CSCI 4061: Inter-Process Communication

Chris Kauffman

Last Updated:
Tue Apr 2 09:32:20 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

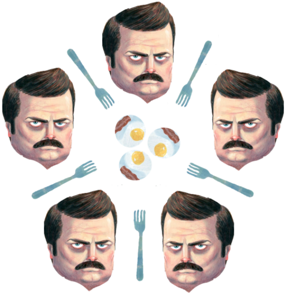
Model Problem: Dining "Philosophers"

- 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

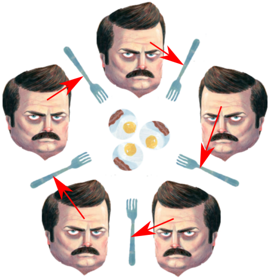
Dijkstra: One Swanson goes Right first

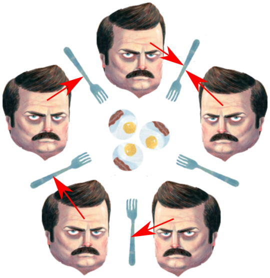
- Breaks the cycle so no deadlock possible
- Generalization establishes a **partial ordering** for each process to acquire resources, can prove lack of deadlocks

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`

- `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, listed on file system under `/dev/shm`
- POSIX semaphores are single values
- `sem_wait(sem)` will wait until semaphore is non-zero, the atomically decrement it
- `sem_post(sem)` will increment a semaphore and schedule a process waiting on it to run
- Semaphore operations are guaranteed to be **atomic**: only one process can increment/decrement at a time, function as efficient **locks**
- One semaphore per fork, must acquire both to eat an egg
- Deadlock avoided by having Swanson N change order of fork acquisition
  - Swanson N: Get right fork, then left fork
  - Other Swansons: Get left fork, the right fork

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 Swanson locks a fork 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

Most POSIX IPC items are visible on the file system in Linux

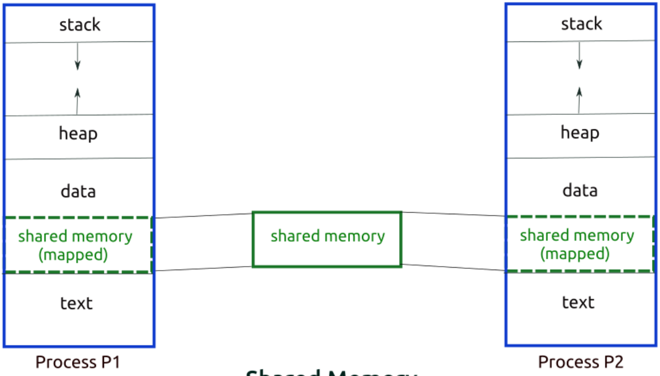
```
> gcc -o philosophers philosophers_posix.c -lpthread
> ./philosophers
Swanson 0: wants utensils 0 and 1
Swanson 2: wants utensils 2 and 3
Swanson 1: wants utensils 1 and 2
...
Swanson 3 (egg 10/10): leaving the diner
pausing prior to cleanup/exit (press enter to continue)
while you're waiting, have 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
```

Posix Conventions so Far

- IPC elements like semaphores can be **named**, use name to open them
- Creates a globally accessible object with permissions similar to files: any process with the name can access the IPC object
- In Linux, POSIX IPC elements usually have a handle on the file system in `/dev/shm`
- Can also create **anonymous** IPC objects which are shared only in a process tree
- Anonymous objects require **shared memory**, won't discuss them too much

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 works
- Importantly, this program creates shared memory that outlives the program: must clean it up at some point



Shared Memory

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
- Related concept: RAM Disk, a main memory file system, high performance but no permanence