CSCI 4061: Finale

Chris Kauffman

Last Updated:
Mon May 3 03:59:13 PM CDT 2021

Logistics

P2: Due Tue 5/04

Tests posted, Gradescope Open, more comments in a moment

Today

- Evals Reminders
- P2 Discussion
- Finale / Review

Date	Event
Mon 5/03	Last Lecture HW 13 / Lab13 Due Univ Evals Due
Tue 5/04	Kauffman OHs P2 Due
Thu 5/06	Kauffman OHs P2 Late Deadline
Mon 5/10	Final Exam 9am - 11pm 4-6pm Discord Qs

Questions on anything?

P2 Testing: A mess in several parts

1. I'm tired and didn't have a lot of time+energy to work on them. Problems always occur when that's the case.
2. Gradescope Labs Home Linux: old kernel bugs in `sem_open()` and `dl_open()` which required special exceptions in Valgrind
3. I forgot to include a file on releasing and also did get the updated zip file posted on discovering this. Thanks to those who were patient enough to notify me of this.
4. Assuming ordering of events between programs is a Race Condition and (almost) every test assumes `sleep()` can achieve this
5. May need to adjust `TICKTIME` parameter to be longer to get more stability at the expense of longer test runtimes.
6. May have to submit a couple times to Gradescope to get a "good" run if tests pass locally

I am deeply sorry that this is such a pain for everyone and I'll be in office hours **Tue/Thu 3pm** to help folks get the project done.

Reminder: Course Evals

Official UMN Evals

```
CSCI 4061   : Intro to Operating Systems
Lecture 001 : Kauffman
```

- Official UMN Evals are done online this semester
- Available here: https://srt.umn.edu/blue
- Due Mon 5/03, last day of semester

What have we done?

Unix Systems Programming

API of Unix system for files, processes, signals, IPC, threads, sockets, memory

Glimpses of OS Internals

Process accounting, file representation, communication buffers

Concurrency and Communication

Protocols to allow distinct operators to cooperate/communicate without deadlocking

C Programming

Memory allocation, pointers, structs, conventions for errors

Did I miss anything?

Further Coursework

- **CSCI 5103 Operating Systems**: Study internal design issues associated with operating systems, handling hardware, tradeoffs on different approaches to management, theoretical algorithms around resource coordination.
- **CSCI 4211 Introduction to Computer Networks**: Learn more about communication protocols, hardware/software architecture of the Internet, operating system supports for networks.
- **CSCI 4271W Development of Secure Software Systems**: Focus on security issues, methods to circumvent OS/hardware protections and how ensure safety in programs, incorporating security features into system design.
- **CSCI 5143 Real-Time and Embedded Systems**: Small systems often lack an OS and fancy hardware, more direct interactions with hardware, must manage resources in your own programs, teaches much about what the OS does as usually less is provided in embedded systems.

Further Reading

- INTERNALS: The Design of the UNIX Operating System by Maurice A. Bach : Step-by-step treatment of the original design internals of the Unix OS. Lots of pictures and great discussion of concurrency issues in the kernel.
- DESIGN: The Art of Unix Programming by Eric S. Raymond : Fantastic philosophical and pragmatic discussion of how to build systems that work especially in the Unix environment. (free online)
- HUMANS: Coders at Work: Reflections on the Craft of Programming by Peter Seibel : Fascinating interviews with notable programmers who got the job done including AI giant Peter Norvig, Scheme Inventor Guy Steele, original Unix inventor Ken Thompson, and CS godfather Donald Knuth.

Final Exam

Logistics

- Mon 5/10
  - 9am - 11pm on Gradescop
  - 4-6pm Questions on Discord
- 5-6 "sides of paper" in 120min
  - Midterms were 4 sides of paper in 80min
- Comprehensive, combination of coding, analysis, short answer
- Open Resource as were the midterm exams

Topics Request

Any particular topics folks would like to discuss prior to review questions?

Review Problem 1

A binary file stores many `mesg_t` structs in it; these structs have the definition:

```
typedef struct {
  int kind;
  char name[256];
  char body[1024];
} mesg_t;
```

Define the following function

```
int print_all(char *filename, char *user_name)
// Opens the given filename which stores a sequence of binary mesg_t
// structs, scans the file for mesg_t's with a name field that matches
// the given user_name and prints their bodies. Closes the file and
// return the number of messages found for the given the user_name.
```

Bonus point: describe a 2nd method you could use to print the output you have using a different I/O mechanism from the one you coded.

Answers:

```
 1  // use read() to access data in file
 2  int print_all(char *filename,
 3                char *user_name)
 4  {
 5    int fd = open(filename, O_RDONLY);
 6    mesg_t msg;
 7    int count = 0;
 8    while(1){
 9      int nbytes = read(fd, &msg,
10                        sizeof(mesg_t));
11      if(nbytes==0){
12        break;
13      }
14      if( strcmp(msg.name, user_name) == 0 )
15      {
16        printf("%s\n",msg.body);
17        count++;
18      }
19    }
20    close(fd);
21    return count;
22  }
```

```
 1  // use mmap() to access data in file
 2  int print_all(char *filename,
 3                char *user_name)
 4  {
 5    int fd = open(filename, O_RDONLY);
 6    struct stat statbuf;
 7    fstat(fd, &statbuf );
 8    int len = statbuf.size / sizeof(mesg_t);
 9    mesg_t *mesgs =
10      mmap(NULL,statbuf.size,
11           PROT_READ, MAP_PRIVATE,
12           fd, statbuf.size);
13    int count = 0;
14    for(int i=0; i<len; i++){
15      if(strcmp(mesgs[i].name, user_name)==0)
16      {
17        printf("%s\n",mesgs[i].body);
18        count++;
19      }
20    }
21    munmap(mesgs);
22    close(fd);
23    return count;
24  }
```

Review Problem 2

We studied several different mechanisms to coordinate threads that are provided in the PThreads library. Describe the similarities and differences between each of the following coordination tools and how they are best used

1. Mutex (`pthread_mutex_t`)
2. Spin Lock (`pthread_spinlock_t`)
3. Condition Variable (`pthread_cond_t`)

Answers:

1. **Mutex (`pthread_mutex_t`)**: Like a semaphore, used to lock/unlock a resource/protect a critical code region, thread that locks moves ahead, if a thread can't acquire a mutex, it blocks until the thread that owns the mutex unlocks it

   See: `pthreads_picalc_mutex.c` and `odds_evens_busy.c`

2. **Spin Lock (`pthread_spinlock_t`)**: Similar to a mutex except locking is done "busily": if thread cannot immediately acquire the spinlock, it will "spin" using 100% CPU until the spinlock is acquired, receives lock faster at the cost of higher CPU usage

   See: `time_spinlock.c`

3. **Condition Variable (`pthread_cond_t`)**: More like a "notification queue", used to notify other threads that some condition has arisen (`pthread_cond_signal()`), threads can block until they receive such notification (`pthread_cond_wait()`), always used in conjunction with a mutex to avoid wasteful lock/unlock cycles

   See: `odds_evens_two_condvars.c` and `produce_consumer.c`

All codes are in `13-threads-code.zip` associated with Threads lecture.

Review Problem 3

The blather server `bl_server` was required to use the `poll()` system call to check whether its various input sources were ready. The general pattern was as follows.

```
// REQUIRED version
repeat {
  poll() to check join/client FIFOs

  if join is ready{
    read join request, process it
  }

  for each client C{
    if client C is ready{
      read message from C, process it
    }
  }
}
```

A much simpler pattern of I/O would not use `poll()` such as the below.

```
// ALTERNATIVE version
repeat {
  read join request, process it

  for each client C {
    read message from C, process it
  }
}
```

Discuss the differences in behavior between these two. Why was the first pattern required in Blather when it is more complex?

Answers:

In the ALTERNATIVE version, the top of each server loop will `read()` from the join FIFO. At first this may seem to work as a client will be able to join. However, if a client joins successfully, it will likely send a message which will not be immediately accepted by the server. This is likely due to the server again `read()`'ing from the join FIFO which blocks until another client joins. Only then will the server enter the loop to check for client inputs. The REQUIRED version avoid this by using `poll()` which blocks only when no input sources are available and returns immediately when any source is ready. The specific source that is ready is indicated in the data `poll()` modifies allowing the server to `read()` without blocking at each step.

Review Problem 4

`bl_server` uses `poll()` to detect which input sources are ready while `bl_client` uses multiple threads to handle its input sources. **Discuss** using multiple threads in `bl_server` instead of `poll()` to handle its various input sources. In your answer describe the following:

1. How many threads will be required in `bl_server` to handle its input sources?
2. When will threads be created and ended (canceled)?
3. What would each server thread DO (deal with joining, deal with one client or multiple clients, etc.)?
4. What kind of coordination needs to exist between server threads to facilitate broadcast operations (writing to multiple client output)?
5. What kind of coordination needs to exist between server threads for client joining and departing?

Consider carefully the shared data structures of the server in your answer.

Answers:

- Discuss together during lecture
- Worth a Piazza post : discuss online

Conclusion

It's been a hell of a semester.
I'm proud of all of you.
Keep up the good work.
Stay safe. Happy Hacking.