CSE 344 System Programming

Synchronization between processes

- Semaphores
- Shared Memory

Semaphores

A semaphore is a kernel-maintained integer whose value is restricted to being greater than or equal to 0. Various operations (i.e., system calls) can be performed on a semaphore, including :

- setting the semaphore to an absolute value;
- adding a number to the current value of the semaphore;
- subtracting a number from the current value of the semaphore;
- waiting for the semaphore value to be equal to 0.

Semaphores

The last two of these operations may cause the calling process to block.

When lowering a semaphore value, the kernel blocks any attempt to decrease the value below 0.

Similarly, waiting for a semaphore to equal 0 blocks the calling process if the semaphore value is not currently 0.

In both cases, the calling process remains blocked until some other process alters the semaphore to a value that allows the operation to proceed, at which point the kernel wakes the blocked process.

Semaphores

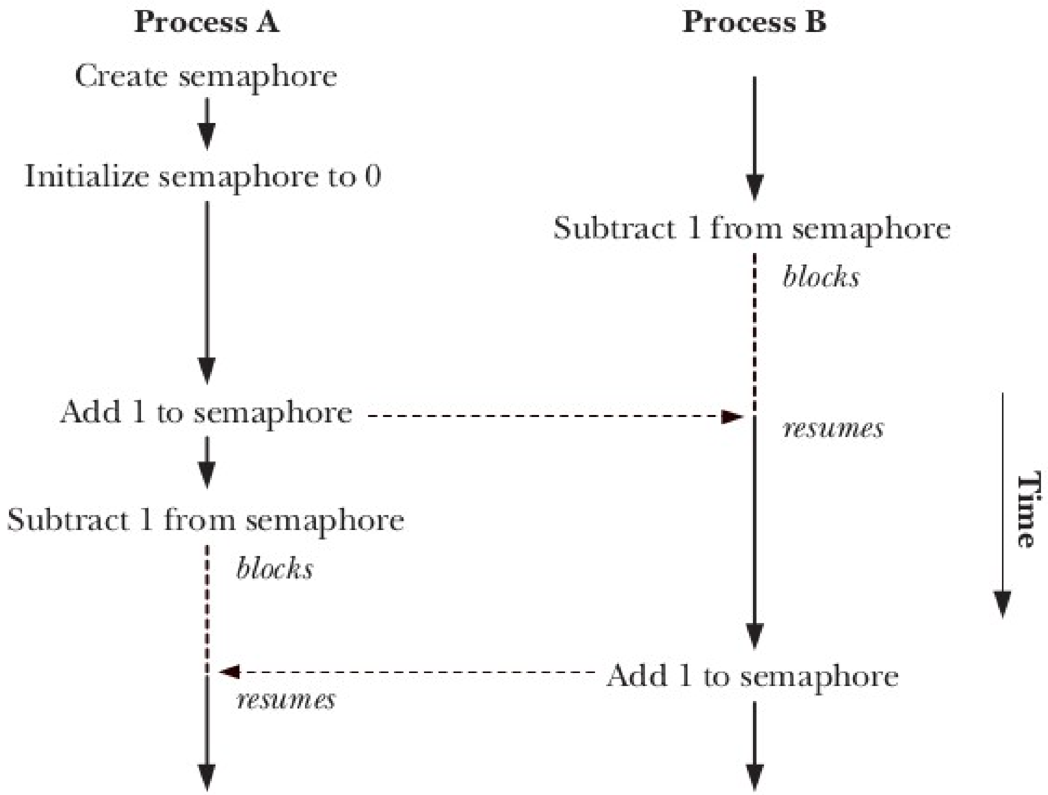


Figure illustrates the use of a semaphore to synchronize two processes

Semaphores

In terms of controlling the actions of a process, a semaphore has no meaning in and of itself.

Its meaning is determined only by the associations given to it by the processes using the semaphore. Typically, processes agree on a convention that associates a semaphore with a shared resource, such as a region of shared memory.

Other uses of semaphores are also possible, such as synchronization between parent and child processes after *fork()*.

Semaphores

In `POSIX:SEM` terminology, the `wait` and `post` operations are called ***semaphore lock*** and ***semaphore unlock***, respectively.

We can think of a semaphore as an integer value and a list of processes waiting for a post operation.

Semaphore implementations use **atomic operations** of the underlying operating system to ensure correct execution.

Suppose *process 1* must execute statement `a` before *process 2* executes statement `b`.
The semaphore sync enforces the ordering in the following pseudocode, provided that sync is initially 0.

```
Process 1 executes:            Process 2 executes:
  a;                             wait(&sync);
  post(&sync);                   b;
```

What happens in the following pseudocode if semaphores S and Q are both initialized to 1?

```
Process 1 executes:            Process 2 executes:
   for( ; ; ) {                  for( ; ; ) {
      wait(&Q);                      wait(&S);
      wait(&S);                      wait(&Q);
      a;                             b;
      post(&S);                      post(&Q);
      post(&Q);                      post(&S);
   }                             }
```

Semaphores

In this course, we will focus on two types of semaphore implementations :

- POSIX Semaphores (or simply semaphores)
- SYSTEM V Semaphores (semaphore sets)

For a relatively simpler introduction to semaphore concept students are referred to old CSE 244 slides and notes.

POSIX Semaphores

POSIX: SEM specifies **two types** of semaphores:

- ***Named semaphores***: This type of semaphore has a name. By calling `sem_open()` with the same name, unrelated processes can access the same semaphore.
- ***Unnamed semaphores***: This type of semaphore doesn't have a name; instead, it resides at an agreed-upon location in memory. Unnamed semaphores can be shared between processes or between a group of threads.

* note that some systems might not have a full implementation of POSIX semaphores (i.e. Linux 2.4)

Opening a named semaphore

```
#include <fcntl.h>              /* Defines O_* constants */
#include <sys/stat.h>           /* Defines mode constants */
#include <semaphore.h>

sem_t *sem_open(const char * name , int oflag , ...
                 /* mode_t mode , unsigned int value */ );
```

Returns pointer to a semaphore on success, or SEM_FAILED on error and errno is set

// #define SEM_FAILED ((sem_t *) 0)

The `sem_open()` function creates and opens a new named semaphore or opens an existing semaphore.

Regardless of whether we are creating a new semaphore or opening an existing semaphore, `sem_open()` returns a pointer to a `sem_t` value

POSIX Semaphores: Named Semaphores

Note that the results are **undefined** if we attempt to perform operations (*sem_post()*, *sem_wait()*, and so on) on a **copy** of the *sem_t* variable pointed to by the return value of *sem_open()*.

In other words, the following use of *sem2* is not permitted:

```
sem_t *sp, sem2;
sp = sem_open(...);
sem2 = *sp;
sem_wait(&sem2);
```

When a child is created via *fork()*, it inherits references to **all of the named semaphores** that are open in its parent. After the *fork()*, the parent and child can use these semaphores to synchronize their actions.

Closing a Semaphore

When a process opens a named semaphore, the system records the association between the process and the semaphore.

The `sem_close()` function terminates this association, releases any resources that the system has associated with the semaphore for this process, and decreases the count of processes referencing the semaphore.

```
#include <semaphore.h>

int sem_close(sem_t *sem)

                            Returns 0 on success, -1 on error
```

Closing a semaphore does not delete it. For that purpose, we need to use *`sem_unlink()`*.

Removing a Named Semaphore

The `sem_unlink()` function removes the semaphore identified by name and marks the semaphore to be destroyed once all processes cease using it.

```
#include <semaphore.h>

int sem_unlink(const char *name)
```

Returns 0 on success, -1 on error

Semaphore Operations

```
#define _XOPEN_SOURCE 600                    /* only required for timedwait case */
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);

int sem_timedwait(sem_t * sem , const struct timespec * abs_timeout );
                       /* !!!!!! not available on all UNIX Implementations !!!!!! */
```

Returns 0 on success, -1 on error

The `sem_wait()` function decrements (decreases by 1) the value of the semaphore referred to by `sem`.

If the semaphore currently has a value greater than 0, `sem_wait()` returns immediately. If the value of the semaphore is currently 0, `sem_wait()` blocks until the semaphore value rises above 0; at that time, the semaphore is then decremented and `sem_wait()` returns. The `sem_trywait()` function is a nonblocking version of `sem_wait()` (returns the EAGAIN error in lieu of blocking).

Semaphore Operations: posting/signalling

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Returns 0 on success, -1 on error

The `sem_post()` function increments (increases by 1) the value of the semaphore referred to by *`sem`*.

If the value of the semaphore was 0 before the `sem_post()` call, and some other process (or thread) is blocked waiting to decrement the semaphore, then that process is awoken, and its `sem_wait()` call proceeds to decrement the semaphore.

If multiple processes (or threads) are blocked in `sem_wait()`, then, if the processes are being scheduled under the default time-sharing policy, <u>**it is indeterminate which one will be awoken**</u> and allowed to decrement the semaphore.

Retrieving the Current Value of a Semaphore

The `sem_getvalue()` function returns the current value of the semaphore referred to by `sem` in the int pointed to by `sval`.

```
#include <semaphore.h>

int sem_getvalue(sem_t * sem , int * sval );
```

Returns 0 on success, -1 on error

If one or more processes (or threads) are currently blocked waiting to decrement the semaphore's value, then the value returned in *sval* depends on the implementation.

There are two possibilities: 0 or a negative number whose absolute value is the number of waiters blocked in `sem_wait()`

(On Linux the former behavior is adapted)

POSIX Semaphores: Unnamed Semaphores

Unnamed semaphores (also known as memory-based semaphores) are variables of type `sem_t` that are stored in memory allocated by the application.

The semaphore is made available to the processes or threads that use it by placing it in an area of memory that they share.

Operations on unnamed semaphores use the same functions that are used to operate on named semaphores. In addition, two further functions are required:

- The `sem_init()` function initializes a semaphore and informs the system of whether the semaphore will be shared between processes or between the threads of a single process.
- The `sem_destroy(sem)` function destroys a semaphore

Initializing an Unnamed Semaphore

The `sem_init()` function initializes the unnamed semaphore pointed to by `sem` to the value specified by value.

```
#include <semaphore.h>

int sem_init(sem_t * sem , int pshared , unsigned int value);
```

Returns 0 on success, -1 on error

The `pshared` argument indicates whether the semaphore is to be shared between threads or between processes.

- If `pshared` is 0, then the semaphore is to be shared between the threads of the calling process. In this case, sem is typically specified as the address of either a global variable or a variable allocated on the heap
- If `pshared` is nonzero, then the semaphore is to be shared between processes. In this case, sem must be the address of a location in a region of shared memory (a POSIX shared memory object, a shared mapping created using `mmap()`, or a System V shared memory segment).

Take a peek into the future (first the thread function definition)

```
#include <semaphore.h>
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;
static sem_t sem;

static void *threadFunc(void *arg)            /* Loop 'arg' times incrementing 'glob' */
{
 int loops = *((int *) arg);
 int loc, j;
 for (j = 0; j < loops; j++){
      if (sem_wait(&sem) == -1)
           errExit("sem_wait");
      loc = glob;
      loc++;
      glob = loc;
      if (sem_post(&sem) == -1)
           errExit("sem_post");
 }
 return NULL;
}
```

Take a peek into the future

```
int main(int argc, char *argv[])
{
   pthread_t t1, t2;
    int loops, s;
   loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

   /* Initialize a thread-shared mutex with the value 1 */
   if (sem_init(&sem, 0, 1) == -1)
      errExit("sem_init");
   /* Create two threads that increment 'glob' */
   s = pthread_create(&t1, NULL, threadFunc, &loops);
   if (s != 0)
      errExitEN(s, "pthread_create");
   s = pthread_create(&t2, NULL, threadFunc, &loops);
   if (s != 0)
      errExitEN(s, "pthread_create");
   /* Wait for threads to terminate */
   s = pthread_join(t1, NULL);
   if (s != 0)
      errExitEN(s, "pthread_join");
   s = pthread_join(t2, NULL);
   if (s != 0)
      errExitEN(s, "pthread_join");
   printf("glob = %d\n", glob);
   exit(EXIT_SUCCESS);
}
```

Destroying an Unnamed Semaphore

The `sem_destroy()` function destroys the semaphore `sem`, which must be an unnamed semaphore that was previously initialized using `sem_init()`. It is safe to destroy a semaphore only if no processes or threads are waiting on it.

```
#include <semaphore.h>

int sem_destroy(sem_t * sem );
```

Returns 0 on success, -1 on error

After an unnamed semaphore segment has been destroyed with `sem_destroy()`, it can be reinitialized with `sem_init()`. An unnamed semaphore should be destroyed **before its underlying memory is deallocated**. (!!! beware of leaks !!!)

Comparisons with Other Synchronization Techniques

POSIX semaphores versus System V semaphores

- The POSIX semaphore interface does not support waiting/decrementing atomically for n > 1
- The POSIX semaphore interface is simpler than the System V semaphore interface. This simplicity is achieved without loss of functional power.
- POSIX named semaphores eliminate the initialization problem associated with System V semaphores (sem. sets)

Comparisons with Other Synchronization Techniques

POSIX semaphores versus Pthreads mutexes

POSIX semaphores and Pthreads mutexes can both be used to synchronize the actions of threads within the same process, and their performance is similar.

However, mutexes are usually preferable, because the ownership property of mutexes enforces good structuring of code. By contrast, one thread can increment a semaphore that was decremented by another thread. This flexibility can lead to poorly structured synchronization designs.

Comparisons with Other Synchronization Techniques

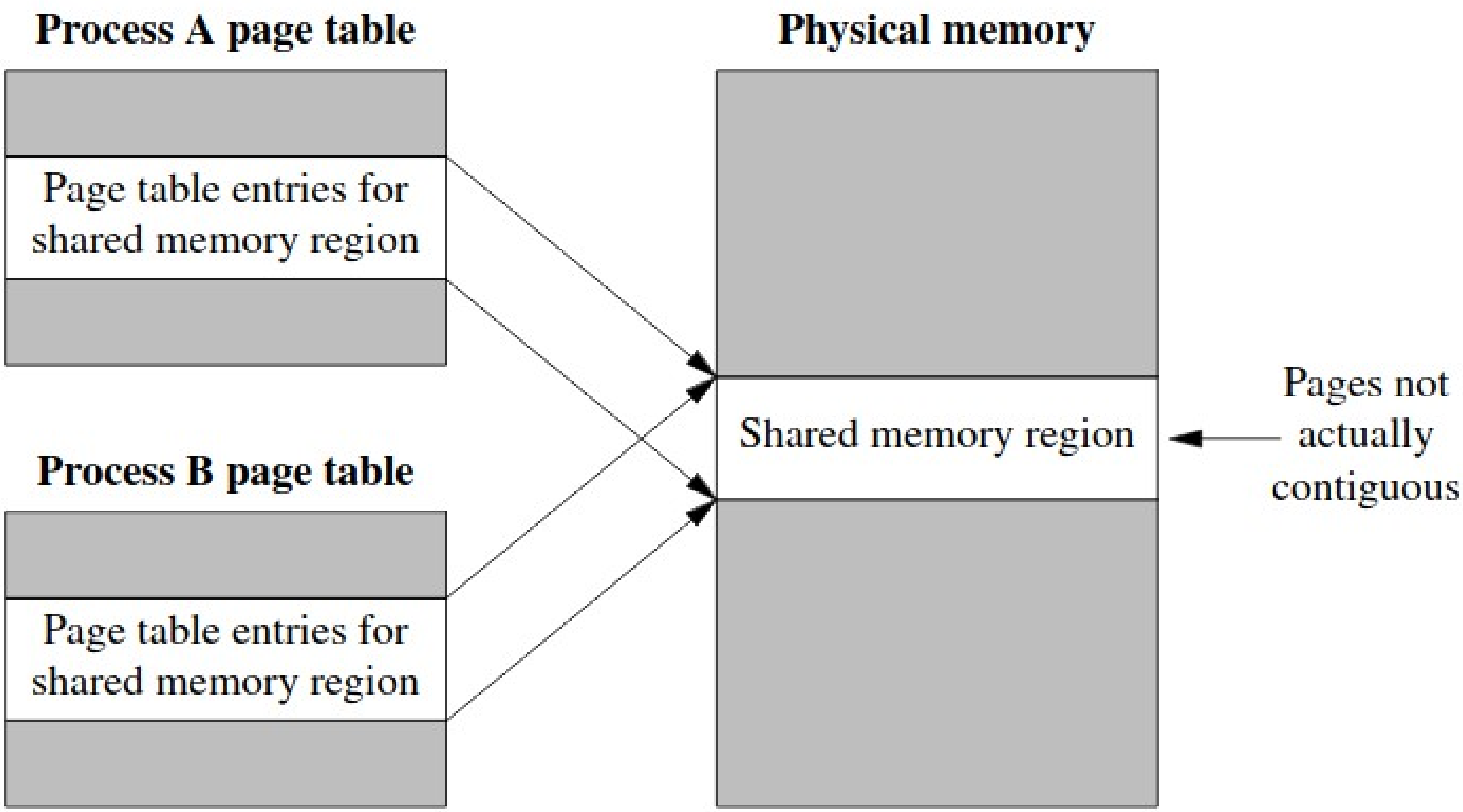
POSIX semaphores versus Pthreads mutexes

There is one circumstance in which mutexes can't be used in a multi-threaded application and semaphores may therefore be preferable.

Because it is async-signal safe , the `sem_post()` function can be used from within a signal handler to synchronize with another thread. This is not possible with mutexes, because the Pthreads functions for operating on mutexes **are not async-signal-safe.** However, because it is usually preferable to deal with asynchronous signals by accepting them using *sigwaitinfo()* (or similar), rather than using signal handlers, this advantage of semaphores over mutexes is seldom required.

Shared Memory

- Shared memory allows two or more processes to share the same region of physical memory.



Shared Memory

- Since a shared memory segment becomes part of a process's user-space memory, no kernel intervention is required for IPC. All that is required is that one process copies data into the shared memory; that data is immediately available to all other processes sharing the same segment.
- This provides fast IPC by comparison with techniques such as pipes or message queues, where the sending process copies data from a buffer in user space into kernel memory and the receiving process copies in the reverse direction.
- The fact that IPC using shared memory is not mediated by the kernel implies that some method of synchronization is required so that processes don't simultaneously access the shared memory

Other Shared Memory Techniques

- The unix world has many flavors

1) **System V shared memory**

→ Original shared memory mechanism, still widely used

→ Sharing between unrelated processes

2) **Shared mappings –mmap(2)**

→ Shared anonymous mappings (between related processes)

→ Shared file mappings (between unrelated processes, backed by file in FS)

3) **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

POSIX shared memory allows the user to share a mapped region between unrelated processes without needing to create a corresponding mapped file. To use a POSIX shared memory object, we perform two steps:

1) Use the `shm_open()` function to open an object with a specified name. The `shm_open()` function is analogous to the `open()` system call. It either creates a new shared memory object or opens an existing object.

2) Pass the file descriptor obtained in the previous step in a call to `mmap()` that specifies MAP_SHARED in the flags argument. This maps the shared memory object into the process's virtual address space.

In the case of Linux

- They are implemented as files in a dedicated **tmpfs** filesystem

→ tmpfs is a virtual memory filesystem that employs swapspace when needed

- Objects have **kernel persistence**

→ Objects exist until explicitly deleted, or system reboots (so exiting does not mean it is deleted automagically)

→ Can map an object, change its contents, and unmap

→ Changes will be visible to next process that maps object

Creating Shared Memory Objects

The `shm_open()` function creates and opens a new shared memory object or opens an existing object. The arguments to `shm_open()` are analogous to those for `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 );
```

Returns file descriptor on success, or –1 on error

When a new shared memory object is created, it initially has zero length. This means that, after creating a new shared memory object, we normally call `ftruncate()` to set the size of the object before calling `mmap()`. Following the `mmap()` call, we may also use `ftruncate()` to expand or shrink the shared memory object as desired.

Example Program : > $ pshm_create -c /demo_shm 10000

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

static void usageError(const char *progName) {
   fprintf(stderr, "Usage: %s [-cx] name size [octal-perms]\n", progName);
   fprintf(stderr, " -c Create shared memory (O_CREAT)\n");
   fprintf(stderr, "-x Create exclusively (O_EXCL)\n");
   exit(EXIT_FAILURE); }

int  main(int argc, char *argv[])
{
   int flags, opt, fd;
   mode_t perms;
   size_t size;
   void *addr;

   flags = O_RDWR;
   while ((opt = getopt(argc, argv, "cx")) != -1) {
     switch (opt) {
           case 'c': flags |= O_CREAT; break;
           case 'x': flags |= O_EXCL;  break;
           default:  usageError(argv[0]);
   }
}
```

Example Program :cont...

```
   if (optind + 1 >= argc)
         usageError(argv[0]);

   size = getLong(argv[optind + 1], GN_ANY_BASE, "size");
   perms = (argc <= optind + 2) ? (S_IRUSR | S_IWUSR) :
              getLong(argv[optind + 2], GN_BASE_8, "octal-perms");

   /* Create shared memory object and set its size */
     fd = shm_open(argv[optind], flags, perms);
         if (fd == -1)
             errExit("shm_open");
         if (ftruncate(fd, size) == -1)
              errExit("ftruncate");

   /* Map shared memory object */
     addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
     if (addr == MAP_FAILED)
         errExit("mmap");

   exit(EXIT_SUCCESS);
}
```

Using Shared Memory Objects

- The programs given on the next two slides demonstrate the use of a shared memory object to transfer data from one process to another. The first program copies the string contained in its second command-line argument into the existing shared memory object named in its first command-line argument.

- Before mapping the object and performing the copy, the program uses *ftruncate()* to resize the shared memory object to be the same length as the string that is to be copied.

Example Program :writing to a shared memory object

```
———————————————————————————————————————————————————— pshm/pshm_write.c
#include <fcntl.h>
#include <sys/mman.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd;
    size_t len;                 /* Size of shared memory object */
    char *addr;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s shm-name string\n", argv[0]);

    fd = shm_open(argv[1], O_RDWR, 0);      /* Open existing object */
    if (fd == -1)
        errExit("shm_open");

    len = strlen(argv[2]);
    if (ftruncate(fd, len) == -1)           /* Resize object to hold string */
        errExit("ftruncate");
    printf("Resized to %ld bytes\n", (long) len);

    addr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1)
        errExit("close");                   /* 'fd' is no longer needed */

    printf("copying %ld bytes\n", (long) len);
    memcpy(addr, argv[2], len);             /* Copy string to shared memory */
    exit(EXIT_SUCCESS);
}
———————————————————————————————————————————————————— pshm/pshm_write.c
```

pshm/pshm_read.c

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

int
main(int argc, char *argv[])
{
    int fd;
    char *addr;
    struct stat sb;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s shm-name\n", argv[0]);

    fd = shm_open(argv[1], O_RDONLY, 0);    /* Open existing object */
    if (fd == -1)
        errExit("shm_open");

    /* Use shared memory object size as length argument for mmap()
       and as number of bytes to write() */

    if (fstat(fd, &sb) == -1)
        errExit("fstat");

    addr = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1);                   /* 'fd' is no longer needed */
        errExit("close");

    write(STDOUT_FILENO, addr, sb.st_size);
    printf("\n");
    exit(EXIT_SUCCESS);
}
```

pshm/pshm_read.c

Removing Shared Memory Objects

When a shared memory object is no longer required, it should be removed using `shm_unlink()`.

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

int shm_unlink(const char * name );
```

Returns 0 on success, or –1 on error

The `shm_unlink()` function removes the shared memory object specified by name. Removing a shared memory object doesn't affect existing mappings of the object (which will remain in effect until the corresponding processes call munmap() or terminate), but prevents further `shm_open()` calls from opening the object. Once all processes have unmapped the object, the object is removed, and its contents are lost.

Overall...

- Shared Memory provides fast IPC, and applications typically must use a semaphore (or other synchronization primitive) to synchronize access to the shared region.
- Once the shared memory region has been mapped into the process's virtual address space, it looks just like any other part of the process's memory space.
- The system places the shared memory regions within the process virtual address space .
- Assuming that we don't attempt to map a shared memory region at a fixed address, we should ensure that all references to locations in the region are calculated as offsets (rather than pointers), since the region may be located at different virtual addresses within different processes .

Overall...

e.g. a linkedlist stored in shared memory.

```
struct node{
  DATA d;
  struct node * next;
}
struct node * shm = (struct node *)mmap(…);
```

Assume process P1 is attached to shm at address 10.000.

In which case `next` could point to address 10.030 for instance.

If process P2 is attached to the shm at address 55.000 then what happens when it tries to access `shm->next`?

Overall...

Solution: use indexes instead of pointers in shared memory segments.

```
struct node{
  DATA d;
  int next;
};
```

The linkedlist becomes an array `shm` of nodes

e.g. `shm[shm[0].next].d`

In the general case pointers need to be replaced by offsets w.r.t. the base address of the shared memory segment.

Overall...

`fork()`**:** a child process inherits its address space from its parent process, hence all shared memory segments are inherited at the same addresses.

`exit()`**:** all shared memory segments are detached prior to the descruction of the process.

`exec()`**:** same as `exit()`