Inter-Process Communication Using Shared Memory

Mark Sitkowski C.Eng, M.I.E.E
http://www.designsim.com.au

Everyone knows how to design a TCP/IP based client-server system (don't they...?) so we're going to do something different.
Often, we don't need to connect to another machine, but need to transfer data between processes on the same machine.
Consider the situation, where we have forked twenty, or so child processes, each of which runs a cursor in the database. The child processes save the data in local data structures then, when the cursors have run to completion, they need to send their data back to the parent.
Here's a neat inter-process communication technique, capable of achieving this, the principal functions of which we will examine in detail

Shared Memory client-Server Systems

A shared memory segment, is a section of RAM, whose address is known to more than one process. The processes to which this address is known, have either read only, or read/write permission to the memory segment, whose access rights are set in the manner used by chmod.
Most machines dedicated to manipulation of large databases are not short of RAM, and figures of 3 to 5 GB are fairly common. Where two processes coexist on the one machine, communication of data through the mechanism of shared memory becomes an attractive proposition.

Among the advantages of a shared memory system are:

- Memory-to-memory data transfers are inherently fast, and there are never any connection problems, as can occasionally occur with TCP/IP.
- The total amount of memory used by a TCP/IP client server system, in the worst case, is double the amount necessary to store the data. First, the client has to extract the data, and store it in local data structures, like arrays of structures, or linked lists and, then, the server has to allocate the same amount of memory, to receive the same data. Memory is returned only when the client terminates.

The drawbacks include:

- The amount of free RAM must always be adequate to cater for the maximum which may be required.
- If a process terminates unexpectedly, without first deleting its shared memory segment, that segment remains unusable. If the segment is of significant size, this could have an adverse effect on the performance of the machine.
- The parent/child interaction, at the beginning of the operation is slightly more complicated. The child needs to communicate the address of the shared memory segment, which it has allocated for the data it is about to send back to the parent. In order for this to be possible, the parent must, first, establish a small piece of shared memory, where the child can place this address.
- The timing of connections and disconnections is not event-driven.

Shared Memory Commands

A shared memory segment is requested with the shmget() system call, which has the synopsis:

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

The return value is the shared memory identifier, an integer value, which is used in subsequent manipulations.
On some versions of Unix, the 'key' parameter can be synthesised, by calling a special function but, for most purposes and, certainly for ours, the symbolic value IPC_PRIVATE, which is #defined as zero, will be exclusively used.
The variable 'size' is merely the memory segment size, in bytes, while the 'shmflag parameter is the logical OR of one or more of the following:

IPC_CREAT create segment if key doesn't exist
IPC_EXCL fail if key already exists
IPC_NOWAIT flag error if we must wait for the segment

SHM_R make segment readable
SHM_W make segment writeable
SHM_RND attach on page boundary

SHM_RDONLY attach as read-only. If this is omitted, the default is read/write.

SHM_SHARE_MMU share virtual memory among processes which share this segment. This may be useful, if there is a danger of one or more of such processes being swapped out.
SHM_PAGEABLE As above, but the memory may be dynamically resized, within the size allocated.

Typically, we would make the call as follows:

```
#include <shm.h>
int shmid;
size_t size = 10000000;

if((shmid = shmget(IPC_PRIVATE, size, IPC_CREAT | SHM_PAGEABLE |
                                          SHM_R | SHM_W)) <= 0){
    perror("Error obtaining shared memory");

 }
```

Having acquired our shared memory, we now have to attach it to the data segment of our process. This is achieved by using the shmat() system call.

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

The return value is a pointer to the start address of the attached memory segment. It is declared (void *) for the same reason as that of malloc(). It is the responsibility of the user to cast this to the datatype for which the memory will be used.
The 'shmid' parameter is that returned from the shmget call, above, while shmaddr has the following common options:

- shmaddr = 0 the segment is attached to the first available suitably aligned address.
- shmaddr != 0 AND shmflag is either SHM_SHARE_MMU (which means the kernel will share its unpageable memory resources) or SHM_PAGEABLE (memory is pageable), the segment is attached to the first suitably aligned address at shmaddr. This is the most commonly used value, and one we shall use.

The shmflag argument can have most of the values passed to shmget():

SHM_R | SHM_W | SHM_RDONLY | SHM_RND | SHM_SHARE_MMU | SHM_PAGEABLE

We will use shmat() as follows:

```
unsigned char *mptr;

if((mptr = (unsigned char *)shmat(shmid, 0, SHM_RND)) == (unsigned char *)-1
){
        perror(" Error attaching to shared memory\n");
 }
```

Unlike malloc, which returns NULL, on failure, shmat returns –1, which results in the need for the clumsy cast to (unsigned char *), above.

Each attached memory segment has associated with it, a structure, of type struct shmid_ds, which may be used to obtain information about the segment:

```
struct shmid_ds {
        struct ipc_perm shm_perm;       /* permissions struct */
        size_t          shm_segsz;      /* size of segment I(bytes) */
        struct anon_map *shm_amp;       /* segment anon_map pointer */
        ushort_t        shm_lkcnt;      /* number of times it is being locked */
        pid_t           shm_lpid;       /* pid of last shmop */
        pid_t           shm_cpid;       /* pid of creator */
        shmatt_t        shm_nattch;     /* used only for shminfo */
        ulong_t         shm_cnattch;    /* used only for shminfo */
        time_t          shm_atime;      /* last shmat time */
        time_t          shm_dtime;      /* last shmdt time */
        time_t          shm_ctime;      /* last change time */
};
```

The shmctl() system call, is designed to load the contents of this structure into a local structure of the above type:

```
if(shmctl(shmid2, IPC_STAT, &buf) < 0){
    printf("Unable to get shm status\n");
 }
```

The variable IPC_STAT signifies that this is a query. The variable IPC_SET allows the setting of the members of the ipc_perm structure, and changing the following permissions:

shm_perm.uid
shm_perm.gid

shm_perm.mode

Still considering our hypothetical database access program, described at the beginning of this chapter, the sequence of events, for creating a shared memory client-server system, would be:

- Parent process allocates a 100-byte shared memory segment, large enough to hold a token, with the child's ID, the number of bytes, or data structures being returned and the shared memory ID allocated and returned by the child
- Parent .forks child processes, each of which is passed the shared memory ID of the 100-byte token memory segment.. .
- Child process accesses the database, and queries the number of rows which will be returned by the cursor, which it intends to run.
- Child process allocates shared memory, large enough to hold the data, then retrieves the data from the database, and loads it into the memory segment.
- Child process places its identifier, the number of rows being returned and the shared memory ID of the retrieved data in the 100-byte token memory segment.
- Parent reads the child's identifier, the number of rows being returned abd the shared memory ID. It then attaches to the shared memory segment and accesses the data.

Server

This code would probably reside in the routine which launched child processes, and require the following global declarations:

```
/* the shmid of the token memory being passed to the child */
Int shmid_s;

/* an array for storing pointers to all the tokens, passed to all child processes */
Unsigned char chptr[NCHILDREN];


Server(char *cursor_SQL_string, int which_cursor)

{

 /*
    * token memory, so for child to write its ID and shmid
    */
   token = sizeof(unsigned char) * 200;

   /*
    * shmid_s is global, so it can be viewed by the child process,
    * and attached.
    */
   if((shmid_s = shmget((key_t)IPC_PRIVATE, token, IPC_CREAT | 0666)) <= 0){
      perror("Server: Error obtaining shared memory");
      return(-1);
   }

   /*
    * shmat returns a pointer to the segment defined by shmid
```

```
     */
    if((chptr[which] = (unsigned char *)shmat(shmid_s, 0, SHM_RND)) == (unsigne
d char *)-1){
        perror("Server: Error attaching to shared memory");
        return(-1);
    }

/* the next line cleans the memory we're going to use */

 memset((char *)chptr[which], '\0', token);

     /* launch child process */

    switch((pid = fork())){
        case –1:
            perror("Fork");
        break;
        case 0:                  /* in child process */
             /* connect to database, prepare cursor from cursor string,
              * declare it, and open it
             */

             /* attach the child to the token memory */

             if((chptr[which_cursor] = (unsigned char *)shmat(shmid_s, 0,
SHM_PAGEABLE)) == (unsigned char *)-1){
               perror("Client: Error attaching to incoming shared memory");
               exit(-1);
             }
             switch(which_cursor){
                 case 1:
                    /* run SQL query to determine number of rows to be returned */

                    /* allocate shared memory to hold all the data */

                    if((shmid = shmget(IPC_PRIVATE, size,  IPC_CREAT | 0666)) <= 0){
                        printf("Memory allocation failed\n");
                        quit(-1);
                     }
                      /*
                       *  now get a pointer to the actual memory,
                       *  cast to the data type of the structure we expect to receive
                       */
                  if((mpt = (struct xyz *)shmat(shmid, 0, SHM_RND)) == (struct xyzg *)-1){
                          perror("Client: Error attaching to shared memory");
                          quit(-1);
                       }

                        /* code to fetch cursor into the mpt[] array of structures */

                        /* when all rows have been retrieved, write results to token */
                        sprintf((char *)chptr[which], "%d %d %d", which, rcnt, shmid);
                   break;
                    case 2:
                        /*  code for second cursor, which has
```

```
                    *  different cursor string and data structures
                    */
                break;
                case 3:       /* etc …. */
                 break;
                 default:
                    printf("Unknown cursor\n");
                 break;
            }
         /* code to close the cursor */
        break;
        default:      /* in the parent process */
            children++
         break;
      }

  }
```

The above routine would be called once for every cursor and, after the last call, each element of the array chptr[] would contain a pointer to the shared memory tokens, passed to all the children.
We would then call a monitor routine, which would scan the elements of the array, looking for a child identifier, a row count and a shmid. The presence of all three signifies that the cursor in the child has run, and that data is available.

```
monitor()                   /* monitor */

{

int one, two, three;              /* dummy variables for testing token */
int flag = 0;                     /* termination flag */
int done[NCHILDREN];              /* log of completed children */

   printf("Server waiting for clients to connect shm segments...\n");
   th = 0;
   memset((char *)done, '\0', sizeof(done));

   while(1){
      for(i = 1; i <= NCHILDREN; i++){
         if(sscanf((char *)chptr[i], "%d %d %d", &one, &two, &three) == 3){
            if(one == 0 || two == 0 || three == 0) continue;
            if(done[i] == 99) continue;
            printf("Child %d returned\n", i);
            children--;
            done[i] = 99;           /* mark this child as having completed */

            /* let a thread deal with this, while we continue to look */
            if((pthread_create(&thr[th], NULL, xserve, (void *)chptr[i])) != 0){
               printf("Failed to create thr[%d]\n",th);
            }
            /* we don't want to wait for the thread */
            if(pthread_detach(thr[th]) != 0){
               printf("Failed to start thr[%d]\n",th);
            }
            th++;
```

```
            printf("Server %d: Threads running: %d Children: %d\n",
                                        getpid(), th, children);
          }
          if(children == 0){       /* all our cursors have run, so process the data */
            flag = 1;
            break;
          }
        }
         if(flag == 1) break;
    }
}
```

We send a thread to perform the housekeeping on the data that has just arrived, so we can continue to search, uninterrupted, for returned children.

In the function xserve(), we attach to the memory segment, defined by the shmid, returned in the token. We store the pointer, returned by shmat(), in a global array of such pointers, which we will use in the subsequent data manipulation routines, to access our data.

```
Void *
Xserve(unsigned char shm)                    / xserve */

{

int cur_id;
int size;
int shmid_c;

  /* extract token data */
     sscanf((char *)shm, "%d %d %d", &cur_id, &size, &shmid_c);
     printf("Thread %d cursor %d shmid %d ..\n", pthread_self(), cur_id, shmid_c);

     /*
     * shmat returns a pointer to the segment defined by shmid_c
     */
    if((data[cur_id] = (unsigned char *)shmat(shmid_c, 0, SHM_RND)) == (unsigne
d char *)-1){
        perror("Server: Error attaching to shared memory");
        return((void *)-1);
    }

    /*
     * Cast pointers to correct data types,
     * and set no. of records
     */
    switch(cur_id){
        case 1:
            tpt = (struct xyz *)data[cur_id];
            lpt = size;
        break;
        case 2:
            /* same for next cursor */
        break;
```

```
        case 3:   /* etc… */
        break;
    }
}                                       /* xserve */
```