ADVANCED OPERATING SYSTEMS
Milestone 5: User-space Process Management
Fall Term 2014


1 Overview

In this milestone, you will implement functionality in your system for starting and managing multiple processes from user-space.

Note that, from now on, each milestone will build on your existing code base. From this point forward, it’s important not to get too far behind with the implementation!

2 Getting prepared

For this milestone you should set the milestone variable in Config.hs to 5. This should disable the in-kernel endpoint setup code as well as enable some little bits of code in user-space.

If your user-level threads don’t work yet you may want to modify lib/barrelfish/threads.c:1006 to not use user-level threads for the main function of programs other than init.

3 Starting applications from user-space

Up to this point, you have started all your applications from kernel space. In this step you will modify your system to start new processes from user-space.

This step poses the question of how we start the first process if we’re starting processes from user-space in general. To solve this problem, you should use the method we’ve used so far and start the first process (init) from kernel-space. Once the first process is up, it should start any other processes itself (e.g. memeater). For this step, the goal is to be able to correctly start a process from init.

At a high level, starting a process from user-space will involve

- Creating an initial cspace layout for the new process
- Finding the appropriate ELF image.
- Loading the ELF image in both parent and child address space.
- Setting up the cspace for the new process
• Adding a dispatcher for the kernel to do scheduling/running.

You might want to use one or more of the following functions for the process creation procedure outlined above:

• spawn_map_module
• spawn_arch_load
• elf_allocate

You will notice that some of these functions call various paging_* functions which will mean that if your self-paging implementation isn’t complete yet, now is the time to rectify this. In particular, if you haven’t implemented paging_map_fixed_attr yet, you should seriously consider doing so now.

You can have a look into the functions spawn_load_with_bootinfo and bootstrap in the following reference code (from upstream Barrelfish) to see more details of what is happening during the process creation procedure in “standard” Barrelfish.

You also need to think about how to pass the address-space created by the parent process to the child process so that the child process will be aware of where exactly its ELF image is located.

We recommend avoiding this issue (at least in your initial code) by assuming in the child process that all virtual addresses below 1GB (where the ELF image is typically located) are already mapped and the process will not have to handle any page-faults in the area below 1GB. This way, the parent process can set up the ELF image in the first 1GB of address space and start the child process. This child process then takes over its address space and does self-paging for space above 1GB, and we don’t have to pass the address-space layout between two user-space applications.

3.1 Hints

We recommend that you comment out all code that does LMP channel management in init and the application (i.e. memeter) to simplify testing and debugging. We will redo the LMP channel initialization (in a much simplified form) in the next step.

Once you have set up the address space for the child process, you can call spawn_run to make the application runnable. After this call, you should be able to see the output from both the child and parent process (using debug_printf for the moment).

4 LMP channel setup, take 2

In this step, you will re-establish the communication between init (or the memory server) and a newly created process so that that process can do self-paging.

If you’ve set the milestone variable in Config.hs to 5, the code that was initializing the first few endpoints in the kernel should be disabled (you may have to explicitly rebuild the kernel by deleting armv7/kernel in the build directory).

You should now add the following function call early in the code for the init process in order to create the self endpoint for init again:

```c
// Create our endpoint to self
err = cap_retype(cap_selfep, cap_dispatcher, ObjType_EndPoint, 0);
if (err_is_fail(err)) {
    DEBUG_ERR(err, "Failed to create our endpoint to self");
}  ```
// bail out, there isn’t much we can do without a self endpoint.
abort();
}

Now, as init will work as a process manager, you should integrate the setting up of initial endpoints and LMP channels with creating a new process.

Once the capability space (cspace) for a child is created, init should copy a capability for its own endpoint into the newly created process’s cspace so that it can be used by this child process to establish communication with init. Once the communication channel is established, you should be able to reuse your code from the previous milestones to enable self-paging within the newly started process.

5 A process manager service

Until now, we have started processes from the kernel, and – more recently – init. The reason for this is that init has all capabilities to access the memory region where the multiboot image (and hence all ELF images) is located. Since we don’t support a file system (yet), we need access to this boot-image to load the ELF images for new processes. For security reasons, we want to limit access to the boot-image memory, and also limit who can start new processes.

To do this you will implement a new service, called a process manager, which will start new processes on behalf of other processes (just like the memory server manages physical memory on behalf of other processes).

This service will have the following RPC calls as an interface:

```c
errval_t aos_rpc_process_spawn(struct aos_rpc *chan, char *name, domainid_t *newpid);
errval_t aos_rpc_process_get_name(struct aos_rpc *chan, domainid_t pid, char **name);
errval_t aos_rpc_process_get_all_pids(struct aos_rpc *chan, domainid_t **pids, size_t *pid_count);
```

Note that the name passed to the process_spawn call will not include the prefix “armv7/sbin”. The other two RPC calls are there to give you (and us) a standard interface for process management tools (think ps).

The semantics are as follows: process_get_name takes a process id and returns the name of that process and process_get_all_pids returns an array of the process ids of all currently running processes. See the provided patch for more detailed interface specifications.

As always, it’s up to you whether you want to implement the process manager service as part of the init process or as a stand-alone process that will get the necessary data from init on startup. For simplicity in the instructions we will assume that the process manager is implemented as part of the init process.

If an application needs to start another application it will have to make a call to the RPC interface shown above, indicating which process it wants to start (i.e., the name of the ELF binary in the boot image) and the process manager should then load the ELF image of the requested application (if it exists) and create a new process (using the steps outlined above) and respond to the application that sent the request with a success/failure notification which should include a process id for the new process if the spawning procedure was successful.

Now you should be able to start processes by sending requests to do so to the process manager from any process that is already running.

**EXTRA CHALLENGE**

Implement support for a special communication channel between the newly created process and the process which requested the creation. This direct communication link should allow communication between
parent and child process directly without any involvement from the process manager (apart from initial setup).

**EXTRA CHALLENGE**
Instead of implementing the process manager service inside the init process, implement it in separate application which will handle process creation requests. You can use start this application from the init process using the mechanism shown in Starting applications from user-space.

### 6 Accepting input

In this step, you will add functionality to your system in order to specify which application will be able to accept input from user-space.

User-space applications should be able to accept input over the serial port if they want to. You will need to find a way of arbitrating between applications that want to access serial input. Note that at any given point there should be **at most** one process that is able to read characters from serial input.

### 7 Extending the shell application from milestone 4

You should now extend your command line interface from the last milestone to be able to spawn any program that’s included in the multiboot image. Below there’s a few suggestions for such programs:

- Application to turn on LED
- Application to turn off LED
- Application to demonstrate/stress self-paging (M5)
- Application to demonstrate multiple threads (M4)

Additionally you should implement a command `ps` that lists all running commands.

**EXTRA CHALLENGE**
Implement the functionality that is needed to start processes in the background (we suggest that your command line interface uses the standard shell convention of adding a `&` after the name of the application you want to start in the background).

A process started in the background should not be able to accept input until and unless it is brought to foreground (when it will own the console).

**EXTRA CHALLENGE**
Implement a functionality to support process management commands like `kill`.

### 8 Lab demonstration

You are expected to demonstrate the following functionality during the lab-session:

- You can start an application from user-space.
• Applications started from user-space work correctly with self-paging.
• New application can be started by sending a message to a process manager service.
• A shell which can run applications and has a `ps` command.

Once your code is demonstrated, you should submit it over Online submission system (accessible only from ETHZ network) before midnight on Friday night / Saturday morning.