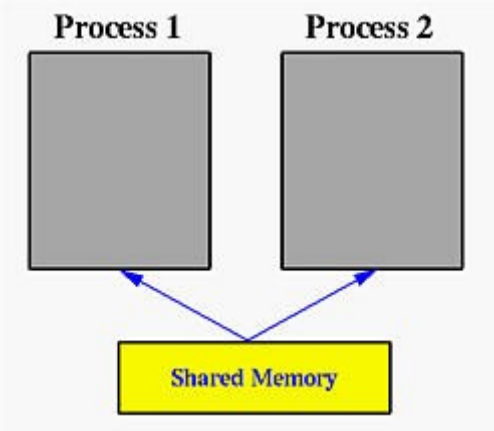
In the discussion of the **fork()** system call, we mentioned that a parent and its children have separate address spaces. While this would provide a more secured way of executing parent and children processes (because they will not interfere each other), they shared nothing and have no way to communicate with each other. A ***shared memory*** is an extra piece of memory that is *attached* to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two processes and their address spaces. The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space. In some sense, the original address spaces is "extended" by attaching this shared memory.



Shared memory is a feature supported by UNIX System V, including Linux, SunOS and Solaris. One process must explicitly ask for an area, using a ***key***, to be shared by other processes. This process will be called the *server*. All other processes, the *clients*, that know the shared area can access it. However, there is no protection to a shared memory and any process that knows it can access it freely. To protect a shared memory from being accessed at the same time by several processes, a synchronization protocol must be setup.

A shared memory segment is identified by a unique integer, the ***shared memory ID***. The shared memory itself is described by a structure of type **shmid_ds** in header file **sys/shm.h**. To use this file, files **sys/types.h** and **sys/ipc.h** must be included. Therefore, your program should start with the following lines:

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

A general scheme of using shared memory is the following:

- For a server, it should be started before any client. The server should perform the following tasks:
    1. Ask for a shared memory with a memory key and memorize the returned shared memory ID. This is performed by system call **shmget()**.
    2. Attach this shared memory to the server's address space with system call **shmat()**.
    3. Initialize the shared memory, if necessary.
    4. Do something and wait for all clients' completion.
    5. Detach the shared memory with system call **shmdt()**.
    6. Remove the shared memory with system call **shmctl()**.
- For the client part, the procedure is almost the same:
    1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
    2. Attach this shared memory to the client's address space.
    3. Use the memory.
    4. Detach all shared memory segments, if necessary.

5. Exit.

Keys

Unix requires a ***key*** of type **key_t** defined in file **sys/types.h** for requesting resources such as shared memory segments, message queues and semaphores. A key is simply an integer of type **key_t**; however, you should not use **int** or **long**, since the length of a key is system dependent.

There are three different ways of using keys, namely:

1. a specific integer value (*e.g.*, 123456)
2. a key generated with function **ftok()**
3. a uniquely generated key using **IPC_PRIVATE** (*i.e.*, a ***private*** key).

The first way is the easiest one; however, its use may be very risky since a process can access your resource as long as it uses the same key value to request that resource. The following example assigns 1234 to a key:

```
key_t     SomeKey;

SomeKey = 1234;
```

The **ftok()** function has the following prototype:

```
key_t  ftok(
            const char *path,      /* a path string       */
            int        id          /* an integer value    */
           );
```

Function **ftok()** takes a character string that identifies a path and an integer (usually a character) value, and generates an integer of type **key_t** based on the first argument with the value of **id** in the most significant position. For example, if the generated integer is $35028A5D_{16}$ and the value of **id** is 'a' (ASCII value = 61_{16}), then **ftok()** returns $61028A5D_{16}$. That is, 61_{16} replaces the first byte of $35028A5D_{16}$, generating $61028A5D_{16}$.

Thus, as long as processes use the same arguments to call **ftok()**, the returned key value will always be the same. The most commonly used value for the first argument is ".", the current directory. If all related processes are stored in the same directory, the following call to **ftok()** will generate the same key value:

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

key_t     SomeKey;

SomeKey = ftok(".", 'x');
```

After obtaining a key value, it can be used in any place where a key is required. Moreover, the place

where a key is required accepts a special parameter, **IPC_PRIVATE**. In this case, the system will generate a unique key and guarantee that no other process will have the same key. If a resource is requested with **IPC_PRIVATE** in a place where a key is required, that process will receive a unique key for that resource. Since that resource is identified with a unique key unknown to the outsiders, other processes will *not* be able to share that resource and, as a result, the requesting process is guaranteed that it owns and accesses that resource exclusively.

Asking for a Shared Memory Segment - shmget()

The system call that requests a shared memory segment is **shmget()**. It is defined as follows:

```
shm_id = shmget(
               key_t     k,        /* the key for the segment
*/
               int       size,     /* the size of the segment
*/
               int       flag);    /* create/use flag
*/
```

In the above definition, **k** is of type **key_t** or **IPC_PRIVATE**. It is the numeric key to be assigned to the returned shared memory segment. **size** is the size of the requested shared memory. The purpose of **flag** is to specify the way that the shared memory will be used. For our purpose, only the following two values are important:

1. **IPC_CREAT | 0666** for a server (*i.e.*, creating and granting read and write access to the server)
2. **0666** for any client (*i.e.*, granting read and write access to the client)

Note that due to Unix's tradition, **IPC_CREAT** is *correct* and **IPC_CREATE** is *not*!!!

If **shmget()** can successfully get the requested shared memory, its function value is a non-negative integer, the shared memory ID; otherwise, the function value is negative. The following is a server example of requesting a private shared memory of four integers:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>
#include  <stdio.h>
     .....
int       shm_id;        /* shared memory ID      */
     .....
shm_id = shmget(IPC_PRIVATE, 4*sizeof(int), IPC_CREAT | 0666);
if (shm_id < 0) {
     printf("shmget error\n");
     exit(1);
}

/* now the shared memory ID is stored in shm_id */
```

If a client wants to use a shared memory created with **IPC_PRIVATE**, it must be a child process of

the server, created ***after*** the parent has obtained the shared memory, so that the private key value can be passed to the child when it is created. For a client, changing **IPC_CREAT | 0666** to **0666** works fine. **A warning to novice C programmers:** don't change **0666** to **666**. The leading **0** of an integer indicates that the integer is an octal number. Thus, **0666** is 110110110 in binary. If the leading zero is removed, the integer becomes six hundred sixty six with a binary representation 1111011010.

Server and clients can have a parent/client relationship or run as separate and unrelated processes. In the former case, if a shared memory is requested and attached prior to forking the child client process, then the server may want to use **IPC_PRIVATE** since the child receives an identical copy of the server's address space which includes the attached shared memory. However, if the server and clients are separate processes, using **IPC_PRIVATE** is unwise since the clients will not be able to request the same shared memory segment with a unique and unknown key.

Suppose process 1, a server, uses **shmget()** to request a shared memory segment successfully. That shared memory segment exists somewhere in the memory, but is not yet part of the address space of process 1 (shown with dashed line below). Similarly, if process 2 requests the same shared memory segment with the same key value, process 2 will be granted the right to use the shared memory segment; but it is not yet part of the address space of process 2. To make a requested shared memory segment part of the address space of a process, use **shmat()**.

Attaching a Shared Memory Segment to an Address Space - shmat()

After a shared memory ID is returned, the next step is to attach it to the address space of a process. This is done with system call **shmat()**. The use of **shmat()** is as follows:

```
shm_ptr = shmat(
              int       shm_id,        /* shared memory ID    */
              char      *ptr,          /* a character pointer */
              int       flag);         /* access flag         */
```

System call **shmat()** accepts a shared memory ID, **shm_id**, and attaches the indicated shared memory to the program's address space. The returned value is a pointer of type **(void *)** to the attached shared memory. Thus, casting is usually necessary. If this call is unsuccessful, the return value is **-1**. Normally, the second parameter is **NULL**. If the flag is **SHM_RDONLY**, this shared memory is attached as a read-only memory; otherwise, it is readable and writable.

In the following server's program, it asks for and attaches a shared memory of four integers.

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

int       shm_id;
key_t     mem_key;
int       *shm_ptr;
```

```
mem_key = ftok(".", 'a');
shm_id = shmget(mem_key, 4*sizeof(int), IPC_CREAT | 0666);
if (shm_id < 0) {
     printf("*** shmget error (server) ***\n");
     exit(1);
}

shm_ptr = (int *) shmat(shm_id, NULL, 0);  /* attach */
if ((int) shm_ptr == -1) {
     printf("*** shmat error (server) ***\n");
     exit(1);
}
```

The following is the counterpart of a client.

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

int       shm_id;
key_t     mem_key;
int       *shm_ptr;

mem_key = ftok(".", 'a');
shm_id = shmget(mem_key, 4*sizeof(int), 0666);
if (shm_id < 0) {
     printf("*** shmget error (client) ***\n");
     exit(1);
}

shm_ptr = (int *) shmat(shm_id, NULL, 0);
if ((int) shm_ptr == -1) { /* attach */
     printf("*** shmat error (client) ***\n");
     exit(1);
}
```

Note that the above code assumes the server and client programs are in the current directory. In order for the client to run correctly, the server must be started first and the client can only be started after the server has successfully obtained the shared memory.

Suppose process 1 and process 2 have successfully attached the shared memory segment. This shared memory segment will be part of their address space, although the actual address could be different (*i.e.*, the starting address of this shared memory segment in the address space of process 1 may be different from the starting address in the address space of process 2).

Detaching and Removing a Shared Memory Segment - shmdt() and shmctl()

System call **shmdt()** is used to detach a shared memory. After a shared memory is detached, it cannot be used. However, it is still there and can be re-attached back to a process's address space, perhaps at a different address. To remove a shared memory, use **shmctl()**.

The only argument of the call to **shmdt()** is the shared memory address returned by **shmat()**. Thus, the following code detaches the shared memory from a program:

```
shmdt(shm_ptr);
```

where **shm_ptr** is the pointer to the shared memory. This pointer is returned by **shmat()** when the shared memory is attached. If the detach operation fails, the returned function value is non-zero.

To remove a shared memory segment, use the following code:

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

where **shm_id** is the shared memory ID. **IPC_RMID** indicates this is a remove operation. Note that after the removal of a shared memory segment, if you want to use it again, you should use **shmget()** followed by **shmat()**.

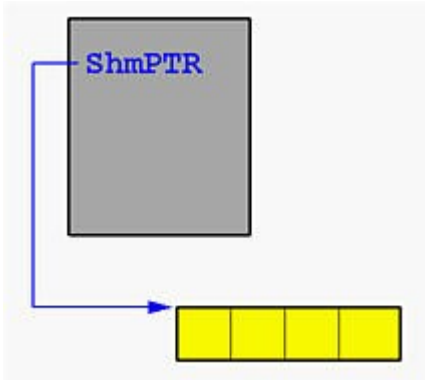
Communicating Between Parent and Child

The following main function runs as a server. It uses **IPC_PRIVATE** to request a private shared memory. Since the client is the server's child process created *after* the shared memory has been created and attached, the child client process will receive the shared memory in its address space and as a result no shared memory operations are required. Code of the file (**shm-01.c**) is shown below:

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

void  ClientProcess(int []);

void  main(int  argc, char *argv[])
{
     int    ShmID;
     int    *ShmPTR;
     pid_t  pid;
     int    status;

     if (argc != 5) {
          printf("Use: %s #1 #2 #3 #4\n", argv[0]);
          exit(1);
     }

     ShmID = shmget(IPC_PRIVATE, 4*sizeof(int), IPC_CREAT | 0666);
     if (ShmID < 0) {
          printf("*** shmget error (server) ***\n");
          exit(1);
     }
     printf("Server has received a shared memory of four
integers...\n");

     ShmPTR = (int *) shmat(ShmID, NULL, 0);
```

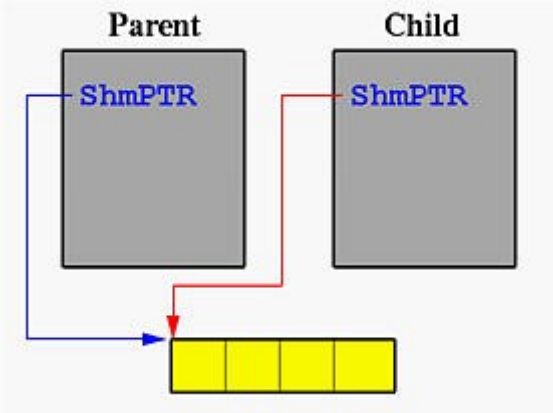
```
        if ((int) ShmPTR == -1) {
             printf("*** shmat error (server) ***\n");
             exit(1);
        }
        printf("Server has attached the shared memory...\n");

        ShmPTR[0] = atoi(argv[1]);
        ShmPTR[1] = atoi(argv[2]);
        ShmPTR[2] = atoi(argv[3]);
        ShmPTR[3] = atoi(argv[4]);
        printf("Server has filled %d %d %d %d in shared memory...\n",
               ShmPTR[0], ShmPTR[1], ShmPTR[2], ShmPTR[3]);

        printf("Server is about to fork a child process...\n");
        pid = fork();
        if (pid < 0) {
             printf("*** fork error (server) ***\n");
             exit(1);
        }
        else if (pid == 0) {
             ClientProcess(ShmPTR);
             exit(0);
        }

        wait(&status);
        printf("Server has detected the completion of its child...\n");
        shmdt((void *) ShmPTR);
        printf("Server has detached its shared memory...\n");
        shmctl(ShmID, IPC_RMID, NULL);
        printf("Server has removed its shared memory...\n");
        printf("Server exits...\n");
        exit(0);
}

void  ClientProcess(int  SharedMem[])
{
        printf("   Client process started\n");
        printf("   Client found %d %d %d %d in shared memory\n",
                   SharedMem[0], SharedMem[1], SharedMem[2],
SharedMem[3]);
        printf("   Client is about to exit\n");
}
```

This program asks for a shared memory of four integers and attaches this shared memory segment to its address space. Pointer **ShmPTR** points to the shared memory segment. After this is done, we have the following:



Then, this program forks a child process to run function **ClientProcess()**. Thus, two *identical* copies

of address spaces are created, each of which has a variable **ShmPTR** whose value is a pointer to the shared memory. As a result, the child process has already known the location of the shared memory segment and does not have to use **shmget()** and **shmat()**. This is shown below:



The parent waits for the completion of the child. For the child, it just retrieves the four integers, which were stored there by the parent *before* forking the child, prints them and exits. The **wait()** system call in the parent will detect this. Finally, the parent exits.

Communicating Between Two Separate Processes

In this example, the server and client are separate processes. First, a naive communication scheme through a shared memory is established. The shared memory consists of one status variable **status** and an array of four integers. Variable **status** has value **NOT_READY** if the data area has not yet been filled with data, **FILLED** if the server has filled data in the shared memory, and **TAKEN** if the client has taken the data in the shared memory. The definitions are shown below. You write this in a header file **shm-02.h**.

```
#define  NOT_READY  -1
#define  FILLED     0
#define  TAKEN      1

struct Memory {
     int  status;
     int  data[4];
};
```

Assume that the server and client are in the current directory. The server uses **ftok()** to generate a key and uses it for requesting a shared memory. Before the shared memory is filled with data, **status** is set to **NOT_READY**. After the shared memory is filled, the server sets **status** to **FILLED**. Then, the server waits until **status** becomes **TAKEN**, meaning that the client has taken

the data.

The following is the server program. Code of the server program **server.c** is shown below.

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

#include  "shm-02.h"

void  main(int  argc, char *argv[])
{
     key_t          ShmKEY;
     int            ShmID;
     struct Memory  *ShmPTR;

     if (argc != 5) {
          printf("Use: %s #1 #2 #3 #4\n", argv[0]);
          exit(1);
     }

     ShmKEY = ftok(".", 'x');
     ShmID = shmget(ShmKEY, sizeof(struct Memory), IPC_CREAT | 0666);
     if (ShmID < 0) {
          printf("*** shmget error (server) ***\n");
          exit(1);
     }
     printf("Server has received a shared memory of four
integers...\n");

     ShmPTR = (struct Memory *) shmat(ShmID, NULL, 0);
     if ((int) ShmPTR == -1) {
          printf("*** shmat error (server) ***\n");
          exit(1);
     }
     printf("Server has attached the shared memory...\n");

     ShmPTR->status  = NOT_READY;
     ShmPTR->data[0] = atoi(argv[1]);
     ShmPTR->data[1] = atoi(argv[2]);
     ShmPTR->data[2] = atoi(argv[3]);
     ShmPTR->data[3] = atoi(argv[4]);
     printf("Server has filled %d %d %d %d to shared memory...\n",
            ShmPTR->data[0], ShmPTR->data[1],
            ShmPTR->data[2], ShmPTR->data[3]);
     ShmPTR->status = FILLED;

     printf("Please start the client in another window...\n");

     while (ShmPTR->status != TAKEN)
          sleep(1);

     printf("Server has detected the completion of its child...\n");
     shmdt((void *) ShmPTR);
     printf("Server has detached its shared memory...\n");
     shmctl(ShmID, IPC_RMID, NULL);
     printf("Server has removed its shared memory...\n");
     printf("Server exits...\n");
     exit(0);
}
```

The client part is similar to the server. It waits until **status** is **FILLED**. Then, the clients retrieves the data and sets **status** to **TAKEN**, informing the server that data have been taken. The following is the client program **client.c**.

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

#include  "shm-02.h"

void  main(void)
{
     key_t          ShmKEY;
     int            ShmID;
     struct Memory  *ShmPTR;

     ShmKEY = ftok(".", 'x');
     ShmID = shmget(ShmKEY, sizeof(struct Memory), 0666);
     if (ShmID < 0) {
          printf("*** shmget error (client) ***\n");
          exit(1);
     }
     printf("   Client has received a shared memory of four
integers...\n");

     ShmPTR = (struct Memory *) shmat(ShmID, NULL, 0);
     if ((int) ShmPTR == -1) {
          printf("*** shmat error (client) ***\n");
          exit(1);
     }
     printf("   Client has attached the shared memory...\n");

     while (ShmPTR->status != FILLED)
          ;
     printf("   Client found the data is ready...\n");
     printf("   Client found %d %d %d %d in shared memory...\n",
                ShmPTR->data[0], ShmPTR->data[1],
                ShmPTR->data[2], ShmPTR->data[3]);

     ShmPTR->status = TAKEN;
     printf("   Client has informed server data have been
taken...\n");
     shmdt((void *) ShmPTR);
     printf("   Client has detached its shared memory...\n");
     printf("   Client exits...\n");
     exit(0);
}
```

Since the server program must allocate a shared memory segment to be used by the client, the server must run *before* running the client. One way to do this is that start the server in a window and then move to a second window to start the client.

Background Processes and Shared Memory Status

Background and Foreground Processes

Starting a process in background is easy. Suppose we have a program named **bg** and another program named **fg**. If **bg** must be started in the background, then do the following:

```
bg &
```

If there is an **&** following a program name, this program will be executed as a background process. You can use Unix command **ps** to take a look at the process status report:

```
3719 ... info ...  program name
7156 ... info ...  program name
```

The **ps** command will generate some output similar to the above. At the beginning of each line, there is a number, the *process ID*, and the last item is a program name. If **bg** has been started successfully, you shall see a line with program name **bg**.

To kill any process listed in the **ps** command's output, note its process ID, say **7156**, then use the following

```
kill  7156
```

The program with process ID 7156 will be killed. If you use **ps** to inspect the process status output again, you will not see the process with process ID 7156.

Note that any program you start with a command line is, by default, a foreground process. Thus, the following command starts **fg** as a foreground process:

```
fg
```

There is a short form to start both **bg** (in background) and **fg** (in foreground) at the same time:

```
bg & fg
```

With this technique, the server program can be started as a background process. After the message telling you to start the client, then start the client. The client can be background or a foreground process. In the following, the client is started as a foreground process:

```
server  -4  2  6  -10 &
client
```

Since the server and the client will display their output to the same window, you will see a mixed output. Or, you can start processes in different windows.

Checking Shared Memory Status

Before starting your next run, check to see if you have some shared memory segments that are still there. This can be done with command **ipcs**:

```
ipcs -m
```

A list of shared memory segments will be shown. Then, use command **ipcrm** to remove those unwanted ones:

```
ipcrm -m xxxx
```

where **xxxx** is the shared memory ID obtained from command **ipcs**. Note that without removing allocated shared memory segments you may jeopardize the whole system.

Use **man ipcs** and **man ipcrm** to read more about these two commands.