Note the different room for tonight's help session.

There will be a help/debug/dev code party for CS344 in **KEAR 212** on **Wednesday evening** (February 18th) from **7:00pm to 9:00pm**.

Attendance is completely optional.

The example of my program that is like the one you are working on for Homework #4 can be found in:

/usr/local/classes/eecs/winter2015/cs344-001/src/Homework4

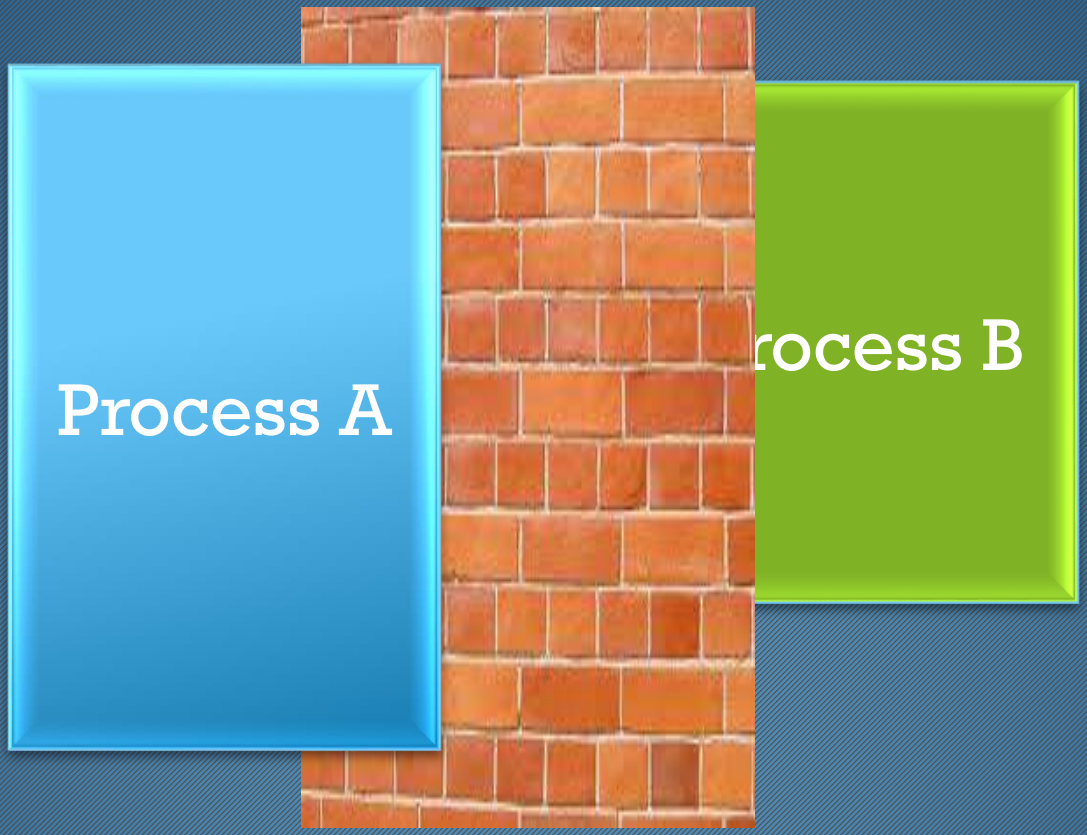
Shared Memory

The Vulcan Mind Meld of IPC

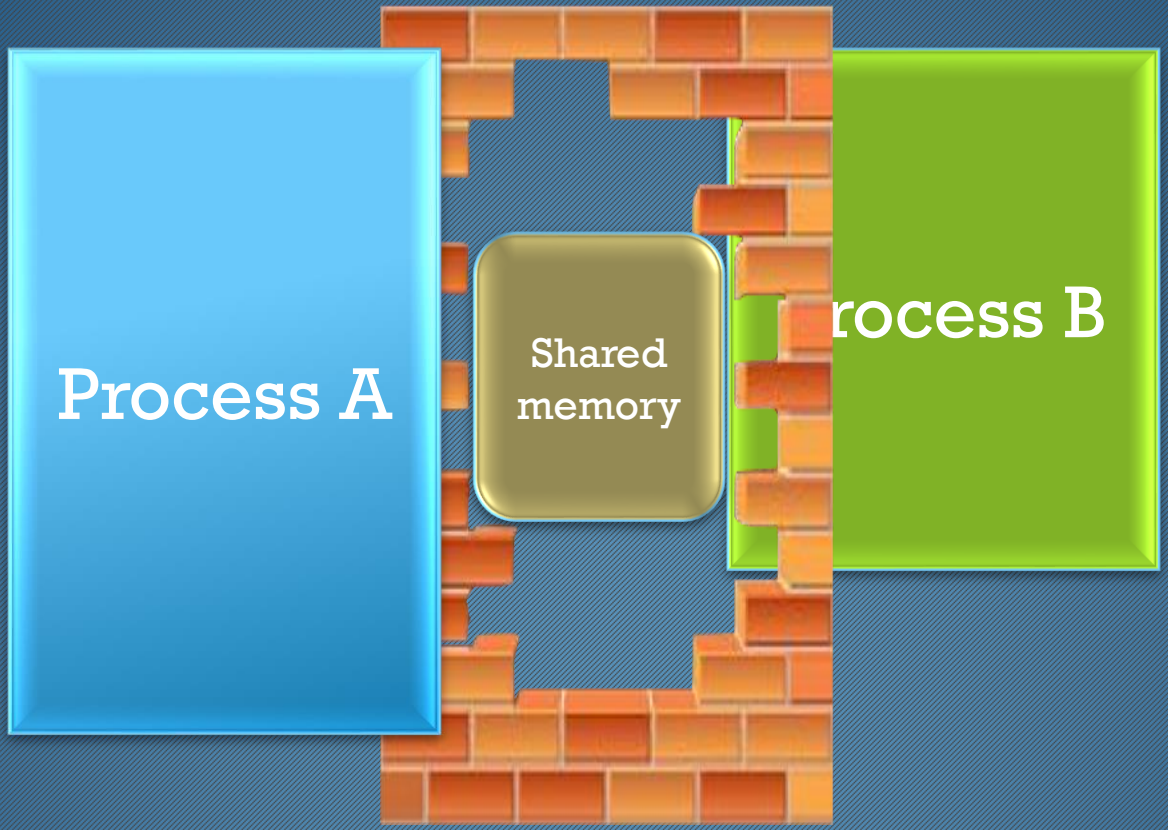
Shared Memory is a form of IPC that allows processes to exchange large amounts of information very quickly.

Once processes maps a **Shared Memory** segment into the process address space, there are no system calls necessary to to share data between the attached processed.

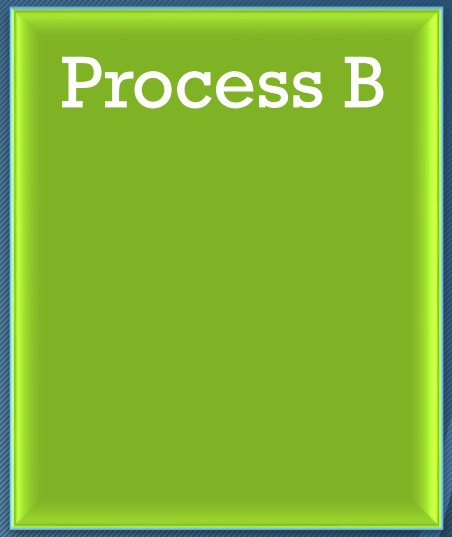
Shared Memory does **not** provide any means of synchronization or mutual-exclusion for multiple or concurrent access. You need to build that into your application. A common method for providing synchronization of access to Shared Memory is using semaphores (covered later).



There's normally a brick wall between the memory of different processes. Process A cannot access or modify the memory of Process B. The brick wall is enforced by the kernel.
Normally, the brick wall is a good thing. It protects applications from accidental corruption from other applications.



Shared Memory allows you to punch a hole in the brick wall and allow processes to share regions of memory. The processes inform the kernel (through system calls) that the processes are ***cooperating*** and that they want to share some memory resources.



Two processes A and B are running on a system and have been coded to coordinate and share data using shared memory. The processes do not need to be related.

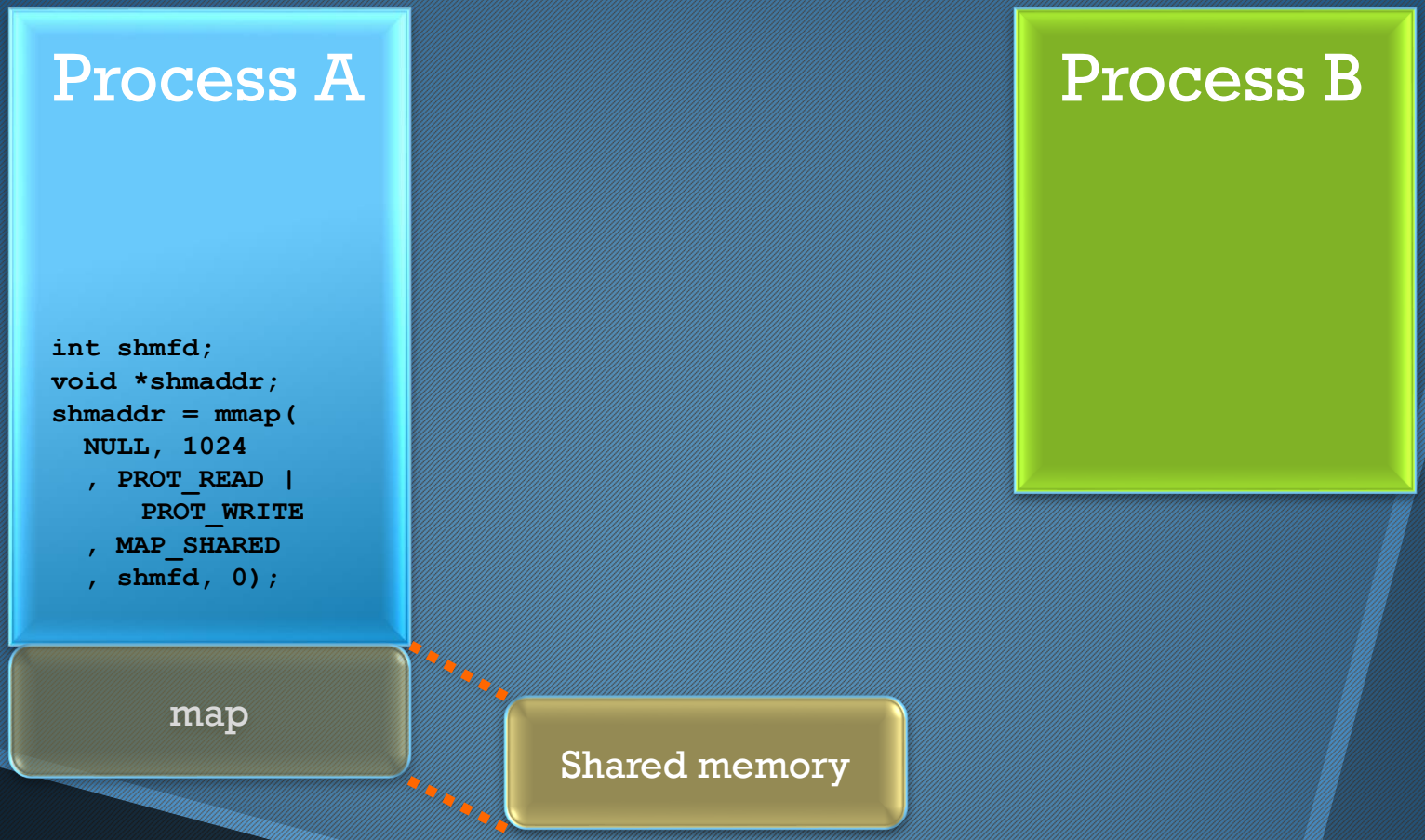
Process A

```
int shmfd;
shmfd = shm_open(
  "/shm-demo"
  , O_RDWR | O_CREAT
  , S_IRUSR | S_IWUSR
);
ftruncate(
  shmfd, 1024);
```

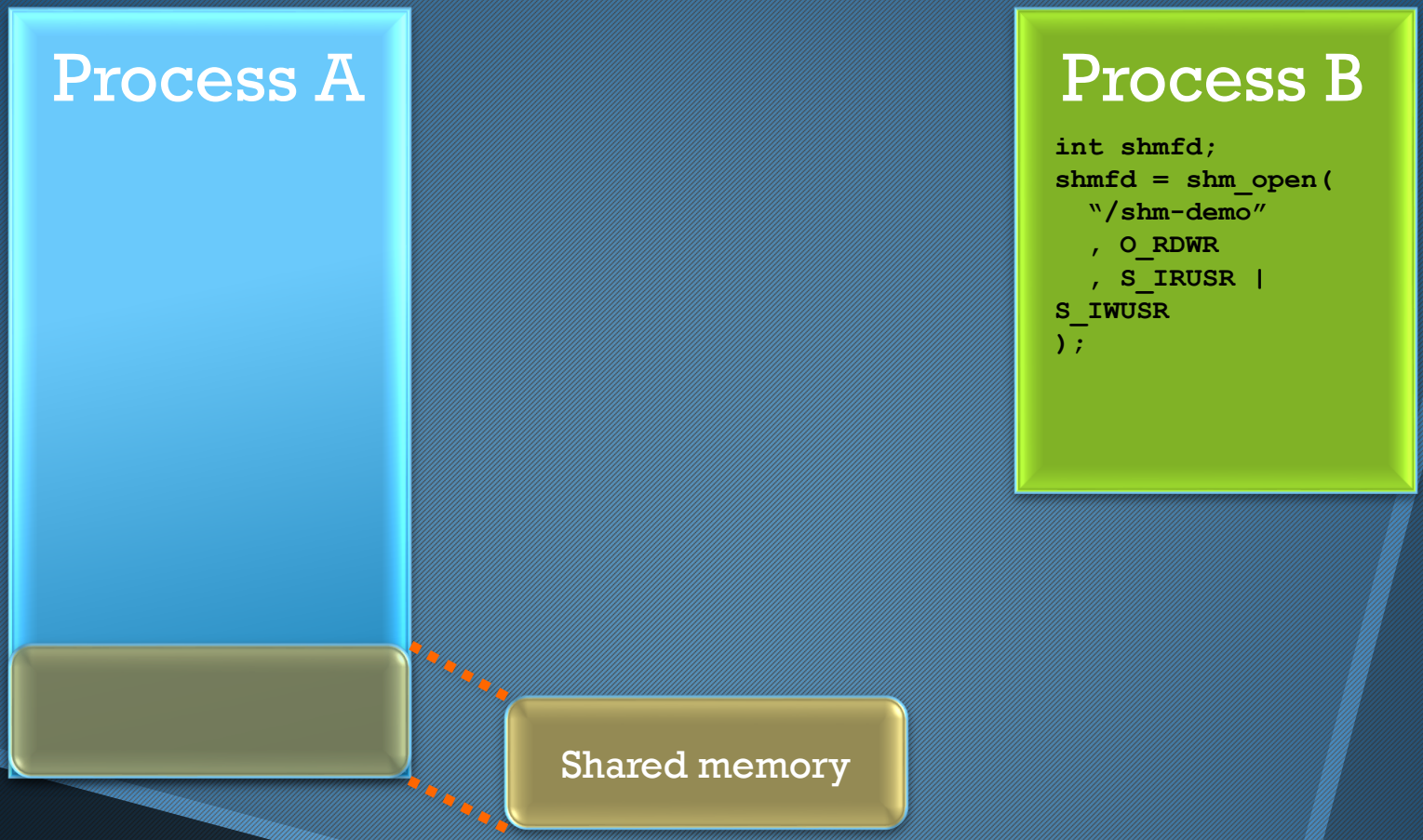
Process B

Shared memory

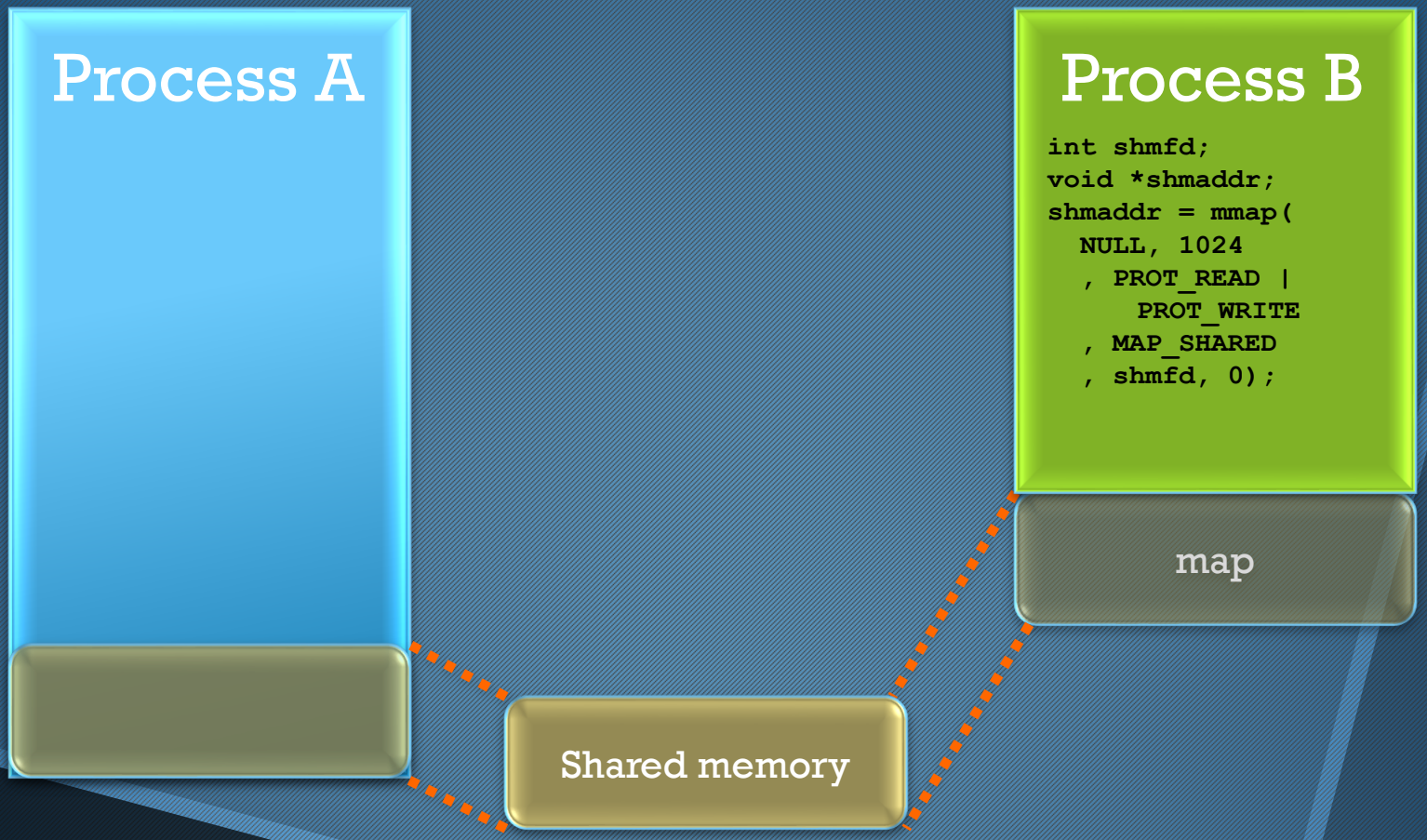
Process A requests a segment of shared memory be created and opened, using the `shm_open()` call. Process A calls `ftruncate()` on the shared memory segment, setting it size to 1024 bytes.



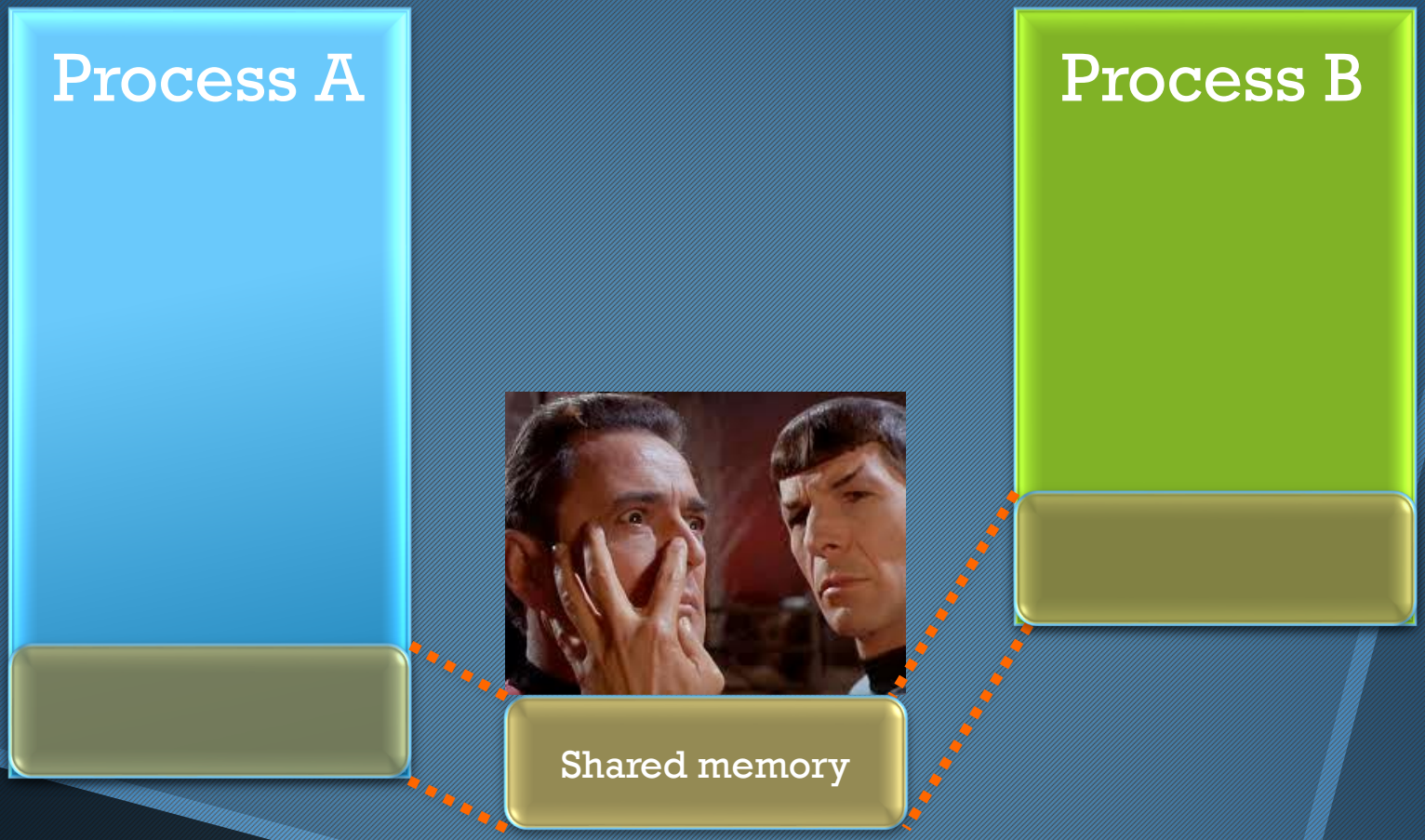
Process A maps the shared memory segment into its own memory space, using `mmap()`. The `shmaddr` pointer points to the beginning of the shared memory segment, **it can be cast** to another pointer type (such as a `struct`) for use in the code.



Process B now opens the existing shared memory segment, using `shm_open()`. Since Process A has already sized the segment, Process B does not need to resize (`fruncate()`) it.



Process B maps the shared memory segment into its own memory space, using mmap(). The pointer shmaddr pointer points to the beginning of the shared memory segment, **it can be cast** to another pointer type (such as a struct) for use in the code.



Process A and Process B can now freely read from and write to the shared memory segment. The shared memory segment is treated the same as local memory in Process A and Process B. The mind meld is active.

Process A

```
int shmfd;
void *shmaddr;

munmap(
  shmaddr, 1024);
close(shmfd);
shm_unlink(
  "shm-demo");
```

Process B

```
int shmfd;
void *shmaddr;

munmap(
  shmaddr, 1024);
close(shmfd);
```

Once Process A and Process B have finished the use of the shared memory segment, each process should un-map it and **one process** should **unlink** the shared memory name. A POSIX shared memory segment will exist until explicitly removed (with the shm_unlink() call) or the system is rebooted.

References

TLPI, chapter 54

http://www.ibm.com/developerworks/aix/library/au-spunix_sharedmemory/

http://www.qnx.com/developers/docs/6.4.0/neutrino/lib_ref/s/shm_open.html

Most of the examples I found on the web are for SysV shared memory. POSIX shared memory calls are a lot easier to use than the SysV calls.

This contains a brief mention of the care you must take with pointers in shared memory.

http://stackoverflow.com/questions/8080055/deep-copy-structures-to-posix-shared-memory