Linux/UNIX System Programming

POSIX Shared Memory

Michael Kerrisk, man7.org © 2015

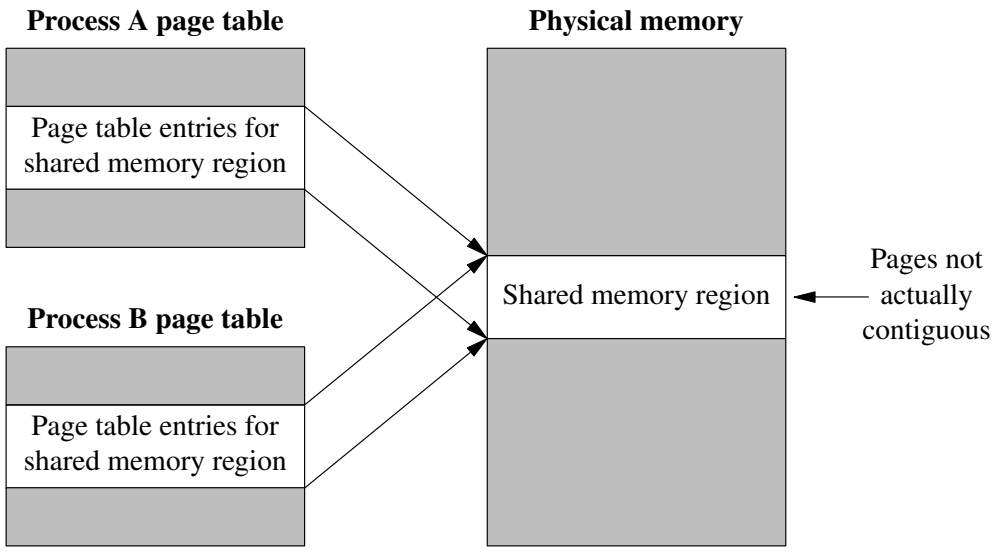
February 2015

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

UNIX has multiple shared memory APIs

UNIX systems have multiple shared memory APIs:

- **System V shared memory** (TLPI Ch. 48)
  - Original shared memory mechanism, still widely used
  - Sharing between **unrelated processes**
- **Shared mappings** – *mmap(2)* (TLPI Ch. 49)
  - Shared **anonymous** mappings
    - Sharing between **related processes** only (related via *fork()*)
  - Shared **file** mappings
    - Sharing between **unrelated processes, backed by file in filesystem**
- **POSIX shared memory**
  - Sharing between **unrelated processes, without overhead of filesystem I/O**
  - Intended to be simpler and better than older APIs

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

POSIX shared memory APIs

3 main APIs used with POSIX shared memory:

- *shm_open()*:
  - Open existing shared memory (SHM) object, or
  - Create and open new SHM object

  Returns a file descriptor used in later calls
- *ftruncate()*: set size of SHM object
- *mmap()*: map SHM object into caller's address space
- Compile with `cc -lrt`

POSIX shared memory APIs

Other APIs used with POSIX shared memory:

- *munmap()*: unmap a mapped object from 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
- *fstat()*: retrieve *stat* structure describing objects
  - Includes size of object, ownership, and permissions

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`)
- *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
- Returns file descriptor on success, or -1 on error

[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

Sizing a shared memory object

- New SHM objects have length 0
- Before mapping, 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
- Can obtain size of existing object using *fstat()*
  - *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);
```

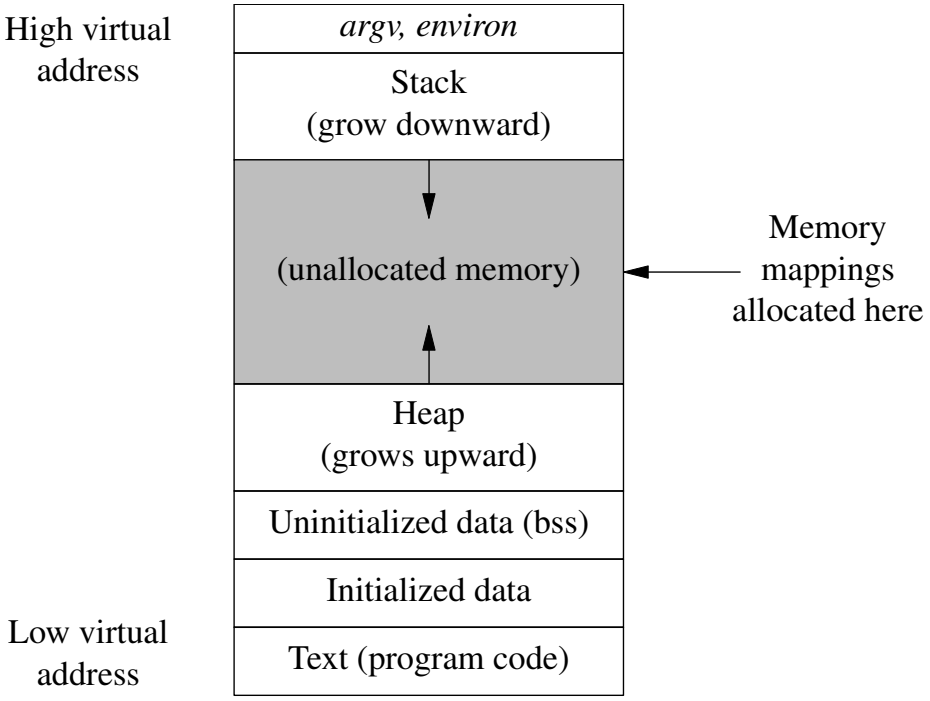
- **Complex, general-purpose API** for creating **memory mapping** in caller's virtual address space
  - Full details in TLPI Ch. 49
- We consider only use with POSIX SHM
  - In practice, only a few decisions to make
    - Usually just *length* and *prot*

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

Process memory layout (schematically)



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
  - NULL == let system choose address
    - "a hint" (ignored if address is already occupied)
  - 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)*
    - $\Rightarrow$ *ftruncate() size* is commonly multiple of page size
- *prot*: memory protections
  - `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()*

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: only need `MAP_SHARED`
  - `MAP_SHARED` == make caller's modifications to mapped memory visible to other processes mapping same object
- *fd*: file descriptor specifying file to map
  - Use FD returned by *shm_open()*
- *offset*: starting point of mapping in underlying file or SHM object
  - Must be multiple of system page size
  - Normally specified as 0 (map from start of 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

```
1 int fd;
2 size_t size;
3 void *addr;
4 size = atoi(argv[2]);
5 fd = shm_open(argv[1], O_CREAT | O_EXCL | O_RDWR,
6                S_IRUSR | S_IWUSR);
7 ftruncate(fd, size);
8 addr = mmap(NULL, size, PROT_READ | PROT_WRITE,
9             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/phm_write.c

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

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

Example: pshm/phm_write.c

```
 1 int fd;
 2 size_t len;           /* Size of shared memory object */
 3 char *addr;
 4 fd = shm_open(argv[1], O_RDWR, 0);
 5 len = strlen(argv[2]);
 6 ftruncate(fd, len);
 7 printf("Resized to %ld bytes\n", (long) len);
 8 addr = mmap(NULL, len, PROT_READ | PROT_WRITE,
 9             MAP_SHARED, fd, 0);
10 close(fd);               /* 'fd' is no longer needed */
11 printf("copying %ld bytes\n", (long) len);
12 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/phm_read.c

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

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

Example: pshm/phm_read.c

```
 1 int fd;
 2 char *addr;
 3 struct stat sb;
 4
 5 fd = shm_open(argv[1], O_RDONLY, 0);
 6
 7 fstat(fd, &sb);
 8 addr = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED,
 9              fd, 0);
10 close(fd);       /* 'fd' is no longer needed */
11
12 write(STDOUT_FILENO, addr, sb.st_size);
13 printf("\n");
```

- 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:

- *mmap()* maps SHM object at arbitrary location in memory
  - Assuming *addr* is specified as *NULL*, as recommended
- ⇒ Mapping may be placed at different address in each process
- Suppose we want to build dynamic data structures, with pointers inside shared memory...
- ⇒ 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 return value from *mmap()*
  - Want to store pointer to *target* in **p*
- ⚠ Wrong way:

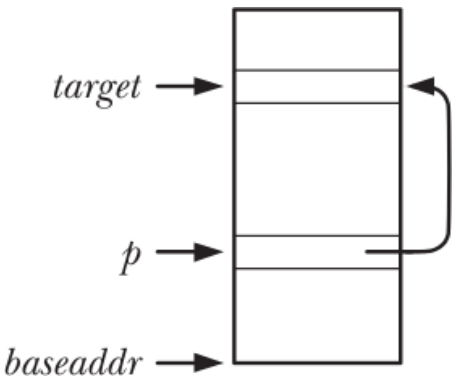
```
*p = target
```

- 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
- Any synchronization technique can be used
- Semaphores are most 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

Example: synchronizing with POSIX 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`

```
 1 #include <sys/mman.h>
 2 #include <fcntl.h>
 3 #include <semaphore.h>
 4 #include <sys/stat.h>
 5 #include "tlpi_hdr.h"
 6
 7 struct shmbuf {         /* Shared memory buffer */
 8     sem_t sem;          /* Semaphore to protect access */
 9     int seqnum;         /* Sequence number */
10 };
```

- 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

```
1 struct shmbuf *shmp;
2 shm_unlink(argv[1]);
3 fd = shm_open(argv[1], O_CREAT|O_EXCL|O_RDWR, S_IRUSR|S_IWUSR);
4 ftruncate(fd, sizeof(struct shmbuf));
5 shmp = mmap(NULL, sizeof(struct shmbuf),
6           PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
7 sem_init(&shmp->sem, 1, 1);
8 if (argc > 2)
9     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 number to that value
   - **Note**: 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 (1)

```
1 int fd, runLength;
2 struct shmbuf *shmp;
3
4 fd = shm_open(argv[1], O_RDWR, 0);
5
6 shmp = mmap(NULL, sizeof(struct shmbuf),
7           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 (2)

```
1  sem_wait(&shmp->sem);
2  printf("Current value of sequence number: %d\n",
3          shmp->seqnum);
4  if (argc > 2) {
5      runLength = atoi(argv[2]);
6      if (runLength <= 0)
7          fprintf(stderr, "Invalid run-length\n");
8      else {
9          sleep(3);             /* Make update slow */
10         shmp->seqnum += runLength;
11         printf("Updated semaphore value\n");
12     }
13 }
14 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:
   - The "writer" (`pshm_xfr_writer.c`) reads blocks of data from *stdin* and copies them a block at a time to the shared memory region.
   - The "reader" (`pshm_xfr_reader.c`) copies each block of data from the shared memory object to *stdout*

   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*). This will require the use of a pair of semaphores. (Using two **unnamed** semaphores stored inside the shared memory object is simplest, but remember that the "reader" will also need to map the shared memory for writing, so that it can update the semaphores.) The `psem_tty_lockstep_init.c` and `psem_tty_lockstep_second.c` programs provided in the POSIX semaphores module show how semaphores can be used for this task. (Those programs use a pair of **named** semaphores.)

   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 and the reader has indicated that it has consumed that block (e.g., by posting the "writer's" semaphore once more), the "writer" should destroy the unnamed semaphores and unlink the shared memory object.

Outline

Unmapping a shared memory object

```
include <sys/mman.h>
int munmap(void *addr, size_t length);
```

- Removes mapping described by *addr* and *length* from caller's virtual address space
- *addr*: starting address of mapping
  - Must be page-aligned
  - Typically, specify address returned by *mmap()*
- *length*: number of bytes to unmap
  - Rounded up to next page boundary
  - Typically, specify same *length* as was given to *mmap()*
- SHM objects are **automatically unmapped** on process termination

Removing shared memory objects: *shm_unlink()*

```
#include <sys/mman.h>
int shm_unlink(const char *name);
```

- Removes SHM object name
- *name*: pathname of SHM object
- Object will be deleted once all processes have unmapped it and all FDs from *shm_open()* have been closed
- Example: `pshm/pshm_unlink.c`

[TLPI §54.4]