Assignment 3: Asynchronous I/O

Due Date: Friday, November 7th

Overview

At this point, our thread interface is well-defined enough to write programs that can interleave various computations by cooperatively yielding between them. This is interesting, but in order to write useful programs, we'd like to be able to interleave computation with blocking I/O.

The problem with this is that all of our user-level threads are running on a single kernel thread; if just one of our user-level threads does blocking I/O, the kernel thread becomes blocked, and thus *all* of our user-level threads becomes blocked.

What are some solutions to this problem?

- We could modify the Linux kernel to implement scheduler activations, though such an undertaking is beyond the scope of a this project.
- We could "wrap" blocking I/O calls such that they called into our user-level scheduler, which would then block the user-level thread while it either:
    - Used the Linux kernel thread interface (either directly through clone or indirectly through Pthreads to create a new kernel thread to perform the blocking I/O.
    - Used Linux's implementation of the POSIX asynchronous I/O interface to perform the I/O. (Interestingly, Linux's implementation of POSIX AIO uses exactly the Pthreads approach described above).

  In either case, once the I/O is complete, the scheduler would unblock the thread that initiated the blocking call and provide it with the results of I/O.

In this assignment, we will use the wrapper approach with POSIX AIO to implement a wrapper that emulates the `read` system call. `read` is used to perform blocking I/O on a file descriptor. Our wrapper should mirror the semantics of `read` as closely as possible, while only blocking one user-level thread.

At this point, our system is beginning to approach the "sweet spot" described in Adya, et al.: cooperative task management with automatic stack management.

Preparation

Successfully completing this assignment requires you to read, understand, and interact with a potentially unfamiliar interface.

Before continuing, thoroughly read the following Linux `man` pages:

- `aio(7)`
- `aio_read(3)`

- aio_error(3)
- aio_return(3)

You may find it helpful to brush up on the syntax and semantics of the synchronous `read` call:

- read(2)

The following will also be helpful (though you can safely ignore the section about "holes"):

- lseek(2)

Design Choices

Managing Notification: Signals vs. Polling

Now that we have a general idea of what we want to implement and what interface we are going to use, we have some important design choices to make. The first question is how we'd like our scheduler to be notified that an asynchronous I/O request has finished. In general, the choices are: having the kernel generate some kind of asynchronous software interrupt (this is called a "signal" in UNIX/Linux), or synchronously polling for a result.

Signals are naturally preemptive, and thus introduce the possibility of a signal handler entering into a race condition with the scheduler if they need to manipulate the same data. If we had a preemptive scheduler, we would already have machinery to deal with this, but since our system is cooperative, polling is easier to implement.

If you have a look at the `aio` man page, you'll see that an `aiocb` (asynchronous I/O control block) has several fields that specify the semantics of the request. One of these is `aio_sigevent`, which "specifies how the caller is to be notified when the I/O request completes." `aio_sigevent` has a subfield called `sigev_notify`, and its possible values are `SIGEV_NONE`, `SIGEV_SIGNAL`, and `SIGEV_THREAD`. These are described further in sigevent(7), but suffice to say that `SIGEV_NONE` is the correct choice for a polling approach. If `SIGEV_NONE` is selected, none of the other fields in `aio_sigevent` need to be specified.

See the man page for `aio_error` to see how you might poll for the status of an outstanding AIO request.

Managing Thread State: Blocking vs. Busy-waiting

Parallel to the question of how our scheduler is to be notified of a completed I/O request is how the scheduler should manage the state of the thread that performed the request. Typical descriptions of cooperative multithreading (e.g. in Adya et al.), suggest that the user-level thread should be blocked during the request.

What does it mean for a thread to be blocked? Generally it means that the thread is not enqueued on the ready list, but is instead placed on a separate waiting queue related to the specific blocking event (such as a lock or condition variable).

Let's say we decide to use special condition variables to represent pending I/O calls that have the `aiocb` as part of their structure. As part of the blocking call, the thread would enqueue one of these condition variables into a `pending_IO` list maintained by the scheduler, and then wait on that condition variable.

On each call into the scheduler, it would scan the `pending_IO` list, polling the status of each request. Completed requests would be removed from the list, and the scheduler would signal that condition variable.

This is a somewhat reasonable design, but it requires extra data structures (the `pending_IO` list and special condition variables) to implement, and involves potentially wasted work scanning the pending list on every scheduler call.

An important realization to make is that we already have all the data structures we need: the `aiocb` for a particular request is on the stack of the thread making that request! Instead of blocking, we can busy-wait, and use the thread itself to poll for the status of its request, yielding on every unsuccessful poll.

The busy-waiting solution involves no extra data structures or copying of data, and no extra queue manipulation. It is faster and easier to implement, and it is the design you should use for this assignment.

A note on efficiency

Note that any solution involving polling requires the scheduler to perform an *O(n)* operation, where *n* is the number of threads with a pending I/O request. In the blocking case, the scheduler must scan the list of blocked threads and poll the I/O request of each of them. In the busy-waiting case, the scheduler must run every waiting thread so it can poll its own I/O request.

Recall (from section 2.2 in Behren, et al.) that *O(n)* scheduler operations can introduce unacceptable latency to a workload where most threads are waiting for I/O (such as a web server). If we wanted a more scalable approach, we'd have to devise a solution that used signals instead of polling. You don't have to implement such a solution here, but you are asked to think about how you might do it in "Discussion", below.

Implementation

1. Start a new file, `async.c`. Because our design involves busy-waiting, if your scheduler's implementation from the previous assignment is sound, you will not have to modify any part of your previous work.

2. In addition to your `scheduler.h` file, `async.c` should `#include` the following libraries (there are some strong hints here about which functions you might need to use):

   `<aio.h>` For `struct aiocb`, `aio_read`, `aio_error`, `aio_return`.
   `<errno.h>` For `EINPROGRESS`.
   `<unistd.h>` For `lseek`, `SEEK_CUR`, `SEEK_END`, `SEEK_SET`.
   `<string.h>` For `memset`.

   You will also have to compile your code with the `-lrt` flag, for example:

   ```
   gcc switch.s scheduler.c queue.c async.c main.c -lrt
   ```

   Or more compactly (provided no other code is in your directory):

   ```
   gcc *.s *.c -lrt
   ```

3. Write a function, `read_wrap`. It should have exactly the same signature as `read`:

```
ssize_t read_wrap(int fd, void * buf, size_t count) {
   // your code here!
}
```

   Use the AIO interface to create an AIO control block and initiate an appropriate asynchronous read. Yield until the request is complete.

   `read_wrap` should have as close to the semantics of `read` as possible. That is, it should return the same value that `read` would have, and put the same result into `buf` that `read` would have.

   Furthermore, if the file is seekable, `read_wrap` should start reading from the current position in the file, and then seek to the appropriate position in the file, just as `read` would have. This is arguably the most difficult part of this assignment, since `aio_read` does not seek automatically.

Testing

There is not very much code to write for `read_wrap`, so testing is a significant portion of this assignment.

Correct Scheduling Behavior

You must write a test that proves that `read_wrap` correctly only suspends the current thread, allowing other threads to continue during the I/O.

The easiest way to do this is to re-use your test code from assignment 2, but add a thread that reads from standard input (file descriptor 0), which should cause the thread to wait until the user inputs something at the terminal and presses enter. Ensure that your computations proceed while the reading thread is busy-waiting, and that the thread properly resumes when input arrives.

Correct Reading Semantics

Furthermore, it is your responsibility to ensure that your implementation of `read_wrap` mirrors the semantics of `read` as closely as possible. This means that you must test that `read_wrap` correctly returns the same number of bytes as `read`, and that `read_wrap` returns an error whenever `read` returns an error.

You must also test that `read_wrap` has the same seeking behavior as `read`. Standard input is not seekable, so you will have to run tests on a real file. See `open(2)` for more on how to create a new file descriptor from a path name. Test code that uses `open` must `#include <fcntl.h>`.

Snake Game!

To show off the utility of our library so far, I've written a "snake game" that uses it.
Here is the source code. In addition to writing your own tests, you may try linking your library with the game code to see if you can get it to compile and run. Think of it as your reward for a successful implementation!

Discussion

Put your responses to these questions in your writeup.

1. In class, we've talked about how to synchronize in in a cooperative multi-threaded environment. Specifically, we've said that not yielding (or calling a function that yields) in a critical section is just like acquiring a lock for that critical section. Is this still true?

2. Briefly discuss a potential design for a scheduler that uses asynchronous software interrupts (signals) and true blocking instead of polling and busy-waiting to handle notification of I/O completion. What information should be passed to the signal handler? What information should the signal handler manipulate? Signals turn your scheduler into a concurrent program. Where do we have to look out for problems such as race conditions, deadlock, etc?

What To Hand In

If you are working with a partner, please prepare just one submission for the both of you. Make sure your (and your partner's) names are included in each file you submit.

You should submit:

1. All code, including tests. This should include:
    - `switch.s`
    - `queue.h`
    - `queue.c`
    - `scheduler.h`
    - `scheduler.c`
    - `async.c`
    - `main.c`
2. The plain-text file you used to test seeking behavior.
3. A brief written report, including:
    1. A description of what you did and how you tested it.
    2. Your response to the questions in "Discussion", above.
    3. ~~Your high score in the snake game.~~ (just kidding!)

Please submit your code files *as-is*; do not copy them into a Word document or PDF.
Plain text is also preferred for your write-up.
You may wrap your files in an archive such as a .tar file.

Email your submission to the TA at kstew2 at cs.pdx.edu on or before the due date. The subject line should be "CS533 Assignment 3".

Need Help?

If you have any questions or concerns, or want clarification, feel free to contact the TA by coming to office hours or sending an email.

You may also send an email to the class mailing list, but please pose discussion questions only; **do not**

email any code to the class mailing list! (It is okay to send code directly to the TA, however).