Phase 1: The Kernel

1 Overview

For this first phase of the operating system project, you will implement low-level kernel functions including process creation, process termination, and low-level CPU scheduling. This phase provides the building blocks needed by the other phases, which will implement more complicated process control functions, inter-process communication primitives, and device drivers.

2 Processes

Phase 1 of the kernel implements five primary routines that are used to control processes: **`fork1`**, **`join`**, **`quit`**, **`zap`**, and **`is_zapped`**. The first three should be relatively familiar. The **`fork1`** function creates a new process that calls a specified function as the entry point and returns its process ID (PID). The process that calls **`fork1`** is the parent process, and the created process is the child process. It will be important for your kernel library to keep track of the parent-child relationships between processes. A parent process waits for one of its children to call **`quit`** by calling **`join`**. The **`join`** function returns the PID and the status of the child that called **`quit`**. Processes that call **`quit`** terminate and must return a status value to their parent. The status value is passed in a parameter to the **`quit`** function.

Calling **`quit`** only kills the process that called it. A process cannot kill another process directly. But a process can arrange for another to be killed by calling **`zap`** and specifying the victim process's PID; subsequent calls to **`is_zapped`** by the victim will return a non-zero value. The idea is that all "zapped" processes eventually call **`quit`**. Most of the zapping functionality will be handled in a later phase of the project where all user-level processes that have been zapped call **`quit`** at the end of all system call handlers and all zapped kernel-level server processes will check **`is_zapped`** and call **`quit`** at the top of their infinite loops. For now, in Phase 1, you must implement the functions **`zap`** and **`is_zapped`**.

2.1 Phase 1 Interface Definition

The Phase 1 library must implement a set of functions that create an interface. The interface is used by the test programs to run processes and verify the operation of the kernel.

```
int fork1(char *name, int (*func)(char *), char *arg, int stacksize, int priority)
```

Creates a child process executing function **`func`** with a single argument **`arg`** and with the indicated priority and **`stacksize`** (in bytes). The name parameter is a descriptive name for the process; it should not be longer than **`MAXNAME`** (defined in phase1.h) characters or will be **`NULL`** if no argument is being passed to the child. You may assume a maximum of **`MAXPROC`** processes (defined in phase1.h).

Return Values:

-2: stacksize is less than **`USLOSS_MIN_STACK`** (defined in usloss.h)

-1: no empty slots available in the process table, or priority out-of-range, or **`func`** is **`NULL`**, or name is **`NULL`**

≥ 0: PID of the created process

```
int join(int *status)
```

This operation synchronizes the termination of a child with its parent. When a parent process calls **`join`** it is blocked until one of its children has called **`quit`**. The **`join`** function returns immediately if a child has already **`quit`** and has not already been **`join`**ed. . The **`join`** function returns the PID of the child process that **`quit`** and stores the status value passed to **`quit`** by the child in the location pointed to by status. The **`join`** function returns information about children in the order in which they **`quit`**.

Return values:

-2: the process does not have any children

-1: the process was zapped while waiting for a child to **`quit`**

≥ 0: the PID of the child that **quit**

void quit(int status)

This operation terminates the current process and facilitates the return of **status** to a **join** by its parent. A process that has children cannot call **quit**; if this happens the kernel should print an error message and call **halt(1).** The returning of a process's start function (the function indicated by the func parameter to fork1) means that the process is done and should have the same effect as calling **quit** (this is the reason for the launch function in the provided the skeleton.c file).

int zap(int pid)

This operation marks a process as being zapped. Subsequent calls to **is_zapped** by that process will return 1. The **zap** function does not return until the zapped process has called **quit**. The kernel should print an error message and call **halt(1)** if a process tries to **zap** itself or attempts to **zap** a non-existent process.

Return values:

-1: the calling process itself was zapped while in **zap**

0: the zapped process has called **quit**

int is_zapped(void)

The function returns a Boolean value indicating whether a process is zapped.

Return values:

0: the process has not been zapped

1: the process has been zapped

int getpid(void)

Returns the PID of the currently running process.

`void dump_processes(void)`

This routine should print process information to the console. For each PCB in the process table print (at a minimum) its PID, parent's PID, priority, process status (e.g. unused, running, ready, blocked, etc.), # of children, CPU time consumed, and name. No format is necessary, but make it look nice.

`int block_me(int new_status)`

This operation blocks the calling process. The **`new_status`** parameter is the value used to indicate the status of the process in the dump processes command. The **`new_status`** parameter must be larger than 10; if it is not, then halt USLOSS with an appropriate error message.

Return values:

-1: if the process was zapped while blocked

0: otherwise

`int unblock_proc(int pid)`

This operation unblocks the process with **`pid`** that had previously been blocked by calling **`block_me().`** The status of that process is changed to READY, and it is put on the Ready List. The dispatcher will be called as a side-effect of this function.

Return values:

-2: if the indicated process was not blocked, does not exist, is the Current process, or is blocked on a status less than or equal to 10. Thus, a **`zap`** or **`join`** blocked process cannot be unblocked with this function call.

-1: if the calling process was zapped

0: otherwise

`int read_cur_start_time(void)`

This operation returns the time (in microseconds) at which the currently executing process began its current time slice.

`void time_slice(void)`

This operation calls the dispatcher if the currently executing process has exceeded its time slice. Otherwise, it simply returns.

`int readtime(void)`

This operation returns the CPU time (in milliseconds) used by the current process. This means that the kernel must record the amount of processor time used by each process. Do not use the clock interrupt to measure CPU time as it is too coarse-grained; use **`sys_clock`** instead.

`void dispatcher(void)`

The dispatcher function is the heart of the kernel and contains the logic for process scheduling and context switching. This function is called when it's time, or potentially time, to change processing to another thread.

2.2 Support for Later Phases

The functionality that we implement in Phase 1 provides the foundation for future phases. In fact, each phase builds on the previous one by adding features to this OS. In this layered approach, it is important that each layer has a well-defined interface. The interface functions described in the previous section define most of the Phase 1 interface. But there are a few functions that must be developed to support Phase 2.

`fork1`: your **`fork1`** routine must call **`p1_fork(int pid)`** passing it the PID of the newly created process. **`p1_fork`** must be called before the new process runs for the first time.

quit: your **quit** routine must call **p1_quit(int pid)** with the PID of the process that has **quit**.

dispatcher: your dispatcher should call **p1_switch(int old, int new)** with the PID's of the process that was previously running and the next process to run. You will enable interrupts before you call **context_switch**. The call to **p1_switch** should be called just before you enable interrupts.

For phases 1 through 4, the bodies of **p1_fork, p1_quit,** and **p1_switch** will be empty. You may, if you wish, put debugging output statements in these functions; they are useful in determining when a process is created, when it starts/resumes executing, etc. The provided p1.c file contains sample debug output. These debugging statements should be turned off when you turn in your code. Note: turn in p1.c with your Phase 1 code, even if you have not changed it!

2.3 CPU Scheduling

Your dispatcher must implement a round-robin priority scheduling scheme with preemption. That is, the dispatcher should select for execution the process with the highest priority, and the currently running process is preempted if a higher-priority process becomes runnable. Processes within a given priority are served round-robin. Use 80 milliseconds as the quantum for time-slicing. New processes should be placed at the end of the list of processes with the same priority.

There are five priorities for all processes except for the sentinel process (see below). The priorities are numbered one through five, with one being the highest priority and five the lowest.

The priority of a process is given as an argument to **fork1** by its parent (see above).

3 Kernel Protection

The functions provided by this level of the kernel may only be called by processes running in kernel mode. The kernel must confirm this and in case of execution by a user mode process, should print an error message and invoke **halt(1)**.

With one exception, all data structures needed in the kernel (and in your later phases as well) should be statically allocated. The one exception occurs when allocating a stack to a new

process. Thus, the only place in this phase (and in the entire project) where you will call malloc is when allocating stack space for a new process.

4 Interrupts

Interrupts will be handled by code that you will write in Phase 2. However, you will need to write a small interrupt handler for the clock interrupt for this phase. USLOSS invokes interrupt handlers with two parameters: the first is the interrupt number, and the second is the unit of the device that caused the interrupt. The clock interrupt number is defined in usloss.h. Since there is only one clock device, the unit number is zero.

The Interval Timer of USLOSS will be used both for CPU scheduling (enforcing the time slice) and implementing the pseudo-clock. For Phase 1, the CPU scheduling function is the only one that you will need to implement. The pseudo-clock will be implemented in Phase 2

5 Deadlock

The kernel should detect certain very simple deadlock states. For Phase 1, this simply means the kernel will need to determine when all processes have terminated and it is time to halt the simulator. The simplest way of doing this is to use a sentinel process that is always runnable and has the lowest priority in the system (six). Thus, the sentinel will only run when there are no other runnable processes. The sentinel executes an infinite loop in which it verifies that other processes exist and there is not a deadlock using a function named **`check_deadlock()`**. If **`check_deadlock`** determines that there are processes blocked on i/o devices, the sentinel will call **`waitint()`**. If **`check_deadlock`** determines there are no processes blocked on i/o devices, then terminate the simulation with a normal message “All processes completed”, or an abnormal message “Sentinel detected deadlock”.

For Phase 1, there are no i/o devices (they will appear in Phase 2). Thus, the **`check_deadlock`** function determines if all processes have **`quit`** (normal termination of USLOSS - **`halt(0)`**) or if processes remain (abnormal termination of USLOSS - **`halt(1)`**).

6 Initial Startup

Your kernel will begin execution in a function called **`startup()`**. This function is called by the USLOSS simulator library, which has **`main()`**. The **`startup`** function is the entry point into your Phase 1 code. The function will initialize the necessary data structures, create the sentinel process, and create a single process running the function **`start1`** in kernel mode with interrupts enabled. This initial process should be allocated 320 Kbytes of stack space and should run at priority one. The **`start1()`** function is the entry point into test cases for Phase 1 (and will become the entry point for the Phase 2 code).

7 Writing and Testing the Kernel

To help you get started, I have provided a code package that is available to download at D2L course site. You will find skeleton.c, kernel.h, and Makefile in this directory. You are not required to use them, but they might give you some ideas to get started. Testing the kernel is your responsibility. The test files for Phase 1 can be found in the subfolder *testcases* in the provided code package. You can copy it to your phase1 directory, then compile it and load it along with the rest of your kernel and the simulator. You are encouraged to run test cases as you implement specific features of Phase 1, allowing you to test the code as you develop it. You may want to create more test cases to completely test all aspects of your kernel.

8 Submitting Phase 1 for Grading

For the turn-in, you will need to submit all the files that make up your kernel. Design and implementation will be considered, so make sure that your code contains insightful comments and that variables and functions have reasonable names, no hard-coded constants, etc. You may, if helpful, turn in a README file to help me understand your kernel.

Your makefile must be arranged so that typing 'make' in your directory will create an archive named libphase1.a. Your makefile must also have a target called 'clean' which will delete any generated files, e.g. .o files, .a files, core files, etc. Your makefile should have a definition for TESTS= that defines a list of test cases, and a target for $(TESTS): that allows the creation of executable test cases (see the provided makefile for an example). We will use the directory of the test cases of Phase 1 as the TESTS= definition. You can also add your own test cases to this

definition. You should NOT turn in any files that I provide, e.g. libusloss.a, usloss.h, phase1.h, etc., nor should you turn in any generated files.