Virtual Memory - programmer views memory as large address space without concerns about the amount of physical memory or memory management. (What do the terms 32-bit (or 64-bit) operating system or overlays mean?)

Benefits:
1) programs can be bigger than physical memory size since only a portion of them may actually be in physical memory.
2) higher degree of multiprogramming is possible since only portions of programs are in memory

Operating System’s goals with hardware support are to make virtual memory efficient and transparent to the user.
Demand paging is a common way for OSs to implement virtual memory. Demand paging ("lazy pager") only brings a page into physical memory when it is needed. A "Valid bit" is used in a page table entry to indicate if the page is in memory or only on disk.

A page fault occurs when the CPU generates a logical address for a page that is not in physical memory. The MMU will cause a page-fault trap (interrupt) to the OS.

Steps for OS’s page-fault trap handler:
1) Check page table to see if the page is valid (exists in logical address space). If it is invalid, terminate the process; otherwise continue.

2) Find a free frame in physical memory (take one from the free-frame list or replace a page currently in memory).

3) Schedule a disk read operation to bring the page into the free page frame. (We might first need to schedule a previous disk write operation to update the virtual memory copy of a “dirty” page that we are replacing.)

4) Since the disk operations are soooooo slooooooow, the OS would context switch to another ready process selected from the ready queue.

5) After the disk (a DMA device) reads the page into memory, it involves an I/O completion interrupt. The OS will then update the PCB and page table for the process to indicate that the page in now in memory and the process is ready to run.

6) When the process is selected by the short-term scheduler to run, it repeats the instruction that caused the page fault. The memory reference that caused the page fault will now succeed.

Lecture 30- 2
Performance of Demand Paging
To achieve acceptable performance degradation (5-10%) of our virtual memory, we must have a very low page fault rate (probability that a page fault will occur on a memory reference).

When does a CPU perform a memory reference?
1) Fetch instructions into CPU to be executed
2) Fetch operands used in an instruction (load and store instructions on RISC machines)

Example:
Let p be the page fault rate, and ma be the memory-access time.
Assume that p = 0.02, ma = 50 ns and the time to perform a page fault is 12,200,000 ns (12.2 ms).

\[
\text{effective memory access time} = (1 - p) \times \text{main memory access time} + p \times \text{page fault time}
\]

\[
= (1 - 0.02) \times 50\text{ns} + 0.02 \times 12,200,000\text{ns}
\]

\[
= 0.98 \times 50\text{ns} + 0.02 \times 12,200,000\text{ns}
\]

\[
= 244,049\text{ns}
\]

The program would appear to run very slowly!!!

If we only want say 10% slow down of our memory, then the page fault rate must be much better!

\[
55 = (1 - p) \times 50\text{ns} + p \times 12,200,000\text{ns}
\]

\[
55 = 50 - 50p + 12,200,000p
\]

\[
p = 0.0000004 \text{ or 1 page fault in 2,439,990 references}
\]
Fortunately, programs exhibit **locality of reference** that helps achieve low page-fault rates

1) **spatial locality** - if a (logical) memory address is referenced, nearby memory addresses will tend to be referenced soon.

2) **temporal locality** - if a memory address is referenced, it will tend to be referenced again soon.

Typical Flow of Control in a Program

```
for i := 0 to 2n do
    Data references within the loop
    i:
end for
```

Possible reference string: 5, (5, 101, 5, 5, 101, 5, 5, 100, 5, 103, 6, 6, 6), 6, 6, 6, 6, 7, 7, 7, 7, (7, 7, 103, 7, 101, 7, 7, 7, 101, 7, 8, 8, 8, 100, 8), 8, 8, ...

Terms:
- **reference string** - the sequence of page #’s that a process references
- **locality** - the set of pages that are actively used together
- **working set** - the set of all pages accessed in the current locality of reference
Storage of the Page Table Issues

1) Where is it located?
If it is in memory, then each memory reference in the program, results in two memory accesses; one for the page table entry, and another to perform the desired memory access.

Solution: TLB (Translation-lookaside Buffer) - small, fully-associative cache to hold PT entries
Ideally, when the CPU generates a memory reference, the PT entry is found in the TLB, the page is in memory, and the block with the page is in the cache, so NO memory accesses are needed. However, each CPU memory reference involves two cache lookups and these cache lookups must be done sequentially, i.e., first check TLB to get physical frame # used to build the physical address, then use the physical address to check the tag of the L1 cache.

Alternatively, the L1 cache can contain virtual addresses (called a virtual cache). This allows the TLB and cache access to be done in parallel. If the cache hits, the result of the TLB is not used. If the cache misses, then the address translation is under way and used by the L2 cache.

2) Ways to handle large page tables:
Page table for each process can be large e.g., 32-bit address, 4 KB (212 bytes) pages, byte-addressable memory, 4 byte PT entry

1 M (220) of page table entries, or 4MB for the whole page table with 4 byte page table entries

Solutions:
a) two-level page table - the first level (the "directory") acts as an index into the page table which is scattered across several pages. Consider a 32-bit example with 4KB pages and 4 byte page table entries.
b) inverted page table - use a hash table of what's actually in the physical memory frames to reduce the size of the necessary data structures

Process ID  Virtual Address

Hash Function

Virtual Address

Inverted Page Table

<table>
<thead>
<tr>
<th>PID</th>
<th>Page #</th>
<th>PT Entry</th>
<th>Chain</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Page #</th>
<th>Offset</th>
</tr>
</thead>
</table>

Physical Address

Frame Offset
**Design issues for Paging Systems**

Conflicting Goals:
1) Want as many (partial) processes in memory (high degree of multiprogramming) as possible so we have better CPU & I/O utilization ⇒ allocate as few page frames as possible to each process

2) Want as low of page-fault rate as possible ⇒ allocate enough page frames to hold all of a process’ current working set (which is dynamic as a process changes locality)

![Diagram showing CPU utilization vs. degree of multiprogramming]

*Thrashing* occurs when processes spend more time in page fault wait than doing useful work

Operating systems need to have
1) **frame-allocation algorithm** to decide how many frames to allocate to each process
2) **page-replacement algorithm** to select a page to be removed from memory in order to free up a page frame on a page fault