Lab 3:: General Description

- Lab 3: “Execution, termination and synchronization of user programs”
  - Handling program arguments
  - Execution of several user programs
  - Termination of a user program
  - Synchronization of shared data structures
  - Wait system call

- We will go through some of the issues one more time!

Lab 3::Main Goals

Tasks:

- Part A
  - Provide synchronization for multiple programs
  - Provide synchronization between programs
    (the most important part of the lab)

- Part B
  - Loading program arguments
    (probably, the most difficult part of the lab)

- File synchronization: Not yet addressed. It is a part of Lab 4

Lab 3::Exec, Exit and Wait (1)

`pid_t exec (const char *cmd_line)`

- Runs the executable whose name is given in `cmd_line`, passing any given arguments, and returns the new process’s program id (`pid`)

- Must return `pid -1`, if the program cannot load or run for any reason (!)

Lab 3::Exec, Exit and Wait (2)

`void exit (int status)`

- Terminates the current user program, returning the exit code `status` to the kernel.
- `status` of 0 indicates success and nonzero values indicate errors
- Remember to free all the resources that will be not needed anymore.
Lab 3::Exec, Exit and Wait (3)

int wait (pid_t pid)

- Provides synchronization between user programs.
- "Parent" process waits until its pid-"child" process dies (if the child is still running) and receives the "child" exit code.
- If the child has been finished, wait() should return child's exit value without waiting.

Lab 3::Exec

pid = process ID
tid = thread ID

Add your implementation of exec() functionalities into process_execute() and process_start() in process.c

Your Exec() system call in syscall.c { process_execute() { 
  ... tid = thread_create 
  ... generate pid from tid; 
  ... wait until start_process(); 
  return pid or -1 
}

Lab 3::Exec

start_process() { 
  loading – DONE! 
  initialization – DONE! 
  putting program arguments into stack is a PART B; 
  signal to process_execute
}

 pid = -1, if the program cannot load or run for any reason.

Use an array or a list to keep track of pid:s.

pid might equal tid, because we have only one thread per process.

Limit the number of user programs (t.ex. 64 or 128).

Lab 3::Exit (1)

The most suitable place for Exit() functionalities is in your implementation of systems calls in syscall.c

Your Exit() system call in syscall.c { get exit code from user; save the exit code if needed. thread_exit }

Exit() must return -1 to the "parent" program if something is wrong, for example, if the child has caused a memory violation.

You should take care of it!

Clean up program’s resources before the exit!

print("%s: exit(%d)%n", thread-name, thread-exit-value) before any exit. (This is needed for testing purposes.)

Lab 3::Exit (2)

/* Free the current process’s resources. */
void process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;

  /* Destroy the current process’s page directory and switch back to the kernel-only page directory. */
  pd = cur->pd;
  if (pd == NULL)
  {
    /* correct ordering here is crucial. We must get cur->pagemap() 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 the process’s page directory, or our active page directory will be one that’s been freed (and cleared). */
    cur->pagemap = NULL;
    pagemap_destroy (pd);
  }
}

Lab 3::Situation with Wait (1)

- "Parent" exits without calling wait() while the "child" is still running
- "Child" exits before the "parent" and:
  - "parent" calls wait() afterwards, or
  - "parent" will exit without calling wait().
- "Parent" calls wait() before the "child" exits.

All the situations above under the condition that the child does not exit normally.

Lab 3::Wait

Once you get pid, just call process_wait() (located in process.c) from Wait() system call:

"wait for thread tid to die and return its exit status. If it was terminated by the kernel (e.g. killed due to an exception), return 0. If PID is invalid or if it was not a process that has already been successfully called for the given TID, return -1 immediately, without waiting."

Steps to accomplish wait():

1. Wait until the exit code of child pid is available
2. Get the exit code and remove it from the system
3. Return the exit code (or -1 if something is wrong)
Lab 3::Situations with Wait (2)

- "Parent" exits without calling `wait()` while the "child" is still running.

Do not store the exit code!

```
Parent
exec(Child) exit(0)
```

```
Child
exit(0)
```

Lab 3::Situations with Wait (3)

- "Child" exits before the "parent" and:
  - "parent" calls `wait()` afterwards.

Wait() returns child's exit value without waiting.

```
Parent
exec(Child) wait(Child) exit(0)
```

```
Child
exit(0)
```

Lab 3::Situations with Wait (4)

- "Child" exits before the "parent" and:
  - "parent" will exit without calling `wait()`.

You should keep child’s exit value until the parent exits
(since the child doesn’t know if the parent calls `wait()` later on)

```
Parent
exec(Child) exit(0)
```

```
Child
exit(0)
```

Lab 3::Situations with Wait (5)

- "Parent" calls `wait()` before the "child" exits.

```
Parent
exec(Child) wait(Child) exit(0)
```

```
Child
exit(0)
```

Lab 3::To Hit The Target!

1. Parts of the functions accessing shared resources **must** be thread safe, e.g. employ synchronization techniques such as locks and semaphores.
2. Particularly, access to global objects and data **must** be synchronized.
3. Only one thread can have access to the **console** at a time. Other threads must wait until completion of reading/writing.

Do it as a homework!
Lab 3::Part B (PB)

Task:
- Load program arguments.

Lab 3::Preparatory Steps (1)

- Lab 3: STEP 3
- into userprog/process.c, find setup_stack()
  - *esp = PHYS_BASE
  - change to: *esp = PHYS_BASE - 12;
- So that you have
  - *esp = PHYS_BASE;

Lab 3::Preparatory Steps (3)

- The user program with arguments should be called with ‘...’ from the Pintos command line:
  pintos --qemu -- run 'some_program arg1 arg2 arg3'
- When the user program with arguments is called from exec(), you have to call it like this:
  exec("some_program arg1 arg2 arg3")
- The implementation of all these things have to be done in start_process();

Lab 3::Implementation (1)

- STEP 1. Parse the string:
- Use strtok_r(), prototyped in lib/string.h
- Read comments in lib/string.c or man page (run man strtok_r)
- Limit the number of arguments (for simplicity)
- STEP 2. Set up the stack:
- Necessary details about setting up the stack for this task you can find in Program Startup Details section of Pintos documentation.

Lab 3::Implementation (2)

- pid = process ID
- tid = thread ID
- Add your implementation of exec() functionalities into process_execute() and process_start() in process.c
- Your Exec() system call in syscall.c { tid = thread_create; } process_execute() { ... eax = process_execute } start_process() { loading – DONE! initialization – DONE! putting program arguments into stack is a PART B; signal to process_execute } pid = -1, if the program cannot load or run for any reason.
  - Use an array or a list to keep track of pid’s.
  - pid might equal tid, because we have only one thread per process.
  - Limit the number of user programs (t.ex. 64 or 128).
Lab 3::PB::Implementation (2)

- some_program arg1 arg2 arg3
- After parsing: some_program, arg1, arg2, arg3
- Place the words at the top of the stack
- Align to 4-byte-words, add 0’s
- Reference the words through the pointers (pointers should point to the addresses of the words in the stack)
- Put the pointers to the stack (followed with NULL pointer)

Lab 3::Test (1)

Now you are ready for a complete check!
Run `gmake check` from `userprog/build`
The following tests should pass:
1) Argument passing when executing:
   - args-none, args-single, args-multiple, args-many, args-dbl-space
2) Different exec-tests:
   - exec-once, exec-arg, exec-multiple, exec-missing, exec-bad-ptr
3) Wait-tests:
   - wait-simple, wait-twice, wait-killed, wait-bad-pid

Lab 4:: General Description

Lab 4: “File System”
- Synchronization of read-write operations
  - One writer writes at a time
  - Many readers can read
- Additional system calls to work with files
  - seek()
  - tell()
  - filesize()
  - remove()
- Creating and removing files without destroying the file system

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Data</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffffe0</td>
<td>argv[4]</td>
<td>0</td>
<td>char*</td>
</tr>
<tr>
<td>0xbffffdc</td>
<td>argv[3]</td>
<td>0xbffffc</td>
<td>char*</td>
</tr>
<tr>
<td>0xbffffd8</td>
<td>argv[2]</td>
<td>0xbffffb</td>
<td>char*</td>
</tr>
<tr>
<td>0xbffffd4</td>
<td>argv[1]</td>
<td>0xbffffa</td>
<td>char*</td>
</tr>
<tr>
<td>0xbffffd0</td>
<td>argv[0]</td>
<td>0xbffffe6</td>
<td>char*</td>
</tr>
<tr>
<td>0xbffffc</td>
<td>argv</td>
<td>0xbffffd0</td>
<td>char**</td>
</tr>
<tr>
<td>0xbffffc8</td>
<td>argc</td>
<td>4</td>
<td>int</td>
</tr>
<tr>
<td>0xbffffc4</td>
<td>return address</td>
<td>0</td>
<td>void (*) ()</td>
</tr>
</tbody>
</table>
Lab 4::Files (1)

- filesys/file.[h|c] - operations on files. A file object represents an open file.
- filesys/filesys.[h|c] - operations on the file system.
- filesys/directory.[h|c] - operations on directories.
- filesys/inode.[h|c] - the most important part of the implementation related to the file system. An inode object represents an individual file (e.g. several open files fd1, fd2, fd3 may belong to one inode "student.txt").

Lab 4::Files (2)

- devices/disk.[h|c] - implementation of the low-level access to the disk-drive.
- filesys/free-map.[h|c] - implementation of the map of free disk sectors.
- These two last ones are not important for you!

Lab 4::Reading/Writing (1)

- Several readers should be able to read from a file at the same time
- Reading should be forbidden if the file content is being changed by the writer
- Only one writer can write to a file at a time
- The writer must not write if at least one reader is reading from the file

Lab 4::Reading/Writing (2)

- Readers should not starve
- Writers can starve
- However, think about solution to avoid the problem of starvation, even though you are not obliged to implement it
- Note that, in this lab assignment, you are not asked to implement dynamic enlargement / reducing of the file size (if you want to write more than the file size currently is)

Lab 4::Reading/Writing (3)

- Ways to implement:
  - Locks/Conditions
  - Semaphores
- One file can be opened several times and processes can attempt to read/write simultaneously at any place in the file
- You need to apply synchronization not to each file structure, but to the entire file!
- Synchronization can be done on one of three levels: system calls, files, and inodes.
- Which level would you choose? Motivate!!!

Lab 4:: Additional System Calls

- void seek (int fd, unsigned position)
  - Sets the current position in the open file fd to position. If the position exceeds the file size, it should be set to the end of file.
- unsigned tell (int fd)
  - Returns the current position in the open file fd.
- int filesize (int fd)
  - Returns the file size of the open file fd.
- bool remove (const char *file_name)
  - Removes the file file_name.
  - Note that the open files must not be deleted from the file system before they are closed. If the file is to be removed, the operating system should wait until the file is closed and only then it can delete it. This functionality has been already implemented!
Lab 4:: Create and Remove

- Creating and removing of files must not lead to destructive consequences to the file system
- Create and remove are writing operations on the directories (filesys/directory/[h|c]).
- Open is the reading operation on the directories
- In principle, synchronization of reading/writing operations on directories should be handled as synchronization of files, but...
- For the sake of making student life more convenient, in this lab assignment, you are advised to handle synchronization of directories as the critical section problem

Lab 4::Final Tests (1)

- The following tests should pass if your implementation is correct in addition to the tests from Lab 2 and Lab 3:
  - tests/filesys/base/lg-create
  - tests/filesys/base/lg-full
  - tests/filesys/base/lg-random
  - tests/filesys/base/lg-seq-block
  - tests/filesys/base/lg-seq-random
  - tests/filesys/base/sm-create
  - tests/filesys/base/sm-full
  - tests/filesys/base/sm-random
  - tests/filesys/base/sm-seq-block
  - tests/filesys/base/sm-seq-random
  - tests/filesys/base/syn-read
  - tests/filesys/base/syn-remove
  - tests/filesys/base/syn-write
  - tests/userprog/close-twice
  - tests/userprog/read-normal
  - tests/userprog/multi-recruse
  - tests/userprog/multi-child-fd

Lab 4::Final Tests (2)

- For testing your readers-writers algorithm, we provide the following user programs: pfs.c, pfs_r.c, pfs_w1.c, and pfs_w2.c.
- These programs try to emulate several readers and writers accessing the same file.
- In order to run these programs, you should do the following steps:
  1. Copy these programs to "examples" directory.
  2. Modify the following line in the "examples/Makefile":
     - PROGS = pfs pfs_r pfs_w1 pfs_w2
  3. Add the following lines to the "examples/Makefile":
     - pfs_SRC = pfs.c
     - pfs_r_SRC = pfs_r.c
     - pfs_w1_SRC = pfs_w1.c
     - pfs_w2_SRC = pfs_w2.c
  4. Run gmake from "examples" directory.
  5. Copy the executables pfs, pfs_r, pfs_w1 and pfs_w2 to the Pintos disk (which is in "userprog/build").
  6. Run pfs in Pintos.

Lab 4::Final Tests (4)

- The result is copied into messages file, which should only contain the word "cool" like this:
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  - cool
  ...