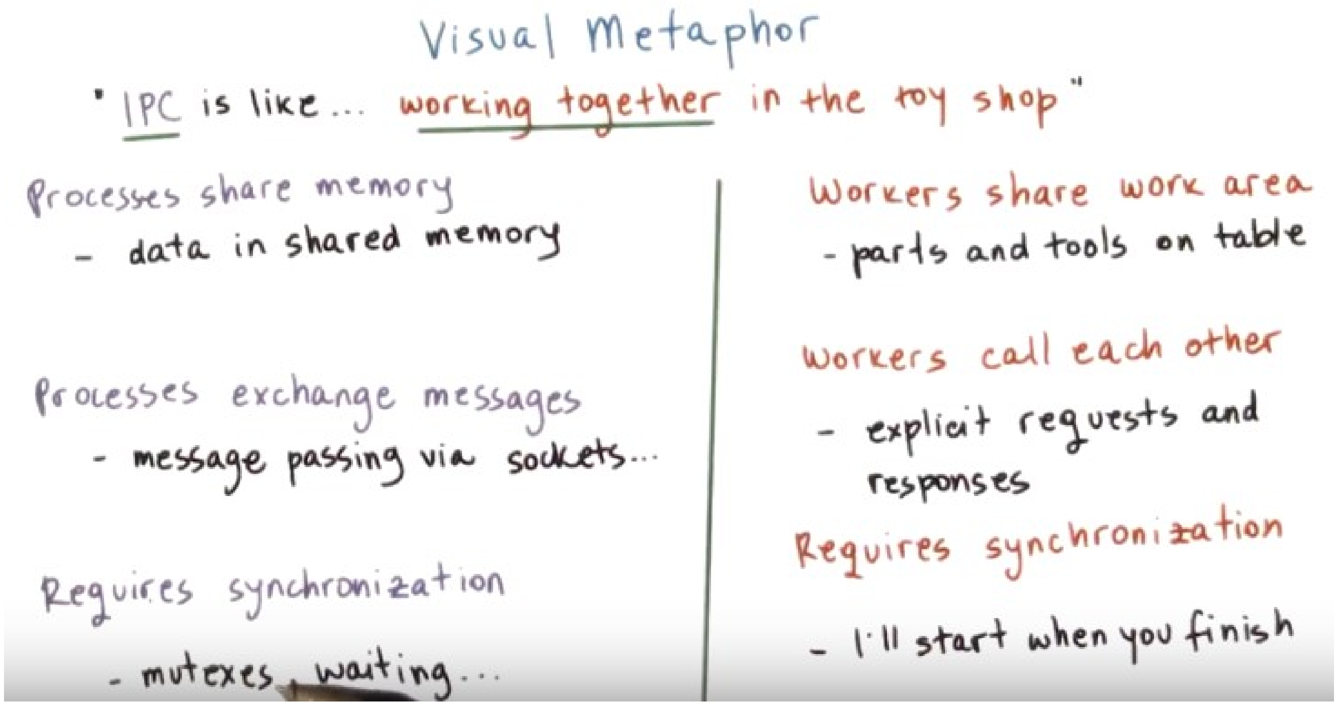
Lesson Preview

Inter-Process Communications

- IPC
- Shared Memory IPC

1. In this lesson we will talk about inter-process communications, or IPC. We will primarily talk about shared memory and describe some of the APIs for shared memory based IPC. In addition we will describe some of the other IPC mechanisms that are common in operating systems today.



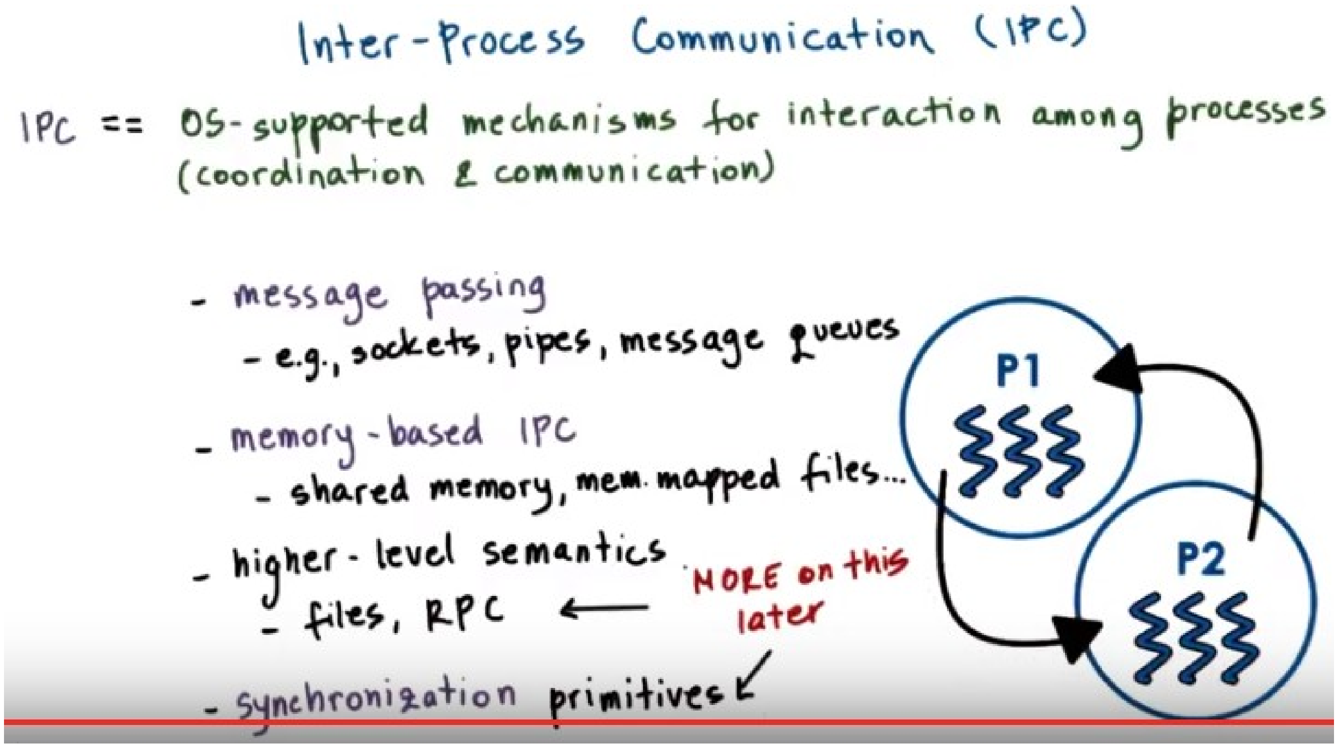
Visual Metaphor

" IPC is like ... working together in the toy shop "

Processes share memory - data in shared memory	Workers share work area - parts and tools on table
Processes exchange messages - message passing via sockets...	workers call each other - explicit requests and responses
Requires synchronization - mutexes, waiting...	Requires synchronization - I'll start when you finish

2. In an earlier lesson, we described the process like an order of toys 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 inter-process communication is achieved in a toy shop illustration. IPC is like working together in the 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 a 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 APIs like like sockets. For both of these methods, processes may need to wait on one another and may need to rely on some synchronization mechanism like mutexes to make sure that the communication proceeds in a correct manner.



Inter-Process Communication (IPC)

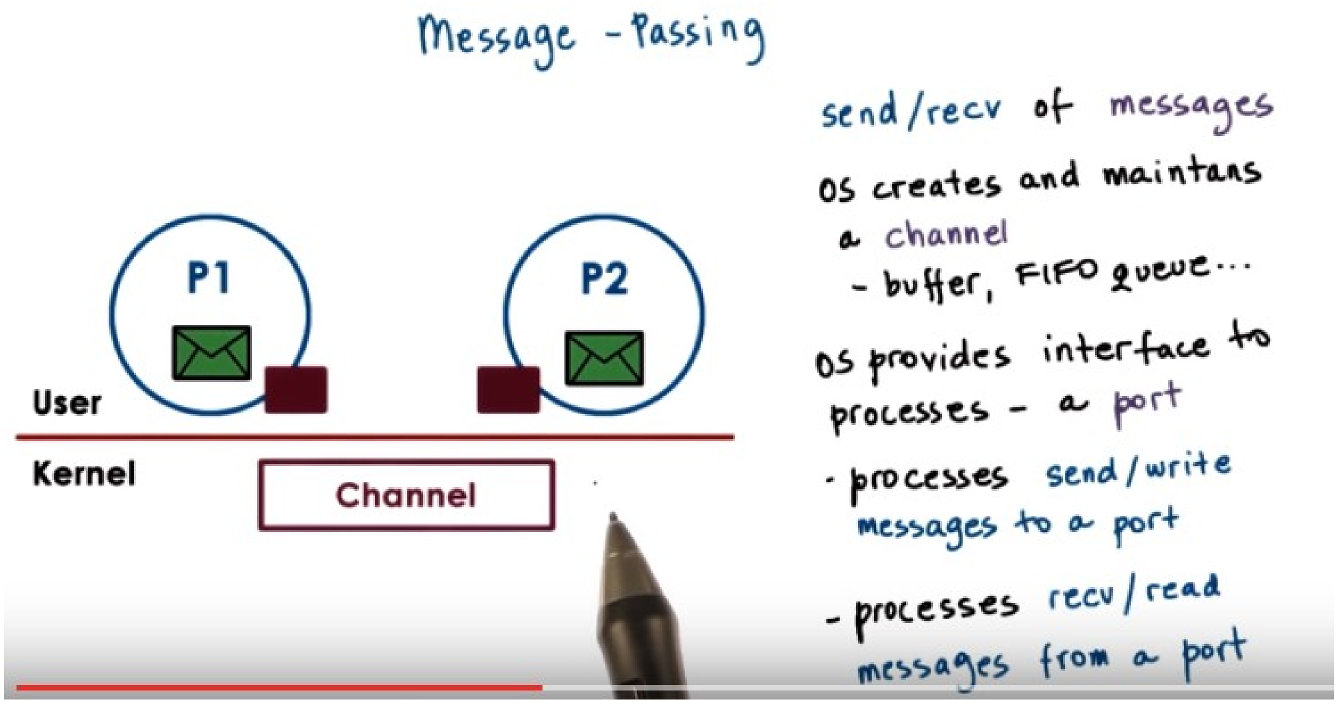
IPC == OS-supported mechanisms for interaction among processes (coordination & communication)

- message passing
  - e.g., sockets, pipes, message queues
- memory-based IPC
  - shared memory, mem. mapped files...
- higher-level semantics
  - files, RPC ← MORE on this later
- synchronization primitives

RPC: remote procedure calls (P4L1 題目就是這個)

3. Inter-process communication 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 include sockets, that most of you 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 set of pages of physical memory or also may be in the form of memory mapped files. Speaking of files, these two could 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 procedure calls, or RPC. Here, by higher-level semantics, we mean that it's a mechanism that supports more than simply a channel for two 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, et cetera. 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 two bullets, and we will talk about these remaining topics later.

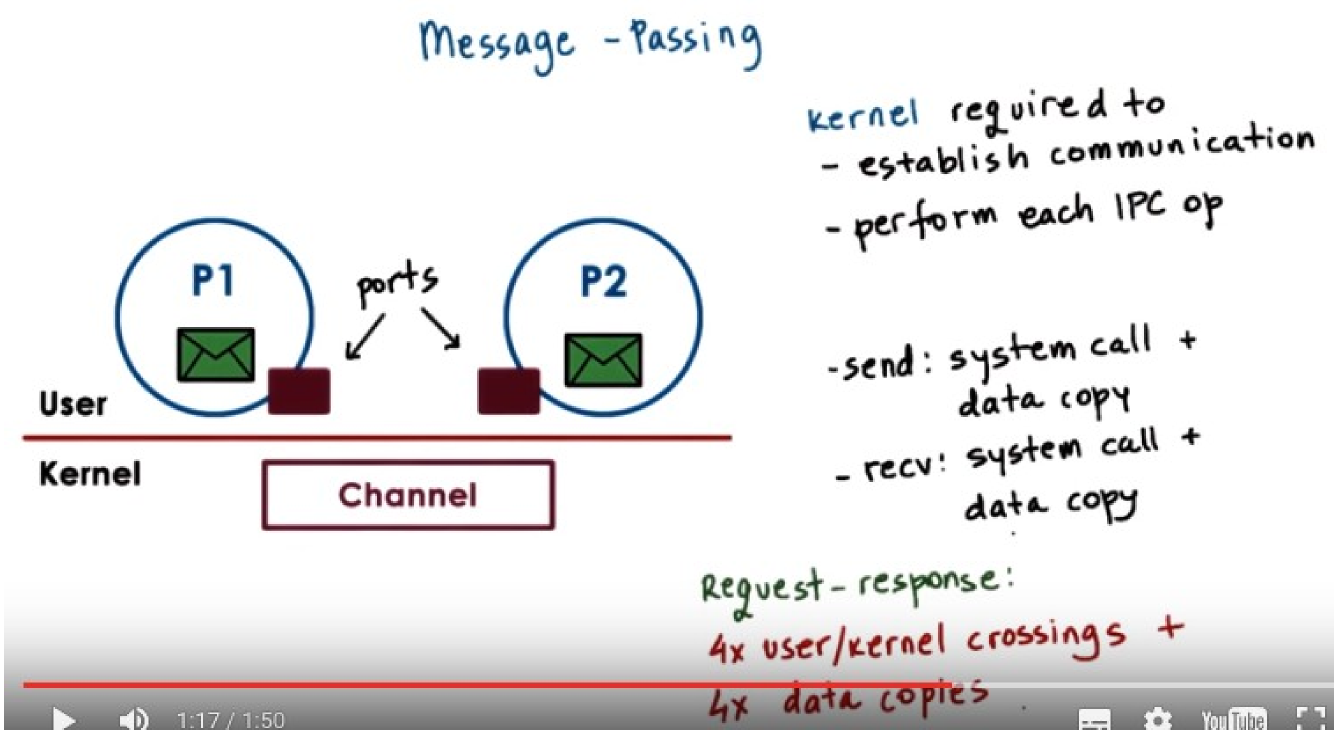

Message - Passing

send/recv of messages

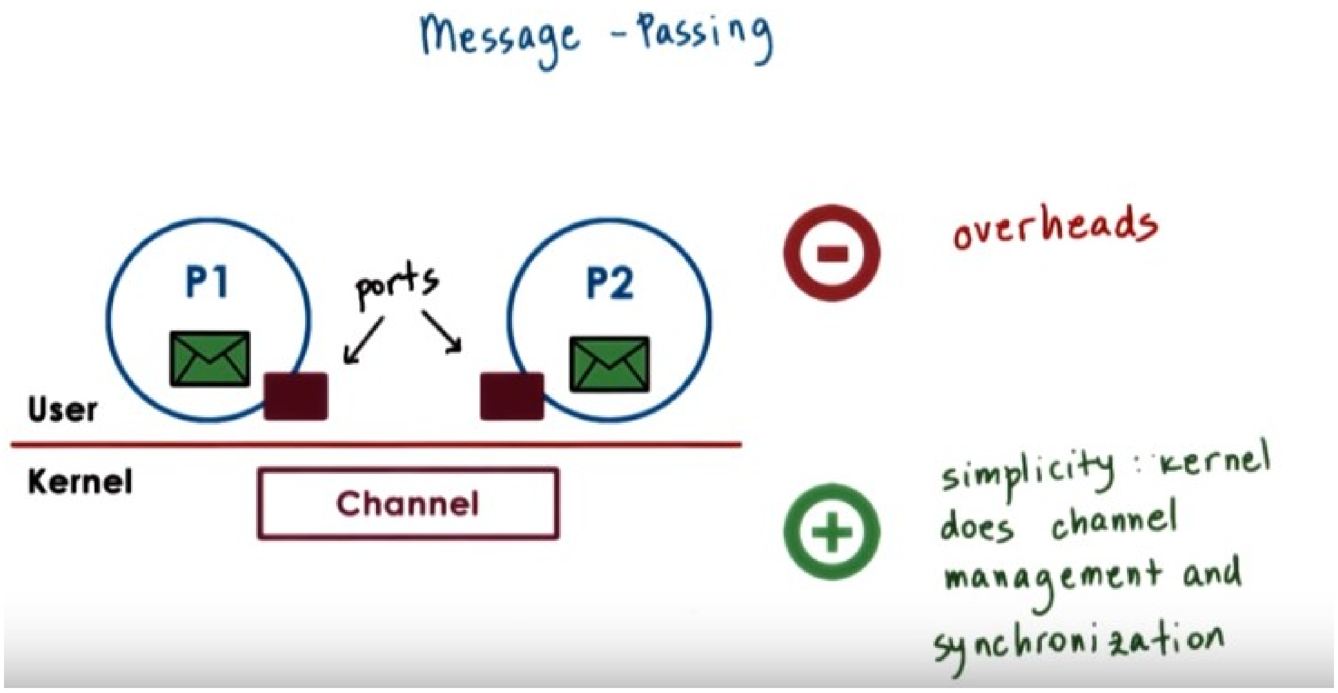
OS creates and maintans a channel
- buffer, FIFO queue...

OS provides interface to processes - a port
- processes send/write messages to a port
- processes recv/read messages from a port

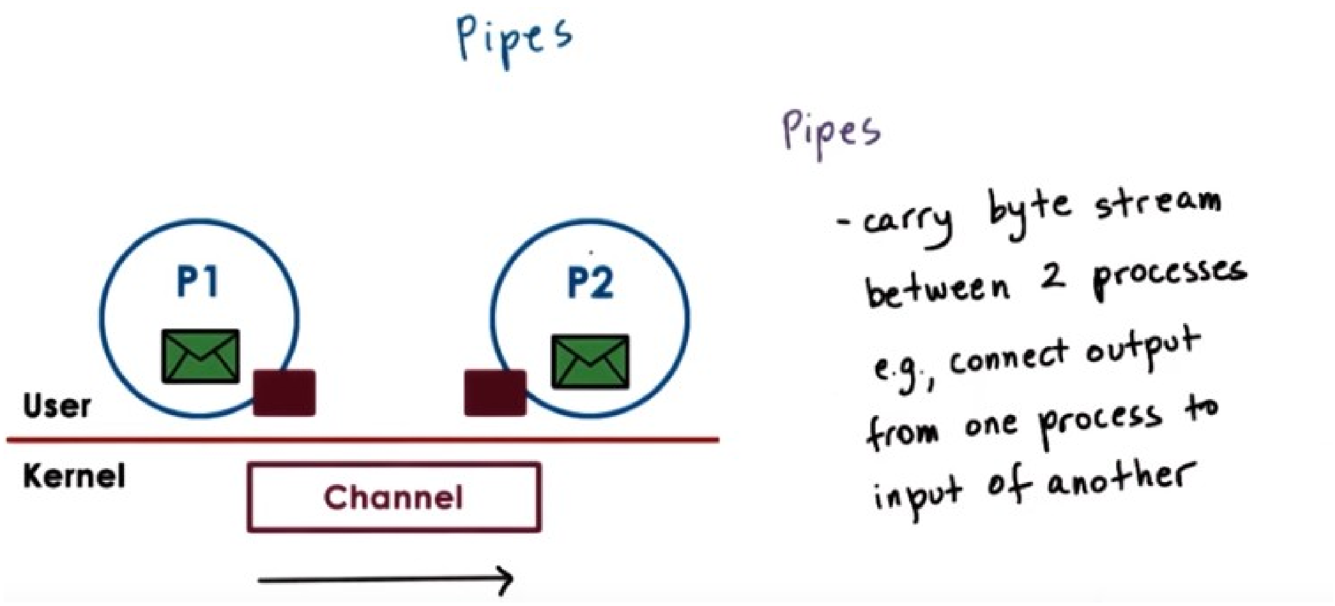
4. One mode of IPC that operating system support is called message passing. As the name implies, processes create messages and then send or receive them. The operating system is responsible for creating and maintaining the channel that will be used to pass messages among processes. This can be thought of as some sort of buffer or FIFO queue. 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.



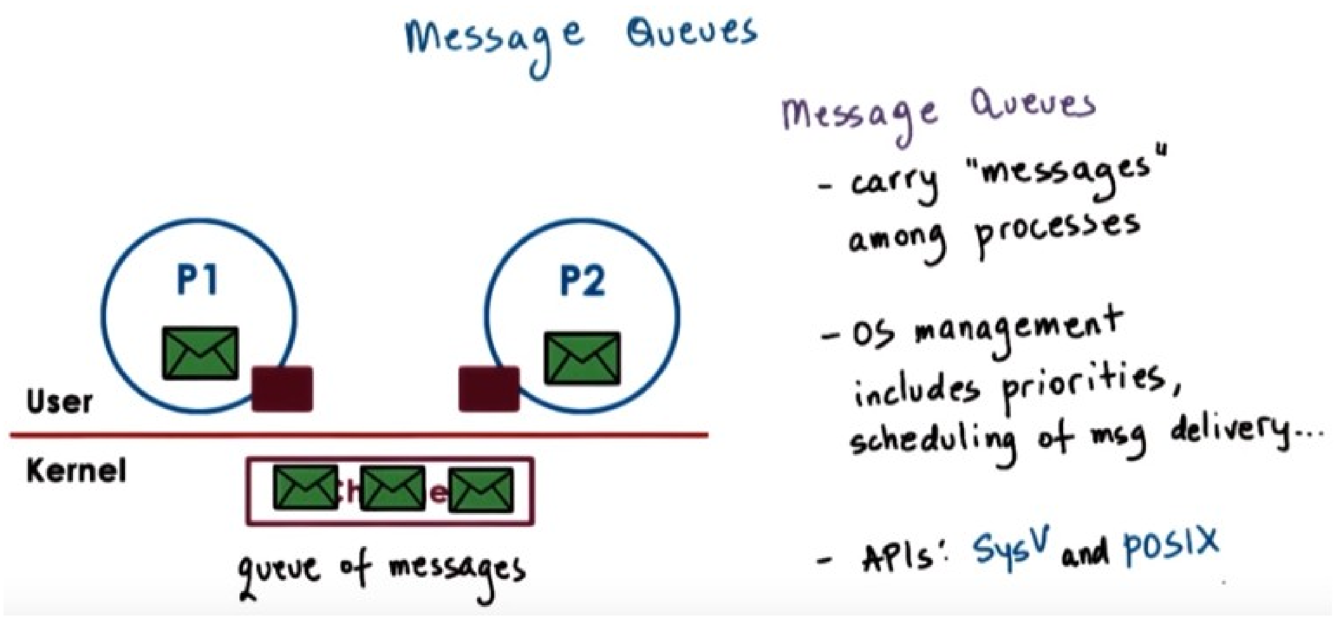
The OS Kernel is required to both establish the communication channel, as well as to perform every single IPC operation. What that means is that both the send and receive operation require a system call, and a data copy as well. In the case of send, from the process address base into the communication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


Message - Passing
P1
ports
P2
User
Kernel
Channel
overheads
simplicity : kernel does channel management and synchronization

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 it's 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 data is not overwritten or corrupt in some way, as processes are trying to send or receive it, potentially at the same time. So that's a plus.

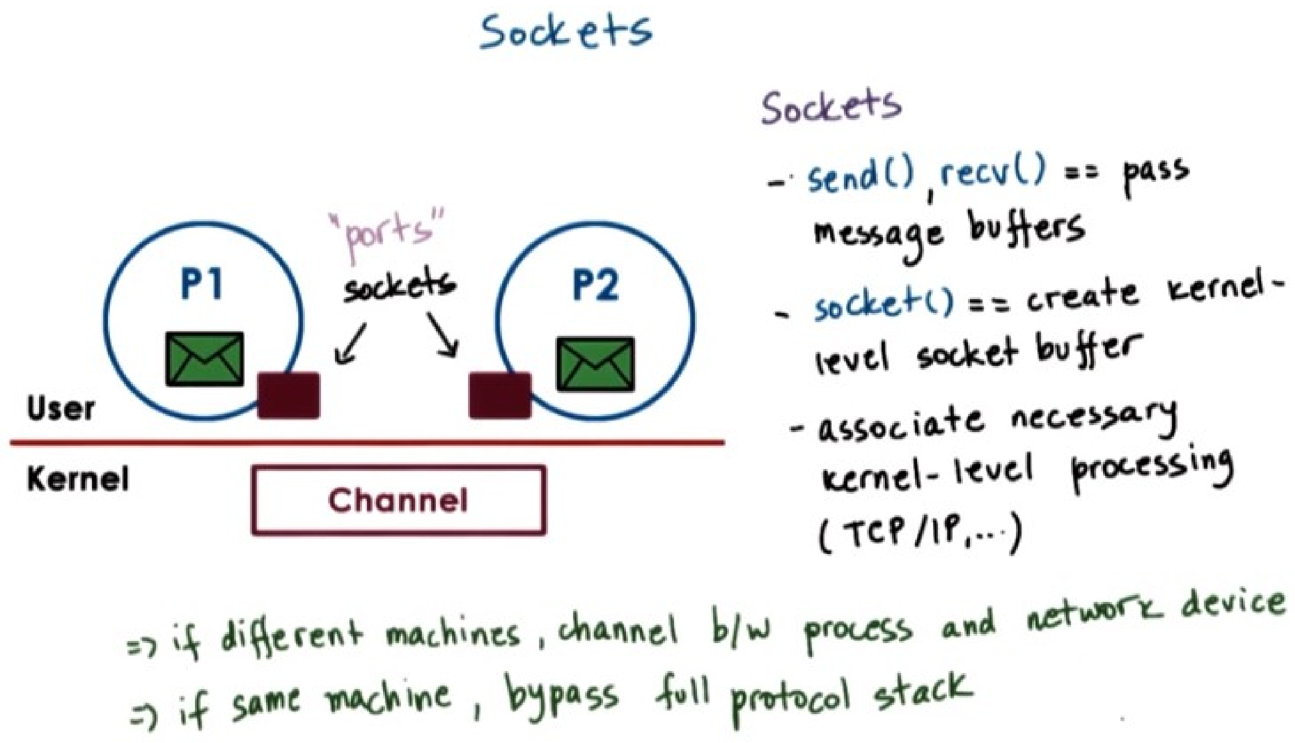

Pipes
Pipes
- carry byte stream between 2 processes
e.g., connect output from one process to input of another
P1
P2
User
Kernel
Channel

5. In practice, there are 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 end points, 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 pushed into the pipe from one process and then received, but into another. And 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 would be delivered as input to P2 instead of somebody typing it in, for instance.


Message Queues
Message Queues
- carry "messages" among processes
- OS management includes priorities, scheduling of msg delivery...
- APIs: SysV and POSIX
P1
P2
User
Kernel
queue of messages

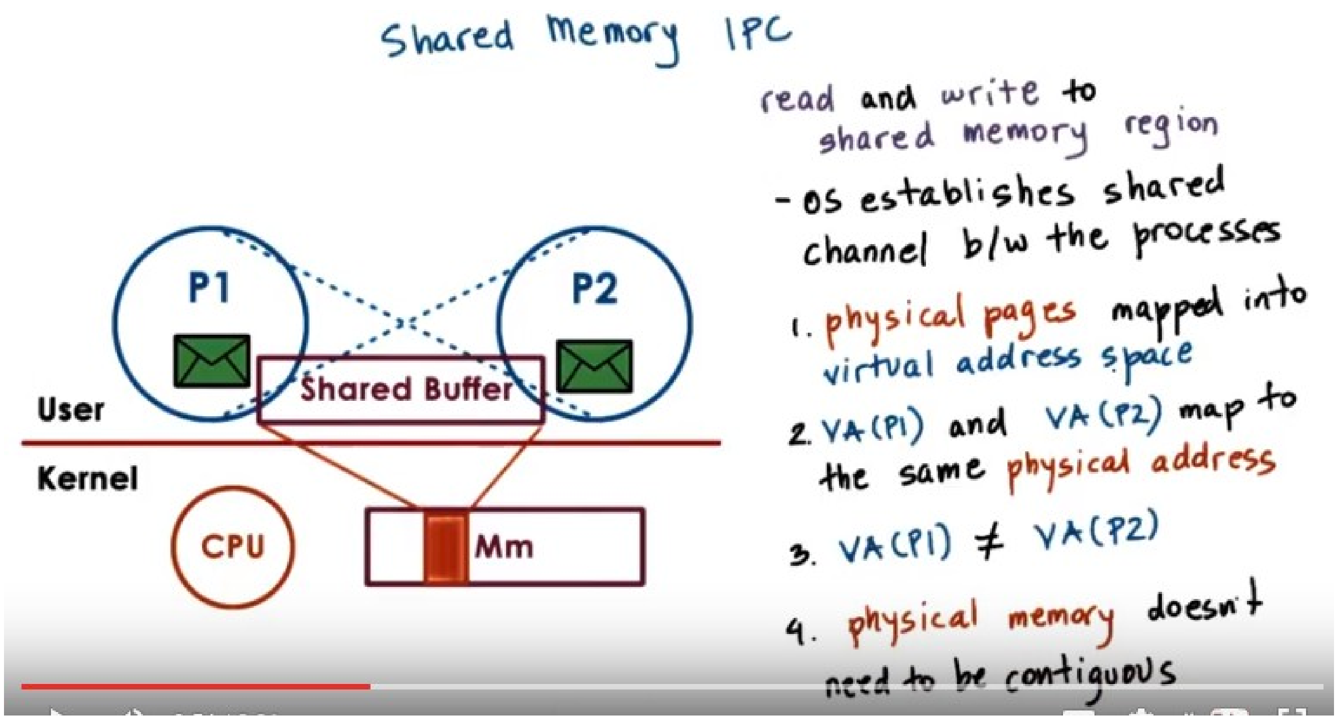
For message queue, there is also a good page (same author as the shared memory example codes): http://beej.us/guide/bgipc/output/html/multipage/mq.html

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 APIs. In Unix-based systems, these include the POSIX API and the System V API.

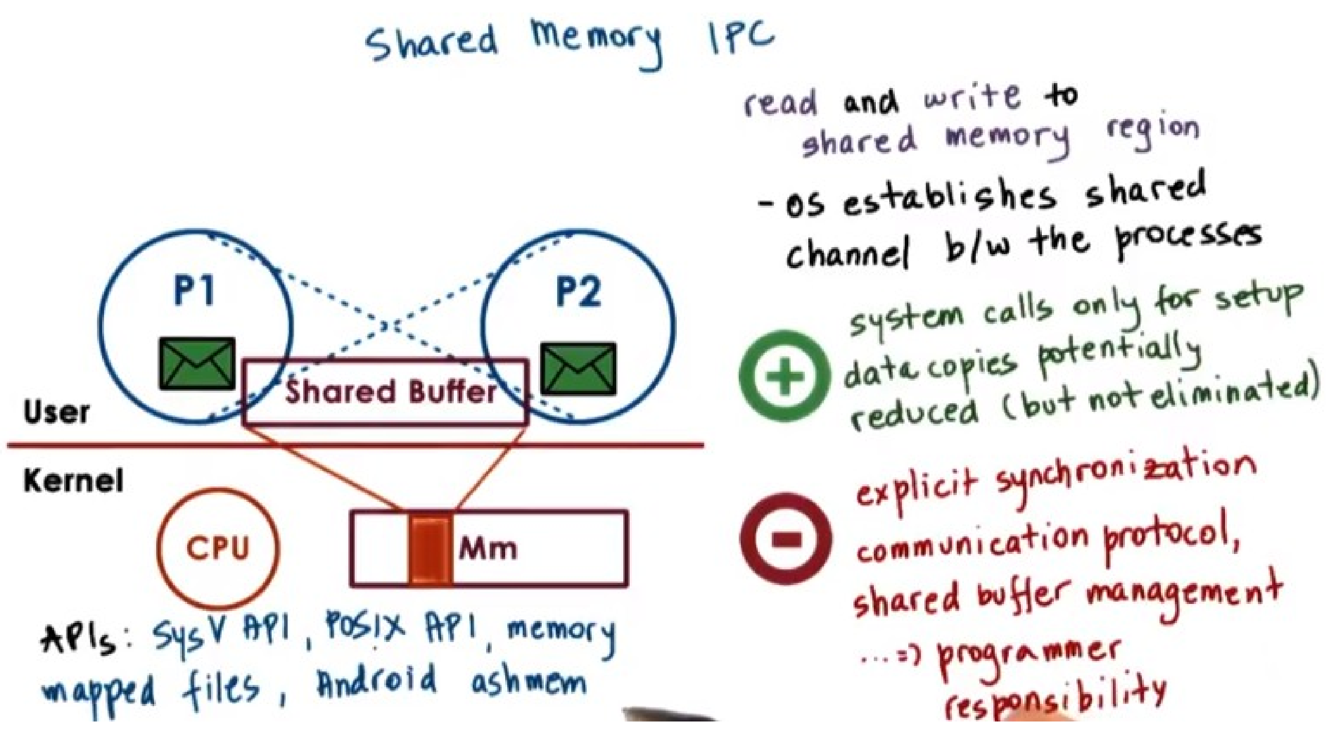

Sockets
Sockets
- send(), recv() == pass message buffers
- socket() == create kernel-level socket buffer
- associate necessary kernel-level processing (TCP/IP,...)
"ports"
sockets
P1
P2
User
Kernel
Channel
=> if different machines, channel b/w process and network device
=> if same machine, bypass full protocol stack

b/w: between

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 operating system. Which sockets processes send messages or receive messages via an API that looks like this, and then receive. The socket API supports send and receive operations that allow processes to send messages 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 a network device that will actually send the data. In addition, the operating system is sufficiently smart to figure out that if two 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.

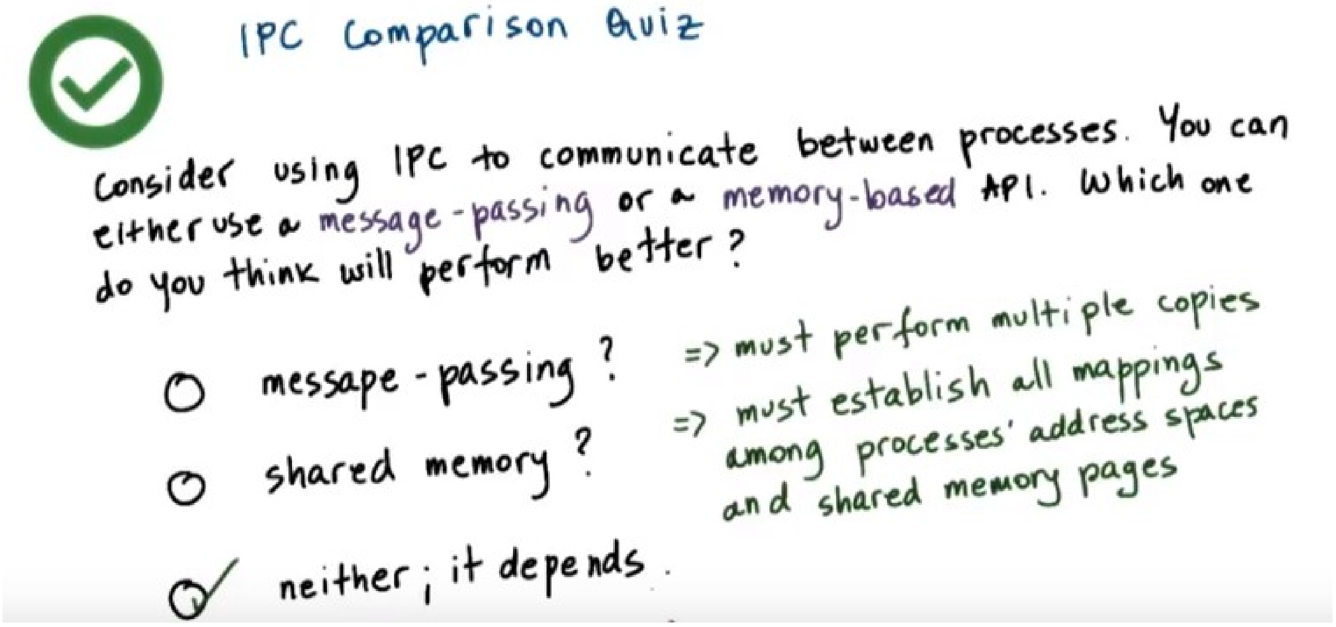

Shared Memory IPC
read and write to shared memory region
- OS establishes shared channel b/w the processes
1. physical pages mapped into virtual address space
2. VA(P1) and VA(P2) map to the same physical address
3. VA(P1) ≠ VA(P2)
4. physical memory doesn't need to be contiguous
P1
P2
Shared Buffer
User
Kernel
CPU
Mm

6. 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 in our modern hardware.


Shared Memory IPC
read and write to shared memory region
- OS establishes shared channel b/w the processes
P1
P2
Shared Buffer
User
Kernel
CPU
Mm
system calls only for setup
data copies potentially reduced (but not eliminated)
explicit synchronization
communication protocol, shared buffer management
...⇒ programmer responsibility
APIs: SysV API, POSIX API, memory mapped files, Android ashmem

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 (即圖中那個 Shared Buffer 框). So if that's not the case, then data within the same address space 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 two 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 are 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 is the developer's responsibility to determine any communication protocol related issues such as, how are messages going to be formatted? How will they be delimited? What are their headers going to look like? And also, how this shared memory buffer will be allocated? When, which process, will be able to use a portion of this buffer for its needs. So this adds some complexity, obviously. Unix based systems, including Linux, support two popular shared memory APIs. One of these was originally developed as part of System 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 the memory wrapped files in both address spaces. This API's 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 system files POSIX APIs, 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 space shared memory APIs

7. We saw two major ways to implement IPC using a message-passing or a memory-based API. Which one of the two do you think will perform better? The message-passing? The shared memory-based API? Or neither, it depends? Mark your answer from the following choices.

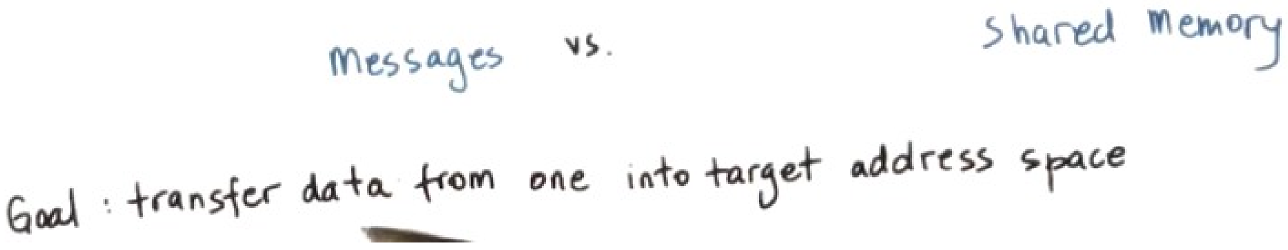


IPC Comparison Quiz

Consider using IPC to communicate between processes. You can either use a message-passing or a memory-based API. Which one do you think will perform better?

- message-passing? => must perform multiple copies
- shared memory? => must establish all mappings among processes' address spaces and shared memory pages
- [✓ neither; it depends.]

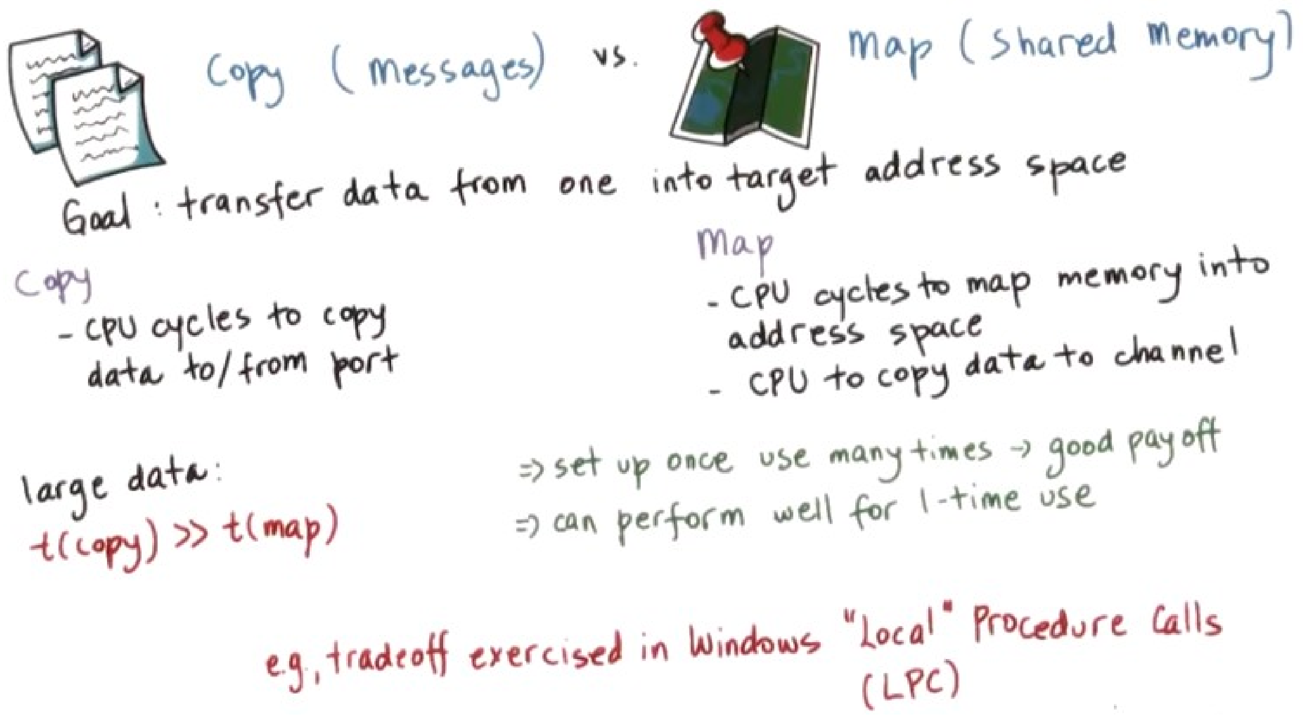
8. The answer to this question is the it depends answer that's common in many systems questions. Here is why. We mentioned that in message passing multiple copies of the data must be made between the processes that communicate and the kernel. That leads to overhead, clearly. For shared memory IPC, there are a 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 the both sides. And the correct answer will be, it depends. In the next video, we will explain the trade-offs that exists among these two types of IPC mechanisms.



Messages vs. Shared Memory

Goal: transfer data from one into target address space

9. Before I continue I want to make one important comment to contrast the message-based and the shared memory-based approaches to IPC. The end result of both of these approaches is that some data is transferred from one address space into the target address space.



Copy (messages) vs. Map (shared memory)

Goal : transfer data from one into target address space

Copy
- CPU cycles to copy data to/from port

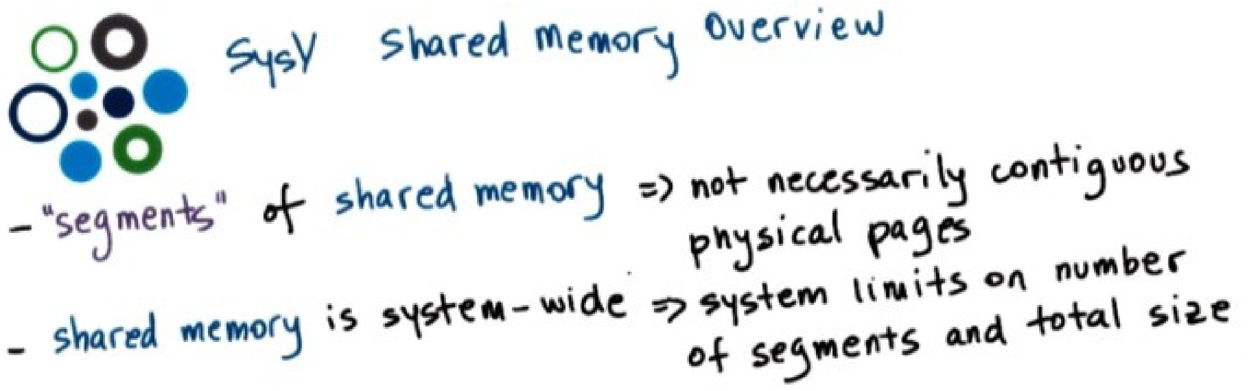
Map
- CPU cycles to map memory into address space
- CPU to copy data to channel

large data:
t(copy) >> t(map)

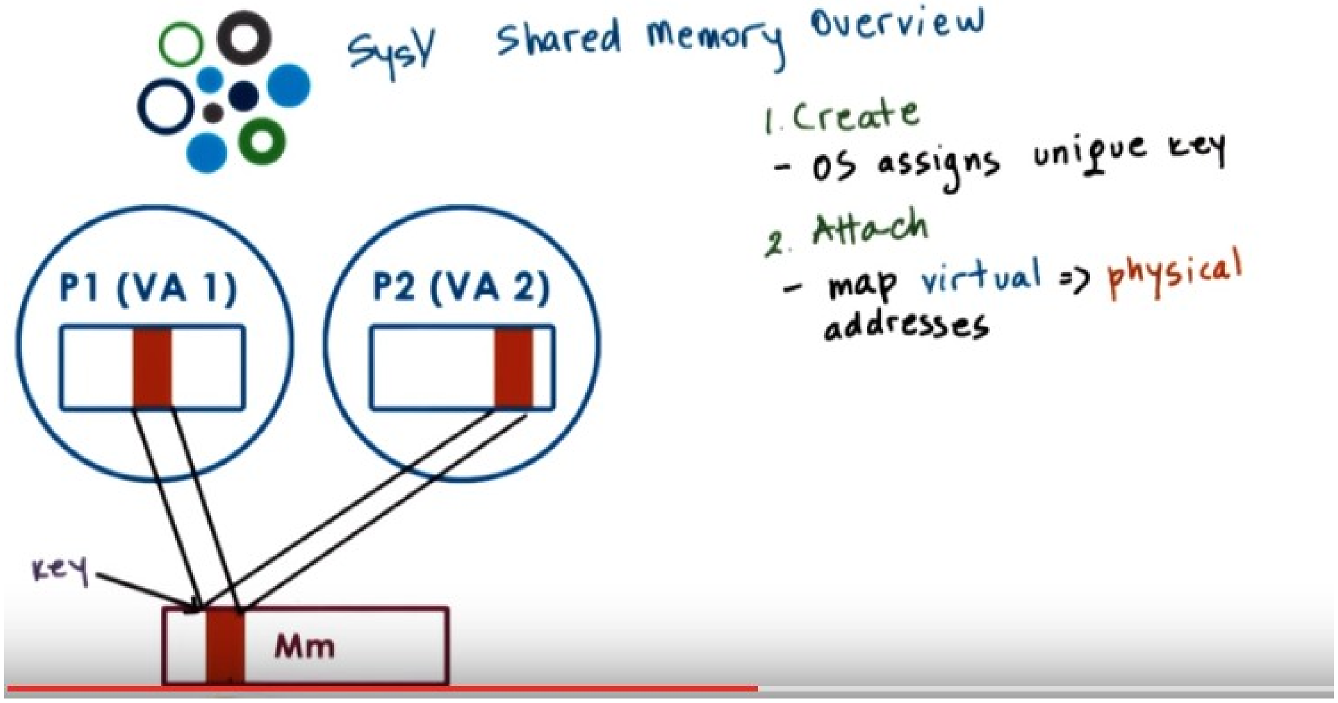
=> set up once use many times -> good payoff
=> can perform well for 1-time use

e.g., tradeoff exercised in Windows "Local" Procedure Calls (LPC)

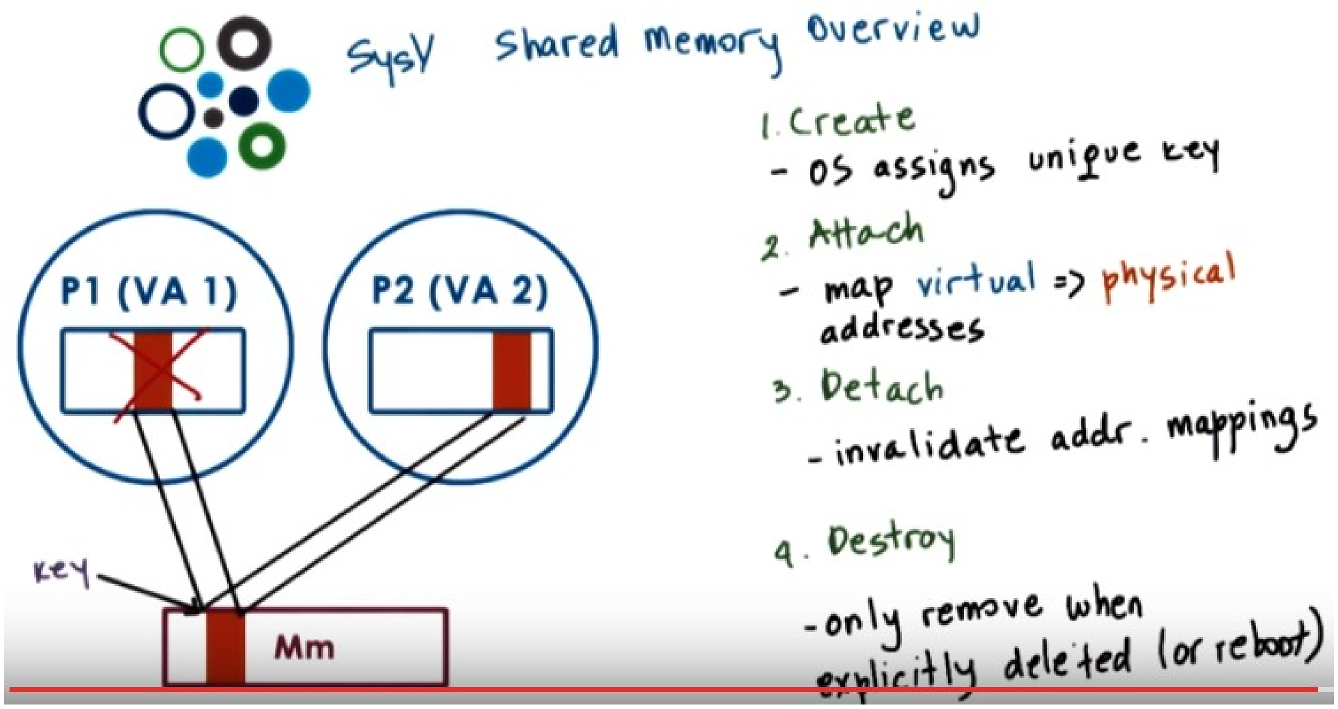
In message passing, this requires that the CPU is involved in copying the data. This takes some number of CPU cycles to copy the data into the channel via the port and then from the port and into the target address space. In the shared memory-based case, at the minimum, there's CPU cycles that are spent 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 large amounts of data from one 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, Windows systems internally in the communication mechanisms they support between processes, leverage the fact that there exists this difference. So if the 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 a port like interface. Otherwise, the data is potentially copied once to make sure that it's in a page aligned area. And then that areas is mapped into the address space of the target process. This mechanism that the Windows kernel supports is called Local Procedure Calls, or LPC.


SysV Shared Memory Overview
- "segments" of shared memory => not necessarily contiguous physical pages
- shared memory is system-wide => system limits on number of segments and total size

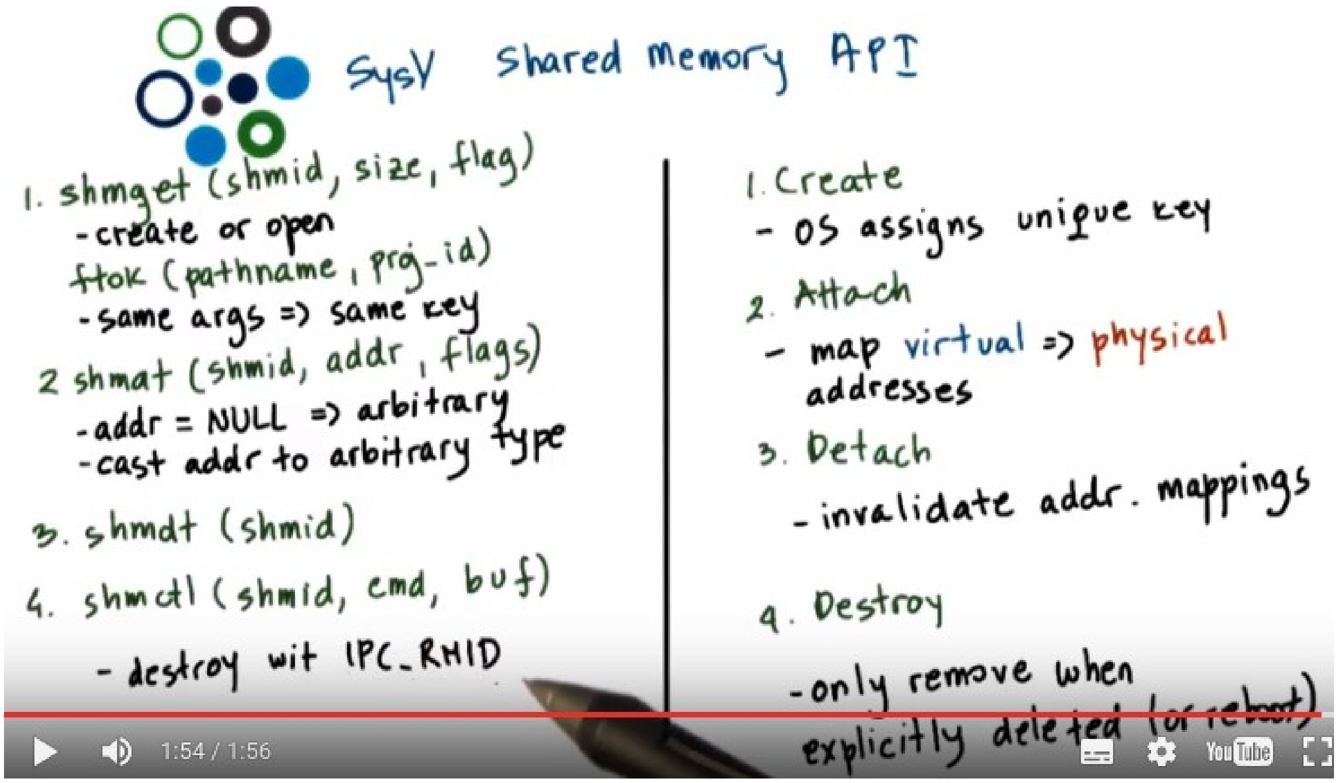
10. Now that we've described the shared memory mechanisms in a general way, let's look at the specific details of the system five (SysV 中的 V 是羅馬數字 5) Unix API. First the operating system supports segments of shared memory, that don't necessarily have to correspond to contiguous physical pages. Also, the operating system treats shared memory as a system-wide resource using system-wide policies. That means that there is a limit on the total number of segments of 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 4,000 segments. However, in the past it used to be much less and in certain OSs, it was as few as six segments. More recent versions of Linux had a limit of 128 segments. The operating system 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 operating system 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 operating system. 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 either 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. 此句說得好: This means that the operating system 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 physical 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 address in different processes.


SysV Shared Memory Overview
1. Create
- OS assigns unique key
2. Attach
- map virtual => physical addresses
3. Detach
- invalidate addr. mappings
4. Destroy
- only remove when explicitly deleted (or reboot)
P1 (VA 1)
P2 (VA 2)
key
Mm

Detaching a segment means invalidating the address mappings for the virtual address 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 maybe attached and detached then reattached multiple times by different processes during it's life time. What this means is that once a segment is created it's like a persistent entity until there is an explicit request for it to be destroyed. This is similar to what would happen to a file. We create a file and then the file persists until it is explicitly deleted. In the mean time, we can open it and close it and read it and write 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 Malloced and then it will disappear as soon as the process exits.


SysV Shared Memory API

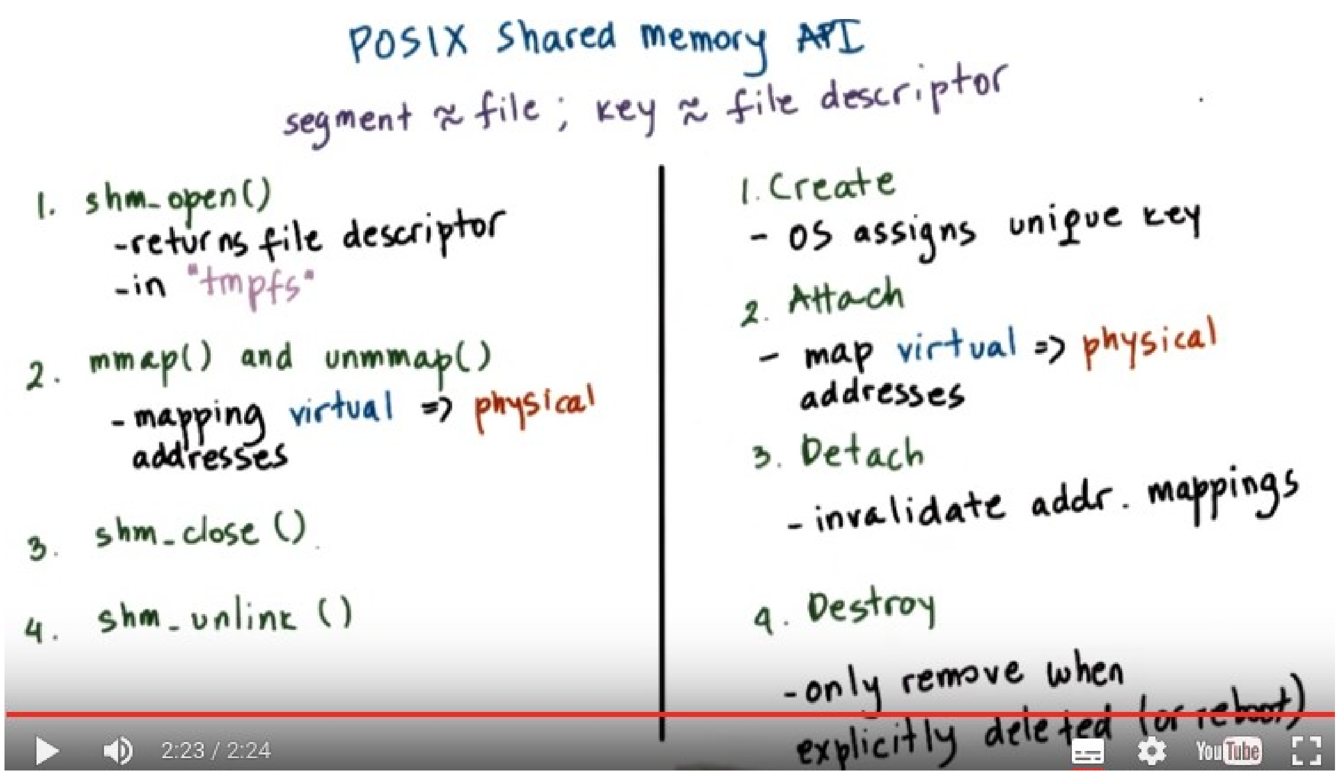
1. shmget (shmid, size, flag)
   - create or open
   ftok (pathname, prg-id)
   - same args => same key
2. shmat (shmid, addr, flags)
   - addr = NULL => arbitrary
   - cast addr to arbitrary type
3. shmdt (shmid)
4. shmctl (shmid, cmd, buf)
   - destroy wit IPC_RMID

1. Create
   - OS assigns unique key
2. Attach
   - map virtual => physical addresses
3. Detach
   - invalidate addr. mappings
4. Destroy
   - only remove when explicitly deleted (or reboot)

shm 即 shared memory. 上圖左半邊跟右半邊是對應的.

11. SysV 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. And the flags include the areas options like permissions. This unique identifier is the key and that is not actually magically created by the operating system. Instead it is explicitly passed to the OS by the application (即 schmid 是 application 提供的). 由第 14 段的代知, shmget 的返回值即 shmat 中的 shmid(應該也是 shmget 輸入的那個 shmid).
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 (由第 14 段的代知, ftok 的返回值即 shmget 中的 shmid). That'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 (shmat) attaches the shared memory segments into the virtual address space of the process. So we'll map them into the user address space. The programmer has an option to provide the specific virtual addresses where the segment should be mapped, or if NULL is passed then the operating system will choose and return some arbitrary suitable addresses that are available in the processes 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 (見第 14 段中的代碼).
The following operation (shmdt) detaches the segment identified by this identifier, so the virtual to physical memory mappings are no longer valid.
And then finally (schmctl) the control operation that the shared memory API supports is used to pass

certain commands related to the shared memory segment management to the operating system. Including the command to remove a particular segment. And that command is IPC_RMID.



POSIX Shared Memory API

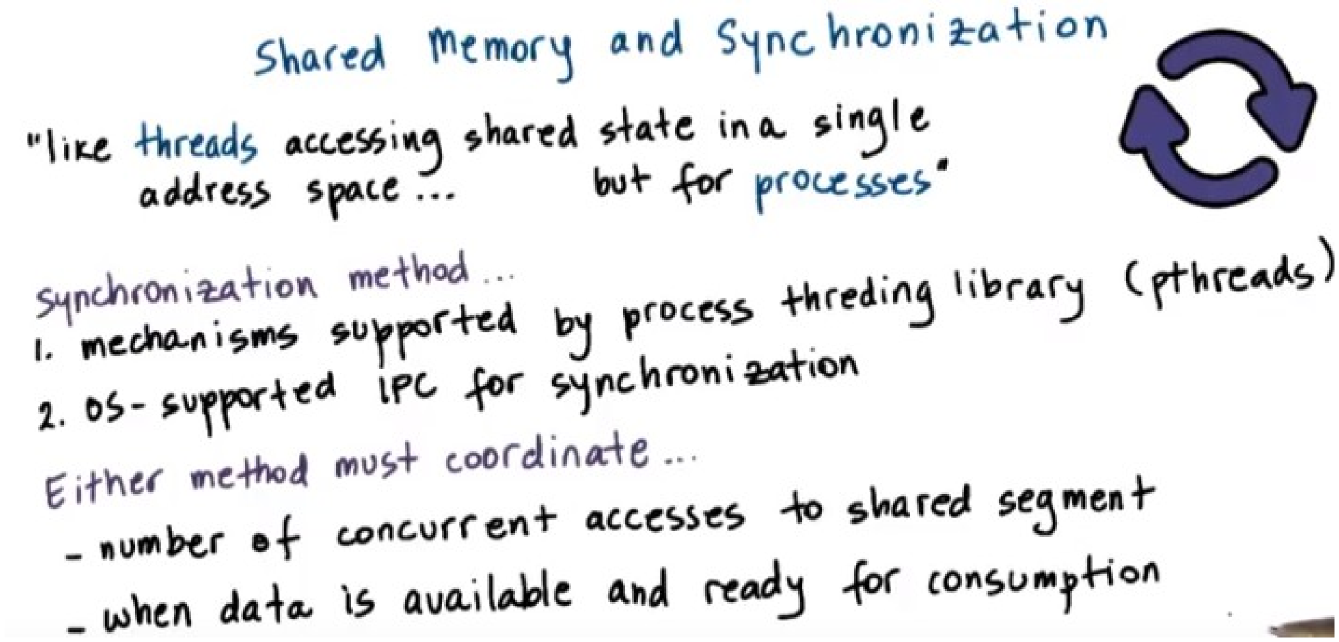
segment ≈ file ; key ≈ file descriptor

1. shm_open()
   - returns file descriptor
   - in "tmpfs"
2. mmap() and unmmap()
   - mapping virtual => physical addresses
3. shm_close()
4. shm_unlink()

1. Create
   - OS assigns unique key
2. Attach
   - map virtual => physical addresses
3. Detach
   - invalidate addr. mappings
4. Destroy
   - only remove when explicitly deleted (or reboot)

12. There is also the POSIX API for shared memory. On Linux systems, 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 SysV 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 the real files that exist in some file system that used otherwise by the operating system. Instead, these are files that only exist in the so called tmpfs file system, which is really intended to look and feel like a file system. So, you can always reuse the same type of mechanisms that is used for file systems. But, in essence, is just a bunch of state that's present in physical and volatile memory. The I/O simply uses the same representation and the same data structures that used for representing a file to represent bunch of pages in memory 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're explicit, shared memory, open and close operations. But, in fact, it can really only call the regular open and close operations, since you will anyways pass a file. And, the operating system 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 unmap 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

operating system to delete all of the shared memory-related data structures, and to free up that memory segment, you must call the explicit unlink operation. I have provided a link to the reference of the POSIX Shared Memory API in the instructor notes.



Shared Memory and Synchronization

"like threads accessing shared state in a single address space ... but for processes"

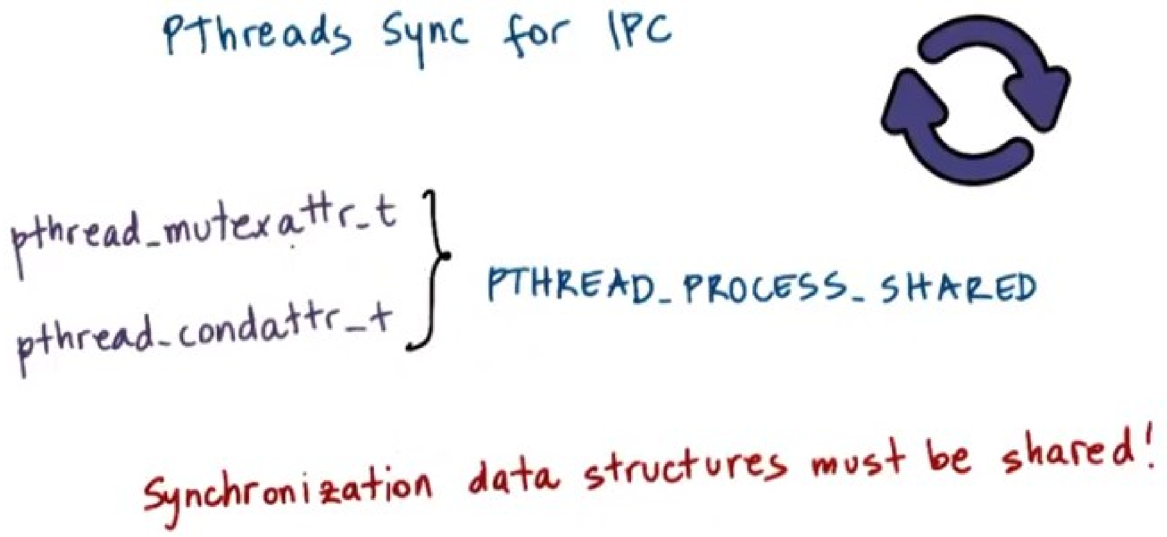
synchronization method ...

1. mechanisms supported by process threding library (pthreads)
2. OS-supported IPC for synchronization

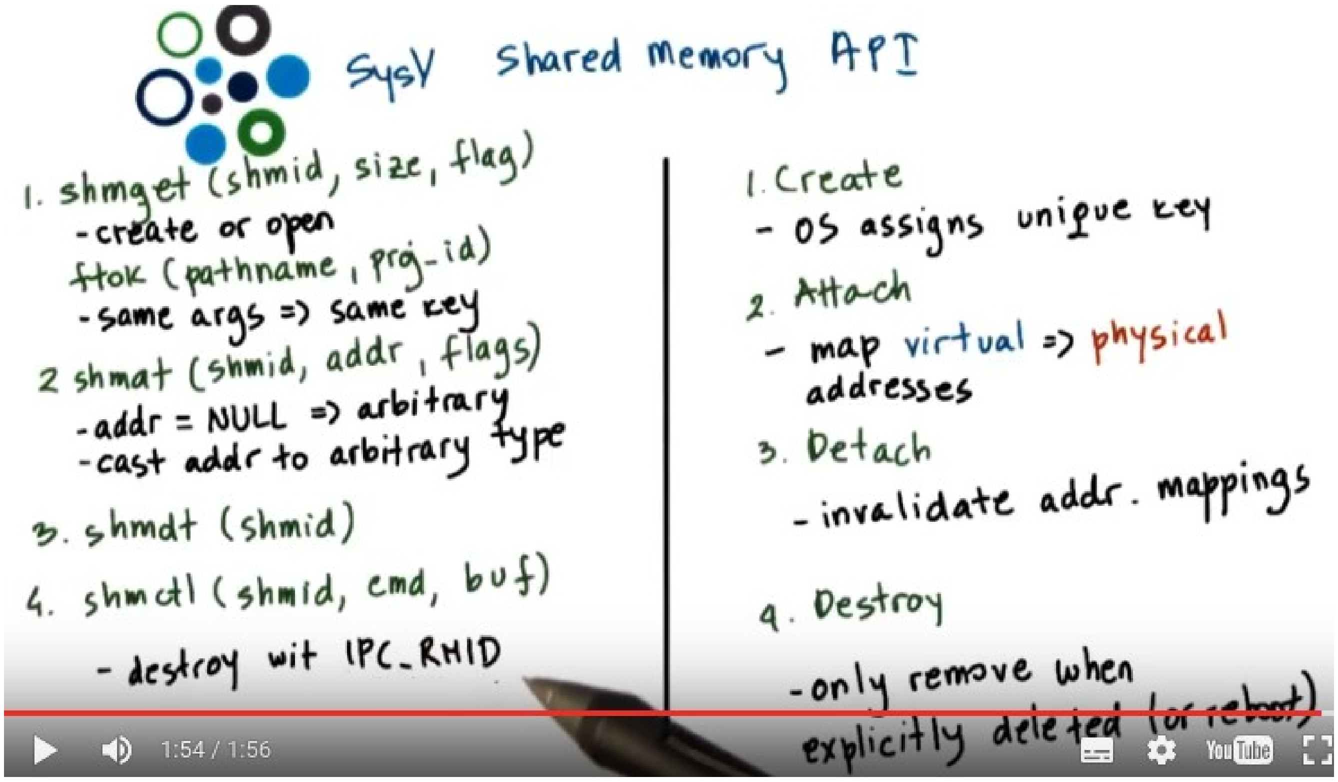
Either method must coordinate ...

- number of concurrent accesses to shared segment
- when data is available and ready for consumption

13. 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 certain synchronization constructs, such as mute accessor 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 mutex and condition variables that have been appropriately set. In addition, the operating system 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 must, 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. 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 your process or shared among processes. The keyword for this is PTHREAD_PROCESS_SHARED. So when synchronizing the shared memory accesses of two pthreads multithreaded processes, we can use mutexes and condition variables that have been correctly initialized with PTHREAD_PROCESS_SHARED. One important thing, however, is that the synchronization variables (應該就是指的 mutex 和 condition variable) themselves have to be also shared. Remember, in multithreaded 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 construct are allocated from the shared memory region that's visible to both processes.

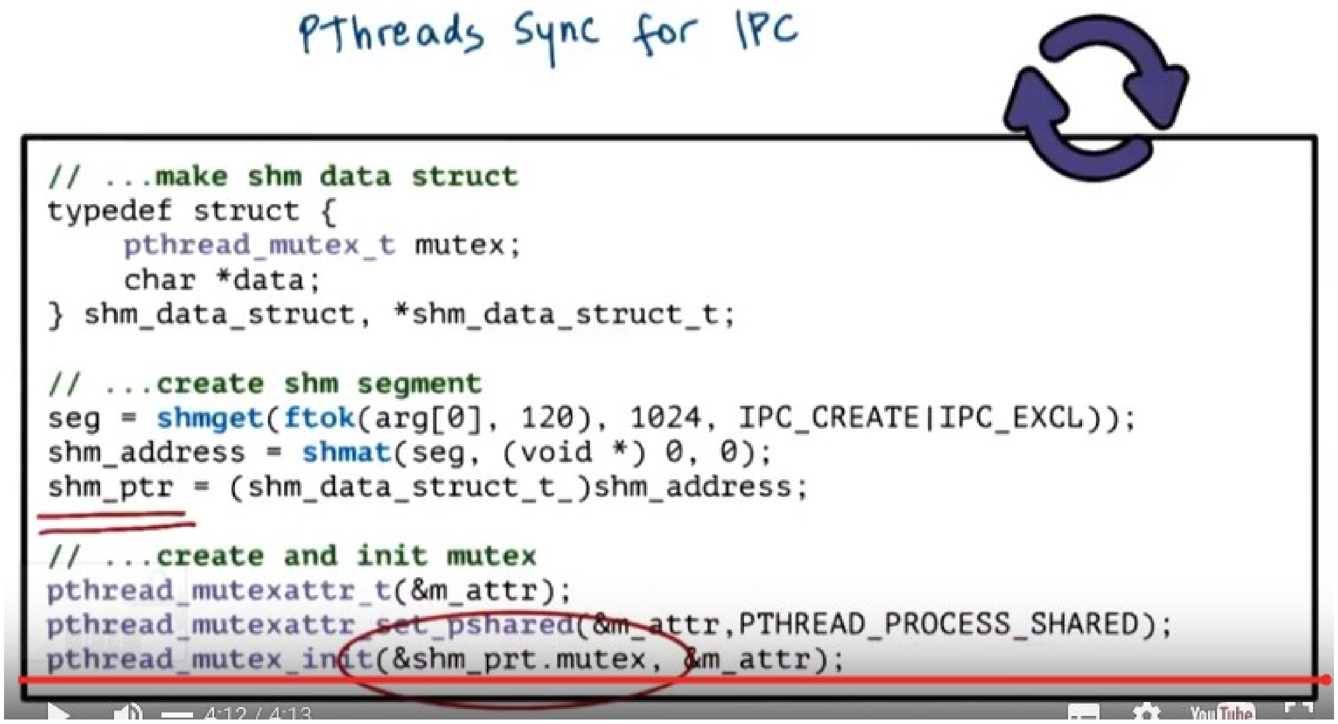


SysV Shared Memory API

1. shmget (shmid, size, flag)
   - create or open
   ftok (pathname, prj-id)
   - same args => same key
2 shmat (shmid, addr, flags)
   - addr = NULL => arbitrary
   - cast addr to arbitrary type
3. shmdt (shmid)
4. shmctl (shmid, cmd, buf)
   - destroy wit IPC_RMID

1. Create
   - OS assigns unique key
2. Attach
   - map virtual => physical addresses
3. Detach
   - invalidate addr. mappings
4. Destroy
   - only remove when explicitly deleted (or reboot)

上圖是第 11 段的圖, 將它拷過來方便看



PThreads Sync for IPC

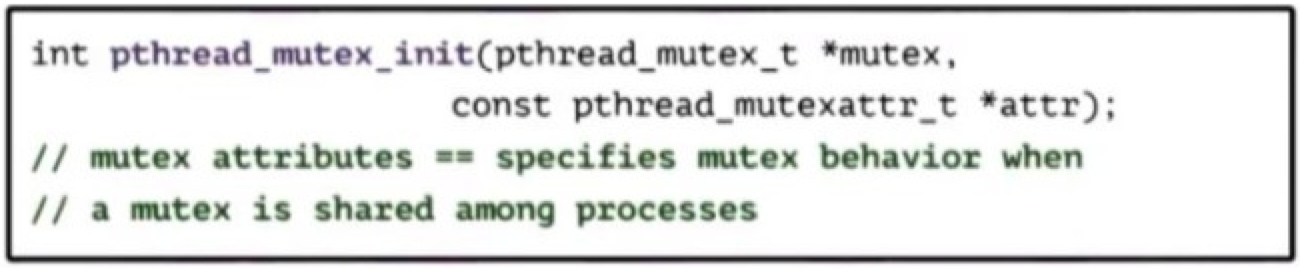
```
// ...make shm data struct
typedef struct {
     pthread_mutex_t mutex;
     char *data;
} shm_data_struct, *shm_data_struct_t;

// ...create shm segment
seg = shmget(ftok(arg[0], 120), 1024, IPC_CREATE|IPC_EXCL));
shm_address = shmat(seg, (void *) 0, 0);
shm_ptr = (shm_data_struct_t_)shm_address;

// ...create and init mutex
pthread_mutexattr_t(&m_attr);
pthread_mutexattr_set_pshared(&m_attr,PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&shm_prt.mutex, &m_attr);
```

上圖是本段的圖. 上面的 typedef 不知道為甚麼要那樣寫(可能寫錯了). Project 1 中的原代碼是這樣寫的:
typedef struct gfserver_t gfserver_t;
上圖中的 IPC_CREATE 應為 IPC_CREAT, 因為編譯時: error: ‘IPC_CREATE’ undeclared, 且以下的

online 代碼用的 IPC_CREAT, 能編譯.



```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
// mutex attributes == specifies mutex behavior when
// a mutex is shared among processes
```

上圖是前面課中的圖, 方便看

Attach 的意思是 map virtual address to physical address, 而不是 attach to a process. 按上面弄出來的 shared memory 應該是所有 process 都可以用

<From online starts>

key_t ftok(pathname, id) :

The ftok() function uses the identity of the file named by the given pathname (which must refer to an existing, accessible file) ...
What would be the best practice for it? On my current system I can pass "/home/Andrew/anyfile" but it is not possible that other systems, on which my program has to work, will have this file.

Well, generally you would use a file associated with the application itself.
...
Worst case, if you have no configuration files for your application, try to use the executable itself. You can be pretty certain that it exists on the system somewhere (since you're running it).

pathname 就是你指定的文件名（已经存在的文件名），一般使用当前目录，如：
key_t key = ftok(".", 1); 这样就是将 fname 设为当前目录。

id is usually just set to some arbitrary char, like 'A'.

Before you can use a shared memory segment, you have to attach yourself to it using the shmat() call: void *shmat(int shmid, void *shmaddr, int shmflg); What does it all mean? Well, shmid is the shared memory ID you got from the call to shmget(). Next is shmaddr, which you can use to tell shmat() which specific address to use but you should just set it to 0 and let the OS choose the address for you. Finally, the shmflg can be set to SHM_RDONLY if you only want to read from it, 0 otherwise.

Here's a more complete example of how to get a pointer to a shared memory segment: key_t key; int shmid; char *data; key = ftok("/home/beej/somefile3", 'R'); shmid = shmget(key, 1024, 0644 | IPC_CREAT); data = shmat(shmid, (void *)0, 0); And bammo! You have the pointer to the shared memory segment! Notice that shmat() returns a void pointer, and we're treating it, in this case, as a char pointer. You can treat is as anything you like, depending on what kind of data you have in there.

Pointers to arrays of structures are just as acceptable as anything else.

Lets say you have the data pointer from the above example. It is a char pointer, so we'll be reading and writing chars from it. Furthermore, for the sake of simplicity, lets say the 1K shared memory segment contains a null-terminated string. It couldn't be easier. Since it's just a string in there, we can print it like this: printf("shared contents: %s\n", data); And we could store something in it as easily as this: printf("Enter a string: "); gets(data); Of course, like I said earlier, you can have other data in there besides just chars.

Sample code
from a good page: http://beej.us/guide/bgipc/output/html/multipage/shm.html

This program does one of two things: if you run it with no command line parameters, it prints the contents of the shared memory segment. If you give it one command line parameter, it stores that parameter in the shared memory segment.

Here's the code for shmdemo.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHM_SIZE 1024 /* make it a 1K shared memory segment */
int main(int argc, char *argv[])
{
key_t key;
int shmid;
char *data;
int mode;
if (argc > 2) {
fprintf(stderr, "usage: shmdemo [data_to_write]\n");
exit(1);
}
/* make the key: */
if ((key = ftok("shmdemo.c", 'R')) == -1) {
perror("ftok");
exit(1);
}

/* connect to (and possibly create) the segment: */
if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
perror("shmget");
exit(1);
}
/* attach to the segment to get a pointer to it: */
data = shmat(shmid, (void *)0, 0);
if (data == (char *)(-1)) {
```

```
perror("shmat");
exit(1);
}
/* read or modify the segment, based on the command line: */
if (argc == 2) {
printf("writing to segment: \"%s\"\n", argv[1]);
strncpy(data, argv[1], SHM_SIZE);
} else
printf("segment contains: \"%s\"\n", data);
/* detach from the segment: */
if (shmdt(data) == -1) {
perror("shmdt");
exit(1);
}
return 0;
}
```

以上紅色部分即體現了不同 process share 這個 memory.
先運行此程序(第一個 process), 若輸入為

```
./shared_memory_test helo
```

則輸出為

```
writing to segment: "helo"
```

此時再運行此程序(第二個 process), 若輸入為

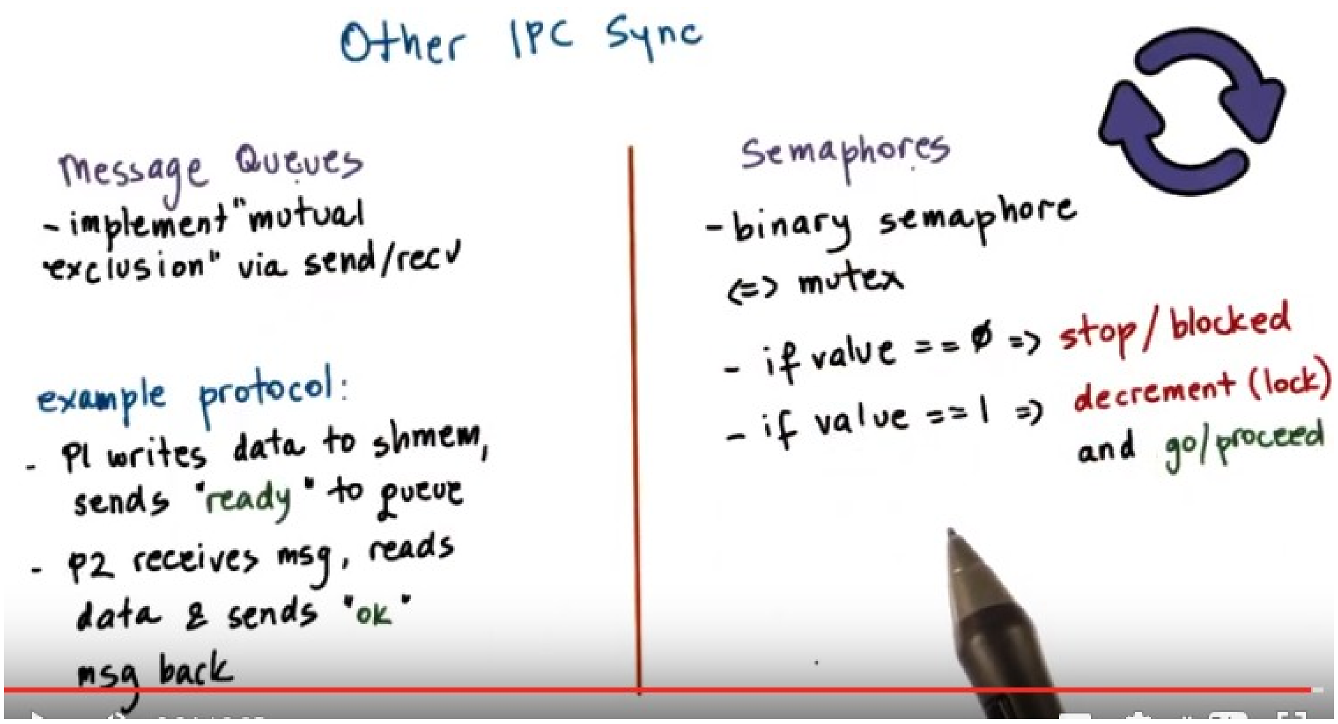
```
./shared_memory_test
```

則輸出還是為

```
writing to segment: "helo"
```

<From online ends>

For instance, let's look at this code snippet. Let's look here at how the shared memory segment is created. Here we are using the system V API. In the get operation, the segment id, the shared memory identifier, is uniquely created from the token operation where we use argument zero (即 arg[0]) from the command line, so the path name (即 arg[0]) for the program executable, and then some integer parameter, so in this case this is 120. We're also requesting that we create a segment size of 1 kilobyte (1024), and then we specify the areas permissions for that segment. Then using that segment identifier that's returned from the get operation. We are attaching this segment and that will provide us with a shared memory address (shm_address). So this (shm_address) is the virtual memory address in this instance of the process, in the execution of this particular process in its address space. That (shm_address) points to the physically shared memory. Now, we are casting that address to point to something that's of the following data type (即上面定義的那個 struct). 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 (char *data) is the actual byte stream that corresponds to the data (即該 shared memory 中的 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 (char *data). So as to avoid concurrent writes, race conditions, and similar issues. So this is how we will interpret what is 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. Now concerning the mutex attributes, we see that we have here set the, 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 piece. 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 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.


Other IPC Sync

Message Queues
- implement "mutual exclusion" via send/recv

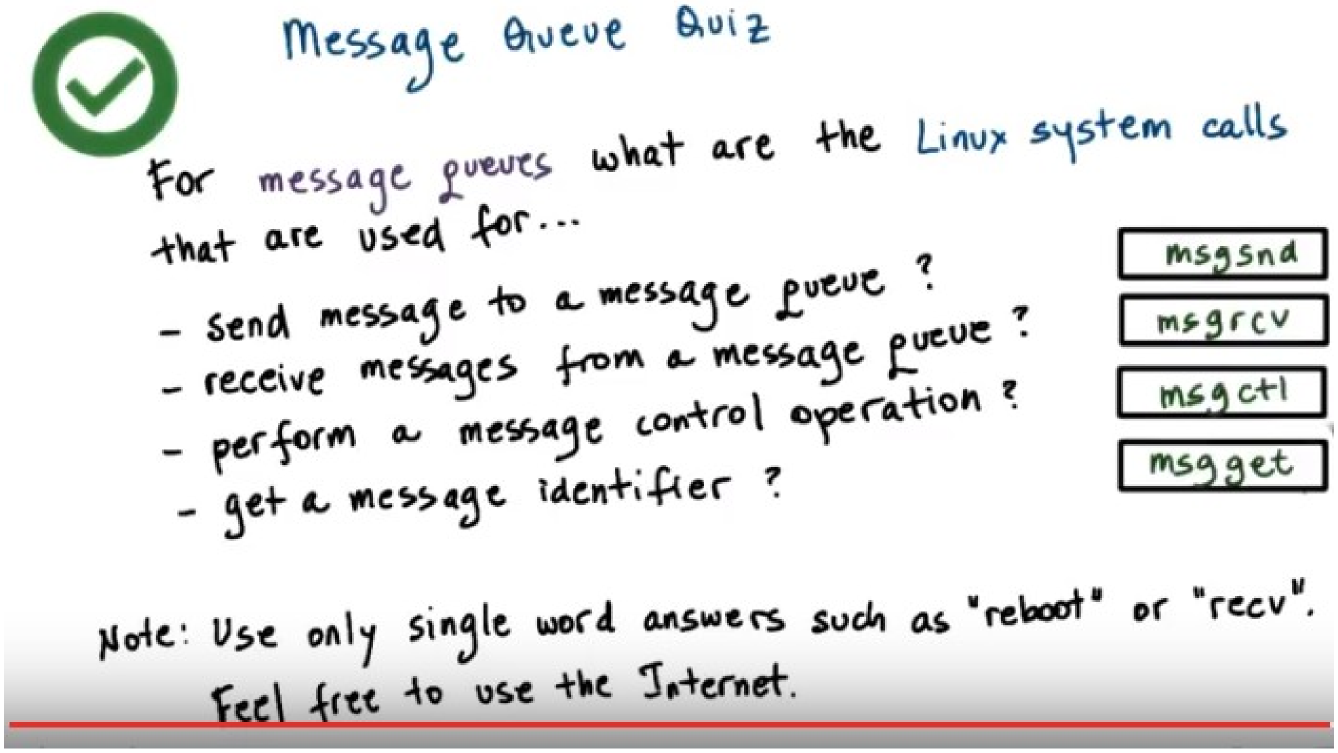
example protocol:
- P1 writes data to shmem, sends "ready" to queue
- P2 receives msg, reads data & sends "ok" msg back

Semaphores
- binary semaphore <=> mutex
- if value == 0 => stop / blocked
- if value == 1 => decrement (lock) and go/proceed

P3L4 會專門講 semaphore

15. In addition shared memory accesses can by synchronized operating system provided mechanisms for inter process interactions. This is particularly important because the process shared option for the mutex condition variables with pthreads, isn't necessarily 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 of protocol how this can be achieved. Two processes are communicating via shared memory and they're using message cues 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's okay 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 operating system supported synchronisation contract and a binary semaphore can have two values, zero one. And it can be achieved, the similar type of behavior like what is achieved with a mutex. Depending on the value of semaphore, a process is either allowed to proceed or it will be

stopped at the semaphore and it will have to wait for something to change. For instance, a binary semaphore with value zero and one, we use it in a following way. If its value's zero, the process will be blocked. And if its value is one, the semantics of the semaphore construct is such that a process will automatically decrement that values. It will turn it to zero, and it will proceed. So this decrement operation is equivalent to obtained a lock. In the instructor's notes, I'm providing a code example that uses shared memory and message queues and semaphores for synchronization. And the example uses the System V, or the System five API as a reference. The system five APIs 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, et cetera, message queues or semaphores. For both of these constructs are also posex equivalent to APIs.,


Message Queue Quiz

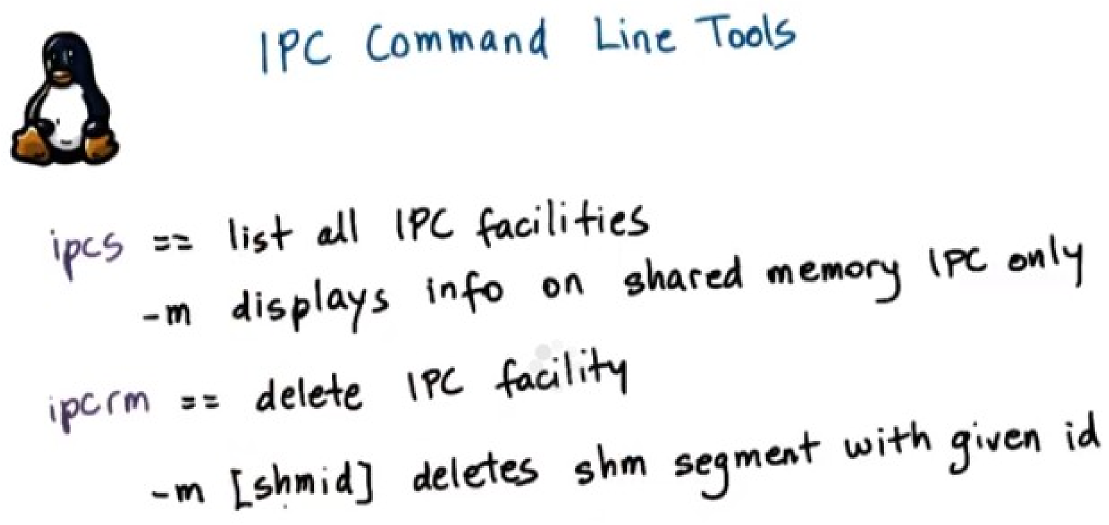
For message queues what are the Linux system calls that are used for...

- send message to a message queue? msgsnd
- receive messages from a message queue? msgrcv
- perform a message control operation? msgctl
- get a message identifier? msgget

Note: Use only single word answers such as "reboot" or "recv". Feel free to use the Internet.

16. Now, let's take a treasure hunt type of quiz concerning the Message Queue construct. The question has four parts. For a message queues, what are the Linux system calls that are used for? Send a message to a message queue? Receive messages from a message queue? Perform a message control operation? Or, to get a message identifier? Provide answers for each of the following questions. Remember to use only single word answers, like just reboot or just recv and feel free to use the Internet.

17. The answers to these questions are as follows. Sending messages to a message queue uses the following command, msgsnd, message send. Receiving a message uses the following msgrcv. Performing a control operation on the message can be done using the following command, msgctl, control. And finally obtaining an identifier for a message can be done using the msgget call.



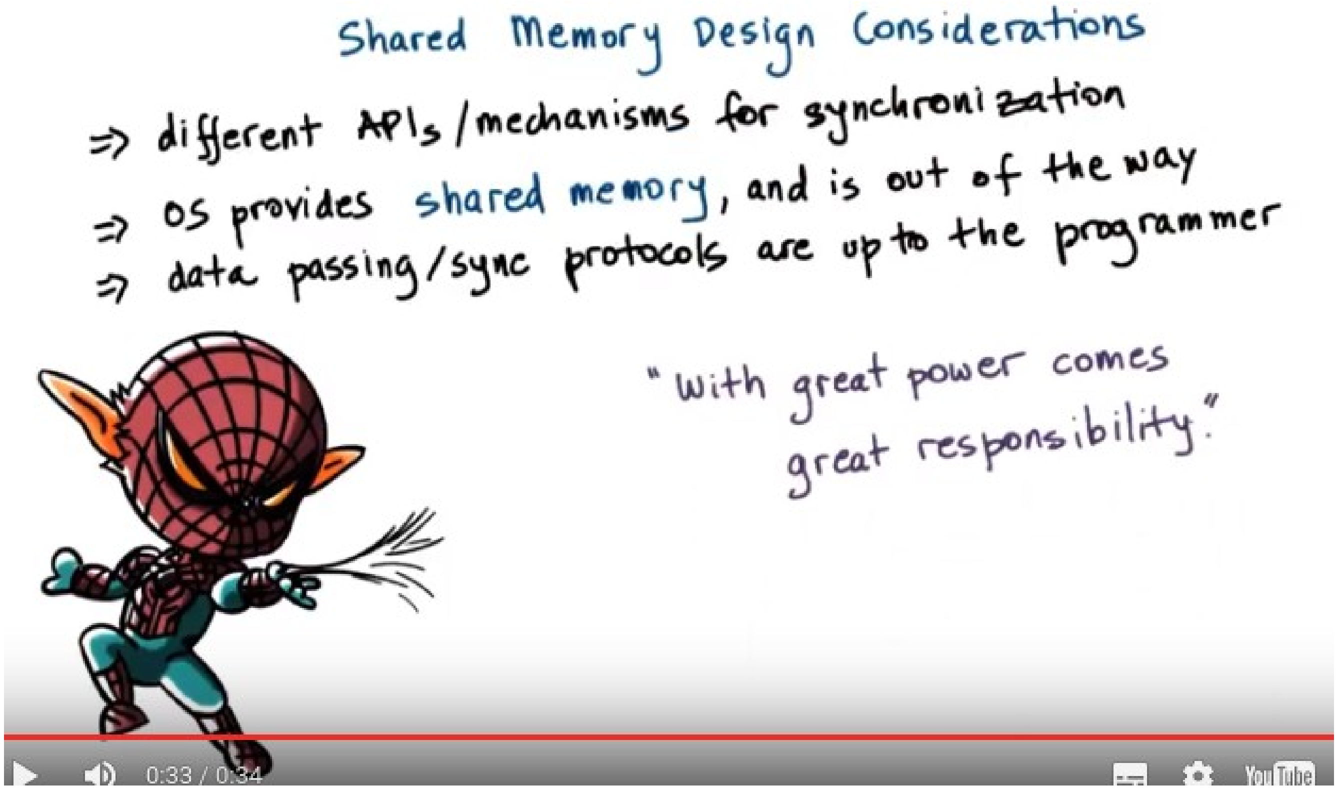
IPC Command Line Tools

ipcs == list all IPC facilities
-m displays info on shared memory IPC only

ipcrm == delete IPC facility
-m [shmid] deletes shm segment with given id

以上命令在我的 Linux terminal 中試過, 可以用.

18. As you start using IPC methods, it is useful to know that Linux provides some command line utilities for using IPC and shared memory in general. Ipcs will list all of the IPC facilities that exist in the system. This will include all types of IPC, 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.

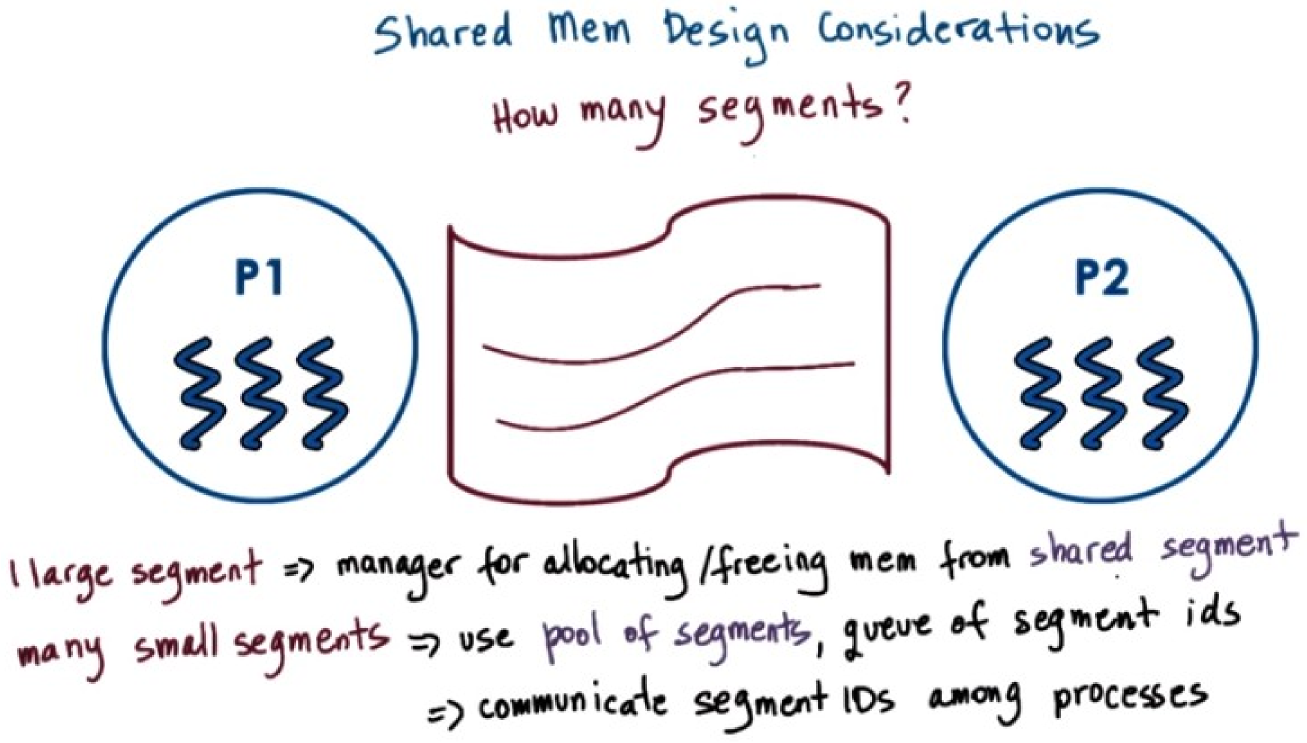


Shared Memory Design Considerations

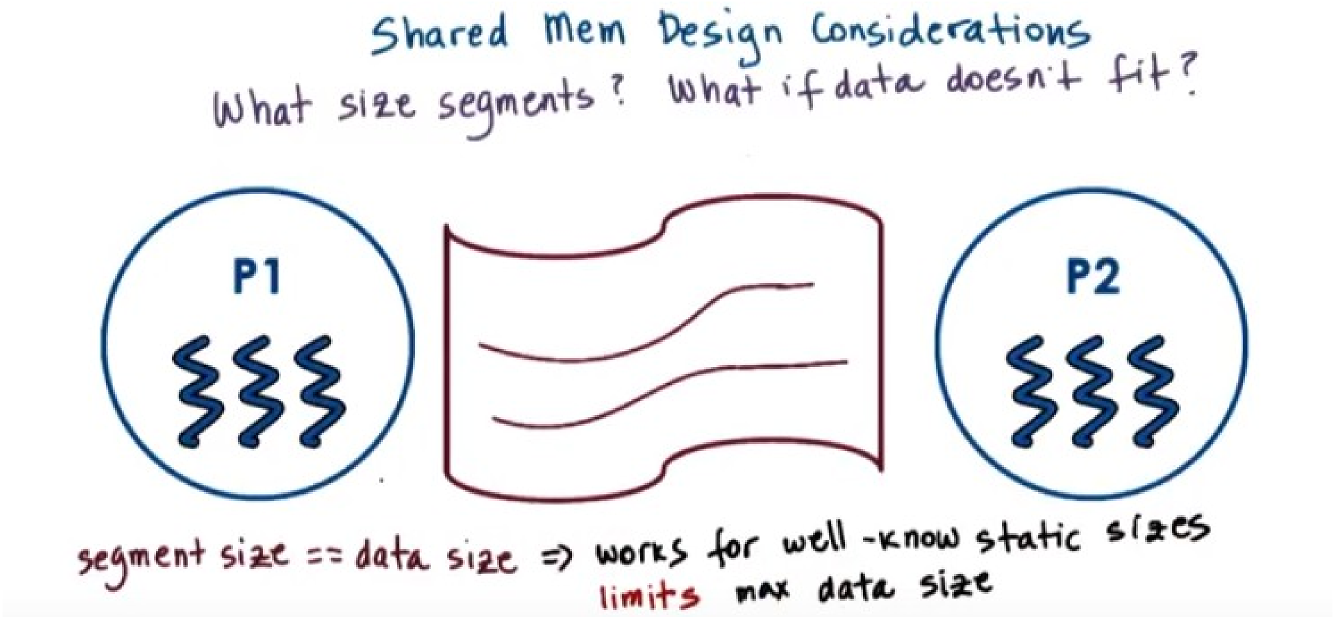
⇒ different APIs/mechanisms for synchronization
⇒ OS provides shared memory, and is out of the way
⇒ data passing/sync protocols are up to the programmer

"with great power comes great responsibility"

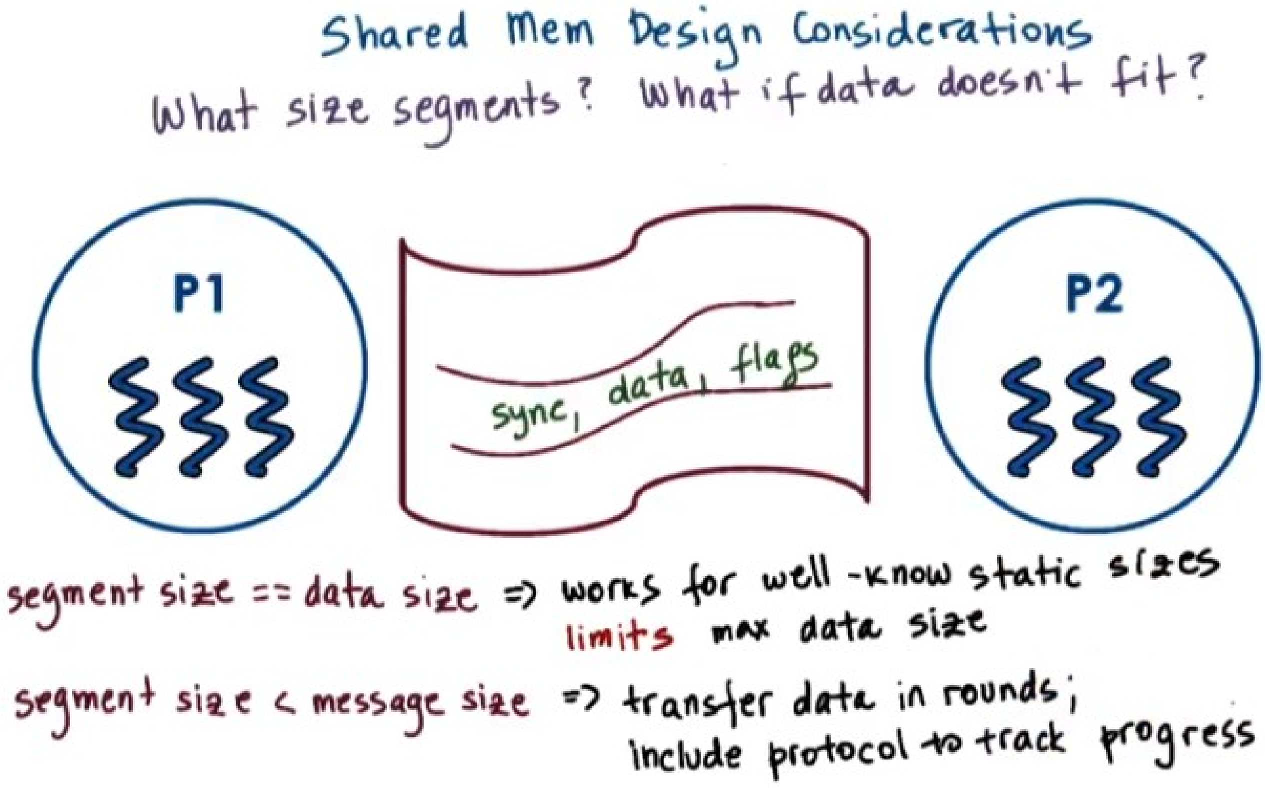
19. When using shared memory, the operating system doesn't restrict you how the memory will be used. However, the choice of the API or the specific mechanisms that will be used for synchronization are not the only decisions that you need to make. Remember, with shared memory, the operating system. Provides 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 more we will mention a few things that you can consider to assist with your design process


Shared Mem Design Considerations
How many segments?
P1
P2
1 large segment => manager for allocating /freeing mem from shared segment
many small segments => use pool of segments, queue of segment ids
=> communicate segment IDs among processes

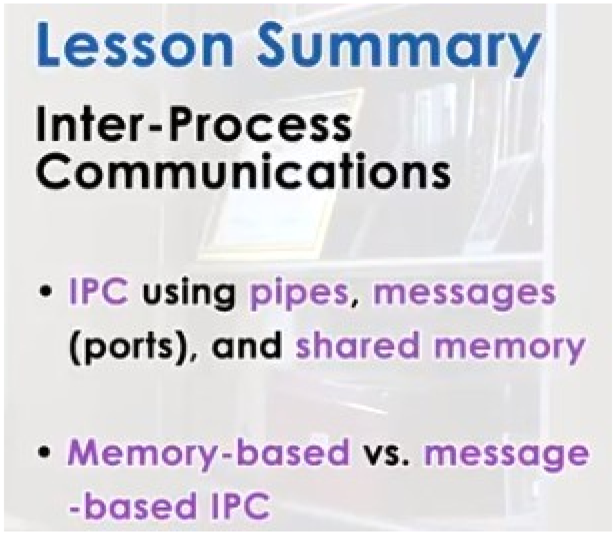
20. To make things concrete, let's consider two multi threaded processes in which the threads need to communicate via shared memory. First consider how many segments will your processes need to communicate. Will they use one large segment? In that case you will have to implement some type of management 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 use multiple segments, smaller ones, one for each pair-wise communication. If you choose to do this, it's probably a good idea to prealloacate, 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 will threads pick up which of the available segments they will end up using for their inter process communication. So, using some type of queue of segment identifiers will be, probably, a good idea for that. The tricky part here, if you are using a queue of segment identifiers, that means that a thread doesn't know up front which particular segment it's going to use for a communication with a peer thread in the other process. If that's important for the type of application that you're developing, you can consider communicating the segment identifier from one process to another via some other type of communication mechanism, like via message queue.


Shared Mem Design Considerations
What size segments? What if data doesn't fit?
P1
P2
segment size == data size => works for well-know static sizes
limits max data size

21. Another design question is how large should a segment be? That will work really well if the size of the data is known up front and static. It doesn't change. However, in addition to the fact that data sizes may not be static, that they may be dynamic, the other problem with this is that it will limit what is the maximum data that could be transferred between processes because typically an operating system will have a limit on the maximum segment size.


Shared Mem Design Considerations
What size segments? What if data doesn't fit?
P1
P2
sync, data, flags
segment size == data size => works for well-know static sizes
limits max data size
segment size < message size => transfer data in rounds;
include protocol to track progress

If you want to potentially support arbitrary message 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, P2 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.



Lesson Summary

Inter-Process Communications

- IPC using pipes, messages (ports), and shared memory
- Memory-based vs. message-based IPC

22. In this lesson we talked about inter-process communication, or IPC. We described several IPC mechanisms that are common in operating systems today. We spent a little bit 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 inter-process communication mechanisms in your projects.

23. As the final quiz, please tell us what you learned in this lesson. Also, we would love to hear your feedback on how we might improve this lesson in the future.