Today

• Mid-term 2 review
  ○ OS Memory Management
  ○ Threads
  ○ Synchronization
OS Memory management

- Virtual Memory
  - Hiding the physical memory details and constraints from the processes and the programmer

- Paging
  - Breaking a process memory layout to “fit” it into available memory

- Swapping
  - Using secondary storage to multiplex memory among multiple processes
Paging

- Page: Contiguous chunk of memory addresses
- Process virtual memory is divided up into equal-size pages
- Frame: Physical memory is also divided up into same sized chunks
- Virtual memory mapping: Maps virtual pages into physical frames
Paging
Swapping

- Move inactive process to secondary storage (disk)
- Bring process into main memory when scheduled to run
- Swap space: Portion of disk devoted to swapping
Demand paging

- Combines paging and swapping
- Process uses only a part of its whole memory at a given time

-> Move inactive pages to disk
-> Bring in pages from disk when required
Page faults

- What happens when a process tries to access an address, but the corresponding page is on the disk?
- OS generates an exception called a page fault
- Performance penalty: Takes a long time when compared to a direct memory access
Threads

- A thread is a lightweight process
- Multiple threads can run concurrently within the same process
Processes vs. threads

**Process**

- Data
- Code
- Heap

**Threads**

- Data
- Code
- Heap

- PC
- Stack

- PC
- Stack

- PC
- Stack

- PC
- Stack
Thread resources

• Each thread has its own:
  ○ thread ID
  ○ Stack, registers, program counter, signal mask
  ○ errno

• Threads share
  ○ Process code, data, heap
  ○ Files, signals

• Threads within the same process can communicate using shared memory - *Must be done carefully!*
Pthreads

- POSIX threads
- Most widely supported threading library

- Program needs to be linked with “-lpthread”
- For example, in the makefile

  \[
  \text{CC} = \text{gcc} \\
  \text{LDFLAGS} = -lpthread
  \]
Thread management APIs

- Create a thread:
  pthread_create()

- Join a thread:
  pthread_join()

- Detach a thread:
  pthread_detach()

- Terminate a thread:
  pthread_cancel()
Synchronization

- A section of code which works with shared resources can have unpredictable results depending on the order in which the threads execute.

- This is called a critical section.
Mutual Exclusion

- A mutex lets you lock a code section so that only one thread at a time executes a critical section.
Mutex signatures

```c
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```
Condition variables

- Useful when you only want to enter a critical section under certain conditions.

- Avoid “busy waiting”, where a single thread wastes time repeatedly checking for a condition while waiting for it to become true.
Condition variables

- To create a conditional variable: `pthread_cond_t condvar = PTHREAD_COND_INITIALIZER`

- To wait for a conditional variable:
  ```c
  pthread_cond_wait(&condvar, &mutex);
  ```

- This will unlock the mutex, wait for a signal condvar, then try to lock the mutex when the signal arrives.
Condition variables

- To unblock a thread waiting on a condition variable:
  ```
  int pthread_cond_broadcast(pthread_cond_t *cond);
  int pthread_cond_signal(pthread_cond_t *cond);
  ```
- `pthread_cond_broadcast()` unblocks all the threads currently blocked
- `pthread_cond_signal()` unblocks at least one of the threads
Semaphores

- Semaphores are another synchronization method, besides Mutex and Cond. Variables.
- A semaphore is a special kind of integer: it can be incremented or decremented atomically.
Semaphores

- If the value of the semaphore is 0, it cannot be decremented.
- Threads that try to decrement it when its value is 0 go to sleep.

- Increment: `int sem_post(sem_t *sem);`
- Decrement: `int sem_wait(sem_t *sem);`
Questions?