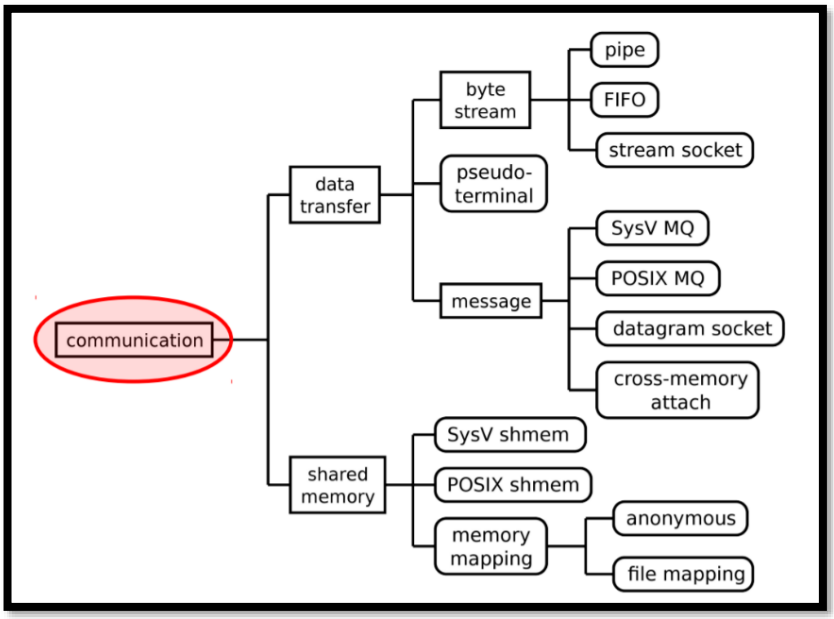
Operating System

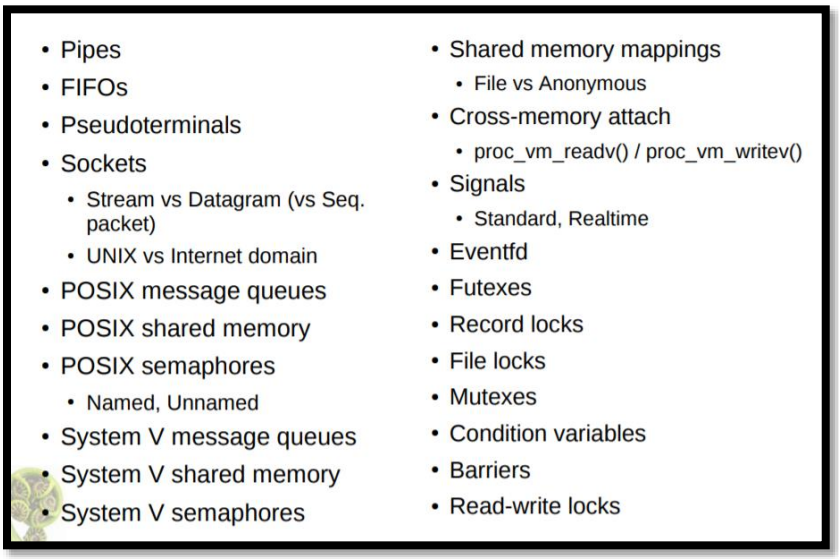
Homework #4 97521036

Question 2 Explanation:

- **This is the section which I want to illustrate the difference between Cross Memory Attach (CMA) and other models of IPCs like Pipe or Shared Memory**

  First of all, let me show a big picture of all IPC's ways:





Operating System

Homework #4 97521036

One of the interesting differences between the different IPC schemes is their mechanism for specifying the **destination for a message**.

CMA **uses a process id (PID) number combined with offsets in the address space of that process – a message is simply copied to that location**. This has the advantage of being very simple and efficient. PIDs are already managed by the kernel and piggy-backing on that facility is certainly attractive. The obvious disadvantage is that there is no room for any sophistication in access control, so messages can only be sent to processes with exactly the same credentials. This will not suit every context, but it is not a problem for the target area (the MPI message passing interface) which is aimed at massively parallel implementations in which all the processes are working together on one task. In that case having uniform credentials is an obvious choice.

CMA achieves single-copy performance by providing a system call which simply copies from one address space to the other with various restrictions as we have already seen. This is simple, but not secure in a hostile environment.

These system calls were designed to permit fast message passing by allowing messages to be exchanged with a single copy operation (rather than the double copy that would be required when using, for example, shared memory or pipes).

{ The system calls process_vm_readv and process_vm_**writev are meant for fast data transfer between processes.** They are supposed to be used in addition to some traditional way of IPC.

For example, you may use a regular pipe or fifo(which is named pipe) to transfer the required addresses between your processes. Then you may use those addresses to establish faster process_vm_ communication. The simplest way to transfer something between forked processes should be the pipe() function (man 2 pipe has a good example of its usage). They just do a single copy of the message that is planned to be sent.}

Operating System

Homework #4 97521036

- **This section we'll talk about pipe mechanism's safety (while reading and writing in it)**

  This question is answered based on asking a friend!

  Unfortunately, I couldn't understand anything which was valuable to write down for this question by googling it

  As I found out:

  Security of using pipe method depends on 3 things:

  - The number of bytes we want to read/write, from/to the pipe
  - The environment we are using pipe in (which is blocking or non-blocking) for file descriptor
  - The number of times we are writing in the pipe

1)In this state, our Buffer has enough free space to get written, and also our environment is blocking, so the data will automatically be written in the buffer if pipe is free, if the pipe it's full, this pipe would be blocked and other pipes will do writing action

2)In this state, our Buffer enough space to get written, but our environment is non-blocking, so the data will automatically be written in the buffer if pipe is free, otherwise if the pipe is working and full, we're going to get an error with "No resource available" statement!

3)In this state, we have blocking environment, and Buffer doesn't have enough space to be written , The write is nonatomic, the data given to write(2) may be interleaved with write(2)s by other process, the write(2) blocks until n bytes have been written.

4)In this state we have non-blocking environment, and Buffer doesn't have enough space to be written, If the pipe is full, then write(2) fails, with errno set to EAGAIN. Otherwise, from 1 to n bytes may be written "partial write" may occur; the caller should check the return value from write(2) to see how many bytes were actually written), and these bytes may be interleaved with writes by other processes.

So last two states may not be safe for piping !

Operating System

Homework #4 97521036

- **This section is about the ability of pthread_mutex_lock wheatear it could be shared between processes or not**

  Which the answer is Yes!

  pthread_mutex_lock could alos be shared between process same as semaphore for example1

  The POSIX threads library has some useful primitives for locking between multiple threads, primarily mutexes and condition variables.

  Typically these are only effective to lock between threads within the same process. However, pthreads defines a PTHREAD_PROCESS_SHARED attribute for both of these primitives which can be used to specify that they should also be used between processes.

  This attribute simply indicates that the primitives be used in a way which is compatible with shared access, however — application code must still arrange to store them in shared memory. This is fairly easily arranged with an anonymous mmap(), or shmget() and shmat() if your processes don't have a parent/child relationship.