CS 452 (Spring 23): Operating Systems

Phase 1 - Process Control

milestone 1a due at 5pm, Thu 2 Feb 2023
milestone 1b due at 5pm, Thu 9 Feb 2023

NOTE NOTE NOTE
Unlike all of the other Phases, Phase 1 is broken into two "milestones," with different due dates. This spec describes **milestone 1b** (the second one). See the other spec to understand **milestone 1a.**
NOTE NOTE NOTE

1 Semester Overview

This semester, you will be writing some of the major features of an operating system kernel. The various components of the kernel are separated into various libraries; each "Phase" of the project implements a different library.

The various phases will be strictly separated from each other; at **no time** can they share variables! Their only interaction will be through function calls (which you can think of a bit like "methods" in an object-oriented system).

Moreover, when we test each Phase, we will have instructor-provided libraries that give the full implementation for each previous phase. For this reason, it is **critical** that you implement the functions exactly as stated in the spec - otherwise, your code will not be able to interact with the instructor code, and you will fail all of the testcases.

1.1 Phase List

The project will have 5 Phases, as follows:

- **Phase 1 - Process Control**

  Keeps track of what processes exist, what their names are and what the parent/child relationships are. Handles "zapping" (which is our kernel's equivalent of the `kill` operation from UNIX).

  Blocks and unblocks processes as requested by higher-level code (must be implemented in Phase 1, but won't be used until Phase 2), and handles all dispatching decisions.

- **Phase 2 - Messages and Interrupt Handlers**

  Implements "mailboxes,“ which are data structures that keep track of messages. It uses the functions from Phase 1 to block and unblock processes as the processes send, receive, and wait for messages.

  In addition, this Phase implements the basic mechanisms for interrupts and system calls. While you will write interrupt handlers for all of the devices that USLOSS supports (clock, terminals, and disks), you will not write the actual device drivers themselves - that happens in Phase 4. Instead, in this Phase, you will write the interrupt handler for each device, which turns a device interrupt into a message.

- **Phase 3 - User Processes and Syscalls**

  Implements a set of system calls, which allow user-mode code to make requests of the kernel.

- **Phase 4 - Device Drivers**

  Adds device drivers for the clock, terminal, and disk devices. You will also implement system calls that allow the user-mode code to access these devices.

- **Phase 5 - MMU**

  Implements the very beginnings of an MMU, including one of its most basic features: demand paging.

  Back in Phase 1, when a process was created or destroyed, or we switched from running one process to another, your functions would call MMU handler functions to "notify" them about the changed situation. Until now, those handlers were always NOPs. Now, you are going to fill them in.

  In addition, you will implement a handler for page faults - which are special interrupts that occur when a process attempts to access a page which is not (yet) mapped by the MMU.

2 Phase 1 Overview

In this Phase, you will implement one major data structure - the process table - along with any number of smaller data structures to help. Although I will require that your process table has a certain size - and that we use those slots in a certain way - I will otherwise give you complete freedom about how you implement your data.

You will be required to implement the following functions, which may be called from other kernel-mode code (never from user-mode code). See below in this spec for details on each required function.

- `phase1_init()`

  This will be called exactly once, when the simulation starts up. Initialize your data structures, including setting up the process table entry for the starting process, `init`. (We'll give details about what `init` does below).

  Although you will create a process table entry for `init`, you must not run it (yet).

- `startProcesses()`

  This function never returns.

  Called by the testcase, once all of the data for the various phases have been initialized. At the time when this function is called, no process is "running," and the only process that has been created is `init`.

  This function should call the dispatcher; since `init` is the only process, it obviously will run. `init` will create all of the other processes (see below).

- `fork1(...)`

  This function creates a new process, which is a child of the currently running process. The term "fork" is a misnomer here; this function doesn't actually duplicate the current process like the UNIX fork. Instead, it creates a brand-new process, which runs inside a fresh MMU address space, and calls a new function as its `main()`.

- `join(...)`

  Like the UNIX `wait()` syscall, this function blocks the current process until one of its children has terminated; it then delivers the "status" of the child (that is, the parameter that the child passed to `quit()`) back to the parent.

- `quit(...)`

  This function never returns.

  Like the UNIX `exit()` syscall, this function terminates the current process, with a certain "status" value. If the parent of this process is already waiting in a `join()`, then the parent will be awoken.

  While the current process will never again run any code, it will continue to take up space in the process table until its status has been delivered to its parent.

- `zap(...)`

  Like the UNIX `kill()` syscall, this function asks another process to terminate itself. It does **NOT** actually terminate the other process, however. Instead, it sets a flag which tells the process that it should `quit()` as soon as is practical.

  If the other process is blocked, that process will `quit()` at some point after it unblocks. Unlike the UNIX `kill` syscall, `zap` does **not** unblock a blocked target process.

  The zapping process will block until the target process terminates.

  Any number of processes can be waiting to "zap" the same target process; when the process finally calls `quit()`, all of these blocked processes will awake at the same time.

- `isZapped()`

  Asks if the current process has been zapped by another. Your code in Phase 1 will not force the process to call `quit()` - but we expect that well-behaved processes will call `quit()` as soon as they can, once they have been zapped.

- `getpid()`

  Returns the PID of the current executing process.

- `dumpProcesses()`

  Prints human-readable debug data about the process table.

- `blockMe(...)`

  Used by other Phases to ask that the current process be blocked for a while. The process will not run again until `unblockProc()` is called on it.

- `unblockProc(...)`

  Moves a process (not the current process) from the blocked state to unblocked; it will run again as soon as the dispatcher allows.

- `readCurStartTime()`

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

- `currentTime()`

  Reads the current wall-clock time from the `CLOCK` device.

- `readtime()`

  Returns the total amount of CPU time (in microseconds) used by the current process, since it was created.

- `timeSlice()`

  Checks to see if the currently executing process has exceeded its time slice; if so, it calls the dispatcher. Otherwise, it simply returns.

3 Special Process: `init`

Your Phase 1 code must implement a special kernel-mode process called `init`. `init` must always have PID 1, and it will never die. It must run at the second-lowest priority (6).

`init` has two purposes. First, it takes part in the Bootstrap process; it will call functions, in all other 4 Phases, to allow them to start any service processes that they require, and then will call `fork1()` twice, to create two basic processes: `sentinel`, and `testcase_main`.

After Bootstrap is complete, it enters a loop which calls `join()` over and over, simply to clean up the process table. So long as you are getting good return values from `join()`, keep calling it (ignoring entirely what processes are dying, or what their statuses are). However, if you ever receive -2 from `join()` (meaning that you have no children left), print out an error message and terminate the simulation.

No process will ever `zap()` `init`, and so it doesn't need to call `isZapped()`. It will never call `quit()`.

4 Special Process: `sentinel`

Your Phase 1 code must implement a special kernel-mode process called `sentinel`. It will never die, but it must run at the lowest priority (7).[1]

`sentinel` exists to detect deadlock; it only runs if every other process, in the entire system, is blocked. In Phase 1, there should never be a situation where all processes block (unless a testcase is **testing** the deadlock check). However, once we implement device drivers, there will be situations where all processes might block for a short while - when we have processes waiting to get responses from various devices.

Thus, `sentinel` must implement a loop like this:

```
while (1) {
    if (phase2_check_io() == 0)
        report deadlock and terminate simulation
    USLOSS_WaitInt();
}
```

The `USLOSS_WaitInt()` function call blocks the simulation until an interrupt arrives. Thus, `sentinel` (and all other processes will block, not consuming any CPU, until some sort of interrupt arrives.

5 Special Process: `testcase_main`

Your Phase 1 code must implement a special kernel-mode process called `testcase_main`. It runs at priority 3 (the middle "normal" priority).

This function simply calls one other function: `testcase_main()`.[2] This function will be provided by the **testcase** - not by your own code.

If the function `testcase_main()` ever returns, then your code must call `USLOSS_Halt()`, passing it the return value from the function. But if the return code is nonzero, you must **ALSO** print out an error message. That is,

[1] I recommended, above, that you use `fork1()` to create this process, but normally `fork1()` won't create a process with a priority lower than 5. You may either (a) put special code into `fork1()` to handle `sentinel`, or (b) create `sentinel` manually, in the process table, without `fork1()`. Your choice!

[2] Pay attention to the difference between the "process" and the "function." They have the same name, but they are not the same thing.

if `testcase_main()` returns nonzero, it is reporting "some error was detected by the testcase," while if it returns zero, that means that the testcase did not notice a problem.

6 Bootstrap

One of the harder problems in an OS is figuring out how to "bootstrap"[3] the kernel. Once it's running, it's possible to imagine how proceses can share the CPU, how the dispatcher works, etc. But how do we start things off at first? It might seem easy, but the details are complex: you have to initialize all of the data structures for all of the Phases; you have to start a certain number of "service processes," which help the OS work properly (and some of them might depend on other ones), and then you have to start up the testcase code itself. To make matters worse, our OS implementation is broken into Phases, and so we have to support partially-implemented OSes.

Overall, the bootstrap process for our project works like this:

- Initialize the data structures for all of the Phases, in order from 1 to 5. (The testcase will skip phases that don't exist yet.)

  To do this, the **testcase** will call:

  ```
  phase1_init()
  phase2_init()
  phase3_init()
  phase4_init()
  phase5_init()
  ```

  During this part of Bootstrap, no processes are running, and therefore calling `fork1()` is forbidden.

  All code running at this time is running in kernel mode.

- Once all of the phases have initialized their data, it's time to create our initial processes. The testcase calls `startProcesses()`, which is provided by the Phase 1 code. This function will create the `init` process, and then call the dispatcher, which will cause the `init` process to begin running. For a very short while, `init` is the **only** process in the entire system!

  `startProcesses()` will never return.

- Now, at last, we are running **inside** the context of a process. The `init` process will call into each of the other phases (in order), asking them to create processes if they wish. (Since we are now running in process context, `fork1()` can be called.)

  To do this, the **`init` process (provided by Phase 1)** will call:

[3] `https://en.wikipedia.org/wiki/Bootstrapping`

```
phase2_start_service_processes()
phase3_start_service_processes()
phase4_start_service_processes()
phase5_start_service_processes()
```

in that order.

NOTE:
Each testcase knows which Phase it is testing; it will provide dummy implmentations for these functions, for any Phase(s) that are not implemented yet. (That way, we don't have to change our Phase 1 code as we add new Phases.)

- Next, `init` creates two additional processes: `sentinel` and `testcase_main`. See above for details about what each process does.
- Finally, `init` enters its infinite loop; it will simply call `join()` over and over, cleaning up statuses from any of its children which eventually die. If **all** of its children die, it will report an error an halt the simulation; otherwise, it will run forever.

  As noted above, `init` will never die.

7 Interrupts and Concurrency

Beware concurrency issues!!! Any time that your code is running - even if it is running code that is definitely part of the kernel - it is possible that your function might be interrupted. In the kernel, this generally happens because of interrupts - an interrupt can fire at **any time,** while your code is running. This means that **any operation** - even things that seem simple - can be interrupted!

For example, a simple line like

```
x++;
```

might break down, inside your CPU, into 3 (or more!) assembly language statements:

```
LOAD       reg[foo], x
INCREMENT  reg[foo]
STORE      x, reg[foo]
```

What would happen if an interrupt occurred between the load and store operations - might code in the interrupt handler modify the variable `x` without you noticing???

For this reason, all functions you write in Phase 1 (except if one simply reads a simple state, such as `getpid()`, or anything that runs before `startProcesses()` is called) need to disable interrupts, and then restore them later. This will ensure that no other code can run; your program will have complete control over the CPU until you restore the interrupts.

See the USLOSS documentation to see how to read and edit the PSR to disable and restore interrupts.

NOTE:
While all of the functions you write need to run with interrupts disabled, if you write helper functions it is peremissible for them to simply **assume** that interrupts are disabled - and not disable them again - so long as their callers always disable interrupts before calling the helper.

7.1 Restore vs. Enable

Notice that I said that, at the end of a function, you must **restore** interrupts, not enable them! This is because it is quite possible (likely, even!) that the code which calls one of your functions may have **already** disabled interrupts. For instance, perhaps an interrupt handler is running, and is trying to send a message; this might require waking up a process, and waking up a process requires updating fields in the process table.

Therefore, when you disable interrupts at the top of your function, save the old PSR state, and make a point of **restoring the PSR** to exactly its previous value before you return.

8 Basic Process Rules

8.1 Parent and Child Processes

A process may create as many child processes as it likes, but of course each process will have exactly one parent. Unlike in UNIX, parent processes in our OS cannot end until all of their children have ended **and** the parent has collected all of their statuses (using `join()`).

Your process table must include fields which allow you to track the parent/child relationships between the currently running (or recently terminated) processes.

8.2 Process Table and PIDs

You must implement an array, known as the "process table," which stores information about all of the current and recently-run processes. You must define the struct which is used to store process information, and you can organize it however you like; however, your process table must be an array of these structures, with exactly `MAXPROC`[4] elements.

Each time that `fork1()` is called, you will allocate another slot in this table, and use it to store information about the process (or return an error to the user). This slot **must not** be freed until the parent process has collected the ending status for the process with `join()`; however, once this status has been

[4] `phase1.h`

collected, you must re-use the slot for other, additional processes that might be created.

Each process must be assigned a PID (process ID); this is a non-negative `int`. At all times, the PID, and the slot where the process is stored in the process table, **MUST** be linked as follows:

```
    int slot = pid % MAXPROC;
```

This is critical because other code, in other Phases, may have their own tables that parallel the process table, and they will use the calculation above to deteremine the slot they should use for each process's data.[5]

We do not define how you must assign PIDs, but simply assigning them sequentially is an easy strategy. However, because of the modulo rule above, be aware that sometimes you will need to skip over a PID in the sequence, since the process table entry where it normally would be assigned is still in use.[6]

9 Priorities and the Dispatcher

Each process is assigned a priority, which must be in the range 1-7 (inclusive). Lower numbers mean higher priority, and no process will run if a higher priority process is runnable. `init` must run at priority 6, and `sentinel` must run at priority 7; for all other processes, the range is limited to 1-5 (inclusive).

The dispatcher must treat priorities as absolute; that is, a process will **never** run if there is a runnable process with a more favored priority. For example, suppose that there are 3 processes with priority 4, which have been waiting for a very long time for the CPU; further suppose that there is a fourth process, with priority 2, which is also runnable. The priority 2 process will run **forever** - until it blocks or dies - entirely starving the priority 4 processes.

Within a given priority level, dispatcher must implement round-robin scheduling, with an 80 millisecond quantum for time-slicing. This means that any time that a process becomes runnable (when it is created, or when it moves from blocked to runnable), it is added to the end of a queue, and will wait its turn behind other processes already in the queue.

If a process runs for its full 80ms timeslice, it will be moved back to the rear of the queue, and another process will get a turn (assuming, of course, that there are any other runnable processes at the same priority).

If a process becomes runnable and it has a higher priority than the currently running process, the dispatcher will immediately switch to it.

[5] In a real OS, where the various subcomponents of the OS are more tightly linked than our Phases, the other subcomponents would have their own fields in the process table struct, and thus the layout of the process table might be less restrictive.

[6] For example, since `init` will never die and `init` has PID 1, this means that it will be **impossible** for any process to have PID `1+MAXPROC`.

9.1 Queues are Required

You **must** implement a separate run-queue for each priority. You **must not** do a global search of all processes in the process table. You **must not** have blocked processes in any run-queue[7] You **must not** mix processes with different priorities into a single run-queue.

9.2 Context Switching

An operating system kernel performs a "context switch" to change the CPU to make it ready to execute a different process than the one that is currently running. This generally involves loading the CPU up with new registers (including things like the Instruction Pointer a.k.a. Program Counter) as well as setting up basic machine state variables (like the PSR in USLOSS).

Usually (pay attention to the exceptions below!), a context switch also requires that we save the old state of the process. To implement this, the kernel must have some sort of data structure which keeps track of the state of a process while it is "switched out."

If implemented properly, a process can be switched out of the CPU, stored in memory for as long as we want, and then switched back into the CPU, and the code will never notice it (unless they are closely monitoring the clock). That is, a context switch must restore the process to **exactly** the same state as it had before it was switched out.

While conceptually simple, context switches are notoriously hard to actually implement. Therefore, you should use the `ContextSwitch` mechanism built in to USLOSS. You should store a `USLOSS_Context` variable in each process's slot in the process table. Call `USLOSS_ContextInit()` to initialize this when a new process is created, and then use `USLOSS_ContentSwitch()` to switch from one context to another.

9.2.1 When Is Saving Not Required?

As noted above, there are exceptional situations where you do not need to save the current state when performing a context switch. The first is when the testcase calls `startProcesses()` during the bootstrap process. As noted above, this call switches from no-processes-running to starting processes in the first place, and it **never returns.** Thus, you don't need to save anything.

The second (much more mundane) situation is `quit()`. When a process quits, it posts its status into the process table (and perhaps wakes up a few processes waiting for it), but then it simply ceases to run. In this case, there's no need to ask USLOSS to save the current state, as you will never return to it.

[7] Do you want the current running process in a run-queue, or will you remove it? Your choice!

10 Kernel Mode

The code in Phase 1 must only be called in kernel mode. Therefore, your code must **check**, at the top of each of the required functions, to confirm that the PSR shows that you are running in kernel mode.

If any one of the required functions is called from user mode, print an error message and call `USLOSS_Halt(1)`.

Of course, if you implement any helper functions in Phase 1 (I recommend it!), it's up to you whether or not to perform this cheeck; it is only required on the functions listed in this spec.

11 Contexts

Before we get to the formal spec for each function, you must understand the difference between **process context** and **interrupt context.**

Once `startProcess()` is called, the OS is **always** running one process or another; there is no way to be "not running a process." However, when an interrupt occurs, it often **completely unrelated** to the current process. For instance, we might be calculating the digits of PI in the CPU, and be interrupted by the disk, telling us that some I/O has completed.

Interrupt Context
Thus, as we handle most interrupts, it's important that we understand that the currently running process is probably not related to the interrupt - and as such, there is a very limited set of actions that we might perform. We call this "interrupt context," and it means that we can only do things that are (a) non-blocking, and (b) not specific to the currently running process. For instance, calling `getpid()` in interrupt context is meaningless; we shouldn't care which process is currently running. Likewise, calling `join()` or `blockMe()` is illegal in interrupt context, since both calls might block the currently-running process. On the other hand, calling `unblockProc()` is perfectly valid, since it is changing something about **another** process - not related to the one that is currently running.

Process Context
On the other hand, sometimes (even in an interrupt) we are specifically running in the context of a certain process. For instance, syscalls are implemented as a special type of interrupt; likewise, page faults (which you will deal with in Phase 5) are interrupts that are specific to the currently running process.

Following the Rules
Each function below is labelled as "Process Context Only" or "Interrupt Context OK." This imposes rules on both the caller, and the called function.

If the function is "Process Context Only," then this function **must not** be called in Interrupt Context. Some of these functions will block the current process.

If the function is “Interrupt Context OK,” then this function is safe to be called from Process Context or Interrupt Context. This puts severe limitations on the called function; while it may disable (and later restore) interrupts, it must **never** block, and it must not access any information that is specific to the currently-running process.

Of course, as mentioned above, it is permissible for an Interrupt Context function to read and write the process table, modifying some **other** function. But it must never do anything that is specifically about the **current** function.

12 MMU Interactions

We will not implement the MMU (Memory Management Unit, for Virtual Memory) until Phase 5. However, your code in Phase 1 must call the MMU, in three different situations, to let it know that it needs to do something. Until Phase 5, the testcases will provide dummy versions for these functions, that do nothing.

init
Almost at the end of `fork1()` - after you’ve fully initialized the process entry for the child - you must call `mmu_init_proc(pid)` to let it know that a new process exists. It will create a new page table for this process, and store it in the Phase 5 data structures.

For very basic Phase 1 processes - `init`, `sentinel`, etc. - it is permissible for you to call `mmu_init_proc()` for them, but it is not required. If you decide to call it, however, make sure that you do this sometime **after** `startProcesses()` is called - so that we know that the Phase 5 data structures have already been initialized.

quit
Any time that a process is terminated with `quit()`, you must call `mmu_quit(pid)` to inform the MMU that it can free the page table for that process.[8]

context switch
If the dispatcher performs a context switch, it must call `mmu_flush(void)` to notify the MMU that it needs to discard any page tables that it might have. Note that you must do this anytime that you call `USLOSS_ContextSwitch()` - even if you don’t have an old context (such as during `startProcesses()`).

[8] Worried that perhaps you didn’t call `mmu_init_proc()` to create the page table, and thus you might be destroying something that doesn’t exist? Don’t worry - those processes aren’t allowed to die, anyway!

13 Detailed Function Specifications

13.1 `void phase1_init(void)`

Context: n/a, called before `startProcesses()`. See Bootstrap above.

May Block: n/a

May Context Switch: n/a

Initializes the data structures for Phase 1. This function may, if you wish, populate the proc table with the information for `init`, but it **must not** perform any context switch, so `init` will not run (yet).

13.2 `void startprocesses(void)`

Context: n/a

May Block: **This function never returns**

May Context Switch: **This function never returns**

Called during bootstrap, after all of the Phases have initialized their data structures. Either this function, or `phase1_init()`, must create the process table entry for `init`, and this function must call your dispatcher. Since there is no current process, the dispatcher will context switch to `init`; see above to see what `init` does when it runs.

13.3 `int fork1(char *name, int (*startFunc)(char*), char *arg, int stackSize, int priority)`

Context: Process Context ONLY

May Block: No

May Context Switch: Yes

Args:

- `name` - Stored in process table, useful for debug. Must be no longer than `MAXNAME`[9] characters.
- `startFunc` - The `main()` function for the child process.
- `arg` - The argument to pass to `startFunc()`. May be `NULL`.

[9] phase1.h

- stackSize - The size of the stack, in bytes. Must be no less than USLOSS_MIN_STACK[10].
- priority - The priority of this process. Priorities 6,7 are reserved for init,sentinel, so the only valid values for this call are 1-5 (inclusive).[11]

Return Value:

- -2 : stackSize is less than USLOSS_MIN_STACK
- -1 : no empty slots in the process table, priority out of range, startFunc or name are NULL, name too long
- > 0 : PID of child process

Creates a child process of the current process. This function creates the entry in the process table and fills it in, and then calls the dispatcher; if the child is higher priority than the parent, then the child will run before fork1() returns.

When the child process runs, startFunc() will be called, and passed the parameter specified as arg. If startFunc() ever returns, this terminates the process; it has the same effect as if the process had called quit() (with the return value being the process status).

Since it is possible for startFunc() to return, Phase 1 must provide a wrapper function, which is the actual main() (from the perspective of USLOSS). This function must look up the process in the process table, find the startFunc(), call it, and call quit() if startFunc() returns.

The child process starts in kernel mode; it may switch itself into user mode if desired, but this must be done by the startFunc(); fork1() is not responsible for doing it.

fork1() will allocate a stack, of stackSize bytes, for the process. It will do so using malloc(), which is not true to how a real OS works - but it's a simplification we will use for this project.

fork1() must call the function mmu_init_proc(pid) for the newly created process. Until Phase 5, this will be a dummy function that accomplishes nothing - but our Phase 1 code must call it, so that when Phase 5 is implemented, the MMU can take this chance to create a new page table.

fork1() must call USLOSS to create a new context, using USLOSS_ContextInit(). Each process in the process table should have its own USLOSS_Context variable, to store the state of this process whenever it is blocked.

Comments:

- Should fork1() put the new child at the head or the tail of the list of children? **Either is permissible,** but you will probably find it easier to match my output (and thus not require as many README entries) if you add the child at the head.

[10] usloss.h

[11] If you want to create sentinel with fork1(), you are permitted to make an exception for that one process.

And after all, adding a child at the head is $O(1)$. Why would you do anything else, if you were given the choice???

- In UNIX, the `fork()` syscall creates a **duplicate** of the current process, not a new process with its own `main()`. But to implement this requires some Virtual Memory trickery which is too complex for Phase 1, so we're doing this (hackish) solution.
- Because creating a new process requires allocating memory for it, `fork()` in the Real World may block, while new memory pages are allocated; other, older pages might have to be swapped out to disk, discarded from the disk cache, etc.

  To simplify Phase 1, we are ignoring all that; your code will simply call `malloc()`. This is entirely unrealistic in the Real World, but it's good enough for our project.

13.4 `int join(int *status)`

Context: Process Context ONLY

May Block: Yes

May Context Switch: Yes

Args:

- `status` - **Out pointer.** Must point to an `int`; `join()` will fill it with the status of the process joined-to.

Return Value:

- -2 : the process does not have any children (or, all children have already been joined)
- > 0 : PID of the child joined-to

Blocks the current process, until one of its children terminates. If a child has already died, then this will not block, and will return immediately; otherwise, it blocks until one of its children dies. The PID of the dead child is returned, and the status is stored through the `status` out-pointer.

If multiple children have died before this call, or if multiple children die in rapid succession before this process can awake, then only one child status will be returned per call to `join()`. The status of other child(ren) will be saved, and will be available for later calls to `join()`.

The status returned, in the out-pointer, is always the same status as passed to `quit()`.

13.5 `void quit(int status)`

Context: Process Content ONLY

May Block: **This function never returns**

May Context Switch: **Always context switches, since the current process terminates.**

Args:

- `status` - The exit status of this process. It will be returned to the parent (eventually) through `join()`.

Return Value: **This function never returns**

Terminates the current process; it will never run again. The status for this process will be stored in the process table entry, for later collection by the parent process.

If the parent's process is already waiting in a `join()`, then this call will wake it up; the `join()` function will promptly return, delivering to the caller information about this newly-dead process, or perhaps a newly-dead sibling.

If the parent's process is not yet waiting in a `join()`, then this process table entry must continue to be in use until the parent collects the status from the process table entry.

If, when this process dies, one or more processes are blocked trying to `zap()` this process, all of them will be awoken immediately.

13.5.1 Freeing the Stack

You **must not** free the stack memory in `quit()`, since you are still using it! Instead, you must free the stack memory in `join()`; when the parent process collects the status of the dead process, you should clean up the stack at that point.

13.6 `void zap(int pid)`

Context: Process Context ONLY

May Block: Yes

May Context Switch: Yes

Args:

- `pid` - The PID of the process to zap.

Return Value: None

Requests that another process terminate. However, the process is not automatically destroyed; it must call `quit()` on its own. (This is important so that processes inside kernel code do not accidentally get destroyed while in the middle of a critical section.)

Unlike `join()`, `zap()` can target any process (not just children). However, if the caller attempts to zap itself, or to zap a non-existent process (including a process that is still in the process table but has terminated), `zap()` will print an error message and call `USLOSS_Halt(1)`. Likewise, if `zap()` targets PID 1 (`init`), `zap()` must print an error message and halt.

Like `join()`, `zap()` will block until the target process dies; however, unlike `join()`, `zap()` does not have the ability to inspect the status of the target process.

While a single process cannot `zap()` multiple processes at the same time, it may `zap()` several processes in sequence, one after another. On the other hand, it is permissible for **any** number of processes to all `zap()` the same target; they will all get woken up at the same time.

WARNING:
Unlike the `kill()` syscall in UNIX, `zap()` never unblocks any blocked process. If the target process is blocked in `zap(), join(),` or `blockMe()`, they will continue to be blocked, even though a process is attempting to `zap()` them. At some point in the future, when they wake up (for some unrelated reason), the process may, if they wish, call `isZapped()`, and notice that a `zap()` is pending.

13.7 `int isZapped(void)`

Context: Process Context ONLY

May Block: No

May Context Switch: No

Args: None

Return Value:

- 0 : the calling process has not been zapped (yet)
- 1 : the calling process has been zapped

Checks to see if the current process has been zapped by another.

NOTE:
This only asks the question whether or not a `zap()` is being requested; this

function **will not** automatically kill the current process. Instead, the current process, if zapped, will quit at a time of its own choosing.

13.8 `int getpid(void)`

Context: Process Context ONLY

May Block: No

May Context Switch: No

Args: None

Return Value: PID of the current process

Returns the PID of the current process.

13.9 `void dumpProcesses(void)`

Context: Interrupt Context OK

May Block: No

May Context Switch: No

Args: None

Return Value: None

Prints out process infromation from the process table, in a human-readable format. (Make it look nice, you're going to be using it for debug.) The exact format and content are up to the student, however, it must include, for each process still in the process table (that is, processes that are alive, and those that have ended but their status has not been collected yet), the following fields:

- Name
- PID
- Parent PID
- Priority
- Runnable status - 0=runnable, > 0 =blocked. See `blockMe()` below.

Remember: This is the **minimum!** I urge you to add more fields - as many as help you debug. And you may also find it useful to look at old (freed) process table entries, in addition to the live ones.

13.10 `void blockMe(int newStatus)`

Context: Process Context ONLY

May Block: **Must block**

May Context Switch: **Must block**

Args:

- `newStatus` - The reason for blocking the process

Return Value: None

Used heavily by the other Phases to block processes for a wide variety of reasons. The `newStatus` describes why the process is blocked; in this call, it must be greater than 10. (Print an error message, and halt USLOSS if it is not.)

Record the status in the process table entry for this process; once this is nonzero, the dispatcher should never allow the process to run (until the process is unblocked, of course!).

Call the dispatcher, to cause the OS to switch to some other process.

`blockMe()` never returns until some other process has called `unblockProc()` for the process.

13.11 `int unblockProc(int pid)`

Context: Interrupt Context OK

May Block: No

May Context Switch: Yes

Args:

- `pid` - The process to unblock

Return Value:

- -2 : the indicated process was not blocked, does not exist, or is blocked on a status ≤ 10.
- 0 : otherwise

Unblocks a process that was previously blocked with `blockMe()`. Note that, while `blockMe()` always blocks the current process, `unblockProc()` must of course be called by a **different** process.

The new process is placed on the appropriate run queue (according to its priority). It is always placed at the **end** of the queue, after any already-queued processes.

`unblockProc()` must call the dispatcher just before it returns, in case the newly-awoken process is higher priority than the currently running process. Thus, like `fork1()`, this function will never block, but it might **context switch** to another process temporarily.

13.12 `int readCurStartTime(void) / int currentTime(void) / int readtime(void)`

Context: Process Context ONLY

May Block: No

May Context Switch: No

Args: None

Return Value: See below

These three functions deal with timing processes as they consume the CPU. `readCurStartTime()` returns the wall-clock time (in microseconds) when the current process started its current timeslice. This value should be stored when the dispatcher switches to a process; it is reset any time that a context switch occurs. It is also reset if the dispatcher notices that the time slice has expired, it makes a new scheduling decision, but decides to keep running the same process - that is, a new time slice begins.

When the variable for `readCurStartTime()` is set, it must be set equal to `currentTime()`, so that it is a high-accuracy clock.

`currentTime()` simply reads the current wall-clock time (in microseconds) and returns it. This must use `USLOSS_DeviceInput(USLOSS_CLOCK_DEV, ...)` to read the time; simply counting how many interrupts have occurred is not accurate enough.

`readtime()` returns the **total** time (in microseconds) consumed by a process, across all of the times it has been dispatched, since it was created. It must include information for the current time slice, but it **must not** update the variable used for `readCurStartTime()`; that variable is only reset when a new timeslice begins.

When a process is swapped out for **any** reason, the dispatcher must calculate how long the process ran in the current time slice, and update the total in the process's entry in the process table.

13.13 `void timeSlice(void)`

Context: Process Context ONLY

May Block: No

May Context Switch: Yes

Args: None

Return Value: None

This function compares `readCurStartTime()` and `currentTime()`; it calls your dispatcher if the current timeslice has run for 80 milliseconds or more.

It would be possible, of course, to simply call the dispatcher directly, and let it make decisions about time slicing (it must also be aware of the 80 millisecond limit). However, the dispatcher is likely to be moderately expensive to run; this is a fairly cheap check to make. Moreover, this may be called by other Phases (or by a testcase), and those are not supposed to have access to call your dispatcher directly.

Note that, even if a process' timeslice has expired, it sometimes will not get switched out. This happens when there are no other runnable processes with the same priority - and thus the process continues to run. However, its variable for `readCurStartTime()` will be updated, so that the dispatcher won't be called again soon - not until another 80 milliseconds have elapsed.

(spec continues on the next page)

14 Tip: Linked Lists and Trees Without `malloc()`

The spec says that you **must not** allocate memory with `malloc()` in this simulation, except for the stack allocation in `fork1()`. And yet, you are required to build various process queues (run-queues, zapper queues) as well as trees (where a parent can have **many** children). How do we make this work?

The first trick is to **include the `next` fields in your process structures**:

```
struct Example {
    struct Example *run_queue_next;
    struct Example *first_child;
    struct Example *next_sibling;
};

void some_func(...)
{
    struct Example *process_before = ...
    struct Example *process_after  = ...
    process_before->run_queue_next = process_after;
}
```

To build a tree, where each parent can have an arbitrary number of children, the trick is to have only a pointer to the **first child** in the parent; then we will build a linked list of the siblings from there:

```
void build_parent_child_tree(...)
{
    struct Example *parent       = ...
    struct Example * first_child = ...
    struct Example *second_child = ...
    struct Example * third_child = ...
    parent      ->first_child  =  first_child;
    first_child ->next_sibling = second_child;
    second_child->next_sibling =  third_child;
    third_child ->next_sibling = NULL;
}
```

(spec continues on the next page)

15 Useful Code

You are permitted to use the following code in your solution, if you wish. (Give attribution in a comment.)

```
/* this is the interrupt handler for the CLOCK device */
static void clockHandler(int dev,void *arg)
{
    if (debug)
    {
        USLOSS_Console("clockHandler(): PSR = %d\n", USLOSS_PsrGet());
        USLOSS_Console("clockHandler(): currentTime = %d\n", currentTime());
    }

    /* make sure to call this first, before timeSlice(), since we want to do
     * the Phase 2 related work even if process(es) are chewing up lots of
     * CPU.
     */
    phase2_clockHandler();

    // call the dispatcher if the time slice has expired
    timeSlice();

    /* when we return from the handler, USLOSS automatically re-enables
     * interrupts and disables kernel mode (unless we were previously in
     * kernel code).  Or I think so.  I haven't double-checked yet.  TODO
     */
}


int currentTime()
{
    /* I don't care about interrupts for this process.  If it gets interrupted,
     * then it will either give the correct value for *before* the interrupt,
     * or after.  But if the caller hasn't disabled interrupts, then either one
     * is valid
     */

    int retval;

    int usloss_rc = USLOSS_DeviceInput(USLOSS_CLOCK_DEV, 0, &retval);
    assert(usloss_rc == USLOSS_DEV_OK);

    return retval;
}
```

16 Turning in Your Solution

You must turn in your code using GradeScope. While the autograder cannot actually assign you a grade (because we don't expect you to match the "correct" output with perfect accuracy), it will allow you to compare your output, to the expected output. Use this report to see if you are missing features, crashing when you should be running, etc.

16.1 README File

Since OSes are complex systems, it is sometimes impractical to exactly match the exact output from a testcase. This is especially true because I gave you a moderate amount of freedom about how to implement your data structures and algorithms.

While it is critical that you **implement the requirements** that I have given you, you have freedom about details. If your code matches the requirements but has different output in some minor detail, you may submit a README file, which explains to your TA why you think it still follows the spec. If your TA agrees, they will give you back the points that you lost from the testcase.

For each testcase that you would like points back for, your README file must:

- Give the testcase number
- Explain, **as precisely as you can,** why the difference doesn't matter. Do not simply say, "My code works correctly" **- give an argument** which explains why.