Shared Memory

Memory mapped files

Shared Memory

- Introduction
- Creating a Shared Memory Segment
- Shared Memory Control
- Shared Memory Operations
- Using a File as Shared Memory

Introduction

- Shared memory allows multiple processes to share virtual memory space.
- This is the fastest but not necessarily the easiest (synchronization-wise) way for processes to communicate with one another.
- In general, one process creates or allocates the shared memory segment.
- The size and access permissions for the segment are set when it is created.
- The process then attaches the shared segment, causing it to be mapped into its current data space.

Introduction

- If needed, the creating process then initializes the shared memory.
- Once created, and if permissions permit, other processes can gain access to the shared memory segment and map it into their data space.
- Each process accesses the shared memory relative to its attachment address.
- While the data that these processes are referencing is in common, each process uses different attachment address values.

Introduction

- For each process involved, the mapped memory appears to be no different from any other of its memory addresses.

Creating a Shared Memory Segment

- The shmget system call is used to create the shared memory segment and generate the associated system data structure or to gain access to an existing segment.
- The shared memory segment and the system data structure are identified by a unique shared memory identifier that the shmget system call returns.

(Table 8.1)

Creating a Shared Memory Segment

Include File(s)	<sys/ipc.h> <sys/shm.h>	Manual Section	2
Summary	int shmget(key_t key, int size,int shmflg);		
Return	Success	Failure	Sets errno
	Shared memory identifier.	-1	Yes

Table 8.1. Summary of the shmget System Call.

Creation - shmget

- First parameter is an integer key specifies which segment to create
  - Unrelated processes can access the same shared segment by specifying the same key value
  - Other processes may have also chosen the same fixed key (conflict). Using the special constant IPC_PRIVATE as the key value guarantees that a brand new memory segment is created.
- Second parameter specifies the number of bytes in the segment
  - number of actually allocated bytes is rounded up to an integral multiple of the page size
- The third parameter is the bitwise or of flag values that specify options to shmget.

The flag values include these

Creation - shmget

- The third parameter is the bitwise or of flag values that specify options to shmget. The flag values include these
  - IPC_CREAT—a new segment should be created.
  - IPC_EXCL—This flag, which is always used with IPC_CREAT, causes shmget to fail if a segment key is specified that already exists. Therefore, it arranges for the calling process to have an “exclusive” segment. If this flag is not given and the key of an existing segment is used, shmget returns the existing segment instead of creating a new one.
  - Mode flags—This value is made of 9 bits indicating permissions granted to owner, group, and world to control access to the segment. Execution bits are ignored. An easy way to specify permissions is to use the constants defined in <sys/stat.h> and documented in the section 2 stat man page.1 For example, S_IRUSR and S_IWUSR specify read and write permissions for the owner of the shared memory segment, and S_IROTH and S_IWOTH specify read and write permissions for others.

Shmget - example

- For example, this invocation of `shmget` creates a new shared memory segment (or access to an existing one, if `shm_key` is already used) that's readable and writeable to the owner but not other users

```
int segment_id = shmget (shm_key, getpagesize (),
IPC_CREAT | S_IRUSR | S_IWUSER);
```

If the call succeeds, `shmget` returns a segment identifier. If the shared memory segment already exists, the access permissions are verified and a check is made to ensure that the segment is not marked for destruction.

Creating a Shared Memory Segment

- The shmget system call creates a new shared memory segment if
  - The value for its first argument, key, is the symbolic constant IPC_PRIVATE, or
  - the value key is not associated with an existing shared memory identifier and the IPC_CREAT flag is set as part of the shmflg argument or
  - the value key is not associated with an existing shared memory identifier and the IPC_CREAT along with the IPC_EXCL flag have been set as part of the shmflg argument.

Creating a Shared Memory Segment

- As with previous IPC system calls for message queues and semaphores, the ftok library function can be used to generate a key value.
- The argument size determines the size in bytes of the shared memory segment.
- If we are using shmget to access an existing shared memory segment, size can be set to 0, as the segment size is set by the creating process.

Creating a Shared Memory Segment

- The last argument for shmget, shmflg, is used to indicate segment creation conditions (e.g., IPC_CREAT, IPC_EXCL) and access permissions (stored in the low order 9 bits of shmflg).
- At this time the system does not use the execute permission settings.
- To specify creation conditions along with access permissions, the individual items are bitwise ORed.

Creating a Shared Memory Segment

- The shmget system call does not entitle the creating process to actually use the allocated memory.
- It merely reserves the requested memory.
- To be used by the process, the allocated memory must be attached to the process using a separate system call.

Creating a Shared Memory Segment

- If shmget is successful in allocating a shared memory segment, it returns an integer shared memory identifier.
- If shmget fails, it returns a value of -1 and sets the value in errno to indicate the specific error condition.

Shared Memory Control

- The shmctl system call permits the user to perform a number of generalized control operations on an existing shared memory segment and on the system shared memory data structure.

Shared Memory Control

Include File(s)	<sys/ipc.h> <sys/shm.h>	Manual Section	2
Summary	int shmctl(int shmid, int cmd, struct shmid_ds *buf);		
Return	Success	Failure	Sets errno
	0	-1	Yes

Table 8.4. Summary of the shmctl System Call.

Shared Memory Control

- There are three arguments for the shmctl system call:
  - The first, shmid, is a valid shared memory segment identifier generated by a prior shmget system call.
  - The second argument, cmd, specifies the operation shmctl is to perform.
  - The third argument, buf, is a reference to a structure of the type shmid_ds.

Shared Memory Control

- If shmctl is successful, it returns a value of 0; otherwise, it returns a value of -1 and sets the value in errno to indicate the specific error condition.

Shared Memory Operations

- There are two shared memory operation system calls.
- The first, shmat, is used to attach (map) the referenced shared memory segment into the calling process's data segment.

(Table 8.6.)

Shared Memory Operations

Include File(s)	<sys/ipc.h> <sys/shm.h>	Manual Section	2
Summary	void *shmat(int shmid, const void *shmaddr, int shmflg);		
Return	Success	Failure	Sets errno
	Reference to the data segment	-1	Yes

Table 8.6. Summary of the shmat System Call.

Shared Memory Operations

- The first argument to shmat, shmid, is a valid shared memory identifier.
- The second argument, shmaddr, allows the calling process some flexibility in assigning the location of the shared memory segment.
  - If a nonzero value is given, shmat uses this as the attachment address for the shared memory segment.
  - If shmaddr is 0, the system picks the attachment address.
  - In most situations, it is advisable to use a value of 0 and have the system pick the address.

Shared Memory Operations

- The third argument, shmflg, is used to specify the access permissions for the shared memory segment and to request special attachment conditions, such as an aligned address or a read-only segment.
- The values of shmaddr and shmflg are used by the system to determine the attachment address.

Shared Memory Operations

- When shmat is successful, it returns the address of the actual attachment.
- If shmat fails, it returns a value of -1 and sets errno to indicate the source of the error.
- Remember that after a fork, the child inherits the attached shared memory segment(s).
- After an exec or an exit attached, shared memory segment(s) are detached but are not destroyed.

Shared Memory Operations

- The second shared memory operation, shmdt, is used to detach the calling process's data segment from the shared memory segment.

(Table 8.8.)

Shared Memory Operations

Include File(s)	<sys/types.h> <sys/shm.h>	Manual Section	2
Summary	int shmdt (const void *shmaddr);		
Return	Success	Failure	Sets errno
	0	-1	Yes

Table 8.8. Summary of the shmdt System Call

Shared Memory Operations

- The shmdt system call has one argument, shmaddr, which is a reference to an attached memory segment.
- If shmdt is successful in detaching the memory segment, it returns a value of 0.
- If the shmdt call fails, it returns a value of -1 and sets errno

Cycle of Usage

- To use a shared memory segment, one process must allocate the segment
- Then each process desiring to access the segment must attach the segment
- After finishing its use of the segment, each process detaches the segment
- At some point, one process must deallocate the segment

Example

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  /* Allocate a shared memory segment.  */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
                       IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

Example

```
/* Attach the shared memory segment.  */
shared_memory = (char*) shmat (segment_id, 0, 0);
printf ("shared memory attached at address %p\n", shared_memory);
/* Determine the segment's size.  */
shmctl (segment_id, IPC_STAT, &shmbuffer);
segment_size = shmbuffer.shm_segsz;
printf ("segment size: %d\n", segment_size);
/* Write a string to the shared memory segment.  */
sprintf (shared_memory, "Hello, world.");
/* Deatch the shared memory segment.  */
shmdt (shared_memory);
```

Example

```
  /* Reattach the shared memory segment, at a different address.  */
  shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
  printf ("shared memory reattached at address %p\n", shared_memory);
  /* Print out the string from shared memory.  */
  printf ("%s\n", shared_memory);
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Deallocate the shared memory segment.  */
  shmctl (segment_id, IPC_RMID, 0);

  return 0;
}
```

Mapped memory

- Mapped memory allows processes to communicate via a shared file
- There are technical differences w.r.t. shared segments:
  - Mapped memory can be used for IPC or to access the contents of a file
  - Mapped memory forms an association between file process's memory
- Linux splits the file into page-sized chunks and then copies them into virtual memory pages to make them available in a process's address space.
  - This permits fast access to files (likewise ordinary memory access)
  - Linux handles the writing back out to the file afterward

Using a File as Shared Memory

- mmap system call can be used to map a file to a process's virtual memory address space.
- In many ways mmap is more flexible than its shared memory system call counterpart.
- Once a mapping has been established, standard system calls rather than specialized system calls can be used to manipulate the shared memory object.
- Unlike memory, the contents of a file are nonvolatile and will remain available even after a system has been shut down (and rebooted).

(Table 8.11).

Using a File as Shared Memory

Include File(s)	<unistd.h> <sys/nman.h>	Manual Section	2
Summary	#ifdef _POSIX_MAPPED_FILES <-- 1 void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset); #endif (1)If _POSIX_MAPPED_FILES has been defined.		
Return	Success	Failure	Sets errno
	A pointer to the mapped area	MAP_FAILED ((void *) -1)	Yes

Table 8.11. Summary of the mmap System Call.

Using a File as Shared Memory

- The mmap system call requires six arguments.
  - The first, start, is the address for attachment. As with the shmat system call, this argument is most often set to 0, which directs the system to choose a valid attachment address.
  - The number of bytes to be attached is indicated by the second argument, length.
  - The third argument, prot, is used to set the type of access (protection) for the segment.

Using a File as Shared Memory

- The fifth argument, fd, is a valid open file descriptor. Once the mapping is established, the file can be closed.
- The sixth argument, offset, is used to set the starting position for the mapping.

- If the mmap system call is successful, it returns a reference to the mapped memory object.
- If the call fails, it returns the defined constant MAP_FAILED (which is actually the value -1 cast to a void *)

Flags

- The flag value is a bitwise “or” of these constraints:
  - MAP_FIXED—If you specify this flag, Linux uses the address you request to map the file rather than treating it as a hint. This address must be page-aligned.
  - MAP_PRIVATE—Writes to the memory range should not be written back to the attached file, but to a private copy of the file. No other process sees these writes. This mode may not be used with MAP_SHARED
  - MAP_SHARED—Writes are immediately reflected in the underlying file rather than buffering writes. Use this mode when using mapped memory for IPC.This mode may not be used with MAP_PRIVATE.
- If the call succeeds, it returns a pointer to the beginning of the memory. On failure, it returns MAP_FAILED.

Using a File as Shared Memory

- While the system will automatically unmap a region when a process terminates, the system call munmap can be used to explicitly unmap pages of memory.
- Pass it the start address and length of the mapped memory region

(Table 8.16)

Using a File as Shared Memory

Include File(s)	<unistd.h> <signal.h>	Manual Section	2
Summary	#ifdef _POSIX_MAPPED_FILES int munmap(void *start, size_t length); #endif		

Return	Success	Failure	Sets errno
	0	-1	Yes

Table 8.16. Summary of the munmap System Call.

Using a File as Shared Memory

- The munmap system call is passed the starting address of the memory mapping (argument start) and the size of the mapping (argument length).
- If the call is successful, it returns a value of 0.
- Future references to unmapped addresses generate a SIGVEGV signal.
- If the munmap system call fails, it returns the value -1 and sets the value in errno to EINVAL.

Example: write a random nnumber on a mm file

```
#include <stdlib.h> #include <stdio.h> #include <fcntl.h> #include <sys/mman.h> #include <sys/stat.h>
#include <time.h> #include <unistd.h>
#define FILE_LENGTH 0x100

/* Return a uniformly random number in the range [low,high].  */
int random_range (unsigned const low, unsigned const high)
{ unsigned const range = high - low + 1;
  return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));}

int main (int argc, char* const argv[])
{ int fd;
  void* file_memory;

  /* Seed the random number generator.  */
  srand (time (NULL));

  /* Prepare a file large enough to hold an unsigned integer.  */
  fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  lseek (fd, FILE_LENGTH+1, SEEK_SET);
  write (fd, "", 1);
  lseek (fd, 0, SEEK_SET);

  /* Create the memory-mapping.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
  close (fd);
  /* Write a random integer to memory-mapped area.  */
  sprintf((char*) file_memory, "%d\n", random_range (-100, 100));
  /* Release the memory (unnecessary since the program exits).  */
  munmap (file_memory, FILE_LENGTH);

  return 0; }
```

Example: read an integer from a mm file and double it

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;
  int integer;

  /* Open the file.  */
  fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
  /* Create the memory-mapping.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd, 0);
  close (fd);

  /* Read the integer, print it out, and double it.  */
  sscanf (file_memory, "%d", &integer);
  printf ("value: %d\n", integer);
  sprintf ((char*) file_memory, "%d\n", 2 * integer);
  /* Release the memory (unnecessary since the program exits).  */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

Usage

```
/tmp/integer-file.
% ./mmap-write /tmp/integer-file
% cat /tmp/integer-file
42
% ./mmap-read /tmp/integer-file
value: 42
% cat /tmp/integer-file
84
```

- Text 42 was written to the disk file without ever calling write, and was read back in again without calling read.
- In these sample programs write and read the integer as a string (using sprintf and sscanf) for demonstration purposes only—no need for the contents of a memory-mapped file to be text