Project-2 Review

(Extending the functionality of PintOS user program implementation)

Summer 2018
Overview

- Necessities and Organization
- Argument passing in PintOS
- Implementation of System Calls
• Sub-checkpoint 1
  • Implement Argument Passing
• Sub-checkpoint 2
  • Implement System Calls

Only after you implement sub-checkpoints 1 and 2 properly, your tests would begin to pass
Overview

• Necessities and Organization
• Argument passing in PintOS
• Implementation of System Calls
Necessities and Organization

[Recall Project1 Approach]

**Step 1:** Familiarize with the tests associated with the component to implement.

**Step 2:** Use gdb to identify the code that is invoked during test execution.

**Step 3:** Note the functions that you feel need to be extended.

**Step 4:** Implement different combinations of functions written down in paper to the functions noted in Step 3.
Necessities and Organization:

**Step 1**

**STEP1** : Familiarize with the tests associated with the component to implement.
Necessities and Organization: Step 2

**Step 2**: Use `gdb` normal execution to identify the code that is invoked during test execution.
**Necessities and Organization: Step 3**

**Step 3**: Note the **functions** that you feel need to be extended.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>process_execute</td>
<td>Executes the user program from the designated file in the argument</td>
</tr>
<tr>
<td>process_wait</td>
<td>Waits for the child process with designated tid to finish before continuing execution</td>
</tr>
<tr>
<td>process_exit</td>
<td>Terminates user program currently running</td>
</tr>
<tr>
<td>process_activate</td>
<td>Sets up CPU to run user program in current thread</td>
</tr>
</tbody>
</table>
Necessities and Organization: Step 4

Step 4: Implement different combinations of functions written down in paper to the functions noted in Step 3.

My scratch paper regarding possible places for extensions.

In Project 2, rather than just applying different combinations, we need to create our own functions. (and thus challenging)
Overview

• Necessities and Organization
• Argument passing in PintOS
• Implementation of System Calls
Argument passing in PintOS

1. Where a User Program Starts
2. Emulate process_wait()
3. Setup Stack
Argument passing in PintOS

1. Where a User Program Starts
2. Emulate process_wait()
3. Setup Stack
Argument passing in PintOS: Where a User Program Starts

You create a filesystem disk and run a program called echo

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

The program internally starts in `process_execute()`

```c
tid = thread_create(file_name, PRI_DEFAULT, start_process, fn_copy);
```

In `load()` function in `process.c`, you’ll find function called `setup_stack`. You’ll need to setup stack here.
Argument passing in PintOS: Where a User Program Starts

Keep in mind these constraints.

Points of note for working with current file system:
* No internal synchronization. (You should use synchronization to ensure only one process is accessing the file system).
* File size is allocated at creation time.
* File sized contiguously.
* No subdirectories.
* File names limited to 14 characters.
* A system crash would corrupt the disk.
Argument passing in PintOS

1. Where a User Program Starts
2. Emulate process_wait()
3. Setup Stack
Argument passing in PintOS: Emulate process_wait()

process_wait(). Returns immediately with processing input arguments

```
int process_wait(tid_t child_tid)
{
    return -1;
}
```

You need to change `process_wait()` to wait for the child test case that has spawned.

```
int process_wait(tid_t child_tid)
{
    while(true)
    {
        thread_yield();
    }
}
```
Argument passing in PintOS

1. Where a User Program Starts
2. Emulate process_wait()
3. Setup Stack
Argument passing in PintOS: Setup Stack

setup_stack only takes the stack pointer `void** esp`

`setup_stack` doesn’t have access to filename. You’ll need to pass in the filename and file arguments to setup_stack using `strtok_r`.

Make sure you account for *NULL terminated arguments* and *word alignments*. Otherwise, input arguments would not be read properly.

```c
char argument[] = "arg1\0"
*esp -= strlen(argument);
memcpys(*esp, argument, strlen(argument));

int word_align = 0, 1, 2, or 3
*esp -= word_align;
memset(*esp, 0, word_align);
```
Argument passing in PintOS: Using `hexdump()` for debugging

**Format**

```c
static void hex_dump((uintptr_t)**, void**, int, bool);
```

**Example**

```c
hex_dump((uintptr_t)*esp, *esp, sizeof(char) * 8, true);
```

**Hex Dump for args-none**

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfffffff0</td>
<td>00 00 00 00 01 00 00 00-ec ff ff bf f6 ff ff bf</td>
</tr>
<tr>
<td>bfffffff0</td>
<td>00 00 00 00 00 61 72-67 73 2d 6e 6f 6e 65 00</td>
</tr>
</tbody>
</table>
```

**Hex Dump for args-single**

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfffffff0</td>
<td>00 00 00 00-02 00 00 00 e0 ff ff bf</td>
</tr>
<tr>
<td>bfffffff0</td>
<td>ed ff ff bf f9 ff ff bf-00 00 00 00 61 72 67</td>
</tr>
<tr>
<td>bfffffff0</td>
<td>73 2d 73 69 6e 67 6c 65-00 6f 6e 65 61 72 67 00</td>
</tr>
</tbody>
</table>
```

**Hex Dump for args-multiple**

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfffffff0</td>
<td>00 00 00 00-05 00 00 00 c0 ff ff bf</td>
</tr>
<tr>
<td>bfffffff0</td>
<td>da ff ff bf e8 ff ff bf-ed ff ff bf f7 ff ff bf</td>
</tr>
<tr>
<td>bfffffff0</td>
<td>fb ff ff bf 00 00 00-00 00 61 72 67 73 2d 6d</td>
</tr>
<tr>
<td>bfffffff0</td>
<td>75 6c 74 69 70 6c 65 00-73 6f 6d 65 00 61 72 67</td>
</tr>
<tr>
<td>bfffffff0</td>
<td>75 6d 65 6e 74 73 00 66-6f 72 00 79 6f 75 21 00</td>
</tr>
</tbody>
</table>
```
Overview

• Necessities and Organization
• Argument passing in PintOS
• Implementation of System Calls
Implementation of System Calls

1. All 13 syscalls
2. The create syscall
3. The wait syscall
4. The exit syscall
Implementation of System Calls: All 13 syscalls

halt, exit, exec, wait, create, remove, open, filesize, read, write, seek, tell, close.

• When user program calls one of the functions in lib/user/syscall.h, a software interrupt happens and an interrupt frame is created.
• The frame is dispatched to syscall_handler(struct intr_frame* f);
• The type of syscall to execute is stored in (f->esp)

  int sys_code = *(int*)f->esp;
Implementation of System Calls: All 13 syscalls

syscall codes in src/lib/syscall-nr.h
Implementation of System Calls: All 13 syscalls

Parsing the `sys_code` in `src/userprog/sySCALL.c`

```c
static void
syscall1_handler(struct intr_frame* f)
{
    // first check if f->esp is a valid pointer
    if (f->esp is a bad pointer)
    {
        exit(-1);
    }

    // cast f->esp into an int*, then dereference it for the SYS_CODE
    switch(*(int*)f->esp)
    {
        case SYS_HALT:
        {
            // Implement syscall HALT
            break;
        }
        case SYS_EXIT:
        {
            // Implement syscall EXIT
            break;
        }
        (...)  
    }
```
Implementation of System Calls: All 13 syscalls

Extracting arguments from user program and passing to syscalls

```c
static void
syscall_handler(struct intr_frame* f)
{
    switch(*(int*)f->esp)
    {
    case SYS_WRITE:
    {
        int fd = *((int*)f->esp + 1);
        void* buffer = (void*)(*((int*)f->esp + 2));
        unsigned size = *((unsigned*)f->esp + 3);

        //run the syscall, a function of your own making
        //since this syscall returns a value, the return value should be
        //stored in f->eax
        f->eax = write(fd, buffer, size);
    }
    }
}
```
Implementation of System Calls: The create syscall

- All file system calls will be using function from either `src/filesys/file.h` or `src/filesys/filesys.h`
- All file-related syscalls are rather straightforward.

**Prototype for create syscall**

```c
bool create (const char* file, unsigned initial_size)
{
    check to see if valid file pointer
    using synchronization constructs:
    //bool filesystem_create (const char *name, off_t initial_size);
    bool = filesystem_create(file pointer, initial size);
    return bool
}
```
Implementation of System Calls: The wait syscall

\texttt{wait} is the trickiest syscall to implement.

Points of note:

\textbf{Careful interactions between the parent and the child process. Possible scenarios:}

\begin{itemize}
  \item Child becoming an orphan.
  \item Child becoming a zombie.
  \item Resume execution of parent thread after child completes.
  \item Processes sharing their child information to all other processes.
  \item Which situations should \texttt{wait} fail?
  \item etc.
  \item etc........ (much more scenarios need to be self-discovered along the project.)
\end{itemize}
Implementation of System Calls: The exit syscall

PintOS requires a specific format for \texttt{exit} syscall.

\texttt{<thread\_current()\textgreater \rightarrow name>: exit(<exit\ status>)}

For example:

\texttt{Main Thread: exit(1)}
Passing Tests
Passing Tests

1. Setup the stack properly.
2. Implement `process_wait`
3. Implement `write` syscall for `STDOUT_FILENO` with `putbuf`
4. Implement `exit` syscall
References
