Subsections

IPC:Shared Memory

Shared Memory is an efficeint means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.

In the Solaris 2.x operating system, the most efficient way to implement shared memory applications is to rely on the `mmap()` function and on the system's native virtual memory facility. Solaris 2.x also supports System V shared memory, which is another way to let multiple processes attach a segment of physical memory to their virtual address spaces. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent inconsistencies and collisions.

A process creates a shared memory segment using `shmget()|`. The original owner of a shared memory segment can assign ownership to another user with `shmctl()`. It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl()`. Once created, a shared segment can be attached to a process address space using `shmat()`. It can be detached using `shmdt()` (see `shmop()`). The attaching process must have the appropriate permissions for `shmat()`. Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structures and prototypews can be found in <`sys/shm.h`>.

Accessing a Shared Memory Segment

`shmget()` is used to obtain access to a shared memory segment. It is prottyped by:

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

The `key` argument is a access value associated with the semaphore ID. The `size` argument is the size in bytes of the requested shared memory. The `shmflg` argument specifies the initial

access permissions and creation control flags.

When the call succeeds, it returns the shared memory segment ID. This call is also used to get the ID of an existing shared segment (from a process requesting sharing of some existing memory portion).

The following code illustrates `shmget()`:

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

...

key_t key; /* key to be passed to shmget() */
int shmflg; /* shmflg to be passed to shmget() */
int shmid; /* return value from shmget() */
int size; /* size to be passed to shmget() */

...

key = ...
size = ...
shmflg) = ...

if ((shmid = shmget (key, size, shmflg)) == -1) {
   perror("shmget: shmget failed"); exit(1); } else {
   (void) fprintf(stderr, "shmget: shmget returned %d\n", shmid);
   exit(0);
  }
...
```

Controlling a Shared Memory Segment

`shmctl()` is used to alter the permissions and other characteristics of a shared memory segment. It is prototyped as follows:

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The process must have an effective `shmid` of owner, creator or superuser to perform this command. The `cmd` argument is one of following control commands:

SHM_LOCK
      -- Lock the specified shared memory segment in memory. The process must have the effective ID of superuser to perform this command.
SHM_UNLOCK
      -- Unlock the shared memory segment. The process must have the effective ID of superuser to perform this command.
IPC_STAT
      -- Return the status information contained in the control structure and place it in the buffer pointed to by buf. The process must have read permission on the segment to perform this command.
IPC_SET
      -- Set the effective user and group identification and access permissions. The process must have an effective ID of owner, creator or superuser to perform this command.
IPC_RMID
      -- Remove the shared memory segment.

The `buf` is a sructure of type `struct shmid_ds` which is defined in `<sys/shm.h>`

The following code illustrates `shmctl()`:

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

...

int cmd; /* command code for shmctl() */
int shmid; /* segment ID */
struct shmid_ds shmid_ds; /* shared memory data structure to
                              hold results */
...

shmid = ...
cmd = ...
if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
   perror("shmctl: shmctl failed");
   exit(1);
  }
...
```

Attaching and Detaching a Shared Memory Segment

shmat() and shmdt() are used to attach and detach shared memory segments. They are prototypes as follows:

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

```
int shmdt(const void *shmaddr);
```

shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid. shmdt() detaches the shared memory segment located at the address indicated by shmaddr

. The following code illustrates calls to shmat() and shmdt():

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

static struct state { /* Internal record of attached segments. */
        int shmid; /* shmid of attached segment */
        char *shmaddr; /* attach point */
        int shmflg; /* flags used on attach */
       } ap[MAXnap]; /* State of current attached segments. */
int nap; /* Number of currently attached segments. */

...

char *addr; /* address work variable */
register int i; /* work area */
register struct state *p; /* ptr to current state entry */
...

p = &ap[nap++];
p->shmid = ...
p->shmaddr = ...
p->shmflg = ...

p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
if(p->shmaddr == (char *)-1) {
    perror("shmop: shmat failed");
    nap--;
  } else
```

```
    (void) fprintf(stderr, "shmop: shmat returned %#8.8x\n",
p->shmaddr);

...
i = shmdt(addr);
if(i == -1) {
    perror("shmop: shmdt failed");
    } else {
  (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);

for (p = ap, i = nap; i--; p++)
  if (p->shmaddr == addr) *p = ap[--nap];

}
...
```

Example two processes comunicating via shared memory: `shm_server.c`, `shm_client.c`

We develop two programs here that illustrate the passing of a simple piece of memery (a string) between the processes if running simulatenously:

shm_server.c
      -- simply creates the string and shared memory portion.
shm_client.c
      -- attaches itself to the created shared memory portion and uses the string (`printf`.

The code listings of the 2 programs no follow:

shm_server.c

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

#define SHMSZ     27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We'll name our shared memory segment
     * "5678".
     */
    key = 5678;

    /*
     * Create the segment.
     */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
```

```
 */
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
}

/*
 * Now put some things into the memory for the
 * other process to read.
 */
s = shm;

for (c = 'a'; c <= 'z'; c++)
    *s++ = c;
*s = NULL;

/*
 * Finally, we wait until the other process
 * changes the first character of our memory
 * to '*', indicating that it has read what
 * we put there.
 */
while (*shm != '*')
    sleep(1);

exit(0);
}
```

shm_client.c

```
/*
 * shm-client - client program to demonstrate shared memory.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We need to get the segment named
     * "5678", created by the server.
     */
    key = 5678;

    /*
     * Locate the segment.
     */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
```

```
    /*
     * Now read what the server put in the memory.
     */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
}
```

POSIX Shared Memory

POSIX shared memory is actually a variation of mapped memory. The major differences are to use `shm_open()` to open the shared memory object (instead of calling `open()`) and use `shm_unlink()` to close and delete the object (instead of calling `close()` which does not remove the object). The options in `shm_open()` are substantially fewer than the number of options provided in `open()`.

Mapped memory

In a system with fixed memory (non-virtual), the address space of a process occupies and is limited to a portion of the system's main memory. For example, in Solaris 2.x virtual memory the actual address space of a process occupies a file in the swap partition of disk storage (the file is called the backing store). Pages of main memory buffer the active (or recently active) portions of the process address space to provide code for the CPU(s) to execute and data for the program to process.

A page of address space is loaded when an address that is not currently in memory is accessed by a CPU, causing a page fault. Since execution cannot continue until the page fault is resolved by reading the referenced address segment into memory, the process sleeps until the page has been read. The most obvious difference between the two memory systems for the application developer is that virtual memory lets applications occupy much larger address spaces. Less obvious advantages of virtual memory are much simpler and more efficient file I/O and very efficient sharing of memory between processes.

Address Spaces and Mapping

Since backing store files (the process address space) exist only in swap storage, they are not included in the UNIX named file space. (This makes backing store files inaccessible to other processes.) However, it is a simple extension to allow the logical insertion of all, or part, of one, or more, named files in the backing store and to treat the result as a single address space. This is called mapping. With mapping, any part of any readable or writable file can be logically included in a process's address space. Like any other portion of the process's address space, no page of the file is not actually loaded into memory until a page fault forces this action. Pages of memory are written to the file only if their contents have been modified. So, reading from and writing to files is completely automatic and very efficient. More than one process can map a single named file. This provides very efficient memory sharing between processes. All or part of other files can also be shared between processes.

Not all named file system objects can be mapped. Devices that cannot be treated as storage, such as terminal and network device files, are examples of objects that cannot be mapped. A process address space is defined by all of the files (or portions of files) mapped into the address space. Each mapping is sized and aligned to the page boundaries of the system on which the process is executing. There is no memory associated with processes themselves.

A process page maps to only one object at a time, although an object address may be the subject of many process mappings. The notion of a "page" is not a property of the mapped object. Mapping an object only provides the potential for a process to read or write the object's contents. Mapping makes the object's contents directly addressable by a process. Applications can access the storage resources they use directly rather than indirectly through read and write. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (single-step updates rather than the read, modify buffer, write cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

Because the file system name space includes any directory trees that are connected from other systems via NFS, any networked file can also be mapped into a process's address space.

Coherence

Whether to share memory or to share data contained in the file, when multiple process map a file simultaneously there may be problems with simultaneous access to data elements. Such processes can cooperate through any of the synchronization mechanisms provided in Solaris 2.x. Because they are very light weight, the most efficient synchronization mechanisms in Solaris 2.x are the threads library ones.

Creating and Using Mappings

mmap() establishes a mapping of a named file system object (or part of one) into a process address space. It is the basic memory management function and it is very simple.

- First open() the file, then
- mmap() it with appropriate access and sharing options
- Away you go.

mmap is prototypes as follows:

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

caddr_t mmap(caddr_t addr, size_t len, int prot, int flags,
        int fildes, off_t off);
```

The mapping established by mmap() replaces any previous mappings for specified address range. The flags MAP_SHARED and MAP_PRIVATE specify the mapping type, and one of them must be specified. MAP_SHARED specifies that writes modify the mapped object. No further operations on the object are needed to make the change. MAP_PRIVATE specifies that an initial write to the mapped area creates a copy of the page and all writes reference the copy. Only modified pages are copied.

A mapping type is retained across a fork(). The file descriptor used in a mmap call need not be kept open after the mapping is established. If it is closed, the mapping remains until the mapping is undone by munmap() or be replacing in with a new mapping. If a mapped file is shortened by a call to truncate, an access to the area of the file that no longer exists causes a SIGBUS signal.

The following code fragment demonstrates a use of this to create a block of scratch storage in a program, at an address that the system chooses.:

```
int fd;
caddr_t result;
if ((fd = open("/dev/zero", O_RDWR)) == -1)
    return ((caddr_t)-1);

result = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
(void) close(fd);
```

Other Memory Control Functions

`int mlock(caddr_t addr, size_t len)` causes the pages in the specified address range to be locked in physical memory. References to locked pages (in this or other processes) do not result in page faults that require an I/O operation. This operation ties up physical resources and can disrupt normal system operation, so, use of `mlock()` is limited to the superuser. The system lets only a configuration dependent limit of pages be locked in memory. The call to mlock fails if this limit is exceeded.

`int munlock(caddr_t addr, size_t len)` releases the locks on physical pages. If multiple `mlock()` calls are made on an address range of a single mapping, a single `munlock` call is release the locks. However, if different mappings to the same pages are mlocked, the pages are not unlocked until the locks on all the mappings are released. Locks are also released when a mapping is removed, either through being replaced with an mmap operation or removed with `munmap`. A lock is transferred between pages on the *copy-on-write* event associated with a `MAP_PRIVATE` mapping, thus locks on an address range that includes `MAP_PRIVATE` mappings will be retained transparently along with the copy-on-write redirection (see mmap above for a discussion of this redirection)

`int mlockall(int flags)` and `int munlockall(void)` are similar to `mlock()` and `munlock()`, but they operate on entire address spaces. `mlockall()` sets locks on all pages in the address space and `munlockall()` removes all locks on all pages in the address space, whether established by mlock or mlockall.

`int msync(caddr_t addr, size_t len, int flags)` causes all modified pages in the specified address range to be flushed to the objects mapped by those addresses. It is similar to `fsync()` for files.

`long sysconf(int name)` returns the system dependent size of a memory page. For portability, applications should not embed any constants specifying the size of a page. Note that it is not unusual for page sizes to vary even among implementations of the same instruction set.

`int mprotect(caddr_t addr, size_t len, int prot)` assigns the specified protection to all pages in the specified address range. The protection cannot exceed the permissions allowed on the underlying object.

`int brk(void *endds)` and `void *sbrk(int incr)` are called to add storage to the data segment of a process. A process can manipulate this area by calling `brk()` and `sbrk()`. `brk()` sets the system idea of the lowest data segment location not used by the caller to addr (rounded up to the next multiple of the system page size). `sbrk()` adds incr bytes to the caller data space and returns a pointer to the start of the new data area.

Some further example shared memory programs

The following suite of programs can be used to investigate interactively a variety of shared ideas (see exercises below).

The semaphore **must** be initialised with the `shmget.c` program. The effects of controlling shared memory and accessing can be investigated with `shmctl.c` and `shmop.c` respectively.

`shmget.c`:Sample Program to Illustrate shmget()

```
/*
 * shmget.c: Illustrate the shmget() function.
 *
 * This is a simple exerciser of the shmget() function. It
prompts
 * for the arguments, makes the call, and reports the results.
 */

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

extern void     exit();
extern void     perror();

main()
{
 key_t  key;   /* key to be passed to shmget() */
 int  shmflg;   /* shmflg to be passed to shmget() */
 int  shmid;   /* return value from shmget() */
 int  size;   /* size to be passed to shmget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
     "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the key. */
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 /* Get the size of the segment. */
 (void) fprintf(stderr, "Enter size: ");
 (void) scanf("%i", &size);

 /* Get the shmflg value. */
 (void) fprintf(stderr,
     "Expected flags for the shmflg argument are:\n");
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n",
IPC_CREAT);
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter shmflg: ");
 (void) scanf("%i", &shmflg);

 /* Make the call and report the results. */
 (void) fprintf(stderr,
     "shmget: Calling shmget(%#lx, %d, %#o)\n",
     key, size, shmflg);
 if ((shmid = shmget (key, size, shmflg)) == -1) {
```

```
   perror("shmget: shmget failed");
   exit(1);
  } else {
   (void) fprintf(stderr,
      "shmget: shmget returned %d\n", shmid);
   exit(0);
  }
 }
```

shmctl.c: Sample Program to Illustrate `shmctl()`

```
/*
 * shmctl.c: Illustrate the shmctl() function.
 *
 * This is a simple exerciser of the shmctl() function. It lets you
 * to perform one control operation on one shared memory segment.
 * (Some operations are done for the user whether requested or
not.
 * It gives up immediately if any control operation fails. Be
careful
 * not to set permissions to preclude read permission; you won't
be
 *able to reset the permissions with this code if you do.)
 */

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>
#include   <time.h>
static void   do_shmctl();
extern void   exit();
extern void   perror();

main()
{
 int  cmd;  /* command code for shmctl() */
 int  shmid;  /* segment ID */
 struct shmid_ds  shmid_ds;      /* shared memory data structure to
         hold results */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
     "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get shmid and cmd. */
 (void) fprintf(stderr,
     "Enter the shmid for the desired segment: ");
 (void) scanf("%i", &shmid);
 (void) fprintf(stderr, "Valid shmctl cmd values are:\n");
 (void) fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
 (void) fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
 (void) fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
 (void) fprintf(stderr, "Enter the desired cmd value: ");
 (void) scanf("%i", &cmd);

 switch (cmd) {
  case IPC_STAT:
   /* Get shared memory segment status. */
   break;
  case IPC_SET:
   /* Set owner UID and GID and permissions. */
   /* Get and print current values. */
```

```
        do_shmctl(shmid, IPC_STAT, &shmid_ds);
        /* Set UID, GID, and permissions to be loaded. */
        (void) fprintf(stderr, "\nEnter shm_perm.uid: ");
        (void) scanf("%hi", &shmid_ds.shm_perm.uid);
        (void) fprintf(stderr, "Enter shm_perm.gid: ");
        (void) scanf("%hi", &shmid_ds.shm_perm.gid);
        (void) fprintf(stderr,
         "Note: Keep read permission for yourself.\n");
        (void) fprintf(stderr, "Enter shm_perm.mode: ");
        (void) scanf("%hi", &shmid_ds.shm_perm.mode);
        break;
     case IPC_RMID:
        /* Remove the segment when the last attach point is
           detached. */
        break;
     case SHM_LOCK:
        /* Lock the shared memory segment. */
        break;
     case SHM_UNLOCK:
        /* Unlock the shared memory segment. */
        break;
     default:
        /* Unknown command will be passed to shmctl. */
        break;
    }
    do_shmctl(shmid, cmd, &shmid_ds);
    exit(0);
}

/*
 * Display the arguments being passed to shmctl(), call shmctl(),
 * and report the results. If shmctl() fails, do not return; this
 * example doesn't deal with errors, it just reports them.
 */
static void
do_shmctl(shmid, cmd, buf)
int    shmid,    /* attach point */
    cmd;    /* command code */
struct shmid_ds    *buf;    /* pointer to shared memory data structure */
{
 register int    rtrn;  /* hold area */

 (void) fprintf(stderr, "shmctl: Calling shmctl(%d, %d,
buf)\n",
   shmid, cmd);
 if (cmd == IPC_SET) {
   (void) fprintf(stderr, "\tbuf->shm_perm.uid == %d\n",
      buf->shm_perm.uid);
   (void) fprintf(stderr, "\tbuf->shm_perm.gid == %d\n",
      buf->shm_perm.gid);
   (void) fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n",
      buf->shm_perm.mode);
 }
 if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {
  perror("shmctl: shmctl failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
      "shmctl: shmctl returned %d\n", rtrn);
 }
 if (cmd != IPC_STAT && cmd != IPC_SET)
   return;

 /* Print the current status. */
 (void) fprintf(stderr, "\nCurrent status:\n");
 (void) fprintf(stderr, "\tshm_perm.uid = %d\n",
     buf->shm_perm.uid);
 (void) fprintf(stderr, "\tshm_perm.gid = %d\n",
     buf->shm_perm.gid);
```

```
(void) fprintf(stderr, "\tshm_perm.cuid = %d\n",
    buf->shm_perm.cuid);
(void) fprintf(stderr, "\tshm_perm.cgid = %d\n",
    buf->shm_perm.cgid);
(void) fprintf(stderr, "\tshm_perm.mode = %#o\n",
    buf->shm_perm.mode);
(void) fprintf(stderr, "\tshm_perm.key = %#x\n",
    buf->shm_perm.key);
(void) fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
(void) fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
(void) fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
(void) fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
(void) fprintf(stderr, "\tshm_atime = %s",
  buf->shm_atime ? ctime(&buf->shm_atime) : "Not Set\n");
(void) fprintf(stderr, "\tshm_dtime = %s",
  buf->shm_dtime ? ctime(&buf->shm_dtime) : "Not Set\n");
(void) fprintf(stderr, "\tshm_ctime = %s",
    ctime(&buf->shm_ctime));
}
```

shmop.c: Sample Program to Illustrate shmat() and shmdt()

```
/*
 * shmop.c: Illustrate the shmat() and shmdt() functions.
 *
 * This is a simple exerciser for the shmat() and shmdt() system
 * calls. It allows you to attach and detach segments and to
 * write strings into and read strings from attached segments.
 */

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

#define   MAXnap   4 /* Maximum number of concurrent attaches. */

static   ask();
static void   catcher();
extern void   exit();
static   good_addr();
extern void   perror();
extern char   *shmat();

static struct state    { /* Internal record of currently attached
segments. */
 int  shmid;   /* shmid of attached segment */
 char  *shmaddr;   /* attach point */
 int  shmflg;   /* flags used on attach */
} ap[MAXnap];      /* State of current attached segments. */

static int    nap;  /* Number of currently attached segments. */
static jmp_buf   segvbuf;   /* Process state save area for SIGSEGV
        catching. */

main()
{
 register int    action;   /* action to be performed */
 char    *addr;   /* address work area */
 register int    i;   /* work area */
 register struct state   *p;     /* ptr to current state entry */
 void   (*savefunc)();  /* SIGSEGV state hold area */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
```

```
(void) fprintf(stderr,
 "\t0x... is interpreted as hexadecimal,\n");
(void) fprintf(stderr, "\t0... is interpreted as octal,\n");
(void) fprintf(stderr, "\totherwise, decimal.\n");
while (action = ask()) {
 if (nap) {
  (void) fprintf(stderr,
     "\nCurrently attached segment(s):\n");
  (void) fprintf(stderr, " shmid address\n");
  (void) fprintf(stderr, "------ ----------\n");
  p = &ap[nap];
  while (p-- != ap) {
   (void) fprintf(stderr, "%6d", p->shmid);
   (void) fprintf(stderr, "%#11x", p->shmaddr);
   (void) fprintf(stderr, " Read%s\n",
    (p->shmflg & SHM_RDONLY) ?
    "-Only" : "/Write");
  }
 } else
  (void) fprintf(stderr,
   "\nNo segments are currently attached.\n");
 switch (action) {
 case 1:   /* Shmat requested. */
  /* Verify that there is space for another attach. */
  if (nap == MAXnap) {
   (void) fprintf(stderr, "%s %d %s\n",
      "This simple example will only allow",
      MAXnap, "attached segments.");
   break;
  }
  p = &ap[nap++];
  /* Get the arguments, make the call, report the
   results, and update the current state array. */
  (void) fprintf(stderr,
   "Enter shmid of segment to attach: ");
  (void) scanf("%i", &p->shmid);

  (void) fprintf(stderr, "Enter shmaddr: ");
  (void) scanf("%i", &p->shmaddr);
  (void) fprintf(stderr,
   "Meaningful shmflg values are:\n");
  (void) fprintf(stderr, "\tSHM_RDONLY = \t%#8.8o\n",
   SHM_RDONLY);
  (void) fprintf(stderr, "\tSHM_RND = \t%#8.8o\n",
   SHM_RND);
  (void) fprintf(stderr, "Enter shmflg value: ");
  (void) scanf("%i", &p->shmflg);

  (void) fprintf(stderr,
   "shmop: Calling shmat(%d, %#x, %#o)\n",
   p->shmid, p->shmaddr, p->shmflg);
  p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
  if(p->shmaddr == (char *)-1) {
   perror("shmop: shmat failed");
   nap--;
  } else {
   (void) fprintf(stderr,
    "shmop: shmat returned %#8.8x\n",
    p->shmaddr);
  }
  break;

 case 2:   /* Shmdt requested. */
  /* Get the address, make the call, report the results,
   and make the internal state match. */
  (void) fprintf(stderr,
   "Enter detach shmaddr: ");
  (void) scanf("%i", &addr);
```

```
    i = shmdt(addr);
    if(i == -1) {
     perror("shmop: shmdt failed");
    } else {
     (void) fprintf(stderr,
      "shmop: shmdt returned %d\n", i);
     for (p = ap, i = nap; i--; p++) {
      if (p->shmaddr == addr)
       *p = ap[--nap];
     }
    }
    break;
   case 3: /* Read from segment requested. */
    if (nap == 0)
     break;

    (void) fprintf(stderr, "Enter address of an %s",
     "attached segment: ");
    (void) scanf("%i", &addr);

    if (good_addr(addr))
     (void) fprintf(stderr, "String @ %#x is `%s'\n",
      addr, addr);
    break;

   case 4: /* Write to segment requested. */
    if (nap == 0)
     break;

    (void) fprintf(stderr, "Enter address of an %s",
     "attached segment: ");
    (void) scanf("%i", &addr);

    /* Set up SIGSEGV catch routine to trap attempts to
     write into a read-only attached segment. */
    savefunc = signal(SIGSEGV, catcher);

    if (setjmp(segvbuf)) {
     (void) fprintf(stderr, "shmop: %s: %s\n",
      "SIGSEGV signal caught",
      "Write aborted.");
    } else {
     if (good_addr(addr)) {
      (void) fflush(stdin);
      (void) fprintf(stderr, "%s %s %#x:\n",
       "Enter one line to be copied",
       "to shared segment attached @",
       addr);
      (void) gets(addr);
     }
    }
    (void) fflush(stdin);

    /* Restore SIGSEGV to previous condition. */
    (void) signal(SIGSEGV, savefunc);
    break;
   }
  }
 exit(0);
 /*NOTREACHED*/
}
/*
** Ask for next action.
*/
static
ask()
{
 int  response;   /* user response */
 do {
```

```
      (void) fprintf(stderr, "Your options are:\n");
      (void) fprintf(stderr, "\t^D = exit\n");
      (void) fprintf(stderr, "\t 0 = exit\n");
      (void) fprintf(stderr, "\t 1 = shmat\n");
      (void) fprintf(stderr, "\t 2 = shmdt\n");
      (void) fprintf(stderr, "\t 3 = read from segment\n");
      (void) fprintf(stderr, "\t 4 = write to segment\n");
      (void) fprintf(stderr,
        "Enter the number corresponding to your choice: ");

      /* Preset response so "^D" will be interpreted as exit. */
      response = 0;
      (void) scanf("%i", &response);
  } while (response < 0 || response > 4);
  return (response);
}
/*
** Catch signal caused by attempt to write into shared memory
segment
** attached with SHM_RDONLY flag set.
*/
/*ARGSUSED*/
static void
catcher(sig)
{
 longjmp(segvbuf, 1);
 /*NOTREACHED*/
}
/*
** Verify that given address is the address of an attached
segment.
** Return 1 if address is valid; 0 if not.
*/
static
good_addr(address)
char *address;
{
 register struct state           *p;   /* ptr to state of attached
segment */

 for (p = ap; p != &ap[nap]; p++)
   if (p->shmaddr == address)
     return(1);
 return(0);
}
```

Exercises

Exercise 12771

Write 2 programs that will communicate via shared memory and semaphores. Data will be exchanged via memory and semaphores will be used to synchronise and notify each process when operations such as memory loaded and memory read have been performed.

Exercise 12772

Compile the programs shmget.c, shmctl.c and shmop.c and then

- investigate and understand fully the operations of the flags (access, creation *etc.* permissions) you can set interactively in the programs.
- Use the prgrams to:
  - Exchange data between two processe running as shmop.c.
  - Inquire about the state of shared memory with shmctl.c.
  - Use semctl.c to lock a shared memory segment.

○ Use `semctl.c` to delete a shared memory segment.

Exercise 12773

Write 2 programs that will communicate via mapped memory.

Dave Marshall
1/5/1999