Paging

A Memory Management Technique

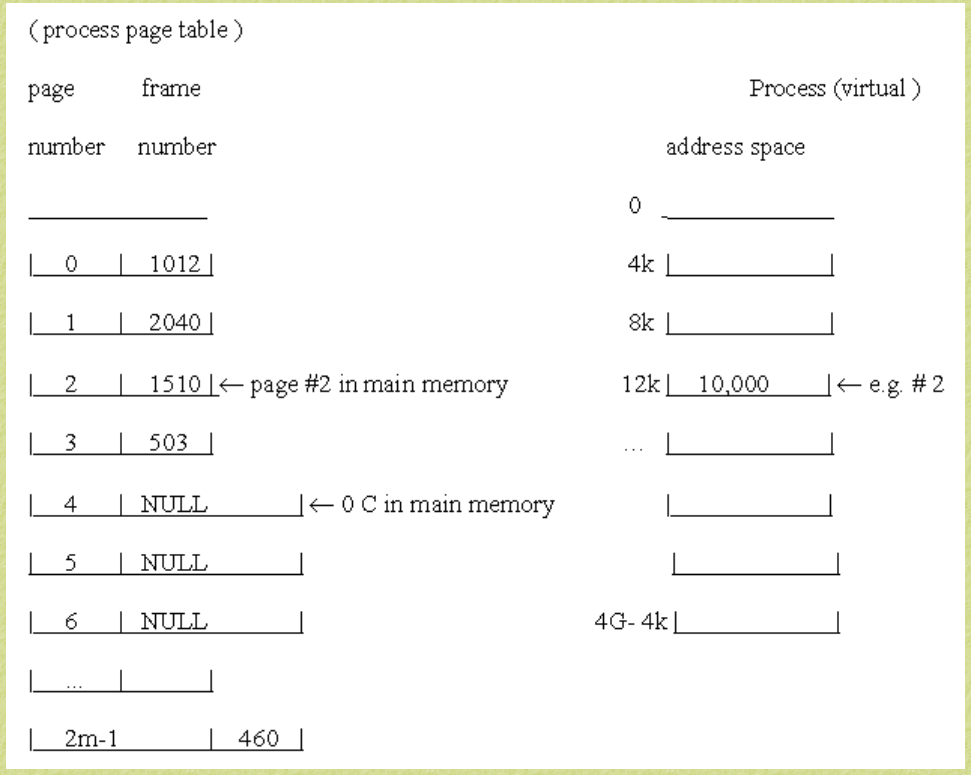
Page : one of numerous equally sized chunks of memory

Page Table: stores where in memory each page is

Main Memory : divided into **page frames**, a space large enough to hold one page of data
(e.g. 4k)

Swap Space :

- divided into pages
- assume the complete process is first loaded into the swap space
- more pages go back and forth between swap space and main memory
- when a program is loaded, put it into the swap space
- memory and page size is always a power of 2



Address Translation with Paging

- for each process there is an address **A**

- when doing computations, always truncate

1. page number = A / page_size

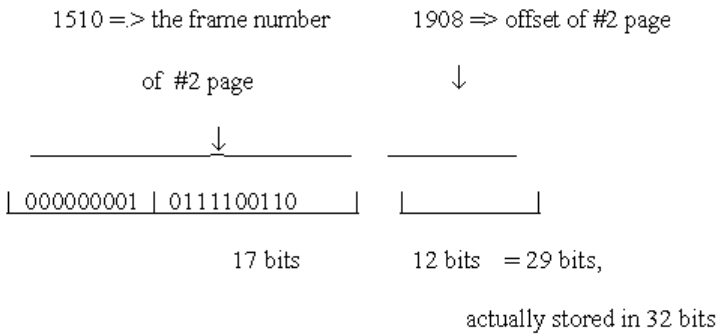
- this is the page number within the process address space
- e.g. address in the process, A = 10,000
- page size = 4k
- page number = 10000 / 4k = 10,000 / 4096 = 2.xxx = truncate to 2
- this calculation is done quickly on the computer since the page size is power of 2, e.g., 4k = 2^12
- to determine the page number, shift the address right by 12 bits
- if the virtual address size = 32 bits, then since the page size is 4k = 2^12, then the the last 12 bits give the page offset, and the first 32- 12 = 20 bits give the page number

```
Example Address (in binary):
---------------------------------------
| 10101010101010101010 | 101010101010 |
---------------------------------------
```

2. offset = A mod page_size

- this is the distance from the beginning of the page
- e.g. address in the process, A = 10,000
- page size = 4k
- page offset = 10000 mod 4k = 10,000 mod 4096 = 1908
- this calculation is done quickly on the computer since the page size is power of 2, e.g., 4k = 2^12
- to determine the page offset, mask out all but the rightmost 12 bits

first convert the number to base 2



Now calculate the physical address:

To compute the physical address:

1. look up the page number in the page table and obtain the frame number
2. to create the physical address, frame = 17 bits; offset = 12 bits; then
   512 = 29
   1m = 220 => 0 - (229-1)
   if main memory is 512 k, then the physical address is 29 bits

Effective Memory Access Time
The percentage of times that a page number is found in the associative registers is called the *hit ratio*. An 80-percent hit ratio means that we find the desired page number in the associative registers 80 percents of the time. If it takes 20 nanoseconds to search the associative registers, and 100 nanoseconds to access memory, then a mapped memory access takes 120 nanoseconds when the page number is in the associative registers. If we fail to find the page number in the associative registers (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds), and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds. To find the *effective access time,* we must weigh each case by it probability:

```
effective access time = 0.80 * 120 + 0.20 * 220

                      = 140 nanoseconds.
```

In this example, we suffer a 40-percent slowdown in memory access time (from 100 to 140 nanoseconds).
For a 98-percent hit ratio, we have

```
effective access time = 0.98 * 120 + 0.02 * 220
                      = 122 nanoseconds
```

The increased hit rate produces only a 22-percent slowdown in memory access time.
The hit ratio is clearly related to the number of associative registers. With the number of associative registers ranging between 16 and 512, a hit ratio of 80 to 98 percent can be obtained.

To do page table look-ups quickly:

- a translation lookaside buffer (TLB) is used
- associative cache, where cache is the fastest memory available and associative looks at all the entries at the same time
- expensive
- relatively small
- only a few page number/frame number pairs can be stored at once
- stores the entries from the most recently used pages
- updated each time a page fault occurs

Page Fault

- when a process tries to access an address whose page is not currently in memory
- process must be suspended, process leaves the processor and the ready list, status is now ``waiting for main memory''

Address: can be an address of an instruction, or of data (heap, stack, static, variables)

Paging Operations

- fetch " page in " => bring a page into main memory

1. **Fetch policy**

a) demand fetching = demand paging

- fetch a page when it is referenced but not stored in main memory
- causes a page fault whenever a new page is required
- disadvantage - cold start fault: many page faults when a process is just starting
- advantage - no unnecessary pages are ever fetched

b) anticipatory fetching = prepaging

- guess which pages will be required soon and fetch them before they are referenced

There are three variations:

i) working set prepaging

- make sure to fetch all pages in a process' working set before restarting
- **working set**: a set of pages accessed in the last 'w' working time units
  where w = the window size based on temporal locality
- intent is that all pages required soon are in main memory when the process starts

ii) clustering prepaging

- when a page is fetched, also fetch the next page(s) in the process address space
- cost of fetching *n* consecutive pages from disk is less than fetching *n* non-consecutive pages
- common variant is to fetch pairs of pages whenever one is referenced
- the extra page may be before or after - used in Windows NT, WIndows 2000

iii) advised prepaging

- programmer/compiler adds hints to the OS about what pages will be needed soon
- e.g. hint: & myfunction
- problem: cannot trust programmers; they will hint that all their pages are important

2. ***Placement Policy***

- determine where to put the page that has been fetched
- easy for paging, just use any free page frame

3. ***Replacement Policy***

- determines which page should be removed from main memory (when a page must be fetched)
- want to find the least useful page in main memory
- candidates, in order of preference:

1. page of a terminated process

2. page of a long blocked process

- danger: do not want to swap out pages from a process that is trying to bring pages into main memory
- thrashing:
  - where the system is preoccupied with moving pages in and out of memory
  - feature: the disk can be very busy while the CPU is nearly idle
  - one cure is to reduce the number of processes in main memory
    i.e., reduce the level of multiprogramming

3. take a page from a ready process that has not been referenced for a long time

4. take a page that has not been modified since it was swapped in

- saves copying to the swap space

5. take a page that has been referenced recently by a ready process

- => performance will downgrade

Local versus Glogal Page Replacement:

- **Local page replacement**: when a process pages "against itself" and removes some of its own pages
- **Global page replacement**: when pages from all processes are considered
- Example: Windows NT/XP/Vista use both a local page replacement method (based on FIFO) and a global page replacement method (based on PFF)

 Table of Contents