Week 5 - Wednesday

COMP 3400

Last time

- What did we talk about last time?
- Overview of **`getopt()`**
- POSIX IPC
- Started POSIX message queues

Questions?

Assignment 3

Message Queues

Message queues

- Message queues are a form of message-passing IPC
- But don't we already have pipes and FIFOs?
- Differences from pipes:
  - Messages are sent as units: one whole message is retrieved at a time
  - Message queues use identifiers, not file descriptors, requiring special functions instead of **`read()`** and **`write()`**
  - Messages have priorities, not just first-in-first-out
  - Messages exist in the kernel, so killing off the sending process won't destroy them
- The big difference is structure:
  - Pipes and FIFOs send bytes, and the reader can read any number of available bytes at a time
  - Message queues send messages as units

POSIX message queues

- POSIX message queues have additional features that other implementations, like System V, might not have
- POSIX message queues:
  - Are only removed once they're closed by all processes using them
  - Include an asynchronous notification feature that allows processes to alerted when a message is available
  - Have priority levels for messages
  - Allow application developers to specify attributes (such as message size or capacity of the queue) via optional parameters passed when opening the queue

POSIX message queue functions

- `mqd_t mq_open (const char *name, int oflag, ...`
  `/* mode_t mode, struct mq_attr *attr */);`
  - Open (and possibly create) a POSIX message queue.
- `int mq_getattr(mqd_t mqdes, struct mq_attr *attr);`
  - Get the attributes associated with a given message queue
- `int mq_close (mqd_t mqdes);`
  - Close a message queue
- `int mq_unlink (const char *name);`
  - Remove a message queue's name (and the message queue itself, when all processes close it)
- `int mq_send (mqd_t mqdes, const char *msg_ptr,`
  `size_t msg_len, unsigned int msg_prio);`
  - Send a message with a given length and priority
- `ssize_t mq_receive (mqd_t mqdes, char *msg_ptr,`
  `size_t msg_len, unsigned int *msg_prio);`
  - Receive a message into a buffer and get its priority

Message queue sending example

- The following code creates a message queue and sends "**WOMBAT**"

```
mqd_t mqd = mq_open ("/comp3400_mq", O_CREAT | O_EXCL | O_WRONLY,  0600,
   NULL);  // mq_open() requires four arguments when creating

if (mqd == -1) // Check for error
   {
     perror ("mq_open failed");
     exit (1);
   }

mq_send (mqd, "WOMBAT", 7, 10); // Send WOMBAT (7 chars) with priority 10
mq_close (mqd);
```

- Priority increases as the number increases
- Priorities start at 0 and go up to at least 31, but some systems go as high as 32768
- Read documentation to find out how many priority levels there are

Warning!

- With pipes and FIFOs, it's common to create a fixed-size buffer and then read into it, usually only filling part of it
- With message queues, you have to read *exactly* the size of a message that's waiting for you
  - If not, the read will fail
- Two strategies:
  - Use a system where the sizes of messages are always the same
  - Use the **`mq_getattr()`** function to get the attributes of a message waiting in the message queue and create a buffer exactly the right size to read it

Message queue receiving example

- The following code reads the **"WOMBAT"** message sent by the other code
- It uses **mq_getattr()** to find out how big of a buffer it needs

```
mqd_t mqd = mq_open ("/comp3400_mq", O_RDONLY); // Only two arguments to open
assert (mqd != -1);

struct mq_attr attr;
assert (mq_getattr (mqd, &attr) != -1); // Get attributes

char *buffer = calloc (attr.mq_msgsize, 1); // Allocate buffer with size
assert (buffer != NULL);

unsigned int priority = 0;
if ((mq_receive (mqd, buffer, attr.mq_msgsize, &priority)) == -1) // Get message
  printf ("Failed to receive message\n");
else
  printf ("Received [priority %u]: '%s'\n", priority, buffer);

free (buffer);
buffer = NULL;
mq_close (mqd);
```

Asynchronous message queues

- What if you don't want your code to block when it's trying to read from or write to a message queue?
- Three alternatives:
  1. Bitwise OR **`O_NONBLOCK`** with **`oflag`** when opening the queue
     - Doing so will cause your code to return immediately with an error if there's nothing to read (or no space to write)
  2. Use **`mq_timedsend()`** and **`mq_timedreceive()`** which will eventually time out
  3. Use **`mq_notify()`** to send a signal to a process that can then go read a message after one is added to the message queue

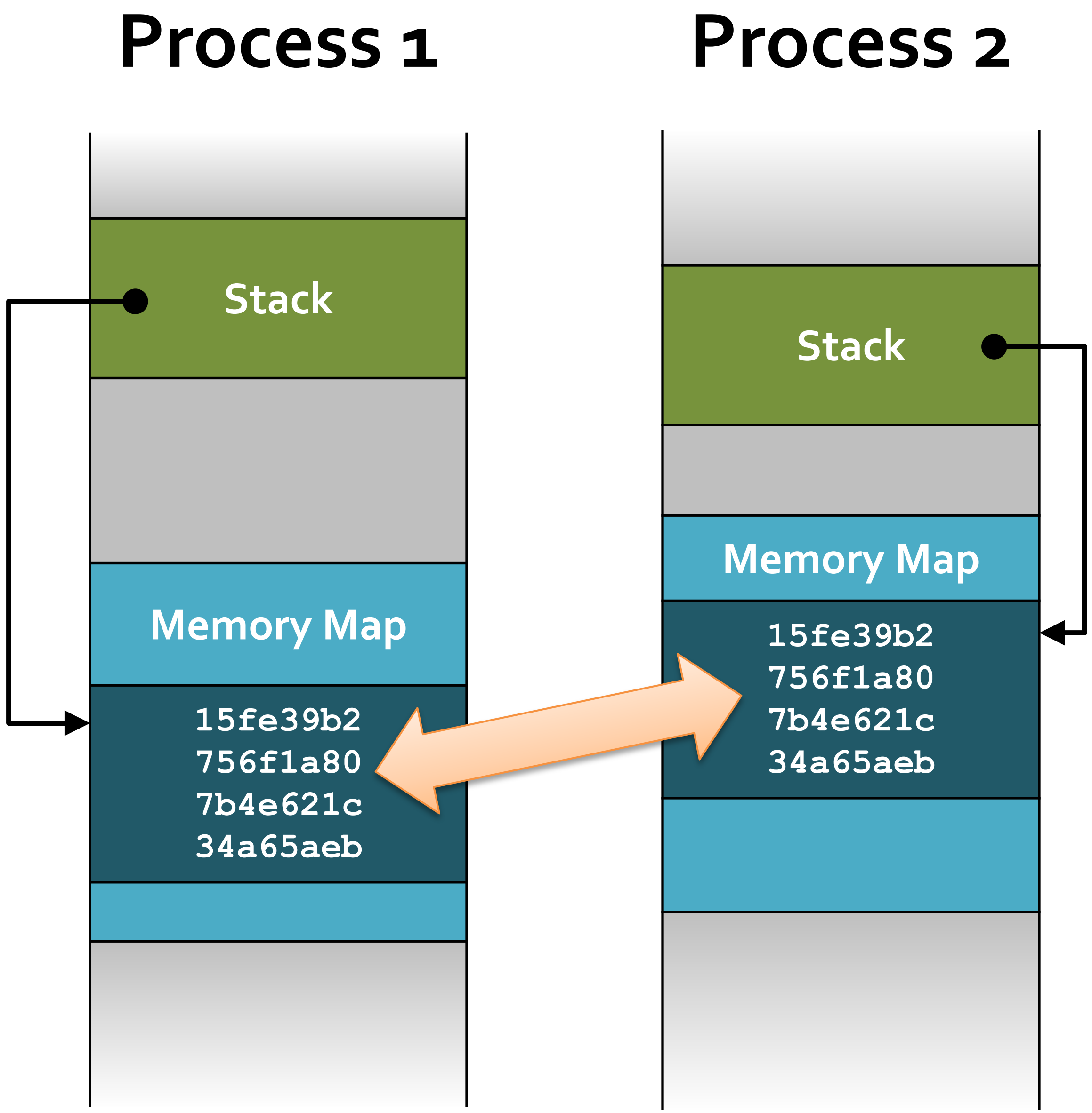
Shared Memory

Shared memory

- Shared memory is pretty much the same as using memory-mapped files
  - Except that there's no file associated with the share
  - So there's no persistent record of the memory
- To share memory, create a shared memory object (like a file, but isn't) with **`shm_open()`**
- The size of this object is often resized with **`ftruncate()`**
- Then, this shared memory object is mapped with **`mmap()`**, as was done with memory mapped files
- To delete the shared memory object, use **`shm_unlink()`**

Visualization

- The shared memory mapping means that a region of memory in one process exactly corresponds to memory in another region of memory in another process
- It's unlikely that the mapped memory will be in the same location in virtual memory for the two processes



Pointer problems

- When sharing memory, it could be tempting to share *any* memory
  - Even pointers
- For example, what if you wanted to have two processes both have access to a linked list?
- **It won't work.**
- Even shared memory has different addresses in each process's virtual memory
- If you *have* to use pointers, use offsets from the start of the shared memory, rather than pointer variables declared inside the memory

Functions

```
int shm_open (const char *name, int oflag, mode_t mode);
```

- **`name`** gives the name of the object
- **`oflag`**: Access needed, a bitwise OR of flags like **`O_RDONLY`**, **`O_WRONLY`**, **`O_RDWR`**, **`O_CREAT`**, and **`O_EXCL`**
- **`mode`**: Permissions, a bitwise OR of flags like **`S_IWUSR`** and **`S_IRGRP`**

```
int shm_unlink (const char *name);
```

- **`name`** is the object to delete

```
int ftruncate (int fd, off_t length);
```

- **`fd`** is a descriptor for the object or file to resize
- **`length`** the is the new size

Review of memory mapping functions

- The **mmap()** function returns memory mapped to a file descriptor or IPC object

```
void *mmap (void *addr, size_t length, int prot, int flags,
  int fd, off_t offset);
```

  - **addr** is a suggestion for where the memory goes but should usually be **NULL**
  - **length** is how many bytes to map
  - **prot** are flags shown on the right that can be combined
  - **flags** are **MAP_SHARED** or **MAP_PRIVATE** (and others), depending on whether the area is shared
  - **fd** is an open file descriptor for a file
  - **offset** is the starting point inside the file

Protection	Actions permitted
PROT_NONE	May not be accessed
PROT_READ	Region can be read
PROT_WRITE	Region can be modified
PROT_EXEC	Region can be executed

- The **munmap()** function unmaps an existing map

```
void munmap (void *addr, size_t length);
```

  - **addr** is the start of the mapped address
  - **length** is how much to unmap

Example of memory mapping

- First, let's imagine a struct declaration for structs that contain permission information

```
struct permission
{
  int user;
  int group;
  int other;
};
```

Example of memory mapping continued

- A parent process:
  - Creates a memory-mapped object
  - Stretches it to be exactly the right size
  - Maps some memory to this object

```
int shmfd = shm_open ("/comp3400_shm", O_CREAT | O_EXCL | O_RDWR,
  S_IRUSR | S_IWUSR);
assert (shmfd != -1);

// Resize to hold one struct
assert (ftruncate (shmfd, sizeof (struct permission)) != -1);

// Map the object into memory
struct permission *perm = mmap (NULL, sizeof (struct permission),
  PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);
assert (perm != MAP_FAILED);
```

Example of memory mapping continued

- Fork the process
- Then, the child process:
  - Sets values in the struct
  - Unmaps the memory
  - Closes the object

```
pid_t child_pid = fork();
if (child_pid == 0)
  {
    perm->user = 6;
    perm->group = 4;
    perm->other = 0;

    // Unmap and close the child's shared memory
    munmap (perm, sizeof (struct permission));
    close (shmfd);
    exit(0);
  }
```

Example of memory mapping finished

- Finally, the parent process:
  - Waits for the child to finish
  - Outputs the data stored by the child
  - Unmaps the memory and closes the object
  - Deletes the object

```
wait (NULL); // Wait for the child to finish

// Read from mapped memory
printf ("Permission bit-mask: 0%d%d%d\n",
        perm->user, perm->group, perm->other);

munmap (perm, sizeof (struct permission)); // Unmap
close (shmfd); // Close object
shm_unlink ("/comp3400_shm"); // Delete object
```

Why do we use both?

- It might strange to use **`shm_open()`** to create a POSIX object and **`mmap()`** to memory map this "file"
  - We could just memory map some existing file
  - We could use make a POSIX object and read and write it like it was a file
- Advantages to using both:
  - **`shm_open()`** creates POSIX objects instead of using other files
  - **`shm_unlink()`** only deletes POSIX objects when no other processes have connections to them, making it safer
  - Using **`mmap()`** makes it convenient to do memory accesses instead of file operations

Semaphores

Synchronization

- Both of the kinds of shared-memory IPC we've talked about often need synchronization
- **Synchronization** means controlling when reads and writes happen to avoid getting meaningless results
- In the previous example, a parent process waited for the child process to finish writing (and die) before reading
- In general, doing so is undesirable:
  - Many communicating processes do not have a parent/child relationship
  - Waiting for a process to die means that there can't be back-and-forth communication

Semaphores

- **Semaphores** are a simple kind of synchronization
- Internally, they have a counter
- If a process calls *wait* on a semaphore and the semaphore's value is 0 or lower, the process will get blocked
- When another process calls *post* and the counter goes up, a blocked process will resume (decrementing the counter back to 0 first)
- Many processes can be waiting on a single semaphore, but only one will resume per call to post
- Waiting on a semaphore is also called decrementing, downing, or P
- Posting on a semaphore is also called incrementing, upping, or V

Example

- Processes A and B have access to shared memory
- A is writing data, and B wants to read after the writing is done
- A and B also have access to a semaphore initialized to 0
- A increments the semaphore after it finishes writing
- B decrements the semaphore before reading
- Everything works out:
  - If B decrements the semaphore before A increments, B will block until A is done
  - If A increments the semaphore before B tries to decrement it, the semaphore will already be 1, so B will decrement it but not block

POSIX semaphores

- There are POSIX semaphores and System V semaphores
- They have many similarities, but we're only talking about POSIX semaphores
- POSIX semaphores come in named and unnamed varieties
- Like other POSIX IPC objects, named POSIX semaphores:
  - Must have a name that starts with slash, followed by non-slash characters
  - Should be unique from other named POSIX IPC objects

Semaphore functions

```
sem_t *sem_open (const char *name, int oflag,
/* mode_t mode, unsigned int value */ );
```

- Return (and possibly create) a named semaphore, using the usual **oflag** and **mode** flags
- **value** determines the initial value of the semaphore (often 0)

```
int sem_wait (sem_t *sem);
```

- Block if the semaphore's value is 0, decrement after continuing

```
int sem_post (sem_t *sem);
```

- Increment the semaphore's value, unblocking a process if the value is 0

```
int sem_close (sem_t *sem);
```

- Close a semaphore

```
int sem_unlink (const char *name);
```

- Delete a semaphore

Semaphore example

- The parent process creates a semaphore and forks a child
- The child waits on the semaphore and prints "second" after

```
sem_t *sem = sem_open ("/comp3400_sem", O_CREAT | O_EXCL,
                       S_IRUSR | S_IWUSR, 0); // Value starts at 0
assert (sem != SEM_FAILED);

pid_t child_pid = fork(); // Fork child, which inherits semaphore
assert (child_pid != -1);

if (child_pid == 0)
  {
    sem_wait (sem);  // Wait for semaphore
    printf ("second\n");
    sem_close (sem); // Close semaphore
    exit(0);
  }
```

Semaphore example continued

- Parent process:
  - Prints "first"
  - Posts on the semaphore
  - Waits for child to die before printing "third"

```
printf ("first\n");
sem_post (sem);
wait (NULL);   // Wait for child to die

printf ("third\n");
sem_close (sem);
sem_unlink ("/comp3400_sem"); // Delete semaphore
```

Trying or waiting

- Using a semaphore can be frustrating if you wanted to do other stuff and get blocked
- Instead of calling **`sem_wait()`**, there are two alternatives:

```
int sem_trywait (sem_t *sem);
```

  - Tries to decrement the semaphore but gives an error code if it would block

```
int sem_timedwait (sem_t *sem, struct timespec *time);
```

  - Waits on the semaphore but waits only for the amount of time specified in the **`struct timespec`**

Ticket Out the Door

Upcoming

Next time…

- Finish semaphores
- Review

Reminders

- **3 – 4 p.m. office hours canceled today**
- Work on Assignment 3
  - Due next Monday by midnight!
- Review book sections up to 3.8
- Exam 1 next Monday!
  - Review on Friday