Memory

Physical Memory

If we can’t find a big enough free space, we can consider *compaction* of memory using a technique called *garbage collection*

Memory

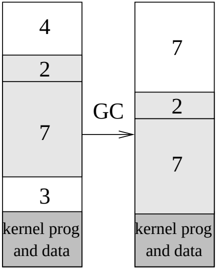
Physical Memory

If we can’t find a big enough free space, we can consider *compaction* of memory using a technique called *garbage collection*

The OS stops all running processes (i.e., stops scheduling processes); shifts their code and data around to close up the gaps; then lets the processes continue (i.e., starts scheduling again)

Memory

Physical Memory



Memory

Physical Memory

GC is not often used in general-purpose OSs

Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive (time consuming) operation to move all those bytes around

Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive (time consuming) operation to move all those bytes around
- this takes a lot of time away from running of processes

Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive (time consuming) operation to move all those bytes around
- this takes a lot of time away from running of processes
- the pause while things are moved is bad for interactive and real-time behaviour

Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive (time consuming) operation to move all those bytes around
- this takes a lot of time away from running of processes
- the pause while things are moved is bad for interactive and real-time behaviour
- the erratic nature of when GCs are needed leads to unpredictable behaviour from the OS

Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive (time consuming) operation to move all those bytes around
- this takes a lot of time away from running of processes
- the pause while things are moved is bad for interactive and real-time behaviour
- the erratic nature of when GCs are needed leads to unpredictable behaviour from the OS
- given the right kind of hardware support, better solutions completely avoiding the need for GC are possible

Memory

Physical Memory

GC *is* successfully used in user languages, e.g., Python, Haskell, Java

Memory

Physical Memory

GC *is* successfully used in user languages, e.g., Python, Haskell, Java

There are ways of implementing GC to avoid the stop-and-copy (ephemeral GC), or mitigating the overhead (generational GC) but even so it is not popular for OSs

Memory

Physical Memory

GC *is* successfully used in user languages, e.g., Python, Haskell, Java

There are ways of implementing GC to avoid the stop-and-copy (ephemeral GC), or mitigating the overhead (generational GC) but even so it is not popular for OSs

Exercise Reflect on whether it would be a good idea to implement an OS in Java (Hint: nobody serious does so!)

Memory

Physical Memory

So what happens when we can’t find a suitable free space for a new process (even if we have GC)?

Memory

Physical Memory

So what happens when we can’t find a suitable free space for a new process (even if we have GC)?

We may choose not to admit the process in the first place

Memory

Physical Memory

So what happens when we can’t find a suitable free space for a new process (even if we have GC)?

We may choose not to admit the process in the first place

Another possibility is the option of killing existing processes: we usually don’t want to and only if the new allocation is for a process that is sufficiently important (recall OOM killers)

Memory

Physical Memory

So what happens when we can't find a suitable free space for a new process (even if we have GC)?

We may choose not to admit the process in the first place

Another possibility is the option of killing existing processes: we usually don't want to and only if the new allocation is for a process that is sufficiently important (recall OOM killers)

Better is to *preempt* memory: take it away from one process and give it to another

Memory

Physical Memory

Remember that preemption takes a resource away from a process and returns it later in the same state

Memory

Physical Memory

Remember that preemption takes a resource away from a process and returns it later in the same state

For memory this means the bits in the memory when it is returned are unchanged from what they were when it was taken away

Memory

Physical Memory

Remember that preemption takes a resource away from a process and returns it later in the same state

For memory this means the bits in the memory when it is returned are unchanged from what they were when it was taken away

Even though that memory has been used by some other process and written its own data or code into it

Memory

Physical Memory

We can preempt a process and copy the contents of the memory it occupies to somewhere else: usually disk

Memory

Physical Memory

We can preempt a process and copy the contents of the memory it occupies to somewhere else: usually disk

Note that only the data need to be saved: the code is already on disk in the file that contains the program

Memory

Physical Memory

We can preempt a process and copy the contents of the memory it occupies to somewhere else: usually disk

Note that only the data need to be saved: the code is already on disk in the file that contains the program

Copying to disk is a (relatively) very slow operation: even the fastest disks are slow

Memory

Physical Memory

We can preempt a process and copy the contents of the memory it occupies to somewhere else: usually disk

Note that only the data need to be saved: the code is already on disk in the file that contains the program

Copying to disk is a (relatively) very slow operation: even the fastest disks are slow

Even solid state disks (SSDs)

Memory

Physical Memory

We can preempt a process and copy the contents of the memory it occupies to somewhere else: usually disk

Note that only the data need to be saved: the code is already on disk in the file that contains the program

Copying to disk is a (relatively) very slow operation: even the fastest disks are slow

Even solid state disks (SSDs)

So this kind of memory preemption has a large overhead

Memory

Physical Memory

We can preempt a process and copy the contents of the memory it occupies to somewhere else: usually disk

Note that only the data need to be saved: the code is already on disk in the file that contains the program

Copying to disk is a (relatively) very slow operation: even the fastest disks are slow

Even solid state disks (SSDs)

So this kind of memory preemption has a large overhead

This is a tradeoff of speed (time spent copying to and from disk) against process size (memory allocation)

Memory

Physical Memory

Swapping

Memory

Physical Memory

Swapping

The simplest case is preemption of the memory of an entire process

Memory

Physical Memory

Swapping

The simplest case is preemption of the memory of an entire process

When a process makes a request for an allocation that the OS cannot immediately satisfy the OS can try *swapping*

Memory

Physical Memory

Swapping

The simplest case is preemption of the memory of an entire process

When a process makes a request for an allocation that the OS cannot immediately satisfy the OS can try *swapping*

This is where one or more other processes are selected by the OS and they are copied out to disk to make space

Memory

Physical Memory

Swapping

The simplest case is preemption of the memory of an entire process

When a process makes a request for an allocation that the OS cannot immediately satisfy the OS can try *swapping*

This is where one or more other processes are selected by the OS and they are copied out to disk to make space

The best choice is usually a blocked process that couldn’t have been run right now anyway

Memory

Physical Memory

When a swapped process is scheduled again it must be copied back by the OS into memory first

Memory

Physical Memory

When a swapped process is scheduled again it must be copied back by the OS into memory first

Which might require swapping out something else to make room

Memory

Physical Memory

When a swapped process is scheduled again it must be copied back by the OS into memory first

Which might require swapping out something else to make room

Data is retrieved from where it was saved, while code is copied back from the original program file—this is why some OS's don't like you deleting programs while they are running

Memory

Physical Memory

This differs from overlays in that it is the OS that does the swapping, not the process doing it to itself

Memory

Physical Memory

This differs from overlays in that it is the OS that does the swapping, not the process doing it to itself

This makes it transparent to the process and the programmer doesn’t have to think about it

Memory

Physical Memory

This differs from overlays in that it is the OS that does the swapping, not the process doing it to itself

This makes it transparent to the process and the programmer doesn’t have to think about it

. . . but they should as swapping is very time consuming, and slows down the speed of execution of programs immensely

Memory

Physical Memory

This differs from overlays in that it is the OS that does the swapping, not the process doing it to itself

This makes it transparent to the process and the programmer doesn’t have to think about it

. . . but they should as swapping is very time consuming, and slows down the speed of execution of programs immensely

A good programmer will try to avoid the need for swapping by requesting memory allocations carefully

Memory

Physical Memory

This differs from overlays in that it is the OS that does the swapping, not the process doing it to itself

This makes it transparent to the process and the programmer doesn't have to think about it

. . . but they should as swapping is very time consuming, and slows down the speed of execution of programs immensely

A good programmer will try to avoid the need for swapping by requesting memory allocations carefully

Something that often is forgotten these days!

Memory

Physical Memory

The OS will take swapping into account when scheduling

Memory

Physical Memory

The OS will take swapping into account when scheduling

There is a clear interaction of scheduling and swapping processes: each will affect the other

Memory

Physical Memory

Variants:

Memory

Physical Memory

Variants:

- Only one process ever in memory, swapped as a whole when scheduled: simple, and used on very early systems

Memory

Physical Memory

Variants:

- Only one process ever in memory, swapped as a whole when scheduled: simple, and used on very early systems
- Swapping of processes: only marginally harder, and fits well with a partitioning system and fits well with scheduling

Memory

Physical Memory

Variants:

- Only one process ever in memory, swapped as a whole when scheduled: simple, and used on very early systems
- Swapping of processes: only marginally harder, and fits well with a partitioning system and fits well with scheduling
- Swapping *parts* of a process: not so easy as the OS has to work harder to determine which parts of a process's code or data might not be needed in the near future

Memory

Virtual Memory

Paging

Memory

Virtual Memory

Paging

This is all augmented by the idea of *paging*

Memory

Virtual Memory

Paging

This is all augmented by the idea of *paging*

Paging is similar to swapping, but simpler in concept

Memory

Virtual Memory

Paging

This is all augmented by the idea of *paging*

Paging is similar to swapping, but simpler in concept

And much harder in the hardware required

Memory

Virtual Memory

Paging

This is all augmented by the idea of *paging*

Paging is similar to swapping, but simpler in concept

And much harder in the hardware required

To describe paging we must first go back to pages

Memory

Virtual Memory

A big problem is memory fragmentation due to the irregular sizes of processes/partitions

Memory

Virtual Memory

A big problem is memory fragmentation due to the irregular sizes of processes/partitions

So to fix this we chop everything up into equally sized chunks

Memory

Virtual Memory

A big problem is memory fragmentation due to the irregular sizes of processes/partitions

So to fix this we chop everything up into equally sized chunks

Recall (from memory protection) a *page* is just a contiguous area of memory: e.g., 4096 bytes

Memory

Virtual Memory

A big problem is memory fragmentation due to the irregular sizes of processes/partitions

So to fix this we chop everything up into equally sized chunks

Recall (from memory protection) a *page* is just a contiguous area of memory: e.g., 4096 bytes

Hardware is designed so copying pages in and out of memory from disk is as efficient as possible

Memory

Virtual Memory

Next, we introduce *virtual* vs. *physical* addresses

Memory

Virtual Memory

Next, we introduce *virtual* vs. *physical* addresses

A physical address is what we are used to, just a numbering of the actual bytes in the system from 0 to n

Memory

Virtual Memory

Next, we introduce *virtual* vs. *physical* addresses

A physical address is what we are used to, just a numbering of the actual bytes in the system from 0 to n

A virtual address is a per-process fictional address

Memory

Virtual Memory

Next, we introduce *virtual* vs. *physical* addresses

A physical address is what we are used to, just a numbering of the actual bytes in the system from 0 to *n*

A virtual address is a per-process fictional address

The user process sees only the virtual addresses: the system will translate them on the fly into physical addresses

Memory

Virtual Memory

The OS has tables, one per process, called *page tables*, that contains the virtual-physical address mappings for each page in each process

Memory

Virtual Memory

The OS has tables, one per process, called *page tables*, that contains the virtual-physical address mappings for each page in each process

For example, with a page size of 4096 bytes, address 12298 is 10 bytes from the start of page 3: $12298 = 3 \times 4096 + 10$

Memory

Virtual Memory

The OS has tables, one per process, called *page tables*, that contains the virtual-physical address mappings for each page in each process

For example, with a page size of 4096 bytes, address 12298 is 10 bytes from the start of page 3: $12298 = 3 \times 4096 + 10$

Under the entry for page 3 in the page table for this process we might find the number 7, meaning physical page 7

Memory

Virtual Memory

The OS has tables, one per process, called *page tables*, that contains the virtual-physical address mappings for each page in each process

For example, with a page size of 4096 bytes, address 12298 is 10 bytes from the start of page 3: $12298 = 3 \times 4096 + 10$

Under the entry for page 3 in the page table for this process we might find the number 7, meaning physical page 7

So virtual address 12298 **in this process** refers to physical byte $7 \times 4096 + 10 = 28682$

Memory

Virtual Memory

In another process, virtual page 3 could be mapped to physical page 42

Memory

Virtual Memory

In another process, virtual page 3 could be mapped to physical page 42

And then the same virtual address 12298 **in this process** refers to physical byte $42 \times 4096 + 10 = 172042$

Memory

Virtual Memory

In another process, virtual page 3 could be mapped to physical page 42

And then the same virtual address 12298 **in this process** refers to physical byte $42 \times 4096 + 10 = 172042$

The *same* virtual address in different processes is mapped to *different* physical addresses

Memory

Virtual Memory

In another process, virtual page 3 could be mapped to physical page 42

And then the same virtual address 12298 **in this process** refers to physical byte $42 \times 4096 + 10 = 172042$

The *same* virtual address in different processes is mapped to *different* physical addresses

We use pages, of course, to make this translation manageable

Memory

Virtual Memory

The table only contains entries for pages that are actually in use by that process: this keeps the tables to a reasonable size

V page	P page
3	7
4	9123
5	121
10	1232
etc.	

Memory

Virtual Memory

The table only contains entries for pages that are actually in use by that process: this keeps the tables to a reasonable size

V page	P page
3	7
4	9123
5	121
10	1232
etc.	

Note: page tables contain page *mappings*, not pages

Memory

Virtual Memory

The table only contains entries for pages that are actually in use by that process: this keeps the tables to a reasonable size

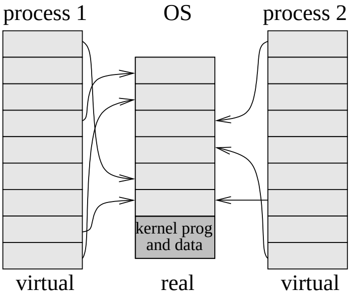
V page	P page
3	7
4	9123
5	121
10	1232
etc.	

Note: page tables contain page *mappings*, not pages

Note: though still called “tables”, in modern OSs they are likely to be more sophisticated datastructures, such as trees

Memory

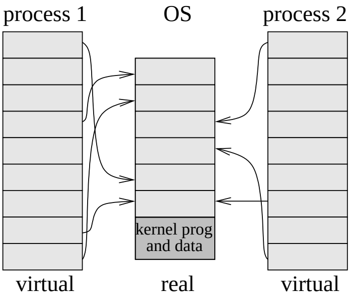
Virtual Memory



Every process gets its own complete and separate address space, mapped into the physical address space

Memory

Virtual Memory



Every process gets its own complete and separate address space, mapped into the physical address space

Even for the same userid: this is usually what you want, protection of one process from another

Memory

Virtual Memory

Where are these page tables?

Memory

Virtual Memory

Where are these page tables?

In kernel memory, of course: and a link to the table is kept in the process's PCB

Memory

Virtual Memory

Where are these page tables?

In kernel memory, of course: and a link to the table is kept in the process's PCB

But it sounds like, *on every memory access*, we have to do (a) a memory read of a page table to find the V to R mapping and then (b) a calculation to get the physical memory location and then (c) a memory access to the physical address we wanted

Memory

Virtual Memory

Where are these page tables?

In kernel memory, of course: and a link to the table is kept in the process's PCB

But it sounds like, *on every memory access*, we have to do (a) a memory read of a page table to find the V to R mapping and then (b) a calculation to get the physical memory location and then (c) a memory access to the physical address we wanted

- every data read

Memory

Virtual Memory

Where are these page tables?

In kernel memory, of course: and a link to the table is kept in the process's PCB

But it sounds like, *on every memory access*, we have to do (a) a memory read of a page table to find the V to R mapping and then (b) a calculation to get the physical memory location and then (c) a memory access to the physical address we wanted

- every data read
- every data write

Memory

Virtual Memory

Where are these page tables?

In kernel memory, of course: and a link to the table is kept in the process's PCB

But it sounds like, *on every memory access*, we have to do (a) a memory read of a page table to find the V to R mapping and then (b) a calculation to get the physical memory location and then (c) a memory access to the physical address we wanted

- every data read
- every data write
- every execute of an instruction

Memory

Virtual Memory

Where are these page tables?

In kernel memory, of course: and a link to the table is kept in the process's PCB

But it sounds like, *on every memory access*, we have to do (a) a memory read of a page table to find the V to R mapping and then (b) a calculation to get the physical memory location and then (c) a memory access to the physical address we wanted

- every data read
- every data write
- every execute of an instruction

This is clearly not sensible as it would be very slow

Memory

Virtual Memory

So, to be practically useful, this is supported by a piece of hardware called the *translation lookaside buffer* (TLB), part of the memory management unit (MMU)

Memory

Virtual Memory

So, to be practically useful, this is supported by a piece of hardware called the *translation lookaside buffer* (TLB), part of the memory management unit (MMU)

The TLB maintains its own copy of *a few* of the virtual-physical mappings from the page table of the current process and can translate very quickly between them

Memory

Virtual Memory

To repeat that: the table in the TLB is a *small subset* of the OS's page table mappings of the current process

Memory

Virtual Memory

To repeat that: the table in the TLB is a *small subset* of the OS's page table mappings of the current process

Only a small subset as TLB memory is very limited in size since it is very expensive to make memory that runs fast enough to make this mechanism practical: it contains perhaps just a few dozens of the virtual to physical mappings

To repeat that: the table in the TLB is a *small subset* of the OS's page table mappings of the current process

Only a small subset as TLB memory is very limited in size since it is very expensive to make memory that runs fast enough to make this mechanism practical: it contains perhaps just a few dozens of the virtual to physical mappings

Note (again): the TLB contains copies of the page *mappings*, not pages

Memory

Virtual Memory

The Intel Nehalem architecture has a 64 entry data TLB (and a 512 entry level 2 TLB); and a separate 64 entry instruction TLB

Memory

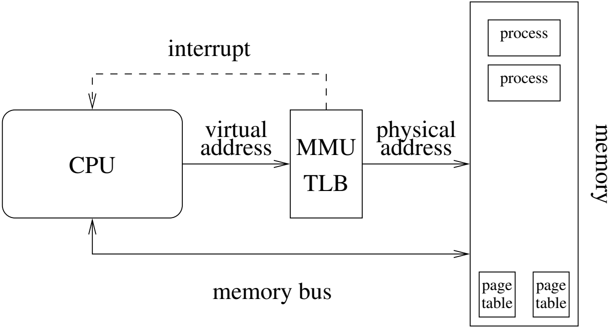
Virtual Memory

The Intel Nehalem architecture has a 64 entry data TLB (and a 512 entry level 2 TLB); and a separate 64 entry instruction TLB

Note that 64 entries typically corresponds to an area of $64 \times 4\text{k page} = 256\text{k}$ bytes, so while not huge, this isn't so bad as it might seem as first

Memory

Virtual Memory



The MMU and TLB are often physically part of the CPU package, for speed of access

Memory

Virtual Memory

When presented with an address from the CPU the TLB first looks the virtual page up in its table. If it is there is—a *TLB hit*—the memory access goes ahead at full speed using the physical address computed from the real page index found there

Memory

Virtual Memory

When presented with an address from the CPU the TLB first looks the virtual page up in its table. If it is there is—a *TLB hit*—the memory access goes ahead at full speed using the physical address computed from the real page index found there

If there is a *TLB miss* then it has to work a bit harder

Memory

Virtual Memory

When presented with an address from the CPU the TLB first looks the virtual page up in its table. If it is there is—a *TLB hit*—the memory access goes ahead at full speed using the physical address computed from the real page index found there

If there is a *TLB miss* then it has to work a bit harder

There are two popular techniques used

Memory

Virtual Memory

In a *hardware managed* TLB, the CPU/TLB itself stops what it is doing and searches for the page number in the page table (in memory) for the current process: this is called a *page walk*

Memory

Virtual Memory

In a *hardware managed* TLB, the CPU/TLB itself stops what it is doing and searches for the page number in the page table (in memory) for the current process: this is called a *page walk*

If it finds it, it installs it in the TLB table and carries on with the memory access

Memory

Virtual Memory

In a *hardware managed* TLB, the CPU/TLB itself stops what it is doing and searches for the page number in the page table (in memory) for the current process: this is called a *page walk*

If it finds it, it installs it in the TLB table and carries on with the memory access

The OS is not involved in the page walk, it is purely hardware

Memory

Virtual Memory

The second technique, a *software managed* TLB, simply raises a *TLB miss* interrupt on a TLB miss

Memory

Virtual Memory

The second technique, a *software managed* TLB, simply raises a *TLB miss* interrupt on a TLB miss

The OS then takes over and has to do the page walk

Memory

Virtual Memory

This deals with the case of when the requested page has already been allocated by the OS to the current process, so there is an entry in the page table for the page walk to find

Memory

Virtual Memory

This deals with the case of when the requested page has already been allocated by the OS to the current process, so there is an entry in the page table for the page walk to find

In either software or hardware case, if the requested virtual page is not yet allocated by the OS to the process and so not in its page table, the OS needs to allocate a page

Memory

Virtual Memory

A hardware managed TLB will now raise a *page fault* interrupt to pass control to the OS

Memory

Virtual Memory

A hardware managed TLB will now raise a *page fault* interrupt to pass control to the OS

A software managed TLB is already running the OS, as the OS is doing the page walk

Memory

Virtual Memory

A hardware managed TLB will now raise a *page fault* interrupt to pass control to the OS

A software managed TLB is already running the OS, as the OS is doing the page walk

The OS allocates a physical page, installs the new page mapping into the page table for that process for that page and writes the relevant page mapping into the TLB

Memory

Virtual Memory

A hardware managed TLB will now raise a *page fault* interrupt to pass control to the OS

A software managed TLB is already running the OS, as the OS is doing the page walk

The OS allocates a physical page, installs the new page mapping into the page table for that process for that page and writes the relevant page mapping into the TLB

(When the process is rescheduled) the memory access can then proceed

Memory

Virtual Memory

Of course, the OS may choose not to allocate a page and it could then send a segmentation violation signal to the process

Memory

Virtual Memory

Of course, the OS may choose not to allocate a page and it could then send a segmentation violation signal to the process

x86 and ARM processors have hardware managed TLBs

Memory

Virtual Memory

Of course, the OS may choose not to allocate a page and it could then send a segmentation violation signal to the process

x86 and ARM processors have hardware managed TLBs

SPARC and MIPS are software managed

Memory

Virtual Memory

Of course, the OS may choose not to allocate a page and it could then send a segmentation violation signal to the process

x86 and ARM processors have hardware managed TLBs

SPARC and MIPS are software managed

Terminology warning: a TLB miss when the page is already allocated and indexed in the page table is sometimes called a *minor* or *soft* page fault; while a miss on an unallocated page is a *major* or *hard* page fault

Memory

Virtual Memory

Speed relies crucially on the TLB containing a good proportion of the addresses currently being used: if a process writes wildly all over memory we are guaranteed to get TLB misses and slow memory access: lots of TLB misses and page walks or page fault interrupts

Memory

Virtual Memory

Speed relies crucially on the TLB containing a good proportion of the addresses currently being used: if a process writes wildly all over memory we are guaranteed to get TLB misses and slow memory access: lots of TLB misses and page walks or page fault interrupts

Fortunately, most well-written programs behave sensibly and tend to use the same addresses over and over, meaning lots of TLB hits

Memory

Virtual Memory

Speed relies crucially on the TLB containing a good proportion of the addresses currently being used: if a process writes wildly all over memory we are guaranteed to get TLB misses and slow memory access: lots of TLB misses and page walks or page fault interrupts

Fortunately, most well-written programs behave sensibly and tend to use the same addresses over and over, meaning lots of TLB hits

After a while, the TLB settles down, caching the indices of the pages the process is using, the *working set*

Memory

Virtual Memory

Note that a page fault can cost a lot of time

Register access	1 cycle
(L1 memory cache hit	≈ 2 cycles)
(L3 memory cache hit	≈ 50 cycles)
Main memory access	≈ 200 cycles
TLB miss (page in memory)	$\approx 10,000$ cycles
Page fault (page on disk)	$\approx 1,000,000,000$ cycles

These are very rough figures and are the combined overhead of OS operations and memory architecture