Frame vs. Page vs. Segment



* A program can be subdivided into segments
  + may vary in length
  + there is a maximum length
  + Addressing consists of two parts:
  + segment number
  + an offset
  + Similar to dynamic partitioning
* Eliminates internal fragmentation
* Usually visible
* Provided as a convenience for organizing programs and data
* Typically the programmer will assign programs and data to different segments
* For purposes of modular programming the program or data may be further broken down into multiple segments
  + Question: Why the programmer must be aware of the maximum segment size limitation?

Whereas paging is invisible to the programmer, segmentation is usually visible

and is provided as a convenience for organizing programs and data. Typically, the

programmer or compiler will assign programs and data to different segments. For

purposes of modular programming, the program or data may be further broken down

into multiple segments. The principal inconvenience of this service is that the programmer

must be aware of the maximum segment size limitation.

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of **segments .** It is not required that all segments of all programs be of the same length, although there is a maximum segment length. As with paging, a logical address using segmentation consists of two parts, in this case a segment number and an offset.

Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning. In the absence of an overlay scheme or the use of virtual

memory, it would be required that all of a program’s segments be loaded into memory for execution. The difference, compared to dynamic partitioning, is that with segmentation a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation. However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less.

the principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation

Paging

* Partition memory into equal fixed-size chunks that are relatively small
* Process is also divided into small fixed-size chunks of the same size

Page Table

* Maintained by operating system for each process
* Contains the frame location for each page in the process
* Processor must know how to access for the current process
* Used by processor to produce a physical address
* Address generated by CPU is divided into:
  + **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  + **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit



* + For given logical address space 2*m* and page size*2*n

Fetch Policy

Determines when a page should be brought into memory

Demand Paging

* Brings pages into main memory when a reference is made to a location on the page
* Many or few page faults when process is first started?
* What happens after more and more pages are brought in?
* Principle of locality suggests that as, most future references will be to pages that have recently been brought in, and page faults should drop to a very low level

Many page faults.

With **demand** paging ,

a page is brought into main memory only when a reference is made to a location on

that page. If the other elements of memory management policy are good, the following

should happen. When a process is first started, there will be a flurry of page

faults. As more and more pages are brought in, the principle of locality suggests that

most future references will be to pages that have recently been brought in. Thus,

after a time, matters should settle down and the number of page faults should drop

to a very low level.

Prepaging

* Pages other than the one demanded by a page fault are brought in
* Exploits the characteristics of most secondary memory devices
* If pages of a process are stored contiguously in secondary memory it is more efficient to bring in a number of pages at one time

What is the downside?

Ineffective if extra pages are not referenced

With **prepaging ,** pages other than the one demanded by a page fault are

brought in. Prepaging exploits the characteristics of most secondary memory

devices, such as disks, which have seek times and rotational latency. If the pages of

a process are stored contiguously in secondary memory, then it is more efficient to

bring in a number of contiguous pages at one time rather than bringing them in one

at a time over an extended period. Of course, this policy is ineffective if most of the

extra pages that are brought in are not referenced.

The prepaging policy could be employed either when a process first starts up,

in which case the programmer would somehow have to designate desired pages, or

every time a page fault occurs. This latter course would seem preferable because

it is invisible to the programmer. However, the utility of prepaging has not been

established [MAEK87].

Prepaging should not be confused with swapping. When a process is swapped

out of memory and put in a suspended state, all of its resident pages are moved out.

When the process is resumed, all of the pages that were previously in main memory

are returned to main memory.

Placement Policy

* Determines where in real memory a process piece is to reside
* Important design issue in a segmentation system
* Paging or combined paging with segmentation placing is irrelevant because hardware performs functions with equal efficiency
* For NUMA systems an automatic placement strategy is desirable

The placement policy determines where in real memory a process piece is to reside.

In a pure segmentation system, the placement policy is an important design issue;

policies such as best-fit, first-fit, and so on, which were discussed in Chapter 7 , are

possible alternatives. However, for a system that uses either pure paging or paging

combined with segmentation, placement is usually irrelevant because the address

translation hardware and the main memory access hardware can perform their

functions for any page-frame combination with equal efficiency.

There is one area in which placement does become a concern, and this is a

subject of research and development. On a so-called nonuniform memory access

(NUMA) multiprocessor, the distributed, shared memory of the machine can be

referenced by any processor on the machine, but the time for accessing a particular

physical location varies with the distance between the processor and the memory

module. Thus, performance depends heavily on the extent to which data reside

close to the processors that use them [LARO92, BOLO89, COX89]. For NUMA

systems, an automatic placement strategy is desirable to assign pages to the memory

module that provides the best performance.

Replacement Policy

* Deals with the selection of a page in main memory to be replaced when a new page must be brought in
* Objective: the page that is removed should be the page least likely to be referenced in the near future

The more elaborate the replacement policy the greater the hardware and software overhead to implement it

In most operating system texts, the treatment of memory management includes a

section entitled “replacement policy,” which deals with the selection of a page in

main memory to be replaced when a new page must be brought in. This topic is

sometimes difficult to explain because several interrelated concepts are involved:

• How many page frames are to be allocated to each active process

• Whether the set of pages to be considered for replacement should be limited

to those of the process that caused the page fault or encompass all the page

frames in main memory

• Among the set of pages considered, which particular page should be selected

for replacement

We shall refer to the first two concepts as *resident set management , which is dealt*

with in the next subsection, and reserve the term *replacement policy for the third*

concept, which is discussed in this subsection.

The area of replacement policy is probably the most studied of any area of

memory management. When all of the frames in main memory are occupied and

it is necessary to bring in a new page to satisfy a page fault, the replacement policy

determines which page currently in memory is to be replaced. All of the policies

have as their objective that the page that is removed should be the page least likely

to be referenced in the near future. Because of the principle of locality, there is

often a high correlation between recent referencing history and near-future referencing

patterns. Thus, most policies try to predict future behavior on the basis of

past behavior. One trade-off that must be considered is that the more elaborate and

sophisticated the replacement policy, the greater will be the hardware and software

overhead to implement it.

Project 5: Data Structures

**tlb entry:** this is a building block for tlb

**tlb:** this is the TLB to speedup the address translation performance

**physical\_memory\_t**: what is the data structure?

**page\_table\_t**: what is the data structure?