Logistics

Reading
- Stevens/Rago Ch 15.6-12
- Wikip: Dining Philosophers

Goals
- Project Plans
- File Append Problem
- Semaphore Basics
- Shared Memory
- Message Queues
- Dining Philosophers

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed 3/31</td>
<td>IPC ShMem</td>
</tr>
<tr>
<td></td>
<td>IPC MsgQ</td>
</tr>
<tr>
<td>Mon 4/5</td>
<td>Spring Break</td>
</tr>
<tr>
<td></td>
<td>No Class</td>
</tr>
<tr>
<td>Mon 4/12</td>
<td>Review</td>
</tr>
<tr>
<td>Wed 4/14</td>
<td>Exam 2</td>
</tr>
</tbody>
</table>

Lab 11
- Email lookup server/client
- Use of FIFO to communicate
- Difficult to write tests for it - sorry for any Gradescope problems
- How did it go?
Don’t have time for 3 projects anymore which is Kauffman’s fault

I apologize for this mistake. I have experienced some personal problems which have interfered with my ability to adequately prepare a solid Version Control project. I regret that I was not able to provide a project that puts the topics we have discussed into practical use.

P2: release after Exam 2

Focus on Interprocess Communication: a local Chat Server/Client

Same size as P1, Worth 20% of grade

Opportunities for some Makeup Credit
Exercise: Forms of IPC we’ve seen

- Identify as many forms of *inter-process communication* that we have studied as you can
- For each, identify *restrictions*
  - Must processes be related?
  - What must processes know about each other to communicate?
- You should be able to name at least 3-4 such mechanisms
Answers: Forms of IPC we’ve seen

- Pipes
- FIFOs
- Signals
- Files
- Maybe mmap()’ed files
Inter-Process Communication Libraries (IPC)

- Signals/FIFOs allow info transfer between unrelated processes
- Neither provides much
  - Communication synchronization between entities
  - Structure to data being communicated
  - Flexibility over access
- **Inter-Process Communication Libraries (IPC)** provide alternatives
  1. Semaphores: atomic counter + wait queue for coordination
  2. Message queues: direct-ish communication between processes
  3. Shared memory: array of bytes accessible to multiple processes

Two broad flavors of IPC that provide semaphores, message queues, shared memory...
Which Flavor of IPC?

System V IPC (XSI IPC)

- Most of systems have System V IPC but it’s kind of strange, has its own namespace to identify shared things
- Part of Unix standards, referred to as XSI IPC and may be listed as optional
- Most textbooks/online sources discuss some System V IPC. Example:
  - Stevens/Rago 15.8 (semaphores)
  - Robbins/Robbins 15.2 (semaphore sets)
  - Beej’s Guide to IPC

POSIX IPC

- POSIX IPC little more regular, uses filesystem to identify IPC objects
- Originated as optional POSIX/SUS extension, now required for compliant Unix
- Covered in our textbooks partially. Example:
  - Stevens/Rago 15.10 POSIX Semaphores
  - Robbins/Robbins 14.3-5 POSIX Semaphores

- Additional differences on StackOverflow

We will favor POSIX
Exercise: Concurrent Appends to a File

C code to append to a file some number of times.

```c
1 // append_loop.c
2 int main(int argc, char *argv[]){
3     char *filename = argv[1];
4     int count = atoi(argv[2]);
5     int key = atoi(argv[3]);
6     int fd = open(filename,
7         O_CREAT | O_RDWR ,
8         S_IRUSR | S_IWUSR);
9
10    char line[128];
11    sprintf(line, "%04d\n", key);
12    int len = strlen(line);
13
14    for(int i=0; i<count; i++){
15        lseek(fd, 0, SEEK_END);
16        write(fd, line, len);
17    }
18
19 } 
20 close(fd);
21 return 0;
22 }
```

Shell code demos its use. What’s wrong with the last count?

```bash
> ./a.out
usage: ./a.out <filename> <count> <key>
> ./a.out thefile.txt 100 5555
> wc -l thefile.txt
100 thefile.txt
> ./a.out thefile.txt 100 7777
> wc -l thefile.txt
200 thefile.txt
> sort thefile.txt | uniq -c
100 5555
100 7777
> rm thefile.txt
> for i in $(seq 10); do
    ./a.out thefile.txt 100 $i &
done
> wc -l thefile.txt
732 thefile.txt
```
Concurrency Principles

Atomic Action

- Cannot be divided; will run completely before any other action taken. Some system calls are atomic like ...
- nbytes = write(fd, data, len); is atomic: nbytes of data written in sequence, data from other write() calls before/after but NOT in the middle
- lseek() is atomic: modifies file position in kernel data structure

Race Condition

- Outcome depends on the ordering of unpredictable events such as the OS scheduler interrupting a process
- Race Conditions are bad: unlucky timing causes unpredictable behavior, bugs that only occasionally occur
Race Condition in append_loop.c

```
FILE PROC1 key=5555 PROC2 key=7777

len=15
5555
5555 lseek(fd, 0, SEEK_END);
7777 // pos = 15
<--------write(fd, line, len);

len=20
5555
5555
7777 lseek(fd, 0, SEEK_END);
5555 // pos = 20
<------------------------write(fd, line, len);
```

All appears well BUT cannot guarantee that lseek() / write() happen uninterrupted

- Individually atomic
- Combination is not
Race Condition in `append_loop.c`

```c
FILE PROC1 key=5555 PROC2 key=7777

len=25
5555 lseek(fd, 0, SEEK_END);
5555 // pos = 25
7777 lseek(fd, 0, SEEK_END);
7777 // pos = 25

<--------write(fd, line, len);

len=30
5555
5555
7777
7777 // pos = 25
5555<------------------------write(fd, line, len);

len=30
5555
5555
7777
7777
7777 # Overwritten

Result: 1 line is lost as the `lseek()` between process is not coordinated
```
Exercise: Solve this with Current IPC

Suggest a way to solve this problem with current IPC mechanisms

Start an arbitrary number of processes. Each repeatedly appends a given key to a given file. All keys must be present at the end.

- Describe new / old processes
- Describe new / old code and IPC to be used

Hint: where have we recently seen a bunch of entities that all want access to data? How were these requests coordinated?
**Answers: Solve this with Current IPC**

Uses a **FIFO to coordinate multiple writers**

**Manager Process**

- Only the manager writes to `thefile.txt`
- Starting the manager creates a FIFO; manager `read()`’s from the FIFO, appends text to the end of the file

**Writer Processes**

- Writer processes `write()` into the FIFO (not `thefile.txt`)
- FIFOs automatically serialize data: no chance for loss as OS controls the singular read/write positions

**Familiar but Unsatisfactory**

- Similar to `em_server / em_client` from Lab/HW
- Works and requires new IPC mechanisms BUT...
- Dissatisfying: **must split code into manager/writer**. Would like a solution without a central manager.
Locking the Critical Region

Critical Region

- Code sequence lseek(); write() is a Critical Region: not atomic, unsafe to have multiple entities in it at the same time
- Typically protect these with a coordination mechanism, a lock for the critical region

OS Locking Mechanisms

- **Semaphore**: general purpose locking mechanism associated with multi-process programming
- **Mutex**: locking mechanism associated with threaded programming
- **File Locks**: lock all or portions of a file, always
Semaphore History

Semaphore: *noun*
A system of sending messages by holding the arms or two flags or poles in certain positions...
– Oxford Dictionary

Semaphore: *(computing)*
In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple processes and avoid critical section problems in a concurrent system such as a multitasking operating system.

The semaphore concept was invented by Dutch computer scientist Edsger Dijkstra...
– Wikipedia
Semaphore Basics: 3 Parts

Counter Variable variable
Semaphores have an integer value indicating how much of a resource is available

▶ S=0: none left
▶ S>0: some available

Most common case is S=1 (available) or S=0 (in-use)

Atomic Operations

▶ Acquire: If S>0, decrement; Else, enter wait-queue and block
▶ Release: Increment S, notify wait-queue of availalibility

Wait Queue
Modern semaphores include a wait-queue. If S==0, Acquire will cause an entity (process) to enter the wait-queue and block.
Posix Implementation of Semaphores

```c
sem_t *sem =
    sem_open("/the_sem", O_CREAT, S_IRUSR | S_IWUSR);
// abstract type sem_t representing semaphores
// file-like semantics with open, semaphore name, flags, permissions

// Note: "the_sem" may or may not appear in the file system somewhere
// Under Linux, will be at /dev/shm/the_sem

sem_init(sem, 1, 1); // Initialize the semaphore value
    // | +------> Initial counter value = 1
    // ++--------> Share among Processes (1: Processes, 0: Threads)

sem_wait(sem);
// ACQUIRE the semaphore; block and queue up if not available

// CRITICAL REGION

sem_post(sem);
// RELEASE the semaphore; notifies any queued processes of availability

sem_close(sem);
// file-like semantics: close when process is finished using it

sem_unlink("/the_sem");
// POSIX named semaphores have kernel persistence: if not removed by
// sem_unlink(), a semaphore will exist until the system is shut down.
```
Examine: append_file_sem.c

Examine and experiment with `append_file_sem.c` which solves coordinates appends using a POSIX semaphore.

Look for use of semaphore functions like

- Opening
- Unlinking, initializing
- Acquiring / Releasing
- How the critical region is protected

```
> gcc -g append_loop_sem.c -lpthread
> ./a.out -init 1 1 initializing

> for i in $(seq 10); do
    ./a.out thefile.txt 100 $i &
    done

> wc -l thefile.txt
1000 thefile.txt # ALL THERE!

> sort thefile.txt | uniq -c
100 0001 # ALL KEYS
100 0002 # FROM ALL
100 0003 # PROCESSES
100 0004
100 0005
100 0006
100 0007
100 0008
100 0009
100 0010

> ./a.out -unlink 1 1 unlinking
```
File Append Alternatives

Semaphores give general purpose coordination but the special case of coordinating file appends have several other simpler solutions.

**POSIX File Locks**

- See `append_loop_lockf.c`
- `lockf()`: apply, test or remove a POSIX lock on an open file
- Protect critical region via
  ```
  lockf(fd, F_LOCK, 0);
  lseek(fd, 0, SEEK_END);
  write(fd, line, len);
  lockf(fd, F_ULOCK, 0);
  ```
- Major Plus: no Init/Unlink funny business
- Drawback: Lock is tied to a file, Semaphores are independent

**O_APPEND Flag**

- See `append_loop_oappend.c`
- `open(..., O_APPEND, ...)`: opens a file in append mode:
  ```
  “The file offset shall be set to the end of the file prior to each write().” – man open(3)
  ```
- Major Plus: no locks, semaphores, or other funny business
- Major Drawback: only works for appending to the end of files; Not Applicable to coordinating any other activity
**Shared Memory Segments**

- An memory area that can be shared by multiple processes
- POSIX shared memory outlives a process like a file BUT with no permanent storage
  - Must clean up / unlink Shared Mem manually
  - Shared Mem Contents unreliable across power off/on
- **Examine** `shmdemo_posix.c` to see how that works much like a memory mapped file
Exercise: Shared Memory Coordination

- Creating shared memory is relatively easy
- Like files, Coordinating shared memory is not automatic
- Consider shared_flip.c
  - Shared memory of all “00000” or “1111”
  - shared_flip -flip flips all characters (0 → 1, 1 → 0)
- What happens if multiple programs simultaneously try to flip bits?
- How does one prevent “corruption” of that data?
- Experiment noting that a command like
  \[
  \text{for } i \text{ in } $(seq 100); \text{ do } ./\text{shared_flip} \text{ -flip } & \text{ done}
  \] will start 100 identical processes as background jobs
Answers: Shared Memory Coordination

- No file to lock so `flock()` wouldn’t work
- Not appending so `O_APPEND` won’t cut it
- A **semaphore** allows coordination of bit flipping through `sem_wait()` / `sem_post()` to protect the critical region

```c
1 // No Coordination: Errors
2
3 printf("flipping bits\n");
4 for(int i=0; i<SHM_SIZE-1; i++){
5   if(shared_bytes[i] == '0'){
6       shared_bytes[i] = '1';
7   }
8 }
9 else if(shared_bytes[i] == '1'){
10       shared_bytes[i] = '0';
11 }
12 }
13
14 // Coordinate via Semaphore
15
16 sem_t *sem =
17   sem_open(sem_name,O_CREAT,S_IRUSR|S_IWUSR);
18 sem_wait(sem); // lock semaphore
19 printf("flipping bits\n");
20 for(int i=0; i<SHM_SIZE-1; i++){
21   if(shared_bytes[i] == '0'){
22       shared_bytes[i] = '1';
23   }
24 }
25 else if(shared_bytes[i] == '1'){
26       shared_bytes[i] = '0';
27 }
28 }
29 sem_post(sem); // unlock sem
30 sem_close(sem);
```
Shared Memory vs mmap’d Files

- Recall Memory Mapped files give direct access of OS buffer for disk files
- Changes to file are done in RAM and occasionally sync()’d to disk (permanent storage)
- POSIX Shared Memory segment cut out the disk entirely: an OS buffer that looks like a file but has no permanent backing storage
- Which to pick?
  - Shared Memory when data does not need to be saved permanently and/or syncing would costly
  - Memory Mapped File when data should be saved permanently
- Related concept: RAM Disk, a main memory file system, high performance with no permanence
Practice Problem: A Semaphore Application

- Process a “jobs” file with a list of shell commands to run
  - seq 100000
  - gcc --version
  - du . -h
  - ls
  - ls -l
  - date
  ...

- Start multiple ’runners’ which execute lines from the jobs file
  > runner jobs.txt & runner jobs.txt &
  # starts 2 runners to work on jobs.txt

- Runners read file lines, execute jobs, mark as done
  D seq 100000
  D gcc --version
  R du . -h
  D ls
  R ls -l
  - date
  ...

- Will provide initial version of this
- To prevent duplication of job running, add coordination to prevent duplicate jobs
Posix Message Queues

- Implements basic send/receive functionality through shared memory
- Message Queues share much with FIFOs
  - `mq_send()` is similar to `write()` to a FIFO
  - `mq_receive()` is similar to `read()` from a FIFO
  - Known global name of a message queue ~ name of FIFO file
- Differences from FIFOs
  - FIFOs/Pipes have a fixed total size (64K)
  - FIFOs allow `read()`/`write()` of arbitrary # of bytes
  - Message Queues limit #messages and max size of messages on queue
  - Message Queues send/receive individual messages
Kirk and Spock: Talking Across Interprocess Space

- Demo the following pair of simple communication codes which use Posix Message Queues.
- Examine source code to figure out how they work.

See `msg_kirk_posix.c` and `msg_spock_posix.c`
Email Lookup with Message Queues

- Recent HWs build an email lookup server using FIFOs
- Another HW compare it to an approach that uses Message Queues
- Worth of study to see the many similarities between FIFOs/Message Queues and a few of the differences between them
- Such contrast between IPC mechanisms make for good Exam questions
Linux shows Posix IPC objects under /dev/shm

```bash
> gcc -o philosophers philosophers_posix.c -lpthread
> ./philosophers
Swanson 0: wants utensils 0 and 1
Swanson 2: wants utensils 2 and 3
Swanson 1: wants utensils 1 and 2
...
Swanson 3 (egg 10/10): leaving the diner
pausing prior to cleanup/exit (press enter to continue)
while you're waiting, have a look in /dev/shm
  C-z
[1]+ Stopped ./philosophers

> ls -l /dev/shm
total 20K
-rw------- 1 kauffman kauffman 32 Apr 1 21:36 sem.utensil_0
-rw------- 1 kauffman kauffman 32 Apr 1 21:36 sem.utensil_1
-rw------- 1 kauffman kauffman 32 Apr 1 21:36 sem.utensil_2
-rw------- 1 kauffman kauffman 32 Apr 1 21:36 sem.utensil_3
-rw------- 1 kauffman kauffman 32 Apr 1 21:36 sem.utensil_4

> fg
./philosophers

> ls -l /dev/shm
total 0
```

/dev/shm is a Linux convention, shard memory under as well, message queues under /dev/mqueue
More Resources IPC

System V IPC

► http://beej.us/guide/bgipc/
► http://www.tldp.org/LDP/tlk/ipc/ipc.html

General Overview

Model Problem: Dining Philosophers

- N Philosophers with N Chopsticks between them
- Philosophers “Algorithm”
  - Think for a while
  - Get adjacent chopsticks
  - Eat for a while
  - Replace Chopsticks
  - Repeat
- Models concurrent processes/thread acquiring multiple resources

Source: Introduction to RTOS Part 10 - Deadlock and Starvation | Digi-Key Electronics
Exercise: Coding Dining Philosophers

Central philosopher algorithm is
- Think for a while
- Get adjacent chopsticks
- Eat for a while
- Replace Chopsticks
- Repeat

Questions:
1. What can be used to model “chopsticks”?
2. How does one avoid deadlock?
Answers: Coding Dining Philosophers

1. Model chopsticks with **semaphores**: only one process can acquire them at a time; the other blocks.
2. All philosophers get right chopstick (lower number) first EXCEPT last philosopher: go left first
   ► Breaks the cycle that would create deadlock

See philosophers_posix.c for demonstration code