin (&this->children); e != list_end (&this->children);
       e = list_next (e))
    {
      child = list_entry (e, struct process, elem);
      if (child->pid == child_pid)
        break;
    }

  /* If there's no such child or the child is already waited, return -1.
*/
  if (child == NULL || child->pid != child_pid || child->waited)
    return -1;

  /* If the child to wait is still alive, wait for it to exit. */
  child->waited = true;
  if (child->state == PROCESS_ALIVE)
```

```
    sema_down (&this->sema);

  /* The child has exited. Get its exit status and clean it up. */
  status = child->status;
  list_remove (&child->elem);
  destroy_process (child);
  return status;
}
```

□□ □□□□□ □□□ child_pid□ □□□□ □□ □□□□□ □□□ □□□ □□□ □□□□□ □□□□. □□ □□, process_wait()□□□ □□ □□ □□□□□ □□ □□□□ □□ □□□ child_pid□ PID□ □□ □□□□□ □□□. □□ □□□□ □□□□□ □□ □□□□ □□ □□ □□□□□ □□ □□□□ □□ □□ -1□ □□□□. □□ □□□□ □□ □□□□□ □□ □ □□□ □□, □□ □□ □□□□□ □□□□ □□ □□, □ □□ □□□□□ state□ PROCESS_ALIVE□ □□ □□ □□ □□□□□ □□□ □□□ □□□□.

□□ □□ □□□□□ □□ □□□ □□□□, □□ □□ □□□□□ □□□□□, □□ □□□ □□□□ □□ □□□ □□□.

make_process()

```
/* From userprog/process.c. */
/* Make process whose parent is PAR and whose associated kernel thread is
T.
   Also, does basic initializations on it. Returns null pointer if memory
   allocation has failed or page directory creation has failed. */
static struct process *
make_process (struct process *par, struct thread *t)
{
  struct process *this = (struct process *) malloc (sizeof (struct
process));

  ASSERT (t != NULL);

  if (this == NULL)
    return NULL;

  this->pid = t->tid;
  this->state = PROCESS_ALIVE;
  strlcpy (this->name, t->name, sizeof this->name);
  sema_init (&this->sema, 0);
  this->thread = t;
  this->waited = false;
  list_init (&this->children);
  list_init (&this->opened);
  t->process = this;

  if (par != NULL)
    {
      list_push_back (&par->children, &this->elem);
      this->parent = par->thread;
    }

  if ((this->pagedir = pagedir_create ()) == NULL)
```

```
    {
      free (this);
      return NULL;
    }

  return this;
}
```

`struct process`□ □□□□□. `struct process`□ □□□□ □□□□□, □□ □□□□ (□□ □□□)□ □□□□. □□ □□ □□□□ □□□ □□□ □□□□□ □□ □□ □□ □□□□□ □□ □□□□ □□ □□□□ □□□□□ □□□□. □□ `malloc()` □□ `pagedir_create()` □□ □ □□ □□□□ □□ □ □□□□ □□□□.

`destroy_process()`

```
/* From userprog/process.c. */
/* Free the resources of P. It neither exits its associated kernel thread,
   nor modifies its parent or children. It only frees memorys taken to
   represent process structure. If P is a null pointer, does nothing. */
static void
destroy_process (struct process *p)
{
  struct process *this = thread_current ()->process;

  if (p == NULL)
    return;

  uint32_t *pd = p->pagedir;

  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory when the destroyed process is
current
     process. */
  if (pd != NULL)
    {
      /* Correct ordering here is crucial.  We must set
         p->pagedir to NULL before switching page directories,
         so that a timer interrupt can't switch back to the
         process page directory. We must activate the base page
         directory before destroying own process's page
         directory, or our active page directory will be one
         that's been freed (and cleared). */
      p->pagedir = NULL;
      if (this == p)
        pagedir_activate (NULL);
      pagedir_destroy (pd);
    }

  free (p);
}
```

struct process□ □□□□□. □□ □□□□□□ □□□□□□ □□□ □□ □□□ □□ □□ □□□□□□ □□ □□□□ □□ p□ □□□. □□, □□□ process_exit() □□□□□□ □□ □□□□□□ □□□ □□□□□□ □□ □□□□ □□ □□ □□ □□□ □□□□□□ □□□□□□ □□□□□ □□□□□□□, □□ □□□ □□□□□□ □□□□□□ □□□ □□ □□□ □□ □□ □□□□□ □□□□□□ □□□□□□ □□□ □□ □□□□□□ □□ □□□□. □□ p□ □□□ struct process□ □□ □□□□.

□□□□ □ □□ □ □□□ □□□□□□ □□-□□ □□□ □□□ □□□□□□ □□□ □□□□ □□ □□□□□□□ □□□ □□ □□□□ □□□. □, □ □□□ □□□□□□ □□ □□□□□□ □□□□ □□ □□□□ □□□□ □□□, □□ □□□□□□ □□□ □□□□ □□□ □ □□□ □□ process_exit()□ □□ □□□□ □□.

```
/* From userprog/process.c. */
/* Sets up the CPU for running user code in the current thread.
   This function is called on every context switch. */
void
process_activate (void)
{
  struct process *this = thread_current ()->process;

  /* Activate thread's page tables. If it is kernel which does not have
     associated user process, activate initial one that has only kernel
     mapping.*/
  if (this == NULL)
    pagedir_activate (NULL);
  else
    pagedir_activate (this->pagedir);

  /* Set thread's kernel stack for use in processing
     interrupts. */
  tss_update ();
}
```

□□ □□(context switch) □□□□ □□□□, □□ □□□□□□ □□□ □□□□□□ □□□□□□ □□□□. □□□□ □□□ □□□ □□□□ pagedir□ □□□□□□ □□□ □□□□□□□, □□□ □□□□□□ sturct thread□ pagedir □□□ □□□□ □□ □□ process □□□ □ □□□□□□□ □ □ □□□□□, □□□ □□□□□□□□ □□□□□□ □□ □□□ □□□ □□□□□.

```
/* From userprog/process.c. */
/* Exits current process with exit code of STATUS. This must be executed
in
   user process's context. */
void
process_exit (int status)
{
  struct file *fp;
  struct list_elem *e, *next;
  struct thread *cur = thread_current ();
  struct process *child, *this = cur->process;

  ASSERT (this != NULL);

  struct thread *par;
  par = this->parent;
  this->state = PROCESS_DEAD;
  ASSERT (!(this->waited && par == NULL));
  ASSERT (!(this->waited && par->process == NULL));

  this->status = status;

  /* This is to maintain consistency of process structures. Interrupt will
be
     enabled after thread_exit() call which causes context switch. */
  intr_disable ();

  /* Exiting process's children are orphaned. Destroys dead children that
     their parents are responsible for destroying. */
  for (e = list_begin (&this->children); e != list_end (&this->children);
)
    {
      child = list_entry (e, struct process, elem);
      next = list_remove (e);
      child->parent = NULL;

      if (child->status == PROCESS_DEAD)
        destroy_process (child);

      e = next;
    }

  /* Close opened files. */
  for (e = list_begin (&this->opened); e != list_end (&this->opened); )
    {
      next = list_remove (e);
      fp = list_entry (e, struct file, elem);
      file_close (fp);
      e = next;
    }

  printf ("%s: exit(%d)\n", this->name, status);

  /* For processes who are orphaned, they are responsible to destory
themselves.
     For those who are not orphaned, their parents are responsible to
destroy
     children. */
  if (this->waited)
    sema_up (&par->process->sema);
  else if (par == NULL || par->process == NULL)
    destroy_process (this);

  cur->process = NULL;
  thread_exit ();
}
```

□□ □□□□□ □□ □□ status□ □□ □□□□□ □□□□. □□ □□ □□ □ □□□ □□□□ □□ □□(invariant)□ □□□, process_exit()□ □□□ □□□ □□□□ □□ □□□ □□□□□ □□. □□, □□ □□□□□ □□ □□ □□□□□ □□□□ □□□, □ this->waited□ □□□□, □□ □□□□□ □□□□ □□ □□ ASSERT □□□ □□□□ □□. □ □□ □□ □□□□, □ □□ □□ □□□□□ □□□ □□□ □ □□□□, □□□ □□ □□□ □□ this□ par, par->process□ □□□ □□□ □ □□ □□□□.

□□□ □□□ □□□ □ process_exit()□□□ □□ □□□□□ □□□□□□□, □□ □□□□ □□□□ □□ □□□□□ □□□ □□□□ □□□ struct process□ □□□□ □□□□ □□□□ □□ □□□□ □□□□ □□□□□ □□□ □□□ □□ □□□□.

□□□ □□ □□□ □□ □□ □ □□□□□ □□ □□□□□□ parent □□□ □ □□□□ □□□□□□ □□ □□□□□□ □□(orphaned) □□□□□ □ □□. □□ □□ □□□□□□ □□□ □ □□□ □□ □□□□□□□ □□□□ □□ □□□ □□ □□ □□ □□□ □□□ □□. □□, □□ □□ □□□ □□ □□□□□ □ □□□□ □□ □□□□□ □□ □□□ □□□ □□□ □□□□□.

□□□□ □□ □□□□□□ □□□ □□□ □□□, □ □□ □□□ □□□□□ □□ □□ □□ □□ □□. □□□, □□□□ □□ □□ □□□ □□ □ □ □□ □□. □ □□□□ □□ □□□ □□□ □□□□□ □□□□ load() □□□ □□□□ □□ □□□□.

Pintos □□□ □□□ □□□ □□ □□□□ □□ □□□□ □□□ □□, □□ □□ □□□□□ □□ □□□□□ □□□□ □□ □□ □□ □□□□□ □□□ □□□ □. □□ □□□ □□□□ □□ □□□ □□□ □□□ □□. □, □□ □□□□□ □□ □□□ □□ □□ □□□ □□ □□ □□□ □□□ destory_process()□ □□□ □□ □□□ struct process□ □□ □□□□.

□□ □□ □□□□ □□□ □□□□□ □□□ □□□□ □□ cur->process□ □ □□□□ □□□□ thread_exit() □□□ □□□□ □□□ □□ □□□□ □□ □□ □□□□.

load()

```c
/* From userprog/process.c. */
/* 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)
{
  ...
  if (thread_current ()->process == NULL)
    goto done;

  /* Activate page directory. */
  process_activate ();

  /* Open executable file. */
  lock_acquire (&filesys_lock);
  file = filesys_open (file_name);
  lock_release (&filesys_lock);
  ...
  file_deny_write (file);
  ...
}
```

□□□ □□□□□ □□ □□□ □□□□ □□□□ □□□□. □□□□ □□□ □□□□ □□ □ □□□□ □□□□□ □□□□ □□□□, □□ □□□ make_process()□□ □□□ □□□□□ □□□□□ □□□□ □□□ □□□□□□, Pintos □□□ □□□ □□ □□ □□□ □□ □□□ □□ □□□ □

□□ □□□ □□□□ lock□ □□□□□ □□□□□. □□ □□ □□□□ □□□ □□□□□ □□□□ □□□□ □□□ □□□□□ □□□□□.

□□ `filesys_open()`□ □□ □□ □□□ □□□, □ □□□□ □□ □□□ inode□ □□ `file_open()`□ □□□□, `file_open()` □ □□□ □□ □□□□□ `opened` □□□□ □□□□. □□ `file_deny_write()` □□□□ □□ □□□ □□ □□□ □□□□□. □ □□ □□□ □ □□ □□ □□ `process_exit()` □□ □ □□□ □□ □□□ □□□□.

`struct file`

```c
/* From filesys/file.h. */
/* An open file. */
struct file
  {
    struct inode *inode;        /* File's inode. */
    off_t pos;                  /* Current position. */
    bool deny_write;            /* Has file_deny_write() been called? */
    int fd;                     /* File descriptor. */
    struct list_elem elem;     /* List element for opened file list. */
  };
```

□□ □□□□□□ □□□ □□□ □□□ □□□□ □□□□ `struct file`□ □□□ □□□□□ □□□ □□ □□□□ □□ □□□ □□□. □□, □□□ □□□ (file descriptor)□ □□□ □□□□□ `opened` □□□□ □□□□ □□ `list_elem`□ □□□□□.

`file_open()`

```c
/* From filesys/file.c. */
/* Opens a file for the given INODE, of which it takes ownership,
   and returns the new file.  Returns a null pointer if an
   allocation fails or if INODE is null. */
struct file *
file_open (struct inode *inode)
{
  struct process *this = thread_current ()->process;
  struct file *file = calloc (1, sizeof *file);

  ASSERT (this != NULL);

  if (inode != NULL && file != NULL)
    {
      file->inode = inode;
      file->pos = 0;
      file->deny_write = false;
      file->fd = allocate_fd ();
      list_push_back (&this->opened, &file->elem);
      return file;
    }
  else
    {
      inode_close (inode);
      free (file);
      return NULL;
    }
}
```

inode□ □□□ □□□ □□ □□□□. □□□ □□□ □□ □□□ □□ □□□□ □□□□ □□□ □□ □□□□□ □□ □□ □□□□ □□ □□□ □□□□ □□□ □□□□□.

```
/* From filesys/file.c. */
/* Returns fresh file descriptor to represent a newly opened file. */
static int
allocate_fd (void)
{
  /* File descriptor of 0 and 1 are reserved for stdin and stdout. */
  static int next_fd = 2;
  int fd;

  lock_acquire (&fd_lock);
  fd = next_fd++;
  lock_release (&fd_lock);

  return fd;
}
```

allocate_fd()□ □□□ □□ □□□□ □□□□ □□□□ □□□□. □□ □□ next_fd□ □□ □□□□ 1 □□□□□. □□ next_fd□ 2□ □□□□□□, □□ □□□ 0□ 1□ □□ □□□□ □□□□□□ □□□□.

thread_exit()

```
/* From threads/thread.c. */
/* Deschedules the current thread and destroys it.  Never
   returns to the caller. It does not destory its associated process,
hence
   it might be dangling. Therefore, it should not be called when there's
an
   associated user process of current thread. Use process_exit() instead.
*/
void
thread_exit (void)
{
  struct thread *cur = thread_current ();

  ASSERT (!intr_context ());
  ASSERT (cur->process == NULL);

  intr_disable ();

  /* Remove thread from all threads list, set our status to dying,
     and schedule another process.  That process will destroy us
     when it calls thread_schedule_tail(). */
```

```
    list_remove (&cur->allelem);
    thread_current ()->status = THREAD_DYING;
    schedule ();
    NOT_REACHED ();
  }
```

□□□ `thread_exit()`□ □□□□□ □□□□□□ □□□ □□□□□□ □□□ □□□□□ □□ □□ □□□□□□ □□□□□ □□□. □□□ □□□□□□ □□□□□□ □ □□ □□□□□ □□ □□ □□□ `destroy_process()`□ □□□□□□ □□ □□□ `thread_exit()`□ □□□ □□□□□□. □□ □□ □□□ □□ □ □□□ □□□ □□□ □□□□□□ □□ □□□□□□ □□□ □□□□□□.

Discussions

How to retreieve actual pointer from descriptors or handles?

Pintos □□□ □□□ `open()` □□□ □□□ □□ □□□□□ □□□ □□ □□□ □□□□□ □□, `exec()` □□□ □□□ □□□□□ □□□□□ □□□□□ □□. □□, □□ □□□□□ □□□ □□ □□□ □□□ □□ □□ □□□□□ □□□ □□□ □□□□□ □□□ □□□ □□□□□ □□.

□□□ □□□ □□ □□□ □□□□□ □□□, □□ □□□ □□ □□ □□□□□□ □□ □□□□□ □□□□□□ □□ □□□ □□□□. □□ □□ □□ □ □□ □□ □□□ □ □□□.

□ □□ □□ □□□ □□ □□□ □□ □□□□□□ □□ □□□□□ □□□□□□ □□□□. □ □□□ □□□□□□, □□ `struct file *` □□ `struct process *`□ `int` □□ `pit_t`□ □ □□(casting)□□ □□□□□ □□□ □□□ □□□. Pintos □□ □□□ Q&A □□□□□□ □□ □□ □□ □□ □□. □ □□□ □□□ □□ □□□□□ □□□□□ (□□□□□□ □□□□□ □□□□□ □ O(1)□ □□□ □□□), □□□□□ □□□ □ □□ □□ □□□□□□ □□ □□ □□□ □□□ □□□ □ □□□ □□□ □□□ □□□□□□.

□ □□ □□ □□□ □□□ □□□□□ □□□□□ □□□ □(table) □□ □□□ □□□ □□ □□□□□ □□□□. □□□□□ □□□□□□ □□□ □ □□□□□□ □□□ □ □ □□□□□ □□□□□ □□□□□□, □□□ □□ `fd` □□ `pid`□ □□ □□ □□ □□□ □□□□□ □□□ □□□ □□. □□□□□□ □□□ □□□ □□□ □□ □□□□□ □□□ □□ □□ □□□ □□□□□□□, □□□□□□ □□□ □□□ □□□□□ □ □□ □□□□□ □□□ □□□ □□ □□ □□ □□ □□□□□□.

□□□ □□ □□□ □□□ □□□□□□ □□ □□ □□□ □□, □□ □□ □□□□□□ □□□□□ □□□□□ □□□ □□□□□ □□□□□ □ □□□ □□ □□□□□ □□□□□ □ □□□. □ □□□ □ □□□□□ □□ □□ □□, □□ □□ □□□□□□ □□□ □□□□□ □□□ □□□□□, □□□□□ □□□ □□□□□ □□□ □□□□□ □□□□□□ □□ □ □□ □□ □□□ □□□□□□.

□ □ □□□ □□□□□□, `struct process`□ □□ □□□□□□ □□ □□ □□□ □□□□□ `opened`□ □□ □□□□□□ □□□□□ `children`□ □ □□ □□□ □□. □□ □□□□□ □□□ □□□ □□ □□□□□ □□□□□ □□□□□ □□□ □□□, `opened`□□ □□ □□ □□□□□ □□ □□□ □□□ `children`□□ □□ □□□□□ □□□□□ □□ □□□□□□ □□□.

Pintos□ □□ □□ □□□□□□ □□□□□ □□□ □□□ □□ □□□ □□□□□ □□□□□□□, □□ □□□□□ □□□□□□ □□ □□□ □□□□□ □□□□□□ □□□ □□ □□□□□□ □□□ □□ □□□ □□ □□□ □□□ □□□. □ □□ □□□□□□ □□□ □□□□□ □□□ □□□□□ □□ □ □□□ □□□□□ □□□□□, □□ □ □□□ □ □ □□□ □□□ □□□ □ □□ □□□ □□□ □□□ □□□□□ □□□ □□□□□ □□□□□ □□□ □□ □□□. □ □□ □□□ □□□□□ □□□□□□ □□ □□□□□ □□ □□ □□□□□ □□□ □□□ □□□□□□ □□□ □ □□□□□ □□□□□□□, □□□ □□□□□ □□, □□ □□□□□ □□□ □□□ □□□□□ □□□ □□□ □□□ `allocate_fd()` □□ `allocate_tid()`□ □□□ □□ □□ □□□ □ □□□□□ □□□.

System Calls

Improvements

□□□□□□ User program□□ system call□ □□□□□ □□□□□ `lib/user/syscall.h` □□ □□□□□ □□ system call□ □□ □□ □□ □□□□□□□ □□□□□ `userprog/syscall.c` □ `syscall_handler()`□ □□□□□ □□ □□□□□ □□□□□. □□□ □□□ □□ □□ □□ □□□□□□□ □□□□□ system call □ □□□□□□ □ □□□ number □ □□ □□□ □□□□□ □□□□□ □□□□□□.

Details and Rationales

dereference()

```
/* lib/user/syscall.c
#define syscall0(NUMBER) */
...
("pushl %[number]; int $0x30; addl $4, %%esp"       \
    : "=a" (retval)                                   \
    : [number] "i" (NUMBER)                           \
    : "memory");                                      \
...

/* Dereferences pointer BASE + INDEX * OFFSET, with a validity test.
Returns
   4 byte chunk starting from BASE + OFFSET * INDEX if it passes the test.
   Else, terminates current process. */
static uint32_t
dereference (const void *base, int index, int offset)
{
  const uint8_t *base_ = base;
  void *uaddr = (void *) (base_ + offset * index);

  if (is_user_vaddr (uaddr))
    return *((uint32_t *) uaddr);
  else
    process_exit (-1);

  NOT_REACHED ();
}
```

□□ □□□□□□ system call□ □□□□□□ □□□ □□ system call□ □□□□□□ □□□ □□ □□□□□ □□□ □□□□□□ □□□□□ □□. □□ lib/user/syscall.c□□ □□□ □ □□ □□□ user program□□ syscall □ □□□□□ □□□ □□□□□ □□□□□ □□ □□, syscall□ □□□ □□□□□ □□ □□□□□ pushl□ □□ □□ □□□□□ □□□□□□ □□ □ □ □□. □□, □□□ int 0x30□ □□ Kernel mode□ □□□□□ syscall_handler□ □□□□□□ □□ □□□ □ □ □□. □ □□□ □□□ □□□□□□ □□□□□ □□ □□□□□ □□□□□. □□□ □□□ □□□ □□□□□ □□ kernel □□□ stack frame □ □□□ □□□□□ □□ □□ □□□□□ □□□□□ □□ □□□ dereference()□ □□□□□□. □ □□□□□□ BASE + INDEX * OFFSET □□□□□ uint32_t □□□ □□□□□ □□□□□ □□□□□. □□, □□ □□□ □□□□□ □□□ □□□□□ □□ is_user_vaddr □ □□ □□□ □□□ □□ □□□□□ □□□□□ □□. User program□□□□ □□□ □□□ □□□□□ □□ □ □□ □□□, □□ □□□□□ □□□ □□□□□□ □□ □ Overhead □ □□ □□□ □□ □□□□□ □□ PHYS_BASE □□ □□ □□ □□□□□ □□ □□□ □□□□□ page_fault □ □□ □□□□□ □□□□□ □□□ □□□□□.

exit()

```
/* System call handler for exit(). */
static void
exit (void *esp)
{
  int status = (int) dereference (esp, 1, WORD_SIZE);
  process_exit (status);
}
```

□□ user program□ □□□ □□□□ □□□, □□□ □□□ □□ exit code□ □□□□ □□. □□□□□ 0□ □□ □□□ □□ □□□ □□□ □□□. □□ user program□ □□□ kernel thread □□ user program□ □□□□ □□ process□ □□□□ □□□□ □□□ □□□□ □□□□, `exit` □□□ □□□□□ process□ □□□□ □□ `process_exit()` □□ `status`□ □□□□ □□□□ □□.

`get_user()` and `put_user()`

```
/* Reads a byte at user virtual address UADDR.
   UADDR must be below PHYS_BASE.
   Returns the byte value if successful, -1 if a segfault
   occurred. */
static int
get_user (const uint8_t *uaddr)
{
  int result;
  asm ("movl $1f, %0; movzbl %1, %0; 1:"
       : "=&a" (result) : "m" (*uaddr));
  return result;
}

/* Writes BYTE to user address UDST.
   UDST must be below PHYS_BASE.
   Returns true if successful, false if a segfault occurred. */
static bool
put_user (uint8_t *udst, uint8_t byte)
{
  int error_code;
  asm ("movl $1f, %0; movb %b2, %1; 1:"
       : "=&a" (error_code), "=m" (*udst) : "q" (byte));
  return error_code != -1;
}
```

user address□□ 1 byte□ □□□ □ □□□□□ □□□ □□□, □□□□□ □□ □□□ □□□□□. result □ 1f□ □□□□ `movzbl` □□ □ □□□ □□□□ `1f`□ □□□□, result □ -1□ □□□□ □□□□ □□. `movzbl` □ □□□ □□□□ □□□□ result □ □□□ 1 byte□ □□□□ □□□□ □□□□. `put_user` □ □□□□ □□□□□ □□ `PHYS_BASE` □□ □□ □□ □□□□ □□□□, User address□ 1 byte □ □□ □□ □□□ □□□□ □□.

`verify_string()` & `verify_read()` & `verify_write()`

```
/* Verifies null-terminated STR by dereferencing each character in STR
until
   it reaches null character. Return true if and only if all characters in
STR
   are valid. */
static bool
verify_string (const char *str_)
```

```
{
  char ch;
  uint8_t *str = (uint8_t *) str_;
  for (int i = 0; ; i++)
    {
      if (!is_user_vaddr (str + i) || (ch = get_user (str + i)) == -1)
        return false;

      if (ch == '\0')
        break;
    }

  return true;
}


/* Verifies read buffer BUF whose size is SIZE by trying to read a
character
   on each bytes of BUF. Return true if and only if all bytes in BUF are
   readable. */
static bool
verify_read (char *buf_, int size)
{
  uint8_t *buf = (uint8_t *) buf_;
  for (int i = 0; i < size; i++)
      if (!is_user_vaddr (buf + i) && (get_user (buf + i) == -1))
        return false;

  return true;
}

/* Verifies write buffer BUF whose size is SIZE by trying to put a
character
   on each bytes of BUF. Return true if and only if all bytes in BUF are
   writable. This function fills 0 on BUF. */
static bool
verify_write (char *buf_, int size)
{
  uint8_t *buf = (uint8_t *) buf_;
  for (int i = 0; i < size; i++)
      if (!is_user_vaddr (buf + i) && put_user (buf + i, 0))
        return false;

  return true;
}
```

user program□□ □□□□ □□□□□ □□□□ □□□□ □□□□□ □□ □□□□ □□ □□ □□ □□□□ □□□ □□□□□□ □□□□ □□. □□□ □□ □□ □□□ □□□ `cmd_line` □□ □□□□ string □ □□□□ □□□ `PHYS_BASE` □□□ □□□ □□□ □□□□ □□□□□ □□□□. □□ `get_user()` □ □□□ □□□□ □□□ □□□□ □□, □□□□ □□ `\0` □ □□□ □□□ string□ □□□□ string □□□ □□□ □□□ □□□□ □□. `verify_read()` □ □□ □□□□ buffer□ □□□ size □□□ byte □ □□□ □ □□ □ □□□ □□□ □□□□□ □□□□. `verify_write()` □ □□□□ write buffer□ □□□□□, buffer□ □□□□ size □□□ write□ □ □ □□□ □□□ □□□□ □□.

`halt()`

```
/* System call handler for halt(). */
static void
halt (void)
{
  shutdown_power_off ();
}
```

□□□□ □□□ □ □□□□ system call □ `shutdown_power_off()` □ □□ `halt()` □ □□□□□.

`exit()`

```
/* System call handler for exit(). */
static void
exit (void *esp)
{
  int status = (int) dereference (esp, 1, WORD_SIZE);
  process_exit (status);
}
```

`exit()` □ □□ status □ □□□□ □□ □□□□ process □ exit □□ □□□ □□□□□□, □□□ □□□ `process_exit()` □□ status□ □□□□, □□ □□□ □□ termination message □□ □ exit □□ □□□ □□□□ □□.

`exec()`

```
/* System call handler for exec(). */
static uint32_t
exec (void *esp)
{
  char *cmd_line = (char *) dereference (esp, 1, WORD_SIZE);
  if (!verify_string (cmd_line))
    return (uint32_t) TID_ERROR;

  return (uint32_t) process_execute (cmd_line);
}
```

`cmd_line` □ □□ □□□□ □□□□ □□□□ User program □ □□□□ □□□, `cmd_line` □□□□□ string□ □□□ □□□ □□□ □ □□□ □□. □□□ □□ □□ □□□□□ user program□ □□□ □□□□ process □ □□□□ □□□□□, `process_execute()` □ □ □ `cmd_line` □ □□□□ Process □ □□□□□ □□□□. □□ □□ □□□ □□ thread □ □□□□ □□□□ □□□ user program □ □ □□□.

`wait()`

```
/* System call handler for wait(). */
static uint32_t
wait (void *esp)
{
  tid_t tid = (tid_t) dereference (esp, 1, WORD_SIZE);
  return (uint32_t) process_wait (tid);
}
```

□□□□ pid□ □□ child process□ □□□ □ □□ □□□□ □□□, □□□ □□ □□□ child process□ exit code□ □□□□ □□. □□ □ □□ □□□ □□□ □□ wait □ □□□ `process_wait()`□ □□ □□□□.

`create()`

```

/* Lock to ensure consistency of the file system. */
struct lock filesys_lock;

/* System call handler for create(). */
static uint32_t
create (void *esp)
{
  uint32_t retval;
  char *file = (char *) dereference (esp, 1, WORD_SIZE);
  unsigned initial_size = dereference (esp, 2, WORD_SIZE);

  if (!verify_string (file))
    return (uint32_t) false;

  lock_acquire (&filesys_lock);
  retval = (uint32_t) filesys_create (file, initial_size);
  lock_release (&filesys_lock);

  return retval;
}
```

□□ file system □□ □□□ □□□ □□□□ □□□ □□□□ system call □□. design □□□ file system □□ sync □ □□ □□□ □ □□□ □□□ □□ □□□□, □□□ □□ □□□□ lock□ □□□□ □□□ □□□□ □□ □□□ □□. □□ □□□□□ `filesys_create()`, `filesys_remove()`, `filesys_open()` □ □□ file system□ □□□□ □□ □□ □□□□ sync□ □□□□ □□□ □□□ `filesys_lock` □ □□□□□. □ lock □ □□□ □□□ □□□ □□□ atomic □□ □□□□ context switch□ □□ □□□ □□□□ □□ □ □□□□. `create()` □ □□□□□ □□□□□ □□. □□, □□□ □□ □□□ □□ □□ `file`□ □□□ □□□ `initial_size`□ □□□□□. □□ file □□, user program □□ □□□□ □ □□□□, □□□□ □□□ □□ □□□□ □□. □□□ □□ □□□ □□□ `filesys_lock□` □ □ file system□ □□□ □□□ atomic□ □□□□ □□, `file`, `inital_size`□ □□□□ `filesys_create()` □ □□□□ □□ □ □□□□ □□ □□□ □□□□ □□.

`remove()`

```
/* System call handler for remove(). */
static uint32_t
remove (void *esp)
{
  uint32_t retval;
  char *file = (char *) dereference (esp, 1, WORD_SIZE);

  if (!verify_string (file))
    return (uint32_t) false;

  lock_acquire (&filesys_lock);
  retval = (uint32_t) filesys_remove (file);
  lock_release (&filesys_lock);

  return retval;
}
```

□□ create() □ □□□ □□□ □□ □□□□, □□□□□ `verify_string()`□ `filesys_lock`, `filesys_remove()`□ □□ □□□□□□□, □□□ file □□□ □□□□ file □ □□□□, □□ □□□ □□□□ □□.

`open()`

```
/* System call handler for open(). */
static uint32_t
open (void *esp)
{
  struct file *fp;
  char *file = (char *) dereference (esp, 1, WORD_SIZE);

  lock_acquire (&filesys_lock);
  if (!verify_string (file) || (fp = filesys_open (file)) == NULL)
    {
      lock_release (&filesys_lock);
      return (uint32_t) FD_ERROR;
    }
  lock_release (&filesys_lock);

  return (uint32_t) fp->fd;
}
```

file□ □□ □□□□ file descripter □ □□□□ □□□□. □□□□ file □□□ `verify_string()` □ □□ □□□□, `filesys_open()` □ □□ file□ □□ □ □□ file pointer□ □□□□ □□. □ □□ □□ filesys_lock□ □□□ atomic □□□ □□ □□ □□. □□, `filesys_open()` □ □□ □□□ file pointer□ file descripter□ □□□□ □□□ □□□ □□□□.

`retrieve_fp()`

```
/* Retrieves file pointer from file descriptor, FD. Returns NULL if it has
   failed to find a file with file descriptor of FD among current
process's
   opened files. This must be called within user process context. */
static struct file *
retrieve_fp (int fd)
{
  struct process *this = thread_current ()->process;
  struct file *fp = NULL;
  struct list_elem *e;

  ASSERT (this != NULL);

  for (e = list_begin (&this->opened); e != list_end (&this->opened);
       e = list_next (e))
    {
      fp = list_entry (e, struct file, elem);
      if (fp->fd == fd)
        break;
    }

  if (fp == NULL || fp->fd != fd)
    return NULL;

  return fp;
}
```

`retrieve_fp()` □ file descripter□ □□□□ □□ process□□ open □□ □□ file pointer□ □□□□ □□□□. □□ system call □ □□□ int type□□ □ file descripter□□. □□□, □□ file system □ □□□ process □□□ □□ □□□□ □□ □□ int □□□□□□ □□□ □□□ □□□□, `file.c` □□ □□□ file pointer□ □□□□ □□□□□□□, □□□□ file descripter □ □□ □ □□□ file pointer □ □□□□ □□□ □□ □□□□□□ □□□□□.

`filesize()`

```
/* System call handler for filesize(). Return -1 if given file descriptor
is
   not a valid file descriptor. */
static uint32_t
filesize (void *esp)
{
  int fd = (int) dereference (esp, 1, WORD_SIZE);
  struct file *fp = retrieve_fp (fd);

  if (fp == NULL)
    return (uint32_t) -1;

  return (uint32_t) file_length (fp);
}
```

□□□□ file descripter□ file size□ □□□□ □□□□. □□ □□□ dereference() □ □□ fd□ □□, □□ □□□ □□□ retrieve_fP() □ □□ □□□□ file pointer□ □□□□ □□. file pointer□ □□ file_length() □ □□□□ □□□ □□□□.

`read()`

```
/* System call handler for read(). Returns -1 if given file descriptor is
not
   a valid file descriptor. Kills current process with exit status -1 if
given
   buffer pointer is invalid. */
static uint32_t
read (void *esp)
{
  int fd = (int) dereference (esp, 1, WORD_SIZE);
  void *buffer = (void *) dereference (esp, 2, WORD_SIZE);
  unsigned pos = 0, size = dereference (esp, 3, WORD_SIZE);
  struct file *fp = retrieve_fp (fd);

  if (!verify_write (buffer, size))
    process_exit (-1);

  if (fp == NULL && fp != STDIN_FILENO)
    return (uint32_t) -1;

  if (fd == STDIN_FILENO)
    {
      while (pos < size)
        ((char *) buffer)[pos++] = input_getc ();
      return (uint32_t) size;
    }

  return (uint32_t) file_read (fp, buffer, size);
}
```

read() □□□ buffer, pos, fd □ □□□□ fd□ □□□□ □□□ □□□□ □□□ □□□□. □ □□□ fd □ □□ 0□ □□□□ □□□ □□□ □□ □ □□ □□□□□ input_getc() □ □□□ □□□ □□□ □□ buffer□ □□□□. □□□ □□□□ □□□□ file pointer □ □□ buffer□ □□□ □ □□□ □□□□ □□. □□ open(), remove(), create() □□ □□□□ file system □□□ □□□□□ □□□□□ lock □ □□□ □□, write(), read() □ □□ □□□ □□□□ file □ □□□□ □□□□ filesys_lock □ □□□□ □□ □□.

`write()`

```
/* System call handler for write(). Returns 0 if it cannot write any byte
at
   for some reason. */
static uint32_t
write (void *esp)
{
  int fd = (int) dereference (esp, 1, WORD_SIZE);
  void *buffer = (void *) dereference (esp, 2, WORD_SIZE);
  unsigned size = dereference (esp, 3, WORD_SIZE);
  struct file *fp = retrieve_fp (fd);

  if ((fp == NULL && fd != STDOUT_FILENO) || !verify_read (buffer, size))
    return (uint32_t) 0;

  if (fd == STDOUT_FILENO)
    {
      putbuf ((char *) buffer, size);
      return (uint32_t) size;
    }

  return (uint32_t) file_write (fp, buffer, size);
}
```

file descripter, write□ □□ buffer, size □ □□□□ □□ file□ size □□ buffer□ □□□ □□□□ □□□ □□□□ □□ □□□□. dereference □ □□□ □□ □□□ □□□ □□ retrieve_fp() □ □□ file pointer□ □□□□ □□. □□ □□ fd□ 1□ □□□□ console □ □□□□ □□□, putbuf() □ □□ □□ □□□□ □□□□ □□. □□□ □□□□ file_write() □ □□ write □ □□□□.

seek()

```
/* System call handler for seek(). */
static void
seek (void *esp)
{
  int fd = (int) dereference (esp, 1, WORD_SIZE);
  unsigned position = dereference (esp, 2, WORD_SIZE);
  struct file *fp = retrieve_fp (fd);

  if (fp == NULL)
    return;

  file_seek (fp, position);
}
```

seek() □ □□ fd □□ file pointer□ □□□□ □□□ □□ position □□ □□□□ □□□ □□□□ □□□□. □□□□□□□ file pointer□□ Position □□ □□□ □□□□ □□□ □□□□. □□□ □□ □□ □□□ □□ □□ file_seek()□ □□□□ □□□□□.

tell()

```
/* System call handler for tell(). Returns -1 if given file descriptor is
not
   a valid file descriptor. */
static uint32_t
tell (void *esp)
{
  int fd = (int) dereference (esp, 1, WORD_SIZE);
  struct file *fp = retrieve_fp (fd);

  if (fp == NULL)
    return (uint32_t) -1;

  return (uint32_t) file_tell (fp);
}
```

□□□□□□ □□□ □□□ file_tell() □ □□□□□ □□□□□□□□, esp □□ □□□ □□□□□ □□ □□□□□ □□ □□□□□ □□ □□□□□ □□□□□ □□.

`close()`

```
/* System call handler for close(). Does notihing if given file descriptor is
   not a valid file descriptor. */
static void
close (void *esp)
{
  int fd = (int) dereference (esp, 1, WORD_SIZE);
  struct file *fp = retrieve_fp (fd);

  file_close (fp);
}
```

□□ file descripter□ □□□□□ □□ file pointer□ □□ □□□ □□□□□, file_close() □ □□ □□□□□□. file_close() □□□ □□□□ □□□ □□ □□ file_allow_write() □ □□ □□□ □□□□□ inode_close(), free(file) □ □□ file □ □□ □□.

Discussions

How should we check address from user program?

□□□ □□□ □□ □□ □□□ □□ □□□ □□ □□□□□□ □□□□□ □□□ page_fault□ □□ □□□ □□□□□ □□ □□□ □□□, □□ user program □□ □□□ address□ □□□□□ □□ □ overhead□ □□□ □ □□□□, □□ □□□□□ □□□□□ □□ □□□ PHYS_BASE □ □ □□□ □□□ □□ □□□□□□ user memory space □□□ □□□ □□ □□□ □□□□□ □□□□□ □□□□□□□□, □□ □□□ □□□ □□□□□□ page_fault □ □□□□□□ □□□□□□. □□□ string□ □□□□□ □□□ □□ □□ write, read □ □□ □□ buffer□ □□□□□ □□□□□ □□□ □ □□ □□□ □□□□□□ □□□□□ □□ □□□ □□□ □□□ □□□ □ □ □ □□□□□□ □□□□□□.

How should we manage with multi user process?

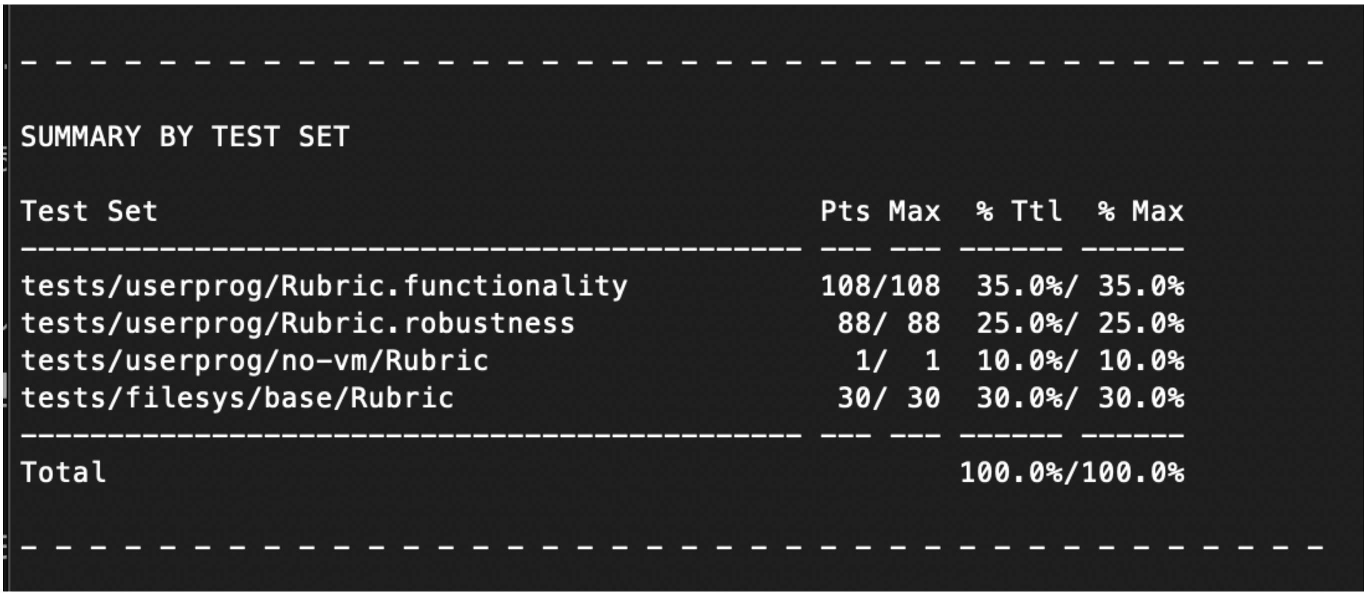
file system □□□ □□□□□□ Lock □ □□□ □□ □□□□□□ □□□ file system□ □□ □□ □□ □□□ □□□ □□ □□□ □□□ □ □□. □ □□ □ system call □□□ file system□ □□□□□ □□□□□ `filesys_lock` □ □□□ □□ lock □ □□ □□□□□ □□□ lock □□□ □ □□ □□ □□□ □□□□□ □□ □□□□□ □□□ □□□ □□□□□ □□□□□ □□□□□ □□□□□□.

difference between file pointer and file descripter?

□□ file pointer□ □□□□□ file descritper□ □□ □□□□□ □□□ casting □ □□ file descripter □ □□ □□ □□ struct file □ □□□□□ □□□□□ □□□ □□ □□□ □□ □□ □□ □□□ □□□□□ □□□ file descripter□ Kernel space □□□ □□□□□ □□□ □□□ □□

□ □□ file descripter □ □□□□ □□□□□□, □ process □□ □□□ opend list □ □□ □□□□ file pointer□□ □□ □□□□ □ □□□ □□□□□.

Conclusion



```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                       Pts Max  % Ttl  % Max
---------------------------------------------- --- --- ------ ------
tests/userprog/Rubric.functionality            108/108  35.0%/ 35.0%
tests/userprog/Rubric.robustness                88/ 88  25.0%/ 25.0%
tests/userprog/no-vm/Rubric                      1/  1  10.0%/ 10.0%
tests/filesys/base/Rubric                       30/ 30  30.0%/ 30.0%
---------------------------------------------- --- --- ------ ------
Total                                                  100.0%/100.0%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

□□ □□□□ □□ □□ □□□□ □□□ □ □□□□, project 2□ process termination message, argument passing, system calls, denying writes to executables □ □□ □□□□□.

□□□ □□□ □□□□□□ struct process□ □□ □□□□ □□, struct thread □ □□□□ □□□□ □□□ □□ □□□, wait() □ □□ □□ □□□□□ □□□□ □□ □□□ □□ □□□□□ □□ □□□□ □□□□ thread □ □□□□ □□□, □□□ □□ □□□□□ process□ exit status □ □□ □□□□ □□ □□□□ □□. □□□ □□□ □□□□ □□□ struct process□ □□□□ □□ □□□ □□ □□□ status □□ Process □□ □□□□ □□ wait() □□ □□□□ □□.

□□ □□ □□□□ □□ □□□□ □□ □□□ □□□□, system call □ □□ □□□□□ system call □ □□□□ □□□□□ □□□□□□, process □ method □□ □□□□ □□ □□□□ □□□□ □□□□, □□□ □□ □□□□ □□□□ □□ □□ □□□ □ □□ □□□ □□□□.