# CS 345 – Operating Systems

Tutorial 2: Treasure Room Simulation

Threads, Shared Memory, Synchronization

Assignment 2

- We have a treasure room, Team A and Team B.

- Treasure room has N coins inside. Each team tries to enter the treasure room and take coins.

- The game ends when the treasure room is empty and the team that has taken more coins wins.

Server - Client

Treasure room:

Team A:

Team B:
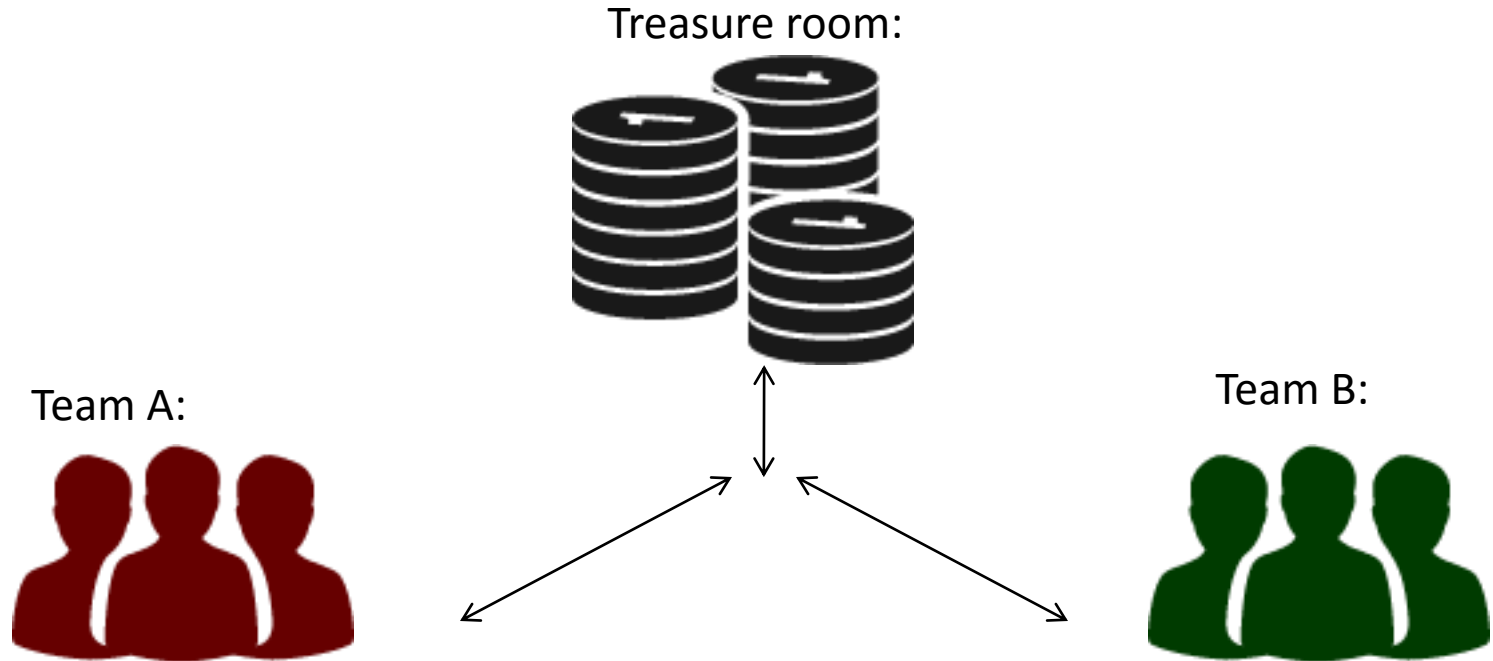
– Three distinct programs
– These programs are communicating with sockets
– treasure_room starts first, and create a "server-side" socket

Sockets

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

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

- A communication endpoint for two processes
  – Both must create their own socket.
  – Two sockets must be connected before they can transfer data.

Socket creation

Socket system call creates sockets on demand.

s = socket (af, type, protocol);

- Af - address family, AF_INET, AF_UNIX etc.
- Type - communication type: SOCK_STREAM, SOCK_DGRAM, 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 and "connect"

- 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.

- Listen: listens for connections on a socket

- Accept: accepts a connection on a socket

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);

s2 = accept(s, &remotesocket, sizeof(remotesocket));

sockaddr_un defines localaddr format for unix family

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

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()*

Threads

- A thread is a light - weight process.
- A thread exists within a process, and uses the process resources.
- It is asynchronous.
- The program in C calls the pthread.h header file.

- How to compile:

  gcc hello.c –pthread –o hello

Create thread

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

Returns 0 for success, (>0) for error.

- 1st arg (*thread) – pointer to the identifier of the created thread.
- 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

Shared memory

Shared memory is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs

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.

Shared memory - concerns

- 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.
  shmflg: flags,
      eg: shm = shmat(shmid, NULL, 0))

- **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

Thread synchronization mechanisms

- Mutual exclusion (mutex)
- Semaphores

Mutexes

- guard against multiple threads modifying the same shared data simultaneously

- provide 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

Mutexes - create and initialize

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:

1. Statically, when it is declared. For example:

pthread_mutex_t mut= PTHREAD_MUTEX_INITIALIZER;

2. 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)

Mutexes – basic 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 together

- a mutex is unlocked **only by the thread that has locked it.**

```
#include <pthread.h>
…
pthread_mutex_t my_mutex;
…
intmain()
{
            inttmp;
            …
            // 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 (if we use binary semaphores it's the same)

- should include the semaphore.h header file

- semaphore functions do not have pthread_prefixes; instead, they have sem_prefixes

Semaphores – basic 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

  – an error will occur if a semaphore is destroyed for which a thread is waiting

Semaphores – basic functions

- Semaphore control:

int sem_post(sem_t*sem);

– 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)

int sem_wait(sem_t*sem);

– atomically decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

```c
#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

Assignment 2

Treasure_room:
- Generate_key ()
- Room key is the shared memory segment key

While (coins > 0)
        - Listen teams attempting to enter room
        - If anyone makes it, pass the key
        - Check number of coins
Say who the winner is and end game.
(Remember to close sockets and detach shared memory segments)

Team_a , team_b:
- Ask for key (first time only)
- If room is empty attempt to enter
        lock room
        coins --
        unlock room
   else -> wait