Topic 7

Inter-Process Communications Mechanisms in Unix

Objectives

- Understand why IPCs are needed for processes to communicate with each other.
- Understand and be able to use unnamed pipes
- Understand and be able to use named pipes (FIFO)
- Understand and be able to program with Message Queues
- Understand and be able to use Semaphores to synchronise multiple processes and to provide mutual exclusion.
- Understand and be able to use Shared Memory to exchange data between processes.

Readings

- This lecture notes.
- Stevens & Rago: Ch 15
- Stallings: Ch 5.3-5.7

1. Introduction

We understand that each process works within its own virtual address space. Processes do not normally share memory space, so how do they communicate with each other?

- signals: - cannot pass data
- shared files: - too slow

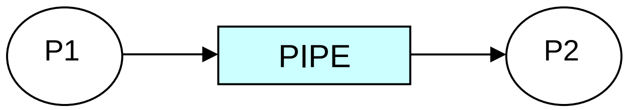
Unix systems provide several inter-process communication mechanisms (IPCs) to suit different applications:

BSD & SV	pipes FIFOS Stream pipes named stream pipes	on the same machine
SV	message queues semaphores shared memory	on the same machine
BSD & SV	sockets	Mainly for network applications running on different machines
SV	streams	Same as above

In this topic we will discuss pipes, FIFOs, message queues, semaphores and shared memory. Sockets will be dealt with in Topic 8 when we discuss network programming.

2. Pipes

A pipe provides a one-way communication channel between *related* processes.



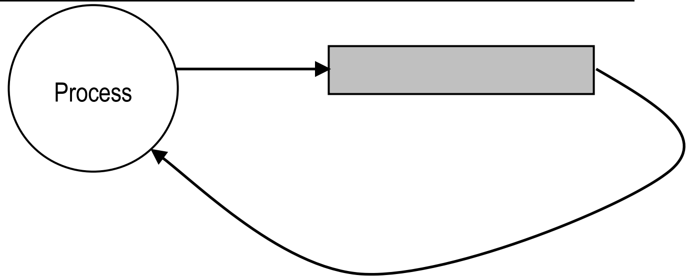

```
#include <unistd.h>

int pipe (int filedesc[2]);
    Return 0 if ok,
         -1 on error
```

- A pipe can be used only to communicate between processes sharing a common ancestry in which the pipe was created.

- A pipe is uni-directional from the write end to the read end, although more than one process can read from (or write to) the same pipe.

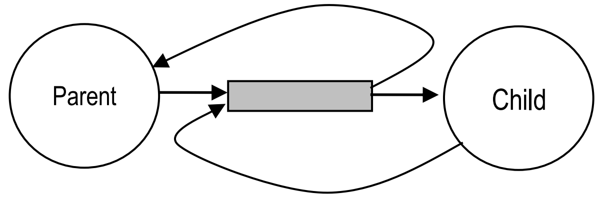
- A pipe is implemented with a First-In First-Out (FIFO) buffer in the memory. *A read is blocked if the pipe is empty. A write is blocked if the pipe is full.*
- If a pipe call is successful, `filedesc[0]` is opened for reading from the pipe and `filedesc[1]` is opened for writing to the pipe.

Example1



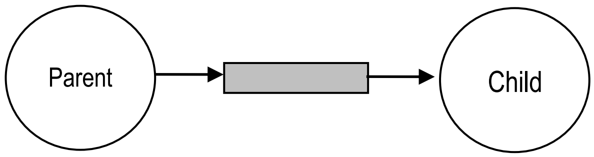

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFSIZE 512

char *msg = "Hello, me!";

int main()
{
     char buf[BUFSIZE];
     int p[2]; // p[0] for read, p[1] for write

     // open a pipe
     if(pipe(p) < 0){
        perror ("Pipe call");
        exit(1);
     }

     // send msg to pipe
     write(p[1], msg, strlen(msg)+1);
              // include null character

     // read from pipe
     read(p[0], buf, BUFSIZE);
     printf("%s\n", buf);
     exit(0);
}
```

Example 2




```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFSIZE 512
char *msg = "Hello";

int main()
{
     char buf[BUFSIZE];
     int p[2], pid;

     if(pipe(p) < 0){
          perror ("pipe call");  exit(1);
     }

     // the pipe must be created before
     // the child process
     if((pid=fork()) < 0){
          perror("Fork call"); exit(1);
     }

     if(pid > 0){ // parent
          write (p[1], msg, strlen(msg) + 1);
          wait ((int *) 0);
     }

     if(pid == 0){ // child
          read (p[0], buf, BUFSIZE);
          printf("%s\n", buf);
     }

     exit(0);
}
```

The above code can be modified so that there is a cleaner interface between the two processes:




```
if(pid > 0){ // parent
     close(p[0]);
     write(p[1], msg, strlen(msg) + 1);
     wait((int *)0);
          // wait for the child to end first
}

if(pid == 0){ // child
     close(p[1]);
     read(p[0], buf, BUFSIZE);
     printf("%s\n", buf);
}
```



```
if(pid > 0){ // parent
   close(p[1]);
   read(p[0], buf, BUFSIZE);
   printf("%s\n", buf);
}

if (pid == 0){ // child
   close(p[0]);
   write(p[1], msg, strlen(msg) + 1);
}
```

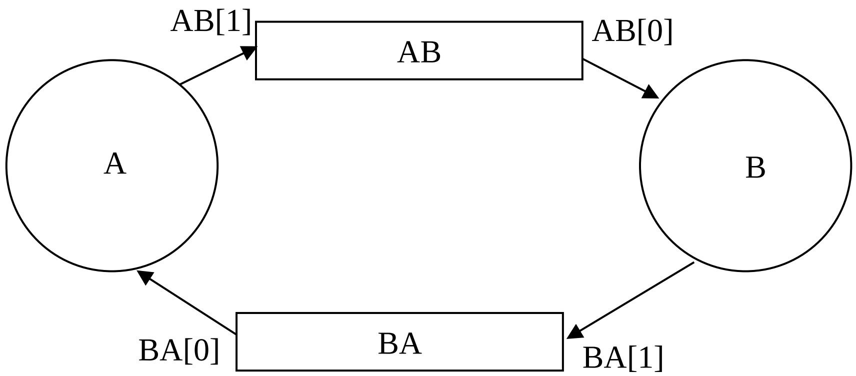
3. The size of the pipe

A pipe is implemented using an internal buffer in the kernel space. The size of the buffer is fixed. POSIX.1 specifies that the size of the buffer must be at least 512 bytes. Most operating systems use a much larger buffer than this minimum size.

The size of the buffer and the number of bytes read or written could affect the behaviours of the read and write system calls:

- if the pipe is empty, a read call will block the calling process
- if the pipe is full, a write call will block the calling process
- if the write end of a pipe is closed and the pipe is empty, a read will return 0 immediately
- if the read end of the pipe is closed, the process which attempts to write to the pipe will be sent the signal `SIGPIPE` by the kernel. If the signal is not caught or ignored, the process will terminate; otherwise the write call will return –1.

The following example demonstrates the above situations:




```
/* pipesync.c - demonstrating synchronisation
 * and deadlocks when using pipes
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char *msg="Hello, there";
int BUFSIZE = 100;

// convert upper case to lower case or vice versa
// (simulating some kind of service)
void conv(char *s, int len)
{
   char c;
   int i;
   for(i=0; i<len; ++i){
      c = s[i];
      if (islower(c))
         s[i]=toupper(c);
      else if (isupper(c))
         s[i]=tolower(c);
   }
}
```

```
int main()
{
   pid_t pid;
   int AB[2], BA[2];
   char buf[BUFSIZE];
   int i, n;

   // create two pipes
   if (pipe(AB)<0 || pipe(BA)<0)
      perror("pipe calls"), exit(1);

   // create process B
   if((pid=fork())<0)
      perror("fork call"), exit(2);

   if (pid>0) {        /* process A */
      close(AB[0]);
      close(BA[1]);                    // L1

      printf("   A sends: %s\n", msg);
      write(AB[1], msg, strlen(msg));
      close(AB[1]);                    // L2

      printf("A receives: ");
      while((n=read(BA[0], buf, BUFSIZE))>0)
         buf[n]='\0', printf("%s", buf);
      printf("\n"); exit(0);
   }

   if (pid==0) {     /* porcess B */
      close(BA[0]);
      close(AB[1]);                    // L3

      while((n=read(AB[0], buf, BUFSIZE))>0){
         conv(buf, n);
         write(BA[1], buf, n);
      }
      exit(0);
   }
}
```

Refer to line labels in the previous program, explain:

What would happen if:

(1) Line L1: "`close(BA[1])`" is deleted?

(2) Line L2: "`close(AB[1])`" is deleted?

(3) Line L3: "`close(AB[1])`" is deleted?

When discussing the behaviour of the program, please remember the behaviours of the read and write system calls when a pipe is empty or full, as detailed at the beginning of this section.

4. Non-blocking Read and Write

Occasionally, we want to read from a pipe and also be able to return immediately if the pipe is empty. Similarly, we may want to write to a pipe and also be able to return immediately if the pipe is full. This is achieved by adding the file status flag `O_NONBLOCK` to the file descriptors:

```
#include <fcntl.h>

int val;
val = fcntl (fd, F_GETFL, 0);
val = val | O_NONBLOCK;  // adding the bit
if (fcntl( fd, F_SETFL, val) == -1)
     perror("fcntl call");
```


Assuming that `fd` is the read end of a pipe and the pipe is empty, with non-blocking mode, `read(fd, ...)` will return immediately with return value `-1` and the `errno` is set to `EAGAIN`.

Similarly, if `fd` is the write end of the pipe, `write(fd, …)` will return immediately with return value `-1` and `errno` set to `EAGAIN` when the pipe is full.

5. FIFOs or Named Pipes

Limitations of the conventional pipes:

- They cannot be used to communicate between unrelated processes (i.e., processes not sharing the same ancestor where the pipe was created)
- They are not permanent. They vanish when the relevant processes terminate.

To address this problem, Unix systems introduced FIFO or named pipes (because a FIFO is a special file with a file name.) Like any other special file, a FIFO can be opened or closed by system calls: `open` and `close`. Like a conventional pipe, a FIFO can be read or written with system calls `read` and `write`.

Create a FIFO

(1) On the command line:


```
% mkfifo example.fifo
```

The file access mode will be `666 & ~umask`

(2) In a program:

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo (const char *pathname, mode_t mode);
```

Returns: `0` if OK, `-1` on error

Example:

```
if(mkfifo ("example.fifo", 0666) < 0){
   printf("Unable to create fifo\n");
   exit(0);
}
```

Once a FIFO is created, the normal system calls such as `open`, `read`, `write`, `close` for files can then be used to access it.

Example:

Create a fifo to communicate with its child

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#define BUFSIZE 512

int main()
{
   char buf[BUFSIZE];
   int pid, fd;
   char *msg = “Hello”;

   // create a fifo
   unlink("/tmp/example.fifo");
   if(mkfifo("/tmp/example.fifo", 0666) < 0){
      printf("Unable to create fifo\n"); exit(1);
   }

   if((pid=fork() < 0){
      printf(“Unable to fork\n”);
      exit(1);
   }

   // in child
   if(pid == 0){
      if((fd = open(“/tmp/example.fifo”, O_RDONLY)) < 0){
         printf("Error opening /tmp/example.fifo\n");
         exit(1);
      }
      if(read(fd, buf, BUFSIZE) < 0)
         printf("Error in reading FIFO\n"), exit(1);
      printf("FIFO: %s\n", buf);
      exit(0);
   }

   // in parent
   if((fd = open("/tmp/example.fifo", O_WRONLY)) < 0 )
       printf(“Unable to open fifo \n”), exit (1);
   write (fd, msg, strlen(msg) + 1);
   exit(0);
}
```

6. System V IPC

In the next few sections, we will discuss three types of IPCs originated from System V, hence are often referred to as System V IPCs. These three types of IPCs are:

- **Message Queue**: a linked list of messages, each with a priority or type number. Multiple processes may access send and retrieve messages to/from the message queue at any order, not necessarily in the first-in-first-out order.
- **Semaphores**: semaphores provide controlled access to shared resources, such as a shared file or a shared memory. They are often used to implement mutual exclusions and synchronisations.
- **Shared Memory**: a shared memory is a piece of physical memory in the kernel. Multiple processes can each map a piece of their own virtual address spaces to that same physical memory, thus sharing it with each other. A shared memory provides a simple and fast way for multiple processes to exchange data. However, the access to the shared memory must be controlled to ensure the integrity of the data. The access control is often implemented with one or more semaphores.

Message queues, semaphores and shared memory are system-wide resources. Once created, they stay in the system until explicitly removed. Each IPC object is associated with a unique key, much like a pathname for a file. It is accessed through a unique id, similar to a file descriptor. The id is unique within the entire operating system, thus different processes would access the same IPC object with the same id. However, the id is allocated each time the IPC object is created (just like a file descriptor which is allocated each time you open the file). Hence, objects with the same key may have different ids at different times.

As system-wide resources, we can query and remove message queues, semaphores and shared memories at the command line by using the command `ipcs` and `ipcrm`. For example, the following command shows that there is one shared memory object owned by the root, one semaphore object owned by root, and two message queues created by the user "hong".

```
$ ipcs

------ Shared Memory Segments --------
key        shmid  owner  perms  bytes  nattch  status
0x73727372 0      root   666    44172  1

------ Semaphore Arrays --------
key        semid  owner  perms  nsems   status
0x6c737372 0      root   666    3

------ Message Queues --------
key        msqid  owner  perms  used-bytes messages
0x0001a000 1792  hong    644    0          0
0x0001a001 1793  hong    644    0          0
```


In the above example, each IPC object has a key (e.g., `0x0001a000`) and an id (e.g., `1792`). To create an IPC object, we must firstly have a unique key (an integer). When the object is created, it is allocated an id (another integer) for that object. Subsequent accesses to the object are through the id, not the key.

Like a file, each IPC object also has a set of access permissions that determines which processes are allowed to access it. However, the “execute” permission doesn’t make sense here.

The owner can remove an IPC object from the command line using `ipcrm` command, e.g.:

```
$ ipcrm  msg  1793
resource deleted
```

To create a new IPC object, we must either find a unique key explicitly or use the constant `IPC_PRIVATE`. If we use the latter, the system will allocate a unique key to the IPC object *each* time. There are pros and cons with each approach:

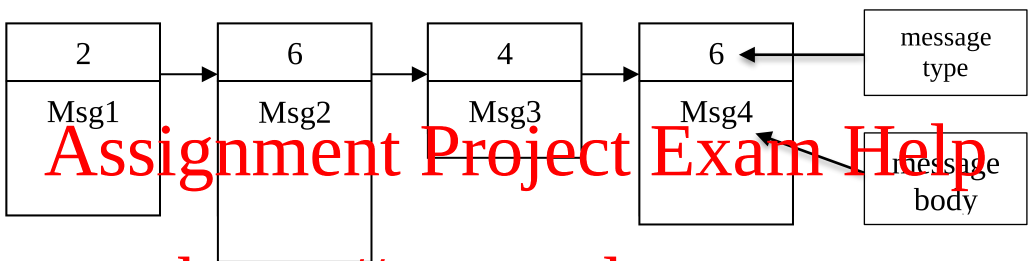
Using an explicit key: we can define the key in a header file. Both the client and the server can use the same key. The difficulty is how to avoid clashes (when the same key is also used by different programs).

Using the constant `IPC_PRIVATE`: it is guaranteed that each time the system will allocate a new key that is not already used by another IPC object. The difficulty is that, if Process A creates an object, using say, `msgget(IPC_PRIVATE, ...)`, to be accessed by Process B, how would Process B gain access to that object, without knowing the key Process A used? Note that Process B cannot use `msgget (IPC_PRIVATE, ...)` to obtain the id, since each time `IPC_PRIVATE` is mapped to a new, unused key.

The good news is that the constant `IPC_PRIVATE` can be used when the communications are among the processes that share the common ancestry where the IPC object was created. This is because these child processes would inherit the id from the parent process after forking. Using `IPC_PRIVATE` will save us the worry of key clashes.

7. Message Queues

A message queue is a linked list of messages in the kernel memory. The message queue is system wide. Therefore, once the queue is created, a process can send a message to the queue. Another process can retrieve the message from that queue as long as the processes have the relevant permissions. Messages may have different sizes. Each message carries with it a message type, which may affect the order it is retrieved. Unlike a pipe, it is not necessary to retrieve the messages in the First-In-First-Out order.



Create a message queue:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int mqid;

mqid = msgget(key_t key,  int  permflags);
```

The `key` is a long integer. To create a new message queue, the key must not already be used by another message queue. The `permflags` are access permissions for the queue, possibly bit-wise Ored with the following flags:

- `IPC_CREAT`: create a new message queue if the message queue associated with the key does not exist.
- `IPC_EXCL`: if `IPC_CREAT` and `IPC_EXCL` are both set, create the queue if no queue with the same key exists and return its id. If the queue already exists, return `-1` with `errno` set to `EEXIST`.

The system call returns the message queue id associated with the given key.

Example 1:

```
int mqid;
key_t   key = 0x000A1800;

mqid = msgget(key, 0600 | IPC_CREAT);
```

If the queue with the key does not exist, create it and return its id. If it already exists, just return its id.

Example 2:

```
mqid = msgget(key, 0600 | IPC_CREAT | IPC_EXCL);
```

If the queue does not exist, create it and return its id. Otherwise, return `-1` immediately with `errno` set to `EEXIST`.

Sending a message

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd( int mqid, void *message,
            int size, int flag );
```

Here `message` must point to a structure similar to this:

```
struct mymsg{
  long  mtype;        // message type, >0
  char  mtext[...];   // message text
}
```

The message type is used to categorise the messages. The value must be positive. The argument `size` is the length of the message stored in `message->mtext`.

The `flag` can be set to `IPC_NOWAIT`. Without this flag, the calling process will be blocked if the message cannot be sent out completely, for example, when the total length of the message queue exceeds the per-queue limit or the system-wide limit. With this flag, the call returns immediately in this situation with return value `–1` and `errno` set to `EAGAIN`.

Retrieve a message

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int mqid, void *message,
           int size, long type, int flag );
```

Here `message`'s type is the same as in `msgsnd`, `size` is the size of the `mtext` buff in the `message` structure. The `type` decides how the message is retrieved.

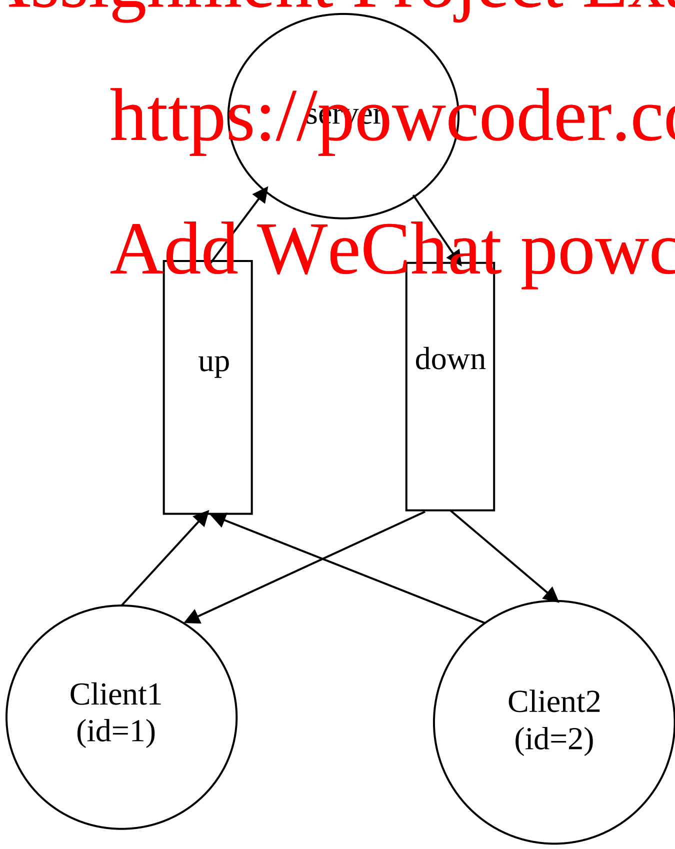
- `type = 0`: the first message in the queue is returned
- `type > 0`: the first message in the queue whose message type `mtype` is `type`.
- `type < 0`: the first message in the queue whose message type `mtype` is the lowest value less than or equal to `|type|`.

The `flag` can be one of the following two:

- `MSG_NOERROR`: if the returned message is larger than the given size, the message is truncated. The truncated part of the message is lost. No notification is given. Without this flag, the system call returns with return value –1 and `errno` set to `E2BIG`. The message is not retrieved. It stays in the queue.
- `MSG_NOWAIT:` similar to `msgsnd`, if no message can be retrieved, the calling process is suspended if the flag is not set. With the flag, the system call returns immediately with return value –1 and `errno` = `ENOMSG`.

8. A Client Server Example using Message Queues

This example consists of one server process that simulates a service (to keep it as simple as possible, it converts the case of letters in a message it received) to one or several clients. There are two message queues: *up* and *down*. The client reads (repeatedly) a message from the standard input, then sends the message to the server for processing via the *up* queue. The server retrieves a message from the *up* queue, then sends the processed message to the *down* queue for the client to collect. There can be several clients working simultaneously, but each client must have a unique client id which is inserted into its messages.



```
/* name:          msgq.h
 *                - message queue example
 * author:        HX
*/


#include <sys/msg.h>

#define  UPQ              0x001A000    /* IPC key for the up queue */
#define  DOWNQ            0x001A001    /* IPC key for the down queue */
#define  MAXMSG           256          /* maximum size of any message */
#define  MAXTYPE          10           /* maximum type number */
```

```
/* name:          msgq_client.c
 *                - message queue (client part)
 * author:        HX
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include "msgq.h"

extern int errno;

int main(int argc, char *argv[])
{
     int type;
     int up, down;
     struct message {
          long mtype;
          char mtext[MAXMSG];
     } msg;
     char *msgp; int n;

     if (argc != 2){
         printf("Usage: %s <a unique type number>\n", argv[0]);
         exit(1);
     }

     type = atoi(argv[1]);
     if (type<=0 || type > MAXTYPE){
        printf("The type number must be between 1 and %d\n", MAXTYPE);
        exit(2);
     }

     /* get the queue ids */
     if ((up=msgget((key_t)UPQ, 0))<0)
         perror("msgget - up"), exit(1);
     if ((down=msgget((key_t) DOWNQ, 0))<0)
         perror("msgget - down"), exit(1);
```

```
    while (1) {
        msgp = fgets(msg.mtext, MAXMSG, stdin);
        if (msgp==NULL) break;   // end of file (ie, Ctrl-D) or error
        msg.mtype = type;

        if (msgsnd(up, (void *)&msg, strlen(msg.mtext)+1, 0)<0) {
            if (errno==EIDRM)
               printf("Message queue (%d) removed\n", UPQ);
            else if (errno==EACCES)
               printf("No permission to send message\n");
            else
               printf("Cann't send message\n");
            exit(2);
        }

        if (msgrcv(down, (void *)&msg, MAXMSG, type, 0)<0) {
            if (errno==EIDRM)
               printf("Message queue (%d) removed\n", DOWNQ);
            else if (errno==EACCES)
               printf("No permission to send message\n");
            else
               printf("Can't retrieve message\n");
            exit(2);
        }

        printf("Msg processed: %s\n", msg.mtext);
    }
    exit(0);
}
```

```
/* name:          msgq_server.c
 *                - message queue, server part
 * author:        HX
*/

#include <ctype.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include "msgq.h"     /* for the IPC keys for the two msg queues */

extern errno;

/* turn a process into a daemon, see Topic 4 */
int daemon_init(void)
{
     pid_t  pid;

     if ( (pid = fork() ) < 0) return (-1);
     else if (pid != 0)
           printf("server pid=%d\n", pid), exit(0);

     /* child continues */
     setsid();       /* become session leader */
     chdir("/");       /* change current working directory */
     umask(0);    /* clear out file mode creation mask */
     return (0);
}
```



```
/* convert upper case to lower case or vice versa */
void conv(char *msg, int size)
{
     int i;

     for (i=0; i<size; ++i)
        if (islower(msg[i]))  msg[i] =  toupper(msg[i]);
        else if (isupper(msg[i]))  msg[i] =  tolower(msg[i]);
}



int main(int agrc, char *argv[])
{
     int up, down;  /* msg queue ids for up and down queues */
     FILE *log; char logfilename[200];
     struct message {
          long mtype;
          char mtext[MAXMSG];
     } msg;
     int n;

     /* create the message queue UPQ */
     if ((up=msgget((key_t) UPQ, 0644 | IPC_CREAT | IPC_EXCL ))<0)
```

```
        if (errno == EEXIST) {
            printf("The msg queue %d exists. Remove it first\n", UPQ);
            exit(1);
        } else
            printf("Error: cann't create UPQ queue\n"), exit(2);

    /* create the message queue DOWNQ */
    if ((down=msgget((key_t) DOWNQ, 0644 | IPC_CREAT | IPC_EXCL ))<0)
        if (errno == EEXIST) {
            printf("The msg queue %d exists. Remove it first\n", DOWNQ);
            exit(1);
        } else
            printf("Error: cann't create DOWNQ queue\n"), exit(2);

    /* create a log file (since daemon cannot write to terminal */
    sprintf(logfilename, "%s.log", argv[0]);   /* log file name */
    log = fopen(logfilename, "w");
    if (log == NULL) {
        printf("Error: cann't create log file %s\n", logfilename);
        exit(3);
    }

    /* become a daemon */
    if (daemon_init() == -1)
        printf("cannot become a daemon\n"), exit(4);
```



```
    /* save the daemon pid to log file */
    fprintf(log, "server pid=%d\n", getpid()); fflush(log);
```



```
    /* repeatedly get a message with the lowest type number from the Up
       queue, process it (case conversion), and then send the processed
       message to the Down queue
     */
```



```
    while (1) {
        // get a message with the lowest type number from Up queue
        RECV:
        if ((n=msgrcv(up, (void *)&msg, MAXMSG, -MAXTYPE,
                                      MSG_NOERROR))<0){
            if (errno==EINTR) {
                fprintf(log, "msgrcv interrupted by a singal.");
                fprintf(log, "continue to receive\n"); fflush(log);
                goto RECV;
            } else if (errno==EIDRM) {
                fprintf(log, "msg queue UPQ (%d) removed\n", UPQ);
                exit(5);
            } else {
                fprintf(log, "error reading from UPQ queue\n");
                exit(6);
            }
        }

        /* process the message */
        conv(msg.mtext, n);


        /* the send the message to Down queue */
        SEND:
```

```
        if (msgsnd(down, (void *)&msg, n, 0)<0) {
            if (errno==EINTR) {
                fprintf(log, "msgrcv interrupted by a signal.”);
                fprintf(log, “Continue to send\n");
                fflush(log);
                goto SEND;
            } if (errno==EIDRM) {
                 fprintf(log, "msg queue down (%d) removed\n", DOWNQ);
                 exit(5);
            } else {
                 fprintf(log, "error sending msg to DOWNQ queue\n");
                 exit(6);
            }
        }
    }
}
```

9. Semaphores

A semaphore is an integer counter used for controlling the sharing of resources (such as files and shared memories). The concept was originally proposed by E W Dijkstra. There are two *atomic* operations on a semaphore, *wait* and *signal*. These two operations are often known as ***P*** (the Dutch word *proberen* – to test) and ***V*** (the Dutch word *verhogen*, to increment).

In our textbook (Stallings), ***P*** operation is called ***semWait*** and ***V*** operation is called ***semSignal***.

```
int sem = 1;

P(sem): atomic {
    wait until sem>0;
         //so the calling process may be suspended!
    sem = sem -1;
}

V(sem): atomic {
    sem = sem + 1;
    wake up all processes that are waiting for sem>0;
}
```


Note the above definition of semaphore is slightly different from what is presented in Stallings, however its functionalities are similar.

For example: to ensure mutually exclusive access to a file, each process must adhere to the following steps when it attempts to access the shared file (assuming the semaphore's initial value is 1):

> P(sem)
> Access the file
> V(sem)

The following example implements the ticket-booking problem: an airline keeps the number of available tickets in its database. Travel agents around the world can access this database when they book tickets. To avoid inconsistency that can be introduced if two or more travel agents are allowed to access the database *simultaneously*, mutual exclusion must be enforced when an agent accesses the database.

The following program implements a solution using a binary semaphore (where the initial semaphore value is 1). The semaphore routines used in the program are defined in the following header file `pv.h`. The implementation of these routines using Unix semaphore system calls can be found in program `pv.c`, which is discussed later in this lecture.


```
/*
 *   pv.h    p & v operation
*/

int getsem(int key, int semval);  // get a
semaphore id

void rmsem(int semid);    // remove the semaphore

void p(int semid);

void v(int semid);
```

```
/* bt.c */

#include <stdio.h>
#include <unistd.h>
#include "pv.h"

#define  key   0x0000aaaa

int main()
{
    int s;
    int num_tickets;
    FILE *fp;

    /* get the id of the binary semaphore */
    s = getsem(key, 1);

    p(s);

    // check the number of tickets available
    fp = fopen("ticket_db", "r");
    fscanf(fp, "%d", &num_tickets);
    fclose(fp);
    printf("Proc %d: # of tickets is %d\n",
                getpid(), num_tickets);

    // book a ticket
    num_tickets = num_tickets - 1;

    // update the ticket database
    fp = fopen("ticket_db", "w");
    fprintf(fp, "%d", num_tickets);
    fclose(fp);
    printf("Proc %d: # of tickets is %d\n",
                getpid(), num_tickets);

    v(s);
}
```

```
$ gcc -o bt bt.c pv.c
$ cat ticket_db
100

$ bt & bt & bt & bt & bt & bt
[1] 15538
Proc 15538: # of tickets is 100
[2] 15539
[3] 15540
[4] 15541
[5] 15542
Proc 15538: # of tickets is 99
Proc 15539: # of tickets is 99
Proc 15539: # of tickets is 98
Proc 15540: # of tickets is 98
Proc 15540: # of tickets is 97
Proc 15541: # of tickets is 97
Proc 15541: # of tickets is 96
Proc 15542: # of tickets is 96
Proc 15542: # of tickets is 95
Proc 15543: # of tickets is 95
Proc 15543: # of tickets is 94
[1]   Done                    bt
[2]   Done                    bt
[3]   Done                    bt
[4]-  Done                    bt
[5]+  Done                    bt
$
```


The above example shows that at any time only ***one*** process can access the file (mutual exclusion).

Semaphores can also be used to implement synchronisations between multiple processes.

10.Semaphore Interface

The classical semaphores are easy to understand and simple to use. However, the System V's implementation of semaphores is a lot more complex. This is because the semaphore IPC implements not a single semaphore, but a set of semaphores. The operations on the semaphores are also made more general.

Under System V's implementation, semaphore operations involve the following steps:

Create a semaphore object

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int perflags);
```

Here `nsems` is the number of semaphores in the semaphore set. `perflags` is the access permission Ored with `IPC_CREAT` and `IPC_EXCL`.

If there is no existing semaphore set associated with the given key, a semaphore set object is created (if `IPC_CREAT` flag is set) and the system call returns its id. Otherwise, the id of the existing semaphore set with the same key is returned.

Note that this system call does not set the initial values to the semaphores in the semaphore set. These initial values must be set with `semctl` call.

Example: create a semaphore set with a single semaphore:

```
semid = semget(key, 1, 0600 | O_CREAT);
```

Set the semaphore values

```
#include <sys/sem.h>
int semctl(int semid,  int sem_num, int command,
                union semun ctl_arg);
```

This system call is used to do numerous things (e.g., change the permission of the semaphore set, remove the semaphore set). Here we are only interested in setting initial semaphore values.

The semaphore value must be included in the integer field of the following union data structure:

```
union semun {
     int val;
     struct semid_ds *buf;
     unsigned short *array;
} arg;
arg.val = <the initial value for a semaphore>;
```

Example: binary semaphore (initial value is 1)


```
arg.val = 1;
semctl(semid, 0, SETVAL, arg);
```

The second argument, `0`, is the index of the first semaphore in the semaphore set.

Semaphore operations


```
#include <sys/sem.h>
int semop(int semid, struct sembuf *op_array,
                 size_t num_ops);
```

The above function is used to perform atomic operations on several semaphores. The semaphore indexes and their corresponding operations are defined in `op_array`, which is an array of the following type:

```
struct  sembuf {
  unsigned short sem_num; // semaphore index
  short sem_op;         // the type of operation
  short sem_flg;       // flags: IPC_NOWAIT
                        // and SEM_UNDO
}
```

The last argument is the number of semaphores involved (the length of the array `op_array`).

The type of operation depends on the value in `sem_op`:

sem_op<0: similar to P operation:

```
    if (semval>=|sem_op|)
        semval = semval-1;
        return 0;
    else if (sem_flg & IPC_NOWAIT)
        return –1 ;
    wait until semval >= |sem_op|;
    semval = semval –1;
    return 0;
```

sem_op>0: similar to V operation:

```
    semval = semval +1;
    if there are processes waiting for this semaphore,
        wake them up
    return 0;
```


sem_op=0:


```
    if (semval != 0 && (sem_flg & IPC_NOWAIT))
        return –1;
    else
        wait for semval == 0;
        return 0;
```


Example: p(semid)

```
struct sembuf{
    unsigned short sem_num;
    short  sem_op;
    short  sem_flg;
} op;
op.sem_num = 0;    // index of the semaphore
op.sem_op =  -1;
op.sem_flg = SEM_UNDO;
semop(semid, &op, 1);  // only one semaphore
```

Example: v(semid)

```
op.sem_num = 0;
op.sem_op = 1;
op.sem_flg = SEM_UNDO;
semop(semid, &op, 1);
```

```
/*
 *   pv.h    p & v operation
*/

int getsem(int key, int semval);

void rmsem(int semid);

void p(int semid); // SEM_UNDO version

void v(int semid); // SEM_UNDO version

void p0(int semid); // no SEM_UNDO version

void v0(int semid); // no SEM_UNDO version
```



```
/*
 *  pv.c    p & v operations
 */

#include <stdio.h>
#include <errno.h>
#include <sys/sem.h>

/* get the semaphore id */
int getsem(int key, int semval)
{
     int semid;
     union semun {
         int val;
         struct semid_ds *buf;
         unsigned short int *array;
     } arg;

     /* create the semaphore if it doesn't exist, otherwise return its
id */
     semid = semget((key_t) key, 1, 0666 | IPC_CREAT | IPC_EXCL);
     if (semid<0 && errno==EEXIST) { // get the id
         if ((semid = semget((key_t) key, 1, 0666)) != -1) return semid;
         perror("semget"); exit(1);
     }

     if (semid<0) {
         perror("cannot get the semaphore id"); exit(1);
     }
```

```
     /* set the semaphore value */
    arg.val = semval;
    if (semctl(semid, 0, SETVAL, arg)<0) {
        perror("cann't set the semaphore value"); exit(2);
    }

    return semid;
}

void rmsem(int semid)
{
    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short int *array;
    } arg;

    if (semctl(semid, 0, IPC_RMID, arg)<0){
        perror("semctl IPC_RMID");
        exit(1);
    }
}

/* p operation - SEM_UNDO version */
void p(int semid)
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op  = -1;
    sb.sem_flg = SEM_UNDO;

    if (semop(semid, &sb, 1)<0) {
        perror("semop in p"); exit(1);
    }
}

/* v operation - SEM_UNDO version */
void v(int semid)
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op  = 1;
    sb.sem_flg = SEM_UNDO;

    if (semop(semid, &sb, 1)<0) {
        perror("semop in v"); exit(1);
    }
}

/* p operation - no SEM_UNDO version */
void p0(int semid)
{
    struct sembuf sb;
```

```
     sb.sem_num = 0;
     sb.sem_op  = -1;
     sb.sem_flg = 0;

     if (semop(semid, &sb, 1)<0) {
         perror("semop in p"); exit(1);
     }
}

/* v operation - no SEM_UNDO version */
void v0(int semid)
{
     struct sembuf sb;

     sb.sem_num = 0;
     sb.sem_op  = 1;
     sb.sem_flg = 0;

     if (semop(semid, &sb, 1)<0) {
         perror("semop in v"); exit(1);
     }
}
```


When a process terminates after a semaphore operation, it could prevent the other processes from accessing the shared resource. For example, if a ticket booking process dies abnormally after executing a P operation but before it executes a V operation on a semaphore, the expected V operation will never be performed. This will prevent other processes from accessing the ticket database.

To prevent this problem from happening, we set the flag `SEM_UNDO` in the functions p and v above. When the semaphore value is incremented or decremented, the process keeps an adjustment value for that semaphore. This adjustment value is added to the semaphore when the process terminates. This has the effect of “undoing” the last semaphore operation.

We have also provided a version of p and v operations without the `SEM_UNDO` flag. These two functions are named p0 and v0 respectively. In some situations, p0 and v0 are preferable over the “undo” version of p and v.

11.Synchronisation: The Producer and Consumer Problem

Semaphores are also used to synchronise processes. A classical problem in operating systems is the so-called *Producer and Consumer Problem*. The producer produces an item and places it in a buffer for the consumer to take out. The two operations must be properly synchronised, because the buffer is only large enough to hold one item. The producer cannot produce faster than the consumer can consume, otherwise, the old item would be overwritten. Similarly, the consumer cannot consume faster than the producer can produce, otherwise, the consumer would consume the same item multiple times.

We will use two semaphores: *empty* and *full* for this problem. The producer is allowed to produce only when the buffer is “empty” and it should signal to the consumer after it has placed an item in the buffer. Similarly, the consumer is allowed to consume only after the buffer is full (because the size of the buffer is 1). It should signal to the producer after it has consumed the item.

Because initially the buffer is empty, we set semaphore full to 0 and semaphore empty to 1. Below is the basic structure of the two processes.

Assume that Producer runs first. It sets the initial values for the two semaphores. Note, the processes share two semaphores as well as the buffer.

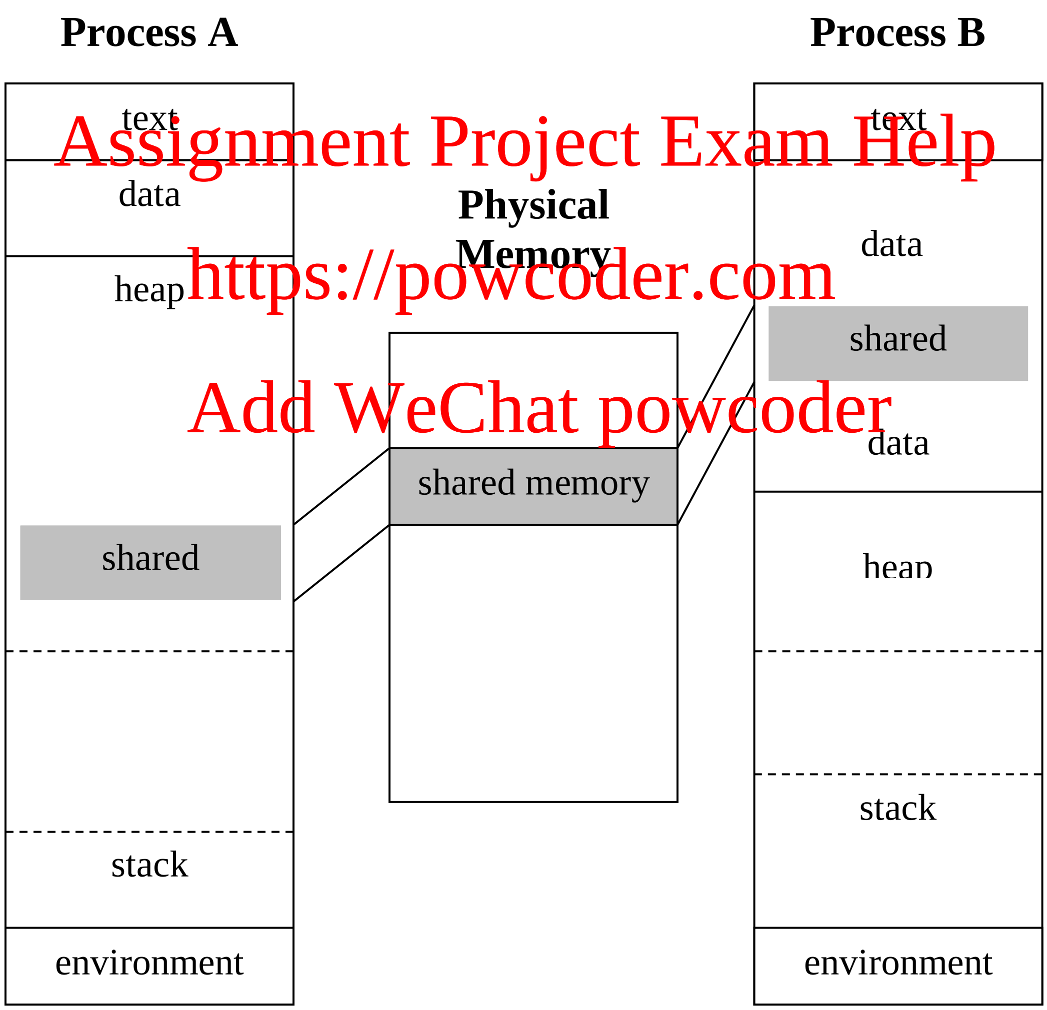
Producer	Consumer
empty = 1 full = 0 Repeat 1. produce one item **2. p(empty)** 3. place the item in buffer **4. v(full)**	repeat **1. p(full)** 2. take the item from buffer **3. v(empty)** 4. consume the item

```
/*
 *  wfile.c HX
 *          synchronising two processes wfile and rfile with semaphores
 *          and a shared file “data”. wfile is a producer and rfile is the
 *          comsumer, they repeatedly do:
 *                wfile places an item to the beginning of the file
 *                rfile takes the item  out from the file.
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include "pv.h"

#define  KEY1    0x00a00001
#define  KEY2    0x00a00002
#define  KEY3    0x00a00003

int main()
{
    int fd, i;
    int end, empty, full;
    char buf[20];

    if ((fd = open("data", O_WRONLY | O_CREAT | O_TRUNC))<0){
       perror("open file"); exit(1);
    }

    /* create semaphores if not already created, and get their ids */
    end = getsem(KEY1, 0);
    empty = getsem(KEY2, 1);
    full  = getsem(KEY3, 0);

    for (i=0; i<10; ++i) {
       /* produce one item */
       sprintf(buf, "data %d\n", i);

       p(empty); // wait until the file is empty (the item taken out)

       lseek(fd, 0, SEEK_SET);
       write(fd, buf, 20);
       fsync(fd);  // make sure the file is updated immediately
       printf("deposit: %s", buf);

       v(full);  // signal to rfile that the file is full again (item placed)
    }

    /* wait until rfile is finished */
    p0(end);

    /* remove all semaphores */
    rmsem(empty);
    rmsem(full);
    rmsem(end);
    exit(0);
}
```

```
/*
 *  rfile.c HX
 *          synchronise two processes wfile and rfile using semaphores
.*          and a shared file “data”.  wfile is a producer and rfile is the
 *          comsumer, they repeatedly do:
 *              wfile places a item to the beginning of the file
 *              rfile takes the item out from the file.
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include "pv.h"

#define  KEY1    0x00a00001
#define  KEY2    0x00a00002
#define  KEY3    0x00a00003

int main()
{
    int fd, i;
    int end, empty, full;
    char buf[20];

    if ((fd = open("data", O_RDONLY))<0) {
       perror("open file");
       exit(1);
    }

    /* create semaphores if not exist and get their ids */
    end = getsem(KEY1, 0);
    empty = getsem(KEY2, 1);
    full  = getsem(KEY3, 0);

    for (i=0; i<10; ++i) {

       p(full);   // wait until the item is placed to file

       lseek(fd, 0, SEEK_SET);
       read(fd, buf, 20);
       printf("remove: %s", buf);

       v(empty); // signal to wfile that the file is empty again
    }

    /* tell wfile that I have completed, so it can remove
       the semaphores.
       Note we must use the none undo version of the v to signal (think
       why this is necessary)
    */
    v0(end);

    exit(0);
}
```

12.Shared Memory

In Unix, processes do not share their logical address spaces. For example, on a 32-bit machine, each process has its own *exclusive* logical address space, usually ranging between 1 and 2GB. However, it is possible for the kernel to map a segment of that logical address space to a segment of the physical memory. If multiple processes map their segments of logical address spaces to the same physical memory segment, these processes would share this physical memory segment.



Three system calls are used to define the shared memory. The system call `shmget` creates the system-wide shared memory object. `shmat` attaches the shared memory object to the calling process, and `shmdt` detached the shared memory from the calling process. Another system call, `shmctl`, performs various miscellaneous operations (such as obtaining and changing the information about the shared memory, removing the shared memory from the system, and lock and unlock the shared memory) on the shared memory object.

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

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


The above system call is similar to `msgget` and `semget`. It creates a shared memory object with at least `size` number of bytes (the actual number of bytes is a multiple of the page size). The last parameter `permflags` is the access permissions Ored with `IPC_CREAT` and/or `IPC_EXCL`.


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

void * shmat(int shmid, void *addr, int flags);
```

This system call attaches the shared memory to a specific logical address in the calling process. The flags can be `SHM_RDONLY` and `SHM_RND` (boundary rounded up). If the flags include `SHM_RDONLY`, then the shared memory is read-only. Otherwise, it is both readable and writable.

The actual address of the attached memory depends on the flags:

- if `addr == NULL`, the kernel will attach the shared memory at the first available address.
- If `addr != NULL` and `SHM_RND` not in the flags, attach the shared memory at the address given by `addr`.
- If `addr != NULL` and `SHM_RND` in the flags, the given address is rounded up to the beginning of the page and the shared memory is attached at the beginning of that page .

The system call returns the starting address of the shared memory. The process can then access the shared memory using the address.

Note the address given by `addr` should not be the address of an existing variable in the data section, heap or stack, unless the variable is meant to be shared. Since memory allocation is implementation dependent, and since it is not easy to figure out the free logical address, it is preferable to set the `addr` to `NULL` and let the kernel select the logical address of the shared memory.

The following example is similar to the read and write file example (`wfile.c` and `rfile.c`) presented earlier. The difference is that in this example we use a shared memory object, rather than a file, to allow the reader (`rshm.c`) and writer (`wshm.c`) processes to exchange data. This would be considerably faster than using a file to exchange data. Since shared memory doesn't have any built-in synchronisation mechanism, we use two semaphores, empty and full, to synchronise the write and read activities.

```
/*
 *  wshm.c        HX
 *          read & write shared memory.  wshm is a producer
 *          and rshm is the comsumer, the two processesare
 *          synchronised with two semaphores.
 *          The two processes repeatedly do:
 *                wshm places a item (a random number) to the buffer
 *                      (shared memory)
 *                rshm takes the item out from the buffer
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "pv.h"

#define  BUF_FULL_KEY         0x00a00001
#define  BUF_EMPTY_KEY        0x00a00002
#define  READER_END_KEY       0x00a00003

#define  SHMKEY  0x00a00001
#define  SEED    (unsigned int)199

#include <errno.h>

int main()
{
    int i;
    int reader_end, empty, full;  // semaphores
    int shmid;
    int *buffer;  // the addr of the shared memory for one random number */
    int item;

    /* create (get) the shared memory object and attach it */
    if ((shmid = shmget(SHMKEY, sizeof(int), 0600 | IPC_CREAT))<0) {
        perror("shmget"); exit(1);
    }
    if ((buffer = (int *)shmat(shmid, NULL, 0)) == (int *)-1) {
        perror("shmat"); exit(1);
    }

    /* set seed for the (pseudo) random number generator */
    srand(SEED);

    /* create semaphores if not already created, and get their ids */
    empty      = getsem(BUF_EMPTY_KEY, 1);
    full       = getsem(BUF_FULL_KEY, 0);
    reader_end = getsem(READER_END_KEY, 0);

    for (i=0; i<10; ++i) {
       item = (int) 100.0*(rand()/(RAND_MAX+1.0)); // produce one item (0-99).
       p(empty);      // wait until the buffer is empty
       *buffer=item;  // put the item to buffer
       v(full);       // signal to rshm that the buffer is full
```

```
    }

    /* wait for reader (rshm) to end first, see comments at the end
     * of reader (rshm) */

    p0(reader_end);

    shmctl(shmid, IPC_RMID, NULL ); // remove shared memory
    rmsem(empty);                   // remove sem empty
    rmsem(full);                    // remove sem full
    rmsem(reader_end);              // remove sem rend

    exit(0);
}
```

```c
/*
 *  rshm.c        HX
 *          read and write shared memory.  wshm is a producer and rshm is the
 *          comsumer, they are synchronised using two semaphores.
 *          the two processes repeatedly do
 *                      wshm places a item to the buffer (shared memory)
 *                      rshm takes the item out from the buffer
 *          Note the buffer can only hold one item each time.
 */

#include <stdio.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/ipc.h>
#include "pv.h"

#define  BUF_FULL_KEY        0x00a00001
#define  BUF_EMPTY_KEY       0x00a00002
#define  READER_END_KEY      0x00a00003
#define  SHMKEY  0x00a00001

int main()
{
    int fd, i;
    int reader_end, empty, full; // semaphores
    int shmid;    // shared memory id
    int *buffer;  // point to the shared buffer
    int item;

    /* get (or create) the shared memmory and attach it */
    if ((shmid = shmget(SHMKEY, sizeof(int), 0600 | IPC_CREAT))<0) {
        perror("shmget"); exit(1);
    }
    if ((buffer = (int *)shmat(shmid, NULL, 0))==(int *)-1){
        perror("shmat"); exit(1);
    }

    /* create semaphores if not exist and get their ids */
    empty = getsem(BUF_EMPTY_KEY, 1);
    full  = getsem(BUF_FULL_KEY, 0);
    reader_end = getsem(READER_END_KEY, 0);

    for (i=0; i<10; ++i) {
       p(full);      // wait until the item is placed in buffer
       item=*buffer; // take the item out of the buffer
       v(empty); // signal to wshm that the buffer is empty again
       printf("The item is %d\n", item); // consume the item
    }

    /* tell the writer we have ended. Note v0, rather than v, is used.
     * otherwise, reader_end will be adjusted when the process exits
     * which effectivelly nulls the v operation and hence making
     * the writer to wait forever */

    v0(reader_end);
    exit(0);
}
```