CS350 Operating Systems Winter 2022

A1 - Assignment Specification

1 Introduction

Welcome to the second programming assignment for CS350! In this assignment, you will implement UNIX-like process management system calls for OS/161 as your A1 kernel space programming component. The A1 `userspace` programming assignment objective is to use Linux system calls to implement a mini shell as a Linux `userspace` program.

Some general advice for this assignment:

- **Start early.** The instructions are detailed, but even debugging simple mistakes are time consuming. This holds doubly true if you are not especially familiar with `C`.
- **Compile often.** By checking whether the code compiles after every programming prompt, explicitly specified in the following sections, you will be able to pinpoint problems very quickly. The code is **should** to compile after every properly implemented programming prompt.

In this assignment, for the A1 - OS/161 kernel programming componenet, we will use two prompts.

Explore prompts: guide you towards a better understanding of the OS/161 kernel code.

Programming prompts: give you step by step implementation instructions for implementing the system calls in OS/161. **Read these carefully!**

2 Prelab-A1: Concept Review - Processes and System Calls

In this assignment, you are asked to implement several OS/161 process-related system calls. Before you start implementing system calls, you should review and understand the lecture material about processes and system calls. We briefly summarize the related material in this prelab section.

2.1 Processes and System Calls

A process is in essence a running program. A program is just a file, a sequence of data that holds executable code. In order for the program to actually execute, the OS loads the executable code in memory and designates it as such. The OS also reserves space for the data created and used during execution. The OS then sets up the processor to start executing the program. The processor state

along with this prepared area of memory is called a process. A process also includes OS resources like open files that it uses for I/O.,

Applications can be implemented as a single process or as multiple processes. For multiprocess applications each process can run the same or different programs. Each process always has its own text (code), data and stack, and processes communicate with each other using mechanisms such as shared files, shared memory, and, or signals.

2.2 System Call for Process Creation

In `UNIX` and similar systems like Linux, users cannot create processes from scratch. The only way to create a process is by using the `fork` call from inside another process. The call creates a duplicate of the calling process, copying the text (code), data, and stack of the calling process and storing it for the newly forked process.

The two processes resulting from `fork` have identical stack, text (code), and data. If the processes were completely identical, then the call would not be very useful since both processes would have the exact same input and output. We would also not be able to use `fork` to run different programs. For this reason `fork` has a different return value for each process. The duplicate process created, called the *child*, receives a return value of `0`. The original process that made the call to fork, is called the *parent* and receives a special identifier called a process ID (`PID`) that it can use to refer to the child. Each process has a single parent, but a process can have multiple children.

As a note, there is an exception to the rule that new processes are only created by calling `fork` from existing ones. The kernel itself constructs the first process manually. We call this initial process the *init* process, and most operating systems keep it running for as long as the system is powered on. The init process has no parent, processes thus form a tree with init at its root.

`OS/161` is an exception, since it does not actually have an init process. Rather, the OS/161 kernel acts as a shell and creates a new userspace process for each command. These processes can then create children processes using `fork`.

This was a design decision as `OS/161` is an educational OS with most system calls missing, so that students practice implementing system calls in a kernel, the OS/161 kernel, in this case. Creating an `init` process in OS/161 is impossible without the essential process creation system calls,which you will implement in this kernel-side programming assignment.

2.3 System calls for Process Management

`PIDs` are unique process identifiers that processes can use to communicate with one another. In this assignment, we use `PIDs` from parent processes to find out when a specific child exits. In this way, we can synchronize the different processes of an application. For example, a popular pattern for applications is for a parent process to spawn children that each do a single task, then exit

when they are done. The parent waits for the children to exit, then continues executing with the guarantee that the children processes have finished their tasks.

We use two system calls to implement this pattern: `_exit` in the child process, and `waitpid` in the parent. When a process exits, it calls `_exit` to notify the operating system that it is done. The operating system then frees some of the resources of the process. The `_exit` system call also takes in an integer argument. This argument is typically `0` for success or an error code in case of a runtime error.

In OS/161, we will implement the `waitpid` system call so that, only a parent process can call `waitpid` on one of its children using its `PID`. The `waitpid` system call will only check if child has exited and does not check for other state changes like blocking or continuing execution.This system call will be implemented as a blocking call so it does not return *until* the child exits.

In UNIX-like operating systems, such as Linux, the `waitpid` system call is versatile and changes depending on the passed arguments. For example, `waitpid` can either return immediately if no children have changed state, or it can block until such a state change occurs. A parent can also monitor either a specific child by passing its `PID`, or all children at once by passing `-1` as the `pid` argument.

3 Prelab-A1: Code Review - OS/161

In this assignment, you are asked to implement several OS/161 process-related system calls. Before you start implementing system calls, you should review and understand those parts of the OS/161 kernel that you will be modifying.

This section gives a brief overview of some parts of the kernel that you should become familiar with.

3.1 `kern/syscall`

This directory contains the files that are responsible for loading and running user-level programs, as well as basic and stub implementations of a few system call handlers.

`proc_syscalls.c`: This file is intended to hold the handlers for process-related system calls, including the calls that you are implementing for this assignment. Currently, it contains a partial implementation of a handler for `_exit()` and stub handlers for `getpid()` and `waitpid()`.

`runprogram.c`: This file contains the implementation of the kernel's `runprogram` command, which can be invoked from the kernel menu. The `runprogram` command is used to launch the first process run by the kernel. Typically, this process will be the ancestor of all other processes in the system.

3.2 kern/arch/mips/

This directory contains machine-specific code for basic kernel functions, such as handling system calls, exceptions and interrupts, context switches, and virtual memory.

`locore/trap.c`: This file contains the function `mips_trap()`, which is the first kernel C function that is called after an exception, system call, or interrupt returns control to the kernel. (`mips_trap()` gets called by the assembly language exception handler.)

`syscall/syscall.c`: This file contains the system call dispatcher function, called `syscall()`. This function, which is invoked by `mips_trap()` determines which kind of system call has occured, and calls the appropriate handler for that type of system call. As provided to you, `syscall()` will properly invoke the handlers for a few system calls. However, you will need to modify this function to invoke your handler for `fork()`. In this file, you will also find a stub function called `enter_forked_process()`. This is intended to be the function that is used to cause a newly-forked process to switch to user-mode for the first time. When you implement `enter_forked_process()`, you will want to call `mips_usermode()` (from `locore/trap.c`) to actually cause the switch from kernel mode to user mode.

3.3 kern/include

The `kern/include` directory contains the include files that the kernel needs. The `kern` subdirectory contains include files that are visible not only to the operating system itself, but also to user-level programs. (Think about why it's named "kern" and where the files end up when installed.)

3.4 kern/vm

The `kern/vm` directory contains the machine-independent part of the kernel's virtual memory implementation. Although you do not need to modify the virtual memory implementation for this assignment, some functions implemented here are relevant to the assignment.

`copyinout.c`: This file contains functions, such as `copyin()` and `copyout` for moving data between kernel space and user space. See the partial implementations of the handlers for the `write()` and `waitpid()` system calls for examples of how these functions can be used.

3.5 In user

The `user` directory contains all of the user level applications, which can be used to test OS/161. Don't forget that the user level applications are built and

installed separately from the kernel. All of the user programs can be built by running `bmake` and then `bmake install` in the top-level diretory (`os161-1.99`).

OS/161 comes with a variety of user-level programs that can run on top of the OS/161 kernel. These include standard UNIX-style utility programs, like `ls` and `cat`, and a variety of test programs. The source files for the utility programs are located in `$OS161TOP/os161-1.99/user/{bin,sbin}`, where the symbol `$OS161TOP` refers to the top-level OS/161 directory that was created when you installed OS/161 into your account. The source files for the user-level programs that we use as the test programs are located in `$OS161TOP/os161-1.99/user/{uw-testbin,testbin}`.

Note that many of the user-level programs in `$OS161TOP/os161-1.99/user/{uw-testbin,testbin}` will not run with current OS/161 distribution, since some system calls are not implemented in this version of the OS/161 operating system.

User-level programs are installed under `$OS161TOP/root/` in the `bin` `sbin`,`testbin` and `uw-testbin` directories. You will test your implementation of the system calls using the following OS/161 user-level test programs:

- `uw-testbin/pidcheck`
- `uw-testbin/widefork`
- `testbin/forktest`

4 A1-OS/161 Kernel System Call Implementation Requirements

All code changes for this assignment should be enclosed in `#if OPT_A1` statements. For example:

```
#if OPT_A1
   // code you created or modified for ASST1 goes here
#else
   // old (pre-A1) version of the code goes here,
   //  and is ignored by the compiler when you compile ASST1
   // the ``else'' part is optional and can be left
   // out if you are just inserting new code for ASST1
#endif /* OPT_A1 */
```

For this to work, you must add `#include "opt-A1.h"` at the top of any file for which you make changes for this assignment.

If in Assignment 0 you wrapped any new code with `#if OPT_A0`, it will *also* be included in your build when you compile for Assignment 1.

For this assignment, you are expected to implement the following OS/161 system calls:

- `fork`

- `getpid`
- `waitpid`
- `_exit`

`fork` enables multiprogramming and makes OS/161 much more useful. `_exit` and `waitpid` are closely related to each other, since `_exit` allows the terminating process to specify an exit status code, and `waitpid` allows another process to obtain that code. You are not required to implement the `WAIT_ANY`, `WAIT_MYPGRP`, `WNOHANG`, and `WUNTRACED` flags for `waitpid()` - see `kern/include/kern/wait.h`.

To help get you started, there is a partially-implemented handler for `_exit` already in place, as well as stub implementatations of handlers for `getpid` and `waitpid`. You will need to complete the implementations of these handlers, and also create and implement a handler for `fork`.

There is a man (manual) page for each OS/161 system call. These manual pages describe the expected behaviour of the system calls and specify the values expected to be returned by the system calls, including the error numbers that they may return. **You should consider these manual pages to be part of the specification of this assignment, since they describe the way that that system calls that you are implementing are expected to behave.** The system call man pages are located in the OS/161 source tree under `os161-1.99/man/syscall`. They are also available on-line through the course web page.

Your system call implementations should correctly and gracefully handle error conditions, and properly return the error codes as described on the man pages. This is because application programs, including those used to test your kernel for this assignment, depend on the behaviour of the system calls as specified in the man pages.

Under no circumstances should an incorrect system call parameter cause your kernel to crash.

Integer codes for system calls are listed in `kern/include/kern/syscall.h`. The file `user/include/unistd.h` contains the user-level function prototypes for OS/161 system calls. These describe how a system call is made from within a user-level application. The file `kern/include/syscall.h` contains the kernel's prototypes for its internal system call handling functions. You will find prototypes for the handlers for `waitpid`, `_exit` and `getpid` there. Don't forget to add a prototype for your new `fork()` handler function to this file.

5 A1: OS/161 Kernel Side Programming

In this section, we will guide you through the OS/161 system call programmig assignment for A1.

5.1 Implementing `getpid`

The correct implementation of `getpid` must return the unique PID of the process to userspace. To find out how the call works we use the UNIX `grep` command line tool to find all references to it, in the `kern/` folder of the OS/161 soruce code, which holds the OS161/kernel code.

```
 grep -r getpid .
./include/kern/syscall.h:#define SYS_getpid       5
./include/syscall.h:int sys_getpid(pid_t *retval);
./syscall/proc_syscalls.c:/* stub handler for getpid() system call  */
./syscall/proc_syscalls.c:sys_getpid(pid_t *retval)
./arch/mips/syscall/syscall.c:  case SYS_getpid:
./arch/mips/syscall/syscall.c:    err = sys_getpid((pid_t *)&retval);
```

Listing 1: All occurences of `sys_getpid` in the kernel.

We see that, headers aside, there are only two references to `getpid`, one in `arch/mips/syscall/syscall.c`, and one in `syscall/proc_syscalls.c`. The latter file holds the actual definition of the function `sys_getpid`. The `sys_` prefix denotes the function corresponds to a system call. In the former, the function just gets called by some other function.

Explore: Inspect the occurences of `sys_getpid` in `syscall.c`.

- How is the function used?
- What does the calling function do?
- How is the return value of `getpid()` propagated to userspace?

Hint: What does `struct trapframe` represent? Find its definition.

The `proc_syscalls.c` file holds the definition itself. This is the code we need to change to get `getpid` working.

```
/* stub handler for getpid() system call                */
int
sys_getpid(pid_t *retval)
{
  /* for now, this is just a stub that always returns a PID of 1 */
  /* you need to fix this to make it work properly */
  *retval = 1;
  return(0);
}
```

Listing 2: The scaffolding code for the `getpid()` call.

The starter code right now returns 1 by default. For the call to work it must instead return the `PID` of the function. A natural place to store the `PID` of a process is the `proc` instance the kernel uses to store the metadata of the process.

Explore: What is the relation between `proc` and the process itself? Is the `proc` the process itself? Can we theoretically call `free` on the `proc` and still run the process afterwards?

Here is the `struct` itself:

```
48 /*
49  * Process structure.
50  */
51 struct proc {
52         char *p_name;                   /* Name of this process */
53         struct spinlock p_lock;         /* Lock for this structure */
54         struct threadarray p_threads;   /* Threads in this process */
55
56         /* VM */
57         struct addrspace *p_addrspace;  /* virtual address space */
58
59         /* VFS */
60         struct vnode *p_cwd;            /* current working directory */
61
62 #ifdef UW
63   /* a vnode to refer to the console device */
64   /* this is a quick-and-dirty way to get console writes working */
65   /* you will probably need to change this when implementing file-related
66      system calls, since each process will need to keep track of all files
67      it has opened, not just the console. */
68   struct vnode *console;                /* a vnode for the console device */
69 #endif
70
71         /* add more material here as needed */
72 };
```

Listing 3: The `proc` struct in `kern/include/proc.h`.

The `proc` has the following contents.

- `p_name`: A string holding the name of the process.
- `p_lock`: A lock for the structure to avoid races.
- `p_threads`: An array of threads for the process. We will talk about threads later in the course, but for now suffice to say they hold the CPU state of the process when the process is not running.
- `p_addrspace`: A pointer to the address space of the process. We will talk about address spaces and virtual memory later in the course. For now we treat address spaces as maps that represent the memory of the process.
- `p_cwd, console`: Open `vnode`s for the current working directory and the serial console. They roughly correspond to open files, and we will talk about them when we discuss file systems.

The `proc` structure crucially does not already have a `PID` field.

Programming: Add a new field called `p_pid` to `proc` structure. Return the value of the field `p_pid` of the current process from `sys_getpid`. Hint: The current process is accessed using the variable `curproc`. What should the type of the `p_pid` be?

Now that we have the field, we must initialize it. Right now there is no mechanism to assign `PID`s, so we must make one ourselves. The simplest way to do this is to define a counter that starts from `PID_MIN` and every time a new process is created the counter's value is assigned to the `p_pid` and the counter

is incremented. The counter must be protected using a binary semaphore that acts as a lock, which must be held to read and increment the counter.

Programming: Initialize an integer counter, `pid_count`, and a semaphore, `pid_count_mutex` to control access to it.

- **Look at how `proc_count` and `proc_count_mutex` are being initialized in function `proc_bootstrap` in `kern/proc/proc.c`. Initialize the counter `pid_count` in the same place. Recall what the starting value of `pid_count` should be.**
  Hint: Where is `PID_MIN` defined, how will you use it in `kern/proc/proc.c`.
- **Create and initialize the semaphore `pid_count_mutex` similar to the way `proc_count_mutex` has been created and initialized.**
- **We just added the *definitions* of the counter and the mutex in `kern/proc/proc.c`.**
- **To successfully use the counter and mutex, you will need to add the declaration for the variables `pid_count` and `pid_count_mutex`.**
  Hint: where are `proc_count` and `proc_count_mutex` declared?
  Think: What is the difference between a definition and a declaration in `C`?

The function `proc_create_runprogram`, in `kern/proc/proc.c`, creates new, initialized instances of `proc`. We will initialize `p_pid` for the newly created processes in this function.

Programming: Initialize `p_pid` in `proc_create_runprogram` using the `pid_count` counter.
Hint: Look at how `proc_count` is incremented and how its semaphore, `poc_count_mutex` is used to protect access to it.

- **Similarly, use `pid_count_mutex` to protect access to `pid_count`.**
- **Assign `p_pid` a value from the counter `pid_count`**

Make sure to read the counter after `P()` and before `V()`. Think: What are those functions? Where can you find their definition?

We now have a way to allocate `PID`s to newly created processes, and can implement the `fork` system call to create new processes.

5.2 Implementing `fork`

Explore how `fork` is being used in the kernel using `grep` to search for the system call macro `SYS_fork`.

```
grep -r SYS_fork .
./include/kern/syscall.h:#define SYS_fork         0
```

Using `grep` for `sys_fork`, the expected name for the implementaiton of the system call, returns nothing. So not only is there no implementation of `fork`, the system call number is not used anywhere.
As a first step we will add the stub for the system call in the kernel. We do this in `arch/mips/syscall/syscall.c`. In this file there is the `syscall` function that is the entry point for all system calls in the kernel. We had a cursory look at the `syscall` function during our implementation of `getpid`. Now we will study it more closely.

```
78 void
79 syscall(struct trapframe *tf)
80 {
...
88
89         callno = tf->tf_v0;
...
102         switch (callno) {
103             case SYS_reboot:
104                 err = sys_reboot(tf->tf_a0);
105                 break;
106
107             case SYS___time:
108                 err = sys___time((userptr_t)tf->tf_a0,
109                                  (userptr_t)tf->tf_a1);
110                 break;
...
143         if (err) {
144                 /*
145                  * Return the error code. This gets converted at
146                  * userlevel to a return value of -1 and the error
147                  * code in errno.
148                  */
149                 tf->tf_v0 = err;
150                 tf->tf_a3 = 1;      /* signal an error */
151         }
152         else {
153                 /* Success. */
154                 tf->tf_v0 = retval;
155                 tf->tf_a3 = 0;      /* signal no error */
156         }
157
158         /*
159          * Now, advance the program counter, to avoid restarting
160          * the syscall over and over again.
161          */
162
163         tf->tf_epc += 4;
```

Listing 4: `syscall` function in `syscall.c`

Explore: What does the code *outside* of the `switch` statement do? Hint: What does the trapframe represent? What would be the common functionality needed by every single system call?

Programming: Adding the definition for `sys_fork` in `kern/syscall/proc_syscalls.c`

- **Define the function `sys_fork`.**
- **It should return an integer.**

- **It should take two arguments:**
  - **an integer pointer to `retval`, which is going to be used to return the `PID` of the child to the parent process.**
    Hint: Look at the header of `waitpid` to pass `retval` correctly to `sys_fork`..
  - **and a pointer to the entire trapframe structure that was passed to `syscall`. Hint: The function `syscall` in `arch/mips/syscall/syscall.c` takes an entire pointer to a trapframe structure.**
- **Also make sure to add the `C` declaration to the header file containing all other syscall function declarations. Hint: The header file with other system call declarations is `syscall.h`**

Programming: In the `syscall.c` file in the function `syscall`, add a case in the switch statement for a call to the new `sys_fork` function.

- **Consider what the system call number is for the case for `sys_fork`.**
  Hint: system call numbers are defined in `/kern/include/kern/syscall.h`
- **Notice that `retval` is defined in `syscall` with type `pid_t`. What is the relationship between `pid_t` and `int`?**

The next step is reasoning about how to actually create the new process. We currently have a stub function in which we must create a new `proc` and initialize it so that it is a copy of the calling process.
The first step is to create the `proc` struct. Recall, we can do this with the `proc_create_runprogram` call that allocates and initializes a new `proc` structure. We use this function instead of `proc_create` because it also takes care of initializing fields like `p_cwd` that are not in scope of this programming assignment.

Programming: Call `proc_create_runprogram` to create a new `proc` struct in `sys_fork`.
Hint: Read the `proc_create_runprogram` function to see how to create a `proc` structure.
Hint: The name of the newly created child process should simply be the string `"child"`

We then fill in the struct's fields with information from the calling process (parent). First we create a copy of the caller's address space using `as_copy`, then assign it directly to the new process. The `PID` of the process is already initialized, since we have added the appropirate code in `proc_create_runprogram`.

Programming: Call `as_copy` to copy the address space of the current process and assign it to the newly created `proc` struct, which represents the child process.
Hint: Consider using `grep` to see the declaration, and or definition of `as_copy` and understand its arguments.
Hint: Use `curproc_getas()` to get the address space of the current process.

Programming: Allocate a new trapframe using `kmalloc` and copy the trapframe of `curproc` into it.

- **create a new `trapframe` structure, suppose `trapframe_for_child` that is dynamically allocated using `kmalloc`, so that it is on the heap in the kernel**
- **now copy the contents of the `trapframe` that was passed to `sys_fork` to this `trapframe_for_child`.**

Next step is the trickiest, since we are creating a thread from scratch, but in OS/161 userspace it must look as if the thread just returned from a successful `fork` with a newly forked process. We must also ensure that the new process "sees" a value of `0` as the return value from `fork`, in contrast to the caller process (parent) that "sees" the `PID` of the new child process.
We use the `thread_fork` call to create the new thread., which has the prototype in `kern/thread/thread.c`:

```
477 int
478 thread_fork(const char *name,
479             struct proc *proc,
480             void (*entrypoint)(void *data1, unsigned long data2),
481             void *data1, unsigned long data2)
```

Understanding the `thread_fork` function is not in the scope of this programming assignment. It is sufficient to know that it creates a new thread in the kernel, attaches it to the given `proc`, and starts executing from the `entrypoint` function in the kernel. Our goal is to set the right `entrypoint` function so that the new thread returns to userspace as if returning from the `fork` call.
As per the prototype, `thread_fork` takes the following arguments:

- a `name` to assign to the thread, you can call this thread any name, e.g. (`"child_thread"`),
- the `proc` to which we will attach the thread. This should be the child process we created using `proc_create_runprogram`
- an `entrypoint` function that must take two arguments. We will use `enter_forked_process` as the entrypoint function. A stub for it is defined in `arch/mips/syscall/syscall.c`. It will take two arguments:
  - the trapframe that is identical to that of the parent process

- the `data2` argument is unused so we can set it to `0`

Programming: Call `thread_fork` with the right arguments.

- **Modify the prototype of `enter_forked_process` so that we can pass it as an argument to `thread_fork`.**
  - **The `enter_forked_process` that is currently declared in OS/161 does not take two arguments, but `thread_fork` expects to call a function that does take two arguments.**
  - **Note, that `thread_fork` expects an entry point function to have the following signature: `void (*entrypoint)(void *data1, unsigned long data2)` where `entrypoint` for `thread_fork` from `sys_fork` should be the `enter_forked_process`**

We are almost there - we just need to get `enter_forked_process` to return to userspace immediately after the `fork` system call, with `0` as the return value. To do this we just need to call the function `mips_usermode` which is in `arch/mips/locore/trap.c`. This function loads a trapframe into the CPU and returns to userspace. We must configure the trapframe correctly, for this step to work correctly.
In `enter_forked_process` we have been given a copy of the caller's trapframe. Unfortunately, we cannot pass it directly to `mips_usermode`, because the function expects the trapframe to be in the stack, and the trapframe is currently on the heap. To solve this, we declare a `struct trapframe` in the beginning of the function. Variable declarations in a `C` function are allocated on the stack, so we can now copy the old trapframe, which was passed to `enter_forked_process`, into the new one that was declared in the function. We can then `kfree` on the former.
We must also modify the trapframe according to the `OS/161` ABI to return execution in userspace at the right place and with the right return value. To do this we must

- increment the program counter (`tf_epc`) by 4, the size of a machine word to start executing the instruction after the `fork` system call.
- set the register that holds system calls' return values, `tf_v0` to 0, since the new process is the child.

Programming: Add a function call to `mips_usermode` with a properly initialized trapframe in `enter_forked_process`.

- **remember to declare the `trapframe` struct in the function so that it can go to the kernel stack for the child process.**
- **remeber to modify**

- **tp_epc register to increment program counter by 4**
- **tp_v0 register to return 0 from the fork to the child process**

Theoretically, our implementation of `sys_fork` is complete. However, without essential synchronization primitives, we have to include an addtional instuction at the end of our implementation of `sys_fork`. This is a call to sleep, which forces the parent process to sleep before returning from the system call.

Programming: Add sleep and return from `sys_fork`. Add the following lines, to delay parent process so that there is no contention for output buffer and to indicate a successful return from `sys_fork`.
Hint: to use `clocksleep`, you must include the header file using the include statement: #include <clock.h>

```
clocksleep(1);
return 0;
```

Listing 5: Last lines of `sys_fork`

We have now completed implementing the `fork` and `getpid` system calls. You can test your implementation using the following OS/161 user-level test programs:

- `uw-testbin/onefork`
- `uw-testbin/pidcheck`

You should read the `onefork` and `pidcheck` OS/161 user-level programs to compare whether the output from your program is the expected output from the OS/161 user-level program.

5.3 Implementing `_exit`

We now turn our attention to `_exit` system call. The call does not do much right now. Let's go through it line by line:

```
27   as_deactivate();
28   /*
29    * clear p_addrspace before calling as_destroy. Otherwise if
30    * as_destroy sleeps (which is quite possible) when we
31    * come back we'll be calling as_activate on a
32    * half-destroyed address space. This tends to be
33    * messily fatal.
34    */
35   as = curproc_setas(NULL);
36   as_destroy(as);
```

These function calls effectively destroy the address space, freeing the code (text) and data regions of the process. `as_deactivate` does not do anything currently, but its intended function has to do with the hardware memory management unit (MMU). We will discuss MMUs during the Virtual Memory lecture. We then disassociate the memory from the process using `curproc_setass(NULL)`, then destroy adddress space structure using `as_destroy`.

Note that we are detaching and destroying the memory of the process while the process is still running! This is possible because the system call is in the kernel, and the address space only holds userspace memory. If we tried to return to userspace after destroying the address space, the system would understandably panic.

```
37
38   /* detach this thread from its process */
39   /* note: curproc cannot be used after this call */
40   proc_remthread(curthread);
41
42   /* if this is the last user process in the system, proc_dest
43      will wake up the kernel menu thread */
44   proc_destroy(p);
45
46   thread_exit();
47   /* thread_exit() does not return, so we should never get her
48   panic("return from thread_exit in sys_exit\n");
```

The next set of function calls destroys the process and the thread state. As we mentioned, `proc` holds general state for the process while the thread is the schedulable instance, that we run on the CPU.
Here is a neat trick: We **disassociate** the thread from the process using `proc_remthread`. Note, if the thread is scheduled on a CPU, it can continue execution, but we can safely destroy the process using `proc_destroy` We destroy the thread by using `thread_exit` function call that tells the OS/161 kernel to remove the current executing thread off the CPU.

5.3.1 Initializing and Destroying `proc` structure for `_exit`

According to the `man` pages for `_exit`, we must "cause the current process to exit. The exit code `exitcode` is reported back to other process(es) via the waitpid() call." Therefore, we must provide a way for a parent process to find the `exitcode`. A solution is to include these attributes in the `proc` structure:

- a pointer to an array of children processes
- a pointer to the parent process
- a variable to store the `exitcode`,
- and a variable to indicate the status of the process: running or exited.

Since, we are adding pointers between children and parent processes, there might be instances where more than one process is accessing a `proc` structure simultaneously. Therefore, we will have to use some form of synchronization to protect access to the `proc` structure. We will look into synchronization in much more detail in the next assignment, and for this assignment we will use suboptimal solutions.

Programming: Add to the `proc` struct, the following attributes:

- **an array of processes to hold the children, `p_children`**
  Hint: OS/161 has a structure for arrays, `struct array`

- **a pointer to the parent process, `p_parent`**
- **an `int` field called `p_exitcode` for storing the `exitcode`**
- **an `int` variable to record the `p_exitstatus` of the process as running or exited.**

Programming: Initialize and destroy the newly added `proc` struct attributes:

- **create the array of processes in `proc_create`**
- **destroy the array of processes in `proc_destroy`.**
- **initialize the newly added members to the `proc` struct, such as `p_parent`, `p_exitcode` and `p_exitstatus`**

Hint: Which functions provided by `OS/161` can be used for creating and destroying arrays?

Programming: Set the parent pointer of the child process in `sys_fork` to the current process.
Hint: This can be done right after creating the new child `proc` struct using `proc_create_runprogram`.

5.3.2 Delaying `proc_destroy`

We have changed `sys_exit` to store the exit code in the `proc` structure of the exiting process. However, the `proc` struct is torn down and destroyed by `proc_destroy` at the very end of `sys_exit`. The data is essentially lost. We must delay calling `proc_destroy` if the exiting process has a parent that has not itself exited.

Programming: Delaying `proc_destroy`.

- **remove the `proc_destroy` call at the end of `sys_exit`.**
- **replace it with a piece of code that checks whether the process has a running parent:**
  - **if it does not: call `proc_destroy` to clean up the `proc` structure.**
  - **otherwise, do not destroy the `proc` struct and use `p_exitstatus` to mark the process as having exited and add the exitcode in `p_exitcode`.**
    Note: This will make the kernel hang, since any process with a parent will not be destroyed at this time, and the kernel will not resume.

- **add synchronization: as we allow parent and child processes to access each others' `proc` struct, it might be read or written by multiple threads at once. To ensure a thread does not read inconsitent state, we need to use synchronization primitives.**
  - **We will use the `spinlock_acquire(&p->p_lock)` function call to lock access to the current `proc` struct. So, before the code that checks for a live parent process add the `spinlock_acquire(&p->p_lock)` function call.**
  - **We will include two calls to function `spinlock_release(&p->p_lock)`,**
    - **once before the call to `proc_destroy` and**
    - **once after you set `p_exitstatus` and store the exitcode in `p_exitcode`,if a parent process is alive**

5.3.3 Monitoring children from the parent

We have added a `p_children` field in the `proc` struct, but we have not yet added code to actually add children to the array. We will do this in the `sys_fork` system call.

Programming: In `sys_fork`: add the pointer to the newly created child `proc` into the parent's `p_children` array using `array_add`.
Note: Your kernel will complain that you are calling `aray_cleanup` on an array that is not empty. Since we have not removed the child processes from our children processes array yet.

Now that the array of children is populated, we must free its contents before calling `array_destroy` in `proc_destroy`. The array has pointers to all of the children processes of the process, both running and exited. For the exited children we need to call `proc_destroy`, since the children themselves have already called `_exit`. For the children that are still running, we must set their parent pointer to `NULL`. That way they will call `proc_destroy` on themselves when exiting as per the code we wrote earlier.

Programming: In `sys_exit`, immediately after calling the `as_destroy` function iterate the process' array of children, `p_children`. For each child in the array:

- **create a temporary `proc` struct to copy the current child process in the array, lets call it `temp_child`**
- **remove the child from the array**
- **check the `p_exitstatus` for `temp_child`**
  - **if it indicates that the child has exited: call `proc_destroy` on the child**

  - **otherwise, set its `p_parent` field to `NULL`.**

- **add synchronization: use the process spinlock to protect against concurrent accesses to the children. We must call `spinlock_acquire(&temp_child->p_lock)` before inspecting the `p_exitstatus` field, and call `spinlock_release(&temp_child->p_lock)` right before `proc_destroy` or right after setting `p_parent`.**

At this point, your kernel should not hang. We have now completed implementing the `_exit`. You can test your implementation using the following OS/161 user-level test programs:

- `uw-testbin/pidcheck`

You should read the `pidcheck` OS/161 user-level programs to compare whether the output from your program is the expected output from the OS/161 user-level program.

5.4 Implementing `waitpid`

The `waitpid` system call lets a parent process wait for a child process to exit, then returns the child's exit code to the parent process. The stub code is shown below:

```
 62 /* stub handler for waitpid() system call                */
 63
 64 int
 65 sys_waitpid(pid_t pid,
 66              userptr_t status,
 67              int options,
 68              pid_t *retval)
 69 {
 70   int exitstatus;
 71   int result;
 72
 73   /* this is just a stub implementation that always reports an
 74      exit status of 0, regardless of the actual exit status of
 75      the specified process.
 76      In fact, this will return 0 even if the specified process
 77      is still running, and even if it never existed in the first place.
 78
 79      Fix this!
 80   */
 81
 82   if (options != 0) {
 83     return(EINVAL);
 84   }
 85   /* for now, just pretend the exitstatus is 0 */
 86   exitstatus = 0;
 87   result = copyout((void *)&exitstatus,status,sizeof(int));
 88   if (result) {
 89     return(result);
 90   }
 91   *retval = pid;
 92   return(0);
 93 }
 94
```

The signature of the function is identical to that of the system call in userspace. The first argument is the `PID` of the child to be monitored. Recall, in UNIX the `PID` can have a value of `-1`, in which case the system call monitors the first child to change state. In OS/161, we assume the `PID` is always larger than `0`.
The second argument, `status`, is a pointer to a userspace variable. The system call will write out the status of the child into that variable. Since `sys_waitpid` is run by the kernel, we cannot use a userspace pointer directly. We instead use the `copyout` function that copies memory from the kernel to userspace.
We don't use the third argument, `options`, in this assignment. The argument allows the user to pass flags to the call to change its behavior. One such flag is `WNOHANG`, which configures the call to exit immediately if no children have exited yet. The default behavior is for the call to block until a child exits.
The `retval` argument is a kernel pointer to which we write the `PID` we ended up waiting on, or `-1` if there was an error. Since we always wait on a specific child, in our case this argument will have a value of `pid`.

Explore: Why is `retval` a kernel pointer and not a userspace pointer like `status`?

- How do we actually communicate the `PID` back to userspace?
- Consider where `sys_waitpid` is called and how `retval` is used.
- What is the number and size of variables the ABI allows us to pass directly to userspace?

Our requirements for implementing `waitpid` are thus the following:

- Go through the process' list of children to find the one with the right `PID`. If it is not there, return an error. If it is, remove it from the array.
- If the child has not exited yet, wait until it does.
- Once the child has exited, get its exit code.
- Destroy the child's `proc` struct.
- Return the exit code to userspace through the `status` argument.

Programming: Search for the correct child process.

- **iterate through the array of children, examining each `p_pid` against the `PID` passed into `sys_waitpid`.**
- **If we do not find the `PID` we exit with an appropriate error message, refer to the `man` pages to find the relevant error codes.**
- **If we do find the child `proc`,**
  - **copy the child pointer from the array into a temporary `proc` struct, say `temp_child`**

- **remove the child `proc` pointer from the array**
- **break out of the loop, with a valid copy of the `temp_child` `proc` struct.**

Next, we have to check if the child is running or exited. If it has not already exited, we have to wait until it exits. Normally, we would use a mechanism called a **condition variable** for this purpose, but we have not implemented this feature in OS/161 yet. Instead, we will use a technique called **busy polling**. With busy polling the process checks at regular intervals whether a condition is true. In case it is not true, it sleeps for a preset amount of time like 1 second, wakes up, and checks the condition again. If the condition has turned true the process breaks out of the loop.
We will implement busy polling, using the spinlocks in the `proc` struct. We will learn more about them in Threads and Synchronization lectures. Therefore, we provide you with the code, you need to add in `sys_waitpid` to implement busy polling. The code must be placed after we have found the child process and copied it into a temporary `temp_child` `proc` struct.

```
#include <clock.h>

...

spinlock_acquire(&temp_child->p_lock);
while (!temp_child->p_exitstatus) {
     spinlock_release(&temp_child->p_lock);
     clocksleep(1);
     spinlock_acquire(&temp_child->p_lock);
}
spinlock_release(&temp_child->p_lock);
```

Listing 6: Busy Polling in `sys_waitpid`

Programming: Add busy polling to in `sys_waitpid` to wait for the correct child `proc` to change `p_exitstatus` to `exited`.
Hint: Read the code to understand what the value of `p_exitstatus` should be when a process exits.

Once a child has exited, we can retrieve the exit code from the `proc` structure and store it into a local variable. We can then destroy the child `proc` struct. Remember that we removed `proc_destroy` from `exit` for this exact reason.

Programming: Extract the `p_exitcode` into the local varaible `exitstatus` and call `proc_destroy` for the `temp_child` child `proc` struct.

Finally, we pass the return value to userspace. We do this by passing the exit code to the local `exitstatus` variable that is passed to `copyout` in the inital stub code.
However, we do not pass the exit code itself. The `waitpid` ssytem call can be used to monitor a lot of other status changes, e.g. if a child process was stopped or started running, and `status` argument in `waitpid` encodes both the event that happened and a possible return value. The `_MKWAIT` family of macros is

used for bit manipulations to encode both the `exitcode` and the event that happened in a single variable. In the scope of this progrmaming assignment, `waitpid` will only check for process termination. Therefore, we can simply pass the `exitcode` to `_MKWAIT_EXIT` before assigning it to `exitstatus`.

Programming: Pass the `exitcode` to `_MKWAIT_EXIT` and then assign it to `exitstatus`.

And we're done!
To test your implementation using the following OS/161 user-level test programs:

- `uw-testbin/pidcheck`
- `uw-testbin/widefork`
- `testbin/forktest`

To test your implementation, use `user/testbin/forktest`, `user/uw-testbin/onefork`, and `user/uw-testbin/waitpid`. If the tests pass, your implementation is correct. Again, read the test program to compare your output with the expected output.

6 A1-`userspace` Programming Requirements

In A1-`userspace` you are required to use process and file management system calls to implement a simple Linux shell in `C` capable of supporting:

redirection: The shell should support redirection of input and, or output.

execution of a sequence of programs that communicate through a pipe: For example, if the user types `command1 | command2 | command3`. The output from the execution of one `command1`, should be used as input to the `command2` that is following.

You will need to use several Linux system calls that were introduced in the lectures such as: `fork`, `exec`, `open`, `close`, `pipe`, `dup2` and `wait`.
The most useful resource will be the `man` pages for these system calls in a Linux or UNIX-like computing environment. You can use the manual pages for a system call, by typing

```
man 2 SYSTEM_CALL
```

to identify the parameters, description and return values for that `SYSTEM_CALL`

Your progrmaming assignment should be implemented in `C` language and your program should in `my_mini_shell.c`. When your mini shell runs, its prompt should be just the symbol `$` followed by a single space. Here is a sample of the compilation and execution of your mini shell.

```
dafault_shell_display_prompt >gcc -o myminish my_mini_shell.c
dafault_shell_display_prompt >./myminish
$ ls
my_mini_shell.c myminish myshell output.txt sample.txt
$
```

7 Submitting Your Work

To submit your work, you must use the `cs350_submit` program in the `linux.student.cs` computing environment.
Important! You must use `cs350_submit`, not `submit`, to submit your work for CS350.

Note the usage for `cs350_submit` command is as follows

```
% usage: cs350_submit <assign_dir> <assign_num_type>
```

The `assign_dir` is the path to the root folder of the programming assignment. For the A1-kernel side programming assignment, the `assign_dir` is the path to your `os161-1.99` folder.
The `assign_num_type` for the kernel side is `ASST2`.

Note: There is no mistake here, we will use the ASST2 kernel to build and test this A1-OS/161 kernel side programming assignemnt. By using the `ASST2` kernel configuration, we will remove some warnings that are part of synchronization primitives kernel, ASST1. Also, note that if you used the `#if OPT_A1` statements, they will be automatically included in the ASST2 kernel. Similarly, if you are using the cs350-container, you will use `ASST2` to build, and test the kernel for this progrmaming assignment, e.g. `build_kernel ASST2`.

For the `userspace` programming assignment, the `assign_dir` is the root directory for the `userspace` programming assignment. The `userspace` programming assignment root directory should contain the `my_mini_shell.c` program. The `assign_num_type` for the `userspace` is `ASSTUSER1`.
Therefore, to run the `cs350_submit` command for submitting the A1-`userspace` programming assignment the command will look like this:

```
% cs350_submit cs350-student/a1 ASSTUSER1
```

The argument `assign_dir` in the `cs350_submit` command, packages up your OS/161 kernel code or `userspace` program, respectively, and submits it to the course account using the regular `submit` command.
This assignment only briefly summarizes what `cs350_submit` does.
Look carefully at the output from `cs350_submit`. It is a good idea to run the `cs350_submit` command like this:

```
cs350_submit cs350-student/a1 ASSTUSER1 | tee submitlog.txt
```

This will run the `cs350_submit` command and also save a copy of all of the output into a file called `submitlog.txt`, which you can inspect if there are problems. This is handy when there is more than a screen full of output.
You may submit multiple times. Each submission completely replaces any previous submissions that you may have made for this assignment.

8 Optional: Writing a Script

This section is optional.
It gives you a brief lesson on writing and running shell scripts. You may be interested in writing a shell script to run commands to test your implementation of the OS/161 Kernel Here is a simple script to run some of the OS/161 user-level test programs to test your kernel implementation for A1.

```
#!/bin/sh

OS161ROOT="$HOME/cs350-os161/root"
CONF="sys161.conf"
KERNEL="kernel-ASST0"

OLDDIR="$PWD"
cd "$OS161ROOT"

sys161 -c $CONF $KERNEL "p uw-testbin/pidcheck;p uw-testbin/pidcheck;q;"
sys161 -c $CONF $KERNEL "p uw-testbin/widefork;q"
sys161 -c $CONF $KERNEL "p testbin/forktest;q"

cd $OLDDIR
```

You can copy the simple script file to your linux student environment. Make the script executable with the `chmod +x filename` command, where `filename` is the name of the script file with the `.sh` extension. The you can simply run the script using `./filename` to run the script.
You should read the script and note the path to `OS/161ROOT`, the `conf` file used and the `kernel` version. The `OS/161ROOT` path may be different for you and you should edit that path, as necessary. This script is using the `ASST0 kernel`, the kernel you built for porgramming assignment zero. As you implement features from A1, you will build and test with the `kernel-ASST1`.