Understanding Concurrent Programming using System Calls

1 Processes and Related System Calls

This section discusses three important system calls available with Linux OS, namely, fork(), wait(), and execvp().

1.1 The fork() system call

System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

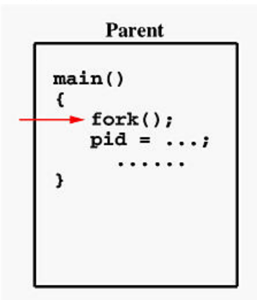
- If fork() returns a negative value, the creation of a child process was unsuccessful.
- fork() returns a zero to the newly created child process.
- fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

Therefore, after the system call to fork(), a simple test can tell which process is the child. Please note that Unix/Linux will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

Let us take an example to make the above points clear. This example does not distinguish parent and the child processes.

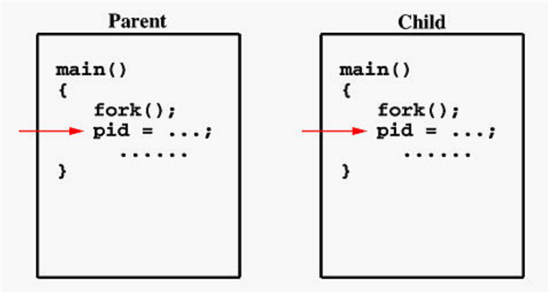
```
/* filename: fork-01.c */
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     fork();       // point of call to fork
     pid = getpid();
     for (i = 1; i <= MAX_COUNT; i++) {
          sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
          write(1, buf, strlen(buf));
     }
}
```

Suppose the above program executes up to the point of call to fork():



If the call to fork() is executed successfully, Unix/Linux will,

- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the fork() call. In this case, both processes will start their execution at the assignment statement as shown below:



Both processes start their execution right after the system call fork(). Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names.

What is the reason of using write rather than printf? It is because printf() is "buffered," meaning printf() will group the output of a process together. While buffering the output for the parent process, the child may also use printf to print out some information, which will also be buffered. As a result, since the output will not be send to screen immediately, you may not get the right order of the expected result. Worse, the output from the two processes may be mixed in strange ways. To overcome this problem, you may consider to use the "unbuffered" write.

If you run this program, you might see the following on the screen:

....

```
This line is from pid 3456, value 13
This line is from pid 3456, value 14
```

....

```
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
```

....

```
This line is from pid 3456, value 21
This line is from pid 3456, value 22
```

....

Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.

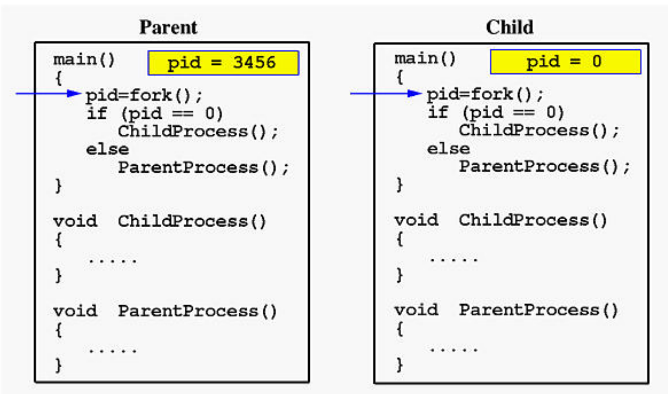
Consider one more simple example, which distinguishes the parent from the child.

```
/* filename: fork-02.c */
#include  <stdio.h>
#include  <sys/types.h>

#define   MAX_COUNT  200

void  ChildProcess(void);                /* child process prototype  */
void  ParentProcess(void);               /* parent process prototype */

void  main(void)
{
     pid_t  pid;

     pid = fork();
```

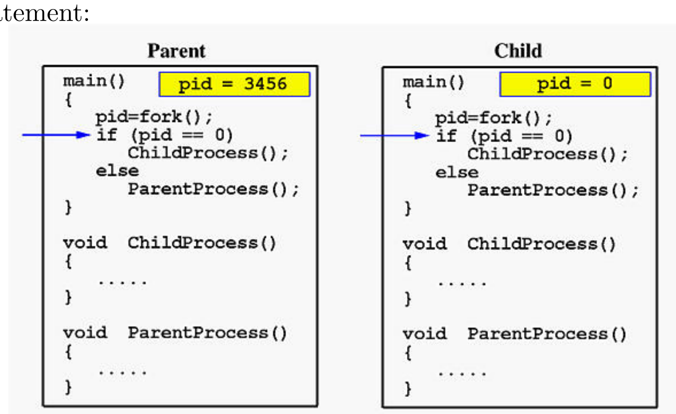
```
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void  ChildProcess(void)
{
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from child, value = %d\n", i);
    printf("### Child process is done ###\n");
}

void  ParentProcess(void)
{
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
    printf("*** Parent process is done ***\n");
}
```

In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable i. For simplicity, printf() is used.

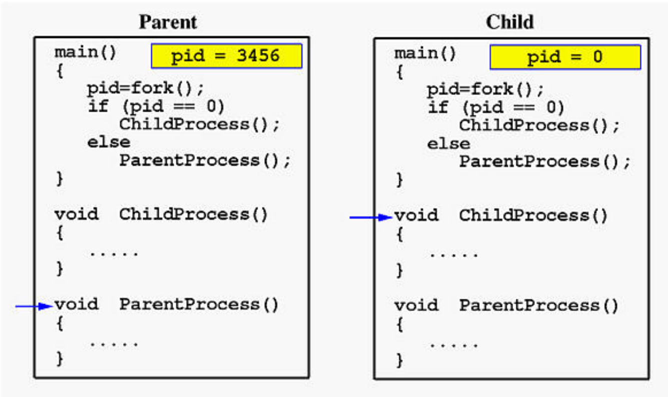
When the main program executes fork(), an identical copy of its address space, including the program and all data, is created. System call fork() returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable pid. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.



Now both programs (i.e., the parent and child) will execute independent of each other starting at the next statement:



In the parent, since pid is non-zero, it calls function ParentProcess(). On the other hand, the child has a zero pid and calls ChildProcess() as shown below:



Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process. Therefore, the value of MAX_COUNT should be large enough so that both processes will run for at least two or more time quanta. If the value of MAX_COUNT is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process. Check what happens if you increase or decrease the value of MAX_COUNT in the above progeram.

1.2 The wait() system call

The system call wait() is easy. This function blocks the calling process until one of its child processes exits or a signal is received. For our purpose, we shall ignore signals. wait() takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of wait() is to wait for completion of child processes.

The execution of wait() could have two possible situations.

1. If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

Consider the following program.

```
/* filename: fork-03.c */
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>
#include  <stdlib.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  ChildProcess(char [], char []);    /* child process prototype  */

void  main(void)
{
     pid_t   pid1, pid2, pid;
     int     status;
     int     i;
     char    buf[BUF_SIZE];

     printf("*** Parent is about to fork process 1 ***\n");
     if ((pid1 = fork()) < 0) {
          printf("Failed to fork process 1\n");
          exit(1);
     }
     else if (pid1 == 0)
          ChildProcess("First", "   ");
```

```
     printf("*** Parent is about to fork process 2 ***\n");
     if ((pid2 = fork()) < 0) {
          printf("Failed to fork process 2\n");
          exit(1);
     }
     else if (pid2 == 0)
          ChildProcess("Second", "      ");

     sprintf(buf, "*** Parent enters waiting status .....\n");
     write(1, buf, strlen(buf));
     pid = wait(&status);
     sprintf(buf, "*** Parent detects process %d was done ***\n", pid);
     write(1, buf, strlen(buf));
     pid = wait(&status);
     printf("*** Parent detects process %d is done ***\n", pid);
     printf("*** Parent exits ***\n");
     exit(0);
}

void  ChildProcess(char *number, char *space)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     pid = getpid();
     sprintf(buf, "%s%s child process starts (pid = %d)\n",
             space, number, pid);
     write(1, buf, strlen(buf));
     for (i = 1; i <= MAX_COUNT; i++) {
          sprintf(buf, "%s%s child's output, value = %d\n", space, number, i);
          write(1, buf, strlen(buf));
     }
     sprintf(buf, "%s%s child (pid = %d) is about to exit\n",
             space, number, pid);
     write(1, buf, strlen(buf));
     exit(0);
}
```

This program shows some typical process programming techniques. The main program creates two child processes to execute the same printing loop and display a message before exit. For the parent process (i.e., the main program), after creating two child processes, it enters the wait state by executing the system call wait(). Once a child exits, the parent starts execution and the ID of the terminated child process is returned in pid so that it can be printed. There are two child processes and thus two wait() calls, one for each child process. In this example, we do not use the returned information in variable status.

However, the parent does not have to wait immediately after creating all child processes. It may do some other tasks. The following is an example.

```
/* filename: fork-04.c */
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  ChildProcess(char [], char []);    /* child process prototype  */
void  ParentProcess(void);               /* parent process prototype */
```

```
void  main(void)
{
     pid_t   pid1, pid2, pid;
     int     status;
     int     i;
     char    buf[BUF_SIZE];

     printf("*** Parent is about to fork process 1 ***\n");
     if ((pid1 = fork()) < 0) {
          printf("Failed to fork process 1\n");
          exit(1);
     }
     else if (pid1 == 0)
          ChildProcess("First", "   ");

     printf("*** Parent is about to fork process 2 ***\n");
     if ((pid2 = fork()) < 0) {
          printf("Failed to fork process 2\n");
          exit(1);
     }
     else if (pid2 == 0)
          ChildProcess("Second", "      ");

     ParentProcess();
     sprintf(buf, "*** Parent enters waiting status .....\n");
     write(1, buf, strlen(buf));
     pid = wait(&status);
     sprintf(buf, "*** Parent detects process %d was done ***\n", pid);
     write(1, buf, strlen(buf));
     pid = wait(&status);
     printf("*** Parent detects process %d is done ***\n", pid);
     printf("*** Parent exits ***\n");
     exit(0);
}

#define  QUAD(x)  (x*x*x*x)

void  ParentProcess(void)
{
     int  a, b, c, d;
     int  abcd, a4b4c4d4;
     int  count = 0;
     char buf[BUF_SIZE];

     sprintf(buf, "Parent is about to compute the Armstrong numbers\n");
     write(1, buf, strlen(buf));
     for (a = 0; a <= 9; a++)
          for (b = 0; b <= 9; b++)
               for (c = 0; c <= 9; c++)
                    for (d = 0; d <= 9; d++) {
                         abcd     = a*1000 + b*100 + c*10 + d;
                         a4b4c4d4 = QUAD(a) + QUAD(b) + QUAD(c) + QUAD(d);
                         if (abcd == a4b4c4d4) {
                              sprintf(buf, "From parent: "
                                      "the %d Armstrong number is %d\n",
                                      ++count, abcd);
                              write(1, buf, strlen(buf));
```

```
                    }
               }
     sprintf(buf, "From parent: there are %d Armstrong numbers\n", count);
     write(1, buf, strlen(buf));
}

void  ChildProcess(char *number, char *space)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     pid = getpid();
     sprintf(buf, "%s%s child process starts (pid = %d)\n",
             space, number, pid);
     write(1, buf, strlen(buf));
     for (i = 1; i <= MAX_COUNT; i++) {
          sprintf(buf, "%s%s child's output, value = %d\n",
                  space, number, i);
          write(1, buf, strlen(buf));
     }
     sprintf(buf, "%s%s child (pid = %d) is about to exit\n",
             space, number, pid);
     write(1, buf, strlen(buf));
     exit(0);
}
```

The main program creates two child processes. Both processes call function ChildProcess(). The main program, the parent process, calls function ParentProcess(). This function computes all Armstrong numbers in the range of 0 and 9999. An Armstrong number in the range of 0 and 9999 is an integer whose value is equal to the sum of its digits raised to the fourth power. After this, the parent enters the wait state, waiting for the completion of its child processes. Note that since we have two processes running concurrently, we have no way to predict which one will terminate first and hence waiting for a specific child process is a risky move. This is why we don't have a "specific" wait in all of the previous programs.

Warning: Although theoretically you can create as many processes as you want, systems always have some limits. Therefore, always check to see if the returned value of fork() is negative and report the result. If this does happen, try to reduce the number of child processes, or re-organize your program.

If the returned pid is unimportant, we can treat function wait() as a procedure. The following code is a simple modification to the last few statements (in the main function) of the previous example.

```
/* filename: fork-05.c modified version of fork-04.c with the following statements */
sprintf(buf, "*** Parent enters waiting status .....\n");
write(1, buf, strlen(buf));
wait(&status);
sprintf(buf, "*** Parent detects a child process was done ***\n");
write(1, buf, strlen(buf));
wait(&status);
printf("*** Parent detects another child process was done ***\n");
printf("*** Parent exits ***\n");
```

1.3 The execvp() system call

The created child process does not have to run the same program as the parent process does. The exec type system calls allow a process to run any program files, which include a binary executable or a shell script. In our following discussion, we only discuss one such system call: execvp(). The execvp() system call requires two arguments:

1. The first argument is a character string that contains the name of a file to be executed.
2. The second argument is a pointer to an array of character strings. More precisely, its type is char **, which is exactly identical to the argv array used in the main program:

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

Note that this argument must be terminated by a zero.

When execvp() is executed, the program file given by the first argument will be loaded into the caller's address space and over-write the program there. Then, the second argument will be provided to the program and starts the execution. As a result, once the specified program file starts its execution, the original program in the caller's address space is gone and is replaced by the new program.

execvp() returns a negative value if the execution fails (e.g., the request file does not exist).

The following is an example.

```
/* filename: interpreter.c */
#include  <stdio.h>
#include  <sys/types.h>

void  parse(char *line, char **argv)
{
     while (*line != '\0') {       /* if not the end of line ....... */
          while (*line == ' ' || *line == '\t' || *line == '\n')
               *line++ = '\0';     /* replace white spaces with 0    */
          *argv++ = line;          /* save the argument position     */
          while (*line != '\0' && *line != ' ' &&
                 *line != '\t' && *line != '\n')
               line++;             /* skip the argument until ...    */
     }
     *argv = '\0';                 /* mark the end of argument list  */
}

void  execute(char **argv)
{
     pid_t  pid;
     int    status;

     if ((pid = fork()) < 0) {     /* fork a child process           */
          printf("*** ERROR: forking child process failed\n");
          exit(1);
     }
     else if (pid == 0) {          /* for the child process:         */
          if (execvp(*argv, argv) < 0) {     /* execute the command  */
               printf("*** ERROR: exec failed\n");
               exit(1);
          }
     }
     else {                                  /* for the parent:      */
          while (wait(&status) != pid)       /* wait for completion  */
               ;
     }
}

void  main(void)
{
     char  line[1024];             /* the input line                 */
     char  *argv[64];              /* the command line argument      */

     while (1) {                   /* repeat until done ....         */
          printf("Interpreter -> ");     /*   display a prompt              */
          gets(line);              /*   read in the command line     */
          printf("\n");
          parse(line, argv);       /*   parse the line               */
```

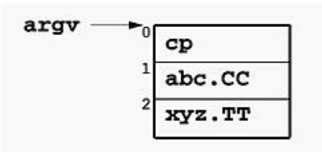
```
          if (strcmp(argv[0], "exit") == 0)  /* is it an "exit"?     */
               exit(0);            /*   exit if it is                */
          execute(argv);           /* otherwise, execute the command */
     }
}
```

Function parse() takes an input line and returns a zero-terminated array of char pointers, each of which points to a zero-terminated character string. This function loops until a binary zero is found, which means the end of the input line *line* is reached. If the current character of *line* is not a binary zero, *parse()* skips all white spaces and replaces them with binary zeros so that a string is effectively terminated. Once *parse()* finds a non-white space, the address of that location is saved to the current position of *argv* and the index is advanced. Then, *parse()* skips all non-whitespace characters. This process repeats until the end of string *line* is reached and at that moment *argv* is terminated with a zero.

For example, if the input line is a string as follows:

```
"cp  abc.CC   xyz.TT"
```

Function *parse()* will return array *argv[]* with the following content:



Function *execute()* takes array *argv[]*, treats it as a command line arguments with the program name in *argv[0]*, forks a child process, and executes the indicated program in that child process. While the child process is executing the command, the parent executes a *wait()*, waiting for the completion of the child. In this special case, the parent knows the child's process ID and therefore is able to wait a specific child to complete.

The main program is very simple. It prints out a command prompt, reads in a line, parses it using function *parse()*, and determines if the name is "exit". If it is "exit", use *exit()* to terminate the execution of this program; otherwise, the main uses *execute()* to execute the command.

1.4 The execlp() system call

The following code demonstrates the use of system call: execlp(). Note carefully which calls to execlp() will get executed and which will note from the following code.

```
/* EXECUTING AN APPLICATION USING EXEC COMMANDS

The execlp() family of commands can be used to execute an application from a process.
The system call execlp() replaces the executing process by a new process image which executes
the application specified as its parameter. Arguments can also be specified.
Refer to online man pages. */

#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/types.h>

main()
{
/*  "cal" is an application which shows the calendar of the current year and month.
    "cal" with an argument specifying year (for example "cal 1999") shows the calendar of that year.
    Try out the "cal" command from your command prompt.

    Here we execute "cal 2015" using the execlp() system call.
    Note that we specify "cal" in the first two arguments.
    The reason for this is given in the online man pages for execlp() */

        execlp("ps","ps","-a",NULL);
```

```
/* The execlp() system call does not return. Note that following statement will not be executed. */
        execlp("cal","cal","2015",NULL);
        printf("The execlp invoking cal is not executed if execlp invoking ps succeeds.\n");

/* The execlp() system call does not return. Note that following statement will not be executed. */
    printf("This statement is not executed if execlp succeeds.\n");
}
```

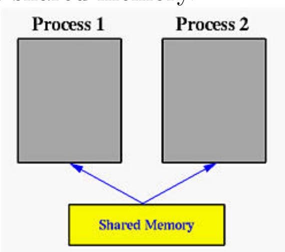
2 Shared Memory

In this section, we will discuss the basics of shared memory along with related system calls useful in allocation and deallocation of shared memory in user programs.

2.1 What is Shared Memory?

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.

In the next few sections, we shall describe these system calls and their usage.

2.2 Keys

Unix/Linux 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.

2.3 Shared Memory and Related System Calls

2.3.1 Requesting for a Shared Memory Segment

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 tradition, IPC_CREAT is correct and IPC_CREATE is not!!!

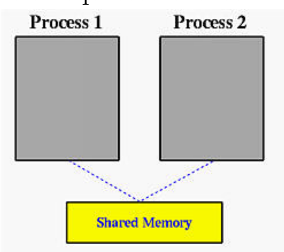
If shmget() can successfully get the requested shared memory, its function value is a non-negative integer, the shared memory ID; otherwise, the function value is negative. The following is a server example of requesting a private shared memory of four integers:

```
#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. Don't change 0666 to 666 as 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().



2.3.2 Attaching a Shared Memory Segment to an Address Space

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.

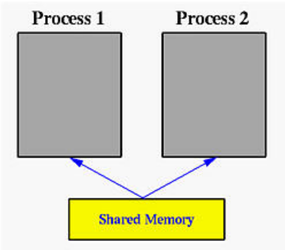
```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>
#include  <stdio.h>

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 (shm_ptr == (int *)-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 (shm_ptr == (int *)-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).



2.3.3 Detaching and Removing a Shared Memory Segment

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().

3 Few Examples

This section presents two examples to showcase how the system calls discussed earlier are put in practice for building application.

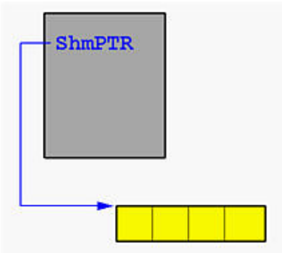
3.1 Communicating Between Parent and Child

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.

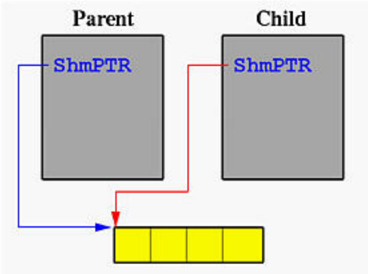
```
/* filename: shm-01.c */
#include  <stdio.h>
#include  <stdlib.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>

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 (ShmPTR == (int *)-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.

3.2 Communicating Between Two Separate Processes

In the following 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.

```
/* filename: 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.

```
/* filename: server.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(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 (ShmPTR == (struct Memory *)-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.

```
/* filename: 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 (ShmPTR == (struct Memory *)-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.

4 Background Processes and Shared Memory Status

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/Linux 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.

4.1 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 un-wanted 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.
