Introduction to Memory Management

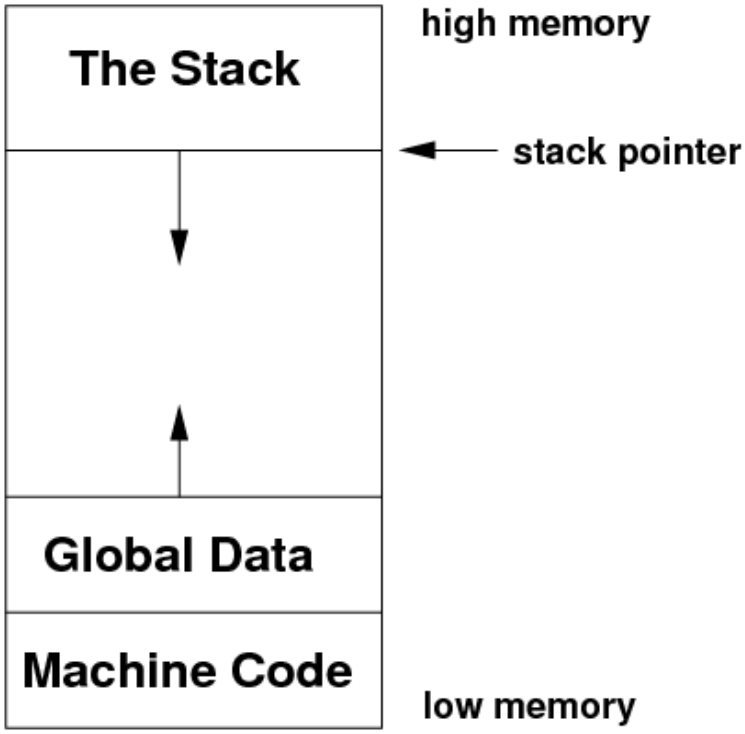
1 Introduction to Memory Management

1.1 What is Memory & Why Manage It?

- Every **process** needs some memory to store its variables and code. But if there is more than one process in memory at any one time, then the operating system must manage memory (as a resource) to prevent processes from reading/damaging each other's memory, and to ensure that each process has enough memory (not too much, not too little). The latter is most difficult, as a process' memory requirements may vary with time, and users schedule processes unpredictably.
- One thing to remember is that memory is device-like, and has an address decoder.

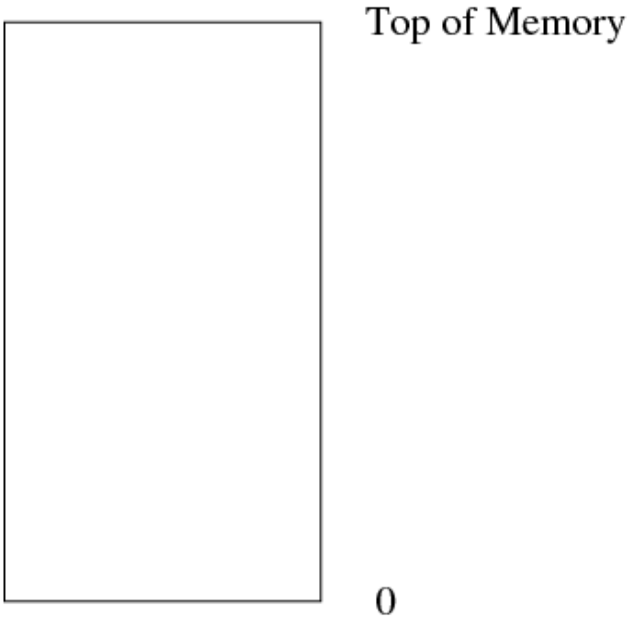
1.2 Process Compilation & Memory Locations

- High level languages are converted to machine code in several steps: compilation, linking to form an executable file, and at a later time loading of the executable into main memory to run.
- If the **linker** determines which variables get which addresses when running, then the program must always be loaded into the same location in memory every time it is run. This is because the addresses of variables and functions are hard-coded into the executable file.
- If the **loader** determines which variables get which addresses when running, then the program can be loaded into different locations in memory each time it is run. This dynamic loading adds overhead to the start-up of a new process.
- A generic memory map of a process looks like the following:



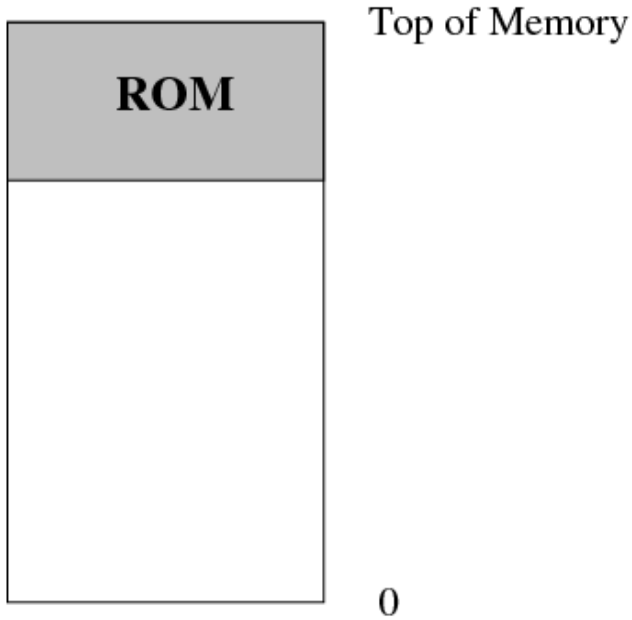
- The **code** section (also called the **text**) holds the program's instructions. This usually only needs to be read-only. The **data** section holds the process' global data and variables. The **stack** holds the local variables and the arguments to each routine.
- The grey areas are invalid, as the process initially doesn't use these locations. But the stack may grow down as the process calls its routines. On most operating systems the process can also ask for an increased data space. Invalid areas outside of the process remain invalid: the process cannot read from or write to these locations.
- Historically, memory management has evolved from none at all to being very sophisticated. Let's quickly cover the history of memory management.

1.3 Bare Machine



- Here, there is no memory management. We give the program all of the memory, with no limitation. This provides maximum flexibility to the user, and minimum hardware cost. There is no special memory hardware, and no need for any operating system software. However, there are no operating system services; the user must provide these.

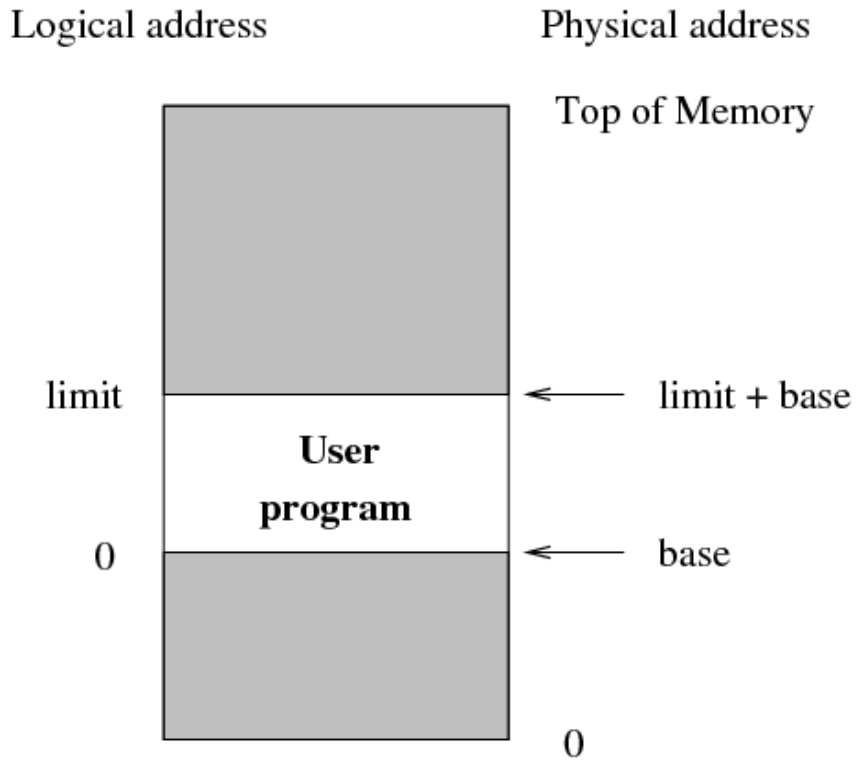
1.4 Operating System in ROM – Resident Monitor



- One way to protect the operating system itself is to put the operating system into ROM, thus it is hardware-protected. The operating system, unfortunately, still needs some RAM memory for its own operations, and this is unprotected from access or modification by any running program. Device addresses are not protected either.
- In this environment, it is impossible to protect processes from other processes. This sort of environment was used in the early minicomputer and microcomputer systems, e.g CP/M and the Apple][. We still have a hangover from these days with the BIOS in PC clones.
- A variation is to load an extended operating system into RAM. In this situation the operating system is totally unprotected. I have one word for this situation: MS-DOS.

1.5 Partitions

- One of the first mechanisms used to protect the operating system, and to protect processes from each other was **partitions**. Here, we add two hardware registers to the memory address decoder: the **base** and **limit** registers. When a process reads from or writes to address `X', the memory decoder adds on the value of the base register, so the actual operation become a read or write to address `base + X'.



- If the input address is lower than `0' or higher than `limit', the memory hardware considers this an error, and informs the operating system of a memory access error (usually via an interrupt). Thus, processes can only access memory within these limits.
- This addition of hardware allows multiple processes to be loaded into memory and to run without interference from each other. When the operating system performs a context switch, it must remember to switch the *memory maps* of two processes. This is done by changing the values of the two registers. Each process has its own individual pair of base/limit registers, and the operating system chooses these pairs so that process' memory maps never overlap, and each process has enough memory.
- On some systems, there is only a base register, and the limit is a constant. And on some systems have two register pairs, one for data and one for machine code. N.B The latter allows two memory address 0s! This is ok, because most processes never read their machine code themselves; the CPU reads and executes the code itself.
- One problem with partitions is how much memory to allocate initially? If too little, the process may run out of memory when data and stack collide. If too much, then memory is unused and wasted. It is usually impossible to change the base/limit registers once set, as there are other processes are usually immediately above/below.
- At this point, let us define two new terms:

  Logical
  : Memory is the memory and its location as seen by the process. In the partition system, all processes see memory as starting at location 0, and going up to the location defined by the limit register. But because the process (running in user mode) cannot modify the limit register, the process sees its logical memory as fixed in size.

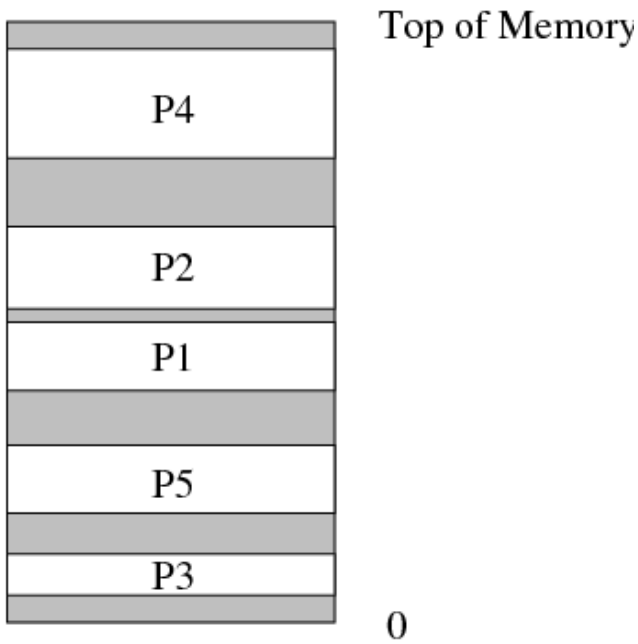
  Physical
  : Memory is the actual main memory and its location as seen by the operating system. This depends on the physical system, but in general main memory starts at location

0, and goes up to a top location set by the amount of RAM in the computer.

- Note well: it's unlikely that any process has its base register set to the value 0. Therefore, physical location 0 is unlikely to ever be logical location 0 for any process.

1.6 Allocating & Placing Partitions in Memory

- Partitions were used mainly on batch systems where there were: programs were waiting to be started (not in memory as yet), several processes ready to run, one running process, and zero or more blocked processes. To the operating system, the available **physical** memory looks like a number of **holes**:



- As new processes are chosen from the pool of programs waiting to be started, the operating system has to choose a partition size and physical location for this partition. We need a partition allocation algorithm. Here are some possible algorithms:

  First fit:
  Find the first hole where the waiting program will fit. This is fast algorithm, but usually leaves a smaller hole, except where the fit is exact.

  Best fit:
  Find the hole that best fits the job. i.e with the least left over. Surprisingly this is not a good algorithm, as it leaves a lot of tiny, useless holes all over the physical memory.

  Worst fit:
  Find the biggest hole, leaving the biggest remainder. This is usually the partition allocation algorithm chosen by system designers.

- When a process exits, **merging** can be done if there is an existing hole above and/or below it.

- **Fragmentation** describes when we have a lot of useless little holes. The system may get to a point when the available memory is enough to start a process, but it is in the form of holes too small to load the process. As the useless holes are outside of any process' memory, this situation is known as **external fragmentation**. This can be solved by

compaction, by moving existing processes (and their partition registers) to consolidate the holes. The operating system should use an algorithm to minimise the amount of **copying** to make the compaction as fast as possible.

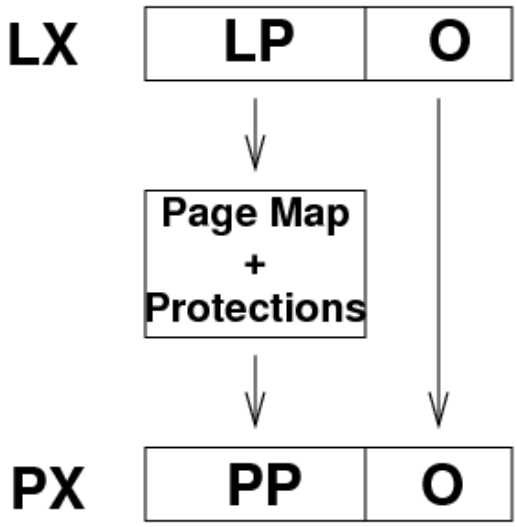
2 Pages

2.1 Problems with Partitions

- There are several problems with using paritions for memory management. First is the allocation of partition size: if the operating sets the partition to be too big, then the process doesn't use all of the memory and it is wasted memory; on the other hand, if the partition is too small, then there is not enough memory for the process to run. Once a partition size is chosen, it is essentially impossible to change.
- The second major problem is fragmentation of physical memory as the number of small holes builds up.
- The cause of both problems is the fact that the logical memory map is **contiguous**, i.e for each process, all the machine code is in one lump, all the data is in one lump etc. The process sees a single region of logical memory, and this is mapped to a single region of physical memory. If we could make these lumps *appear* to be contiguous to the process (i.e contiguous logical memory), but actually break the lumps into small physical memory units, then we could fill the unused phsyical holes with these small units.

2.2 Pages

- The concept of **pages** attempts to do just this. However, pages introduce their own problems. But first let's examine how the page mechanism performs memory mapping from logical to physical memory. Here is how page mapping works:
- Memory is broken into lots of **pages**, which are of *fixed size*. The page size is set by the hardware design, but is generally around 1K to 8K in size.
- We use the terminology of a page as seen by the process, and a **page frame** as a page as seen by the operating system in physical memory.
- A Memory Decoder (or **MMU)** maps the memory addresses to pages requested by a process to a set of page frames in physical memory. The decoder can also set protections on eachof the process' page; for example, a process' page may be marked as read–only, read–write, invalid, or privileged–mode access only.
- When a process access address LX, the MMU divides the address by the size of the system's pages. The resulting dividend is the logical page number LP, and the remainder becomes the offset into that page O (i.e LX=LP+O).
- The MMU then *maps* the logical page to a physical page frame, by using a lookup table. It then adds on O to get PX, the final physical address of the location in the main memory.



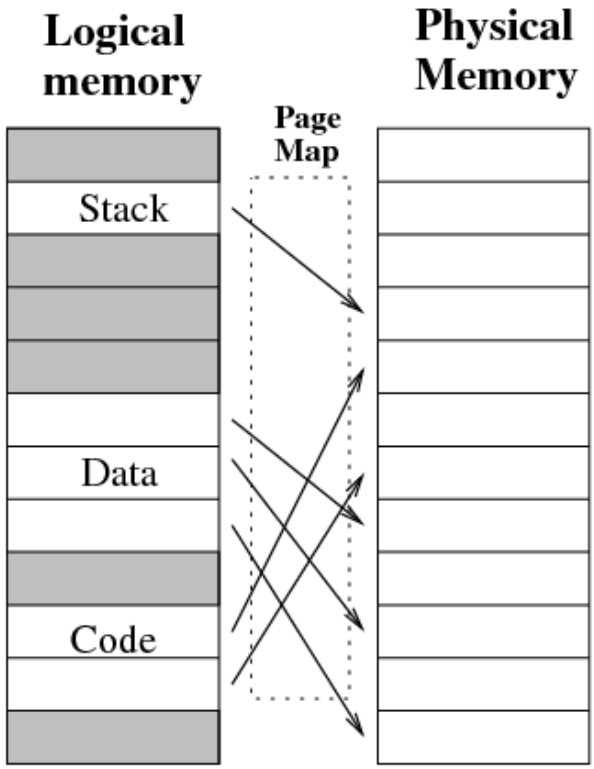
As an example, consider a system where pages are 2,000 bytes in size. A process tries to read from logical location 37,450. The MMU receives this location number from the address bus. It divides 37,450 by 2,000, obtaining logical page number 18 and offset 1,450.

The MMU consults the current page map. Each process has its own page map, just as with partitions each process had its own base/limit registers. In this page map, logical page 18 maps to physical page frame 115.

The MMU multiplies page frame 115 by 2,000 to get the bottom address of the page frame, 230,000. It then adds back on the offset, 1,450, to get 231,450. Finally, the MMU performs the requested operation on physical location 231,450.

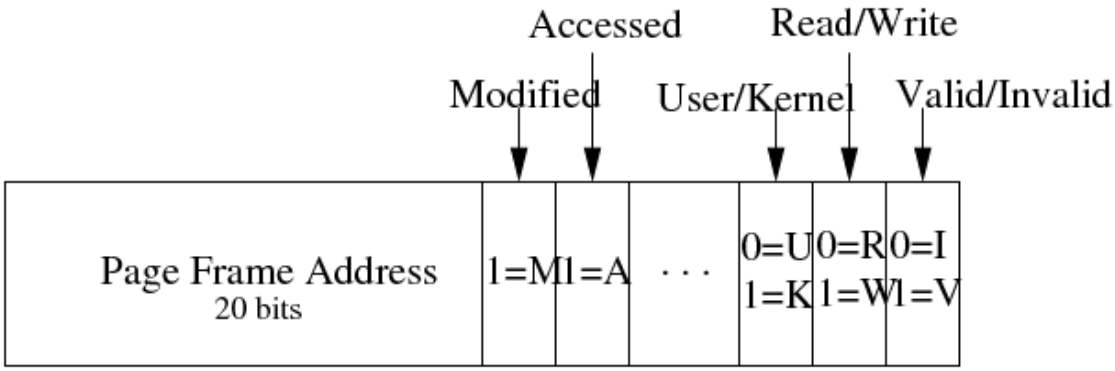
From the process' point of view, it has access location 37,450, which is on page 18. But the MMU has mapped this to physical location 231,450 on page frame 115.

- If the original page has a suitable protection, the memory access is permitted. Otherwise, the MMU sends an interrupt to the CPU to inform an appropriate interrupt handler of the protection error.
- The page map here is the crucial element that makes the system work. Although processes see their logical memory as being contiguous, their page map can spread their logical memory around as a number of separate page frames in physical memory. For example, a process' memory may actually be placed in physical memory like this:



- The `holes' (i.e the available physical memory areas) are now always the same size, that is, the size of the system's pages. So as long as there are enough free page frames, these can be allocated to a new processes as they arrive, or they can be added to the page map of existing process. The latter allows a process' allocated logical memory to grow at any time.
- In the partition scheme, each process' size *must* be less than the actual physical amount of memory. With pages, we can make a process be the size of the memory: we just don't allocate all the pages to the process. In other words, the process will occupy the entire address space, but most of that space will be **invalid**. In this situation, the data section and the stack are as far away from each other as they can possibly get, and they can grow much more than with the partition system.
- There is one drawback of the page system. When we do a context switch between two processes, we must save the page map for the old process out of the MMU, and load the page map for the new process into the MMU. These operations can be expensive if the page map is big.

2.3 An Example Page Entry



- Here is an example page entry (a single page to page frame mapping) from the Intel i386 and up. On this system, pages are 4K in size, and there are 1,048,576 page entries in the page map, numbered from logical page #0 up to logical page #1,048,575.
- On the left of the page entry is the matching page frame number. So, if this was page entry #23, and the page frame address was 117, then logical page 23 would be mapped by the MMU to page frame 117.
- On the right are the page permissions. If the Valid bit is 0, then no page frame has been mapped to this page, and any access will cause an error. Next is the Read/Write bit: if it is 0, only reads are allowed, if 1, writes are also allowed.
- When the User/Kernel bit is 1, access to the page can only be done when the CPU is in kernel mode. Generally, the kernel sets its own pages to have this protection.
- Finally, the Access and Modified bits are used by page replacement algorithms, which we will see in the next lecture.

2.4 Pages vs. Paging

- Memory management can be a confusing topic, and the terminology used doesn't make it easier. For the rest of these notes, I am using a terminology which is different from the textbook, but I believe will help you grasp the concepts of memory management. Here I will briefly explain my terminology, and also they way in which it differs from the textbook.
- A **page system** is a way of mapping from a process' **logical memory** (i.e its address space, or its memory map) to the **physical memory** on the computer. The process' **pages** are mapped to the system's **page frames**.
- **Paging** is a system which copies the contents of unused page frames out to disk, thus making the page frames free for use by other processes. The storage of page frame contents on disk is known as **virtual memory**.
- Note the following:

  Logical
  : memory: a memory space as seen by a process.

  Physical
  : memory: the system's memory space as seen by the operating system.

  Virtual
  : memory: the storage of page frame contents on disk.

  Page
  : system: a form of logical to physical memory mapping which uses a page map.

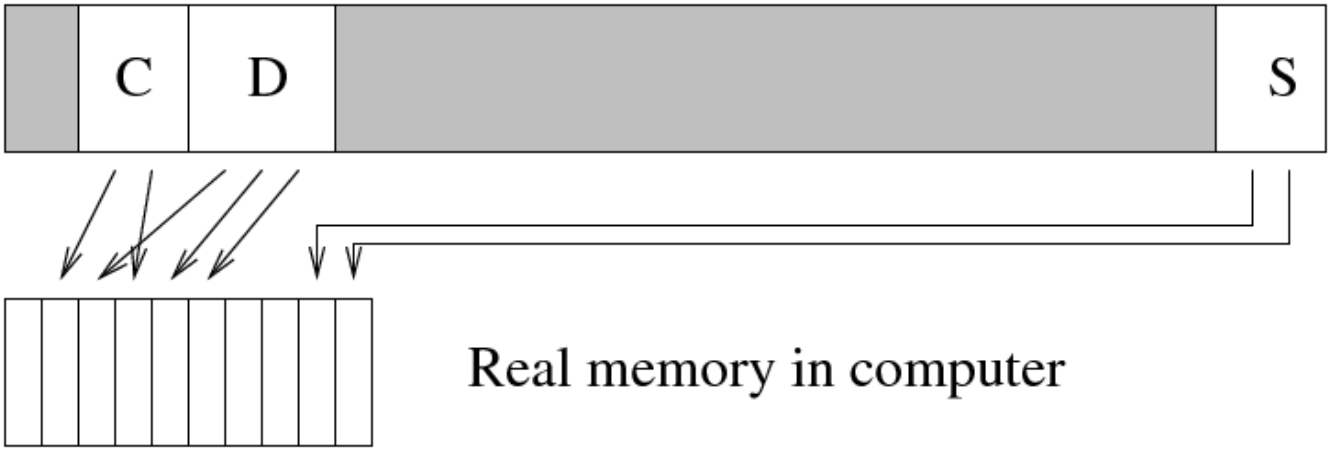
  Paging
  : system: a mechanism which copies the contents of unused page frames out to disk.

- The textbook uses the term `virtual addresses' where I use `logical addresses', and it uses the term `paging' to mean both a `page system' and a `paging system'.

2.5 Huge Logical Memory Maps

- The introduction of 32-bit address spaces (that is, 4 Gbytes of addressable memory) in the early '70s brought a problem: the cost of 4Gbytes of real memory was unimaginable.

Page systems were very useful because they allowed a process to occupy all 4Gbytes of logical address space, but in reality it occupied only a small amount of the system's physical memory.



- This mapping allowed the data and stack of the process to be placed at opposite ends of this huge address space, thus virtually ensuring that they would never collide. They are then able to *grow*; this allows the process to obtain memory as required, rather than requesting it all when it starts execution.

2.6 Problems of Paged Memory Management

- The MMU logical/physical conversion is slow, certainly much slower than the adding on of a base register. This affects the speed at which memory can be read/written.
- One optimisation here is to `cache' the last N logical to physical conversions, as usually the next memory access will be on the same page. This is usually done in the MMU hardware with a **Translation Lookaside Buffer** (TLB).
- When context switching between processes, the operating system must `unmap' all of the pages of the old process, and map in the pages of the next process. If 20 pages have to be unmapped and 30 pages have to be mapped in, the operating system must send 50 commands to the MMU. This can be very slow with processes that have a large number of page map entries.
- A page system also suffers from **internal** fragmentation. If a process uses N pages for its code, then N–1 will be full, and 1 will be partially full. The same sort of internal fragmentation applies for the data and stack regions in each process.
- The number of mappings the MMU must be able to do can be huge. For example, if the page size is 1K, and the address space is 4G, the MMU's page map must hold 4 million page mappings. Thus the MMU's hardware must be very big.
- The page size also affects all of the above problems. If it is small, fragmentation is lower, but the number of mappings is larger and context switching slower.

2.7 Sharing Pages

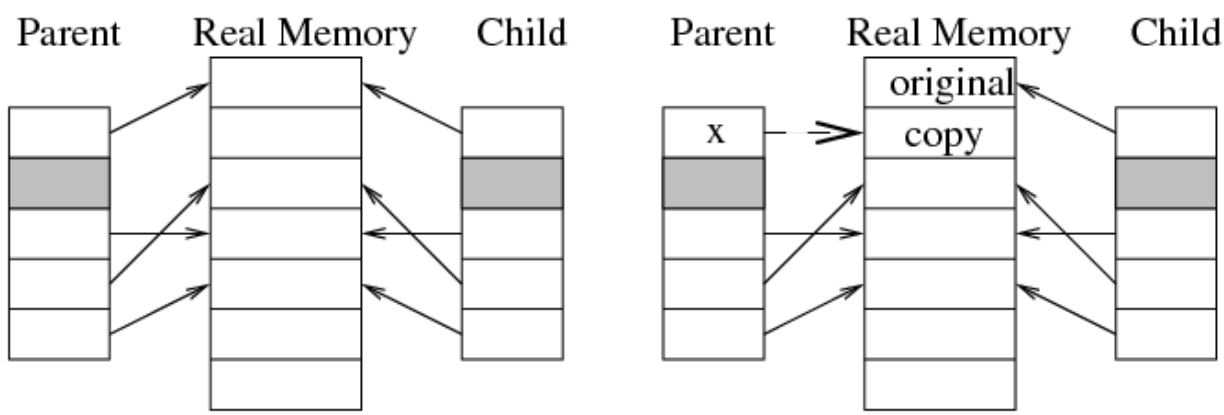
- One thing that a paged system can do is to **share** pages between 2 or more processes. A good example is when 10 `vi` editors are running. All have exactly the same code, so the operating system can allow each process to access the same page frames which hold the

code. The pages for each process are set to be **read-only**, to prevent one process from altering the code for itself and the other 9 processes.

- Sharing can also be done, for example, when a process `fork()`s. The operating system must make copies of the code, data and stack. With page sharing, it can share the page frame that hold the code, and so the code won't need to be copied. Page sharing is also very useful for sharing common libraries between several processes.

2.8 Copy-on-Write

- Copy-on-write is a method which can be implemented on a paged system to avoid doing *any* copying on a `fork()`.
- After a fork, we have two nearly identical processes, the **parent** and the **child**. Note that the contents of their pages are the same, and they will stay the same until one or the other changes a memory value in a page.
- Instead of making copies of the pages in the `fork()`, the operating system lets both processes *share* all the pages, but marks every page as **read-only**.
- If one process tries to alter a page, a memory error occurs. The operating system receives an interrupt, sees that the page is shared, makes a **copy** of the page by physically copying the contents to another page frame and mapping the 2nd page frame to the current process, and gets the process to retry the memory access.



- Not performing the memory copies unless necessary is a big saving under Unix, where a `fork()` is nearly always followed by a `exec()`. This makes the `fork/exec` much faster.

2.9 Operating System Use of Page Entry Protections

- We have seen that in the page entries in the MMU, the last three bits set the available protections on each page. Although the bits provide basic protection, the operating system may wish to set similar protections for very different reasons.
- The operating system can use the set the basic protections on each process' pages, but it also has to remember *why* each page has this protection. Below is an example table of possible memory areas available to a process, and includes the hardware protections and the operating system reason for setting the permissions that way.

Memory Area	Hardware Protection	OS Protection	Reason
Page 0	Invalid	Invalid	Catches NULL pointer use

Code/Constants	Read–only	Read–only	Unchanging memory, may be shared
Global Data	Read–write	Read–write	Normal variable use
	Read–only	Copy–on–write	Allows untouched pages to be shared
The Heap	Read–write	Read–write	Normal variable use
	Read–only	Copy–on–write	Allows untouched pages to be shared
Invalid	Invalid	Invalid	Pages above heap
Memory Mapped	Read–only	Read–only	Read–only file
File	Read–write	Read–write	Read–write file
	Read–only	Locked region	Record locking on file region
	Read–only	Copy–on–write	Allows unaltered sections to be shared
Shared Memory	Read–only	Read–only	Another process has r–w
Region	Read–write	Read–write	Area is shared read–write
	Read–only	Locked region	Record locking on memory region
	Read–only	Copy–on–write	Allows unaltered sections to be shared
Invalid	Invalid	Fill–on–use	Pages below stack
The Stack	Read–write	Read–write	Normal variable use
	Read–only	Copy–on–write	Allows untouched pages to be shared
Shared Libraries	Read–only	Read–only	Unchanging memory, may be shared
OS Code/Constants	Kernel read–only	Read–only	
OS Global Data	Kernel read–write	Read–write	
Invalid	Invalid	Invalid	
OS Stack	Kernel read–write	Read–write	
Device Pages	Kernel read–write	Read–write	

- Observe how part of the process' logical memory map is occupied by the operating system's pages, but these are marked as only accessible in kernel mode. When a process performs a TRAP instruction to do a system call, the CPU switches over to kernel mode, and so the operating system's pages immediately become visible. The TRAP instruction then starts executing the trap handler code at a known location in the memory. The same thing happens when interrupts or exceptions arrive. This is how the operating system makes itself appear when required.

File translated from TEX by TTH, version 3.85.
On 8 Dec 2010, 08:43.