Memory

Virtual Memory

So a page table is a list of pages a process has accessed and the relevant virtual-physical mapping. We have already seen every page also has some permissions attached:

- *read*: the process can read from this page
- *write*: the process can write to this page
- *execute*: the process can execute code on this page

Memory

Virtual Memory

If a process tries to access a page it does not have the appropriate permission for an interrupts happens and the OS sends a segmentation violation signal to the process

Memory

Virtual Memory

If a process tries to access a page it does not have the appropriate permission for an interrupts happens and the OS sends a segmentation violation signal to the process

Even though, through the virtualisation, “all” memory is owned by the current process, it is still useful to have these permissions

Memory

Virtual Memory

If a process tries to access a page it does not have the appropriate permission for an interrupts happens and the OS sends a segmentation violation signal to the process

Even though, through the virtualisation, “all” memory is owned by the current process, it is still useful to have these permissions

This is so the process knows it is trying to read from/write to/execute some unexpected place in memory, rather than some place it should be. This catches many stupid programming errors

Memory

Virtual Memory

If a process tries to access a page it does not have the appropriate permission for an interrupts happens and the OS sends a segmentation violation signal to the process

Even though, through the virtualisation, “all” memory is owned by the current process, it is still useful to have these permissions

This is so the process knows it is trying to read from/write to/execute some unexpected place in memory, rather than some place it should be. This catches many stupid programming errors

Further, permissions are useful when we have shared memory, too

Memory

Virtual Memory

Another big benefit of VM is the natural protection of one process from another: as all user mode memory accesses go though the TLB, the TLB will simply prevent it even being possible for one process to overwrite the memory of another

Memory

Virtual Memory

Another big benefit of VM is the natural protection of one process from another: as all user mode memory accesses go though the TLB, the TLB will simply prevent it even being possible for one process to overwrite the memory of another

Or enable it if we want shared memory. Thus the TLB solves two big problems: memory protection and memory sharing

Memory

Virtual Memory

Another big benefit of VM is the natural protection of one process from another: as all user mode memory accesses go though the TLB, the TLB will simply prevent it even being possible for one process to overwrite the memory of another

Or enable it if we want shared memory. Thus the TLB solves two big problems: memory protection and memory sharing

A process only sees the virtual address: it can access anywhere it wants and the TLB takes care of things

Memory

Virtual Memory

Another big benefit of VM is the natural protection of one process from another: as all user mode memory accesses go though the TLB, the TLB will simply prevent it even being possible for one process to overwrite the memory of another

Or enable it if we want shared memory. Thus the TLB solves two big problems: memory protection and memory sharing

A process only sees the virtual address: it can access anywhere it wants and the TLB takes care of things

The kernel bypasses the TLB lookup and sees physical addresses, but can map back and forth for each process

Shared Memory

So now shared memory is very easy: just let the TLB do the mapping of virtual pages from different processes to the *same* physical pages

Shared Memory

So now shared memory is very easy: just let the TLB do the mapping of virtual pages from different processes to the *same* physical pages

This allows *shared libraries* (`.so` in Unix; DLLs in Windows; `.dylib` in MacOS X/macOS)

Memory

Virtual Memory

Shared Memory

So now shared memory is very easy: just let the TLB do the mapping of virtual pages from different processes to the *same* physical pages

This allows *shared libraries* (`.so` in Unix; DLLs in Windows; `.dylib` in MacOS X/macOS)

Many programs need to do mundane stuff like read or writing to files, formatted printing, drawing on the screen and so on

Memory

Virtual Memory

Shared Memory

So now shared memory is very easy: just let the TLB do the mapping of virtual pages from different processes to the *same* physical pages

This allows *shared libraries* (`.so` in Unix; DLLs in Windows; `.dylib` in MacOS X/macOS)

Many programs need to do mundane stuff like read or writing to files, formatted printing, drawing on the screen and so on

So *libraries* of such code are provided that the programmer can use and not have to reimplement it all themselves

Memory

Virtual Memory

If 10 processes are in memory, each of them using the library `read` function, does that mean there are 10 copies of the code for `read` scattered about in memory?

Memory

Virtual Memory

If 10 processes are in memory, each of them using the library `read` function, does that mean there are 10 copies of the code for `read` scattered about in memory?

Before the advent of shared libraries, yes

Memory

Virtual Memory

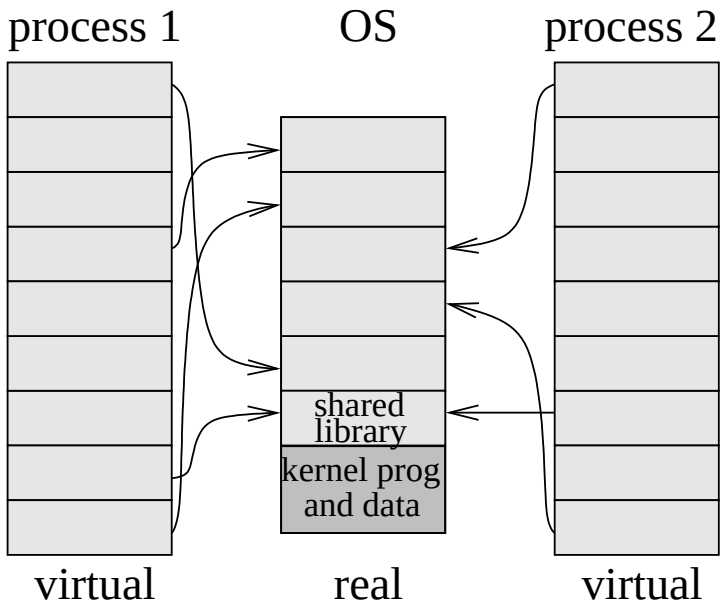
If 10 processes are in memory, each of them using the library `read` function, does that mean there are 10 copies of the code for `read` scattered about in memory?

Before the advent of shared libraries, yes

But now the use of virtual memory can let us *share* code between processes

Memory

Virtual Memory



Shared Libraries

Memory

Virtual Memory

Now the OS can load the code for `read` just once and direct all other processes to use that single copy

Memory

Virtual Memory

Now the OS can load the code for `read` just once and direct all other processes to use that single copy

This reduces memory usage, reduces pages faults and has other beneficial properties (see caching)

Memory

Virtual Memory

Now the OS can load the code for `read` just once and direct all other processes to use that single copy

This reduces memory usage, reduces pages faults and has other beneficial properties (see caching)

This works well as all processes can share identical and unchanging code pages. But *data* in libraries couldn't be shared like this though?

Memory

Virtual Memory

Now the OS can load the code for `read` just once and direct all other processes to use that single copy

This reduces memory usage, reduces pages faults and has other beneficial properties (see caching)

This works well as all processes can share identical and unchanging code pages. But *data* in libraries couldn't be shared like this though?

Perhaps we would need a private copy of the data pages for each process, since if one process updates the data that would mess things up for another process also using that data

Memory

Virtual Memory

Now the OS can load the code for `read` just once and direct all other processes to use that single copy

This reduces memory usage, reduces pages faults and has other beneficial properties (see caching)

This works well as all processes can share identical and unchanging code pages. But *data* in libraries couldn't be shared like this though?

Perhaps we would need a private copy of the data pages for each process, since if one process updates the data that would mess things up for another process also using that data

But there is another trick...