Outline

25  POSIX Shared Memory 25-1
25.1  Overview 25-3
25.2  Creating and opening shared memory objects 25-8
25.3  Using shared memory objects 25-23
25.4  Synchronizing access to shared memory 25-32
25.5  Unmapping and removing shared memory objects 25-45
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

![Diagram of shared memory](image-url)
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
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.

- **close()**: Close file descriptor returned by **shm_open()**.

- **shm_unlink()**: Remove SHM object name, mark for deletion once all processes have closed.

Compile with `cc -lrt`.

- **shm_overview(7)** man page.
Creating/opening a shared memory object: *shm_open()*

```c
#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
  - (The close-on-exec flag is automatically set for the FD)

[TLPI §54.2]
Creating/opening a shared memory object: \texttt{shm\_open()}

```c
#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);
```

\texttt{oflag} specifies flags controlling operation of call

- \texttt{O\_CREATE}: create object if it does not already exist
- \texttt{O\_EXCL}: (with \texttt{O\_CREATE}) create object exclusively
  - Give error if object already exists
- \texttt{O\_RDONLY}: open object for read-only access
- \texttt{O\_RDWR}: open object for read-write access
  - NB: No \texttt{O\_WRONLY} flag...
- \texttt{O\_TRUNC}: truncate an existing object to zero length
  - Contents of existing object are destroyed

\texttt{mode}: permission bits for new object

- RWX for user / group / other
- ANDed against complement of process umask
- \texttt{\textcolor{red}{\triangle}} Required argument; specify as 0 if opening existing object
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()`

```c
#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()`

```c
#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()`
- **addr**: address at which to place mapping in caller’s virtual address space
  - Let’s look at a picture...

---

Process memory layout (schematically)

<table>
<thead>
<tr>
<th>argv, environ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>(grows downward)</td>
</tr>
<tr>
<td>(unallocated memory)</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>(grows upward)</td>
</tr>
<tr>
<td>Uninitialized data (bss)</td>
</tr>
<tr>
<td>Initialized data</td>
</tr>
<tr>
<td>Text (program code)</td>
</tr>
</tbody>
</table>

- High virtual address
- Memory mappings placed here
- Low virtual address
Mapping a shared memory object: `mmap()`

```c
#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

- **length**: size of mapping
  - Normally should be ≤ 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: \textit{mmap()}

\begin{verbatim}
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
\end{verbatim}

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

\begin{verbatim}
include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
\end{verbatim}

- \textit{flags}: bit flags controlling behavior of call
  - POSIX SHM objects: need only \textbf{MAP\_SHARED}
  - \textbf{MAP\_SHARED} \texttt{==} make caller’s modifications to mapped memory visible to other processes mapping same object
Example: pshm/pshm_create_simple.c

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

- Create a SHM object with given name and size

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

- SHM object created with RW permission for user, opened with read-write access mode
- `fd` returned by `shm_open()` is used in `ftruncate()` + `mmap()`
- Same `size` is used in `ftruncate()` + `mmap()`
- `mmap()` not necessary, but demonstrates how it’s done
- Mapping protections `PROT_READ | PROT_WRITE` consistent with `O_RDWR` access mode
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
Example: pshm/pshm_write.c

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

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

Example: pshm/pshm_write.c

```c
int fd;
size_t len; /* Size of shared memory object */
char *addr;
fd = shm_open(argv[1], O_RDWR, 0);
len = strlen(argv[2]);
ftruncate(fd, len);
printf("Resized to %ld bytes\n", (long) len);
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);
```

- Open existing SHM object
- Resize object to match length of command-line argument
- Map object at address chosen by system
- Copy *argv[2]* to object (without '\0')
- SHM object is closed and unmapped on process termination
Example: pshm/pshm_read.c

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

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

```
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, fd, 0);
9
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...
  - 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 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
    
    ```bash
    $ 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)`
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

```c
#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

```bash
./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**

```c
struct shmbuf *shmp;
shm_unlink(argv[1]);
fd = shm_open(argv[1], O_CREAT | O_EXCL | O_RDWR, S_IRUSR | S_IWUSR);
ftruncate(fd, sizeof(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
   - **Note:** newly extended bytes of SHM object are initialized to 0

**Example: pshm/pshm_seqnum_get.c**

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

1. Open existing SHM object
2. Fetch and display current value of sequence number in SHM object `shm-name`
3. If `run-length` supplied, add to sequence number
Example: pshm/pshm_seqnum_get.c (1)

```c
int fd, runLength;
struct shmbuf *shmp;
fd = shm_open(argv[1], O_RDWR, 0);
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 (2)

```c
sem_wait(&shmp->sem);
printf("Current value of sequence number: %d\n",
        shmp->seqnum);
if (argc > 2) {
    runLength = atoi(argv[2]);
    if (runLength <= 0)
        fprintf(stderr, "Invalid run-length\n");
    else {
        sleep(3); /* Make update slow */
        shmp->seqnum += runLength;
        printf("Updated semaphore value\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

Write two programs that exchange a stream of data of arbitrary length via a POSIX shared memory object [Common 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: ex.pshm_xfr_writer.c].
- The “reader” copies each block of data from the shared memory object to stdout [Template: ex.pshm_xfr_reader.c].

Note the following points:

- 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/psem_tty_lockstep_init.c and psem/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, 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
```
Unmapping a shared memory object

```c
#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: \textit{shm_unlink()}

\begin{verbatim}
#include <sys/mman.h>
int shm_unlink(const char *name);
\end{verbatim}

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

Notes