CS345
Operating Systems

Φροντιστήριο Άσκησης 2

Inter-process communication

- Exchange data among processes

- Methods
  - Signals
  - Pipes
  - Sockets
  - Shared Memory

Sockets

- Endpoint of communication link between two programs running on the network
- Inter-process communication flow across a computer network

UNIX Domain sockets

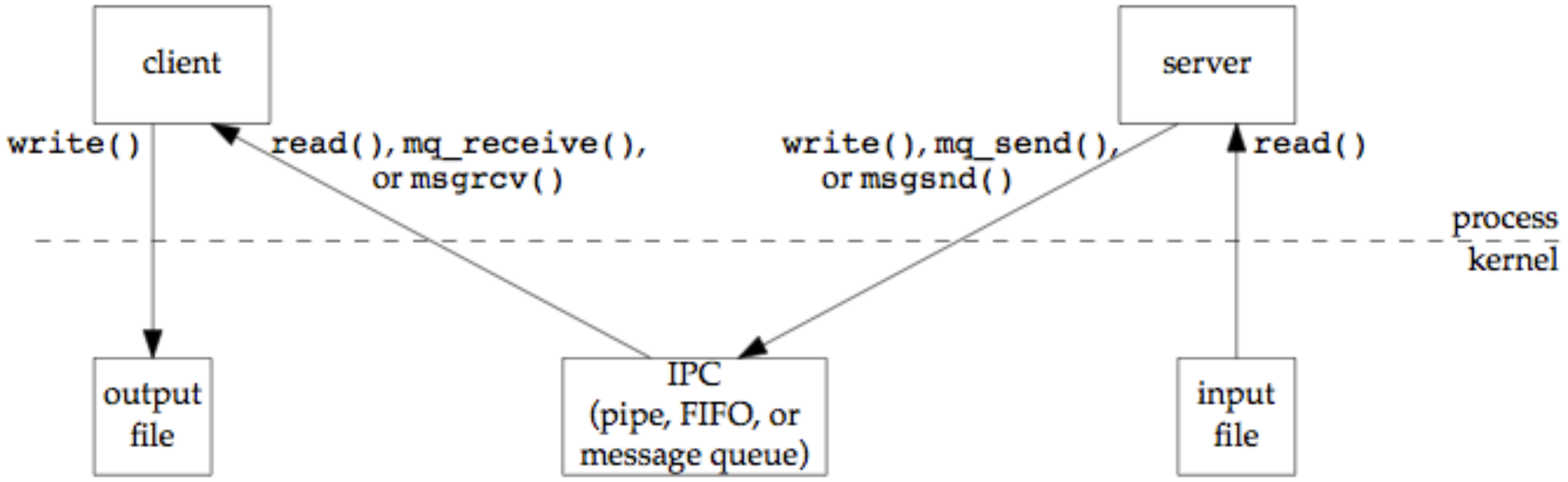
- Sockets for communication between processes on the same Unix system
- Imagine a “two-way” FIFO

Socket Types

- *Stream sockets*, also known as connection-oriented sockets, provides sequenced, reliable, two-way, connection-based byte streams.

- *Datagram sockets*, also known as connectionless sockets.

Simple file copy example


client
server
write()
read(), mq_receive(), or msgrcv()
write(), mq_send(), or msgsnd()
read()
process
kernel
output file
IPC (pipe, FIFO, or message queue)
input file

Create server socket

```
unsigned int s, s2;
struct sockaddr_un local, remote;
int len;
s = socket(AF_UNIX, SOCK_STREAM, 0);
```

returns -1 on error

- Do your error checking!!!

bind() socket

```
local.sun_family = AF_UNIX;
strcpy(local.sun_path, “guess_socket");
unlink(local.sun_path);  //remove if it already exists
len = strlen(local.sun_path) +sizeof(local.sun_family);
bind(s, (struct sockaddr *)&local, len);
```

Bind to an address
(basically a special file on your system)

listen()

```
listen(s, 5);
```

Listen for incoming connections from client programs

2nd argument: incoming connections that can be queued before being **accepted**

accept() connections

```
len = sizeof(struct sockaddr_un);
s2 = accept(s, &remote, &len);
```

Accepts connection
remote filled with the remote side's sockaddr_un
len will be set to its length

simple echo communication

```
done = 0;
do{
        n = recv(s2, str, 100, 0);
        if (n <= 0)
        {
                 if (n < 0) perror("recv");
                done = 1;
        }
        if (!done)
                if (send(s2, str, n, 0) < 0)
                {
                        perror("send");
                        done = 1;
                }
 }  while (!done);
```

Client socket

- socket() to create unix socket
- struct sockaddr_un with the remote address (where the server is listening)
- connect() to sockaddr_un
- send() and recv() for communication
- No need for listen(), accept() !!!

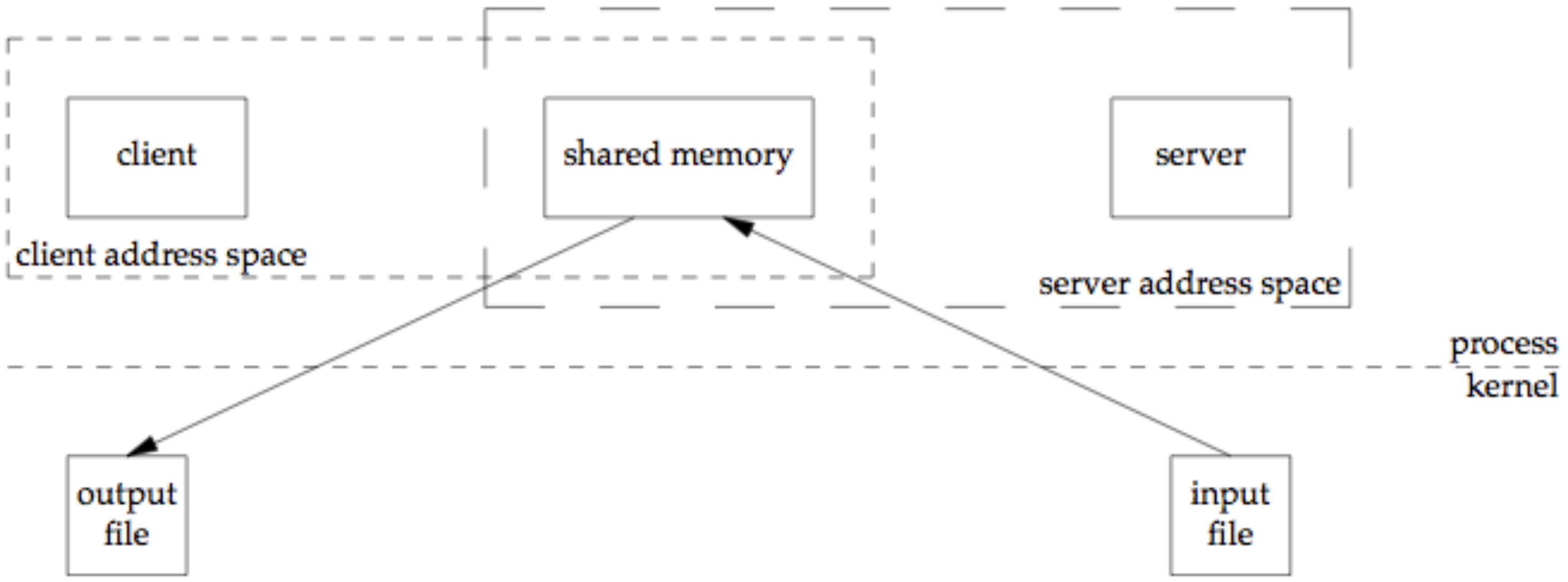
Shared Memory

- Memory mapped into the address space of the processes that are sharing the memory region
- One program creates the segment, the other can access it (if permitted)

- Efficient means of passing data between processes
  - Avoid redundant copies
  - No kernel involvement

➢ Requires some form of synchronization

Simple file copy example



Obtain Access to memory

```
key_t key; /* unique ID */
int shmflg; /* access permissions and create*/
int shmid; /* return value */
int size; /* size */
struct shmid_ds shmbuf; /* needed for detachment

/* … assign values*/

shmid = shmget(key, 1024, 0644 | IPC_CREAT);
shm1=shmat(shmid, NULL, 0);,...//attach segment
(if (shm1==(char *)-1)  //Always do error-checking!!
```

Detach and delete segment

```
shmctl(shmid1, IPC_RMID, &shmbuf);
shmdt(shm1); //address returned from shmat
```

shmctl(): alter permissions and other characteristics of shared memory..only by creator process

IPC_RMID: Remove the shared memory segment

shmbuf needed for other actions

shmdt(): detach from segment, all processes

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

- Writing multithreaded programs require more careful thought
- More difficult to debug than single threaded programs
- For single processor machines, creating several threads in a program may not necessarily produce an increase in 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 *(*thread_function)(void *), void *arg );
```

- 1st arg – pointer to the identifier of the created thread
- 2nd arg – thread attributes. If null, then the thread is created with default attributes
- 3rd arg – pointer to the function the thread will execute
- 4th arg – the argument of the executed function
- returns 0 for success

Waiting threads

```
int pthread_join( pthread_t thread, void **thread_return )
```

- main thread will wait for daughter thread *thread* to finish
- 1st arg – the thread to wait for
- 2nd arg – 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

Threads Programming Model

- pipeline model – threads are run one after the other
- master-slave model – master (main) thread doesn't do any work, it just waits for the slave threads to finish working
- equal-worker model – all threads work

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

Basic Mutex Functions

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *mutexattr);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- a new data type named pthread_mutex_t is designated for mutexes
- a mutex is like a key (to access the code section) that is handed to only one thread at a time
- the attribute of a mutex can be controlled by using the pthread_mutex_init() function
- the lock/unlock functions work in tandem

```
#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

- Restricts number of simultaneous users of a shared resource up to a maximum number
- Threads can request access (decrement)
- Signal they have finished using the resource (increment)
- 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. We will use 1.
  - 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 (unlock) ***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 (lock) ***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().

Daughter thread activities stop only when pthread_join() is called.

Server structure

- Setup Unix socket ()
- Receive M from client
- Send N to client
- Create secret
- Create shared memory segments
- Send mem keys to client
- Create M threads, which run validation code
  - Each thread validates "guessing tries" for specific segment of secret
  - Be careful when synchronizing access to the critical region
  - Remember to time the execution
- Clean up for exit (handle signal, release mem, close socket)

Client structure

- Connect to socket
- Send M to server
- Read N from server
- Receive memory keys
- Attach to memory segments
- Create M threads which run guessing code
- Clean up for exit (handle signal, detach mem, close socket)

Guessing code

```
for (i=0; i<N; i++) {
      for(j='A'; j<='z'; j++) {
            choice[i]=j;
            //sync with server to check whether this choice
            //is correct and wait for server's response in valid[i]
            if (valid[i]==1) break; //found it
      }
}
```

You have to synchronize actions performed on the critical region
DO NOT use 1 semaphore for the whole critical region!!!
Each (N/M) chunk of the region must be separately handled