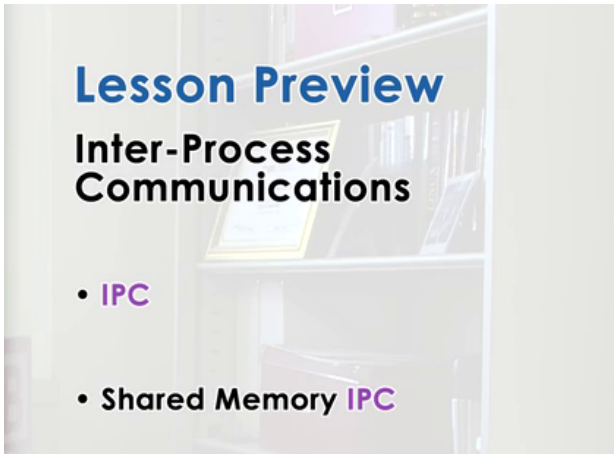
P3L3 - Inter-Process Communication

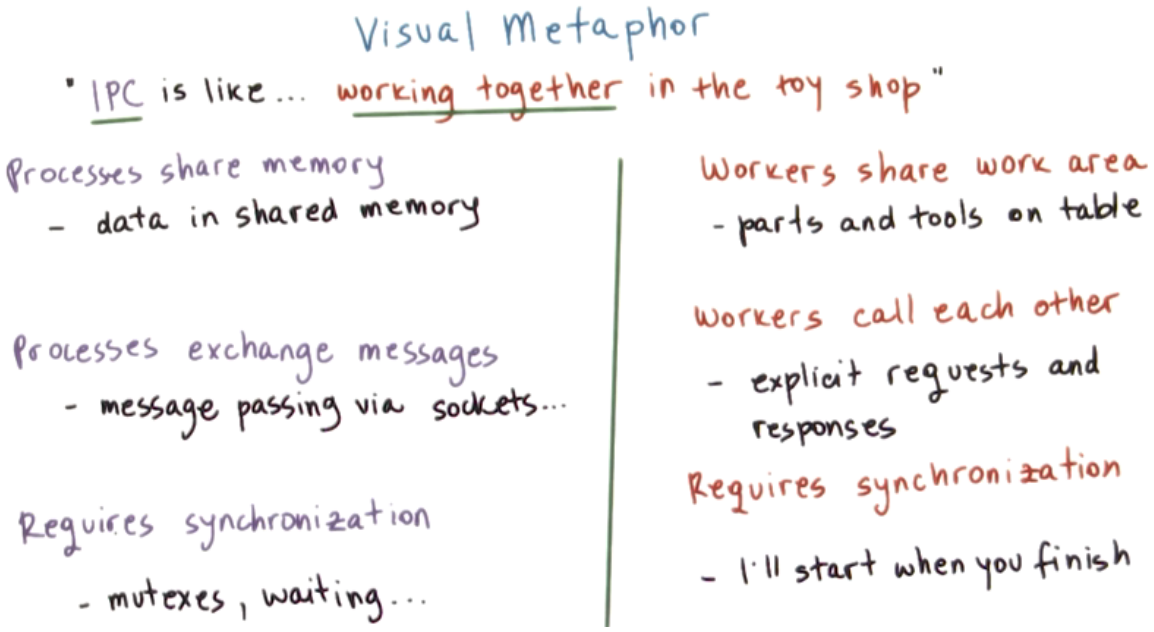
01 - Lesson Preview



In this lesson we will talk about Inter-process Communication or IPC. We will primarily talk about shared memory and describe some of the API's for shared memory based IPC. In addition, we will describe some of the other IPC mechanisms that are common in operating systems today.

02 - Visual Metaphor

In an earlier lesson we described a process like an order in a toy shop. In this lesson we will see how processes can communicate with each other during their execution. But first, let's see how interprocess communication is achieved in a toy shop illustration.

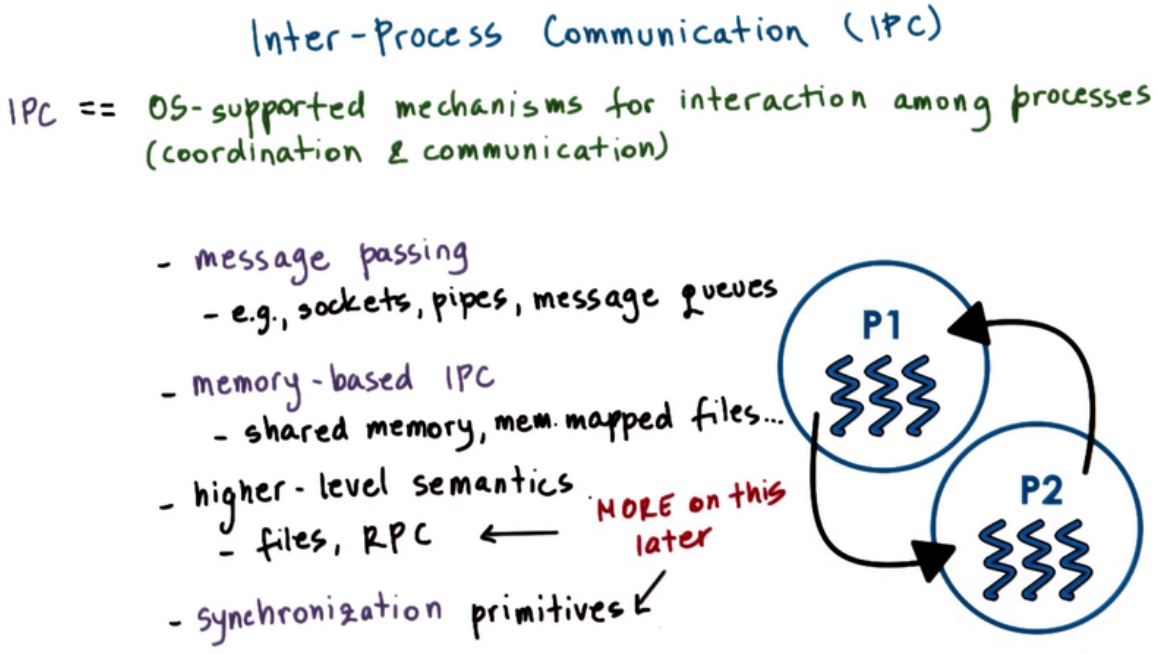


IPC is like working together in a toy shop. First, the workers can share the work areas, the workers can call each other, and finally the interactions among the workers requires some synchronization. Looking at this list, it is fairly obvious that the workers have many options in

terms of how they can interact with one another. When sharing a work area, the workers can communicate amongst each other by leaving common parts and tools on the table to be shared among them. Second, the workers can directly communicate by explicitly requesting something from one another and then getting the required response. And finally, good communication using either one of these methods requires some synchronization so that one worker knows when the other one has finished talking or the other one has finished with a particular tool. One way of thinking about this is that a worker may say I will start a step once you finish yours.

As we will see, processes can also interact amongst each other and work together in similar ways. First, processes can have a portion of physically shared memory and any data they both need to access will be placed in such shared memory. We will discuss this further in this lesson. Second, processes can explicitly exchange messages, requests, and responses via message passing mechanisms that are supported through certain API's like sockets. For both of these methods, processes may need to wait on one another and may need to rely on some synchronization mechanisms like mutexes to make sure that the communication proceeds in a correct manner.

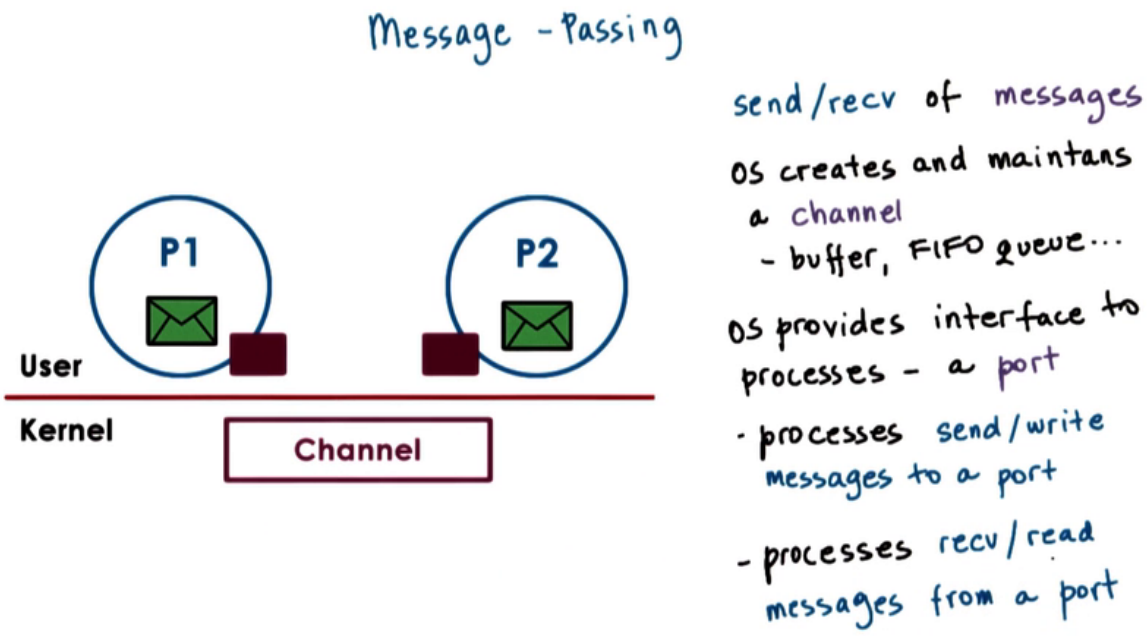
03 - Inter Process Communication



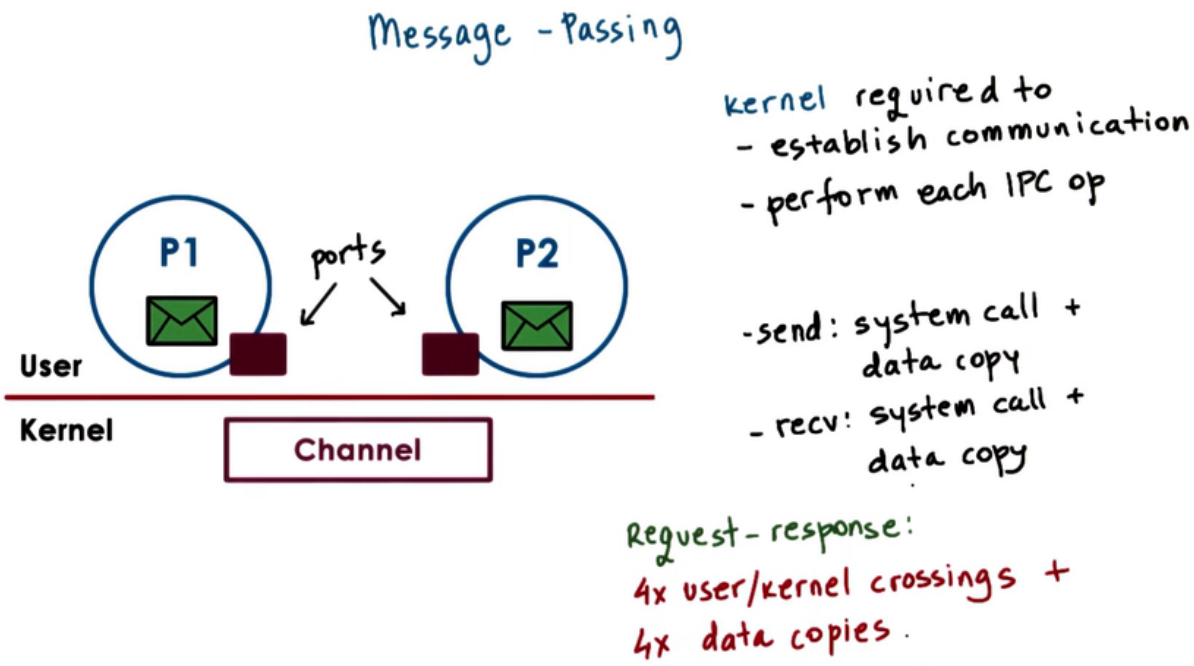
IPC refers to a set of mechanisms that the operating system must support in order to permit multiple processes to interact amongst each other. That means to synchronize, to coordinate, to communicate, all of those aspects of interaction. IPC mechanisms are broadly categorized as message based or memory based. Examples of message passing based IPC mechanisms includes sockets, that most of your are familiar with already, as well as other OS supported constructs like pipes or message queues. The most common memory based mechanism is for the operating system to provide processes with access to some shared memory. This may be in

the form of completely unstructured sets of pages of physical memory or also may be in the form of memory mapped files. Speaking of files, these two can be perceived as a method for IPC. Multiple processes read and write from the same file. We will talk about file systems in a separate lecture. Also, another mechanism that provides higher level semantics when it comes to the IPC among processes is what's referred to as remote procedures calls or RPC. Here, by higher level semantics, we mean that it's a mechanism that supports more than simply a channel for 2 processes to coordinate or communicate amongst each other. Instead, these methods prescribe some additional detail on the protocols that will be used, how will the data be formatted, how will the data be exchanged, etc. RPC too will be discussed in a later lesson in this class. Finally, communication and coordination also implies synchronization. When processes send and receive to each other messages, they in a way synchronize as well. Similarly, when processes synchronize, for instance using some mutex like data structure, they also communicate something about the point in their execution. So from that perspective, synchronization primitives also fall under the category of IPC mechanisms however we will spend a separate lesson talking specifically about synchronization. For that reason, this lesson will focus on the first 2 bullets and we will talk about these remaining topics later.
mak

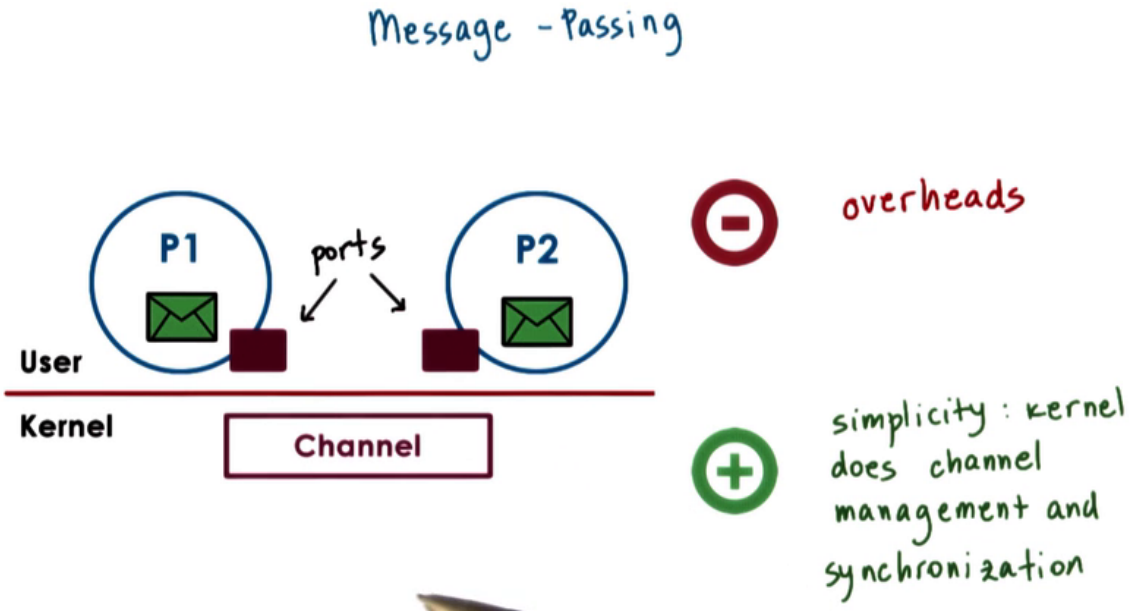
04 - Message Based IPC



One mode of IPC that operating systems support is called message passing. As the name implies, processes create messages and then send or receive them. The operating system is responsible for creating and maintaining the channel that will be used to pass messages among processes. This can be thought of as some sort of buffer, FIFO queue, or other type of data structure. The operating system also provides some interface to the processes so that they can pass messages via this channel. The processes then send or write messages to this port and on the other end, the processes receive or read messages from this port. The channel is responsible for passing the message from one port to the other.



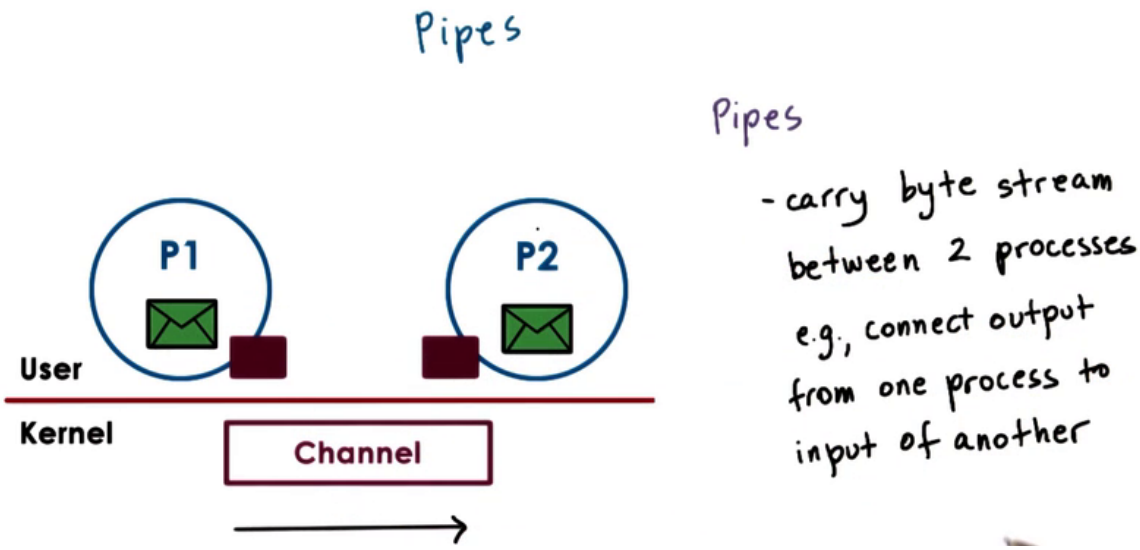
The OS kernel is required to both establish the communication channel as well as to perform every single IPC operation. What that means is that both the send and receive operation require a system call and a data copy as well. In the case of send, from the process address space into the communication channel, and in the case of receive, from this channel into the receiving process address space. What this means is that a simple request-response interaction among two processes will require 4 user-kernel crossings and 4 data copies.



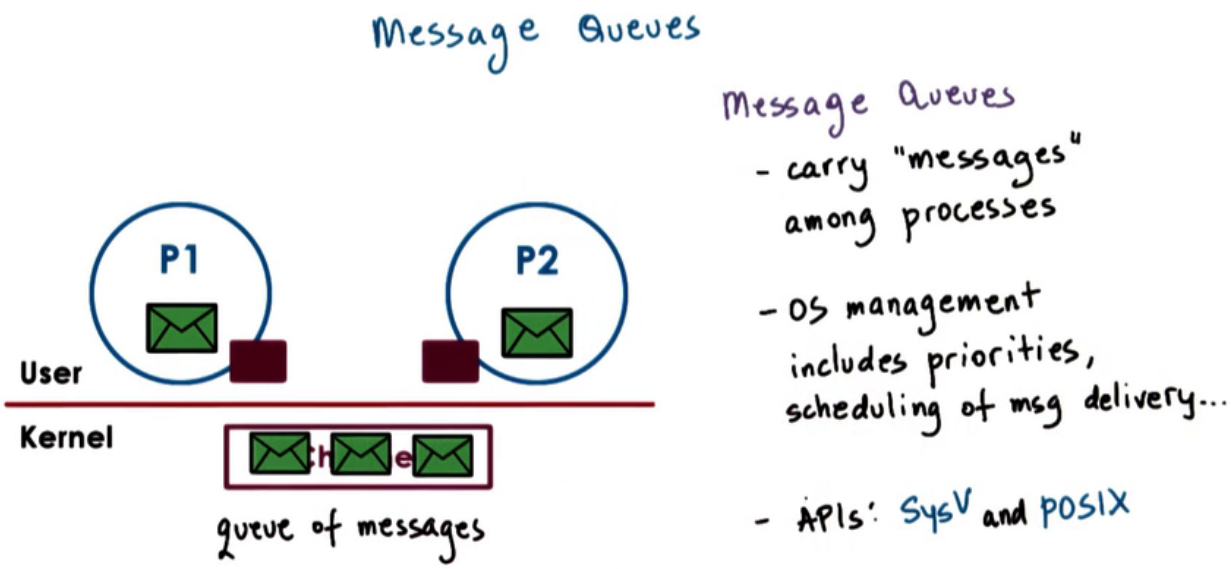
In message passing IPC, these overheads of crossing in and out of the kernel and copying data in and out of the kernel are one of the negatives of this approach. A positive of this approach is the relative simplicity. The operating system kernel will take care of all of the operations regarding the channel, management regarding the synchronization, it will make sure that the data is not overwritten or corrupted in some way as processes are trying to send or receive it, potentially at the same time. That's a plus.

05 - Forms of Message Passing

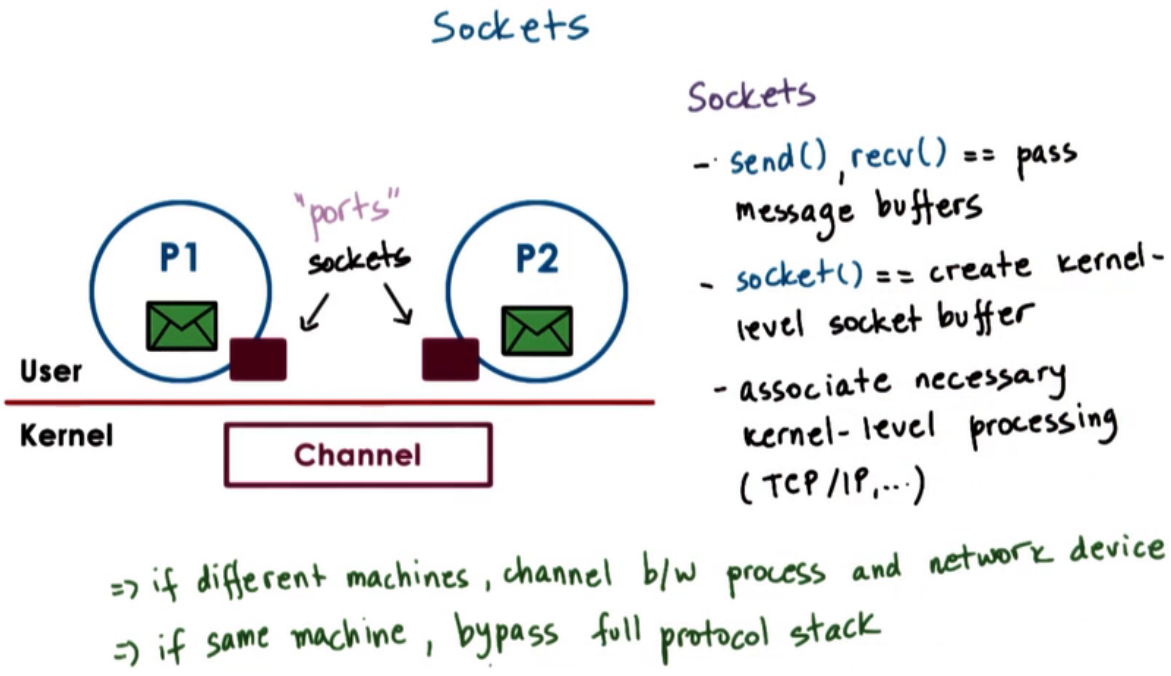
Pipes



In practice, there's several methods of message passing based IPC. The first and most simple form of message passing IPC that's also part of the POSIX standard is called pipes. Pipes are characterized by two endpoints, so only two processes can communicate. There's no notion of a message per se with pipes, instead there's just a stream of bytes that's pushed into the pipe from one process and then received into another. One popular use of pipes is to connect the output from one process to the input of another process. So the entire byte stream that's produced by P1 will be delivered as input to P2, instead of somebody typing it in for instance.



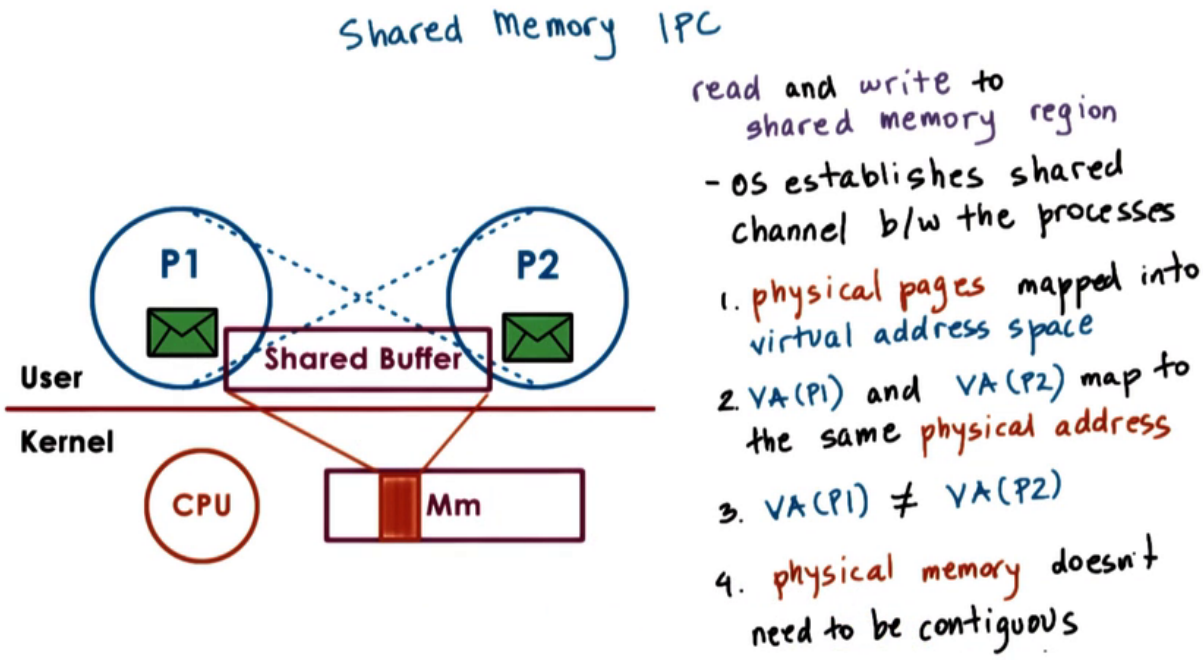
A more complex form of message passing IPC is message queues. As the name suggests, message queues understand the notion of messages that they transfer. So a sending process must submit a properly formatted message to the channel and then the channel will deliver a properly formatted message to the receiving process. The OS level functionality regarding message queues also includes things like understanding priorities of messages, or scheduling the way messages are being delivered, the use of message queues is supported through different API's. In unix based systems these include the POSIX API and the SYSV API.



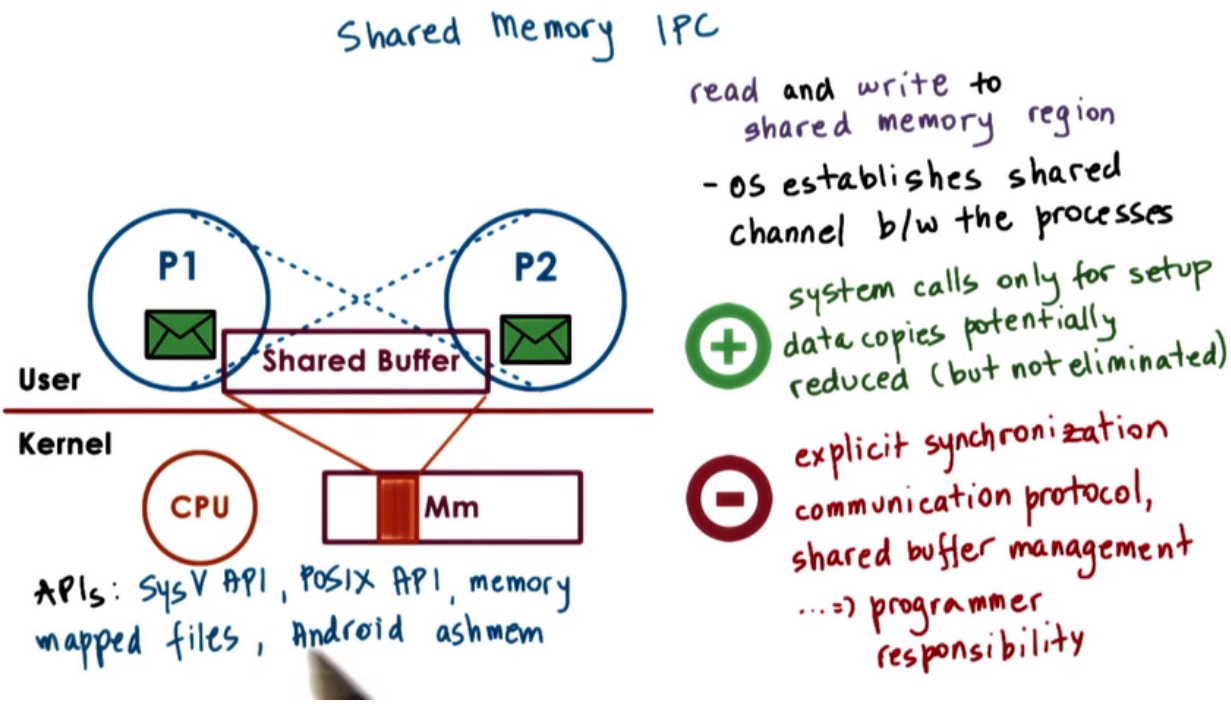
The message passing API that most of you are familiar with is the socket API. With the socket form of IPC, the notion of ports that's required in message passing IPC mechanisms, that is the socket abstraction that's supported by the OS. With sockets, processes send messages or

receive messages via an API that looks like this and then receives. The socket API supports send and receive operations that allow processes to send message buffers in and out of the in-kernel communication buffer. The socket call itself creates a kernel level socket buffer. In addition, it will associate any necessary kernel level processing that needs to be performed along with the message movement. For instance, the socket may be a tcp/ip socket which will mean that the entire tcp/ip protocol stack is associated with the data movement in the kernel. Sockets as you probably know don't have to be used for processes that are on a single machine. If the two processes are on different machines, then this channel is essentially between a process and the network device that will actually send the data. In addition the operating system is sufficiently smart to figure out that if 2 processes are on the same machine, it doesn't really need to execute the full protocol stack, to send the data out on the network, and then just to receive it back, and push it into the process. Instead, a lot of that will be bypassed. This remains completely hidden from the programmer, but you could likely detect it if you perform certain performance measurements.

06 - Shared Memory IPC



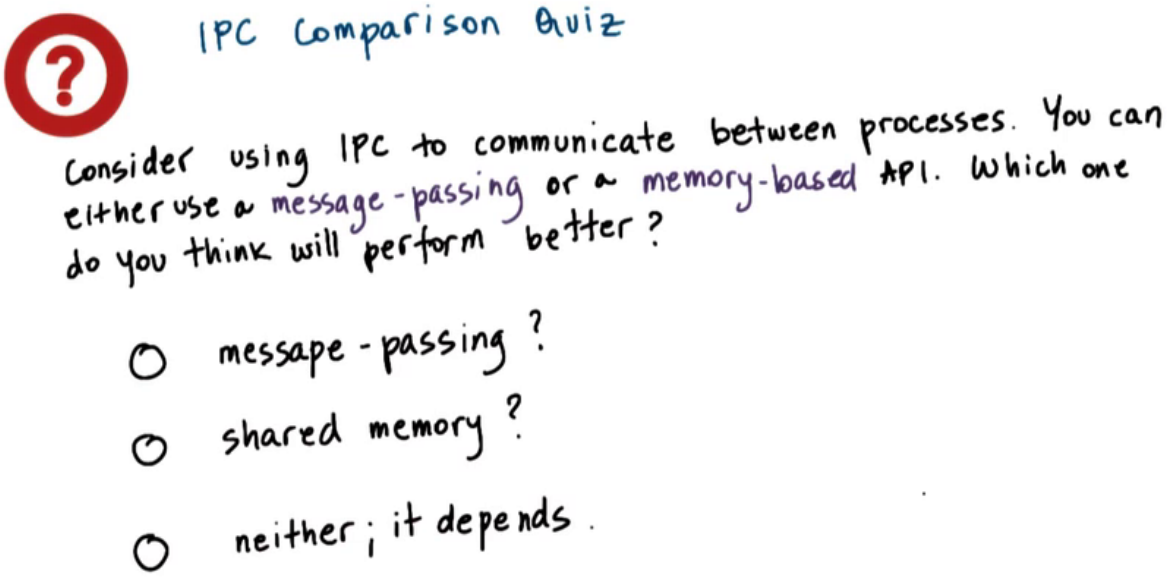
In shared memory IPC, processes read and write into a shared memory region. The operating system is involved in establishing the shared memory channel between the processes. What this means is that the operating system will map certain physical pages of memory into the virtual address spaces of both processes. The virtual addresses in P1 and the virtual addresses in P2 will map to the same physical addresses. At the same time, the virtual address regions that correspond to that shared memory buffer in the two processes, they don't need to have the same virtual addresses. Also the physical memory that's backing the shared memory buffer does not have to be a contiguous portion of physical memory. All of this leverages the memory management support that's available in operating systems and on modern hardware.



The big benefit of this approach is that once the physical memory is mapped into both address spaces, the operating system is out of the way. The system calls are used only in the setup phase. Now data copies are potentially reduced, but not necessarily completely avoided. Note that for data to be visible to both processes, it actually must explicitly be allocated from the virtual addresses that belong to the shared memory region. So if that's not the case, then data within the same address has to be copied in and out of the shared memory region. In some cases however, the number of required copies can be reduced. For instance, if P2 needs to compute the sum of 2 arguments that were passed to it from P1 via the shared memory region, then P2 can only read these arguments, it doesn't actually need to copy them into other portions of its address space, compute the sum, and then pass it back. However there's some drawbacks. Since the shared memory area can be concurrently accessed by both processes, this means that the processes must explicitly synchronize their shared memory operations. Just as what you would have with threads operating within a single address space. Also, it's the developer's responsibility to determine any communication protocol related issue such as how are messages going to be formatted, how will they be delimited, what are their headers going to look like, and those are how the shared memory buffer will be allocated, when which process will be able to use a portion of this buffer for it's needs. So this adds some complexity obviously. Unix based systems including linux support two possible shared memory API's. One of these was originally developed as part of SYS V and the other one is the official POSIX shared memory API. In addition, shared memory based communication can be established between processes using a file based interface, so via memory mapped files in both address spaces. This API is essentially analogous to the POSIX shared memory API. Also the android operating system uses a form of shared memory IPC that's called ashmem. There are a number of differences in the details of how ashmem behaves compared to the SYS V or the POSIX API's,

but I'm just providing it here as a reference only. For the remainder of this lesson, we will focus on briefly describing the unix based shared memory API's.

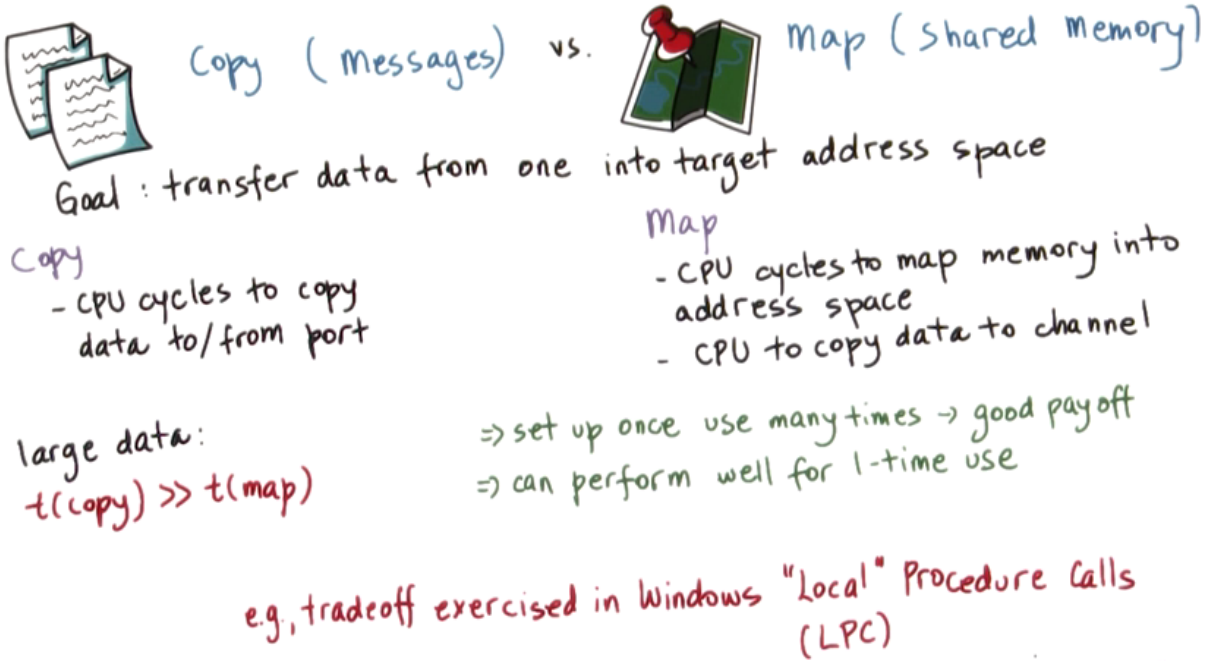
07 - IPC Comparison Quiz



08 - IPC Comparison Quiz

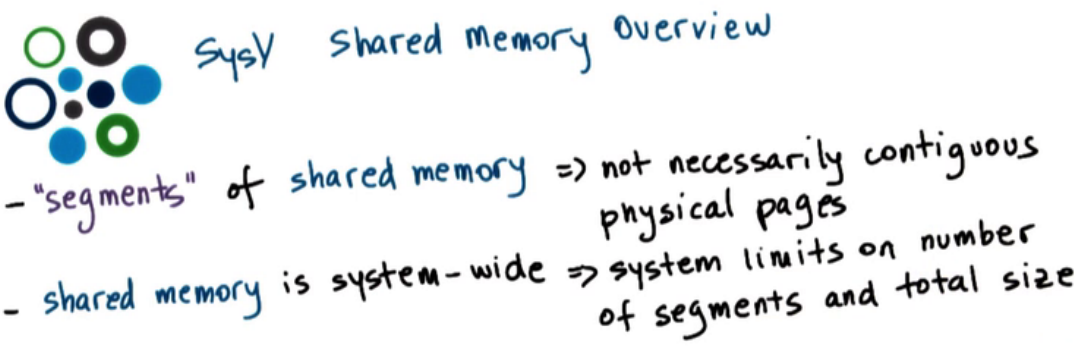
The answer to this question is the "it depends" answer that's common in many systems questions. Here is why. We mentioned that in message passing, multiple copies of the data must be made between the processes that communicate and the kernel. That leads to overhead clearly. For shared IPC, there are lot of costs that are associated with the kernel establishing valid mappings among the processes address spaces and the shared memory pages. Again, these are overheads. So there are drawbacks basically on both sides and the correct answer will be it depends. In the next video we will explain the tradeoffs that exist among these two types of IPC mechanisms.

09 - Copy vs Map

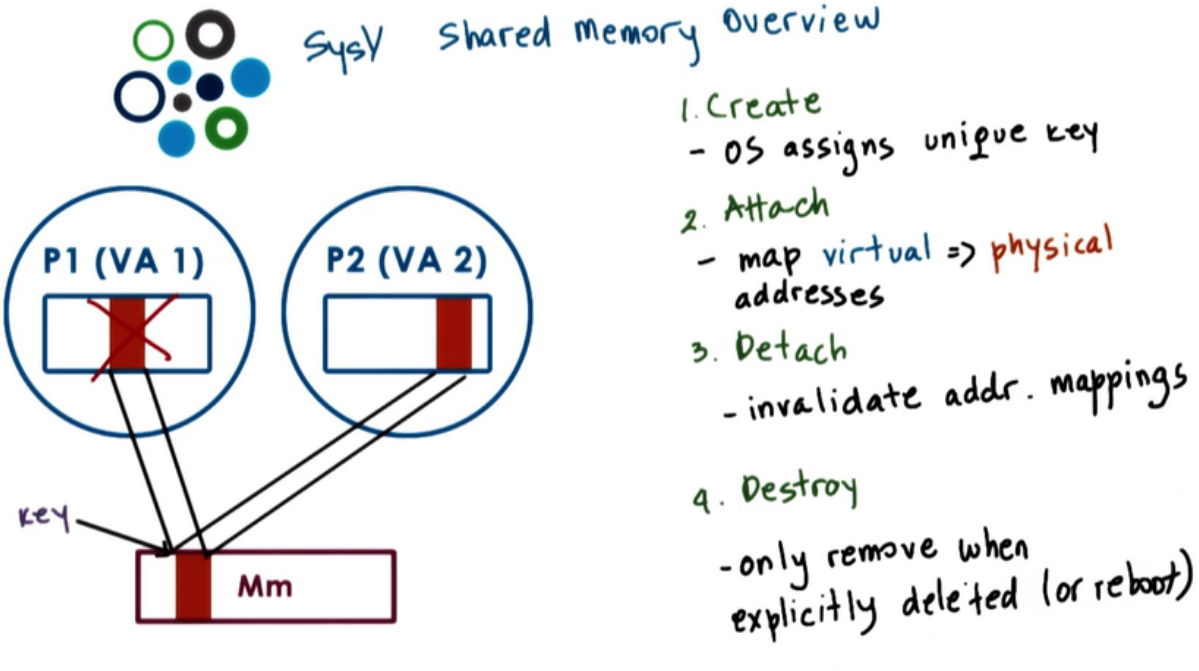


Before I continue, I want to make one important comment to contrast the message based and the share memory based approaches to IPC. The end result of both of these approaches is that some data is transferred from 1 address space into the target address space. In message passing, this requires that the CPU is involved in copying the data. This takes some number of cpu cycles to copy the data into the channel via the port and then from the port and into the target address space. In the shared memory case, at a minimum there are cpu cycles that are spend to map the physical memory into the appropriate address spaces. The cpu is also used to copy the data into the channel when necessary, however note that in this case there are no user to kernel level switches required. The memory mapping operation itself is a costly operation, however if the channel is set up once and used many times then it will result in good payoff. However, even for 1 time use the memory mapped approach can perform well. In particular, when we need to move larger amounts of data from 1 address space into another space, the cpu time that's required to perform the copy can greatly exceed the cpu time that's required to perform the map operation. In fact in windows systems, internally in the communication mechanisms they support between processes, leverage the fact there exists this difference. So, if the data that needs to be transferred among address spaces is smaller than a certain threshold, then the data is copied in and out of a communication channel via port like interface. Otherwise the data is potentially copied once to make sure that it's in a page aligned area and then that area is mapped into the address space of the target process. This mechanisms that the windows kernel supports is called local procedure calls or LPC.

10 - SysV Shared Memory



Now that we described the shared memory mechanisms in a general way, let's look at the specific details of the SYS V unix API. First, the OS supports segments of shared memory that don't necessarily have to correspond to contiguous physical pages. Also the OS treats shared memory as a system wide resource using system wide policies. That means that there's a limit on the total number of segments, the total size of the shared memory. Presently that's not so much of an issue as for instance currently in linux that limit is 4000 segments, however in the past it used to be much less and in certain OS's it was as few as 6 segments. More recent versions of linux had a limit of 128 segments. The OS may also impose other limits as far as the system wide shared memory.



When a process requests that a shared memory segment is created, the OS allocates the required amount of physical memory, provided that certain limits are met and then it assigns to it a unique key. This key is used to uniquely identify the segment within the OS. Any other process can refer to this particular segment using this key. If the creating process wants to communicate with other processes using shared memory then it will make sure that they learn this key in some way, by using some other form of IPC or just by passing it through a file or as a

command line argument or maybe other options. Using the key, the shared memory segment can be attached by a process. This means that the OS establishes valid mappings between the virtual addresses that are part of that process virtual address space and the physical memory that backs the segment. Multiple processes can attach to the same shared memory segment and in this manner, each process ends up sharing access to the same pages. Reads and writes to these pages will be visible across the processes just like when threads share access to memory that's part of the same address space and also the shared memory segment can be mapped to different virtual addresses in the different processes. Detaching a segment means invalidating the address mappings for the virtual region that corresponded to that segment within the process. In other words, the page table entries for those virtual addresses will no longer be valid. However a segment isn't really destroyed once it's detached. In fact a segment may be attached and detached and reattached multiple times by different processes during its lifetime. What this means is that once a segment is created, it's like a persistent entity until there's an explicit request for it to be destroyed. This is similar to what would happen to a file. We create file and then the file persists until it's explicitly deleted. In the meantime, we can open it and close it and read it and write to it but the file will still be there. This property of shared memory, to be removed only when it's explicitly deleted or when there is a system reboot makes it very different than regular non shared memory that is malloc'd and then it will disappear as soon as the process exits.

11 - SysV Shared Memory API

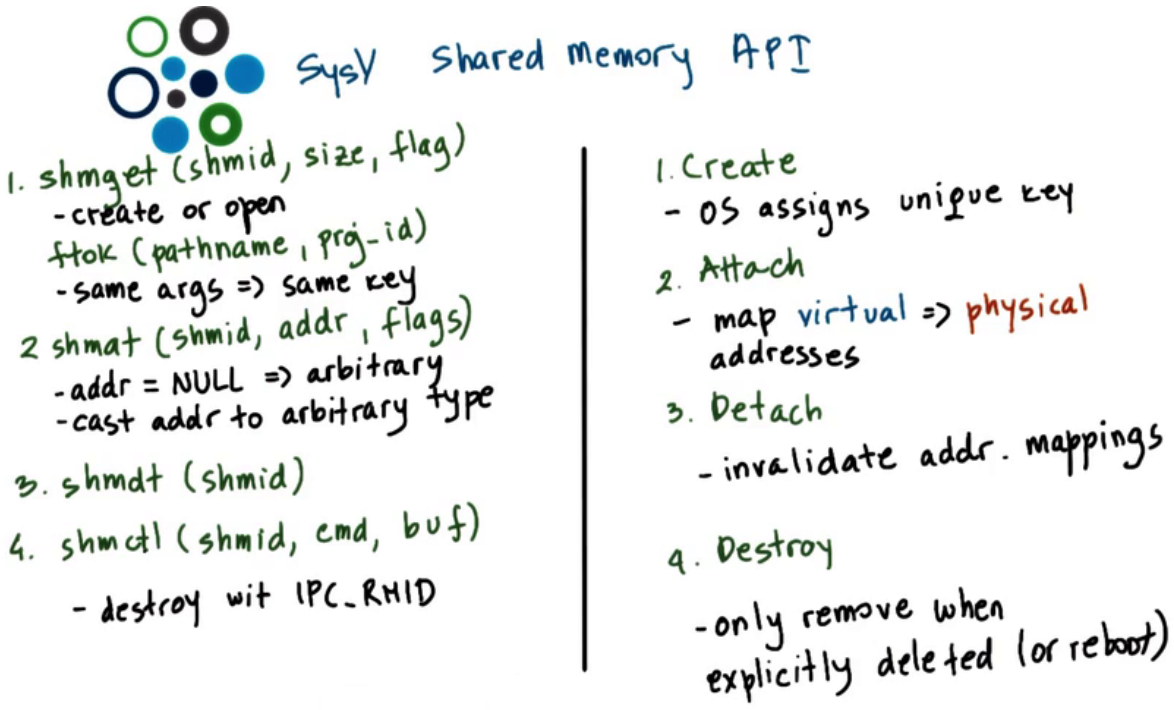
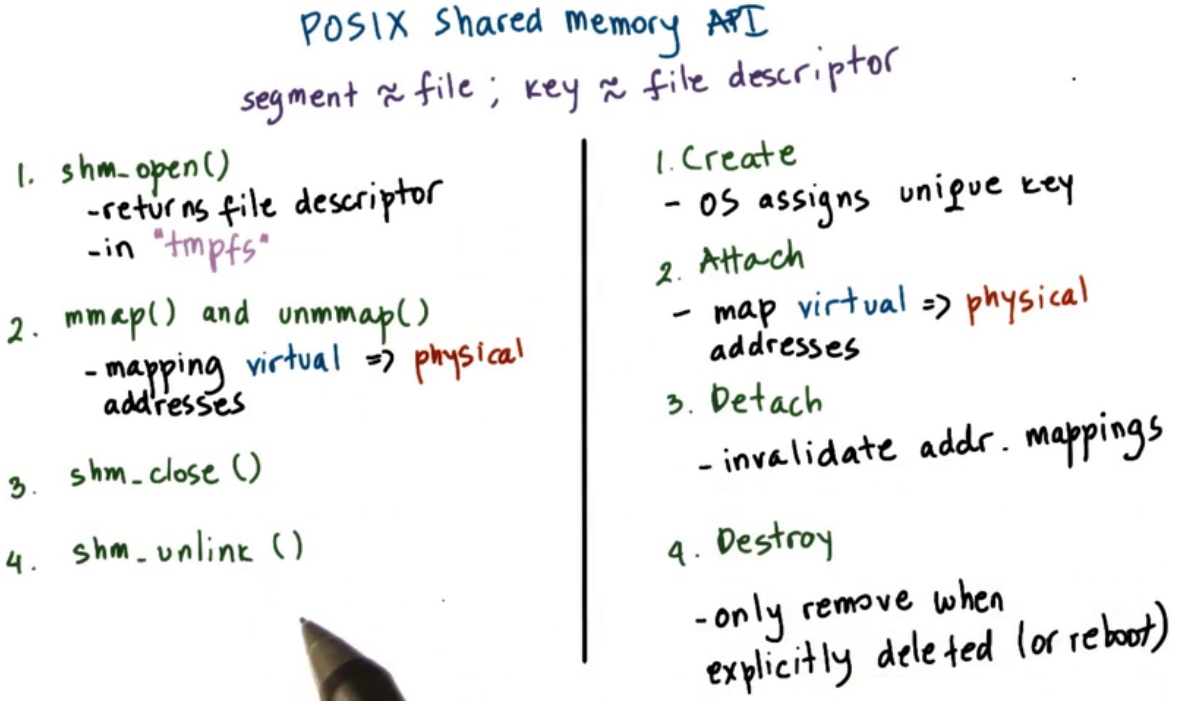


Image Errata: 1. shmget - first argument should be **key**, return value should be **shmid**.
wit = with

SYS V uses the following shared memory API for the high level operations we just discussed. shmget is used to create or open a segment of the appropriate size. The flags include the various options like permissions. This unique identifier is the key and that is not actually magically created by the OS, instead it is explicitly passed to the OS by the application. To generate a unique identifier, the API relies on another operation ftok which generates a token based on its arguments. If you pass to this operation the same arguments, you will always get the same keys. So it's like a hash function. This is how different processes can agree upon how they will obtain a unique key for the shared memory segment they will be using to communicate. The following call attaches the shared memory segments into the virtual address space of the process. So it will map them into the user address space. The programmer has an option to provide the specific virtual addresses for the segment should be mapped, or if NULL is passed then the OS will choose and return some arbitrary suitable addresses that are available in the process's address space. The returned virtual memory can be interpreted in arbitrary ways. So it is the programmer's responsibility to cast that address to that memory region to the appropriate type. The following operation detaches the segment identified by this identifier so the virtual to physical memory mappings are no longer valid and then finally the control operation that the shared memory API supports is used to pass certain commands related to the shared memory segment management to the OS including the command to remove a particular segment and that command is ipc_rmid.

12 - POSIX Shared Memory API

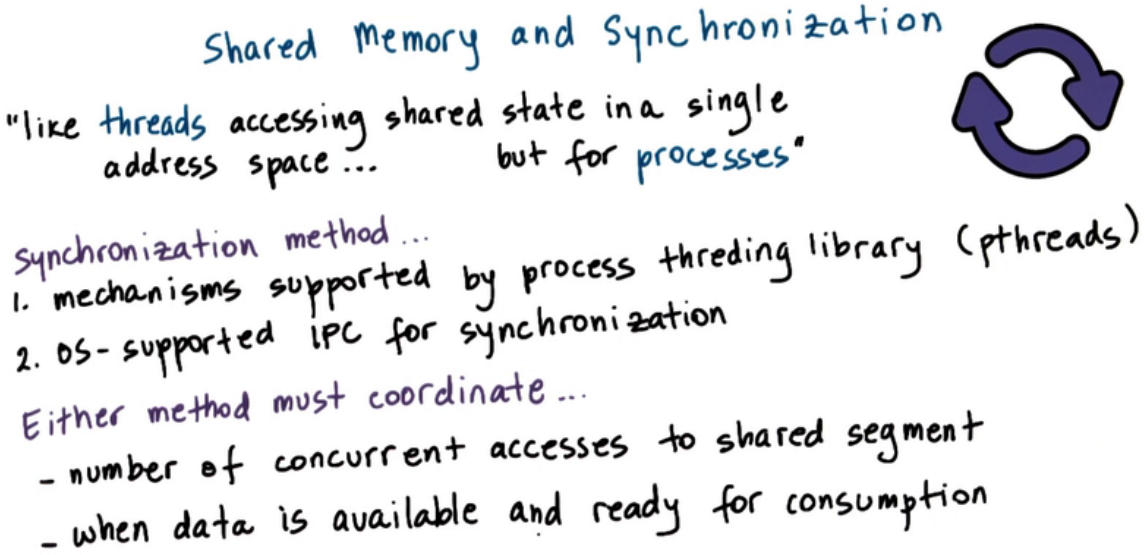


There is also the POSIX API for shared memory on linux system. It has been supported since the 2.4 kernel. Although it's supposed to be the standard, the POSIX API is not as widely supported as for instance the SYS V API. Here is the API. The most notable difference is that the POSIX shared memory standard doesn't use segments. Instead it uses files. Now these are not real files that exist on some file system that's used otherwise by the OS. Instead these are files that only exist in the so called "tmp" file system which is really intended to look and feel like a file system so the OS can reuse the same type of mechanisms that it used for file systems, but in essence, it's just a bunch of state that's present in physical and volatile memory. The OS simply uses the same representation and same data structures that are used for representing a file to represent bunch of pages that correspond to a shared memory region. For this reason, there is no longer a need for the awkward key generation process. Instead shared memory segments can be referenced by the file descriptor that corresponds to the file and then the rest of the operations are analogous to what you'd expect to exist for files. A segment is opened or closed so they are explicit shared memory open and close operations, but in fact you can really only just call the regular open and close operation since you will anyways pass a file and the OS will manage to figure out which file system this file sits in. To attach or detach shared memory, the POSIX shared memory API relies on the mmap and unmmap calls that are used to map or unmap files into the address space of a process. To destroy a shared memory segment, there is an explicit unlink operation. There is also a shared memory close and this will remove the file descriptor from the address space of the process but in order to tell the OS to delete all of the shared memory related data structures and to free up that shared memory segment you just call the explicit unlink operation. I have provided a link to the reference of the POSIX shared memory API in the instructor notes.

Instructor Notes

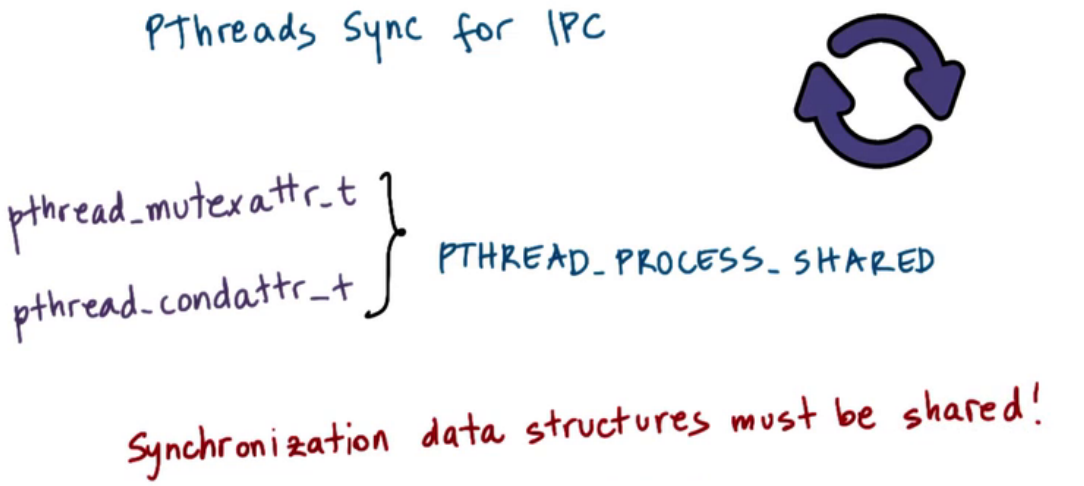
[POSIX Shared Memory API]

13 - Shared Memory and Sync



When data is placed in shared memory it can be concurrently accessed by all processes that have access to that shared memory region. Therefore such accesses must be synchronized in order to avoid race conditions. This is analogous to the manner in which we synchronize threads when they're sharing an address space however it needs to be done for processes as well. So we still must use some synchronization constructs such as mutexes and condition variables for processes to synchronize when they're accessing shared data. There are a couple of options in how this interprocess synchronization can be handled. First, one can rely on the exact same mechanisms that are supported by the threading libraries that can be used within processes. So for instance, two pthreads processes can synchronize amongst each other using pthreads mutexes and condition variables that have been appropriately set. In addition, the OS itself supports certain mechanisms for synchronization that are available for interprocess interactions. Regardless of the method that is chosen there must be mechanisms to coordinate the number of concurrent accesses to the shared memory region, for instance for support for mutual exclusion, mutexes provide this functionality, and also to coordinate when is data available in the shared memory segment and ready to be consumed by the peer processes. This is some sort of notification or signaling mechanism and condition variables are an example of a construct that provides this functionality.

14 - PThreads Sync for IPC



When we talked about pthreads, we said that one of the attributes that's used to specify the properties of the mutex or the condition variable when they're created is whether or not that synchronization construct is private to a process or shared among processes. The keyword for this is pthread_process_shared. So when synchronizing shared memory accesses to pthreads multi threaded processes, we can use mutexes and condition variables that have been correctly initialized with pthread_process_shared status. One important thing however is that the synchronization variables themselves have to be also shared. Remember in multi threaded programs the mutex or condition variables have to be global and visible to all threads. That's the only way they can be shared among them. So it's the same rationale here. In order to achieve this, we have to make sure that the data structures for the synchronization constructs are allocated from the shared memory region that's visible to both processes.

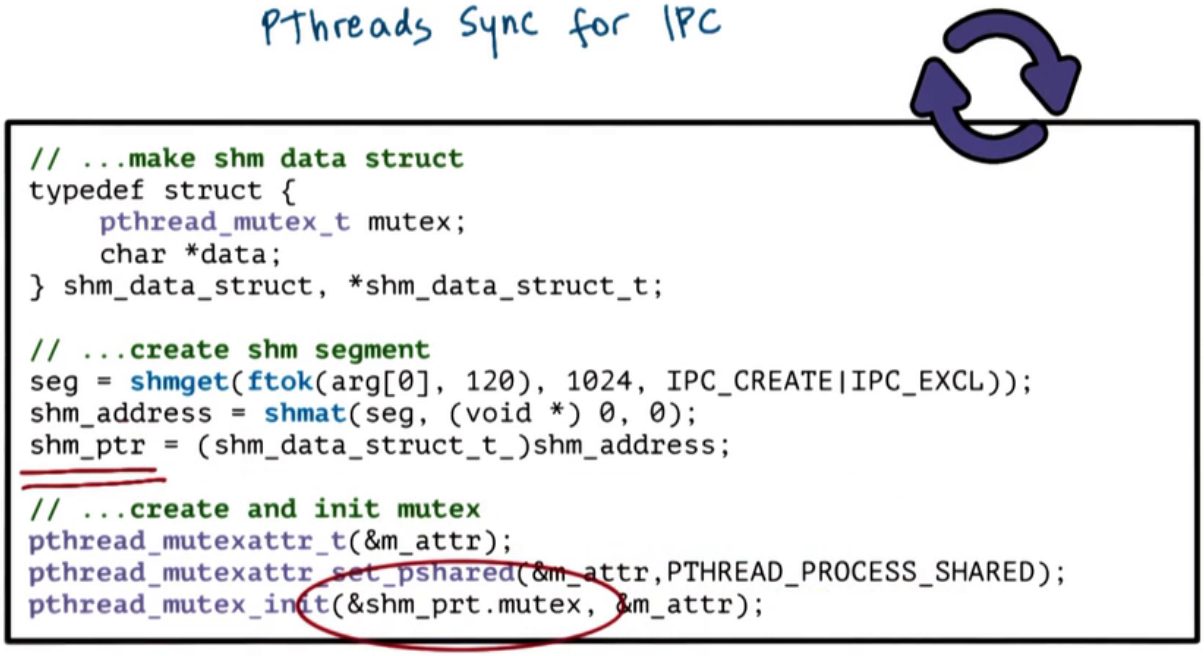
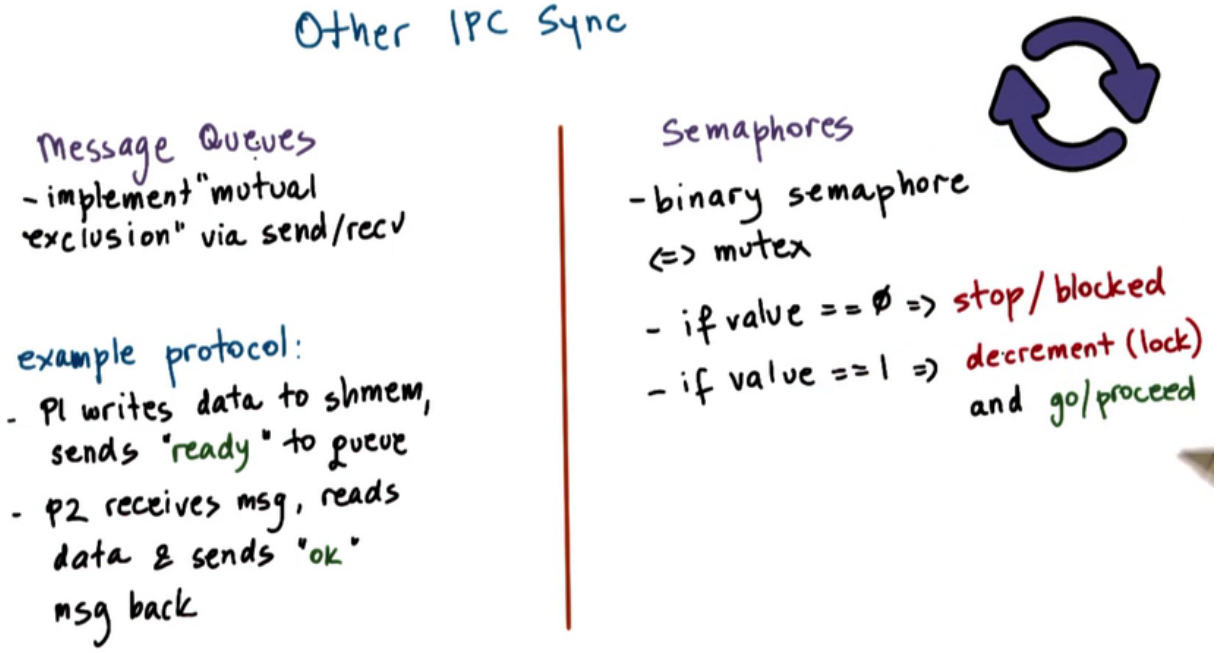


Image Errata: arg = arg**v**, shm_data_struct_t_ = shm_data_struct_t, pthread_mutexattr_t**(&m_attr)**; = pthread_mutexattr_t **m_attr**;, pthread_mutex_init(&shm_**prt**.mutex, &m_attr); = pthread_mutex_init(&shm_**ptr**.mutex, &m_attr);

For instance, let's look at this code snippet. Let's look here at how the shared memory segment is created. Here we're using the SYS V API. In the get operation the segment ID, the shared memory identifier is uniquely created from the token operation where we used arg[0] from the command line so the pathname for the program executable and some integer parameter, so in this case it's 120. We're also requesting that we create a segment size of 1024 and then we specify various permissions for that segment. Using that segment identifier that's returned from the get operation we're attaching the segment and that will provide us with a shared memory address. So this is the virtual memory address in this instance of the process, so in the execution of this particular process, in its address space, that points to the physically shared memory. Now we're casting that address to point to something that's of the following data type. If we take a look at this data type, this is the data structure of the shared memory area that's shared among processes. It has two components. One component is the actual byte stream that corresponds to the data. The other component is actually the synchronization variable, the mutex that will be used among processes when they're accessing the shared memory area, when they're accessing the data that they care for, so as to avoid concurrent writes, race conditions, and similar issues. So this is how we will interpret what's laid out in the shared memory area. Now, let's see how this mutex here is created and initialized. First of all we said that before creating a mutex we must create its attributes and then initialize the mutex with those attributes. Concerning the mutex attributes, we see that we have here set the pthread_process_shared_attribute for this particular attribute data structure. Then we initialize the mutex with that attribute data structure so it will have that property. Furthermore, notice that the location of the mutex we pass to this initialization call is not just some arbitrary mutex in the process address space. It is this particular mutex element that is part of the data structure in shared memory. This set of operations will properly allocate and initialize a mutex that's shared among processes and a similar set of operations should be used also to allocate and initialize any condition variables that are intended for shared use among processes. Once you have properly created and allocated these said data structures, then you can use them just as regular mutexes and condition variables in a multi threaded pthreads process. So there's no difference in their actual usage given that they're used across processes. The key again, let me reiterate, is to make sure that the synchronization variable is allocated within the shared memory region that's shared among processes.

15 - Sync for Other IPC



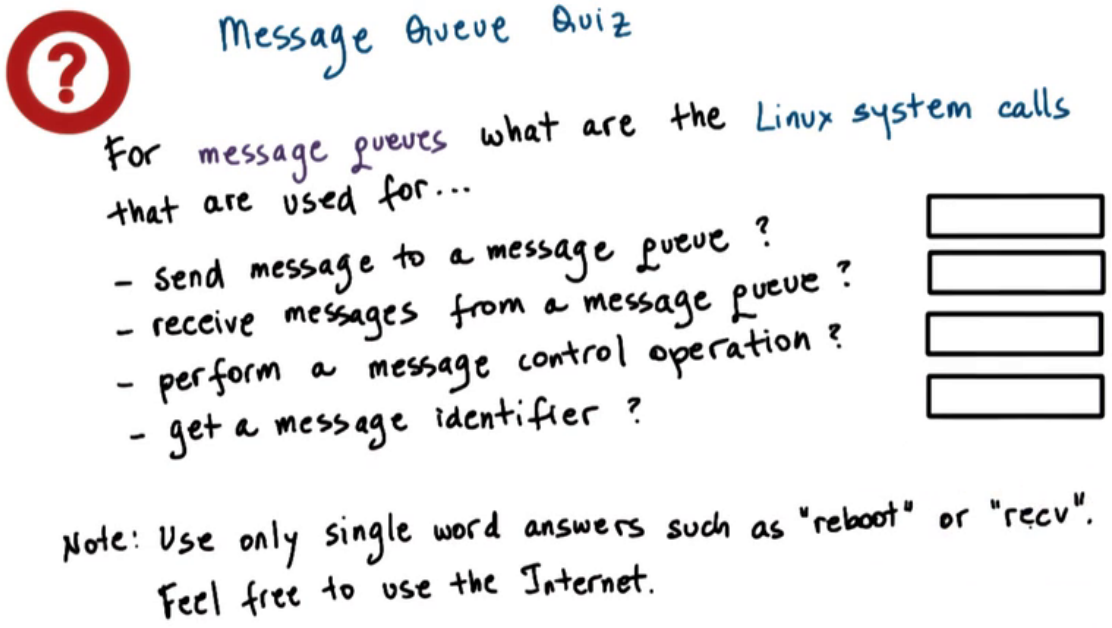
IPC Resources

- SysV IPC Tutorials
  - has example source for SysV IPC
- mq_notify() man page
  - registers for notification when a message is available
- sem_wait() man page
  - locks a semaphore
- shm_overview man page
  - overview of POSIX shared memory

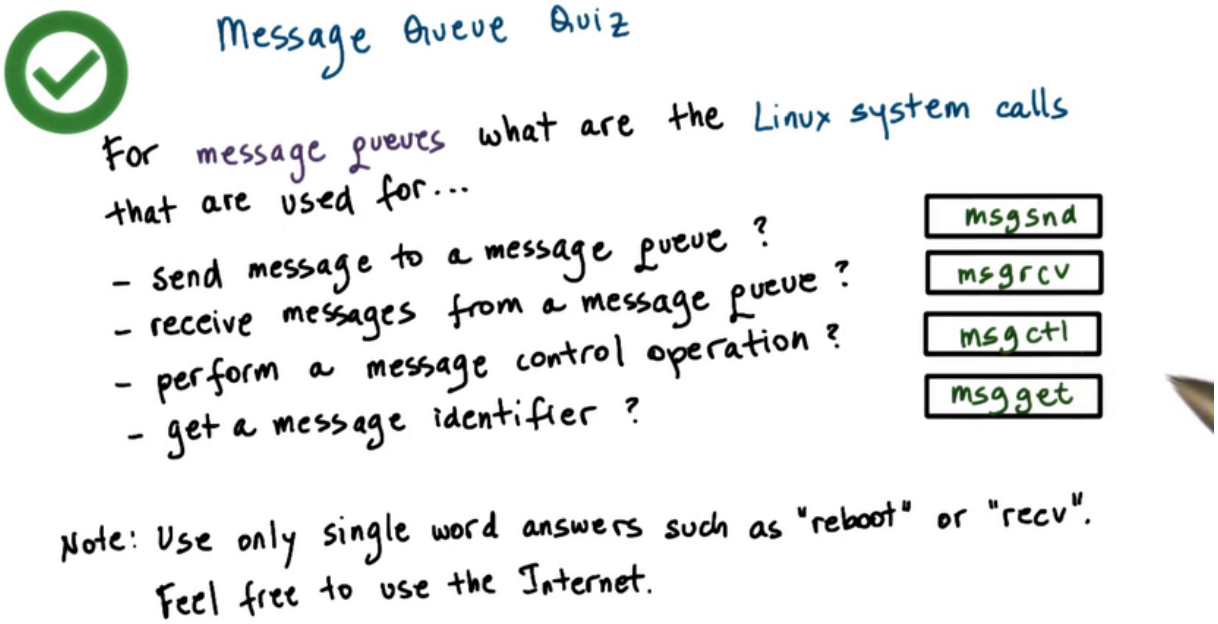
In addition, shared memory accesses can be synchronized using OS provided mechanisms for interprocess interactions. This is particularly important because the process shared option for the mutex condition variables with pthreads isn't necessarily always supported on every single platform. Instead we rely on other forms of IPC for synchronization such as message queues or semaphores. With message queues for instance, we can implement mutual exclusion via send receive operations. Here is an example protocol on how this can be achieved. 2 processes are communicating via shared memory and they're using message queues to synchronize. The first process writes to the data that's in shared memory and then it sends a ready message on the message queue. The second process receives that ready message, knows that it is ok to read the data from the shared memory, and then it sends another type of response, an ok message, back to P1. Another option is to use semaphores. Semaphores are an OS supported synchronization construct and a binary semaphore can have two values, 0 or 1. And it can be achieve a similar type of behavior like what is achieved with a mutex depending on the value of

a semaphore, a process is either allowed to proceed or it will be stopped at the semaphore and will have to wait for something to change. For instance a binary semaphore with value 0 and 1 we can use it in a following way… If it's value is 0, the process will be blocked. And if its value is 1, the semantics of the semaphore construct is such that a process will automatically decrement that value, so it will turn it to zero, and it will proceed. So this decrement operation is equivalent to obtaining a lock. In the instructor's note, I'm providing a code example that uses shared memory and message queues and semaphores for synchronization and the example uses the SYS V API's as a reference. The SYS V API's for these two IPC mechanisms is really somewhat similar to those that we saw for shared memory in terms of how you create and close, etc, message queues or semaphores. For both of these constructs there are also POSIX equivalent API's.

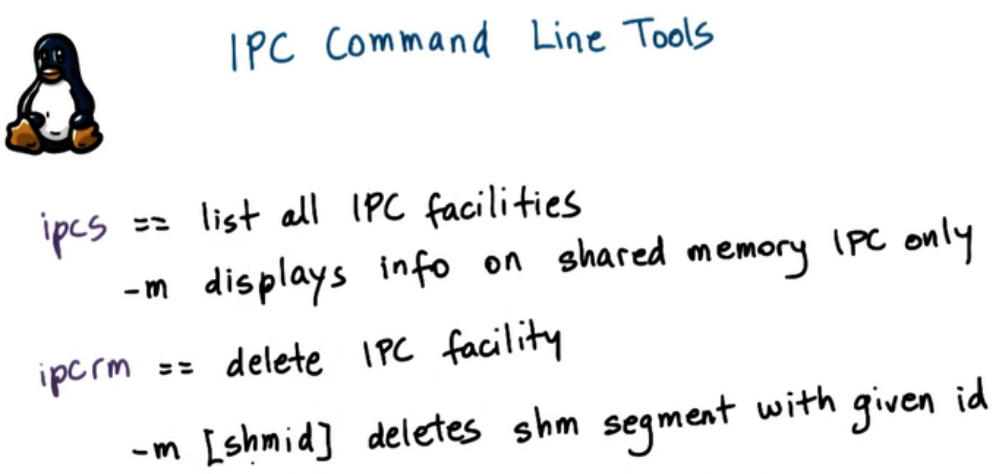
16 - Message Queue Quiz



17 - Message Queue Quiz

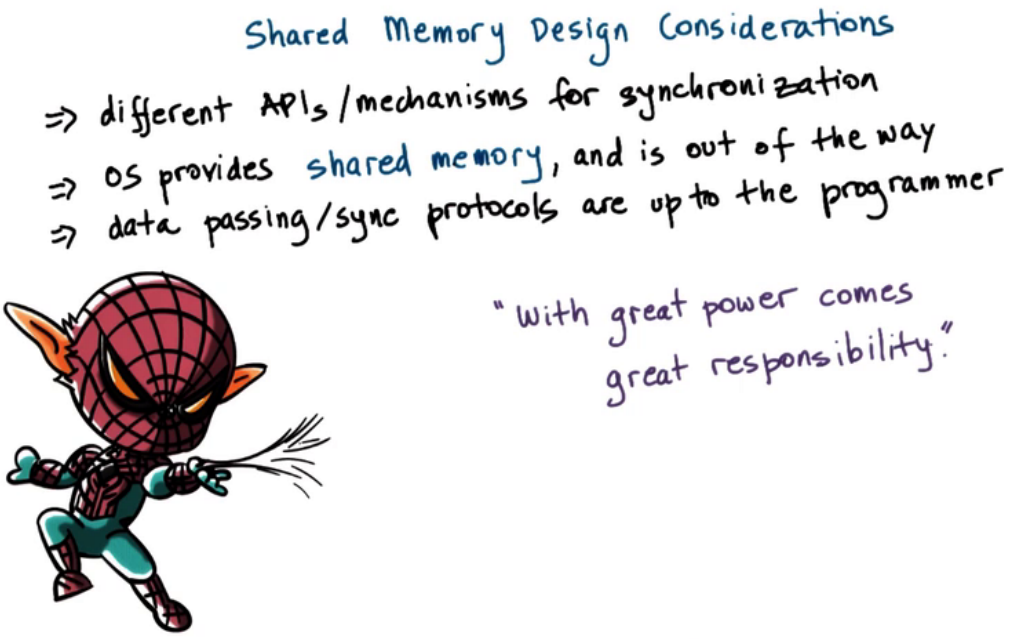


18 - IPC Command Line Tools



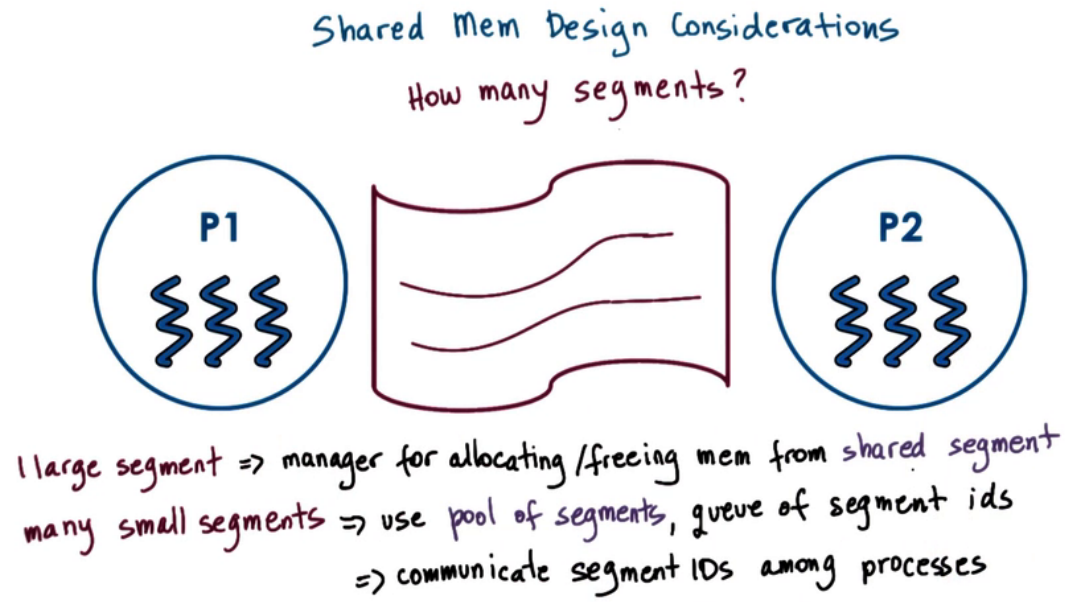
As you start using IPC methods, it is useful to note that linux provides some command line utilities for using IPC in shared memory in general. ipcs will list all of the IPC facilities in the system. This will include all types of IPC's message queues, semaphores. Passing the -m flag will display only the shared memory IPC. There is also a utility to remove an IPC construct. For shared memory, you use the m flag and you specify the shared memory identifier. Look at the man pages for both of these commands for a full set of options.

19 - Shared Memory Design Considerations



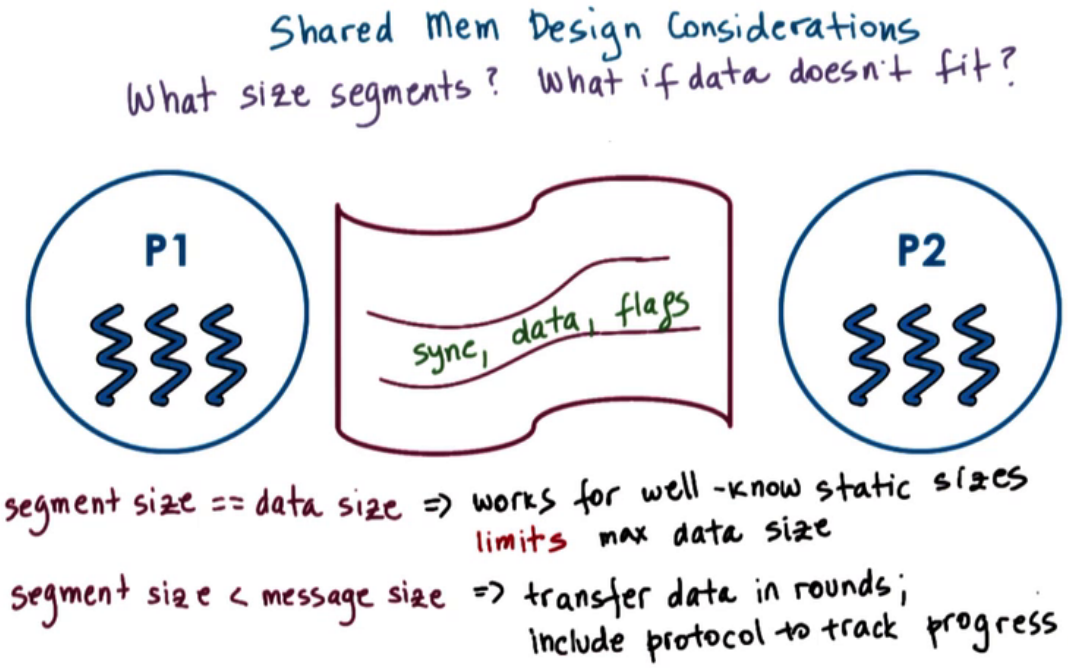
When using shared memory, the OS doesn't restrict you on how the memory will be used. However, the choice of the API or the specific mechanisms that will be used for synchronization are not the only decisions that you need to make. Remember with shared memory, the OS provides the shared memory area and then it's out of the way. All of the data passing and synchronization protocols are up to the programmer. So in the upcoming morsels, we will mention a few things that you can consider to assist with your design process.

20 - How Many Segments



To make things concrete, let's consider two multi threaded processes in which the threads need to communicate via shared memory. First consider how many segments will your processes need to communicate. Will they use one large segment? In that case, you will have to implement some type of management of this shared memory. You'll have to have some memory manager that will allocate and free this memory for the threads from the different processes. Or you can have multiple segments, smaller ones, one for each pairwise communication. If you choose to do this, it's probably a good idea to preallocate ahead of time a pool of segments so you don't have to slow down that way every individual communication with the segment creation overhead. So in that case, you will have to create how threads will pick up which of the available segments they will end up using for their interprocess communication. So using some type of queue of segment identifiers will be probably a good idea for that. fd from one process to another via some other type of communication mechanism like via message queue.

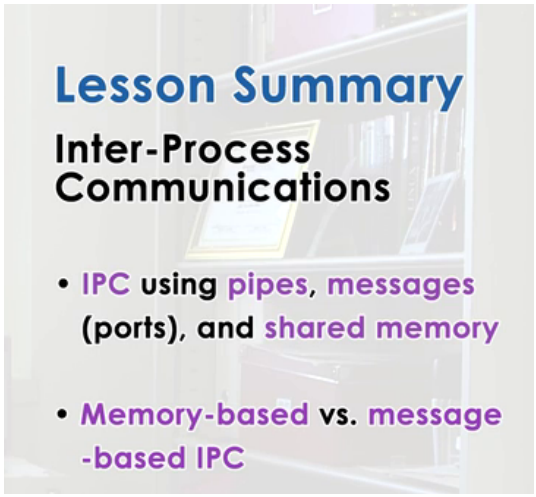
21 - Design Considerations



Another design question is how large should a segment be? That will work really well if the size of the data is known upfront and is static doesn't change. However in addition to the fact that data sizes may not be static, they may be dynamic, the other problem with this is that it will limit what is the maximum data that can be transferred between processes because typically an OS will have a limit on the maximum segment size. If you want to potentially support arbitrary segment sizes that are much larger than the segment size, then one option can be that you can transfer the data in rounds. Portion of the data gets written into the segment and then once P2 picks it up, P1 is ready to move in the next round of that data item. However in this case, the programmer will have to include some protocol to track the progress of the data movement through the rounds. In this case, you will likely end up casting the shared memory area as

some data structure that has the actual data buffer, some synchronization construct, as well as some additional flags to track the progress.

22 - Lesson Summary



In this lesson we talked about interprocess communication or IPC. We described several IPC mechanisms that are common in operating systems today. We spent a little more time on use of shared memory as an IPC mechanism and we contrasted this also with use of message based IPC mechanisms. Based on this lesson, you should have enough information on how to start using IPC mechanisms in your project.