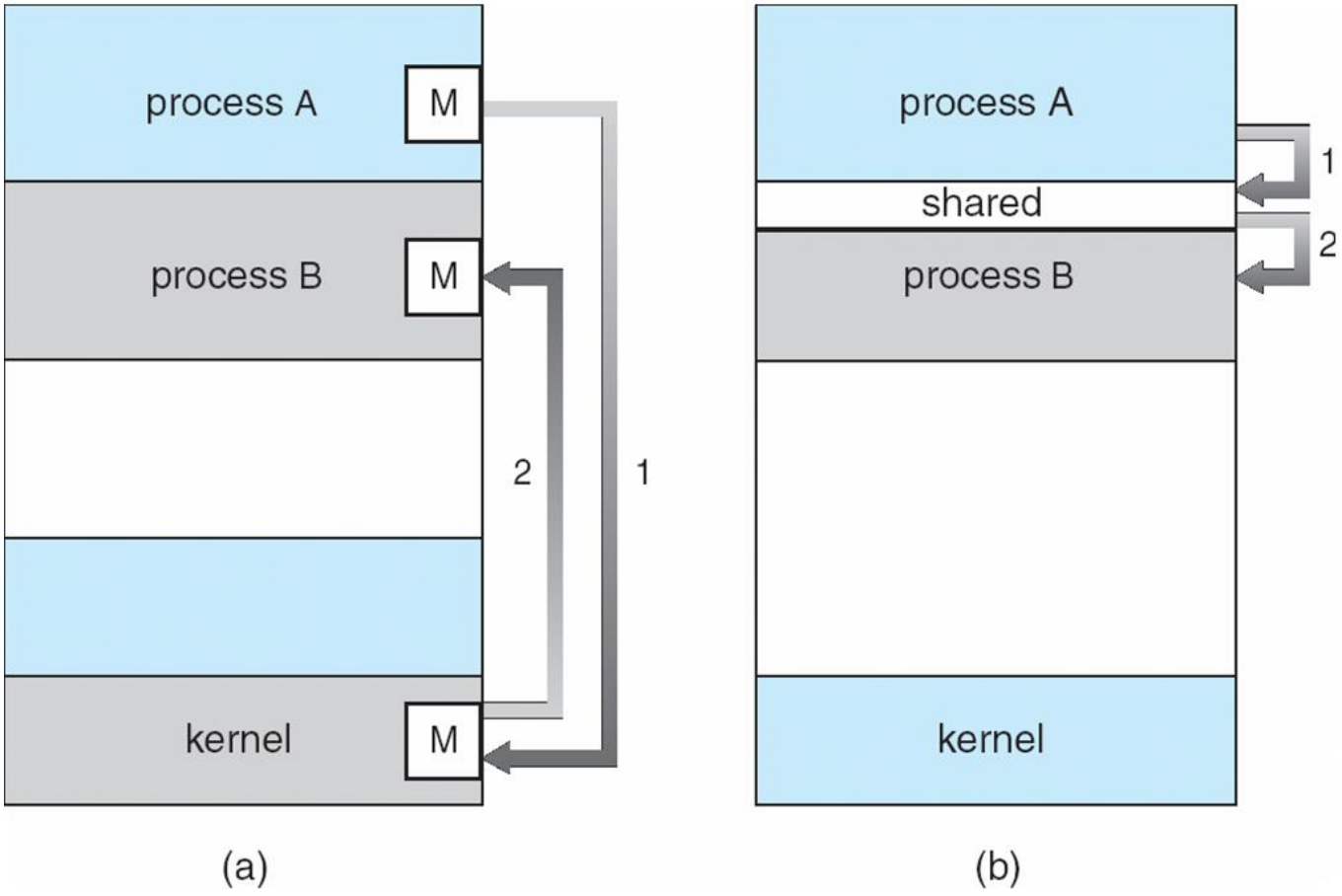
CS345
Operating Systems

Threads

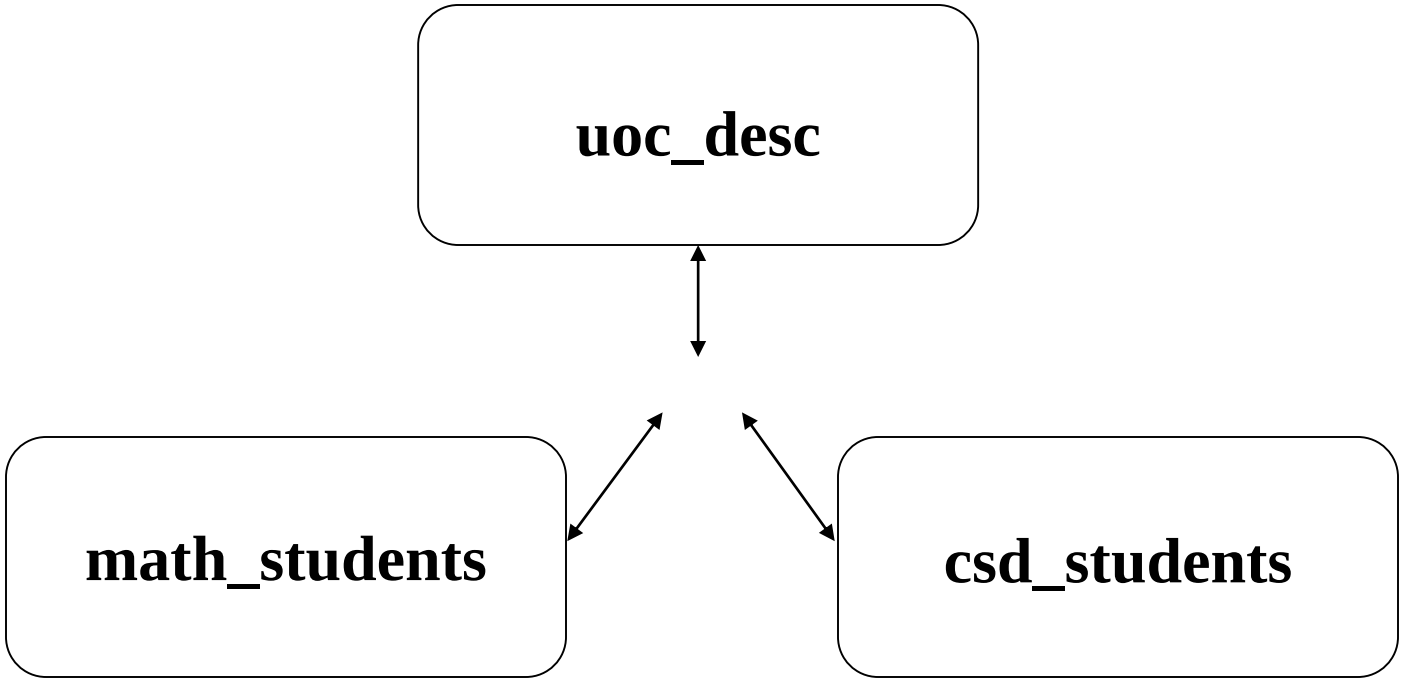
Inter-Process Communication (IPC)

- Exchange data among processes
- Methods
  - pipes
  - **signals**
  - **sockets**
  - **shared memory**

Communications Models



Assignment overview



- Three distinct programs
- These programs are communicating with sockets
- uoc_desc starts first, and create a “server-side” socket

Sockets

- How two processes with no common ancestor can communicate? Ans: sockets
- Socket is an abstraction for an endpoint communication that can be manipulated with a file descriptor.
- It is an abstract object from which messages are sent and received.
- Sockets are created within a communication domain just as files are created within a file system.
- A communication endpoint for two processes
  - Both must create their own socket.
  - Two sockets must be connected before they can transfer data.

Sockets

- Communication link between two programs running on the network (eg client-server)

- Communication between processes on the same Unix system (UNIX domain sockets).

Socket Types

- Stream sockets: connection-oriented, reliable, bidirectional, preserve sequence.
- Datagram sockets: connectionless, unidirectional.

Socket creation

Socket system call creates sockets on demand.

s = socket (af, type, protocol); // where s is an int,

- af - address family, AF_INET, AF_UNIX, AF_AAPLETALK etc.
- type - communication type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW etc.
- protocol - some domains have multiple protocol, usually use a 0.

Example:

```
unsigned int s;
struct sockaddr_un localsocket;
s = socket(AF_UNIX, SOCK_STREAM,0);
```

Returns -1 on error → Do error checking!

bind socket

- Socket binding: A socket is created without any association to local or destination address.

bind(s, localaddr, addrlength)

localaddr - struct of specific format for each address domain;
addrlength - length of this struct; obtained usually by *sizeof.*

sockaddr_un defines localaddr format for unix family (it is in *un.h* file).

```
struct sockaddr_un {
        short sun_family;
        char sun_path[108];
 };
```

bind socket and connect

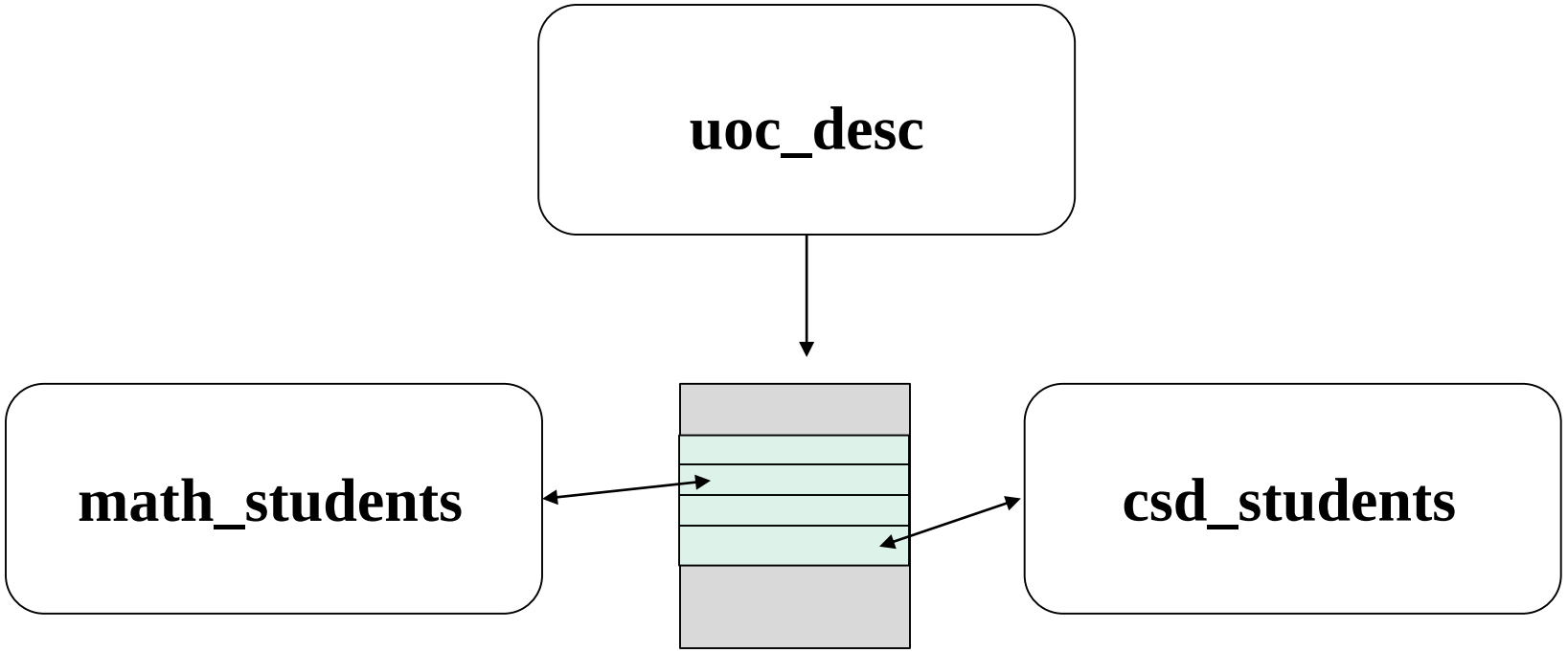
Example:

```
#define SocketName “test_socket”

unsigned int s, s2;
sockaddr_un localsocket, remotesocket;
s = socket(AF_UNIX, SOCK_STREAM,0);
localsocket.sun_family = AF_UNIX;
strcpy(localsocket.sun_path, SocketName);
//Bind to address (a special file on your system)
bind(s, &localsocket, sizeof(localsocket));

listen (s, 5)   // listen for incoming connections
s2 = accept(s, &remotesocket, sizeof(remotesocket));
```

Client sockets

- socket() -- to create unix socket
- struct sockaddr_un -- the remote address where the server is listening (address of client creating the socket).
- connect() -- to server's sockaddr_un address
- send() and recv() for communication
- no need for listen() and accept() !!

Assignment overview



- Synchronization between **math_students** and **csd_students**
- Use of shared memory (created by **uoc_desc**)

Shared Memory

- A shared memory segment is a portion of physical memory that is virtually shared between multiple processes.
- Processes connect to the shared memory segment and get a pointer to the memory
- A process can read and write to this pointer, and all changes are visible to every process connected to the shared memory segment.
- The form of IPC because data does not need to be copied between processes.

Shared Memory

+ Fast bidirectional communication among any number of processes
+ Saves resources (in comparison to other IPC forms).
+ No kernel involvement.
- Needs concurrency control/synchronization (data inconsistencies are possible)

Processes should be informed if it's ***safe*** to read and write data to the shared resource.

Shared Memory - creation

- ***SHMGET*** allocates a shared memory segment

  int shmget(key_t key, int size, int shmflg);

  returns the segment memory id created (shmid).

  key : unique id for memory identification

  size : size of memory allocation

  shmflg : flags for creation and permissions (IPC_CREATE)

  IPC_CREATE : if no segment is found with same key/size, it will create the memory segment.

  (If we try to create a memory segment with different size but same key, it will return an error.)

Shared Memory - attachment

- ***SHMAT*** attach the shared memory segment to the process

  void *shmat(int shmid, const void *shmaddr, int shmflg);

  returns the pointer of the shared memory segment, or -1 if failed

  shmid : shared memory identifier returned by SGMGET

  ***shmaddr** : defines where the process is situated in the segment.

  (NULL → beginning)

  shmflg : flags, eg SHM_RDONLY for read only access

- ***SHMDT*** detach the shared memory segment of the process

  int shmdt(const void *shmaddr);

  returns 0 if success, -1 if failed

Shared Memory - control

- ***SHMCTL*** is used to free the shared memory segment

•shmctl can also be used to execute different commands on the shared memory segment

•With the command IPC_RMID we can remove the shared memory segment.

int shmctl(int shmid, int cmd, struct shmid_ds *buff);

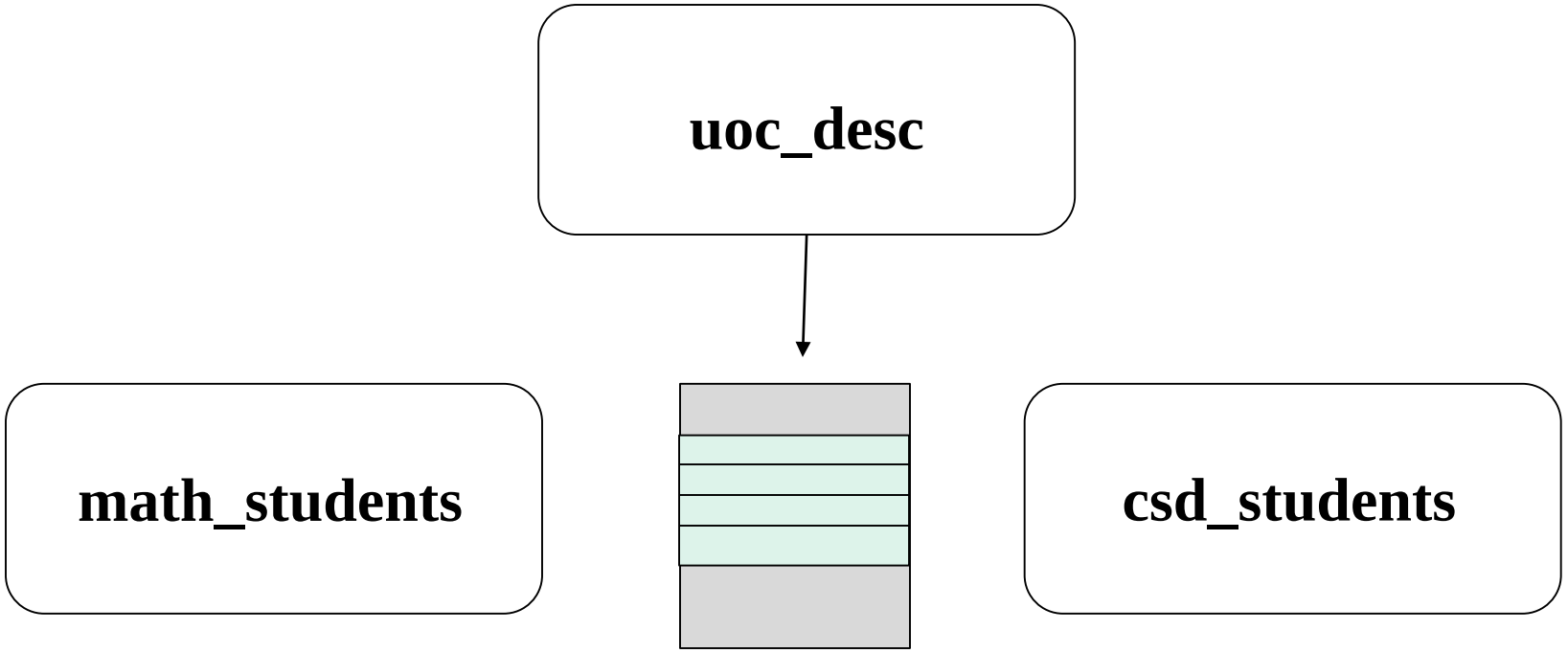
returns 0 if success, -1 if failed

shmid : shared memory identifier returned by **shmget**

cmd : command to operate : IPC_RMID for removal

***buff** : pointer to shared memory data structure

Assignment overview



- Use **CTRL+C** to stop the running programs.
- **CTRL+C → terminal sends an interrupt,** signal (SIGINT)
- Memory leakage? Programs run a cleanup function and die

Signals

- Software generated interrupts that are sent to a process when an event happens.
  - Synchronously generated by an application error, eg. SIGFPE, SIGSEGV
  - Most signals are asynchronous.
- Each signal has a default action, one of the following:
  - The signal is discarded after being received
  - The process is terminated after the signal is received
  - Stop the process after the signal is received
  - A core file is written, then the process is terminated

Signals

Defined in <signal.h> for common signals

```
SIGHUP    1   /*hangup*/
SIGINT    2   /*interrupt*/
SIGQUIT   3   /*quit*/
SIGILL    4   /*illegal instruction*/
SIGABRT   6   /*used by abort*/
SIGKILL   9   /*hard kill*/
SIGALRM  14   /*alarm clock*/
SIGCONT  18   /*continue a stopped process*/
SIGSTOP  19   /*process is paused, its state is preserved*/
```

Signals can be numbered from 0 to 31

Catching a Signal - Example

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void sig_handler(int signo)
{
        if (signo == SIGINT)
                printf("received SIGINT\n");
}

int main(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
                printf("\ncan't catch SIGINT\n");

        // A long wait so that we can easily issue a signal to this process
        while(1)
                sleep(1);
        return 0;
}
```

```
int  send_signal (int pid)
{
        kill(pid, SIGINT)
        printf("SIGINT sent\n");
}
```

Process vs. Thread

- process:
  - an address space with 1 or more threads executing within that address space, and the required system resources for those threads
  - a *program* that is running
- thread:
  - a sequence of control within a process
  - shares the resources in that process

Advantages of Threads

- The overhead for creating a thread is significantly less than that for creating a process
- Multitasking, i.e., one process serves multiple clients
- Switching between threads requires the OS to do much less work than switching between processes

Drawbacks of Threads

- Not as widely available as longer established features
- Writing multithreaded programs require more careful thought
- More difficult to debug than single threaded programs
- For single processor machines, creating several threads may not necessarily increase performance

main thread

- initial thread created when main() is invoked by the process loader
- once in the main(), the application has the ability to create daughter threads
- if the main thread returns, the process terminates even if there are running threads in that process, unless special precautions are taken
- to explicitly avoid terminating the entire process, use pthread_exit()

Create thread

- int pthread_create(
  pthread_t * thread,
  pthread_attr_t *attr,
  void * (*func)(void *),
  void *arg
  );

- 1st arg (***thread**) – pointer to the identifier of the created thread. (save the identifier, it is used for **pthread_join**)
- 2nd arg (***attr**) – thread attributes. If NULL, then the thread is created with default attributes
- 3rd arg (***func**) – pointer to the function the thread will execute
- 4th arg (***arg**) – the argument of the executed function
- Returns 0 for success, (>0) for error.

Waiting threads

int pthread_join(pthread_t thread, void **thread_return)

- main thread will wait for daughter *thread* to finish
- 1st arg (**thread**) – the thread to wait for
- 2nd arg (****thread_return**) – pointer to a pointer to the return value from the thread
- returns 0 for success
- threads should always be joined; otherwise, a thread might keep on running even when the main thread has already terminated

Sample Pthreads Program in C

- The program in C calls the **pthread.h** header file. Pthreads related statements are preceded by the pthread_ prefix (except for semaphores).
- How to compile:
  - gcc hello.c –pthread –o hello

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

int main(int argc, char **argv){
        pthread_t t1;
        int thread_id = 1;

       if ( (pthread_create(&t1, NULL, (void *)&worker, (void *)&thread_id)) != 0) {
             printf("Error creating thread\n");
             exit(1);
       }

       pthread_join(t1, NULL);

       return 0;
}

void worker(void *a)  {
        int *cnt = (int *)a;

         printf("This is thread %d\n", *cnt);
         pthread_exit(0);
}
```

Thread Synchronization Mechanisms

- Mutual exclusion (mutex):
  - guard against multiple threads modifying the same shared data simultaneously
  - provides locking/unlocking critical code sections where shared data is modified
  - each thread waits for the mutex to be unlocked (by the thread who locked it) before performing the code section

Create and initialize mutex

Mutex variables are declared with type pthread_mutex_t, and must be initialized before they can be used.

There are two ways to initialize a mutex variable:

- Statically, when it is declared. For example:
  pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

- Dynamically, with the **pthread_mutex_init()** routine. This method permits setting mutex object attributes, ***attr***.

•The mutex is initially unlocked.

•Routines

```
pthread_mutex_init (mutex,attr)
pthread_mutex_destroy (mutex)
```

Basic Mutex Functions

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- a mutex is like a key (to access the code section) that is handed to only one thread at a time
- the lock/unlock functions work in tandem
- Mutex is unlocked **only** by the thread that has locked it.
- Threads wait for access a locked section of code

```
#include <pthread.h>
...
pthread_mutex_t my_mutex;
...
int main()
{
        int tmp;
        ...
        // initialize the mutex
        tmp = pthread_mutex_init( &my_mutex, NULL );
        ...
        // create threads
        ...
        pthread_mutex_lock( &my_mutex );
            do_something_private();
        pthread_mutex_unlock( &my_mutex );
        ...
        …
        pthread_mutex_destroy(&my_mutex );
        return 0;
}
```

- Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked. If so, it waits until it is unlocked. Otherwise, it takes the mutex, locks the succeeding code, then frees the mutex and unlocks the code when it's done.

Semaphores

- Counting Semaphores:
  - permit a limited number of threads to execute a section of the code
  - similar to mutexes
  - should include the semaphore.h header file
  - semaphore functions do not have pthread_ prefixes; instead, they have sem_ prefixes

Basic Semaphore Functions

- creating a semaphore:

int sem_init(sem_t *sem, int pshared, unsigned int value);

  - initializes a semaphore object pointed to by sem
  - pshared is a sharing option; a value of *0* means the semaphore is local to the calling process
  - gives an initial value value to the semaphore

- terminating a semaphore:

int sem_destroy(sem_t *sem);

  - frees the resources allocated to the semaphore sem
  - usually called after pthread_join()
  - an error will occur if a semaphore is destroyed for which a thread is waiting

Basic Semaphore Functions

- semaphore control:

  int sem_post(sem_t *sem);

  int sem_wait(sem_t *sem);

  - sem_post *atomically* increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)
  - sem_wait *atomically* decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

```
#include <pthread.h>
#include <semaphore.h>
...
void *thread_function( void *arg );
...
sem_t semaphore;        // also a global variable just like mutexes
...
int main()
{
        int tmp;
        ...
        // initialize the semaphore
        tmp = sem_init( &semaphore, 0, 0 );
        ...
        // create threads
        pthread_create( &thread[i], NULL, thread_function, NULL );
        ...
        while ( still_has_something_to_do() )
        {
                sem_post( &semaphore );
                ...
        }
        ...
        pthread_join( thread[i], NULL );
        sem_destroy( &semaphore );
        return 0;
}
```

```
void *thread_function( void *arg )
{
        sem_wait( &semaphore );
        perform_task_when_sem_open();
        ...
        pthread_exit( NULL );
}
```

- the main thread increments the semaphore's count value in the while loop
- the threads wait until the semaphore's count value is non-zero before performing perform_task_when_sem_open() and further
- daughter thread activities stop only when pthread_join() is called