OSD 19CS2106S
Session – 34
Shared Memory IPC
Lecture Notes

Shared Memory

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server. The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access. (But as we saw at the end of the previous section, record locking can also be used.)
The Single UNIX Specification includes an alternate set of interfaces to access shared memory in the shared memory objects option of its real-time extensions. We do not cover the real-time extensions in this text.
The kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* see Earlier Section */
  size_t           shm_segsz;   /* size of segment in bytes */
  pid_t            shm_lpid;    /* pid of last shmop() */
  pid_t            shm_cpid;    /* pid of creator */
  shmatt_t         shm_nattch;  /* number of current attaches */
  time_t           shm_atime;   /* last-attach time */
  time_t           shm_dtime;   /* last-detach time */
  time_t           shm_ctime;   /* last-change time */
  .
  .
  .
};
```

- The `ipc_perm` structure is initialized as described earlier. The `mode` member of this structure is set to the corresponding permission bits of flag. These permissions are specified with the values given earlier.
- `shm_lpid`, `shm_nattach`, `shm_atime`, and `shm_dtime` are all set to 0.
- `shm_ctime` is set to the current time.
- `shm_segsz` is set to the size requested.

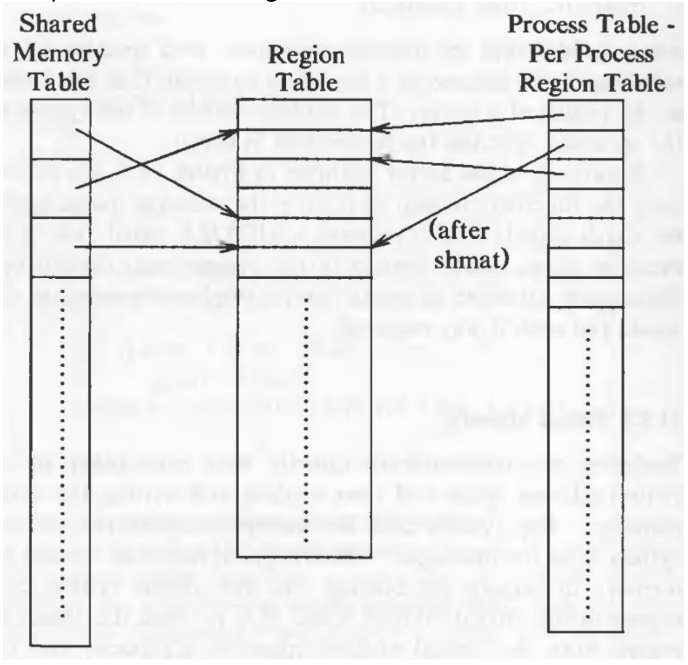
Shared Memory

Sharing the part of virtual memory and reading to and writing from it, is another way for the processes to communicate. The system calls are:

- *shmget* creates a new region of shared memory or returns an existing one.
- *shmat* logically attaches a region to the virtual address space of a process.
- *shmdt* logically detaches a region.
- *shmctl* manipulates the parameters associated with the region.

```
shmid = shmget(key, size, flag);
```

where `size` is the number of bytes in the region. If the region is to be created, *allocreg* is used. It sets a flag in the shared memory table to indicate that the memory is not allocated to the region. The memory is allocated only when the region gets attached. A flag is set in the region table which indicates that the region should not be freed when the last process referencing it, *exits*. The data structure are shown below:



Syntax for *shmat*:

```
virtaddr = shmat(id, addr, flags);
```

where addr is the virtual address where the user wants to attach the shared memory. And the `flags` specify whether the region is read-only and whether the kernel should round off the user-specified address. The return value *virtaddr* is the virtual address where the kernel attached the region, not necessarily the value requested by the process.

The algorithm is given below:

```
/*  Algorithm: shmat
 *  Input: shared memory descriptor
 *         virtual addresses to attach memory
 *         flags
 *  Output: virtual address where memory was attached
 */
{
        check validity of descriptor, permissions;
        if (user specified virtual address)
        {
                round off virtual address, as specified by flags;
                check legality of virtual address, size of region;
        }
        else                    // user wants kernel to find good address
                kernel picks virtual address: error if none available;
        attach region to process address space (algorithm: attachreg);
        if (region being attached for first time)
                allocate page tables, memory for region (algorithm: growreg);
        return (virtual address where attached);
}
```

If the address where the region is to be attached is given as 0, the kernel chooses a convenient virtual address. If the calling process is the first process to attach that region, it means that page tables and memory are not allocated for that region, therefore, the kernel allocated both using *growreg*.

The syntax for *shmdt*:

```
shmdt(addr);
```

where addr is the virtual address returned by a prior *shmat* call. The kernel searches for the process region attached at the indicated virtual address and detaches it using *detachreg*. Because the region tables have no back pointers to the shared memory table, the kernel searches the shared memory table for the entry that points to the region and adjusts the field for the time the region was last detached.

Syntax of *shmctl*

```
shmctl(id, cmd, shmstatbuf);
```

which is similar to *msgctl*

Shared Memory IPC

A segment of memory is shared between processes. You just connect to the shared memory segment, and get a pointer to the memory. You can read and write to this pointer and all changes you make will be visible to everyone else connected to the segment.

Creating the segment and connecting

Similarly to other forms of System V IPC, a shared memory segment is created and connected to via the **`shmget()`** call:

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

Upon successful completion, **`shmget()`** returns an identifier for the shared memory segment. The `key` argument should be created the same was as shown in the Message Queues document, using **`ftok()`**. The next argument, `size`, is the size in bytes of the shared memory segment. Finally, the `shmflg` should be set to the permissions of the segment bitwise-ORd with `IPC_CREAT` if you want to create the segment, but can be `0` otherwise. (It doesn't hurt to specify `IPC_CREAT` every time—it will simply connect you if the segment already exists.)

```
key_t key;
int shmid;
char *data;
key = ftok("/home/vishnu/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

Here's an example call that creates a 1K segment with `644` permissions (`rw-r--r--`):

But how do you get a pointer to that data from the `shmid` handle? The answer is in the call **`shmat()`**, in the following section.

Attach me—getting a pointer to the segment

Before you can use a shared memory segment, you have to attach yourself to it using the **`shmat()`** call:

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

shmid is the shared memory ID you got from the call to **shmget()**. Next is *shmaddr*, which you can use to tell **shmat()** which specific address to use but you should just set it to 0 and let the OS choose the address for you. Finally, the *shmflg* can be set to SHM_RDONLY if you only want to read from it, 0 otherwise.
Here's a more complete example of how to get a pointer to a shared memory segment:

```
data = shmat(shmid, (void *)0, 0);
```

Now you have the pointer to the shared memory segment! Notice that **shmat()** returns a void pointer, and we're treating it, in this case, as a char pointer. You can treat is as anything you like, depending on what kind of data you have in there. Pointers to arrays of structures are just as acceptable as anything else.
Also, it's interesting to note that **shmat()** returns -1 on failure. But how do you get -1 in a void pointer? Just do a cast during the comparison to check for errors:

```
data = shmat(shmid, (void *)0, 0);
if (data == (char *)(-1))
    perror("shmat");
```

Reading and Writing

Lets say you have the *data* pointer from the above example. It is a char pointer, so we'll be reading and writing chars from it. Furthermore, for the sake of simplicity, lets say the 1K shared memory segment contains a null-terminated string.
It couldn't be easier. Since it's just a string in there, we can print it like this:

```
printf("shared contents: %s\n", data);
```

And we could store something in it as easily as this:

```
printf("Enter a string: ");
gets(data);
```

Detaching from and deleting segments

When you're done with the shared memory segment, your program should detach itself from it using the **shmdt()** call:

```
int shmdt(void *shmaddr);
```

The only argument, *shmaddr*, is the address you got from **shmat()**. The function returns -1 on error, 0 on success.
When you detach from the segment, it isn't destroyed. Nor is it removed when *everyone* detaches from it. You have to specifically destroy it using a call to **shmctl()**, similar to the control calls for the other System V IPC functions:

```
shmctl(shmid, IPC_RMID, NULL);
```

The above call deletes the shared memory segment, assuming no one else is attached to it. The **shmctl()** function does a lot more than this, though, and it worth looking into. As always, you can destroy the shared memory segment from the command line using the **ipcrm** Unix command. Also, be sure that you don't leave any usused shared memory segments sitting around wasting system resources. All the System V IPC objects you own can be viewed using the **ipcs** command.

```
/*** shmdemo.c --demonstrate shared memory IPC - execute with
and without command line arguments */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHM_SIZE 1024 /* make it a 1K shared memory segment */
int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;
    if (argc > 2) {
         fprintf(stderr, "usage: shmdemo
         [data_to_write]\n"); exit(1);
    }
    /* make the key: */
    if ((key = ftok("shmdemo.c", 'R')) == -1)
         { perror("ftok");
         exit(1);
    }
    /* connect to (and possibly create) the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1)
         { perror("shmget");
         exit(1);
    }
    /* attach to the segment to get a pointer to it:
    */ data = shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1))
         { perror("shmat");
         exit(1);
```

```
     }
     /* read or modify the segment, based on the command line: */
     if (argc == 2) {
          printf("writing to segment: \"%s\"\n", argv[1]);
          strncpy(data, argv[1], SHM_SIZE);
     } else
          printf("segment contains: \"%s\"\n", data);
     /* detach from the segment: */
     if (shmdt(data) == -1) {
          perror("shmdt");
          exit(1);
     }
}
/*
[vishnu@mannava ~]$ cc shmdemo.c -o shmdemo
[vishnu@mannava ~]$ ./shmdemo hello
writing to segment: "hello"
[vishnu@mannava ~]$ ./shmdemo
segment contains: "hello"
*/
*/[vishnu@mannava ~]$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 98304      vishnu     600        393216     2          dest
0x00000000 131073     vishnu     600        393216     2          dest

0x00000000 163842     vishnu      600        393216     2          dest
*/
```

Program to Demonstrate System V Shared memory

Writing a message into shared memory

In this part of this recipe, we will learn how a message is written into shared memory.

How to do it…

1. Invoke the ftok function to generate an IPC key by supplying a filename and an ID.
2. Invoke the shmget function to allocate a shared memory segment that is associated with the key that was generated in step 1.
3. The size that's specified for the desired memory segment is 1024. Create a new memory segment with read and write permissions.
4. Attach the shared memory segment to the first available address in the system.
5. Enter a string that is then assigned to the shared memory segment.
6. The attached memory segment will be detached from the address space.

The writememory.c program for writing data into the shared memory is as follows:

/* writememory.c – Program to write data into the attached shared memory segment */

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
      char *str;
      int shmid;

      key_t key = ftok("sharedmem",'a');
      if ((shmid = shmget(key, 1024,0666|IPC_CREAT)) < 0) {
                  perror("shmget");
                  exit(1);
      }
            if ((str = shmat(shmid, NULL, 0)) == (char *) -1) {
                  perror("shmat");
                  exit(1);
      }
      printf("Enter the string to be written in memory : ");
      gets(str);
            printf("String written in memory: %s\n",str);
      shmdt(str);
      return 0;
}
[vishnu@team-osd ~]$ vi writememory.c
[vishnu@team-osd ~]$ cc writememory.c -o writememory
[vishnu@team-osd ~]$ ./writememory
Enter the string to be written in memory : hello
String written in memory: hello
```

Reading a message from shared memory

In this part of the recipe, we will learn how the message that was written into shared memory is read and displayed on screen.

How to do it…

1. Invoke the ftok function to generate an IPC key. The filename and ID that are supplied should be the same as those in the program for writing content into shared memory.
2. Invoke the shmget function to allocate a shared memory segment. The size that's specified for the allocated memory segment is 1024 and is associated with the IPC key that was generated in step 1. Create the memory segment with read and write permissions.
3. Attach the shared memory segment to the first available address in the system.
4. The content from the shared memory segment is read and displayed on screen.
5. The attached memory segment is detached from the address space.
6. The shared memory identifier is removed from the system, followed by destroying the shared memory segment.

The readmemory.c program for reading data from shared memory is as follows:

/* readmemory.c – Program to read data from the attached shared memory segment */

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
      int shmid;
      char * str;
      key_t key = ftok("sharedmem",'a');
            if ((shmid = shmget(key, 1024,0666|IPC_CREAT)) < 0) {
                  perror("shmget");
                  exit(1);
      }
            if ((str = shmat(shmid, NULL, 0)) == (char *) -1) {
                  perror("shmat");
                  exit(1);
      }
      printf("Data read from memory: %s\n",str);
            shmdt(str);
      shmctl(shmid,IPC_RMID,NULL);
      return 0;
}
[vishnu@team-osd ~]$ vi readmemory.c
[vishnu@team-osd ~]$ cc readmemory.c -o readmemory
[vishnu@team-osd ~]$ ./readmemory
Data read from memory: hello
```