Assignment 2: A Preemptive, User-Level Thread Package

In the previous assignment, you implemented a cooperative, user-level thread package in which explicit calls to yield cause control to be passed from one thread to another. This assignment has three goals:

1. Implement preemptive threading, in which simulated "timer interrupts" cause the system to switch from one thread to another.

2. Implement sleep and wakeup scheduling functions. These functions will enable you to implement blocking mutual exclusion and synchronization.

3. Use the sleep and wakeup functions to implement join, which will block a thread until the target thread has finished executing.

Contents

1 Logistics

Before starting to program, we highly recommend reading the entirety of this handout. This section describes how to acquire the starter files, compile code, and submit the assignment.

1.1 Setting up your repository

Your assignment is downloaded from, and submitted to, MarkUs using `git`. Login to MarkUs and navigate to CSC369. From there, navigate to 'a2', where you can see (among other things) the URL of the repository. Before cloning the repository, click 'Add Starter Files to Repository'. Now you can clone the repository.

1.2 Working in your repository

Once you have cloned your repository you should find the starter files in the 'a2' subdirectory. Start by reading the `README.md` in that subdirectory, which includes instructions on how to compile everything. The majority of your work should be done in `a2/src/csc369_thread.c`. In addition, you can write unit tests much like in Assignment 1. Finally, you can also write your own example applications. Some examples, referred to in this handout, are already given to you.

You should, under no circumstances, change other files in the `src` directory, such as `a2/src/csc369_thread.h` or `a2/src/CMakeLists.txt`. It is also unlikely that you will need to change other `CMakeLists.txt`, except perhaps to add more examples or comment out an `add_subdirectory`. An understanding of CMake is not required for this assignment, but you do need to know how to use the command line to build your project. Again, please refer to `a2/README.md` for more information.

1.3 Submitting changes to your repository

You can find the files you have modified by running the `git status` command. You can commit modified files (e.g., `csc369_thread.c`) to your local repository as follows:

```
git add csc369_thread.c
git commit -m "My useful commit message"
```

We suggest committing your changes frequently by rerunning the commands above (with different meaningful messages to the commit command), so that you can go back to see the changes you have made over time. Once you have tested your code, and committed it locally (check that by running git status), you can `git push` it back to MarkUs. You should also do this frequently. We collect and grade the last version pushed to MarkUs before the due date, up to the maximum late penalty. Please see the Syllabus and MarkUs for more details on the late penalty.

2 Background: Using signals as interrupts

User-level code cannot use hardware timer interrupts directly. Instead, POSIX operating systems provide a software mechanism called signals that can be used to simulate "interrupts" at the user level. These signals interrupt your program and give you a chance to handle that interrupt.

For example, when you type `Ctrl-C` to kill a program, that causes the OS to send the SIGINT signal to your program. Most programs don't handle this signal, so by default, the OS kills the process. However, if you wanted to save the state of your program before it was killed (e.g., a text editor could save any unsaved files), you can register a handler with the OS for SIGINT. Then, when the user types `Ctrl-C`, the OS calls your handler; in your handler you could write out the state of your program, and then exit.

More generally, signals are a form of asynchronous, inter-process communication mechanism. A signal can be sent from one process to another, or from a process to itself. We will use the latter method to have the process that invokes your user-level scheduler, i.e., your thread library functions, deliver timer signals to itself.

The operating system delivers a signal to a target (recipient) process by interrupting the normal flow of execution of the process. Execution can be interrupted after any instruction. If a process has registered a signal handler, that handler function is invoked when the signal is delivered. After the signal handler finishes executing, the normal flow of execution of the process is resumed. Notice the similarities between signals and hardware interrupts.

Because signals can be delivered after any instruction, signal handling is prone to race conditions. For example, if you increment a counter (e.g., `counter = counter + 1`), either during normal execution or in your signal handler code, the increment operation may not work correctly because it is not atomic. The signal may be delivered in between the instructions implementing the increment operation. To avoid this problem, you should disable signal delivery while the counter is being updated. Note that this is essentially the old OS strategy of disabling interrupts to protect critical sections on a uniprocessor.

Please read a short introduction to signals (`http://en.wikipedia.org/wiki/Unix_signal`) to understand how they work in more detail. Make sure to read the "Risks" section or else you may not able to answer some questions below.

Now go over the code in the files `csc369_interrupts.c` and `csc369_interrupts.c`. You do not need to understand all the code, but it will be helpful to know how these functions should be used. Below, we use the terms "interrupts" and "signals" interchangeably.

- `void CSC369_InterruptsInit();`

  Initializing the interrupts library installs a timer handler in the program using the sigaction system call. When a timer signal fires (see `CSC369_INTERRUPTS_SIGNAL_INTERVAL`), then `HandleSignal` is called. The function `HandleSignal`, in turn, schedules another timer to go off and then calls `CSC369_ThreadYield`.

  Assume that there are multiple ready threads. Whichever thread is running will, at some point,

be interrupted by a timer signal. The function `HandleSignal` calls `CSC369_ThreadYield`, and so the current executing thread has been "preempted" for the next thread in the ready queue.[1] Another thread will run until the next timer signal fires and it, too, is preempted.

- `CSC369_InterruptsState CSC369_InterruptsSet(CSC369_InterruptsState state);`

  This function allows the caller to enable or disable interrupts. We call the current enabled or disabled state of the signal the *signal state*. This function also returns whether the signals were previously enabled or not (i.e., the previous signal state). Notice that the operating system ensures that these two operations (reading previous state, and updating it) are performed atomically when the sigprocmask system call is issued. Your code should use this function to disable signals when running any code that is a critical section.

  The convenience functions, `CSC369_InterruptsDisable` and `CSC369_InterruptsEnable`, simply use `CSC369_InterruptsSet`.

- `int CSC369_InterruptsAreEnabled();`

  This function returns whether signals are enabled or disabled currently. You can use this function to check (i.e., assert) whether your assumptions about the signal state are correct.

- `void CSC369_InterruptsSetLogLevel(CSC369_InterruptsOutput level);`

  This function lets you control whether the signal handler prints to stdout.

Why does `CSC369_InterruptsSet` (and its convenience functions) return the previous signal state? The reason is that it allows "nesting" calls to this function. The typical usage of these functions is shown in Listing 1.

```
1  fn() {
2    int const prev_state = CSC369_InterruptsDisable();
3    // ... critical section ...
4    CSC369_InterruptsSet(prev_state);
5  }
6
7  fn_caller() {
8    int const prev_state = CSC369_InterruptsDisable();
9    // begin critical section
10   fn();
11   // fn_caller expects that signals are STILL disabled
12   // end critical section
13   CSC369_InterruptsSet(prev_state);
14 }
```

Listing 1: Correctly restoring interrupt state for nested interrupts.

The call in `fn` to `CSC369_InterruptsDisable` disables signals. The call in `fn` to `CSC369_InterruptsSet` restores the signal state to its previous state (i.e., the signal state before the call to `CSC369_InterruptsDisable`), rather than unconditionally enabling signals. This is useful because the caller of the function `fn` may be expecting signals to remain disabled after the `fn` finishes executing.

[1] That is, the current executing thread did not *voluntarily* call yield.

Before moving on, make sure you can answer the following questions:

1. What is the name of the signal that is used to implement preemptive threading?

2. Which system call is used by the process to deliver signals to itself?

3. How often is this signal delivered?

4. When this signal is delivered, which function in `csc369_thread.c` is called?

5. Is the signal state enabled or disabled when the function in `csc369_thread.c` above is invoked? If the signal is enabled, could it cause problems? If the signal is disabled, what code will enable them? Hint: look for `sa_mask` in `CSC369_interrupts.c`

6. What does `CSC369_InterruptsPrintf` do? Why is it needed? Will you need other similar functions?

3 Implementing the library

3.1 Task 1: Preemptive Threading

Signals can be sent to the process at any time, even when a thread is in the middle of a yield, create, or exit call. It is a very bad idea to allow multiple threads to access shared variables (such as your ready queue) at the same time. You should therefore ensure mutual exclusion, i.e., only one thread can be in a critical section (accessing the shared variables) in your thread library at a time.

A simple way to ensure mutual exclusion is to disable signals when you enter procedures of the thread library and restore the signal state when you leave. Think carefully about the invariants you want to maintain in your thread functions about when signals are enabled and when they are disabled. Make sure to use the `CSC369InterruptsAreEnabled—` function to check your assumptions. An "Assignment 2 Mutual Exclusion Pattern" is shared in Listing 2 and uses the concepts from Listing 1 to avoid issues.

```
1  CriticalFunctionHelper() {
2    assert(!CSC369_InterruptsAreEnabled());
3
4    // ...
5  }
6
7  SomeA1Function() {
8    int const prev_state = CSC369_InterruptsDisable();
9    CriticalFunctionHelper();
10   CSC369_InterruptsSet(prev_state);
11 }
```

Listing 2: Mutual Exclusion Pattern

The pattern in Listing 2 may not always apply. Because of thread context switches, the thread that disables signals may not be the one enables them. In particular, recall that `setcontext` restores the register state saved by `getcontext`. The signal state is also saved when `getcontext` is called, and restored by `setcontext`. As a result, if you would like your code to be running with a specific signal state (i.e., disabled or enabled) when `setcontext` is called, make sure that `getcontext` is called with the same signal state. Maintain the right invariants, and you will have no trouble dealing with context switches.

To complete Task 1, you should:

1. Copy over your code from Assignment 1. Note the changes in Assignment 2 to `CSC369_ThreadExit` and `CSC369_ThreadKill`. You must now consider "exit codes"; see the enum `CSC369_ExitCode` near the top of `csc369_thread.h`.

2. Identify the critical sections in your code and ensure mutual exclusion to avoid data races with preemptive threading.

3. Test whether the program worked correctly.

3.2 Task 2: Sleep and Wakeup

Now that you have implemented preemptive threading, you will extend your threading library to implement the sleep (`CSC369_ThreadSleep`) and wake (`CSC369_ThreadWakeNext`, `CSC369_ThreadWakeAll`) functions. These functions will allow you to implement mutual exclusion and synchronization primitives. In real operating systems, these functions would also be used to suspend and wake up a thread that performs IO with slow devices, such as disks and networks. The sleep primitive blocks or suspends a thread when it is waiting on an event, such as a mutex lock becoming available or the arrival of a network packet. The wake primitive awakens one or more threads that are waiting for the corresponding event.

The sleep and wake functions that you will be implementing for this assignment are summarized here:

- `int CSC369_ThreadSleep(CSC369_WaitQueue* queue);`

  This function suspends the caller and then runs some other thread. The calling thread is put in a wait queue passed as a parameter to the function. The `CSC369_WaitQueue` data structure is similar to the ready queue; there can be many queues in the system, one per type of event or condition. Upon success, this function returns the identifier of the thread that took control as a result of the function call. The calling thread does not see this result until it runs later. Upon failure, the calling thread continues running, and returns an error code (see header).

- `int CSC369_ThreadWakeNext(CSC369_WaitQueue* queue);`

  This function wakes up one thread from the (FIFO) queue. The awoken thread is put in the ready queue. The calling thread continues to execute and receives the result of the call, which is the number of threads woken up. It should return zero if there were no suspended threads in the wait queue.

- `int CSC369_ThreadWakeAll(CSC369_WaitQueue* queue);`

  Nearly identical to `CSC369_ThreadWakeNext`, except all threads in queue are moved to the ready queue (again, in FIFO order; i.e., first thread to sleep is first thread woken up).

You will need to implement a `CSC369_WaitQueue` data structure before implementing the functions above. The `csc369_thread.h` file provides the interface for this data structure. Note that **each thread can be in only one queue at a time**. In addition, all the thought that you put into ensuring that thread preemption works correctly previously will apply to these functions as well. In particular, these functions access shared data structures (which ones?), so be sure to enforce mutual exclusion.

When implementing sleep, it will help to think about the similarities and differences between this function versus yield and exit. Make sure that sleep suspends (blocks) the current thread rather than spinning (running) in a tight loop (we have given you a function for spinning, see: `CSC369_ThreadSpin`). This would defeat the purpose of invoking sleep because the thread would still be using the CPU.

3.3 Task 3: Waiting for threads to exit

Now that you have implemented the sleep and wakeup functions for suspending and waking up threads, you can use them to implement blocking synchronization primitives in your threads library. You should start by implementing the join function, which will block or suspend a thread until a target thread terminates (or exits). Once the target thread exits, the thread that invokes join should continue operation. As an example, this synchronization mechanism can be used to ensure that a program (using a master thread) exits only after all its worker threads have completed their operations.

The join function is summarized below:

- `int CSC369_ThreadJoin(Tid tid, int* exit_code);`

  This function suspends the calling thread until the thread whose identifier is tid terminates. A thread terminates when it invokes `CSC369_ThreadExit`. Upon success, this function returns the identifier of the thread that exited. The exit status of the thread that exited (i.e., the value it passed to `CSC369_ThreadExit`) will be copied into the location pointed to by `exit_code`. Upon failure, the calling thread continues running, and returns an error code (see header).

Implementation: You need to associate a `CSC369_WaitQueue` with each thread (e.g., in the thread control block). When a thread invokes `CSC369_ThreadJoin`, it should sleep on the wait queue of the target thread. When the target thread invokes `CSC369_ThreadExit`, it should wake up the threads that are waiting on it.

Before you begin this task, there are a few technicalities with this function that you should consider:

- You can assume that if a thread exits (e.g., via `CSC369_ThreadExit`), then subsequent join will fail unless the exiting thread has not been fully cleaned up yet.

- When a thread is killed, remember to set its exit code to `CSC369_EXIT_CODE_KILL`. This way, the join function will be able to see if the thread was killed.

- One issue with implementing join is that a deadlock may occur. For example, if Thread A waits on Thread B, and then Thread B waits on Thread A, then both threads will deadlock. We do not expect you to handle this condition for the assignment.

4 Compiling and testing

The starter files include a `README.md`, which contains instructions on how to compile everything. **Make sure you follow those instructions.** Do not attempt to compile files directly using `gcc`.

You can compile your program into two "types" of builds. A debug build is useful for debugging because it includes debug symbols (used by `gdb`) and does not attempt to optimize your code. A release build is what is typically "shipped" to users of your library. The compiled library in this build is optimized for, e.g., performance. As a result, you may encounter errors in a release build that you would not encounter in a debug build.

The rest of this chapter discusses how to test different aspects of your library. It assumes that you have compiled your program in a debug build, but the same instructions apply for a release build. Simply replace `cmake-build-debug` with `cmake-build-release` below.

4.1 Testing earlier functionality

Because the majority of Task 1 is ensuring mutual exclusion in critical sections, the tests from Assignment 1 (mostly) still apply.[1] These tests should be without interrupts enabled, so no preemption would occur. The tests serve as a useful sanity check to ensure that any code you added for mutual exclusion in Task 1 does not break something.

Assuming you are able to successfully compile your code, then you can run these tests as below:

Example output from A1 tests running against A2

```
wolf$ ./cmake-build-debug/tests/check_a2_thread_a1
Running suite(s): Student Test Suite
100%: Checks: 16, Failures: 0, Errors: 0
my-home/a2/tests/check_a2_thread_a1.c:118:P:Errors Test Case:test_error_0_yieldto_invalid:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:124:P:Errors Test Case:test_error_0_kill_self:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:130:P:Errors Test Case:test_error_0_kill_negative_tid:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:136:P:Errors Test Case:test_error_0_kill_uncreated_tid:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:151:P:Errors Test Case:test_error_create_more_than_max:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:160:P:One Thread Case:test_main_thread_has_id_0:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:166:P:One Thread Case:test_main_thread_yield_itself:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:172:P:One Thread Case:test_main_thread_yieldto_itself:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:190:P:Two Threads Case:test_create_with_explicit_exit:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:219:P:Two Threads Case:test_create_with_recursion:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:97:P:Two Threads Case:test_0_with_explicit_exit:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:245:P:Max Threads Case:test_yield_and_recreate_all:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:286:P:Max Threads Case:test_yield_and_kill_all:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:303:P:Memory Case:test_dynamically_allocates_stack:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:333:P:Memory Case:test_stacks_sufficiently_apart:0: Passed
my-home/a2/tests/check_a2_thread_a1.c:347:P:Memory Case:test_fp_alignment:0: Passed
```

4.2 Testing preemption

You can try the Hot Potato example, where a "hot potato" is passed around from one thread to another. In this scenario, the thread functions never explicitly yield to one another and instead rely completely

[1] There have been changes in Assignment 2 to the APIs of kill and exit. As a result, one test was removed and other tests had calls to exit changed to accommodate the new exit code argument.

on being preempted. Assuming you are able to successfully compile your code, then you can run the example as below:

Example output from hot potato

```
wolf$ ./cmake-build-debug/examples/hot_potato
Starting hot potato.
 0.000101: hot potato passed from 0 to 1.
 0.000317: hot potato passed from 1 to 2.
 0.000529: hot potato passed from 2 to 3.
 0.000746: hot potato passed from 3 to 4.
 0.000969: hot potato passed from 4 to 5.
...
 9.989883: hot potato passed from 60 to 61.
 9.990097: hot potato passed from 61 to 62.
 9.990319: hot potato passed from 62 to 63.
 9.990539: hot potato passed from 63 to 0.
 9.990974: hot potato passed from 0 to 1.
 9.991201: hot potato passed from 1 to 2.
 9.991423: hot potato passed from 2 to 3.
 9.991641: hot potato passed from 3 to 4.
 9.991858: hot potato passed from 4 to 5.
 9.992072: hot potato passed from 5 to 6.
 9.992302: hot potato passed from 6 to 7.
 9.992511: hot potato passed from 7 to 8.
 9.992753: hot potato passed from 8 to 9.
 9.992971: hot potato passed from 9 to 10.
 9.993214: hot potato passed from 10 to 11.
 9.993432: hot potato passed from 11 to 12.
Killing all created threads.
Hot potato is done.
```

The first number in each line is the number of seconds that have elapsed so far.[2] After approximately 10 seconds, the example will end. That is, after roughly 10 seconds, the main thread kills the hot potato threads.

4.3 Testing join

You can try the Spin and Join example, where mosts thread join the "previous" thread. All threads "spin" (i.e., use up the CPU) for some random amount of time. The majority of threads (i.e., with the exception of one) never explicitly yield to one another and instead rely completely on being preempted. One created thread (i.e., not the main thread) waits until all threads are asleep and then exits, triggering the domino. Assuming you are able to successfully compile your code, then you can run the example as below:

Example output from spin and join

```
wolf$ ./cmake-build-debug/examples/spin_and_join
TID(2) waited for TID(1), which exited with 32
TID(3) waited for TID(2), which exited with 33
TID(4) waited for TID(3), which exited with 34
TID(5) waited for TID(4), which exited with 35
TID(6) waited for TID(5), which exited with 36
TID(7) waited for TID(6), which exited with 37
TID(8) waited for TID(7), which exited with 38
TID(9) waited for TID(8), which exited with 39
TID(10) waited for TID(9), which exited with 40
TID(11) waited for TID(10), which exited with 41
TID(12) waited for TID(11), which exited with 42
TID(13) waited for TID(12), which exited with 43
TID(14) waited for TID(13), which exited with 44
TID(15) waited for TID(14), which exited with 45
TID(16) waited for TID(15), which exited with 46
TID(17) waited for TID(16), which exited with 47
TID(18) waited for TID(17), which exited with 48
TID(19) waited for TID(18), which exited with 49
TID(20) waited for TID(19), which exited with 50
```

[2] Notice how each thread passes the hot potato to their "neighbour". You can change this by referring to the "TODO" in `hot_potato.c` so that the hot potato is passed to a random thread instead.

```
TID(21) waited for TID(20), which exited with 51
TID(22) waited for TID(21), which exited with 52
TID(23) waited for TID(22), which exited with 53
TID(24) waited for TID(23), which exited with 54
TID(25) waited for TID(24), which exited with 55
TID(26) waited for TID(25), which exited with 56
TID(27) waited for TID(26), which exited with 57
TID(28) waited for TID(27), which exited with 58
TID(29) waited for TID(28), which exited with 59
TID(30) waited for TID(29), which exited with 60
TID(31) waited for TID(30), which exited with 61
TID(32) waited for TID(31), which exited with 62
TID(0) waited for TID(32), which exited with 63
```

4.4 Testing with valgrind

It may be useful to use Valgrind to test for memory leaks and/or other errors in memory. When running Valgrind, it would be helpful to include flags like `--leak-check=full` and `--show-leak-kinds=all` to improve your debugging experience. However, Valgrind does not need to be "informed" of the fact that your library uses a different stack for each thread.

To inform valgrind of each of your stacks, you must do three things. First, you will need to include the valgrind header. Then, when you allocate space for your stack, you will need to register that address with valgrind. Finally, before you free your stack, you will need to deregister that address with valgrind. The different snippets of code are shown in Listing 3. **Before you submit your code**, you should comment out the line `#define DEBUG_USE_VALGRIND` in Listing 3.

```
1  // ...
2  // after the other #includes
3  #define DEBUG_USE_VALGRIND
4  #ifdef DEBUG_USE_VALGRIND
5  #include <valgrind/valgrind.h>
6  #endif
7  // ...
8  // in some function
9    char* stack = malloc(stack_size);
10   if (stack == NULL) {
11     fprintf(stderr, "malloc failed when trying to create a thread's stack");
12     return CSC369_ERROR_SYS_MEM;
13   }
14 #ifdef DEBUG_USE_VALGRIND
15   VALGRIND_STACK_REGISTER(stack, stack + stack_size);
16 #endif
17 // ...
18 // in some other function
19   stack_to_free = ...
20 #ifdef DEBUG_USE_VALGRIND
21   VALGRIND_STACK_DEREGISTER(stack_to_free);
22 #endif
23   free(stack_to_free);
```

Listing 3: Correctly restoring interrupt state for nested interrupts.

We do not recommend running valgrind on the tests themselves. Instead, consider creating your own simple example. Or, use one of the existing examples:

Running valgrind

```
wolf$ valgrind -s --leak-check=full --show-leak-kinds=all ./cmake-build-debug/examples/spin_and_join
==31328== Memcheck, a memory error detector
==31328== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==31328== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==31328== Command: ./cmake-build-debug/examples/spin_and_join
==31328==
TID(2) waited for TID(1), which exited with 32
...
TID(0) waited for TID(32), which exited with 63
==31328==
==31328== HEAP SUMMARY:
==31328==     in use at exit: 0 bytes in 0 blocks
==31328==   total heap usage: 66 allocs, 66 frees, 1,050,376 bytes allocated
==31328==
==31328== All heap blocks were freed -- no leaks are possible
==31328==
==31328== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

In order to handle the situation where the main thread (i.e., TID 0) returns implicitly, but there is still dynamic memory to clean up, you can use the atexit system call. This call allows you to specify a function (that takes no arguments) to call during "normal process termination". Another option is on_exit, which allows an argument to be passed.

However, you should take care in the body of your "cleanup function". Are there still running and ready threads? What about your other queues, are they all empty? Are interrupts enabled?