

CHAPTER VIII

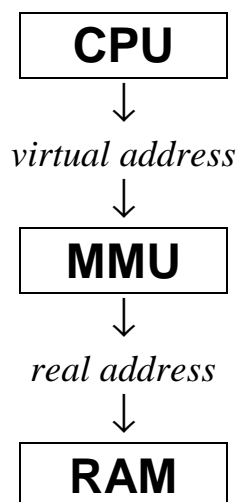
VIRTUAL MEMORY

8.1 INTRODUCTION

- Virtual memory combines two important ideas:
 - (1) **non-contiguous allocation of memory:**

Virtual memory can map process address spaces into disjoint blocks of physical memory called **pages** or **segments**: this is like allocating to a student a bed in one dorm room, closet space in another room and a book shelf in a third room.

This is achieved through the use of **address translation**: we add between the CPU and the main memory a **memory management unit** (MMU) translating the **program addresses**—or **virtual addresses**— into **real physical addresses**.



As John Ousterhout, formerly from U.C. Berkeley likes to say: "there is no problem in computer science that cannot be resolved by adding one more level of indirection."

Fragmentation is eliminated; the price to pay is some extra hardware plus some software overhead as the OS will have to interact with the MMU

(2) **demand fetch:**

Since process address spaces are now divided into several separate pages, the O.S. can now start executing a process without fetching into main memory its **whole** address space; as long as a specific module is not executed or referenced, it will not occupy any space in main memory

The different parts of the process address space will be fetched **on demand**, that is the first time it *accessed* by the process; if a page needs to be fetched when the main memory is full, the O.S. will expel some other page after its content has been saved to disk.

This approach has two great benefits:

- (a) Main memory is used more efficiently
- (b) Program sizes are not limited anymore by the size of the main memory. Five to ten years ago, many PC's had less than 640K of main memory available. Program authors had therefore to chose between writing small programs that could run on many machines or big programs with more features.

Demand fetch introduces another requirement: the CPU must now be able to restart instructions that have been interrupted in the middle of their execution because some data were missing.

Demand fetch does not work equally well with all processes. For instance, a process that accesses randomly a very large array will never execute efficiently unless all the array is in main memory.

Fortunately most processes tend to access at any time a small portion of their address space and keep accessing the same pages again and again: they are said to obey the *locality principle*.

Even then virtual memory is no substitute for increasing the size of the main memory: even a few faults can considerably slow down a process.

Consider for instance the case of a process having one fault every ten thousand references. The average time T_{access} required to process a reference will be given by:

$$T_{access} = (1 - f)T_{mem} + f T_{disk}$$

where f is the frequency of faults, T_{mem} the main memory access time (say, 15ns) and T_{disk} the disk access time (say, 7.5 ms).

Assuming one page fault occurs *every ten thousand references*, the average access time to the memory will be given by:

$$T_{access} = 15 \text{ ns} + 0.0001 \times 7.5 \text{ ms} = 765 \text{ ns}$$

that is, the process will be slowed down by a factor of 51!

- One can distinguish **two kinds** of virtual memory systems:
 - (1) **paging** systems: they divide the process address space into fixed size **pages**, whose size is always a **power of two** (normally 512 bytes, 1 K, 2 K or 4K),
 - (2) **segmentation** systems: they allow the programmer to specify how each program should be divided; segments normally reflect the program organization, correspond to individual procedures or large data structures and have **different sizes**.

Proponents of segmentation have argued that it guarantees that the right data are brought into main memory and expelled from it because the segments reflect the organization of the program. Dividing processes into fixed-size pages is indeed like cutting through book covers to pick out exactly three or six inches of book width from a bookshelf.

The major argument against segmentation is that managing a main memory subdivided into variable size segments is a nightmare. As a result, very few virtual memory systems have ever used segmentation without combining it with paging.

One could say that segmentation is yet another apparently good idea that has failed to succeed because it was found to have some insuperable drawbacks.

A Reminder:

2^{10} bytes = 1 kilobyte = 1024 bytes

2^{20} bytes = 1 Megabyte = 1,048,576 bytes

2^{30} bytes = 1 Gigabyte $\approx 10^9$ bytes

Also 2^b is represented in binary by a single one followed by b zeroes:

$2^8 = 256 = 10000000_2$

8.2 THE ADDRESS TRANSLATION PROCESS

- Consider the case of a very small process whose size is 3.5 kilobytes to be loaded in the main memory of a virtual memory system whose page frame size is 1 kilobyte. Let us further assume that the page frames allocated to the process are the frames 1, 3, 5 and 7. We would then have the following page map:

<i>Page</i>	→	<i>Frame</i>
0	→	1
1	→	3
2	→	5
3	→	7

Note that page number 3 is half empty because the process is allocated 4 kilobytes of main memory while it only needs 3.5 kilobyte. We refer to these wasted bytes as **internal fragmentation**.

To evaluate the amount of main memory wasted because of internal fragmentation, we begin by assuming that *all process sizes all equally likely*. Hence, the probability of having exactly n bytes of the last page used by the process is given by $1/p$, where p is the page size and the *average* number of bytes used by the process is given by:

$$E(n) = \frac{1}{p} \sum_{i=1}^p \frac{i}{p} = \frac{p}{2}$$

and the average number of bytes lost to internal fragmentation is half a page per process.

The MMU will have to translate the addresses according to the following table:

<i>Virtual Addresses</i>	→	<i>Real Addresses</i>
from 0 to 1,023	→	from 1,204 to 2,047
from 1,204 to 2,407	→	from 3,072 to 4,095
from 2,048 to 3,071	→	from 5,120 to 6,143
from 3,072 to 4,095	→	from 7,168 to 8,192

Observing that the binary representation of 1,024 is 10000000000, that is a one followed by ten zeroes (because $1,024 = 2^{10}$), the previous table can be rewritten as:

<i>Virtual Addresses</i>	<i>Real Addresses</i>
000 00000 00000 to 000 11111 11111	001 00000 00000 to 001 11111 11111
001 00000 00000 to 001 11111 11111	011 00000 00000 to 011 11111 11111
010 00000 00000 to 010 11111 11111	101 00000 00000 to 101 11111 11111
100 00000 00000 to 100 11111 11111	111 00000 00000 to 111 11111 11111

Observe that the address translation process *does not alter the last ten bits of the virtual address*. Hence we can replace the previous table by the much shorter **page table**:

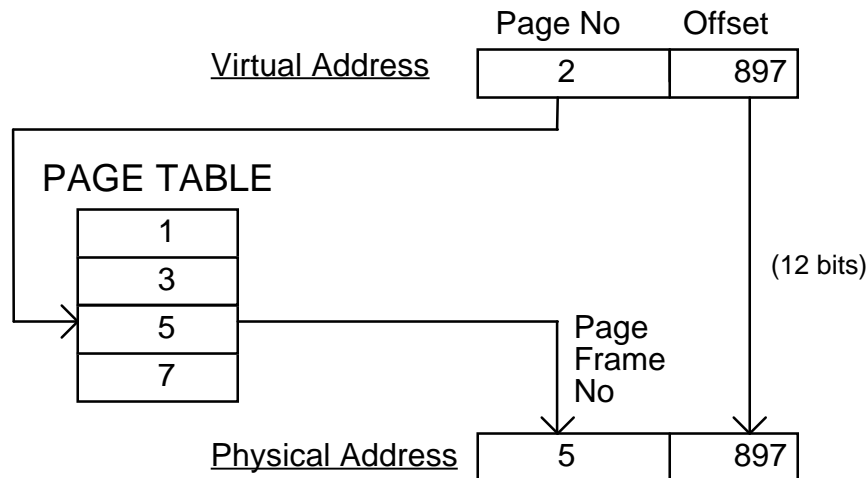
<i>Frame No</i>
<i>1</i>
<i>3</i>
<i>5</i>
<i>7</i>

Note that the page table is indexed by the page number (starting at zero). Hence it does not appear explicitly in the table.

- We can generalize the procedure to systems with pages of arbitrary size **PSIZE**:

```
int Real_Address (int Virtual_Address) {
    int Page_No, Offset;
    Page_No = Virtual_Address / PSIZE;
    Offset = Virtual_Address % PSIZE;
    return Page_Table[Page_No]*PSIZE + Offset;
} /* Real_Address */
```

- In practice, the MMU obtains the page number and the offset by dividing the n bits of the virtual address into two parts:
 - a) the b least significant bits (with $b = \log_2 \text{PSIZE}$) give the **offset**, and
 - b) the $n - b$ most significant bits give the **page number**.



- In practice, most programs will be running with some of their pages not being present in main memory. These pages are said to be *missing*. To identify them, we can either:
 - a) store an **invalid address** in the corresponding page table entry, or
 - b) have a special field identifying these pages: **missing bit** set to **one** or **valid bit** set to **zero**

Assuming that we are using a **missing bit**, the address translation procedure becomes:

```
int Real_Address (int Virtual_Address){
    int Page_No, Offset;
    Page_No = Virtual_Address / P_SIZE;
    Offset = Virtual_Address % P_SIZE;
    if (Page_Table.Valid_Bit[Page_No] == 0)
        System_Request("Page_Fault", ... );
    return Page_Table.Frame_No[Page_No]*P_SIZE + Offset;
} /* Real_Address */
```

Whenever a process attempts to access a missing page, the MMU generates an *interrupt request* that results in the activation of a special part of the kernel called the **page fault handler**. The page fault handler performs two important duties:

- a) it tries to find an empty page frame in main memory; if none can be found, it locates a page that is unlikely to be needed in the next future and expels it from main memory in order to make space for the incoming page
- b) it locates the missing page, fetches it into the main memory, and updates then the corresponding page table entry.

Note that the process attempting to access the missing page will remain in the **waiting state** until these two tasks are completed

- Page table entries also contain several extra fields, among which:
 - a) a **modified bit**, or **dirty bit**, **initially** equal to **zero** and set to **one** whenever the page is **modified**: it indicates that the contents of the page are to be saved on disk when the page will be expelled;
 - b) various **protection bits** indicating whether the page can be **read**, **written** or **executed** by the process;
 - c) an optional **page referenced bit**, or **use bit**, that is set to one whenever the page is accessed; whenever it exists, it is used by the page replacement policy to detect the pages that have been recently accessed and should therefore not be expelled.

8.3 PAGE TABLE REPRESENTATION

Ideally the whole page table should be stored in high-speed registers directly accessible by the MMU. This is not possible because page tables are too big (and cannot even fit in main memory).

The next best solution is to store the most recently used page table entries in a set of associative registers called the **Translation Look aside Buffer** or **TLB**. TLB's can be stored inside the MMU and are organized like a cache memory with the only difference they have much less entries (typically between 64 to 127 entries). A typical TLB entry looks like:

Valid Flag	Page_No	Dirty Bit	Protection Bits	Page_Frame_No
------------	---------	-----------	-----------------	---------------

where the **Valid Flag** indicates whether the TLB entry is valid or not.

Since the TLB contains virtual addresses that are specific to a given program it is normally invalidated at every context switch. This is not as bad as it appears as long as the penalty for not finding a page table entry in the TLB is only not too high. This depends in turn in the way *TLB misses* are handled.

In most virtual memory systems, TLB misses are handled by the MMU, which fetches the missing TLB entry from the page table. The mean time T to process one reference is given by:

$$T = h.T_m + (1-h) 2 T_m$$

where h is the *hit ratio* of the TLB, that is, the probability that the page table entry for the given reference is in the cache and T_m is the main memory access time. Since a hit ratio of 95% already provides an average access time equal to $1.05T_m$, there is no incentive for increasing the cache size and improving its hit ratio.

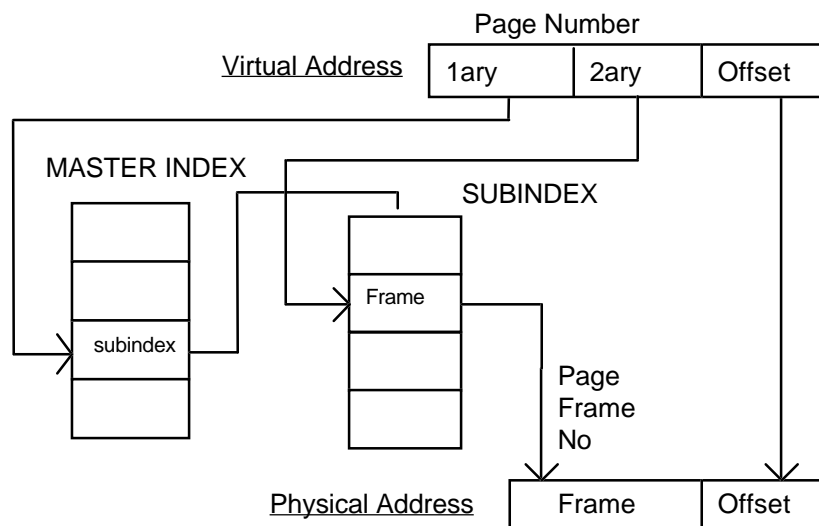
Some RISC architectures let the kernel handle TLB misses. Each TLB miss will cost then *two context switches* and it becomes more important to increase the TLB hit ratio.

- Very few architectures store the entire page table in main memory. UNIX does it but restrict the address space of processes to a few Megabytes so that each page table will only occupy a few kilobytes.

The typical solutions are:

a) **Two-level page tables:** (IBM, Windows NT and others)

The page table is divided into a master index that always remains in main memory and subindexes that can be expelled: one could also say that the page table is paged



(only one of the four subindexes is represented)

This two-level organization is especially suited for a page size of 4 kilobytes and 32 bits virtual addresses. We can then allocate:

- 10 bits of the address for the first level,
- 10 bits for the second level, and
- 12 bits for the offset.

This means that the master index and all the subindexes will have 2^{10} entries. Assuming that each entry occupies four bytes, all indexes will occupy exactly *one page*.

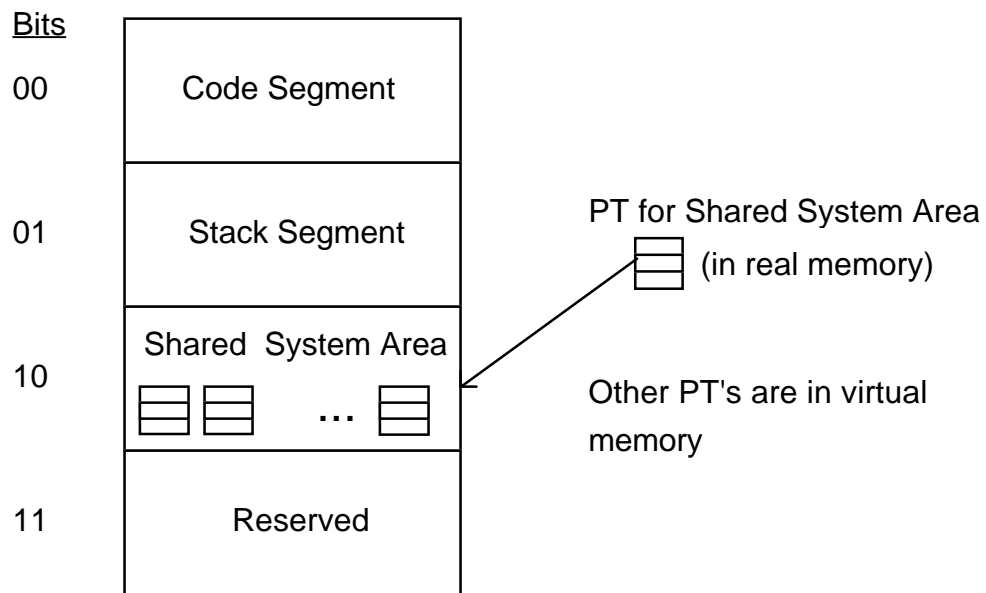
b) **Three level page tables:** (Sun SPARC)

SPARC's page size is 4 kilobytes

- CPU can have up to 4096 different *contexts*, each identifying a specific page table.
- SPARC uses a three level approach with 8 bits for first level, 6 bits for the second and third levels, and 12 bits for the offset.

c) **Storing users' page tables in the system's virtual address space:** (VAX VMS)

The VAX has 32 bit addresses and 512 byte pages. VMS divides the address spaces of user processes into four parts according to the *first* two bits of the address:



Both the code segment and the stack segment are **private** to each process; the shared system area is **shared** among all processes and the kernel.

VMS maintains *one* system-wide page table for the shared system area of *all* user processes. This page table is the *only page table* that is kept in physical memory. Each process has *two* page tables, one for its code segment P0 and one for its data segment P1.

Both page tables are stored in the shared system area, that is, in *virtual memory*.

This organization requires individual pages to be protected against unauthorized accesses. This is one of the reasons why every page has its own protection bits.

d) **Inverted page tables:** (other IBM machines)

An inverted page table has one page table entry per page frame. The page table size is then bounded by the size of the physical memory and not by the size of the process address spaces.

Inverted page tables require (a) a fast mechanism for searching the inverted table at each TLB miss and (b) a conventional page table to locate missing pages on disk (but this page table can be stored in virtual memory).

Inverted page table make shared pages very hard to implement efficiently because every page frame can only belong to the address space of one process

8.4 SELECTING A PAGE SIZE

- Several factors are to be considered when selecting a page size:
 - a) large page sizes reduce the overall number of page faults and the number of page table entries in each page table;
 - b) small page sizes minimize internal fragmentation and require less main memory space.
- Current virtual memory systems tend to have rather large page sizes.

There no point however in increasing page sizes over 4 or 8 kilobytes since it would then be likely that most of the information brought into main memory would then belong to procedures and data structures not currently in use by the process.

8.5 PAGE REPLACEMENT POLICIES

- The page replacement policy is the policy used by the page fault handler to select the page to be expelled from the main memory whenever the main memory is full and a new page needs to be brought in. The page being expelled is often referred to as the *victim*.
- A good page replacement policy should (a) select victims that are not likely to be referenced again in the near future and (b) not result in too much overhead.

Old page replacement policies tended to be more complex than today's policies because main memory was then very expensive and need to be managed very carefully. The current tendency now is towards *simpler* page replacement policies that generate *less overhead*.

- All page replacement policies proposed so far can be grouped into four broad classes:

1. Fixed-Size Local Policies:

These policies allocate to each process a fixed number of page frames in main memory. Whenever a process has used all the page frames it has been allocated, it will have to expel one of its own pages from main memory every time it needs to bring a new page into main memory.

- a) **First In First Out (FIFO):** The FIFO policy expels from main memory the page that has been in main memory for the longest period of time. FIFO is very easy to implement: we can organize the pages frames into a circular queue. It is nevertheless a very bad policy because it does not take into account how the page was used by the process and results in much more page faults than other policies.

If you want to make space on your bookshelves, you should not get rid of your dictionary just because this is the first book you purchased when you came to the University of Houston.

b) Least Recently Used (LRU)

The LRU policy expels the page that has not been used by the program for the longest period of time

It provides an excellent performance because it always selects one the least likely pages to be referenced in the future but is too costly to implement.

Whenever people want to get rid of some of their stuff, they often chose to get rid of the things they have not used for the longest period of time.

2. Global Policies:

Global policies do not allocate fixed numbers of pages frames to each process in main memory. Processes are instead allowed to compete among themselves for page frames and can therefore "*steal*" pages from each other.

a) Global FIFO:

same as Local FIFO but with a single FIFO queue of page frames across the whole main memory

b) Global LRU:

same as Local LRU but with a single LRU list of page frames across the whole main memory

c) **Clock:**

(Multics and Berkeley UNIX)

The Clock policy arranges all page frames into a *circular list*

Each page frame has a **page-referenced** bit or **use** bit that is set to **one** whenever frame is accessed. The page fault handler can reset it to zero.

There is also a pointer (the *hand* of the clock) that can move around the circular list.

Whenever a page must be expelled, the hand moves to the next page in the circular list and inspects its page-referenced bit.

If this bit is equal to zero, the victim is found. Otherwise the page fault handler resets the bit to zero and moves the hand to the next page frame in the circular list. The process is then repeated until a page frame with a page-referenced bit equal to zero is found.

```

Frame *Clock(Frame *Last_Victim){
    Frame *Hand;
    int Not_Found = 1;

    Hand = Last_Victim->Next;
    do {
        if (Hand->PR_Bit == 1) {
            Hand->PR_Bit = 0;
            Hand = Hand->Next;
        }else
            Not_Found = 0;
    } while Not_Found;
    return Hand;
} /* Clock */

```

The policy guarantees that the page being expelled has not been used for one full rotation of the hand of the clock. Hence it provides an acceptable approximation of a Global LRU policy:

*Instead of expelling the **least recently used** page, we expel one of the **not recently used** pages.*

Berkeley UNIX Implementation:

Since the VAX lacks a page-referenced bit, Berkeley UNIX had to simulate it by *software*.

Instead of resetting the page-referenced bit to zero, the Berkeley UNIX implementation of the Clock policy resets the *valid bit* to zero and saves somewhere the *old value* of the valid bit.

When the marked page is accessed again for the first time, the MMU will encounter a valid bit equal to zero and make a system call to activate the page fault handler.

The page fault handler will then check the extra bit and restore the true value of the valid bit so that the page can continue to be accessed. The cost of the operation is *two context switches*.

When main memory sizes started to grow in the mid-eighties, it was found that the hand of the clock was taking too much time to scan the whole main memory. Hence, more recent UNIX implementations have added a second hand that follows the first hand at a fixed angle. Now the first hand clears valid bits and the second hand expels the pages that have not been accessed by the process after their valid bit had been cleared by the first hand.

d) Mach Policy:

Mach divides its main memory into a *pool of page frames* shared by all processes and one *global queue* from which pages can be reclaimed.

The pool of page frames is managed according to a Global FIFO policy but pages expelled from the pool by the FIFO policy are given what essentially amounts to a *second chance*. Instead of being immediately expelled from the main memory, they go at the end of the global queue and their *valid* bit is reset to *zero*.

Whenever a page fault occurs, the page fault handler looks first at the global queue to see if the missing page is cannot be found there. If this is the case, the page is returned to the FIFO pool. If another page needs to be expelled from the FIFO pool to make space for the

returning page it will go at the end of the Global queue and another page put at the end of the global queue.

There is an interesting trade-off in selecting the respective sizes of the FIFO pool and the global queue:

Since FIFO is a poor page replacement policy, many of the pages arriving at the end of the global queue after being expelled from the FIFO pool will be pages that are currently used by the process and should remain in main memory. We need therefore to allocate enough pages to the global queue to give to these pages fair chance to be *reclaimed* before leaving the main memory. Hence a large global queue will result in less page faults.

Since each reclaim requires an interrupt (and *two context switches*), we need also however to allocate enough pages to the FIFO pool to avoid expelling pages too quickly and reclaiming them too frequently.

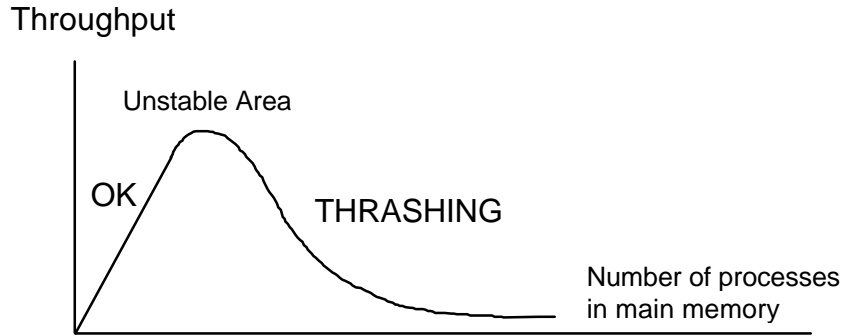
3. Variable Size Local Policies:

These policies adjust dynamically the number of pages allocated to each process to adjust to changes in its page referencing behavior.

Whenever the policy cannot allocate to a process what it believes to be its necessary number of page frames, it notifies the scheduler that the memory is overutilized and asks it to suspend one or more processes.

As a result, variable size local policies guarantee that the main memory will never be overutilized. They are said to prevent **thrashing**, that is

the situation that happens when there are too many processes in main memory, these processes are always stealing pages from each other and most processes are in the waiting state because the paging device cannot keep up with all page requests.



Working Set Policy:

The Working Set (WS) policy (P. Denning, 1967) keeps in main memory all pages that have been referenced during the last τ referenced issued by the process. Conversely, all pages that were **not** referenced during this time interval are immediately expelled.

The set of pages kept in main memory by the WS policy is said to be the *working set* of the process because Denning hypothesized that they were the pages the process needed to run efficiently.

The parameter τ is called the *window size* of the policy. It normally varies between 20,000 and 100,000 references. Selecting the proper window size for a process is not a big problem because there is generally a wide range of window sizes (say, between 30,000 and 75,000 references) where the window size does not significantly affect the size of the working set of the process.

The major limitation of the WS policy still remains the high very cost of the hardware required to detect which pages belong to the working set of a process and which pages should be expelled.

Several simplified versions of Denning's Working Set policy have been proposed. The most popular, *Sampled Working Sets*, expelled pages at fixed time intervals rather than after each reference. All these policies have lost by now any practical interest because of their high overhead and the ever diminishing cost of main memory.

4. Hybrid Policies:

There is only one instance of such policies, namely the VMS page replacement policy: it is said to be a *hybrid* policy because it is neither a global nor a local policy:

a) VMS Policy:

VMS allocates to each process a fixed-size partition that it manages using a FIFO policy.

Like in the Mach policy, which it predates by several years, the pages expelled by the FIFO policy are put at the end of a large global queue from which they can be reclaimed.

The existence of separate partitions for each individual process makes the VMS page replacement policy harder to tune than the Mach policy. There is always the risk that one process will not be allocated enough main memory and that its pages will have to be constantly reclaimed from the global queue, which would result in a high number of context switches

The designers of VMS decided to give nevertheless a fixed partition to each process because one of their major objectives was to provide efficient support for real-time processes. Having a fixed partition for each process allowed them to allocate to each real-time process enough page frames to keep the whole address space of the process in main memory and guarantee that none of its pages would ever be expelled.

More recent releases of VMS also allow the sizes of the individual process partitions to be adjusted by the OS whenever it detects too much or too little exchanges of pages between the process partition and the global queue.

b) Windows NT Policy:

Windows NT uses a variant of the VMS page replacement policy:

Each process is allocated a *minimum* and *maximum working set size* that respectively represent the minimum and maximum sizes of its private partition

Processes can change these parameters at run time by calling a process object service but the security system enforces an absolute minimum and an absolute maximum for each user-mode process. Processes start with their minimum allocation of frames; if the memory is not overly full, the VM manager allows the process to grow up to its maximum allocation; when the physical memory runs low, the VM manager starts trimming the working sets of processes; processes that exhibit a lot a paging can regain some of their lost frames if enough frames remain available

8.6 KEY CONCEPTS

Be sure that you understand and can explain the following concepts:

Clock replacement policy

dirty bit

FIFO replacement policy

global replacment policies

local replacment policies

LRU replacement policy

missing bit

offset

page fault

page number

page referenced bit

page table

page table entry

paging

real address

translation look-aside buffer

valid bit

virtual address

window of a working set

working set of a process