Phase 1: The Kernel

Complete program due: Monday, September 21st, 9:00 pm

1.0 Overview

For this first phase of the operating system, you will implement low-level process support, including process creation and 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, interprocess communication primitives, device drivers, and virtual memory.

2.0 Kernel Protection

The functions provided by this level of the kernel may only be called by processes running in *kernel* mode. The kernel should confirm this and in case of execution by a user mode process, should print an error message and invoke **`USLOSS_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 phases 2, 3, and 4) where you will call **`malloc`** (or **`memalign`**) is when allocating stack space for a new process.

3.0 Processes

Phase 1 of the kernel implements five routines for controlling processes: **`fork1`**, **`join`**, **`quit`**, **`zap`**, and **`isZapped`**. The first three should be relatively familiar. **`fork1`** creates a new process running a specified function, and returns its process ID (PID). The process that called **`fork1`** is called the *parent*, and the new process created is called the *child*. A parent process waits for one of its children to call **`quit`** by calling **`join`**. **`join`** returns the PID and the status of the child that called **`quit`**. Calling **`quit`** causes the process to die and return the given status to its parent.

Note that calling **`quit`** only kills the process that called it. A process cannot kill another process directly, e.g. by marking its process control block (PCB) as unused. The process being killed may have been context-switched in the middle of doing something, i.e., in a critical section of the code. For this reason, a process arranges for another to be killed by calling **`zap`** and specifying the victim process's PID; subsequent calls to **`isZapped`** by the victim will return a nonzero value. The idea is that all "zapped" processes eventually call **`quit`**; this will be handled by a later phase of the project, in which all user-level processes that have been zapped call **`quit`** at the end of all interrupt handlers. Additionally, all zapped kernel-level server processes will check **`isZapped`** and call **`quit`** at the top of their infinite loops. All of these details will be handled in later phases. All Phase 1 does is implement **`zap`** and **`isZapped`**.

```
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 stack size (in bytes). The **`name`** parameter is a descriptive name for the process; it should not be longer than **`MAXNAME`** (defined in *phase1.h*) characters. The **`arg`** parameter will contain a string no longer than **`MAXARG`** (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 in the process table, or **priority** out-of-range, or **func** is **NULL**, or **name** is **NULL**.
>=0: PID of created process.

int join(int *status);

This operation synchronizes termination of a child with its parent. When a parent process calls **join**, the parent is blocked until one of its children has called **quit**. **join** returns immediately if a child has already quit and has not already been joined. **join** 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**. **join** returns information about children in the order in which they quit.

Return values:

-2: the process does not have any children who have not already been joined.
-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 returns 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 **USLOSS_Halt(1)**. A process's start function (the function indicated by the **func** parameter to **fork1**) returning means that the process id done, and should have the same effect as calling **quit** (this is the reason for the **launch** function in the provided *skeleton.c* file).

int zap(int pid);

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

Return values:

-1: the calling process itself was zapped while in **zap**.
0: the zapped process has called **quit**.

int isZapped(void);

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.), number of children, CPU time consumed, and name. No particular format is necessary, but make it look nice.

3.1 Support for Later Phases

In general, in a layered design such as this one, you should not have to refer to global variables or data structures from a previous phase. To achieve this, some additional phase 1 functions are needed to support phase 2 and phase 5 operations.

For use in phase 2, you will need the following four functions:

`int blockMe(int new_status);`

This operation will block the calling process. **`new_status`** is the value used to indicate the status of the process in the **`dump_processes`** command. **`new_status`** must be greater than 10; if it is not, then halt USLOSS with an appropriate error message.

Return values:

-1: if process was zapped while blocked.
0: otherwise.

`int unblockProc(int pid);`

This operation unblocks process **`pid`** that had previously been blocked by calling **`blockMe`**. 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 process that is zap-blocked or join-blocked cannot be unblocked with this function call.
-1: if the calling process was zapped.
0: otherwise.

`int readCurStartTime(void);`

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

`void timeSlice(void);`

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

In phase 5, you will add features to your operating system that will need to know when a process is created, quit, and/or context switched. The details are not important at the moment. Your Phase 1 process routines have to invoke routines when these events occur. For this phase, these routines will not actually do anything; in phase 5, you will fill in the needed functionality.

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 **USLOSS_ContextSwitch**. The call to **p1_switch** should be called just before you enable interrupts.

For phases 1 thru 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 determing 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 turnin your code. Note: turnin *p1.c* with your phase 1 code, even if you have not changed it!

3.2 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 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).

```
int readtime(void);
```

Return 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 **USLOSS_Clock** instead.

4.0 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.

5.0 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 **`checkDeadlock`**. If **`checkDeadlock`** determines that there are processes blocked on i/o devices, the sentinel will call **`waitint`**. If **`checkDeadlock`** determines there are no processes blocked on i/o devices, then terminate the simulation with a normal message "All processes completed", or an appropriate abnormal message.

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

6.0 Initial Startup

Your kernel will begin execution in a function called **`startup`**. This function is called by the USLOSS simulator (which has **`main`**) and is the entry point into your phase 1 code. Your **`startup`** function will initialize the necessary data structures, create the sentinel process, and create a process running the function **`start1`** in kernel mode with interrupts enabled. This initial process should be allocated 32 Kbytes of stack (hint: see the USLOSS manual about stack growth direction), and should run at priority one. The **`start1`** function will be the entry point into test cases for phase 1 (and will become the entry point for the phase 2 code).

7.0 Writing and Testing the kernel

To help you get started, we have provided a code skeleton and include file in */home/cs452/fall15/phase1/*. You will find *skeleton.c*, *kernel.h*, *p1.c*, 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. To give you an idea of how you should do this, a set of test cases can be found in */home/cs452/fall15/phase1/testcases*. You can copy (or link) the testcases directory to your own *phase1* directory. The provided *Makefile* contains targets for each of the provided testcases. You will need to create more test cases to completely test all aspects of your kernel, and to help you in debugging. You are encouraged to develop test cases as you implement specific features of phase 1, allowing you to test the code as you develop it. Sharing test cases with other groups is permitted and encouraged.

8.0 Submitting Phase 1 for Grading

Note: Programs are submitted via D2L only.

Before submitting to D2L, you will need to create a **`phase1.tgz`** file. There is a target in the provided phase1 **`Makefile`** (*/home/cs452/fall15/phase1/Makefile*) named **`submit`**. It is at the very end of the **`Makefile`**.

Doing make submit will create a **`phase1.tgz`** file that contains:

- The .c files for your phase1. This will include **`phase1.c`**, **`p1.c`**, and any other **`.c`** files that you have created. If you have created additional **`.c`** files, you should have already modified **`COBJS`** to include these additional **`.o`** files.

- The **`.h`** files for your phase1. This will include **`kernel.h`**. If you have created other **`.h`** files, you should have already modified **`HDRS`** to include the additional **`.h`** file(s).
- The **`Makefile`**.

You will *not* submit the usloss library or directory. We will put a link for usloss into the grading directory after we extract your files. The same is true for the testcases directory; do not submit the testcases. We will add a link for the testcases directory.

You are encouraged to create the **`phase1.tgz`** file, then copy it to a different area of your home directory, extract the file using **`tar xvfz phase1.tgz`** and do a compile to see if everything has been included.

Any file(s) that are missing from your submission that we have to request from you will result in a reduction of your grade!

To submit the **`phase1.tgz`** file to D2L

- Log on to D2L, and select CSc 452.
- Select "Dropbox".
- You will find *Phase1*, and *Phase1-late*. Only one of these will be active. Click on the active one — will be *Phase1* if you are turning in the work on time.
- You should now be at the page: "Submit Files - Phase1". Click on the "Add a File" button. A pop-up window will appear. Click on the "Choose File" button. Use the file browser that will appear to select your file(s). Click on the "Upload" button in the lower-right corner of the pop-up window.
- You are now back at "Submit Files - Phase1". Click on "Upload" in the lower-right of this window. You should now be at the "File Upload Results" page, and should see the message **`File Submission Successful`**.

D2L will send you a confirmation email after each item is placed in the Dropbox. This email is sent to your UA email (which is <yourUANetId>@email.arizona.edu). Retain these emails at least until you have received a grade for the assignment; it is your confirmation that you did submit the program(s).

You may submit phase1 more than once. We will grade the last one that you put in the Dropbox.