Phase 1: The Kernel

January 24, 2017

Part A due: February 7, 10:00 pm

Part B due: February 21, 10:00 pm

1. Overview

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

For this phase (and subsequent phases) of the project you are expected to work in groups of two. You may switch groups after each phase of the project.

This phase is split into two parts (sub-phases), A and B. Part B extends Part A; however, they are due separately and will be graded separately. This document describes functionality of the complete phase. In Section 11 the requirements for the individual parts are described.

2. Kernel Protection

The functions provided by 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 invoke `USLOSS_IllegalInstruction` which should cause the offending process to quit via `P1_Quit`.

3. Processes

Phase 1 of the kernel implements three routines for controlling processes: `P1_Fork`, `P1_Join`, and `P1_Quit`; and three routines for getting information about processes: `P1_GetPid`, `P1_GetState` and `P1_DumpProcesses`. `P1_Fork` creates a new process running a specified function, and returns its process ID (PID). The process that called `P1_Fork` is called the *parent*, and the created process the *child*. `P1_Quit` causes the current process to stop running (exit). A parent process calls `P1_Join` to wait for one of its children to call `P1_Quit`, at which time `P1_Join` returns the PID and the status of the child that called `P1_Quit`.

```
int P1_Fork(char *name, int (*func)(void *), void *arg,
     int stackSize, int priority, int tag)
```

Creates a child process executing function `func` with a single argument `arg`, and with the indicated priority, stack, and stack size (in bytes). The `name` parameter is a descriptive name for the process. You may assume a maximum of `P1_MAXPROC` processes (defined in *phase1.h*). The `tag` is either 0 or 1, and is used by `P1_Join` to wait for children with a matching tag.

Return Values:

-4:	invalid tag
-3:	invalid priority
-2:	stacksize is less than `USLOSS_MIN_STACK` (see *usloss.h*)
-1:	no more processes
>= 0:	PID of created process

`void P1_Quit(int status)`

This operation terminates the current process and returns `status` to `P1_Join` called by its parent. Should a parent process quit its child becomes an *orphan*, meaning it no longer has a parent process. An orphan that quits does not return `status` to its parent, because it does not have a parent. Should a process's initial function return (the function in the `func` parameter to `P1_Fork`), it should have the same effect as calling `P1_Quit`. Hint: this is best achieved by putting a wrapper function around `func` that detects when `func` returns and calls `P1_Quit`. Another hint: the child must wait (hint: use a semaphore) until its parent calls `P1_Join` so that the parent can get the status from the child.

`int P1_Join(int tag, int *status)`

This operation synchronizes termination of a child with its parent. The `tag` specifies the tag of the child to be waited for. When a parent process calls `P1_Join` waits for one of its children with a matching tag to exit children exits (hint: use a semaphore). `P1_Join` returns immediately if a child has already exited and has not already been joined. `P1_Join` returns the PID of the child process that exited and stores in `status` the status parameter the child passed to `P1_Quit.` Note: the tag functionality will be used in Phase 2 to allow user-level processes to wait for user-level processes and kernel-level processes to wait for kernel-level processes, respectively.

Return values:

-1:	the process doesn't have any children with a matching tag
>= 0:	the PID of the child that quit
0:	success

`int P1_GetPID(void)`

Returns the PID of the currently running process.

`int P1_GetState(int PID)`

Returns the state of the indicated process.

Return values:

-1:	invalid PID
0:	the process is running
1:	the process is ready

2: (not used)

3: the process has quit

4: the process is waiting on a semaphore

`void P1_DumpProcesses(void)`

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

3.1 Phase 3 Support

One of the parameters to USLOSS_ContextInit is the page table for the process. We will be using this functionality in Phase 3 of the project. To facilitate this, `P1_Fork` should call `P3_AllocatePageTable` to get the page table for the process, and `P1_Quit` should call `P3_FreePageTable` to free the page table. We will provide you with a `p3stubs.c` file that define dummy versions of these functions for use in phases 1 and 2.

4. CPU Scheduling

The OS dispatcher must implement a round-robin priority scheduling scheme with preemption. That is, the dispatcher should select the process with the highest priority for execution, and the currently-running process is preempted if a higher-priority process becomes runnable. Processes within a given priority are served round-robin. The quantum used for time-slicing is 80 milliseconds (four clock ticks). 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 Section 8). 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 `P1_Fork` by its parent (see Section 3).

`int P1_ReadTime(void)`

Return the CPU time (in microseconds) 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; read the time from the USLOSS clock device instead.

5. Semaphores

Semaphores are used to synchronize between processes, and between processes and interrupt handlers.

```
int P1_SemCreate(char *name, unsigned int value, P1_Semaphore
   *sem)
```

This operation creates a new semaphore named `name` with its initial value set to `value`. and returns a pointer it in `sem`. You may assume a maximum of `P1_MAXSEM` semaphores.

Return values:

-2:	`P1_MAXSEM` semaphores already exist.
-1:	duplicate semaphore name
0:	success

`int P1_SemFree(P1_Semaphore sem)`

Free the indicated semaphore. If there are processes blocked on the semaphore then an error message is printed and `USLOSS_Halt(1)` is called.

Return values:

-2:	processes are blocked on the semaphore
-1:	the semaphore is invalid
0:	success

`int P1_P(P1_Semaphore sem)`

Perform a P operation on the indicated semaphore.

Return values:

-1:	the semaphore is invalid
0:	success

`int P1_V(P1_Semaphore sem)`

Perform a V operation on the indicated semaphore.

Return values:

-1:	the semaphore is invalid
0:	success

`char *P1_GetName(P1_Semaphore sem)`

Returns the name of the semaphore, NULL if `sem` is invalid.

6. Interrupts

The OS must implement interrupt handlers for all devices supported by USLOSS. Processes will synchronize with interrupt handlers through the `P1_WaitDevice` routine; this routine causes the process to wait on a semaphore associated with the device until the device's interrupt handler V's the same semaphore. The interrupt handler for a device should also save the contents of the device's status register; this is the I/O operation's *completion status* that allows the process that is waiting for the I/O to determine if the I/O completed successfully. It is only necessary to save the most recent completion status for each device. In the next phase the OS will implement *device drivers*, processes that handle the I/O devices and therefore wait for I/O completion. There will be only one device driver process associated with a device, and hence only one process will ever call `P1_WaitDevice` for a particular device.

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 (except for the syscall interrupt, in which case it's the argument passed to `USLOSS_Syscall`; see below).

The Interval Timer of USLOSS will be used both for CPU scheduling (enforcing the time slice) and implementing the *pseudo-clock*. The pseudo-clock is nothing more than a special semaphore that is V'ed by the kernel every 100 milliseconds. Once again, in the next phase the OS will implement a clock device driver that will wait on the semaphore and provide a general process delay facility. For now, however, the clock semaphore will continually be V'ed by the interrupt handler.

The illegal instruction exception occurs when a `USLOSS_IllegalInstruction` is called. Its interrupt handler should print an error message and call `P1_Quit` to cause the current process to quit.

`int P1_WaitDevice(int type, int unit, int *status)`

Perform a `P` operation on the semaphore associated with the given `unit` of the device `type`. The device types are defined in *usloss.h*. The appropriate device semaphore is `V`'ed every time an interrupt is generated by the I/O device, with the exception of the clock device which should only be `V`'ed every 100 ms (every 5 interrupts). This routine will be used to synchronize with a device driver process in the next phase. `P1_WaitDevice` returns the device's status register in `*status`. Note: the interrupt handler calls `USLOSS_DeviceInput` to get the device's status. This is then saved until `P1_WaitDevice` is called and returned in `*status`. `P1_WaitDevice` does not call `USLOSS_DeviceInput`.

Return values:

-3:	the wait was aborted via `P1_WakeupDevice`
-2:	invalid type
-1:	invalid unit
0:	success

`int P1_WakeupDevice(int type, int unit, int abort)`

Causes `P1_WaitDevice` to return. If abort is non-zero then `P1_WaitDevice` returns that the wait was aborted (-3), otherwise `P1_WaitDevice` returns success (0). The interrupt handlers for the devices should call `P1_WakeupDevice` with `abort` set to zero to cause `P1_WaitDevice` to return.

Return values:

-2:	invalid type

-1:	invalid unit
0:	success

7. System Calls

System calls will be implemented in Phase 2. In Phase 1 the handler for the `USLOSS_SYSCALL_INT` interrupt simply prints an error message of the form "System call %d not implemented." then invokes `USLOSS_IllegalInstruction` to kill the offending process.

8. Deadlock

The kernel should detect certain very simple deadlock states. If all of the existing processes are blocked on semaphores *not* associated with I/O devices or the clock, the system should print an error message and call `USLOSS_Halt(1)`. The kernel should also call `USLOSS_Halt(0)` when the last useful process in the system terminates. The simplest way of accomplishing both of these 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 isn't a deadlock, then calls `USLOSS_WaitInt` to wait for an interrupt. If no other processes exist the sentinel calls `P1_Quit(0)`.

9. Phase 2 Startup

After the kernel has completed initialization it should create a process running the function `P2_Startup` in kernel mode with interrupts enabled. This initial process should be allocated four times the USLOSS minimum stack size, and should run at priority two.

10. Testing

Testing the kernel is your responsibility. We will provide a sample test files to get you started, but you will want to create more than one test program to completely test all aspects of your kernel.

11. Submitting Phase 1 for Grading

Phase 1 is done in two parts, A and B. For both parts you are to submit the file phase1.c via D2L. Design and implementation will be considered, so make sure your code contains insightful comments, variables and functions have reasonable names, no hard-coded constants, etc. You should NOT turn in any files that we provide, e.g. the USLOSS source files, *`phase1.h`*, etc., nor should you turn in any generated files.

11.1. Part A

Part A consists of all the process-related functions in Section 3 *except* for `P1_Join`. As a result, `P1_Quit` does not need to wait for its parent to call `P1_Join`. The dispatcher should implement a "run to completion" scheduling policy, in which the current process runs until it calls `P1_Quit` or a higher priority process becomes runnable.

11.1. Part B

Part B consists of the entire phase as described in this document.

12. Project Groups

For this phase (and subsequent phases) of the project you are expected to work in groups of two. Once you have formed a group send us a private post on Piazza and tell us the group's email addresses so we can keep track of who is working with whom.