Phase 1: Processes

Version 1.3

September 21, 2019

Phase1a due: September 17, 10:00 pm

Phase1b due: September 24, 10:00 pm

Phase1c due: October 1, 10:00 pm

Phase1d due: October 8, 10:00 pm

1. Project Overview

You will implement a complete operating system in three phases, each building on the previous phases. Each phase is composed of four parts, each of which builds on the previous parts. You will be provided binaries for each part so that you can complete subsequent parts even if you don't complete a particular parts.

The architecture of the system is as shown:

Applications
Phase 3: Virtual Memory
Phase 2: Support
Phase 1: Processes
USLOSS

At the top level are the applications, which will typically consist of a test program. At the bottom is USLOSS, which is provided. You will write the three middle levels, starting with Phase 1.

2. Phase 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.

Each phase is divided into four parts: a, b, c, and d. Each part uses functionality from the previous parts; however, they are due separately and will be graded separately. You will be provided a correct solution for each part so that a poor grade on one part does not affect your grade on subsequent parts.

3. Naming Conventions

C only provides two namespaces -- a single global namespace and a namespace that is local to each source file. There is no provision for namespaces that span source files, e.g. a namespace that is confined to a single phase that has multiple source files. To avoid naming conflicts the following naming convention will be used throughout the project. Identifiers with the prefix `Foo_` are implemented by the `Foo` phase and can be used by other phases. Identifiers with the prefix `Foo` (note the lack of an underscore) are internal to the `Foo` phase and may only be used by other functions in that phase. Thus the function `P1_Fork` may be used by Phases 2 and 3, whereas the routine `P1ContextCreate` may only be used by other functions in Phase 1.

4. Phase 1a : Context Management

Phase 1a manages USLOSS contexts for the subsequent sub-phases. You may assume a maximum of `P1_MAXPROC` contexts (defined in *phase1.h*). The functions in this part must be called in kernel mode. If a function is invoked in user mode it should invoke `USLOSS_IllegalInstruction` which should print an error message and call `USLOSS_Halt(0)`.

`void P1ContextInit(void)`

Initializes this part. Must be called before the other functions.

`int P1ContextCreate(void (*func)(void *), void *arg, int stackSize, int *cid)`

Creates a new context. The `func` parameter is the function at which the context should start running, `arg` is a parameter to `func`, and `stackSize` is the size of the stack in bytes. A unique identifier is returned in `*cid`. The identifier must be in the range [0, `P1_MAXPROC`).

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, `P1ContextCreate` should call `P3_AllocatePageTable` to get the page table for the context, and `P1ContextFree` 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.

Return Values:

`P1_TOO_MANY_CONTEXTS`: no more contexts are available

`P1_INVALID_STACK`: stacksize is less than `USLOSS_MIN_STACK`

`P1_SUCCESS`: success

`int P1ContextSwitch(int cid)`

Switches from the currently running context (if there is one) to the context indicated by `cid`.

Return Values:

`P1_INVALID_CID`:	`cid` is not valid
`P1_SUCCESS`:	success

`int P1ContextFree(int cid)`

Frees the context indicated by `cid`. Must call `P3_FreePageTable` to free the page table (see description in `P1ContextCreate` above).

Return Values:

`P1_INVALID_CID`:	`cid` is not valid
`P1_CONTEXT_IN_USE`:	`cid` is the currently running context
`P1_SUCCESS`:	success

`int P1DisableInterrupts(void)`

Disables interrupts and returns the previous interrupt state.

Return Values:

`TRUE`:	interrupts previously were enabled
`FALSE`:	interrupts previously were disabled

`void P1EnableInterrupts(void)`

Enables interrupts.

5. Phase 1b : Process Management

Phase 1b implements processes using Phase 1a. The functions in this part must be called in kernel mode. If a function is invoked in user mode it should invoke `USLOSS_IllegalInstruction` which should cause `P1_Quit` to be invoked with a status of 1024 (note that this is different functionality than Phase 1a).

`void P1ProcInit(void)`

Initializes this part. Must be called before the other functions.

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

Creates a child process executing function `func` with a single argument `arg`, and with the indicated `priority`, `tag`, and `stackSize`. The `name` parameter is a descriptive name for the process that must be unique and no more than `P1_MAXNAME` characters. You may assume a maximum of `P1_MAXPROC` processes (defined in *phase1.h*). The priority must be in the range [1,6], with 1 being the highest priority and 6 being the lowest. Only the first forked process may have priority 6, all subsequent processes must have a priority in the range [1,5]. The `tag` is either 0 or 1, and is used by `P1_Join` (see Phase 1d) to wait for children with a matching tag. If `func` returns it should have the same effect as calling `P1_Quit` (see below). Hint: this is best achieved by putting a wrapper function around `func` that calls `P1_Quit` when `func` returns.

The dispatcher in Phase 1b should run a process until either it becomes unrunnable, quits, or creates a higher priority process. As a result, if the process created via `P1_Fork` has higher priority than the currently running process then the dispatcher should run the new process. By default the first forked process has a higher priority than the currently running non-process and should start running immediately.

Return Values:

`P1_INVALID_TAG`:	invalid tag
`P1_INVALID_PRIORITY`:	invalid priority
`P1_INVALID_STACK`:	stacksize is less than `USLOSS_MIN_STACK`
`P1_DUPLICATE_NAME`:	name already in use
`P1_NAME_IS_NULL`:	name is NULL
`P1_NAME_TOO_LONG`:	name is longer than `P1_MAXNAME`
`P1_TOO_MANY_PROCESSES`:	no more processes
`P1_SUCCESS`:	success

`void P1_Quit(int status)`

Terminates the current process. The `status` is saved so that it can be returned to its parent via `P1GetChildStatus`. If the current process has any children they are adopted by the first process created. If the first process calls `P1_Quit` while it still has children print “First process quitting with children, halting.” to the console and call `USLOSS_Halt(1);`

`int P1GetChildStatus(int tag, int *pid, int *status)`

Returns information about the next child with the specified `tag` that has already quit. The PID of the child is returned in `*pid` and the status it passed to `P1_Quit` is returned in `*status`.

Return Values:

`P1_INVALID_TAG`:	invalid tag
`P1_NO_CHILDREN`:	the process has no children with the specified `tag`
`P1_NO_QUIT`:	the process has children, but none have quit
`P1_SUCCESS`:	success

`int P1_GetProcInfo(int pid, P1_ProcInfo *info)`

Returns information about process `pid`. See *usloss.h* for details.

Return Values:

`P1_INVALID_PID`:	invalid pid
`P1_SUCCESS`:	success

`int P1SetState(int pid, P1_State state, int sid)`

Sets the state of the process with PID `pid` to state `state.` The following are the valid values for `state`:

`P1_STATE_READY`:	process `pid` is now ready (runnable)
`P1_STATE_JOINING`:	process `pid` is waiting for a child to quit
`P1_STATE_BLOCKED`	process `pid` is is not runnable
`P1_STATE_QUIT`	process `pid` has called `P1_Quit`

If the state is `P1_STATE_BLOCKED` then `sid` is the ID of the semaphore on which it is blocked (see Phase 1c).

Return Values:

`P1_INVALID_PID`:	invalid `pid`
`P1_INVALID_STATE`:	`state` is not one of those listed above
`P1_CHILD_QUIT`:	`state` is `P1_STATE_JOINING`, but a child has quit
`P1_SUCCESS`:	success

`void P1Dispatch(int rotate)`

Runs the highest-priority runnable process. If the current process has the highest priority of the runnable processes and `rotate` is `FALSE`, the current process continues to run. Otherwise, if the current process is the highest priority and `rotate` is `TRUE` then the current process is moved to the end of the ready queue and the next runnable process of the same priority is run. If there are no runnable processes print "No runnable processes, halting." to the console and call `USLOSS_Halt(0);`

`int P1_GetPID(void)`

Returns the PID of the currently running process.

3. Phase 1c : Semaphores

Phase 1c implements semaphores that are used to synchronize processes. The functions in this part must be called in kernel mode. If a function is invoked in user mode it should invoke `USLOSS_IllegalInstruction.`

`void P1SemInit(void)`

Initializes this part. Must be called before the other functions.

`int P1_SemCreate(char *name, unsigned int value, int *sid)`

This operation creates a new semaphore named `name` with its initial value set to `value` and returns a unique identifier for it in `*sid` . You may assume a maximum of `P1_MAXSEM` semaphores, and the identifier must be in the range [0,P1_MAXSEM).

Return values:

`P1_DUPLICATE_NAME`:	name already in use
`P1_NAME_IS_NULL`:	name is NULL
`P1_NAME_TOO_LONG`:	name is longer than `P1_MAXNAME`
`P1_TOO_MANY_SEMS`:	no more semaphores
`P1_SUCCESS`:	success

`int P1_SemFree(int sid)`

Free the indicated semaphore.

Return values:

`P1_BLOCKED_PROCESSES`:	processes are blocked on the semaphore
`P1_INVALID_SID`:	the semaphore is invalid
`P1_SUCCESS`:	success

`int P1_P(int sid)`

Perform a P operation on the indicated semaphore. The calling process is blocked until the value of the semaphore becomes positive.

Return values:

`P1_INVALID_SID`:	the semaphore is invalid
`P1_SUCCESS`:	success

`int P1_V(int sid)`

Perform a V operation on the indicated semaphore.

Return values:

`P1_INVALID_SID`:	the semaphore is invalid
`P1_SUCCESS`:	success

`int P1_GetName(int sid, char *name)`

Returns the name of semaphore `sid` in `*name`. The parameter name must contain a pointer to a buffer of at least `P1_MAXNAME+1` bytes.

Return values:

`P1_INVALID_SID`:	the semaphore is invalid
`P1_NAME_IS_NULL`:	name is NULL
`P1_SUCCESS`:	success

4. Phase 1d : Interrupts and Booting

Phase 1d installs interrupt handlers for the devices, boots the entire Phase 1, and creates the initial process for Phase 2. The functions in this module must be called in kernel mode. If a function is invoked in user mode it should invoke `USLOSS_IllegalInstruction`.

4.1. Interrupts

Phase 1d implement interrupt handlers for all USLOSS devices. *Device driver* processes created in Phase 2 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 must 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.

The device interrupt handlers should call `USLOSS_DeviceInput` to get the device's status, then call `USLOSS_WakeupDevice` to wake any device driver waiting via `P1_WaitDevice`. The clock device driver is a bit of a special case. It should only call `USLOSS_WakeupDevice` every 5th interrupt so that the clock device driver only wakes up every 100ms rather than every 20ms. The interrupt handler should also call `P1Dispatch(TRUE)` every 4th interrupt so that the dispatcher time-shares between the runnable processes with a quantum of 80ms.

USLOSS invokes device interrupt handlers with two parameters: the first is the interrupt number, and the second is the unit of the device that caused the interrupt. Your interrupt handler can use this information in handling the interrupt.

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.

The illegal instruction exception occurs when a `USLOSS_IllegalInstruction` is called. Its interrupt handler should print an error message and call `P1_Quit` with a status of 1024 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:

`P1_WAIT_ABORTED`:	the wait was aborted via `P1_WakeupDevice`
`P1_INVALID_TYPE`:	invalid type
`P1_INVALID_UNIT`:	invalid unit
`P1_SUCCESS`:	success

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

Causes `P1_WaitDevice` to return. If `abort` is TRUE then `P1_WaitDevice` returns that the wait was aborted (`P1_WAIT_ABORTED`), otherwise `P1_WaitDevice` returns success (`P1_SUCCESS`) with the status passed to this function. The interrupt handlers for the devices should call `P1_WakeupDevice` with `abort` set to zero to cause `P1_WaitDevice` to return successfully.

Return values:

`P1_INVALID_TYPE`:	invalid type
`P1_INVALID_UNIT`:	invalid unit
`P1_SUCCESS`:	success

4.2. Startup and Sentinel

Unlike the previous three parts Phase 1d should define the function `startup` that is invoked when USLOSS starts up. This function should initialize all the parts, then call `P1_Fork` to create the first process, the sentinel. The sentinel runs at the lowest priority, 6, and is the only process to run at that priority. The sentinel is always runnable, so the dispatcher always has a process to run. The sentinel creates the initial process for Phase 2, `P2_Startup`. `P2_Startup` should run in kernel mode with interrupts enabled, a stack that is `USLOSS_MIN_STACK*4`, priority 2, and tag 0.

After forking `P2_Startup` the sentinel goes into a loop waiting for all of its children to quit, at which point it prints "Sentinel quitting." and quits. It checks the status of its children via `P1GetChildStatus`.

4.3. Join

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

This function synchronizes a parent with a child that has quit. The `tag` must match the tag specified when the child was forked. When a process calls `P1_Join` it returns the next child with a matching tag to call `P1_Quit`, waiting if necessary. Children are returned by `P1_Join` in the order in which they called `P1_Quit`. `P1_Join` stores in `*pid` the PID of the child and in `*status` the status 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. For this phase just ensure that `P1_Join` only returns processes that were created via `P1_Fork` with the same tag.

Return values:

`P1_NO_CHILDREN`:	the process doesn't have any children with a matching tag
`P1_SUCCESS`:	success

10. Getting Started and Testing

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

11. Submission

Submit your solutions via GradeScope. Only one partner should submit and should indicate via the GradeScope submission process the name of their partner. 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 (e.g. .o files, core files, etc.)

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.

Changes

Version 1.1

- `P1ContextCreate` returns an `int`.
- `P1DisableInterrupts` was missing.

Version 1.2

- Timer interrupt handler calls `P1Dispatch(TRUE)`.
- `P1ContextFree` returns `P1_CONTEXT_IN_USE` if `cid` is currently running.
- Phase 1a `USLOSS_IllegalInstruction` should print an error message and call `USLOSS_Halt(0)`.
- Removed deadlock detection from sentinel.
- Renamed "modules" to "parts".

Version 1.3

- Added `P1_Join`.
- Added GradeScope turn in instructions.