Task 3

System V IPC *vs* POSIX IPC

Both are using the same concepts – semaphores, shared memory and message queues. Yet, they offer a different interface to those tools.

System V IPC has been around for a longer time, which has resulted in POSIX IPC to be spread less widely. You usually have a better chance having System V being implemented, than with POSIX IPC.

On the other hand, POSIX IPC has learned from the issues and strengths of the System V API – so for many people POSIX IPC is easier to use. Furthermore, POSIX IPC was created to standardize the interface – the same way any interface in the POSIX-Standard was meant to be a standardization.

According to "UNIX, Third Edition: The Textbook", written by Syed Mansoor Sarwar and Robert M. Koretsky, POSIX IPC Interfaces are multithread safe! This is a huge advantage.

Also, POSIX IPC uses Ascii strings for the names, instead of integers like the System V Interface does. This is more usable IMO.

The System V IPC allows semaphores to be automatically released if a process dies (using SEM_UNDO Flag). So, we can ensure that the semaphore will be reset, in case our process terminates before manually releasing it. This Flag does not exist in POSIX IPC.

System V	**POSIX**
℘ Exists longer, more common ℘ SEM_UNDO Flag ℘ Uses integers as keys ℘ Not Threadsafe	℘ Has learned from System V ℘ POSIX -> is a standard ℘ Uses strings as keys ℘ Multithreadsafe ℘ Easier to use for many

POSIX-Interface

Semaphore

Named Semaphores

Create new semaphore

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


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

This will create a new named semaphore or open an existing named semaphore.

Lock a Semaphore

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem); //non-blocking, errno set to EAGAIN
```

This will lock a semphore, meaning it decrements the semaphore. If the value is already on 0, it will block (or return error if trywait).

Unlock a Semaphore

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

Yeah, this unlocks the semaphore. It will return 0 on success.

Close Semaphore

```
int sem_close(sem_t *sem);
```

This will close the named semaphore. You want to use this, if you your process has finished using the semaphore. The semaphore still remains in the system!

Destroy Semaphore

```
int sem_unlink(sem_t *sem);
```

This will remove the Semaphore completely from the system – but only when the reference count reaches 0. Meaning, all process having the semaphore opened must have either closed the semaphore or been terminated.

Unnamed Semaphores

Create new semaphore

```
#include <semaphore.h> //for semaphore...


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

This will initialize the unnamed semaphore at the address pointed to by sem. sem just has to be a declared variable accessible. So you could do:

```
sem_t *sem; //pass this variable as sem to sem_init
```

The pshared argument indicates whether this semaphore is to be shared between the threads of a process or between processes. If it is 0, then the semaphore is shared between the threads and should be located at some address that is visible to all threads (gloabl variable, malloc'ed). Otherwise it is shared between processes and should be located in a region of shared memory.

The value is just the initial value of the semaphore.

Lock a Semaphore
Works identically to named semaphores

Unlock a Semaphore
Too.

Destroy the Semaphore

```
int sem_destroy(sem_t *sem);
```

When the semaphore is no longer required, and before the memory in which it is located is deallocated, the semaphore should be destroyed using sem_destroy.

Note, that destroying a semaphore that other processes are currently blocked on produces undefined behaviour! But you could theoretically reinitialize the semaphore using sem_init and will have a defined behaviour from this moment on again :^}

Shared Memory

Not much to say here, it works the same way System V Shared Memory does, but the Methods differ:

Create a Shared-Memory-Segment

```
#include <sys/mman.h>
#include <sys/stat.h> /* For mode constants */
#include <fcntl.h> /* For O_* constants */

int shm_open(const char *name, int oflag, mode_t mode);
```

Creates and opens a new POSIX-Shared Memory Object. Those Memory Objects are in fact a handle which can be used by unrelated processes to mmap the same region of shared memory.

oflag ist a bit mask and defines, how the object will be opened:

- O_RDONLY: Read-Only
- O_RDWR: Read-Write access
- O_CREAT: Create Memory Object if it doesn't exist
- O_EXCL: Returns an error, if O_CREAT specified and it already exists
- O_TRUNC: Truncates the memory object to zero bytes if it exists

Close a Shared-Memory-Segment

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

Is the converse operation to shm_open, which will remove an POSIX-ShM-Object.