POSIX

- **POSIX** or "Portable Operating System Interface" is the collective name of a family of related standards specified by the IEEE to define the application programming interface (API) for software compatible with variants of the Unix operating system.

Why POSIX?

- POSIX is an industry-standard operating system specification that enables applications to be easily ported across different hardware and operating systems implementations.

- POSIX.1b, Real-time extensions
- POSIX.1c, POSIX Threads

Semaphores

- Semaphores are used to control access to shared resources by processes.
- There are named and unnamed semaphores.
- Named semaphores provide access to a resource between multiple processes.
- Unnamed semaphores provide multiple accesses to a resource within a single process or between related processes.
- Some semaphore functions are specifically designed to perform operations on named or unnamed semaphores.
- Semaphores are global entities and are not associated with any particular process.

Creating and Opening a Semaphore

- The sem_open function establishes a connection between a named semaphore and the calling process.
- Subsequent to creating a semaphore with sem_open, the calling process can reference the semaphore by using the semaphore descriptor address returned from the call.
- The semaphore is available in subsequent calls to the sem_wait, sem_trywait, and sem_post functions, which control access to the shared resource.
- You can also retrieve the semaphore value by calls to sem_getvalue.

Creating and Opening a Semaphore

sem_t *sem_open(const char *_name_, int _oflag_, ...);

- The sem_open() function creates a connection between a named semaphore and a process.
- One the connection has been created for the semaphore name specified by the name argument with a call to sem_open(), the process can use the address returned by the call to reference that semaphore.
- The oflag argument controls whether the semaphore is created or merely accessed by the call to sem_open(). It can have the following flag bits set:
  - O_CREAT
- The O_CREAT flag requires two additional arguments: *mode* of type mode_t, and *value* of type unsigned int.
- The *value* argument specifies the initial value assigned to the newly created semaphore

Locking and Unlocking Semaphores

- After you create the semaphore with a call to sem_open function, you can use the sem_wait, sem_trywait, and sem_post functions to lock and unlock the semaphore.
- To lock a semaphore, you can use either the sem_wait or sem_trywait function.
- If the semaphore value is greater than zero, the sem_wait function locks the specified semaphore.
- If the semaphore value is less than or equal to zero, the process is blocked (sleeps) and must wait for another process to release the semaphore and increment the semaphore value.
- To be certain that the process is not blocked while waiting for a semaphore to become available, use the sem_trywait function.
- The sem_trywait function will lock the specified semaphore if, and only if, it can do so without waiting.
- That is, the specified semaphore must be available at the time of the call to the sem_trywait function.

Locking and Unlocking Semaphores

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

- The sem_wait() function locks the specified semaphore by performing a semaphore lock operation on that semaphore.
- The sem_trywait() function locks the specified semaphore only if that semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore.

int sem_post(sem_t ****sem*);**

- The sem_post() function unlocks the specified semaphore by performing a semaphore unlock operation on that semaphore.

Closing a Semaphore

- When an application is finished using an unnamed semaphore, it should destroy the semaphore with a call to the sem_destroy function.
- For named semaphores, the application should deallocate the semaphore with a call to the sem_close function.
- The semaphore name is disassociated from the process.
- A named semaphore is removed using the sem_unlink function, which takes effect once all processes using the semaphore have deallocated the semaphore with calls to sem_close.
- If needed, the semaphore can be reopened for use through a call to the sem_open function.
- Since semaphores are persistent, the state of the semaphore is preserved, even though the semaphore is closed.
- When you reopen the semaphore, it will be in the state it was when it was closed, unless altered by another process.

Closing a Semaphore

int sem_close(sem_t *_sem_);

- The sem_close() function closes the semaphore specified by the sem argument when the calling process is finished with it.
- When a semaphore is closed, sem_close() frees up any system resources allocated to be used by that semaphore.
- These resources are then available for use by a subsequent invocation of the sem_open() function.
- **int sem_unlink(const char *_name_);**
- The sem_unlink() function removes the specified named semaphore.

EXAMPLE

```
• #include<stdio.h>
  #include<semaphore.h>
  #include<unistd.h>
  #include<sys/types.h>
  #include<fcntl.h>

int main()
  {
  int x,z;
  sem_t *s;
  s=sem_open("/xyz",O_CREAT,0666,1);
  sem_wait(s);
  sleep(6);
  write(1,"hello",5);
  sem_post(s);
  sem_getvalue(s,&z);
  printf("value: %d\n",z);
  sem_close(s);
  sem_unlink("xyz");
  }
```

```
• #include<stdio.h>
  #include<semaphore.h>
  #include<unistd.h>
  #include<sys/types.h>
  #include<fcntl.h>

int main()
  {
  sem_t *s; int z;
  s=sem_open("xyz",O_CREAT);
  sem_wait(s);
  write(1,"world",5);
  sem_getvalue(s,&z);
  printf("z5%d\n",z);
  sleep(5);
  sem_post(s);
  sem_close(s);
  }
```

POSIX SHARED MEMORY

- Shared memory and memory-mapped files allow processes to communicate by incorporating data directly into process address space.
- Shared memory and memory-mapped files follow the same general usage, as follows:
  - Get a file descriptor with a call to the open or shm_open function.
  - Map the object using the file descriptor with a call to the mmap function.
  - Unmap the object with a call to the munmap function.
  - Close the object with a call to the close function.
  - Remove the shared-memory object with a call to the shm_unlink function or optionally remove a memory-mapped file with a call to the unlink function.
- Files, however, may need to be saved and reused each time the application is run. The unlink and shm_unlink functions remove (delete) the file and its contents. Therefore, if you need to save a shared file, close the file but do not unlink it

Opening a Shared-Memory Object

shm_open()

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

DESCRIPTION

- The shm_open() function establishes a connection between a shared memory object and a file descriptor.
- The file descriptor is used by other functions to refer to that shared memory object.
- The name argument points to a string naming a shared memory object.
- If successful, shm_open() returns a file descriptor for the shared memory object that is the lowest numbered file descriptor not currently open for that

Opening a Shared-Memory Object

- The open file description is new, and therefore the file descriptor does not share it with any other processes.
- The file status flags and file access modes of the open file description are according to the value of oflag.
- The oflag argument is the bitwise inclusive OR of the following flags defined in the header <fcntl.h>.
- Applications specify exactly one of the first two values (access modes) below in the value of oflag:
  - O_RDONLY
    - Open for read access only.
  - O_RDWR
    - Open for read or write access.
- Any combination of the remaining flags may be specified in the value of oflag:
  - O_CREAT
  - O_EXCL

Opening Memory-Mapped Files

- The open function points to the data you intend to use; the mmap function establishes how much of the data will be mapped and how it will be accessed.
- After opening a file, call the mmap function to map the file into application address space.
- When you have finished using a memory-mapped file, unmap the object by calling the munmap function, then close the object with the close function.

Mapping Memory-Mapped Files

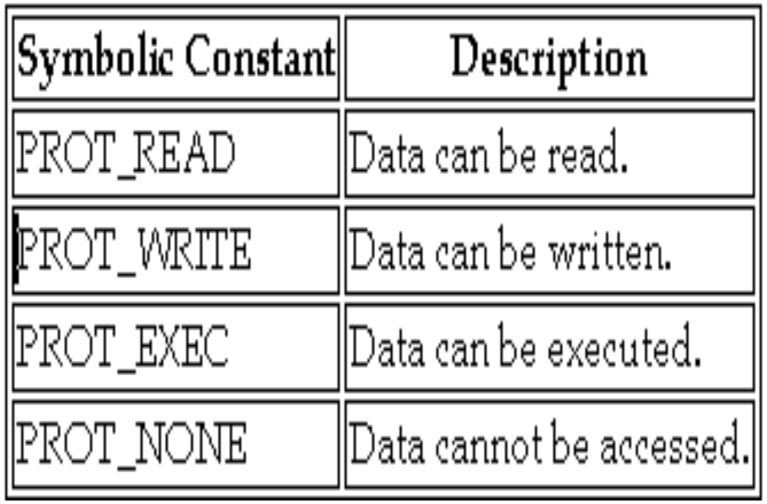
```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flags,
                                        int fildes, off_t off);
```

DESCRIPTION

- The mmap() function establishes a mapping between a process' address space and a file or shared memory object.
- The mmap() function establishes a mapping between the address space of the process at an address sa for len bytes to the memory object represented by the file descriptor fildes at offset off for len bytes.
- The mmap() function is supported for regular files and shared memory objects. Support for any other type of file is unspecified.

Mapping Memory-Mapped Files

- The parameter prot determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped.
- The prot should be either PROT_NONE or the bitwise inclusive OR of one or more of the other flags in the following table, defined in the header <sys/mman.h>.

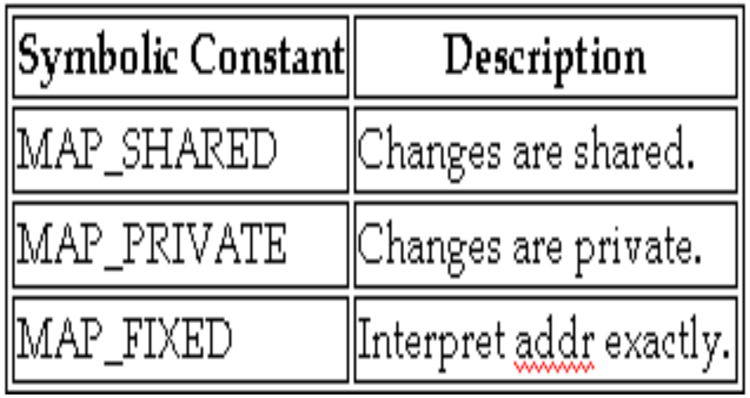


Symbolic Constant	Description
PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.
PROT_NONE	Data cannot be accessed.

Mapping Memory-Mapped Files

- The parameter flags provides other information about the handling of the mapped data.
- The value of flags is the bitwise inclusive OR of these options, defined in <sys/mman.h>:
- If MAP_SHARED is specified, write references change the underlying object.

•If MAP_PRIVATE is specified, modifications to the mapped data by the calling process will be visible only to the calling process and will not change the underlying object.
•When MAP_FIXED is set in the flags argument, the implementation is informed that the value of sa must be addr, exactly.



Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret addr exactly.

Controlling Memory-Mapped Files

- Several functions let you manipulate and control access to memory-mapped files and shared memory.
- These functions include msync and mprotect.
- Using these functions, you can modify access protections and synchronize writing to a mapped file.

int msync(void *addr, size_t len, int flags);

- The msync function synchronizes the caching operations of a memory-mapped file or shared-memory region.
- When you use the MS_SYNC flag, the msync function does not return until all write operations are complete and the integrity of the data is assured. All previous modifications to the mapped region are visible to processes using the read parameter.
- When you use the MS_ASYNC flag, the msync function returns immediately after all of the write operations are scheduled.

Controlling Memory-Mapped Files

int mprotect(void **addr*, size_t *len*, int *prot*);

- The function *mprotect()* changes the access protections to be that specified by *prot* for those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes.
- The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped.
- The *prot* argument should be either PROT_NONE or the bitwise inclusive OR of one or more of PROT_READ, PROT_WRITE and PROT_EXEC.

UNmapping Memory-Mapped Files

- #include <sys/mman.h>

  int munmap(void *addr, size_t len);
- The function munmap() removes any mappings for those entire pages containing any part of the address space of the process starting at addr and continuing for len bytes.
- The implementation will require that addr be a multiple of the page size {PAGESIZE}.
- If a mapping to be removed was private, any modifications made in this address range will be discarded.

Removing Shared Memory

- When a process has finished using a shared-memory segment, you can remove the name from the file system namespace with a call to the shm_unlink function.

  #include <sys/mman.h>

  int shm_unlink(const char * name);

- The shm_unlink function unlinks the shared-memory object.
- Memory objects are persistent, which means the contents remain until all references have been unmapped and the shared-memory object has been unlinked with a call to the shm_unlink function.
- Every process using the shared memory should perform the cleanup tasks of unmapping and closing.