Memory-management Strategies

8.1 Background

Selection of a memory-management scheme for a system depends on many factors, especially on the hardware design of the system.

8.1.1 Basic Hardware

Main memory and the registers built **into the processor itself** are the only general-purpose storage that the CPU can access directly.

* Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
* The same **cannot be said of main memory**, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock.
* In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing.
* **The remedy** is to add fast memory (**cache**) between the CPU and main memory.
* **To manage a cache** built into the CPU, the hardware automatically **speeds up** memory access without any operating-system control.
* We must additionally **protect user processes** from one another.
* This protection must be provided by the hardware because the operating system doesn't usually intervene between the CPU and its memory accesses.

**Hardware implements this production in several different ways:**

* We first need to make sure that each process has a separate memory space.
* To have multiple processes loaded in memory for concurrent execution.
* To determine the range of legal addresses that the process may access.
* **We can provide this protection by using two registers,**

1- **The base register** holds the smallest legal physical memory address.

2- **The limit register** specifies the size of the range.

* Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
* The base and limit registers can **be loaded only** by the operating system, which uses a special **privileged** instruction (executed in kernel mode).
* **This scheme allows** the operating system to **change the value of** the registers but prevents user programs from changing the registers' contents.

8.1.2 Address Binding

* Depending on the memory management in use, the process may be moved between disk and memory during its execution.
* The processes on the disk that are waiting to be brought into memory for execution form the **input queue**.
* Addresses in the source program are generally **symbolic** (such as the variable count).
* A compiler typically **binds** these symbolic addresses to **relocatable** addresses (such as "14 bytes)
* **The linkage editor** or loader in turn binds the **relocatable** addresses to **absolute** addresses (such as 74014).
* Each binding is a mapping from one address space to another.

**Classically, the binding of instructions and data to memory addresses can be done at any step along the way:**

* Compile time. If you **know at compile time** where the process will reside in memory, then absolute code can be generated. The MS-DOS .COM-format programs are bound at compile time.
* If, at some later time, the starting location changes, then it will be necessary to recompile this code.
* Load time. If it **is not known at compile time** where the process will reside in memory, then the compiler must generate **relocatable code.**
* In this case, final binding is delayed until load time.
* If the starting address changes, we need only reload the user code to incorporate this changed value.
* Execution time. If the process **can be moved** during its execution from one memory segment to another, then binding must be delayed until run time.
* Special hardware must be available for this scheme to work.

8.1.4 Dynamic Loading

To obtain **better memory-space utilization**, we can **use dynamic loading**.

With dynamic loading, a routine is not loaded **until it is called.**

All routines are kept on disk in a **relocatable load format**. The main program is loaded into memory and is executed.

* When a routine needs to call another routine, the calling routine first **checks** to see whether the other routine has been loaded.
* If it has not, **the relocatable linking loader is called** to load the desired routine into memory and to update the program's address tables to reflect this change.
* Then control is **passed** to the newly loaded routine.

**The advantage of dynamic loading** is that a routine is loaded only when it is needed.

* Useful when large amounts of code are needed to handle error routines.

8.1.5 Dynamic Linking and Shared Libraries

**Dynamically linked libraries:** are system libraries that are linked to user programs when the programs are run.

* Some operating systems support only **static linking:** **in which system libraries are treated like any other object module and are combined by the loader into the binary program image.**
* **Dynamic linking,** in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time.
* With dynamic linking, a **stub** is included in the image for each library routine reference. **The stub** is **a small piece of code** that indicates how to **locate the appropriate memory**-resident library routine or **how to load the library** if the routine is not already present.

**More than one version** of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. This system is also known as **shared libraries**.

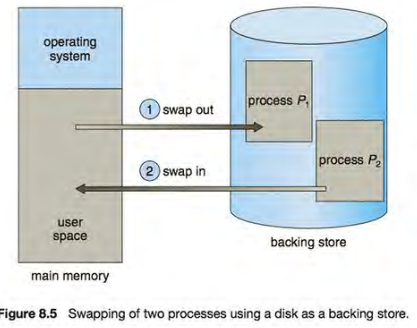
* Unlike dynamic loading, **dynamic linking** and **shared libraries** generally **require** help from the operating system.

8.2 Swapping

A process, can be **swapped** temporarily **out of memory** to a backing store and then brought back into memory for continued execution.

* Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus **increasing** the degree of multiprogramming in a system.

8.2.1 Standard Swapping



* **Standard swapping** involves **moving processes** between main memory and **a backing store**.
* The backing store is commonly a fast disk.
* 1-It must be large enough to accommodate copies of all memory images for all users.
* 2-It must provide direct access to these memory images.

The system maintains a **ready queue** consisting of all processes **whose memory images are on the backing store** or **in memory** and are ready to run.

8.2.2 Swapping on Mobile Systems

**Mobile systems typically do not support swapping in any form because:**

1-The resulting space constraint.

2- The limited number of writes that flash memory can tolerate before it becomes unreliable .

3- The poor throughput between main memory and flash memory in these devices.

* Mobile devices generally use **flash memory** rather than more spacious hard disks as their persistent storage.
* When free memory falls below a certain threshold, Apple's iOS asks applications to voluntarily **relinquish** allocated memory.
* Android **does not support swapping** and adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its application state to flash memory so that it can be quickly restarted.

8.3 Contiguous Memory Allocation

**The memory is usually divided into two partitions:** one for **the resident operating system** and one for the user processes.

Programmers usually place the operating system in low memory where the interrupt vector resides.

We usually want several user processes to reside in memory at the same time.

* We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.
* In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

8.3.1 Memory Protection

We can prevent a process from accessing memory it does not own by combining two ideas:

* The **MMU maps** the logical address dynamically **by adding** the value in the relocation register. This mapped address is sent to memory.
* When the CPU scheduler selects a process for execution, the **dispatcher** loads the **relocation and limit registers** with the correct values as part of the context switch.
* Because every address generated by a CPU is checked against these registers, we can protect both the OS and the other users' programs and data **from being** modified by this running process.
* The **relocation-register scheme** provides an effective way to allow the operating system's size to change dynamically.

8.3.2 Memory Allocation

Now we are ready to tum to memory allocation.

One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions.

* **Each partition** may contain exactly **one** process.
* Thus, the degree of multiprogramming is bound by the number of partitions.
* In this **multiple-partition method**, when a partition is **free**, a process is **selected** from the input queue and is loaded into the free partition.
* When the process **terminates**, the partition becomes **available** for another process.
* In the **variable-partition scheme**, the operating system keeps a **table** indicating which parts of memory are available and which are occupied.

1. Initially, all memory is available for user processes and is considered **one large block** of available memory.
2. A **hole**: memory contains a set of holes of various sizes.

* At any given time, then, we have a list of **available block sizes** and an **input queue**.
* The operating system can order the input queue according to a scheduling algorithm.
* The memory blocks available comprise a set of holes of various sizes scattered throughout memory.
* When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
* If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.
* When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
* If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.
* This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes.
* There are many **solutions to this problem**. The first-fit, best-fit, and worst-fit strategies:

• **First fit**. Allocate the first hole that is big enough.

• **Best fit.** Allocate the smallest hole that is big enough.

• **Worst fit**. Allocate the largest hole.

Simulations have shown that both **first fit and best fit** are **better** than worst fit in terms of decreasing time and storage utilization.

8.3.3 Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation.**

* **External fragmentation** exists when there is **enough total memory** space to satisfy a request but the **available spaces are not contiguous**: storage is fragmented into a large number of small holes.
* **One solution** to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block.
* It is possible only if relocation is dynamic and is done at execution time.
* **Another possible solution** is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available.(segmentation and paging)

8.4 Segmentation

Provide a memory **mechanism** that mapped the programmer's view to the actual physical memory.

8.4.1 Basic Method

* The programmer talks about "the stack," "the math library," and "the main program" without caring what addresses in memory these elements occupy.
* **Segments vary in length**, and the length of each is intrinsically defined by its purpose in the program.
* Elements within a segment are identified by their offset from the beginning of the segment:
* The first statement of the program, the seventh stack frame entry in the stack, the fifth instruction of the Sqrt (), and so on.
* A logical address space is a collection of segments.
* Each segment has a **name** and a **length**. The addresses specify both the segment name and the offset within the segment.
* segments are numbered .Thus, a logical address consists of a two tuple:

<segment-number, offset>.

Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

* A C compiler might create separate segments for the following:

1. The code.

2. Global variables

3. The heap, from which memory is allocated

4. The stacks used by each thread

5. The standard C library.

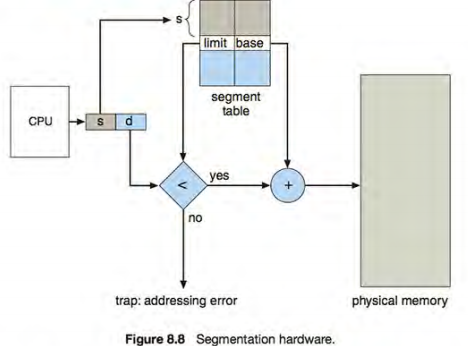
8.4.2 Segmentation Hardware

**Although the programmer can now refer to objects in the program by a two dimensional address, the actual physical memory is still a one dimensional sequence of bytes.**

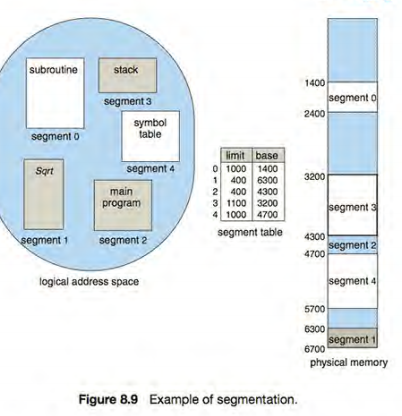
Thus, we must define an implementation to map two-dimensional programmer-defined addresses into one-dimensional physical addresses.

* This mapping is effected by a **segment table**.
* Each **entry** in the segment table has a **segment base** contains the **starting** physical address where the segment resides in memory.
* And the **segment limit** specifies the **length** of the segment.
* **A logical address** consists of two parts: a **segment number**, s, and an **offset**, d, into that segment.

1. The segment number is used as an index to the segment table.
2. The **offset** d of the logical address must be between 0 and the segment limit.
3. If **it is** not, we trap to the operating system (logical addressing attempt beyond end of segment).
4. When an offset is legal, it is added to the segment base to produce **the address in physical memory** of the desired byte.



8.5 Paging



**Segmentation permits the physical address space of a process to be noncontiguous**.

Paging is another memory-management scheme that offers this advantage.

* Paging **avoids** external fragmentation and the need for compaction, whereas segmentation does not.
* It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.
* **The problem arises because**, when code fragments or data residing in main memory need to be swapped out, space must be found on the backing store.
* Paging in its various forms is **used in most operating** systems through cooperation between the operating system and the computer hardware.

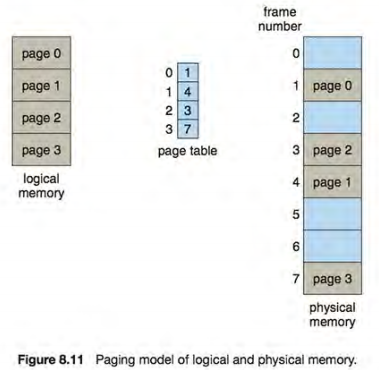
8.5.1 Basic Method

* **The basic method for implementing paging involves**

1. Breaking **physical memory** into fixed-sized blocks called **frames**.
2. Breaking **logical memory** into blocks of the same size called **pages**.

* When a process is to be executed,

1. Its pages are **loaded** into any available memory frames from their source (a file system or the backing store).
2. The backing store is **divided** into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.
3. Every address generated by the CPU is divided into two parts: a **page number** (p) and a **page offset (d)**. The page number is used as an index into a **page table.**

\

* When we use a paging scheme, we have **no external fragmentation**: any free frame can be allocated to a process that needs it.
* Some **internal** fragmentation.
* If process size is **independent** of page size, we expect internal fragmentation to average one-half page per process.
* When a process arrives in the system to be executed, its size, expressed in pages, is examined.
* Each page of the **process needs one frame**. Thus, if the process requires n pages, **n frames** must be available in memory.
* The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process and the second process into another frame and so on.
* The **allocation details** of physical memory-which frames are allocated, which frames are available, how many total frames there are, these information kept in a data structure called a **frame table**.
* Also, the operating system must be aware that user processes operate in user space, and all logical addresses must be **mapped** to produce physical addresses.

8.5.2 Hardware Support

Each operating system has its own methods for storing page tables.

* The hardware implementation of the page table can be done in several ways.
* **In the simplest case**, the page table is implemented as **a set of dedicated registers**.
* These registers should be built with **very high-speed logic** to make the paging-address translation **efficient**.
* Every access to memory must go through the paging map, so efficiency is a major consideration.
* Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map.
* Rather implementing a large page table which is not feasible, the page table is kept in main memory, and **a page-table base register** (**PTBR**) **points** to the page table. Changing page tables requires changing only this one register.
* The problem with this approach is the time required to access a user memory location.
* Two memory accesses are needed to access a byte (one for the page-table entry, one for the byte) slowed by factor of 2.
* **The standard solution** to this problem is to use a special, small, fast lookup hardware cache called a **translation look-aside buffer** (**TLB**) which is associative high speed memory.
* Each entry in the TLB consists of two parts: a key (or tag) and a value.

**The TLB is used with page tables in the following way:**

* The TLB contains only a few of the page-table entries.

1. When a logical address is generated by the CPU, its page number is **presented** to the TLB.

* If the page **number is found**, its frame number is immediately available and is used to access memory.
* If the page **number is not in the TLB** (known as a TLB **miss**), a memory reference to the page table must be made (**automatically** in hardware or via an **interrupt** TO OS).

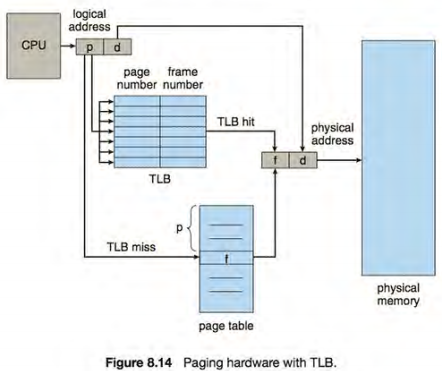
2- When the frame number is obtained, we can use it to access memory.

3-In addition, we add the **page number** and **frame number** to the TLB, so that they will be found quickly on the next reference.

* If the TLB is already full of entries, an existing entry must be selected for **replacement** (by algorithms like LRU).
* Some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB.
* TLB entries for key kernel code are wired down.
* Some TLBs store **address-space identifiers** (ASIDs) in each TLB entry.
* ASID uniquely **identifies** each process and provide address-space **protection** for that process.
* An ASID allows the TLB to contain entries for several different processes simultaneously.

**If the TLB does not support separate ASIDs,**

* Then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process **does not use the wrong translation information**.
* Otherwise, the TLB could **include old entries** that contain valid virtual addresses but have **incorrect or invalid physical addresses** left over from the previous process.
* The **percentage of times** that the page number of interest is found in the TLB is called the **hit ratio.**



8.5.3 Protection

Memory protection in a paged environment is accomplished by **protection bits** associated with each frame.

* Normally, these bits are kept in the **page table**.
* One bit can define a page to be read-write or read-only.
* Every reference to memory goes through the page table **to find the correct frame number**.
* At the same time that the physical address is being computed, the protection bits can be **checked** to verify that no writes are being made to a read-only page.
* An attempt to write to a read-only page causes **a hardware trap** to the operating system (or memory-protection violation).
* One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit.
* **Valid**, the associated page is in the process's logical address space and is thus a legal (or valid) page.
* **Invalid**, the page is not in the process's logical address space.
* Many processes use only a **small fraction** of the address space available to them.
* So, it’s **wasteful t**o create page table with entries for every page in the address range.
* Some systems provide hardware, in the form of a **page-table length register (PTLR),** to indicate the size of the page table.

8.5.4 Shared Pages

**An advantage of paging** is the possibility of sharing common code.

* Important in a time-sharing environment.
* If the code is reentrant code (or pure code), however, it can be shared.
* Reentrant code **is non-self-modifying** **code**: it never changes during execution.
* Two or more processes can execute the same code at the same time.
* Each process has its **own copy of registers** and **data storage** to hold the data for the process's execution.
* The data for two different processes will, be different.
* Other heavily used programs **can also be shared**- compilers, window systems, run-time libraries, database systems, and so on.

8.6 Structure of the Page Table

I couldn’t make a summary please look page 372

8.7 Example: Intel 32 and 64-bit Architectures