CSCI 4061: Inter-Process Communication

Chris Kauffman

Last Updated:
Thu Apr 4 09:40:02 CDT 2019

Logistics

Reading

- Robbins and Robbins Ch 15.1-4
- OR Stevens/Rago Ch 15.6-12

Goals

- Protocols for Cooperation
- Basics of IPC
- Semaphores, Message Queues, Shared mem

Lab08: FIFO, protocol

How did it go?

Project 2

- Kauffman not happy with delay
- You will be happier with result

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

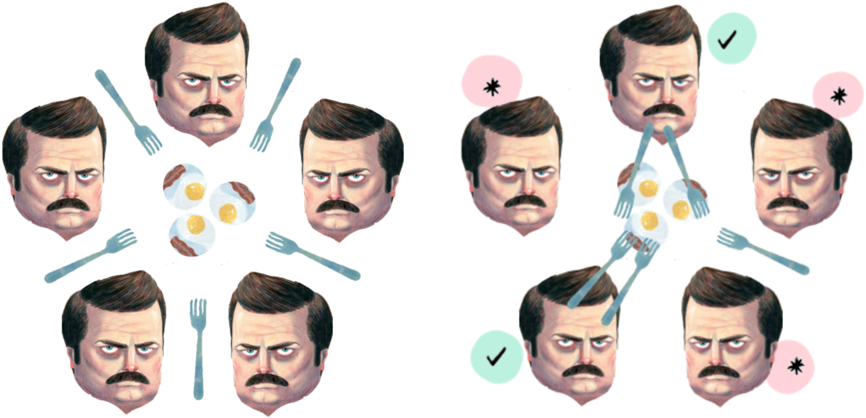
POSIX IPC

- 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

Model Problem: Dining "Philosophers"

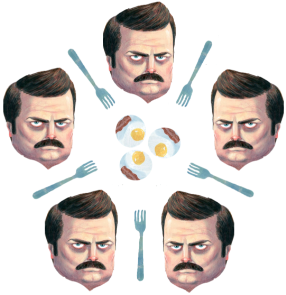
- Each Swansons will only eat with two forks
- JJ's only has 5 forks, must share
- After acquiring 2 forks, a Swanson eats an egg, then puts both forks back to consider how awesome he is
- Algorithms that don't share forks will lead to injury

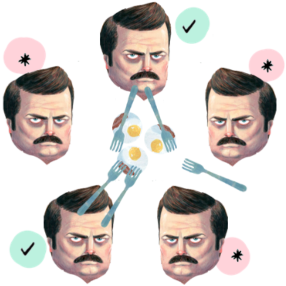


Source: Aditya Y. Bhargava,

Exercise: Protocol for Dining "Philosophers"

- Each Swansons will only eat with two forks
- JJ's only has 5 forks, must share
- Swanson's pick up one fork at a time
- After acquiring 2 forks, a Swanson eats an egg
- After eating an egg a Swanson puts both forks considers how awesome he is, repeats
- Swanson leaves after eating sufficient eggs
- Is there any potential for deadlock? How can this be avoided?
- Is there any chance for starvation?





Answers: Protocol for Dining "Philosophers"

Deadlock: All try for left fork first

- Each Swanson acquires left fork: cycle
- Each Waits forever for right fork

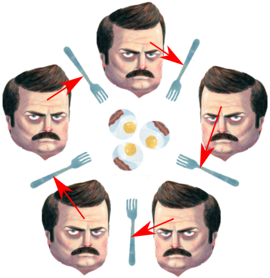
Dijkstra: One Swanson goes Right first

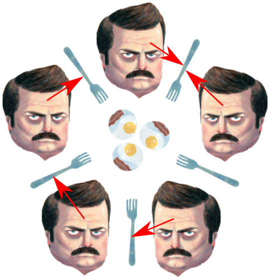
- Breaks the cycle so no deadlock possible
- Generalization establishes a **partial ordering** for each process to acquire resources, can prove lack of deadlocks

Starvation?

- A Swanson may wait indefinitely to get both forks, resource **starvation**
- Requires introduction of **priority** and communication to fix (Chandy/Misra Solution)

Deadlock





Semaphore

Source: Wikipedia Railway Sempahore Signal

- A counter variable with atomic operations
- **Atomic operation**: not divisible, all or none, no partial completion possible
- Used to coordinate access to shared resources such as shared memory, files, connections
- Typically allocate one semaphore per resource and acquire all that are needed

Activity: Dining "Philosophers" with Semaphores

Examine the dining philosophers code here:

http://www.cs.umn.edu/~kauffman/4061/philosophers_posix.c

Use the Man Pages here:

http://man7.org/linux/man-pages/man7/sem_overview.7.html

Find out how the following are done:

1. What does a POSIX semaphore look like?
2. How does one create a POSIX semaphore?
3. What calls are used to "acquire" and "release" a POSIX semaphore?
4. What happens when multiple processes modify the same semaphore?
5. How are semaphores used to coordinate use of forks?
6. How is deadlock avoided in the code?

Lessons Learned from `philosophers_posix.c` (1/2)

Posix Semaphores

- POSIX semaphores are single integer values with atomic operations
- Semaphore operations are guaranteed to be **atomic**: only one process can increment/decrement at a time, function as efficient **locks**
- `sem_t *sem = sem_open(name,...);` is used to obtain a semaphore from the operating system. Uses *named semaphores* which are managed by OS, shared between processes
- `sem_wait(sem);` wait until semaphore is non-zero, then atomically decrement/lock it
- `sem_post(sem);` increment/unlock a semaphore and schedule a process waiting on it to run
- `sem_close(sem);` to stop using a semaphore

Lessons Learned from `philosophers_posix.c` (2/2)

Dining Philosophers

- In Dining Philosophers, one semaphore per utensil, acquire both to eat
- Circular Deadlock avoided via one Philsopher acquiring in a different order
  - Philosopher N: Get right utensil, then left utensil
  - Other Philosopher: Get left utensil, the right utensil

Alternative: System V Semaphores

- File `philosophers_sysv.c` implements same problem with System V semaphores which look stranger than POSIX
  - Always come in an array of multiple semaphores
  - Operate atomically on the array: can incr/decr multiple semaphores at once
  - Requires use of structs to perform operations
  - Provide some other forms of synchronization such as waiting until a semaphore reaches 0
  - C calls such as `semget()`, `semctl()`, `semop()`
- Net effect is the same: each Philosopher locks a utensil by atomically decrementing and incrementing semaphores

The Nature of a Semaphore

- As seen, semaphores have several component parts that the OS manages
  1. A value, usually representing quantity of resources available, often 1 or 0
  2. A locking mechanism allowing atomic operations on the value
  3. A **wait queue** of processes (or threads) that are blocked until the semaphore becomes non-zero
- Simple use of semaphores treats than as an efficient **lock** to hold a resource or protect a critical region code
- Later will discuss each component separately in context of threads
  - Locks are **Mutexes**
  - Wait queues are **Condition variables**
  - Values can be any variable in memory
  - SO: cucufrog on Condition Variables vs Semaphores

Linux shows Posix IPC objects under /dev/shm

```
> 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

Exercise: Concurrent Appends to a File

- C code to the right appends a string to `the_file.txt`
- Shell code below launches 100 processes to append
- What's with the results of `wc`?

```
> gcc -o append_clobber append_clobber.c
> for i in $(seq 100); do
    append_clobber $i &
  done
[3] 18724
[4] 18725
...
[99]-  Done append_clobber $i
[100]+  Done append_clobber $i
> wc the_file.txt
 97  96 281 the_file.txt
```

```c
// append_clobber.c
int main(int argc, char *argv[]){
  if(argc < 2){
    printf("usage: %s <word>\n");
    return 1;
  }

  char *word = argv[1];
  char *filename = "the_file.txt";

  int fd = open(filename,
                O_CREAT | O_RDWR ,
                S_IRUSR | S_IWUSR);
  lseek(fd, 0, SEEK_END);

  int n = strlen(word);
  word[n] = '\n';
  write(fd, word,n+1);
  close(fd);

  return 0;
}
```

Exercise: Concurrent Appends to a File

- `append_clobber.c` does not coordinate writes to the end of `the_file.txt` leading to some data to be overwritten and lost
- Fix this using **semaphores**
- Research alternatives that also allow safe appends to files

```c
// append_clobber.c
int main(int argc, char *argv[]){
  if(argc < 2){
    printf("usage: %s <word>\n");
    return 1;
  }

  char *word = argv[1];
  char *filename = "the_file.txt";

  int fd = open(filename,
                O_CREAT | O_RDWR ,
                S_IRUSR | S_IWUSR);
  lseek(fd, 0, SEEK_END);

  int n = strlen(word);
  word[n] = '\n';
  write(fd, word,n+1);
  close(fd);

  return 0;
}
```

Answers: Concurrent Appends to a File

Semaphore Solution

- `append_sem.c` uses a POSIX semaphore named `/the_lock` to lock `the_file.txt` prior to working on it
- Must lock prior to the `lseek()`, unlock after the `write()`

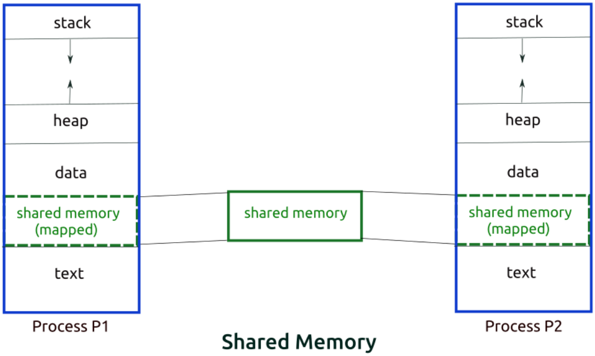
Alternatives

- `append_lockf.c`: Use the `lockf()` function to get an exclusive lock on the file, identical to semaphore solution but tied to individual files
- `append_os.c`: Open with `O_APPEND`, OS guarantees all `write()`'s are appended to the end

```c
// append_sem.c
int main(int argc, char *argv[]){
  if(argc < 2){
    printf("usage: %s <word>\n");
    return 1;
  }

  char *word = argv[1];
  char *sem_name = "/the_lock";
  char *filename = "the_file.txt";

  int fd = open(filename,
                O_CREAT | O_RDWR ,
                S_IRUSR | S_IWUSR);
  sem_t *sem = sem_open(sem_name,
                        O_CREAT,
                        S_IRUSR | S_IWUSR);
  sem_wait(sem);    // wait/lock semaphore
  lseek(fd, 0, SEEK_END);

  int n = strlen(word);
  word[n] = '\n';
  write(fd, word,n+1);
  close(fd);

  sem_post(sem);    // unlock semaphore
  sem_close(sem);   // done with sempahore

  return 0;
}
```

Shared Memory Segments

- The ultimate in flexibility is to get a segment of raw bytes that can be shared between processes
- POSIX shared memory outlives a process
- **Examine** `shmdemo_posix.c` to see how this looks
- Importantly, this program creates shared memory that outlives the program: must clean it up at some point



Source: SoftPrayog System V IPC

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
- General Use Cases
  - 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 but no permanence

Exercise: Email lookup with Shared Memory

- In lab, worked on a simple email lookup "server" or database
- Clients connected to server, server gave back emails based on name
- Shared memory makes server/client less relevant
- Propose how to use shared memory for email lookups AND alterations
- How might multiple processes coordinate use of shared memory?

```
// structure to store a lookup_t of
// name-to-email association
typedef struct {
  char name [STRSIZE];
  char email[STRSIZE];
} lookup_t;

lookup_t original_data[NRECS] = {
  {"Chris Kauffman"       ,"kauffman@umn.edu"},
  {"Christopher Jonathan" ,"jonat003@umn.edu"},
  {"Amy Larson"           ,"larson@cs.umn.edu"},
  {"Chris Dovolis"        ,"dovolis@cs.umn.edu"},
  {"Dan Knights"          ,"knights@cs.umn.edu"},
  {"George Karypis"       ,"karypis@cs.umn.edu"},
  ...
```

```
# Sample of potential use
> email_db lookup 'Chris Kauffman'
Looking up Chris Kauffman
Found: kauffman@umn.edu
> email_db lookup 'Rick Sanchez'
Looking up Rick Sanchez
Not found
> email_db change 'Chris Kauffman' 'kman@kauffmoney.co
Changing Chris Kauffman to kman@kauffmoney.com
Alteration complete
> email_db lookup 'Chris Kauffman'
Looking up Chris Kauffman
Found: kman@kauffmoney.com
```

Answer: Email lookup with Shared Memory

- Store entire array of name/email in a piece of shared memory with a known name/file
- Likely want database of saved so a **memory mapped file** is probably best
- Processes open shared memory/file, scan through looking
- Updates can be done by altering the shared memory
- **Danger** multiple processes writing may corrupt the data
- Use semaphores to control access for reading/writing, would need to establish a **protocol** for this
  - Lock entire database: easy but only one lookup at a time
  - Lock individual name/emails: more complex but potential for more throughput

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 `kirk_posix.c` and `spock_posix.c`

Email Lookup with Message Queues

- Email lookup server from lab used FIFOs for server and clients to talk
- Would not be too hard to rewrite this with message queues
- Message queues allow filtering of messages, easy to direct at a specific process
- Get automatic blocking and resuming when receiving messages so don't need explicit signals
- Will be the subject of next Lab

More Resources IPC

System V IPC

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

General Overview

- http://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf