Examples: Using semaphores and shared memory

See Code disclaimer information for information pertaining to code examples.

The following two examples illustrate programs that support the client/server model.

Server program

This program acts as a server to the client program (see Client program). The buffer is a shared memory segment. The process synchronization is done using semaphores.

Use the Create C Module (CRTCMOD) and the Create Program (CRTPGM) commands to create this program.

Call this program with no parameters before calling the client program.

```
/*********************************************************************/
/*********************************************************************/
/*                                                                   */
/* FUNCTION:  This program acts as a server to the client program.   */
/*                                                                   */
/* LANGUAGE:  ILE C for OS/400                                       */
/*                                                                   */
/* APIs USED: semctl(), semget(), semop(),                           */
/*            shmat(), shmctl(), shmdt(), shmget()                   */
/*            ftok()                                                 */
/*                                                                   */
/*********************************************************************/
/*********************************************************************/

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

#define SEMKEYPATH "/dev/null"  /* Path used on ftok for semget key  */
#define SEMKEYID 1              /* Id used on ftok for semget key    */
#define SHMKEYPATH "/dev/null"  /* Path used on ftok for shmget key  */
#define SHMKEYID 1              /* Id used on ftok for shmget key    */

#define NUMSEMS 2               /* Num of sems in created sem set    */
#define SIZEOFSHMSEG 50         /* Size of the shared mem segment    */

#define NUMMSG 2                /* Server only doing two "receives"
                                   on shm segment                    */

int main(int argc, char *argv[])
{
   int rc, semid, shmid, i;
   key_t semkey, shmkey;
   void *shm_address;
   struct sembuf operations[2];
   struct shmid_ds shmid_struct;
   short  sarray[NUMSEMS];
```

```
/* Generate an IPC key for the semaphore set and the shared      */
/* memory segment.  Typically, an application specific path and  */
/* id would be used to generate the IPC key.                     */
semkey = ftok(SEMKEYPATH,SEMKEYID);
if ( semkey == (key_t)-1 )
  {
    printf("main: ftok() for sem failed\n");
    return -1;
  }
shmkey = ftok(SHMKEYPATH,SHMKEYID);
if ( shmkey == (key_t)-1 )
  {
    printf("main: ftok() for shm failed\n");
    return -1;
  }

/* Create a semaphore set using the IPC key.  The number of      */
/* semaphores in the set is two.  If a semaphore set already     */
/* exists for the key, return an error. The specified permissions*/
/* give everyone read/write access to the semaphore set.         */

semid = semget( semkey, NUMSEMS, 0666 | IPC_CREAT | IPC_EXCL );
if ( semid == -1 )
  {
    printf("main: semget() failed\n");
    return -1;
  }

/* Initialize the first semaphore in the set to 0 and the        */
/* second semaphore in the set to 0.                             */
/*                                                               */
/* The first semaphore in the sem set means:                     */
/*       '1' --  The shared memory segment is being used.        */
/*       '0' --  The shared memory segment is freed.             */
/* The second semaphore in the sem set means:                    */
/*       '1' --  The shared memory segment has been changed by   */
/*               the client.                                     */
/*       '0' --  The shared memory segment has not been          */
/*               changed by the client.                          */

sarray[0] = 0;
sarray[1] = 0;


/* The '1' on this command is a no-op, because the SETALL command*/
/* is used.                                                      */
rc = semctl( semid, 1, SETALL, sarray);
if(rc == -1)
  {
    printf("main: semctl() initialization failed\n");
    return -1;
  }

/* Create a shared memory segment using the IPC key.  The        */
/* size of the segment is a constant.  The specified permissions */
/* give everyone read/write access to the shared memory segment. */
/* If a shared memory segment already exists for this key,       */
/* return an error.                                              */
shmid = shmget(shmkey, SIZEOFSHMSEG, 0666 | IPC_CREAT | IPC_EXCL);
if (shmid == -1)
  {
    printf("main: shmget() failed\n");
```

```
    return -1;
  }

/* Attach the shared memory segment to the server process.       */
shm_address = shmat(shmid, NULL, 0);
if ( shm_address==NULL )
  {
    printf("main: shmat() failed\n");
    return -1;
  }
printf("Ready for client jobs\n");

/* Loop only a specified number of times for this example.       */
for (i=0; i < NUMMSG; i++)
  {
    /* Set the structure passed into the semop() to first wait   */
    /* for the second semval to equal 1, then decrement it to    */
    /* allow the next signal that the client writes to it.       */
    /* Next, set the first semaphore to equal 1, which means     */
    /* that the shared memory segment is busy.                   */
    operations[0].sem_num = 1;
                          /* Operate on the second sem     */
    operations[0].sem_op = -1;
                          /* Decrement the semval by one   */
    operations[0].sem_flg = 0;
                          /* Allow a wait to occur         */

    operations[1].sem_num = 0;
                          /* Operate on the first sem      */
    operations[1].sem_op =  1;
                          /* Increment the semval by 1     */
    operations[1].sem_flg = IPC_NOWAIT;
                               /* Do not allow to wait    */

    rc = semop( semid, operations, 2 );
    if (rc == -1)
      {
        printf("main: semop() failed\n");
        return -1;
      }

    /* Print the shared memory contents.                         */
    printf("Server Received : \"%s\"\n", (char *) shm_address);

    /* Signal the first semaphore to free the shared memory.     */
    operations[0].sem_num = 0;
    operations[0].sem_op  = -1;
    operations[0].sem_flg = IPC_NOWAIT;

    rc = semop( semid, operations, 1 );
    if (rc == -1)
      {
        printf("main: semop() failed\n");
        return -1;
    }

  }  /* End of FOR LOOP */

/* Clean up the environment by removing the semid structure,     */
/* detaching the shared memory segment, and then performing      */
/* the delete on the shared memory segment ID.                   */
```

```
    rc = semctl( semid, 1, IPC_RMID );
    if (rc==-1)
      {
        printf("main: semctl() remove id failed\n");
        return -1;
      }
    rc = shmdt(shm_address);
    if (rc==-1)
      {
        printf("main: shmdt() failed\n");
        return -1;
      }
    rc = shmctl(shmid, IPC_RMID, &shmid_struct);
    if (rc==-1)
      {
        printf("main: shmctl() failed\n");
        return -1;
      }
return 0;
}
```

Client program

This program acts as a client to the server program (see Server program). The program is run after the message Ready for client jobs appears from the server program.

Use the CRTCMOD and CRTPGM commands to create this program.

Call this program with no parameters after calling the server program.

```
/*********************************************************************/
/*********************************************************************/
/*                                                                   */
/* FUNCTION:  This program acts as a client to the server program.   */
/*                                                                   */
/* LANGUAGE:  ILE C for OS/400                                       */
/*                                                                   */
/* APIs USED: semget(), semop(),                                     */
/*            shmget(), shmat(), shmdt()                             */
/*            ftok()                                                 */
/*                                                                   */
/*********************************************************************/
/*********************************************************************/

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

#define SEMKEYPATH "/dev/null"  /* Path used on ftok for semget key  */
#define SEMKEYID 1              /* Id used on ftok for semget key    */
#define SHMKEYPATH "/dev/null"  /* Path used on ftok for shmget key  */
#define SHMKEYID 1              /* Id used on ftok for shmget key    */

#define NUMSEMS 2
#define SIZEOFSHMSEG 50

int main(int argc, char *argv[])
```

```
{
   struct sembuf operations[2];
   void        *shm_address;
   int semid, shmid, rc;
   key_t semkey, shmkey;

   /* Generate an IPC key for the semaphore set and the shared      */
   /* memory segment.  Typically, an application specific path and  */
   /* id would be used to generate the IPC key.                     */
   semkey = ftok(SEMKEYPATH,SEMKEYID);
   if ( semkey == (key_t)-1 )
     {
      printf("main: ftok() for sem failed\n");
      return -1;
     }
   shmkey = ftok(SHMKEYPATH,SHMKEYID);
   if ( shmkey == (key_t)-1 )
     {
      printf("main: ftok() for shm failed\n");
      return -1;
     }

   /* Get the already created semaphore ID associated with key.     */
   /* If the semaphore set does not exist, then it will not be      */
   /* created, and an error will occur.                             */
   semid = semget( semkey, NUMSEMS, 0666);
   if ( semid == -1 )
     {
      printf("main: semget() failed\n");
      return -1;
     }

   /* Get the already created shared memory ID associated with key. */
   /* If the shared memory ID does not exist, then it will not be   */
   /* created, and an error will occur.                             */

   shmid = shmget(shmkey, SIZEOFSHMSEG, 0666);
   if (shmid == -1)
     {
      printf("main: shmget() failed\n");
      return -1;
     }

   /* Attach the shared memory segment to the client process.       */
   shm_address = shmat(shmid, NULL, 0);
   if ( shm_address==NULL )
     {
      printf("main: shmat() failed\n");
      return -1;
     }

   /* First, check to see if the first semaphore is a zero.  If it  */
   /* is not, it is busy right now.  The semop() command will wait  */
   /* for the semaphore to reach zero before running the semop().   */
   /* When it is zero, increment the first semaphore to show that   */
   /* the shared memory segment is busy.                            */
   operations[0].sem_num = 0;
                           /* Operate on the first sem      */
   operations[0].sem_op =  0;
                           /* Wait for the value to be=0    */
   operations[0].sem_flg = 0;
                           /* Allow a wait to occur         */
```

```
    operations[1].sem_num = 0;
                            /* Operate on the first sem      */
    operations[1].sem_op =  1;
                            /* Increment the semval by one   */
    operations[1].sem_flg = 0;
                            /* Allow a wait to occur         */

    rc = semop( semid, operations, 2 );
    if (rc == -1)
      {
        printf("main: semop() failed\n");
        return -1;
      }

    strcpy((char *) shm_address, "Hello from Client");

    /* Release the shared memory segment by decrementing the in-use  */
    /* semaphore (the first one).  Increment the second semaphore to */
    /* show that the client is finished with it.                     */
    operations[0].sem_num = 0;
                            /* Operate on the first sem      */
    operations[0].sem_op =  -1;
                            /* Decrement the semval by one   */
    operations[0].sem_flg = 0;
                            /* Allow a wait to occur         */

    operations[1].sem_num = 1;
                            /* Operate on the second sem     */
    operations[1].sem_op =  1;
                            /* Increment the semval by one   */
    operations[1].sem_flg = 0;
                            /* Allow a wait to occur         */

    rc = semop( semid, operations, 2 );
    if (rc == -1)
      {
        printf("main: semop() failed\n");
        return -1;
      }

    /* Detach the shared memory segment from the current process.    */
    rc = shmdt(shm_address);
    if (rc==-1)
      {
        printf("main: shmdt() failed\n");
        return -1;
      }

return 0;
}
```

IPC: POSIX Message queues

- mq_open(3)
- mq_close(3)
- mq_unlink(3)
- mq_send(3)

mq_receive(3)
mq_getattr(3)
mq_setattr(3)

Notes:
Message queues are an handy way to get IPC when working with unrelated processes that run completely without synchronism. A message queue is a shared "box" where processes can drop and withdraw messages independently. Besides the content, each message is given a "priority". Message queues first appeared in System V. POSIX standardized this IPC mechanism with the 1003.1b ("realtime") standard in 1993, but the resulting standard is describing a slightly different behavior wrt the former historical implementation. This is about the POSIX interface.
Message queues objects are referenced by a unique POSIX object name in the system as usual. The opening function (mq_open()) is the only one to use that name, while an internal reference returned by it will be used to reference the queue for the remainder of the program.
Messages are sent to the queue with the mq_send() (blocking by default). They can be any kind of information, nothing is dictated on the structure of a message. On sending, the message is to be accompanied by a priority. This priority is used for delivering messages: every time a process request for a receive (invoking the mq_receive() function, also blocking by default), the oldest message with the highest priority will be given. This differs from the SysV API, in which the user explicitly request for a specific priority, causing this priority being frequently used as recipient address.
Several features of the message queue can be set on creation or, in part, on a later time with the mq_setattr() function. Between them, the maximum size of a message is worth particular attention: each process invoking a receive won't be serviced, and EMSGSIZE will be set in errno if the length of the buffer for the message is shorter than this maximum size.

Summary:

message queues are available with the System V API and the POSIX API. This is about the latter
what to #include: mqueue.h (sys/stat.h for using permission macros while creating queues)
types:

1. **mqd_t**: a message queue descriptor.
2. **struct mq_attr**: a message queue attributes structure, defined as follows:

```
struct mq_attr {
    long int mq_flags;    /* Message queue flags.  */
    long int mq_maxmsg;   /* Maximum number of messages.  */
    long int mq_msgsize;  /* Maximum message size.  */
    long int mq_curmsgs;  /* Number of messages currently queued.  */
}
```

functions:

1. mqd_t **mq_open**(const char *name, int flags, ... [mode_t mode, struct mq_attr *mq_attr])
   opens the queue referenced by name for access, where flags can request: O_RDONLY, O_WRONLY, O_RDWR for access permission, and O_CREAT, O_EXCL or O_NONBLOCK as intuitive, respectively, for create queue, fail create if already existing, and non-blocking behavior. If O_CREAT is specified, two more fields are requested: mode expliciting object permissions (S_IRWXU, S_IRWXG etc you

can inspect in chmod(2); include sys/stat.h for these), and mq_attr expliciting the characteristics wanted for the queue. The latter can be set to NULL for using defaults, but mind the message size problem discussed in Notes above.
Returns the message descriptor, or (mqd_t)-1 on error.

2. int **mq_close**(mdq_t mqdes)
closes the queue described by mqdes. Returns 0 for ok, otherwise -1.
3. int **mq_unlink**(const char *name)
like unlink(2), but with the posix object referenced by name.
Returns 0, or -1 on error.
4. int **mq_send**(mqd_t mqdes, const char *msgbuf, size_t len, unsigned int prio)
sends len bytes from msgbuf to the queue mqdes, associating a prio priority.
Return 0 on succes, -1 otherwise.
5. ssize_t **mq_receive**(mqd_t mqdes, char *buf, size_t len, unsigned *prio)
takes the oldest message with the highest priority from mqdes into buf. The len field determines an important effect described in Notes above. prio, if not NULL, is filled with the priority of the given message.
Returns the length of the message received, or -1 on error.
6. int **mq_getattr**(mqd_t mqdes, struct mq_attr *mq_attr)
provides the attributes associated to mqdes by filling at mq_attr with the structure shown above.
Returns 0, or -1 on error.
7. int **mq_setattr**(mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr *omqstat)
for mqdes, sets the attributes carried by mqstat. In fact, only the mq_flags member of this structure applies (for setting O_NONBLOCK): the other ones are simply ignored and can be set just when the queue is created. The omqstat is commonly NULL, or it will be filled with the previous attributes and the current queue status.
Returns 0, or -1 on error and the queue won't be touched.

further notes on persistence: message queues both in the POSIX and System V implementation features kernel-level persistence
further notes on System V message queues: while the semantic is quite the same as for POSIX, System V message queues use a different namespace for functions for opening, creating, sending, receiving etc. See the msgget, msgctl, msgsnd, msgrcv man pages for more about this older API.

Examples:
1) Handling creation/opening of message queues, sending/receiving of messages and the essence of queue attributes. Try running several times the message producer before running the consumer. Try running the consumer more times than the producer. Try looping the producer until the queue fills. Try running the consumer after decreasing MAX_MSG_LEN under what's displayed by the queue attributes.

```
/*
 *  dropone.c
 *
 *  drops a message into a #defined queue, creating it if user
 *  requested. The message is associated a priority still user
 *  defined
 *
 *
 *  Created by Mij <mij@bitchx.it> on 07/08/05.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 *
```

```
 */

#include <stdio.h>
/* mq_* functions */
#include <mqueue.h>
#include <sys/stat.h>
/* exit() */
#include <stdlib.h>
/* getopt() */
#include <unistd.h>
/* ctime() and time() */
#include <time.h>
/* strlen() */
#include <string.h>


/* name of the POSIX object referencing the queue */
#define MSGQOBJ_NAME    "/myqueue123"
/* max length of a message (just for this process) */
#define MAX_MSG_LEN     70


int main(int argc, char *argv[]) {
    mqd_t msgq_id;
    unsigned int msgprio = 0;
    pid_t my_pid = getpid();
    char msgcontent[MAX_MSG_LEN];
    int create_queue = 0;
    char ch;            /* for getopt() */
    time_t currtime;


    /* accepting "-q" for "create queue", requesting "-p prio" for message priority */
    while ((ch = getopt(argc, argv, "qi:")) != -1) {
        switch (ch) {
            case 'q':   /* create the queue */
                create_queue = 1;
                break;
            case 'p':   /* specify client id */
                msgprio = (unsigned int)strtol(optarg, (char **)NULL, 10);
                printf("I (%d) will use priority %d\n", my_pid, msgprio);
                break;
            default:
                printf("Usage: %s [-q] -p msg_prio\n", argv[0]);
                exit(1);
        }
    }

    /* forcing specification of "-i" argument */
    if (msgprio == 0) {
        printf("Usage: %s [-q] -p msg_prio\n", argv[0]);
        exit(1);
    }

    /* opening the queue        --  mq_open() */
    if (create_queue) {
        /* mq_open() for creating a new queue (using default attributes) */
        msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRWXU | S_IRWXG, NULL);
    } else {
        /* mq_open() for opening an existing queue */
        msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR);
    }
```

```
    if (msgq_id == (mqd_t)-1) {
        perror("In mq_open()");
        exit(1);
    }

    /* producing the message */
    currtime = time(NULL);
    snprintf(msgcontent, MAX_MSG_LEN, "Hello from process %u (at %s).", my_pid, ctime(&currtime));

    /* sending the message      --  mq_send() */
    mq_send(msgq_id, msgcontent, strlen(msgcontent)+1, msgprio);

    /* closing the queue        -- mq_close() */
    mq_close(msgq_id);


    return 0;
}



/*
 *  takeone.c
 *
 *  simply request a message from a queue, and displays queue
 *  attributes.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 07/08/05.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 *
 */

#include <stdio.h>
/* mq_* functions */
#include <mqueue.h>
/* exit() */
#include <stdlib.h>
/* getopt() */
#include <unistd.h>
/* ctime() and time() */
#include <time.h>
/* strlen() */
#include <string.h>


/* name of the POSIX object referencing the queue */
#define MSGQOBJ_NAME    "/myqueue123"
/* max length of a message (just for this process) */
#define MAX_MSG_LEN     10000


int main(int argc, char *argv[]) {
    mqd_t msgq_id;
    char msgcontent[MAX_MSG_LEN];
    int msgsz;
    unsigned int sender;
    struct mq_attr msgq_attr;


    /* opening the queue        --  mq_open() */
    msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR);
    if (msgq_id == (mqd_t)-1) {
```

```
        perror("In mq_open()");
        exit(1);
    }

    /* getting the attributes from the queue        --  mq_getattr() */
    mq_getattr(msgq_id, &msgq_attr);
    printf("Queue \"%s\":\n\t- stores at most %ld messages\n\t- large at most %ld bytes each\n\t- currently
holds %ld messages\n", MSGQOBJ_NAME, msgq_attr.mq_maxmsg, msgq_attr.mq_msgsize,
msgq_attr.mq_curmsgs);

    /* getting a message */
    msgsz = mq_receive(msgq_id, msgcontent, MAX_MSG_LEN, &sender);
    if (msgsz == -1) {
        perror("In mq_receive()");
        exit(1);
    }
    printf("Received message (%d bytes) from %d: %s\n", msgsz, sender, msgcontent);

    /* closing the queue    --  mq_close() */
    mq_close(msgq_id);


    return 0;
}
```

You can download the original ascii source files with these links: dropone.c and takeone.c.
Compile with:
gcc --pedantic -Wall -o dropone dropone.c
gcc --pedantic -Wall -o takeone takeone.c
(some systems wrap these functions into a posix realtime library: add -lrt for them -- e.g. Linux does)
Run as follows:
./dropone -p some_positive_integer (add -q the first time)
./takeone
and try some of the tests described above the sources.

IPC: POSIX Semaphores

sem_init(2)
sem_open(2)
sem_close(2)
sem_post(2)
sem_wait(2)
sem_trywait(2)

Notes:
In the real world today, POSIX semaphores don't still get full support from OSes. Some features are not yet implemented. The Web is nonetheless poor of documentation about it, so the goal here is to leave some concepts; you will map into this or the other implementation in the need. We'll hopefully enjoy better support during the next years.
Semaphores are a mechanism for controlling access to shared resources. In their original form, they are associated to a single resource, and contain a binary value: {free, busy}. If the resource is shared between more threads (or procs), each thread waits for the resource semaphore to switch free. At this time, it sets the semaphore to busy and enters the critical

region, accessing the resource. Once the work is done with the resource, the process releases it back setting the semaphore to free. The whole thing works because the semaphore value is set directly by the kernel on request, so it's not subject to process interleaving.
Please mind that this mechanism doesn't provide mutex access to a resource. It just provides support for processes to carry out the job themselves, so it's completely up to the processes to take the semaphore up to date and access the resource in respect of it.
Semaphores can be easily extended to support several accesses to a resource. If a resource can be accessed concurrently by *n* threads, the semaphore simply needs to hold an integer value instead of a Boolean one. This integer would be initialized to *n*, and each time it's positive value a process decrements it by 1 and gets 1 slot in the resource. When the value reaches 0, no more slots are available and candidate threads have to wait for the resource to get available ("free" is no more appropriate in this case). In this case the semaphore is told to be *locked*.
This is the concept behind semaphores. Dijkstra has been inspired by actual train semaphores when he designed them in '60s: this justify their being very intuitive.
There are 2 relevant APIs for such systems today:

System V semaphores
POSIX semaphores

The former implementations being inherited by the System V, and the latter being standardized by POSIX 1003.1b (year 1993, the first one with real time extensions). SysV semaphores are older and more popular. POSIX ones are newer, cleaner, easier, and lighter against the machine.
Just like pipes, semaphores can be *named* or *unnamed*. Unnamed semaphores can only be shared by related processes. The POSIX API defines both these personalities. They just differ in creation and opening.

Summary:

always keep an eye on platform-specific implementations. It is common to deal with partial implementations. Make your way with the help of *man*.
what to #include: semaphore.h
types:

1. **sem_t**: a semaphore descriptor. Commonly passed by reference.

functions for activating *unnamed* semaphores:

1. int **sem_init**(sem_t *sem, int shared, unsigned int value)
   the area pointed by sem is initialized with a new semaphore object. shared tells if the semaphore is local (0) or shared between several processes (non-0). This isn't vastly supported.
   value is the number the semaphore is initialized with.
   Returns -1 if unsuccessful.
2. int **sem_destroy**(sem_t *sem)
   deallocates memory associated with the semaphore pointed by sem.
   Returns 0 if successful, -1 otherwise.

functions for activating *named* semaphores:

3. sem_t **sem_open**(const char *name, int flags, ...)
   ", ..." is zero or one occurrence of "mode_t mode, unsigned int value" (see below).
   opens a named semaphore. name specify its location in the filesystem hierarchy.
   flags can be zero, or O_CREAT (possibly associated with O_EXCL). Non-zero means

creating the semaphore if it doesn't already exist. O_CREAT requires two further parameters: mode specifying the permission set of the semaphore in the filesystem, value being the same as in sem_destroy(). O_CREAT causes an exclusive behaviour if it appears with O_EXCL: a failure is returned if a semaphore name already exists in this case. Further values have been designed, while rarely implemented: O_TRUNC for truncating if such semaphore already exists, and O_NONBLOCK for non-blocking mode.
sem_open returns the address of the semaphore if successful, SEM_FAILED otherwise.

4. int **sem_close**(sem_t *sem)
   closes the semaphore pointed by sem. Returns 0 on success, -1 on failure.
5. int **sem_unlink**(const char name *name)
   removes the semaphore name from the filesystem. Returns 0 on success, -1 on failure.

functions for working with open semaphores:

6. int **sem_wait**(sem_t *sem)
   waits until the semaphore sem is non-locked, then locks it and returns to the caller. Even without getting the semaphore unlocked, sem_wait may be interrupted with the occurrence of a signal.
   returns 0 on success, -1 on failure. If -1, sem is not modified by the function.
7. int **sem_trywait**(sem_t *sem)
   tries to lock sem the same way sem_wait does, but doesn't hang if the semaphore is locked.
   Returns 0 on success (got semaphore), EAGAIN if the semaphore was locked.
8. int **sem_post**(sem_t *sem)
   unlocks sem. Of course, this has to be atomic and non-reentrant. After this operation, some thread between those waiting for the semaphore to get free. Otherwise, the semaphore value simply steps up by 1.
   Returns 0 on success, -1 on failure.
9. int **sem_getvalue**(sem_t *sem, int *restrict sval)
   compiles sval with the current value of sem. sem may possibly change its value before sem_getvalue returns to the caller.
   Returns 0 on success, -1 on failure.

```
/*
 *  semaphore.c
 *
 *
 *  Created by Mij <mij@bitchx.it> on 12/03/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unixprg/
 *
 */


#define _POSIX_SOURCE
#include <stdio.h>
/* sleep() */
#include <errno.h>
#include <unistd.h>
/* abort() and random stuff */
#include <stdlib.h>
/* time() */
#include <time.h>
#include <signal.h>
```

```
#include <pthread.h>
#include <semaphore.h>


/* this skips program termination when receiving signals */
void signal_handler(int type);

/*
 * thread A is synchronous. When it needs to enter its
 * critical section, it can't do anything other than waiting.
 */
void *thread_a(void *);

/*
 * thread B is asynchronous. When it tries to enter its
 * critical section, it switches back to other tasks if
 * it hasn't this availability.
 */
void *thread_b(void *);


/* the semaphore */
sem_t mysem;

int main(int argc, char *argv[])
{
    pthread_t mytr_a, mytr_b;
    int ret;


    srand(time(NULL));
    signal(SIGHUP, signal_handler);
    signal(SIGUSR1, signal_handler);

    /*
     * creating the unnamed, local semaphore, and initialize it with
     * value 1 (max concurrency 1)
     */
    ret = sem_init(&mysem, 0, 1);
    if (ret != 0) {
        /* error. errno has been set */
        perror("Unable to initialize the semaphore");
        abort();
    }

    /* creating the first thread (A) */
    ret = pthread_create(&mytr_a, NULL, thread_a, NULL);
    if (ret != 0) {
        perror("Unable to create thread");
        abort();
    }

    /* creating the second thread (B) */
    ret = pthread_create(&mytr_b, NULL, thread_b, NULL);
    if (ret != 0) {
        perror("Unable to create thread");
        abort();
    }

    /* waiting for thread_a to finish */
    ret = pthread_join(mytr_a, (void *)NULL);
    if (ret != 0) {
```

```
        perror("Error in pthread_join");
        abort();
    }

    /* waiting for thread_b to finish */
    ret = pthread_join(mytr_b, NULL);
    if (ret != 0) {
        perror("Error in pthread_join");
        abort();
    }

    return 0;
}


void *thread_a(void *x)
{
    unsigned int i, num;
    int ret;


    printf(" -- thread A -- starting\n");
    num = ((unsigned int)rand() % 40);

    /* this does (do_normal_stuff, do_critical_stuff) n times */
    for (i = 0; i < num; i++) {
        /* do normal stuff */
        sleep(3 + (rand() % 5));

        /* need to enter critical section */
        printf(" -- thread A -- waiting to enter critical section\n");
        /* looping until the lock is acquired */
        do {
            ret = sem_wait(&mysem);
            if (ret != 0) {
                /* the lock wasn't acquired */
                if (errno != EINVAL) {
                    perror(" -- thread A -- Error in sem_wait. terminating -> ");
                    pthread_exit(NULL);
                } else {
                    /* sem_wait() has been interrupted by a signal: looping again */
                    printf(" -- thread A -- sem_wait interrupted. Trying again for the lock...\n");
                }
            }
        } while (ret != 0);
        printf(" -- thread A -- lock acquired. Enter critical section\n");

        /* CRITICAL SECTION */
        sleep(rand() % 2);

        /* done, now unlocking the semaphore */
        printf(" -- thread A -- leaving critical section\n");
        ret = sem_post(&mysem);
        if (ret != 0) {
            perror(" -- thread A -- Error in sem_post");
            pthread_exit(NULL);
        }
    }

    printf(" -- thread A -- closing up\n");

    pthread_exit(NULL);
```

```
}


void *thread_b(void *x)
{
   unsigned int i, num;
   int ret;


   printf(" -- thread B -- starting\n");
   num = ((unsigned int)rand() % 100);

   /* this does (do_normal_stuff, do_critical_stuff) n times */
   for (i = 0; i < num; i++) {
      /* do normal stuff */
      sleep(3 + (rand() % 5));

      /* wants to enter the critical section */
      ret = sem_trywait(&mysem);
      if (ret != 0) {
         /* either an error happened, or the semaphore is locked */
         if (errno != EAGAIN) {
            /* an event different from "the semaphore was locked" happened */
            perror(" -- thread B -- error in sem_trywait. terminating -> ");
            pthread_exit(NULL);
         }

         printf(" -- thread B -- cannot enter critical section: semaphore locked\n");
      } else {
         /* CRITICAL SECTION */
         printf(" -- thread B -- enter critical section\n");

         sleep(rand() % 10);

         /* done, now unlocking the semaphore */
         printf(" -- thread B -- leaving critical section\n");
         sem_post(&mysem);
      }
   }

   printf(" -- thread B -- closing up\n");

   /* joining main() */
   pthread_exit(NULL);
}


void signal_handler(int type)
{
   /* do nothing */
   printf("process got signal %d\n", type);
   return;
}
```

IPC: POSIX shared memory

- shm_open
- mmap
- shm_unlink

Notes:
Shared memory is what the name says about it: a segment of memory shared between several processes. UNIX knows System V shared memory and POSIX shared memory. This covers the POSIX way, but the other one is just matter of different system call names for initializing and terminating the segment.
Shared memory in UNIX is based on the concept of *memory mapping*. The segment of memory shared is coupled with an actual file in the filesystem. The file content is (ideally) a mirror of the memory segment at any time. The mapping between the shared segment content and the mapped file is persistent. The mapped file can be moved (even replicated) arbitrarily; this turns often suitable for reimporting such *image* of the memory segment across different runs of the same process (or different ones too) [See Further notes on mapping, below].
In fact, keeping an image file of the memory segment is occasionally useless. *Anonymous mapping* is when a shared segment isn't mapped to any actual object in the filesystem. This behavior is requested either explicitly (BSD way) or via /dev/zero, depending on the implementation [See Further notes on anonymous mapping, below].
In POSIX shared memory, the memory segment is mapped onto a posix object; consequently, it suffers the same problem discussed in notes on POSIX objects below. The shm_open() function is used to create such object (or open an existing one). The core function mmap() is then used to actually attach a memory segment of user-defined size into the address space of the process and map its content to the one of the ipc object (in case of open existing object, the former content is also imported). Thereafter, if other processes shm_open and mmap the same memory object, the memory segment will be effectively *shared*.
Please mind that shared memory is natural when working with threads in place of tasks. Threading is frequently used as a quick quirk when shared memory is wanted. Similarly, programming with shared memory requires many of the programmer attentions discussed for threads: usually shared memory is used in conjunction with some form of synchronization, often semaphores.
A segment of memory mapped into a process' address space will never be detached unless the munmap() is called. A posix shared memory object will persist unless the shm_unlink() function is used (or manually removed from the filesystem, where possible).

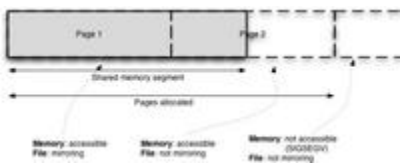

Figure 1

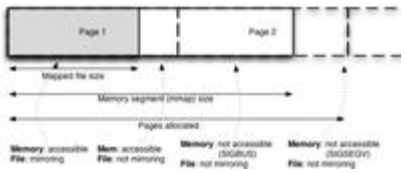

Figure 2

Further notes on mapping:
The mmap function maps objects into the process' address space. Mmap does two things: it causes the kernel to allocate and attach to the process address space a suitable number of page for covering the size requested by the user, and it establish the mapping between the object content and the memory content.
When mmap is called, the kernel allocates a suitable number of pages to cover the size requested by the user. Being memory allocation quantized, the real memory allocated into the process address space is rarely large what the user asked for. In contrast, files on disk can be stretched to whatever size byte by byte. Several effects raise from this fact. Figure 1 illustrates what happens when the memory segment size and mapped file size requested are equal, but not multiple of the page size. Figure 2 shows what happens when the size of

the mapped file is shorter than the size requested for the memory segment.

Further notes on anonymous mapping:
Anonymous mapping is used share memory segment without mapping it to a persistent object (posix object, or file). It is the shared memory counterpart of unnamed semaphores, pipes etc. With this role, it is clear that anonymous mapping can only occur between related processes. From this mechanism, shared memory benefits in terms of performance. The developer is also advantaged not having to handle any persistent object.
There are 2 ways to get anonymous memory mapping. BSD provides a special flag (MAP_ANON) for mmap() to state this wish explicitly. Linux implements the same feature since version 2.4, but in general this is not a portable way to carry out the job.
System V (SVR4) doesn't provided such method, but the /dev/zero file can be used for the same purpose. Upon opening, the memory segment is initialized with the content read from the file (which is always 0s for this special file). Hereafter, whatever the kernel will write on the special file for mirroring from memory will be just discarded by the kernel.

Summary:

what to #include: sys/types.h , sys/mman.h , possibly fcntl.h for O_* macros
functions:

1. int **shm_open**(const char *name, int flags, mode_t mode)
   open (or create) a shared memory object with the given POSIX name. The flags argument instructs on how to open the object: the most relevant ones are O_RDONLY xor O_RDWR for access type, and O_CREAT for create if object doesn't exist. mode is only used when the object is to be created, and specifies its access permission (umask style).
   Returns a file descriptor if successful, otherwise -1.
2. void ***mmap**(void *start_addr, size_t len, int protection, int flags, int fd, off_t offset)
   maps the file fd into memory, starting from offset for a segment long len. len is preferably a multiple of the system page size; the actual size alloced is the smallest number of pages needed for containing the length requested. Notably, if the file is shorter than offset + len, the orphaned portion is still available for accessing, but what's written to it is not persistent.
   protection specifies the rules for accessing the segment: PROT_READ , PROT_WRITE, PROT_EXEC to be ORed appropriately. flags sets some details on how the segment will be mapped, the most relevant ones being private (MAP_PRIVATE) or shared (MAP_SHARED, default) and, on the systems supporting it, if it is an anonymous segment (MAP_ANON), in which case fd is good to be -1. fd is simply the file descriptor as returned by shm_open.
3. int **shm_unlink**(const char *name)
   removes a shared memory object specified by name. If some process is still using the shared memory segment associated to name, the segment is not removed until all references to it have been closed. Commonly, shm_unlink is called right after a successful shm_create, in order to assure the memory object will be freed as soon as it's no longer used.

anonymous mapping is useful when the persistence of the shared segment isn't requested, but like all unnamed mechanisms it only works between related processes. Two methods are available throughout the different OSes:

1. BSD4.4 anonymous memory mapping: the mmap function is passed -1 as fd and an ored MAP_ANON in flags. The offset argument is ignored. The filesystem isn't touched at all.

2. System V anonymous mapping: the /dev/zero file is opened to be passed to mmap. Then, the segment content is initially set to 0, and everything written to it is lost. This machanism also works with systems supporting the BSD way, of course.

Examples:
1) Two processes: a server creates a mapped memory segment and produces a message on it, then terminates. The client opens the persistent object mapping that memory and reads back the message left. The mapped object acts like a persistent memory image in this example.

```
/*
 *  shm_msgserver.c
 *
 *  Illustrates memory mapping and persistence, with POSIX objects.
 *  This process produces a message leaving it in a shared segment.
 *  The segment is mapped in a persistent object meant to be subsequently
 *  open by a shared memory "client".
 *
 *
 *  Created by Mij <mij@bitchx.it> on 27/08/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unixprg/
 *
 */


#include <stdio.h>
/* shm_* stuff, and mmap() */
#include <sys/mman.h>
#include <sys/types.h>
/* exit() etc */
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
/* for random() stuff */
#include <stdlib.h>
#include <time.h>


/* Posix IPC object name [system dependant] - see
http://mij.oltrelinux.com/devel/unixprg/index2.html#ipc__posix_objects */
#define SHMOBJ_PATH         "/foo1423"
/* maximum length of the content of the message */
#define MAX_MSG_LENGTH      50
/* how many types of messages we recognize (fantasy) */
#define TYPES               8

/* message structure for messages in the shared segment */
struct msg_s {
    int type;
    char content[MAX_MSG_LENGTH];
};


int main(int argc, char *argv[]) {
    int shmfd;
    int shared_seg_size = (1 * sizeof(struct msg_s));   /* want shared segment capable of storing 1 message */
    struct msg_s *shared_msg;      /* the shared segment, and head of the messages list */
```

```
    /* creating the shared memory object    --  shm_open()  */
    shmfd = shm_open(SHMOBJ_PATH, O_CREAT | O_EXCL | O_RDWR, S_IRWXU | S_IRWXG);
    if (shmfd < 0) {
        perror("In shm_open()");
        exit(1);
    }
    fprintf(stderr, "Created shared memory object %s\n", SHMOBJ_PATH);

    /* adjusting mapped file size (make room for the whole segment to map)      --  ftruncate() */
    ftruncate(shmfd, shared_seg_size);

    /* requesting the shared segment    --  mmap() */
    shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED,
shmfd, 0);
    if (shared_msg == NULL) {
        perror("In mmap()");
        exit(1);
    }
    fprintf(stderr, "Shared memory segment allocated correctly (%d bytes).\n", shared_seg_size);


    srandom(time(NULL));
    /* producing a message on the shared segment */
    shared_msg->type = random() % TYPES;
    snprintf(shared_msg->content, MAX_MSG_LENGTH, "My message, type %d, num %ld", shared_msg->type,
random());


    /* [uncomment if you wish] requesting the removal of the shm object     --  shm_unlink() */
/*
    if (shm_unlink(SHMOBJ_PATH) != 0) {
        perror("In shm_unlink()");
        exit(1);
    }
*/

    return 0;
}

/*
 *  shm_msgclient.c
 *
 *  Illustrates memory mapping and persistence, with POSIX objects.
 *  This process reads and displays a message left it in "memory segment
 *  image", a file been mapped from a memory segment.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 27/08/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unixprg/
 *
 */

#include <stdio.h>
/* exit() etc */
#include <unistd.h>
/* shm_* stuff, and mmap() */
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
```

```
/* for random() stuff */
#include <stdlib.h>
#include <time.h>


/* Posix IPC object name [system dependant] - see
http://mij.oltrelinux.com/devel/unixprg/index2.html#ipc__posix_objects */
#define SHMOBJ_PATH         "/foo1423"
/* maximum length of the content of the message */
#define MAX_MSG_LENGTH      50
/* how many types of messages we recognize (fantasy) */
#define TYPES               8

/* message structure for messages in the shared segment */
struct msg_s {
    int type;
    char content[MAX_MSG_LENGTH];
};


int main(int argc, char *argv[]) {
    int shmfd;
    int shared_seg_size = (1 * sizeof(struct msg_s));   /* want shared segment capable of storing 1 message */
    struct msg_s *shared_msg;      /* the shared segment, and head of the messages list */


    /* creating the shared memory object    --  shm_open()  */
    shmfd = shm_open(SHMOBJ_PATH, O_RDWR, S_IRWXU | S_IRWXG);
    if (shmfd < 0) {
        perror("In shm_open()");
        exit(1);
    }
    printf("Created shared memory object %s\n", SHMOBJ_PATH);

    /* requesting the shared segment    --  mmap() */
    shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED,
shmfd, 0);
    if (shared_msg == NULL) {
        perror("In mmap()");
        exit(1);
    }
    printf("Shared memory segment allocated correctly (%d bytes).\n", shared_seg_size);


    printf("Message type is %d, content is: %s\n", shared_msg->type, shared_msg->content);

    return 0;
}
```

Run the following:
./shm_msgserver
./shm_msgclient

2) Example on BSD asonymous memory mapping. The process gets a semaphore, establishes an anonymous memory mapping and forks (so the data space get independent). Both the parent and the child, in mutex, update the value of the shared segment by random quantities.

```
/*
```

```
 *  shm_anon_bsd.c
 *
 *  Anonymous shared memory via BSD's MAP_ANON.
 *  Create a semaphore, create an anonymous memory segment with the MAP_ANON
 *  BSD flag and loop updating the segment content (increment casually) with
 *  short intervals.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 29/08/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unixprg/
 *
 */

#include <stdio.h>
/* for shm_* and mmap() */
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
/* for getpid() */
#include <unistd.h>
/* exit() */
#include <stdlib.h>
/* for sem_* functions */
#include <sys/stat.h>
#include <semaphore.h>
/* for seeding time() */
#include <time.h>

/* name of the semaphore */
#define     SEMOBJ_NAME         "/semshm"
/* maximum number of seconds to sleep between each loop operation */
#define     MAX_SLEEP_SECS      3
/* maximum value to increment the counter by */
#define     MAX_INC_VALUE       10


int main(int argc, char *argv[]) {
    int shmfd;
    int shared_seg_size = 2 * sizeof(int);
    int *shared_values;     /* this will be a (shared) array of 2 elements */
    sem_t *sem_shmsegment;  /* semaphore controlling access to the shared segment */
    pid_t mypid;


    /* getting a new semaphore for the shared segment       -- sem_open()   */
    sem_shmsegment = sem_open(SEMOBJ_NAME, O_CREAT | O_EXCL, S_IRWXU | S_IRWXG, 1);
    if ((int)sem_shmsegment == SEM_FAILED) {
        perror("In sem_open()");
        exit(1);
    }
    /* requesting the semaphore not to be held when completely unreferenced */
    sem_unlink(SEMOBJ_NAME);

    /* requesting the shared segment    --  mmap() */
    shared_values = (int *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANON, -1, 0);
    if ((int)shared_values == -1) {
        perror("In mmap()");
        exit(1);
    }
    fprintf(stderr, "Shared memory segment allocated correctly (%d bytes) at %u.\n", shared_seg_size,
(unsigned int)shared_values);
```

```
    close(shmfd);


    /* dupping the process */
    if (! fork() )
        /* the child waits 2 seconds for better seeding srandom() */
        sleep(2);

    /* seeding the random number generator (% x for better seeding when child executes close) */
    srandom(time(NULL));

    /* getting my pid, and introducing myself */
    mypid = getpid();
    printf("My pid is %d\n", mypid);

    /*
       main loop:
        - pause
        - print the old value
        - choose (and store) a random quantity
        - increment the segment by that
    */
    do {
        sleep(random() % (MAX_SLEEP_SECS+1));       /* pausing for at most MAX_SLEEP_SECS seconds */

        sem_wait(sem_shmsegment);
        /* entered the critical region */

        printf("process %d. Former value %d.", mypid, shared_values[0]);

        shared_values[1] = random() % (MAX_INC_VALUE+1);           /* choose a random value up to
MAX_INC_VALUE */
        shared_values[0] += shared_values[1];   /* and increment the first cell by this value */

        printf(" Incrementing by %d.\n", shared_values[1]);

        /* leaving the critical region */
        sem_post(sem_shmsegment);
    } while (1);

    /* freeing the reference to the semaphore */
    sem_close(sem_shmsegment);


    return 0;
}
```