13-UNIX

Shared Memory

Shared Memory, Message Queues

Chapter 45-46-47-48

Memory Organization for an Executed Program

- When a program is loaded into memory, it is organized into three areas of memory, called segments:
  - **text** segment (or **code** segment) is where the compiled code of the program itself resides.
  - **stack** segment is where memory is allocated for automatic variable within functions.
  - **heap** segment provides more stable storage of data for a program since memory allocated in the heap remains in existence for the duration of a program.

Stack

- Local variables (variables declared inside a function) are put on the stack - unless they are declared as 'static' or 'register‘
- Function parameters are allocated on the stack
- Local variables that are stored in the stack are **not** automatically initialized by the system
- Variables on the stack disappear when the function exits

Heap

- Global, static, register variables are stored on the heap before program execution begins

- They exist the entire life of the program (even if scope prevents access to them - they still exist)

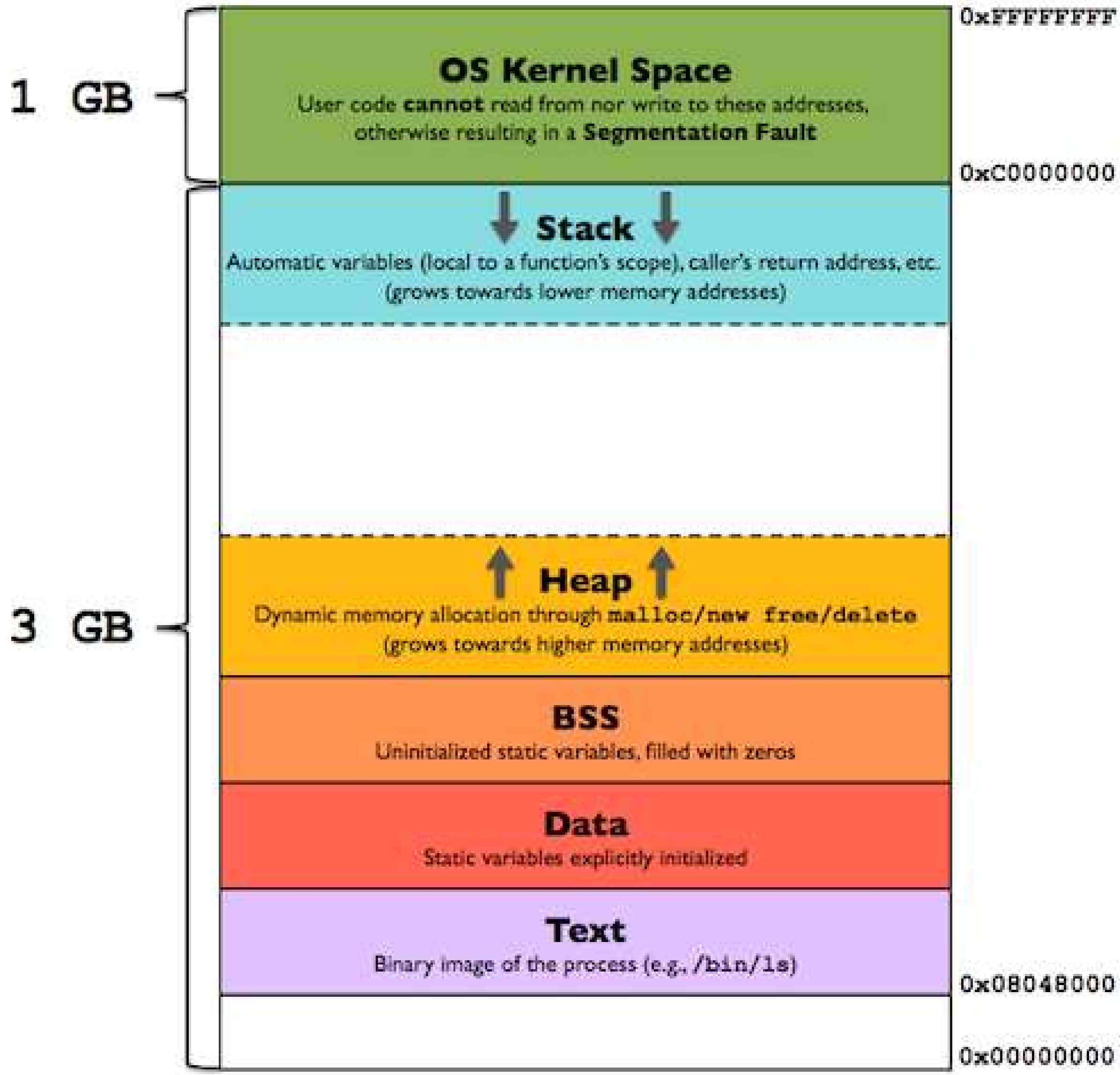
- They are initialized to zero

  Global variables are on the heap

  Static local variables are on the heap (this is how they keep their value between function calls)

- Memory allocated by new, malloc, calloc, etc., are on the heap

A typical Memory Layout on Linux X86/32



Shared Memory

- Normally, the Unix kernel prohibits one process from accessing (reading, writing) memory belonging to another process.

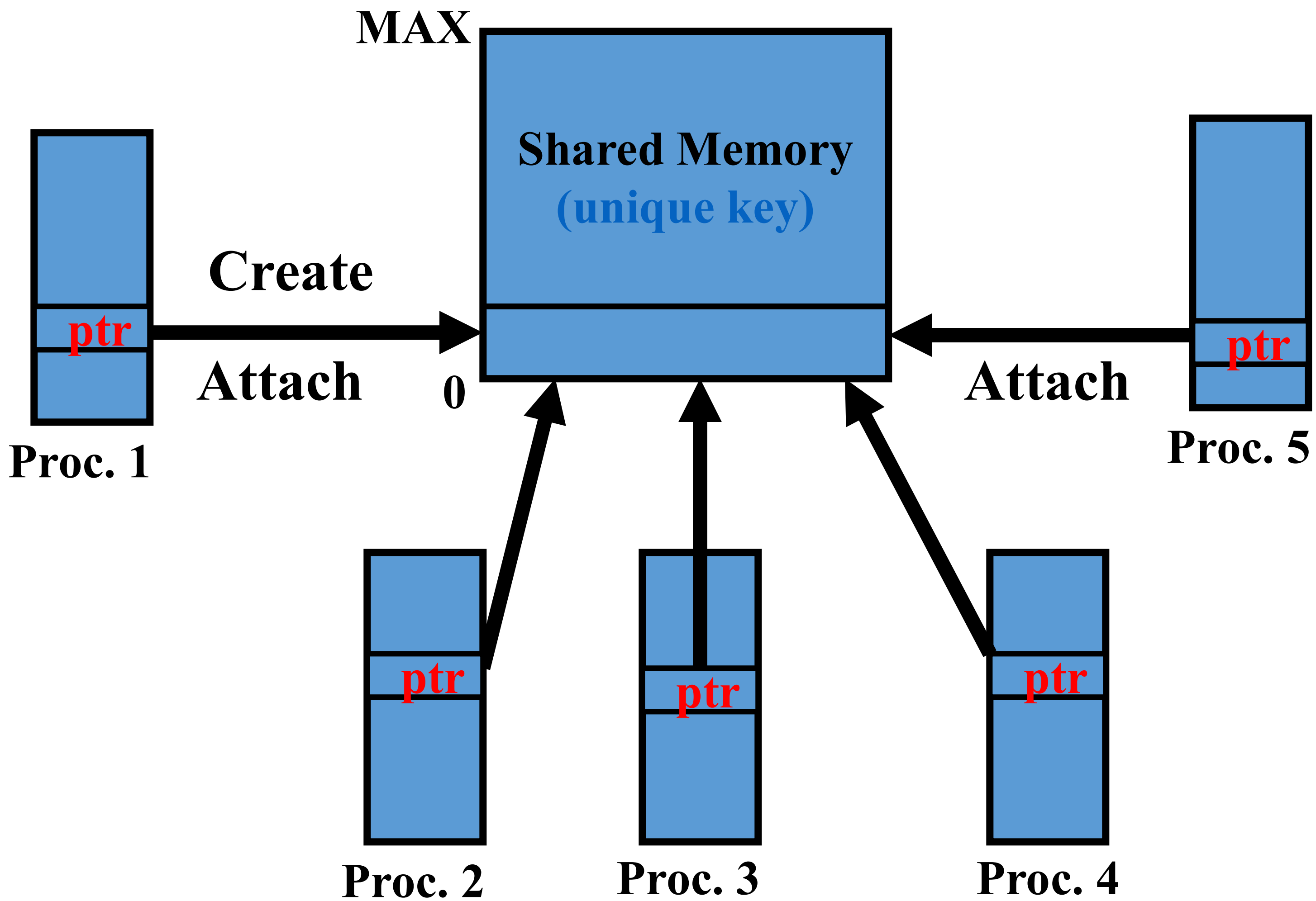
- Sometimes, however, this restriction is inconvenient.

- At such times, System V IPC Shared Memory can be created to specifically allow one process to read and/or write to memory created by another process

Shared Memory

Common chunk of read/write memory

among processes



Advantages of Shared Memory

- **Random Access**
  - you can update a small piece in the middle of a data structure, rather than the entire structure.
  - Very fast: accessed just like regular memory
- **Efficiency**
  - unlike message queues and pipes, which copy data from the process *into* memory within the kernel, shared memory is directly accessed.
  - Shared memory <u>resides in the user process memory</u>, and is then shared among other processes

Disadvantages of Shared Memory

- No automatic synchronization as in pipes or message queues (you have to provide any synchronization). Synchronize with *semaphores* or signals.

- You must remember that pointers are only valid within a given process. Thus, pointer offsets cannot be assumed to be valid across inter-process boundaries. This complicates the sharing of linked lists or binary trees.

- Processes must be on same machine.

Shared Memory:
Overview of the system calls (1 of 2)

- Create/Access Shared Memory
  - *id = **shmget**(KEY, Size, IPC_CREAT | PERM)*

- Deleting Shared Memory
  - *i = **shmctl**(id, IPC_RMID, 0)*
  - Or use ipcrm

Shared Memory: Overview of the system calls (2 of 2)

- Accessing Shared Memory
  - *memaddr =* ***shmat****(id, 0, 0)*
  - *memaddr =* ***shmat****(id, addr, 0)*
  - *memaddr =* ***shmat****(id, 0, SHM_READONLY)*
    - System will decide address to place the memory at
  - ***shmdt****(memaddr)*
    - Detach from shared memory

shmget()

This function is used to create a shared memory segment

the amount of memory required, in bytes

```c
#include <sys/shm.h>

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

returns a shared memory identifier or -1

permission bits and IPC_CREAT

a programmer defined key
i.e. #define SHM_KEY 0x1234 /* Key for shared memory segment */

shmat( )

This function attaches the shared memory segment to the process's address space.

#include <sys/shm.h>

void *shmat(int shm_id, const void *shm_addr, int shmflg);

Returns a pointer to the shared memory, or *(void* *) -1 on error.

the shared memory identifier gotten from shmget()

address in the process's address space where the shared memory is to be attached. Normally this is a NULL pointer, allowing the system to determine where.
This is the recommended form.

if non-zero, the segment is attached for read-only. If 0, it is attached read/write.

shmdt()

This function detaches the shared memory segment

```
#include <sys/shm.h>

int shmdt(const void *shm_addr);
```

returns 0 if successful or -1 if not

pointer to the shared memory segment to be detached.

shmctl()

This call performs a range of control operations on the shared memory segment identified by *shmid*.

```
#include <sys/shm.h>

int shmctl(int shm_id, int command, struct shmid_ds *buf);
```

returns 0 if successful or -1 if fails

the shared memory segment's identifier

```
struct shmid_ds {
  uid_t shm_perm.uid;
  uid_t shm_perm.gid;
  mode_t shm_perm.mode;
}
```

IPC_STAT Sets the data in the shmid structure to reflect the values associated with the shared memory.

IPC_SET Sets the values associated with the shared memory to those provided in the data structure.

IPC_RMID Delete the shared memory

Implementing a binary semaphore protocol

The following code and examples are from the textbook, written by the author of the book. They are NOT part of the Linux operating system.

If you ever need to create semaphores, you could copy (and possible alter) the functions to your project's requirements.

Implementing a binary semaphore protocol (1 of 2)

A binary semaphore can have two values:

available (free), reserved (in use)

Reserve: Attempt to reserve this semaphore for executive use. If semaphore is being reserved by another process, then block.

int reserveSem(int semId, int semNum);

Release: Free a current reserved semaphore, so that it can be reserved by another process.

int releaseSem(int semId, int semNum);

Source:
http://man7.org/tlpi/code/online/book/svsem/binary_sems.h.html (**Functions from the text book**)

Implementing a binary semaphore protocol

(2 of 2)

```
int initSemAvailable(int semId, int semNum);
        + arg.val = 1;

int initSemInUse(int semId, int semNum);
        + arg.val = 0;

int reserveSem(int semId, int semNum);

int releaseSem(int semId, int semNum);
```

---

semNum is used for identifying the semaphore within the set.
semId is for semaphore identification.

(**Functions from the text book**)

Example of shared memory

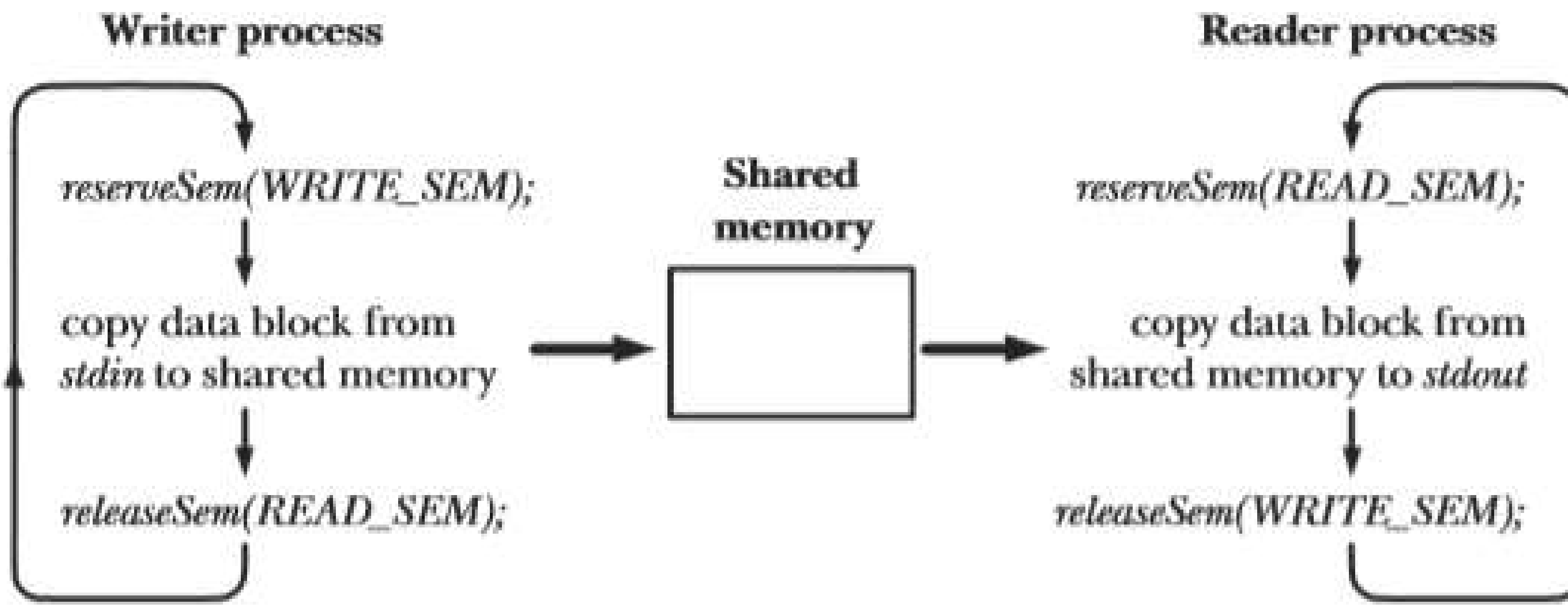


Figure 48-1: Using semaphores to ensure exclusive, alternating access to shared memory

Example of shared memory – Writer (P 1003 in LPI book)

```
main(int argc, char *argv[])
{
    int semid, shmid, bytes, xfrs;
    struct shmseg *shmp;
    union semun dummy;

    /* Create set containing two semaphores; initialize so that
       writer has first access to shared memory. */

    semid = semget(SEM_KEY, 2, IPC_CREAT | OBJ_PERMS);
    if (semid == -1)
        errExit("semget");

    if (initSemAvailable(semid, WRITE_SEM) == -1)
        errExit("initSemAvailable");
    if (initSemInUse(semid, READ_SEM) == -1)
        errExit("initSemInUse");

    /* Create shared memory; attach at address chosen by system */

    shmid = shmget(SHM_KEY, sizeof(struct shmseg), IPC_CREAT | OBJ_PERMS);
    if (shmid == -1)
        errExit("shmget");

    shmp = shmat(shmid, NULL, 0);
    if (shmp == (void *) -1)
        errExit("shmat");

    /* Transfer blocks of data from stdin to shared memory */

    for (xfrs = 0, bytes = 0; ; xfrs++, bytes += shmp->cnt) {
        if (reserveSem(semid, WRITE_SEM) == -1)         /* Wait for our turn */
            errExit("reserveSem");

        shmp->cnt = read(STDIN_FILENO, shmp->buf, BUF_SIZE);
        if (shmp->cnt == -1)
            errExit("read");

        if (releaseSem(semid, READ_SEM) == -1)          /* Give reader a turn */
            errExit("releaseSem");

        /* Have we reached EOF? We test this after giving the reader
           a turn so that it can see the 0 value in shmp->cnt. */

        if (shmp->cnt == 0)
            break;
    }

    /* Wait until reader has let us have one more turn. We then know
       reader has finished, and so we can delete the IPC objects. */

    if (reserveSem(semid, WRITE_SEM) == -1)
        errExit("reserveSem");

    if (semctl(semid, 0, IPC_RMID, dummy) == -1)
        errExit("semctl");
    if (shmdt(shmp) == -1)
        errExit("shmdt");
    if (shmctl(shmid, IPC_RMID, 0) == -1)
        errExit("shmctl");

    fprintf(stderr, "Sent %d bytes (%d xfrs)\n", bytes, xfrs);
    exit(EXIT_SUCCESS);
```

Declare shared memory segment

Create semaphore array of 2 elements

Initialize semaphores

Create and attach shared memory

Wait for Writer turn

Give Reader a turn

Side note on following code: (1 of 2)

It uses “union” which is like a structure.

A ***structure*** is an object consisting of a sequence of named members of various types.

A ***union*** is an object that contains, at different times, any one of several members of various types.

The following code is from: #include "semun.h“

```
union semun {            /* Used in calls to semctl() */
    int               val;
    struct semid_ds *  buf;
    unsigned short *   array;
#if defined(__linux__)
    struct seminfo *   __buf;
#endif
};
```

Side note on following code: (2 of 2)

I found more information with a Google search.

“In c, the difference between struct and union”

Here are links to two articles with more information.

http://cs-fundamentals.com/tech-interview/c/difference-between-structure-and-union-in-c-language.php

http://www.thecrazyprogrammer.com/2015/03/difference-between-structure-and-union.htmlnion-in-c-language.php

Example of shared memory – **Writer**
Transfer blocks of data from *stdin* to a system V shared memory segment (LPI, P. 1003) (**1 OF 6**)

```
/* TLPI page 1003, Listing 48-2 */
#include "semun.h"
#include "svshm_xfr.h"

int main(int agrc, char *argv[ ]) {
   int semid, shmid, bytes, xfrs;
   struct shmseg *shmp;
   union semun dummy;

/* Create a set containing the two semaphores that are used by the writer and
   reader program to ensure that they alternate in accessing the shared
   memory segment. The semaphores are initialized so that the writer has
   first access to the shared memory segment.  Since the writer creates the
   semaphore set, it must be started before the reader. */
```

Declare a shared memory segment

Example of shared memory – **Writer** (2 OF 6)

```
semid = semget(SEM-KEY, 2, IPC_CREAT | OBJ_PERMS);
if (semid == -1)
        errExit("semget");
if (initSemAvailable(semid, WRITE_SEM) ==  -1)
        errExit("initSemAvailable");
if (initSemInUse(semid, READ_SEM) == -1)
        errExit("initSemInUse");

 /* Create the shared memory segment and attach it to the writer's virtual
    address space at an address chosen by the system. */
shmid = shmget(SHM_KEY, sizeof(strut shmseg), IPC_CREAT | OBJ_PERMS);
if (shmid == -1)                /* build the "pipe" before creating child process */
        errExit("shmget");
shmp = shmat(shmid, NULL, 0);
if (shmp == (void *) -1)
        errExit("shmat");
```

Create semaphore array of 2 elements

Initialize semaphores

Create and attach shared memory

Example of shared memory – **Writer** (3 OF 6)

/* Transfer blocks of data from *stdin* to shared memory */

/* Enter a loop that transfers data from standard input to the shared memory segment. */

/* The following steps are performed in each **loop** iteration:

Reserve (decrement) the writer semaphore.

Read data from standard input into the shared memory segment.

Release (increment) the reader semaphore. */

Example of shared memory – **Writer** (4 OF 6)

```
for (xfrs = 0, bytes = 0; ; xfrs++, bytes += shmp->cnt) {
    if (reserveSem(semid, WRITE_SEM) == -1)
            errExit("reserveSem");

    shmp->cnt = read(STDIN_FILENO, shmp->buf, BUF_SIZE);
    if (shmp->cnt) == -1)
       errExit("read");

    if (releaseSem(semid, READ_SEM) == -1)
       errExit   ("reserveSem");
```

Wait for Writer turn

Give for Reader turn

Note: The *for* loop has no terminating value.

Example of shared memory – **Writer** (5 OF 6)

```
/* Have we reached EOF? We test this after giving the reader
    a turn so that it can see the 0 value in shmp->cnt.  */
/* The loop terminates when no further data is available from standard
    input.
    On the last pass through the loop, the writer indicates to the
    reader that there is no more data by passing a block of data of
    length 0 (shmp->cnt is 0).  */

if (shmp->cnt  ==  0)
    break;

}       /* end of for loop */
```

When we reach 0,
break out of the infinite loop

Example of shared memory – **Writer** (6 OF 6)

```
/* Wait until reader has let us have one more turn.  We then know
   reader has finished, and so we can delete the IPC objects. */
/* Upon exiting the loop, the writer once more reserves its semaphore, so that it
   knows that the reader has completed the final access to the shared memory. */
if (reserveSem(semid, WRITE_SEM) == -1)
        errExit("reserveSem");

/* The writer then removes the shared memory segment and semaphore set. */
if (semctl(semid, 0, IPC_RMID, dummy) == -1)
        errExit("semctl");
if (shmdt(shmp) == -1)
        errExit("shmdt");
if (shmctl(shmid, IPC_RMID, 0) == -1)
        errExit("shmctl");

fprintf(stderr, "Send %d bytes (%d xfrs)\n", bytes, xfers);
exit(EXIT_SUCCESS);
}
```

Example of shared memory – Reader (p 1005 LPI Book)

```
main(int argc, char *argv[])
{
    int semid, shmid, xfrs, bytes;
    struct shmseg *shmp;

    /* Get IDs for semaphore set and shared memory created by writer */

    semid = semget(SEM_KEY, 0, 0);
    if (semid == -1)
        errExit("semget");

    shmid  = shmget(SHM_KEY, 0, 0);
    if (shmid == -1)
        errExit("shmget");

    /* Attach shared memory read-only, as we will only read */

    shmp = shmat(shmid, NULL, SHM_RDONLY);
    if (shmp == (void *) -1)
        errExit("shmat");

    /* Transfer blocks of data from shared memory to stdout */

    for (xfrs = 0, bytes = 0; ; xfrs++) {
        if (reserveSem(semid, READ_SEM) == -1)          /* Wait for our turn */
            errExit("reserveSem");

        if (shmp->cnt == 0)                     /* Writer encountered EOF */
            break;
        bytes += shmp->cnt;

        if (write(STDOUT_FILENO, shmp->buf, shmp->cnt) != shmp->cnt)
            fatal("partial/failed write");

        if (releaseSem(semid, WRITE_SEM) == -1)         /* Give writer a turn */
            errExit("releaseSem");
    }

    if (shmdt(shmp) == -1)
        errExit("shmdt");

    /* Give writer one more turn, so it can clean up */

    if (releaseSem(semid, WRITE_SEM) == -1)
        errExit("releaseSem");

    fprintf(stderr, "Received %d bytes (%d xfrs)\n", bytes, xfrs);
```

Get semaphore id

Attach shared memory

Wait for Reader turn

Give Writer a turn

Example of shared memory – **Reader**
Transfer blocks of data from a system V shared memory segment to *stdout* (LPI, P. 1005) (**1 OF 4**)

```
/* LPI page 1005, Listing 48-3 */
#include "svshm_xfr.h"
int main(int agrc, char *argv[ ]) {
   int semid, shmid, bytes, xfrs;
   struct shmseg *shmp;

   /* Get IDs for semaphore set and shared memory created by writer */
   /* Obtain the IDs of the semaphore set and shared memory segment
      that were created by the writer program. */

   semid = semget(SEM-KEY, 0, 0);
   if (semid == -1)
        errExit("semget");
```

Declare a shared memory segment

Get semaphore id

Example of shared memory – **Reader** (2 OF 4)

```
shmid = shmget(SHM_KEY, 0, 0);
if (shmid == -1)
    errExit("shmget");

/* Attach the shared memory segment for read-only access. */
shmp = shmat(shmid, NULL, SHM_RDONLY);
if (shmp == (void *) -1)
    errExit("shmat");
```

Attach shared memory

Example of shared memory – **Reader** (3 OF 4)

```
/* Transfer blocks of data from shared memory to stdout */
for (xfers = 0, bytes = 0; xfers++) {
    if(reserveSem(semid (READ_SEM) == -1)        /* Wait for our turn */
        errExit("reserveSen");
    if (shmp->cnt == 0)                          /* Writer encountered EOF */
        break;
    bytes += shmp->cnt;

    if (write(STDOUT_FILENO, shmp->buf, shmp->cnt, != shmp->cnt)
        fatal("partial/failed write");
    if(releaseSem(semid, WRITE_SEM) == -1)
        errExit("releaseSem");

}  /* end of for loop */
```

Wait for Reader turn

Give Writer a turn

Example of shared memory – **Reader** (4 OF 4)

```
/* After exiting the loop, detach the shared memory segment */
 if (shmdt(shmp) == -1)
      errExit("shmdt");

/* Release the writer semaphore, so that the writer program can remove the
   IPC objects */
/* Give writer one more turn, so it can clean up */
if (reserveSem(semid, WRITE_SEM) == -1)
      errExit("reserveSem");

fprintf(stderr, "Received %d bytes (%d xfrs)\n", bytes, xfers);
exit(EXIT_SUCCESS);
}
```

Shared memory demo

Notes:
Go to sp2
Bring up the Writer then do **^z**
Bring up the Reader with **&**
Type **jobs**
%1 to bring Writer to **fg**

Type words, then Enter.
Get Echo

Ctrl-d (EOF) both go away.

If looking for this code in the on-line distribution, it will be in:
tlpi-dist/svshm

Writer =
svshm_xfr_writer

Reader =
svshm_xfr_reader

Message Queues

Message Queue

- A Message Queue is a linked list of message structures stored inside the kernel's memory space and accessible by multiple processes

- Synchronization is provided automatically by the kernel

- New messages are added at the end of the queue

- Each message structure has a long *message type*

- Messages may be obtained from the queue either in a FIFO manner (default) or by requesting a specific *type* of message (based on *message type*)

msgget()

This function creates a message queue

#include <sys/msg.h>

int **msgget**(key_t key, int msgflg);

returns a queue identifier if successful, otherwise, -1

a key value

permission bits and IPC_CREAT

msgsnd()

The msgsnd() system call is used to send messages to a message queue.

#include <sys/msg.h>

int msgsnd(int msgid, const void *msg_ptr, size_t msgsize, int msgflg);

Returns 0 if successful otherwise -1

The message queue identifier from msgget()

```
struct my_message
{
   long int message_type;
   // the data to transfer
}
```

the message size. Does not include the type.

IPC_NOWAIT   returns without sending the message if the queue is full when set. Otherwise the process waits for space to become available in the queue.

msgrcv()

The msgrcv() system call is used receive messages from a message queue.

#include <sys/msg.h>

```
int msgrcv(int msgid, void *msg_ptr, size_t msg_sz,
           long int msgtype, int msgflg);
```

Message size

0 if successful otherwise -1

The message queue id

if 0, the next message is retrieved. If non-zero, the next message with this message type is retrieved

The message is copied here. Includes the long int.

IPC_NOWAIT if set, the call will return immediately. Otherwise, it waits until a message of the specified type is available to be read.

msgctl()

Performs the control operation specified by cmd on the message queue with identifier msgid

#include <sys/msg.h>

int msgctl(int msgid, int command, struct msgid_ds *buf);

message queue id

0 if successful
otherwise -1

IPC_STAT loads data in msgid_ds structure
IPC_SET gets the data from the msgid_ds structure
IPC_RMID deletes the message queue

13-UNIX

Shared Memory

Shared Memory, Message Queues

The End