Linux/UNIX System Programming

POSIX Shared Memory

Michael Kerrisk, man7.org © 2023

February 2023

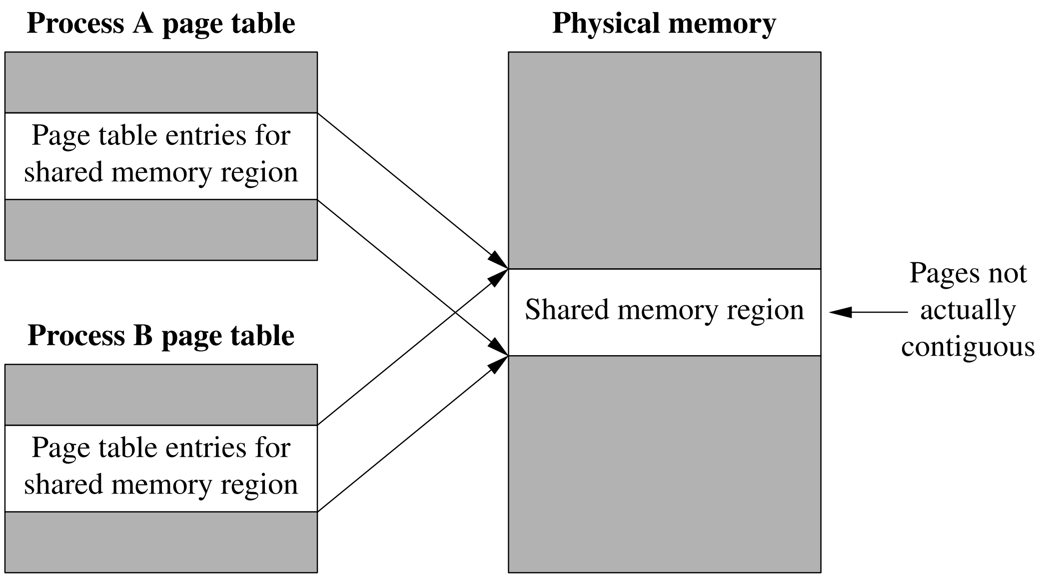
mtk@man7.org

Outline

Outline

Shared memory

- Data is exchanged by placing it in **memory pages shared by multiple processes**
  - Pages are **in user virtual address space** of each process



Shared memory

- **Data transfer is not mediated by kernel**
  - User-space copy makes data visible to other processes
    - ⇒ Very **fast IPC**
  - Compare with pipes and MQs:
    - Send requires copy from user to kernel memory
    - Receive requires copy from kernel to user memory
- But, **need to synchronize access** to shared memory
  - E.g., to prevent simultaneous updates
  - Commonly, semaphores are used

POSIX shared memory objects

- Implemented (on Linux) as files in a dedicated *tmpfs* filesystem
  - *tmpfs* == **virtual memory filesystem** that employs swap space when needed
- Objects have **kernel persistence**
  - Objects exist until explicitly deleted, or system reboots
  - Can map an object, change its contents, and unmap
  - Changes will be visible to next process that maps object
- **Accessibility**: user/group owner + permission mask

POSIX shared memory APIs

- *shm_open()*: open existing shared memory (SHM) object/create and open new SHM object
  - Returns file descriptor that refers to open object
- *ftruncate()*: set size of SHM object
- *mmap()*: map SHM object into caller's address space
- *close()*: close file descriptor returned by *shm_open()*
- *shm_unlink()*: remove SHM object name, mark for deletion once all processes have closed
- *munmap()*: unmap SHM object (or part thereof) from caller's address space
- Compile with `cc -lrt`
  - (No longer needed since glibc 2.34)
- *shm_overview(7)* man page

Outline

Creating/opening a shared memory object: *shm_open()*

```
#include <fcntl.h>           /* Defines O_* constants */
#include <sys/stat.h>        /* Defines mode constants */
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
```

- Creates and opens a new object, or opens an existing object
- *name*: name of object (`/somename`)
- Returns file descriptor on success, or –1 on error
  - This FD is used in subsequent APIs to refer to SHM
  - (The close-on-exec flag is automatically set for the FD)

[TLPI §54.2]

Creating/opening a shared memory object: *shm_open()*

```
#include <fcntl.h>          /* Defines O_* constants */
#include <sys/stat.h>       /* Defines mode constants */
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
```

oflag specifies flags controlling operation of call

- O_CREAT: create object if it does not already exist
- O_EXCL: (with O_CREAT) create object exclusively
    - Give error if object already exists
- O_RDONLY: open object for read-only access
- O_RDWR: open object for read-write access
    - NB: No O_WRONLY flag...
- O_TRUNC: truncate an existing object to zero length
    - Contents of existing object are destroyed

Creating/opening a shared memory object: *shm_open()*

```
#include <fcntl.h>          /* Defines O_* constants */
#include <sys/stat.h>       /* Defines mode constants */
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
```

- *mode*: permission bits for new object
    - RWX for user / group / other
    - ANDed against complement of process umask
    - ⚠ Required argument; specify as 0 if opening existing object

Sizing a shared memory object

- New SHM objects have length 0
- We must set size using *ftruncate(fd, size)*
  - Bytes in newly extended object are initialized to 0
  - If existing object is shrunk, truncated data is lost
  - Typically, *ftruncate()* is called before *mmap()*
    - But the calls can also be in the reverse order
- Can obtain size of existing object using *fstat(fd, &statbuf)*
  - *st_size* field of *stat* structure

Mapping a shared memory object: *mmap()*

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

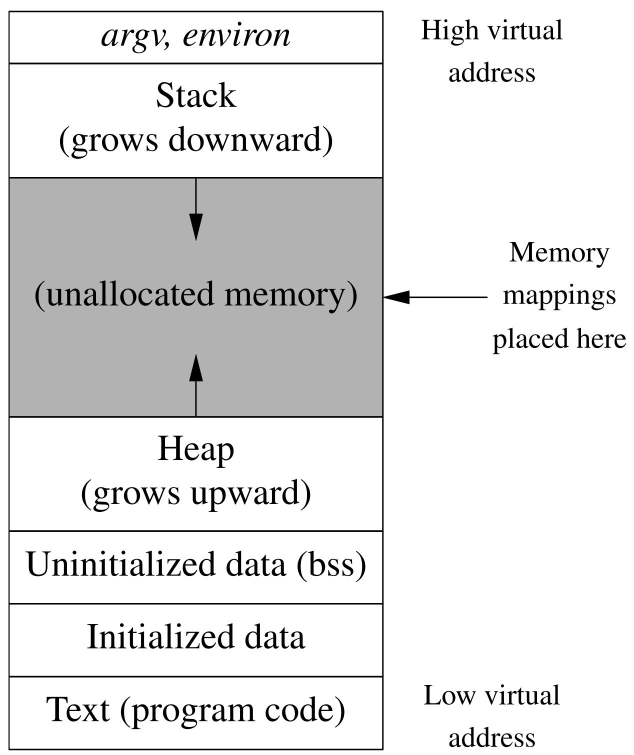
- **Complex, general-purpose API** for creating **memory mapping** in caller's virtual address space
  - 15+ bits employed in *flags*
  - See TLPI Ch. 49 and *mmap(2)*
- We consider only use with POSIX SHM
  - In practice, only a few decisions to make
    - Usually just *length*, *prot*, and maybe *offset*

Mapping a shared memory object: *mmap()*

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- *fd*: file descriptor specifying object to map
  - Use FD returned by *shm_open()*
  - **Note**: once *mmap()* returns, *fd* can already be closed without affecting the mapping
- *addr*: address at which to place mapping in caller's virtual address space
  - Let's look at a picture...

Process memory layout (simplified)



Mapping a shared memory object: *mmap()*

```
include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- *addr*: address at which to place mapping in caller's virtual address space
  - But, this address may already be occupied
    - Therefore, kernel takes *addr* as only a **hint**
    - **Ignored** if address is already occupied
  - *addr* == `NULL` ⇒ let system choose address
    - Normally use `NULL` for POSIX SHM objects
- *mmap()* returns address actually used for mapping
  - Treat this like a **normal C pointer**
- On error, *mmap()* returns `MAP_FAILED`

Mapping a shared memory object: *mmap()*

```
include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- *length*: size of mapping
  - Normally should be $\leq$ size of SHM object
  - System rounds up to multiple of system page size
    - *sysconf(_SC_PAGESIZE)*
- *offset*: starting point of mapping in underlying file or SHM object
  - Must be multiple of system page size
  - Commonly specified as 0 (map from start of object)

Mapping a shared memory object: *mmap()*

```
include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- *prot*: memory protections
  - ⇒ set protection bits in page-table entries for mapping
    - (Protections can later be changed using *mprotect(2)*)
  - `PROT_READ`: for read-only mapping
  - `PROT_READ | PROT_WRITE`: for read-write mapping
  - Must be consistent with access mode of *shm_open()*
    - E.g., can't specify `O_RDONLY` to *shm_open()* and then `PROT_READ | PROT_WRITE` for *mmap()*
  - Also `PROT_EXEC`: contents of memory can be executed

Mapping a shared memory object: *mmap()*

```
include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- *flags*: bit flags controlling behavior of call
  - POSIX SHM objects: need only `MAP_SHARED`
  - `MAP_SHARED` == make caller's modifications to mapped memory visible to other processes mapping same object

Example: pshm/pshm_create_simple.c

```
./pshm_create_simple /shm-object-name size
```

- Create a SHM object with given name and size

Example: pshm/pshm_create_simple.c

```
size_t size = atoi(argv[2]);
int fd = shm_open(argv[1], O_CREAT | O_EXCL | O_RDWR, S_IRUSR|S_IWUSR);
ftruncate(fd, size);
void *addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

1. SHM object created with RW permission for user, opened with read-write access mode
2. *fd* returned by *shm_open()* is used in *ftruncate()* + *mmap()*
3. Same *size* is used in *ftruncate()* + *mmap()*
4. *mmap()* not necessary, but demonstrates how it's done
5. Mapping protections `PROT_READ | PROT_WRITE` consistent with `O_RDWR` access mode

Outline

Using shared memory objects

- Address returned by *mmap()* can be used just like any C pointer
  - Usual approach: treat as pointer to some structured type
- Can read and modify memory via pointer

[TLPI §48.6]

Example: pshm/pshm_write.c

```
./pshm_write /shm-name string
```

- Open existing SHM object *shm-name* and copy *string* to it

Example: pshm/pshm_write.c

```
int fd = shm_open(argv[1], O_RDWR, 0);
size_t len = strlen(argv[2]);
ftruncate(fd, len);
printf("Resized to %ld bytes\n", (long) len);

char *addr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED,
                  fd, 0);
close(fd);              /* 'fd' is no longer needed */

printf("copying %ld bytes\n", (long) len);
memcpy(addr, argv[2], len);
```

1. Open existing SHM object
2. Resize object to match length of command-line argument
3. Map object at address chosen by system
4. Copy *argv[2]* to object (without '\0')
5. SHM object is closed and unmapped on process termination

Example: pshm/pshm_read.c

```
./pshm_read /shm-name
```

- Open existing SHM object *shm-name* and write the characters it contains to *stdout*

Example: pshm/pshm_read.c

```
int fd;
char *addr;
struct stat sb;

fd = shm_open(argv[1], O_RDONLY, 0);

fstat(fd, &sb);
addr = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);

close(fd);              /* 'fd' is no longer needed */

write(STDOUT_FILENO, addr, sb.st_size);
write(STDOUT_FILENO, "\n", 1);
```

- Open existing SHM object
- Use *fstat()* to discover size of object
- Map the object, using size from *fstat()* (in *sb.st_size*)
- Write all bytes from object to *stdout*, followed by newline

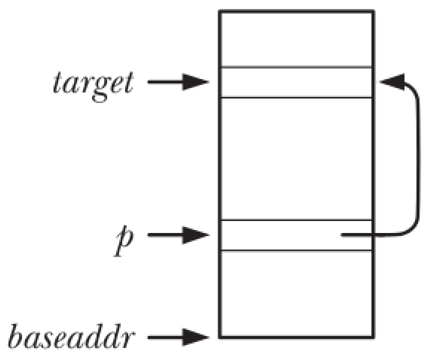
Pointers in shared memory

A little care is required when storing pointers in SHM:

- Assuming we let system choose address at which to place SHM (as is recommended practice)
- ⇒ SHM may be placed at different address in each process
- Suppose we want to build dynamic data structures, with pointers inside shared memory...
  - E.g., linked list
- ⇒ Must **use relative offsets**, not absolute addresses
  - Absolute address has no meaning if mapping is at different location in another process

[TLPI §48.6]

Pointers in shared memory



- Suppose we have situation at right
  - *baseaddr* is start of shared memory region
  - Want to store pointer to *target* in **p*
- ⚠ Wrong way:

```
*p = target
```

- Correct method (relative offset):

```
*p = target - baseaddr;
```

- To dereference "pointer":

```
target = baseaddr + *p;
```

The /dev/shm filesystem

On Linux:

- *tmpfs* filesystem used to implement POSIX SHM is mounted at `/dev/shm`
- Can list objects in directory with *ls(1)*
  - *ls –l* shows permissions, ownership, and size of each object

```
$ ls -l /dev/shm
-rw-------. 1 mtk mtk 4096 Oct 27 13:58 myshm
-rw-------. 1 mtk mtk   32 Oct 27 13:57 sem.mysem
```

- POSIX named semaphores are also visible in `/dev/shm`
  - As small SHM objects with names prefixed with "`sem.`"
- Can delete objects with *rm(1)*

Outline

Synchronizing access to shared memory

- Accesses to SHM object by different processes must be synchronized
  - Prevent simultaneous updates
  - Prevent read of partially updated data
- Semaphores are a common technique
- POSIX unnamed semaphores are often convenient, since:
  - Semaphore can be placed inside shared memory region
    - (And thus, automatically shared)
  - We avoid task of creating name for semaphore

Synchronizing access to shared memory

- Other synchronization schemes are possible
  - E.g., if using SHM to transfer large data volumes:
    - Using semaphore pair to force alternating access is expensive (two context switches on each transfer!)
    - Divide SHM into (logically numbered) blocks
    - Use pair of pipes to exchange metadata about filled and emptied blocks (also integrates with *poll()*/*epoll*!)

Example: synchronizing with unnamed semaphores

- Example application maintains sequence number in SHM object
- Source files:
  - `pshm/pshm_seqnum.h`: defines structure stored in SHM object
  - `pshm/pshm_seqnum_init.c`:
    - Create and open SHM object
    - Initialize semaphore and (optionally) sequence number inside SHM object
  - `pshm/pshm_seqnum_get.c`: display current value of sequence number and (optionally) increase its value

Example: pshm/pshm_seqnum.h

```
#include <sys/mman.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/stat.h>
#include "tlpi_hdr.h"

struct shmbuf {         /* Shared memory buffer */
    sem_t sem;          /* Semaphore to protect access */
    int seqnum;         /* Sequence number */
};
```

- Header file used by pshm/pshm_seqnum_init.c and pshm/pshm_seqnum_get.c
- Includes headers needed by both programs
- Defines **structure used for SHM object**, containing:
  - **Unnamed semaphore** that guards access to sequence number
  - **Sequence number**

Example: pshm/pshm_seqnum_init.c

```
./pshm_seqnum_init /shm-name [init-value]
```

- Create and open SHM object
- Reset semaphore inside object to 1 (i.e., semaphore available)
- Initialize sequence number

Example: pshm/pshm_seqnum_init.c

```
shm_unlink(argv[1]);
int fd = shm_open(argv[1], O_CREAT | O_EXCL | O_RDWR, S_IRUSR | S_IWUSR);

ftruncate(fd, sizeof(struct shmbuf));
struct shmbuf *shmp = mmap(NULL, sizeof(struct shmbuf),
                           PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
sem_init(&shmp->sem, 1, 1);
if (argc > 2)
    shmp->seqnum = atoi(argv[2]);
```

1. Delete previous instance of SHM object, if it exists
2. Create and open SHM object
3. Use *ftruncate()* to adjust size of object to match structure
4. Map object, using size of structure
5. Initialize semaphore state to "available"
   - *pshared* specified as 1, for process sharing of semaphore
6. If *argv[2]* supplied, initialize sequence # to that value
   - Newly extended bytes of SHM object are initialized to 0

Example: pshm/pshm_seqnum_get.c

```
./pshm_seqnum_get /shm-name [run-length]
```

- Open existing SHM object
- Fetch and display current value of sequence number in SHM object *shm-name*
- If *run-length* supplied, add to sequence number

Example: pshm/pshm_seqnum_get.c

```
int fd = shm_open(argv[1], O_RDWR, 0);

struct shmbuf *shmp = mmap(NULL, sizeof(struct shmbuf),
                           PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

- Open existing SHM object
- Map object, using size of *shmbuf* structure

Example: pshm/pshm_seqnum_get.c

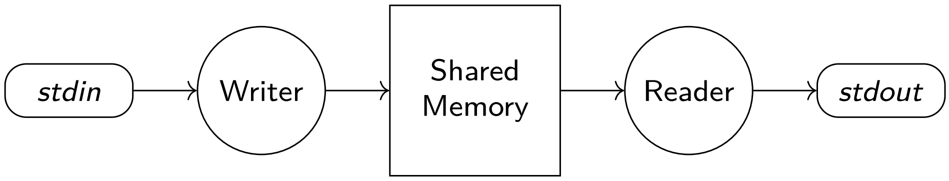
```
sem_wait(&shmp->sem);
printf("Current value of sequence number: %d\n", shmp->seqnum);

if (argc > 2) {
    int runLength = atoi(argv[2]);
    if (runLength <= 0)
        fprintf(stderr, "Invalid run-length\n");
    else {
        sleep(3);                     /* Make update slow */
        shmp->seqnum += runLength;
        printf("Updated sequence number\n");
    }
}
sem_post(&shmp->sem);
```

- Reserve semaphore before touching sequence number
- Display current value of semaphore
- If (nonnegative) *argv[2]* provided, add to sequence number
    - Sleep during update, to see that other processes are blocked
- Release semaphore

Exercise

1. Write two programs that exchange a stream of data of arbitrary length via a POSIX shared memory object [Shared header file: `pshm/pshm_xfr.h`]:
   - The "writer" creates and initializes the shared memory object and semaphores used by both programs, and then reads blocks of data from *stdin* and copies them a block at a time to the shared memory region [Template: `pshm/ex.pshm_xfr_writer.c`].
   - The "reader" copies each block of data from the shared memory object to *stdout* [Template: `pshm/ex.pshm_xfr_reader.c`].



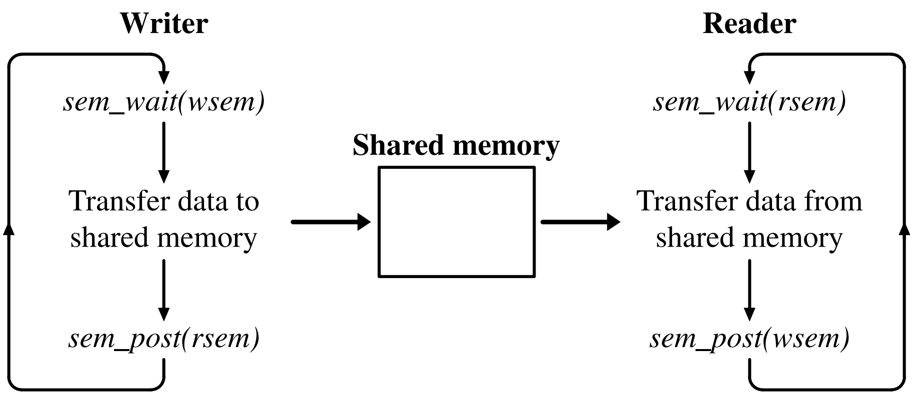
Note the following points:

- **Use the structure defined in** `pshm/pshm_xfr.h` for your shared memory.

[Exercise continues on next page]

Exercise

- You must ensure that the writer and reader have **exclusive, alternating access** to the shared memory region (so that, for example, the writer does not copy new data into the region before the reader has copied the current data to *stdout*). The following diagram shows how two semaphores can be used to achieve this. The semaphores should be initialized as *wsem=1* and *rsem=0*, so that the writer has first access to the shared memory.



(The simplest approach is to use two **unnamed** semaphores stored inside the shared memory object; see the structure definition in `pshm/pshm_xfr.h`.)
[Exercise continues on next page]

Exercise

- When the “writer” reaches end of file, it should provide an indication to the “reader” that there is no more data. To do this, maintain a byte-count field in the shared memory region which the “writer” uses to inform the “reader” how many bytes are to be written. Setting this count to 0 can be used to signal end-of-file. Once it has sent the last data block, the “writer” should unlink the shared memory object.
- Test your programs using a large file that contains random data:

```
$ dd if=/dev/urandom of=infile count=100000
$ ./ex.pshm_xfr_writer < infile &
$ ./ex.pshm_xfr_reader > outfile
$ diff infile outfile
```

  There is also a target in the `Makefile` for performing this test:

```
make pshm_xfr_test
```

[An optional exercise follows on the next page]

Exercise

2. Create a file of a suitable size (e.g., 512 MB in the following):

```
$ dd if=/dev/urandom of=/tmp/infile count=1000000
```

   Then edit the `BUF_SIZE` value in the `pshm/pshm_xfr.h` header file to vary the value from 10’000 down to 10 in factors of 10, in each case measuring the time required for the reader to complete execution:

```
$ ./ex.pshm_xfr_writer < /tmp/infile &
$ time ./ex.pshm_xfr_reader > /dev/null
```

   What is the reason for the variation in the time measurements?