UNIX System Programming

Shared Memory

IPC Data flow

- PIPEs:
  - Unidirectional
  - Must exist between related processes
- FIFOs:
  - Use a local disk file for buffering
  - Could potentially have performance impacts.
- Shared Memory:
- Sockets:

Shared Memory

Shared memory allows for a group of processes to share a declared memory region.

Advantages:

- Memory based IPC Data flow
- All of the processes can, if we want, read and write to the memory region.
- Fast as it is resident in main memory (unless swapped out).
- Access is available to unrelated processes
- Can be tightly controlled in terms of who can read and who can write.

- Drawbacks:
  - Need tight control on who accesses what and when. This is done through the use of Semaphores and/or lock just as we would control a critical section of code.

Shared Memory: Setup….

Before we go through the steps of creating a shared memory region.. what is it we want to share?

Let's say we want a 4 player game to be able to "share" and update some status data:

```
struct player_descr {   // define what a player looks like
       int  pid;
       int  X;
       int  Y;
       char name[1024];
     };

 struct player_descr  single_player_data; // local memory
            //to store the structure for a single player
```

Shared memory: *ftok()*

Step 1: Generate a system wide unique "key" for the memory region. using ***ftok().***
The key is a unique identifier generated by the system.

```
#define KEY_FILE     "/home/jacques/3380/shared_memory/shared_mem_key_target.txt"

key_t shared_mem_key;    // create a key to be used by all routines to
                         //   refer to this specific memory region

shared_mem_key = ftok(KEY_FILE,1313);   // file-to-key: uses the inode#
                                        //  to generate the key

  if( shared_mem_key == -1 ) // if ftok failed (-1).
   {
     printf("Error obtaining the shared memory key\n\n");
     exit(1);
   }

  printf("\nOur shared memory key=%d\n",shared_mem_key);
```

- **NOTE: 1313** is just a user provided "Project Number" to allow multiple projects to use the same key file.

Shared Memory: *shmget()*

Step 2: Get the system to allocate a memory region using ***shmget().***

shmget() requires:

- We provide the IPC key returned from *ftok()*.
- How much memory we want to share.
- What permissions we want to allocate to the memory region.

Shared Memory: *shmget()*

```
#define SHMEM_PERM  ( S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP )

#define MAX_PLAYERS 4   // how many players we will allow in the game
int segment_id;         // the id of the segment returned by shmget

mem_size   = MAX_PLAYERS * sizeof(single_player_data);
segment_id = shmget(shared_mem_key, mem_size, SHMEM_PERM | IPC_CREAT )
  ;
if( segment_id == -1 )
  {
    printf("Error using shmget\n");
    exit(2);
  }
printf("shmget. Size=%d\tSegment ID=%d\n", mem_size, segment_id);
```

Shared Memory: *shmat()*

Step 3: Once the region has been created, we can attach to it using *shmat()*.

```
void * shmat(int shmid, const void *shmaddr, int shmflags)
```

Returns a **pointer** to the memory region

Parameters:

- The id returned from *shmget()*.
- The address we'd like the system to allocate for the shared memory. The system may or may not use this and may or may not round up to a page boundary. By passing a NULL, we allow the system to decide.

Shared Memory: *shmat()*

```
struct player_descr * segment_addr;// pointer/address of the shared
                                   // memory block from shmat()

 segment_addr = shmat( segment_id, (void *) 0,  0 );
 if( segment_addr == (void *)(-1) )
  {
    printf("Error in shmat\n");
    exit(3);
  }
```

Shared Memory: Accessing data

We now have the start address of the memory region.

- The memory region's pointer knows the "structure" used to define the shared region.
- We can therefore deal with the region as an array made up of element of the types we defined.
- "pointer math" gives us access to the individual elements.

NOTE: We are responsible for bounds checking!!!

Shared Memory: Loading Data

```
//
//  now let's update the second palyer's data
//
    single_player_data.pid = 1313;    // change the data
    single_player_data.X    = 15;
    single_player_data.Y    = 15;
    strncpy(single_player_data.name,"Fred\0",5);

//
//  Update the data for the second player. Notice since we know the size of the structure
//  the number we add to the pointer variable can be used as an array index. It knows the
//   size of the structure and will point to the Nth element.
//

    memcpy( segment_addr+2,   &single_player_data,   sizeof(single_player_data)    );
```

Shared Memory: Retrieving Data

```
 printf("\n\nWe now loop to display the content of the share memory. we loop until the user enters an x\n");

 user_input = 'g';  // set some value so the loop will prompt at least once.

while( user_input != 'x' )
  {
    //   let's copy all of the shared memory back into our array and see what we have.
    memcpy( &players, segment_addr, MAX_PLAYERS*sizeof(single_player_data) );

    for( ptr=0; ptr<MAX_PLAYERS; ptr++)
     {
       printf("\tpid=%6d\tX=%6d\tY=%6d\tname=%-20s\n",  players[ptr].pid,    players[ptr].X,
                              players[ptr].Y,   players[ptr].name);
     }

    printf("Press <return> to loop or 'x' to exit\n");
     user_input = getc(stdin);
  }
```

Shared Memory: Data

Once you have the address of the memory region, you can calculate your own “offsets” into the data.

You can then update or inspect the data elements individually if you want to.

There’s no real need to copy/dump/update the whole array each time we want to access one element of the array.

Shared Memory: Cleaning up... *shmdt()*

As with *malloc()*, when you reserve memory for a specific purpose, you must release it back to the system using *shmdt()* when done.

```
status = shmdt(segment_addr);
if( status == -1 )
 {
   printf("Error detaching from the memory segment\n");
   exit(4);
 }
```

Shared Memory:

Allows separate, potentially unrelated, processes to exchange data.

Since it is based in memory, the exchange is “fast”.

Since it is shared, we need to employ mechanisms to ensure data integrity. We need to prevent simultaneous updates using locks/semaphores.