Operating Systems (UE18CS302)
Unit 3

Aronya Baksy

August 2020

1 Introduction to memory management

1.1 Memory Protection

- All system memory is split into a hierarchy, with the differentiation being on the basis of speed, and capacity.
- The **CPU Cache** is closest to the CPU, has the greatest speed, but the smallest capacity. It is used to prevent CPU stalls while reading/writing to frequently used locations in the main memory.
- The presence of one (or many) fast caches in the CPU makes sure that the CPU does not have to wait unduly long time for memory access, as memory access is way slower than other CPU operations and can cause bottlenecks.
- Aside from speed, the concurrency and integrity of memory values stored must be maintained. User processes accessing memory must be protected from one another, and the OS kernel itself must be protected from all user processes.
- The range of memory addresses that a single process is legally allowed to access is defined using a **base register** and a **limit register**.
- The base register holds the smallest memory address that the process can legally access, and the limit register holds the size of the range. The process can access every address in the range [base, base+limit] (both inclusive).
- Every address generated by CPU in user mode is compared with this range, and if it falls outside the range then it leads to a **trap state** to the OS, that flags it as a memory access violation (fatal error).

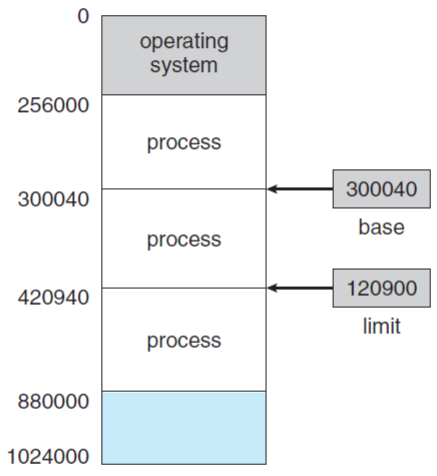

Figure 1: Base and Limit Register

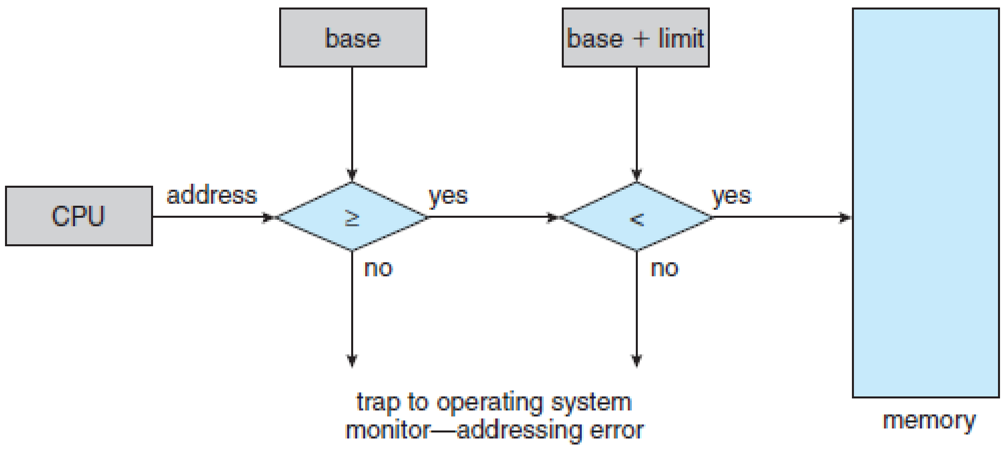

Figure 2: Address Protection using base and limit registers

1.2 Address Binding

- Addresses in programs are represented *symbolically* (using variable names and data types).
- The compiler **binds** these symbolic addresses to *logical addresses* which are also known as relocatable addresses (eg: offset of 14 bytes from the beginning of this module).
- The linkage editor or the loader **binds** the logical addresses to *physical addresses* (eg: offset of 14 bytes from module start gets bound to 70014).
- Each of these bindings are a mapping between 2 address spaces.
- The process of binding instructions and data to actual memory addresses can be done at:
  - **Compile Time:** If the starting location of the process is known at compile time, then absolute addresses can be generated at compile time. If the starting address changes, then the program will have to be recompiled.
  - **Load Time:** If the start address of the program is not known at compile time, then the compiler must generate relocatable (logical) addresses. These logical addresses can be bound to physical (ie. absolute) addresses when the program is loaded into the memory. If the starting address changes, then the program only has to be reloaded into the new address.
  - **Run Time** (Execution Time): If the process may be moved from one memory segment to another during its execution, then the binding must take place only at run time. This is the most commonly used method in modern OSes as it allows more efficient memory usage.

1.3 Virtual and Physical Memory

- Addresses generated by CPU are referred to as **logical addresses**, while the address seen by the actual memory hardware (which is loaded into the **Memory Access Register** in the memory) is called the **physical address**.
- In compile-time and load-time binding, the logical and physical addresses generated are identical. In run-time binding, the logical and physical address are different.
- In compile time binding, the CPU generates the physical memory addresses directly, hence they are identical to the physical addresses in the memory.
- In load-time binding, the relocatable addresses generated by the compiler are translated into physical addresses by the loader. Hence they are identical after loading to the physical addresses.
- The run-time translation between logical (aka **virtual**) addresses and physical addresses is done by a hardware unit called the Memory Management Unit (MMU).

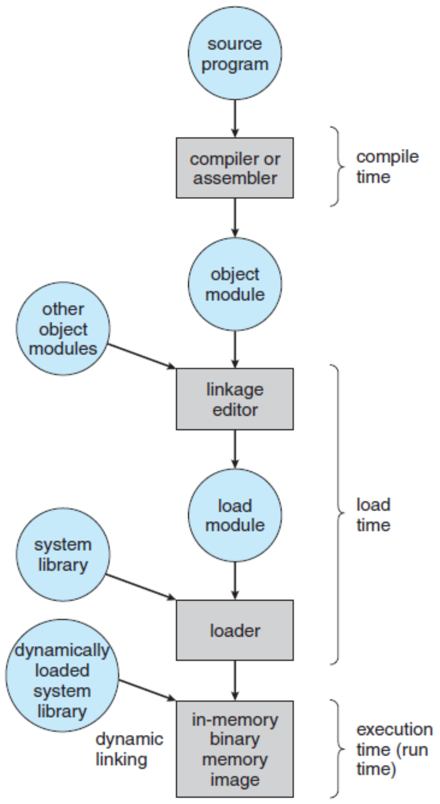

Figure 3: User Program Processing

- All memory related computations (comparison with other addresses, arithmetic operations) are done on the virtual address, and the relocation (ie. translation between virtual and physical) is done only when a load or store is required.
- The virtual address space has a range of (0, max) while the physical address space has a range of (R, R+max) where the base (ie. relocation) register holds the value of R.

1.4 Dynamic Loading

- When an entire program sitting in the memory but some routines are only needed for a short time, it is inefficient in terms of memory usage.
- In dynamic loading, the routines are loaded from storage into the main memory only when they are called.
- When the routine is loaded, the calling program's address table is updated and the relocatable addresses are translated into physical addresses,
- Dynamic loading does not need support from the OS, it is the user programmer's responsibility to implement it. But libraries that implement some dynamic loading functionalities may be provided by the OS.

1.5 Dynamic Linking and Shared Libraries

- In dynamic linking, a system library that needs to be included in a user program is loaded only once, and all programs that use that library execute only one copy of it.
- Without dynamic linking, each user program would need its own copy of that system library which wastes memory space and disk space.
- The user program contains a **stub** which is a reference to the location of the dynamic library in the memory.

- When the stub is executed, it checks if the libary is already in memory. If not, it is loaded to memory, and if it is, then the stub replaces itself with the address of the routine and executes the routine.
- If that library routine is referenced again, then it can be directly executed without any overheads of dynamic linking.
- Shared libraries are a feature that enable several different versions of the same library to exist in memory, and the user program can choose which version to use.
- Using shared libraries ensures that the majority of programs will not be affected by any change in versions of a library.
- Dynamic linking and shared libraries require OS support, as only the OS can inspect the memory space of another process to see if a library exists in memory, and allow multiple processes to access that memory address.

2 Swapping

- Swapping is a mechanism that allows the total physical address space of all the processes in the system to exceed the main memory's address space, thus increasing the degree of multiprogramming.
- Swapping involves processes being moved between main memory and a **backing store**, then being brought back into main memory for continued execution.

2.1 Standard Swapping

- The backing store in this case is a fast disk, that is large enough to store memory images for all users.
- A **ready queue** is used to store the processes that are in disk or memory that are ready to execute.
- When the scheduler decides to run a process, it calls the dispatcher.
- The **dispatcher** checks if the next process to be executed lies in memory or not. If not, and there is not enough main memory space for the new process, the dispatcher swaps a process out of main memory and brings in the next process from the ready queue into the main memory.
- Knowing exactly the amount of memory that process is using (instead of what amount it *might* use) is useful in reducing swap times (because swap time is proportional to the amount of memory being swapped).
- Hence, a process that has changing memory requirements must notify the OS of these requirements using the system calls.
- Only idle processes must be swapped, with special care on processes that have pending I/O.
- Solutions to swapping processes with pending I/O are:
  - Don't swap out such processes
  - Execute I/O ops only into the OS buffers. This means that there are 2 copies of the process (one in the OS buffer, one in the main memory). The transfer between these 2 happens only when process is swapped in again. This is known as **double buffering**.
- Standard swapping is not provided on modern OSes, but its variations are in use
  - On UNIX, Linux and Windows, swapping is disabled by default until the free capacity in main memory falls below a certain amount. Swapping again stops when free capacity increases above another threshold.
  - Swap only parts of processes instead of entire processes, to reduce swap time.

2.2 Swapping on Mobile Systems

- Swapping is generally avoided in mobile systems as memory space is a more precious resource (because it is much rarer) than on desktop systems.
- Other reasons to avoid swapping are the failure of flash memory (used in disk) beyond a certain number of writes, and the low throughput between flash and main memory.
- **iOS** asks applications to relinquish memory. Read only data is removed from main memory and reloaded later if needed. Modified data (like stack) is not removed from main memory, but any apps that fail to give up sufficient memory are terminated by the OS.
- **Android** follows a similar strategy to iOS, but the current application state is written to flash memory before it is terminated (for fast restart).

3 Contiguous Memory Allocation

- Memory Allocation schemes aim to allocate memory to multiple user processes in memory at the same time in the most efficient manner.
- In contiguous memory allocation, all the processes are assigned contiguous blocks of memory.

3.1 Allocation in CMA

- In **fixed partitioning**, the entire memory is divided into fixed size partitions where one process only can occupy one partition.
- The degree of multiprogramming is decided by the number of partitions
- Once a process terminates, that partition becomes available for another process, and the scheduler fills that partition from teh ready queue.
- In **variable partitioning**, the OS maintains a table of free and occupied partitions (available blocks of memory are referred to as *holes*).
- The scheduler takes into account the memory requirement of the process while allocating memory. When the memory is allocated to the process, it runs until it terminates, and then the memory is freed.
- At any point in time, there is an input queue of processes, along with a list of free block sizes.
- Memory is allocated until there is no hole large enough for the process at the front of the queue.
- In this situation, the OS may wait until such a hole is available, or the OS may skip down the queue until a process is found that can fit in the currently available holes.
- The scheduler needs to find holes that satisfy the memory requirement of the process. These holes may not be in any order, rather they are scattered throughout the memory randomly.
- Holes can be managed in the following way:
  - If 2 adjacent holes end up becoming free, then they are merged into one larger hole.
  - If a process ends up in a hole that is too large for it, the hole is split into two parts (one for the process, another free).

  The table of free memory (ie. holes) and allocated memory needs to be updated any time this happens.
- The following strategies are used to select a hole from the available list of holes
  - **First-Fit**: Select the first hole from the list of holes that is large enough for the incoming process.
  - **Best-Fit**: Select the smallest hole that fits the incoming process. This requires searching the entire list of holes every time
  - **Worst-Fit**: Allocate the largest hole that is currently in the list, to the incoming process. This also requires searching the entire list of holes every time.

3.2 Fragmentation

- **External Fragmentation** occurs when there are enough free holes to satisfy a process' requirements, but they are not contiguous hence the process must wait.
- In the worst case, external fragmentation may result in a free hole between every two occupied blocks.
- Statistical analysis of the first-fit allocation shows that for every N allocated blocks, $\frac{N}{2}$ blocks are lost to fragmentation. (loss of 33.33%).
- **Internal Fragmentation** occurs within partitions, when a process takes up memory that is smaller than the available hole size.
- eg: Let the partition size be 18464 bytes, and the process requests a block of size 18462 bytes. The free hole of size 2 bytes is inefficient to maintain (as its maintenance needs more overhead than 2 bytes).
- The solution to the above is to allocate memory in fixed blocks, but this can result in slightly larger allocation than is needed. The difference between the used memory and the allocated block size is the extent of internal fragmentation.
- A solution to external fragmentation is called *compaction*, where all the occupied memory blocks are compacted (ie. combined) together, and the holes all combine to become one large free hole.
- Compaction will not work if address binding is done at compile time or linking time. But if address binding is done at run time, then compaction just means that the base register value needs to be changed once.
- Another solution to fragmentation is to allow non-contiguous process memory spaces (ie. allow processes to take up non-contiguous memory blocks). The solutions that implement this are Segmentation and Paging.

4 Segmentation

- The programmer's model of memory consists of blocks of variable sizes, each holding one specific functionality.
- Each *segment* here defines either a method, a shared library, or some data structure (like stack, heap, objects etc.) with a base size and a length.
- Individual elements inside the segment are referred to by their offset from the base address of that segment.
- Segmentation is a memory-management scheme that supports this programmer's model of memory.
- The logical address space is a collection of segments, each segment having a *segment number* and a *segment length.*
- Every logical address consists of a tuple of ⟨ segment_number, offset ⟩ where the offset is the offset from the base address of the segment number.

4.1 Hardware Support for Segmentation

- An address in the physical address space can be expressed from the programmer's point of view as the combination of the **segment number** s and the **offset** within that segment d.
- The **segment table** maps the 2-dimensional address ⟨ s, d ⟩ to a single physical address.
- The segment number s is used as an index in the segment table.
- The CPU generated value of d is compared with the value of d from the segment table, and if the CPU generated value of d is larger than the d value from segment table, the system is sent into a trap state.

- If the CPU generated d value is less than or equal to the d value from the segment table, it is combined with the base address s from the segment table to generate the physical address.

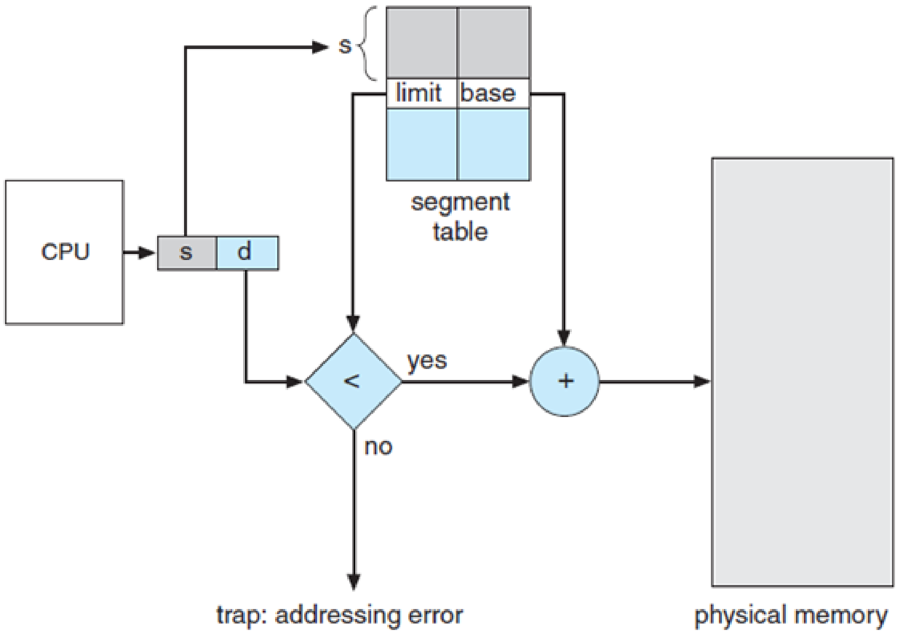

Figure 4: Hardware for Segmentation

5 Paging

- Paging is another memory management technique that allows non-contiguous memory allocation, that avoid external fragmentation and the need for compaction (unlike segmentation).
- Physical memory is split into frames, and virtual memory is split into pages. Pages and frames have the same size.
- The logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 2^{64} bytes of physical memory.
- The CPU generated virtual address consists of a **page number** p and a **page offset** d.
- The offset is carried over into the physical address as is. The page number is translated into the frame number f by the **page table**. The page number is used to index in the page table, and the frame number is taken.
- The combination of the frame number and the previously computed offset create a physical address.
- Page and frame sizes are fixed by the OS as a power of 2.
- If logical address space has size 2^m, and page size is 2^n, then the first $m - n$ bits of the logical address are the page number, and the last n bits are the page offset.
- Paging prevents external fragmentation but internal fragmentation still takes place in paging.
- Large page sizes cause larger extent of internal fragmentation, while smaller page sizes can cause greater overhead in the page table (as more entries will be needed which themselves take up memory).
- Average page size in modern systems is 4-8 kB.

- The **frame table** contains information about a frame of physical memory. Each physical page frame has one frame table entry.
- Each frame table entry stores whether the frame is free or allocated, and if allocated, the pid of the process that is occupying that frame.
- The OS maintains a copy of the frame table for each active process. This copy is used when the user wants to make a system call referring to a particular address (this logical address must be translated into physical address using this copy of the page table).
- The process' copy of the page table is also referred to when the CPU dispatcher edits the hardware page table when the process is allocated the CPU. Therefore paging increases context switching time.

5.1 Hardware Support for Paging

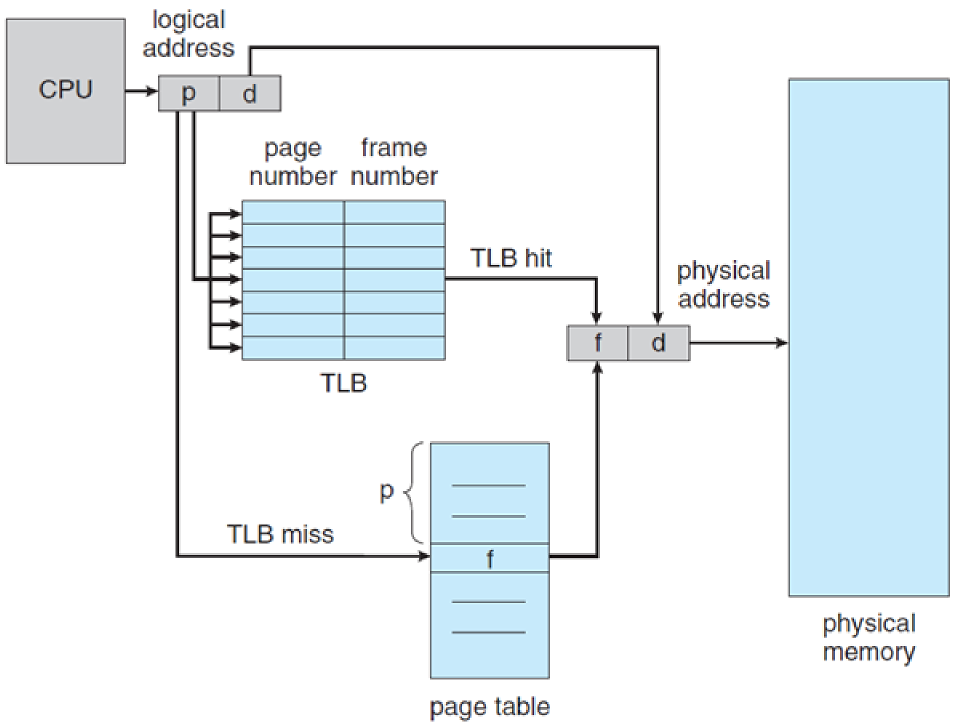

Figure 5: Paging Hardware

- Page table can be implemented as a set of high speed CPU registers with fast logic for address translation. The CPU dispatcher loads and reloads the values in these registers. Instructions to modify the page table register values are privileged instructions.
- In modern systems with massive page tables (1 million+ entries), page tables are stored in main memory, and a **page table base register**(PTBR) denotes the start address of the page table.
- The problem is that for a single address lookup, first the appropriate page table entry must be accessed (using PTBR + offset) and then the frame number in that page table entry must be accessed. Thus, for each memory access, two actual memory accesses are needed, which doubles the memory access time.
- The solution to the above is the use of a **translation look-aside buffer**(TLB), which acts like a high-speed cache for the page table.
- The TLB entry consists of a key and a value. While searching, a key is simultaneously compared with all the keys in the TLB.

- TLB searching is fast as it is part of the instruction pipeline. This means that the TLB must be of small size (between 32 and 1024 entries).
- Some architectures use separate TLB for instruction and data addresses.
- In case of a TLB miss, the address must be looked up in the memory (done either automatically by CPU or by an interrupt to the OS). The new page table entry must be placed into the TLB as well.
- If the TLB is already full, then entries can be replaced using LRU or FIFO or random replacement. CPU or OS can participate in replacement (system dependent).
- Some entries are **wired down** in the TLB, meaning they cannot be replaced. Typically kernel code addresses are wired down in the TLB.
- In some TLBs, an ASID (**address space ID**) is stored in each entry. When the TLB tries to convert virtual address to a physical address, the ASID of the virtual page and that in the TLB entry are matched, and in case of a mismatch it is treated as a TLB miss.
- The ASID offers address space protection, and allows the TLB to contain translation info for multiple processes at once.
- If TLB does not support ASIDs, then everytime a new process is loaded onto the CPU the TLB must be entirely flushed as it might contain invalid translation info (valid virtual addresses but invalid physical addresses).
- The hit rate is defined as the probability of a TLB hit, ie. the chance that a particular request for a frame number will be satisfied by the TLB and not have to go all the way to the main memory page table.
- The effective access time is defined as

  $$EAT = \textit{hit rate} \times a_h + (1 - \textit{hit rate}) \times a_m$$

  where a_h and a_m are the memory access times in case of a hit and a miss.
- Assuming that TLB lookup requires one memory access, we can consider that a_h is equal to the average memory access time (AMAT). And in case of a miss, two memory accesses are needed (one in the page table, another one for the actual memory byte), hence a_m will be twice of the AMAT.
- AMAT calculation is more complicated in case of multiple levels of TLBs (eg: Intel core i7 with 128 instruction entries and 64 data entries in Level 1 TLB, and 512 entries in Level 2 TLB).

5.2 Memory Protection

- Protection is implemented in page tables using separate bits associated with each page table entry. These bits are stored within the page table itself.
- These bits can be used to provide read-only, write-only and execute-only permissions, as well as various combinations of these. Illegal operations lead to a trap state within the OS.
- The **valid-invalid bit** is also attached to all the page table entries.
- If the bit is set to *valid*, then that page is a part of that process' logical address space, hence it can be accessed.
- If the bit is set to *invalid*, that page is not a part of that process' logical address space, and any attempt to access it by that process will lead to a trap.
- The valid-invalid bit can cause problems if there is internal fragmentation, as a page table entry can be marked as valid even though the process only uses a few addresses within it and not the entire page.
- A **page table length register**(PTLR) is maintained to check that the logical addresses generated by the CPU is in the valid range for that process. Failing this check leads to a trap.

5.3 Shared Pages

- Reentrant code (code that can be safely shared by multiple processes) can be placed into shared pages instead of having one copy per process, and it is non-self-modifying (ie. it does not change itself at run time, hence multiple processes can concurrently execute it).
- The page table for each table refers to different pages for the process specific data, but the same pages are referred in all the tables for the reentrant code.
- Heavily used programs like compilers, database systems, window mgmt systems etc. can be placed into shared pages. The OS enforces read-only protection instead of leaving it to the developer of that program.
- Shared memory (used for IPC) is implemented using shared pages in some systems.
- In the below example, the pages associated with a text editor `ed` are shared by 3 processes

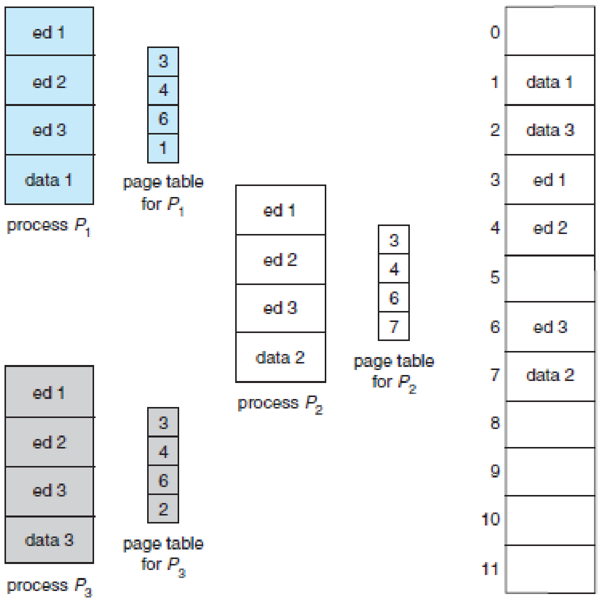

Figure 6: Shared Page Example

6 Page Table Structure

6.1 Hierarchical Page Table

- In order to avoid large page table sizes the page table itself is split into pages.
- The page number p is split into a page table number p_1 and a page table offset p_2.
- These 2 values are combined to get the frame number, and then that entry is combined with the page offset d to get the physical location.
- eg: if virtual address space is of size 2^{32} and page is of size 2^{12} then in a single level page table there are $\frac{2^{32}}{2^{12}} = 2^{20}$ page table entries.
- If the 20 bit page number is split into 10 byte page table number and 10 byte page table offset, then the page table size is $2^{10} + 2^{10}$ which is smaller than the single level one.

- In the DEC VAX architecture (32 bit virtual memory addresses, 512 byte page size), the process' virtual memory is divided into 4 **sections** of size 2^{30} bytes each.

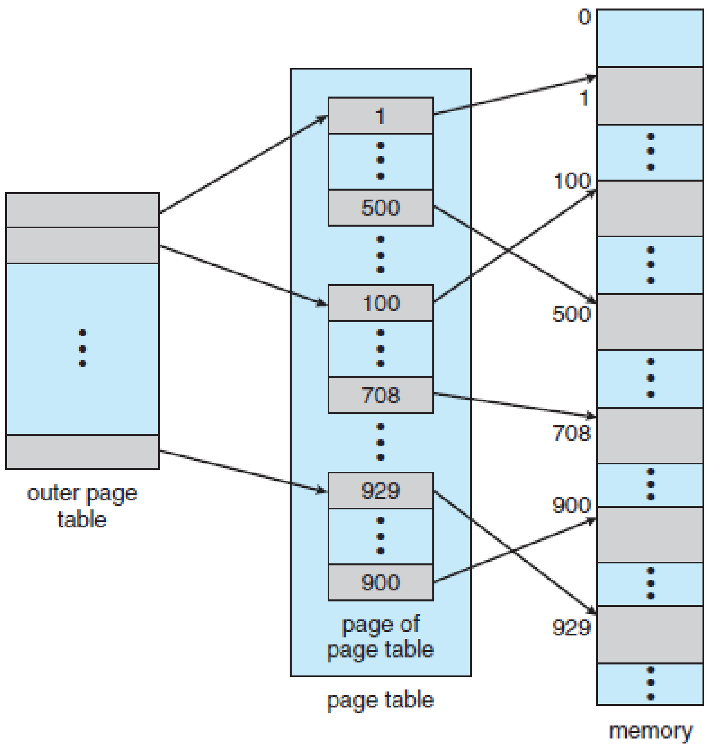

Figure 7: Hierarchical Page Table Structure

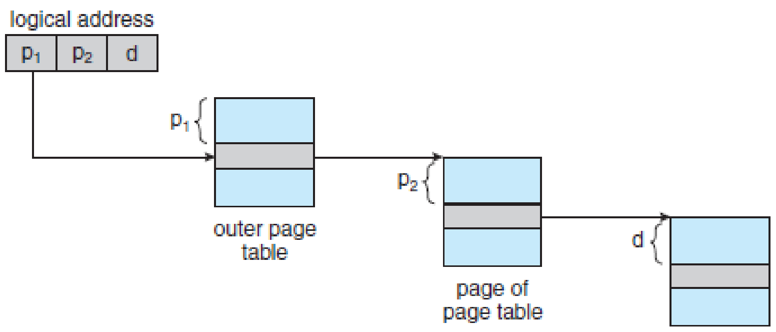

Figure 8: Address translation in Hierarchical Page Table

- The first 2 bits (from left) of the virtual address are the section number, the next 21 are the page number and the last 9 are the page offset.
- In the case where a process uses only one section, the page table is 2^{21} entries long, instead of 2^{23} without the use of hierarchical paging. For further reduction in page table size, each user process' page table is further paged.
- For 64-bit architectures, there is a tradeoff between large page table size (in case of 1 level page table) and prohibitive number of memory accesses (caused by too many levels of hierarchy).
- A 32-10-10-12 page table split causes the outermost table to be 2^{32} entries long, which, if each entry is 4 bytes long, translates to 16 GB.

6.2 Hashed Page Table

- This is used to handle virtual address spaces larger than 2^{64} bytes.

- Each entry of the page table is a linked list of structures. Each element of the list contains: 1) The virtual page number, 2) The frame number corresponding to that page number and 3) pointer to next element in linked list.
- The index is determined by $hash\_fn(virtual\_page\_no)$. The linked list at that index is searched for the given virtual page number, and the appropriate frame number is pulled from the found node.
- In a **clustered page table**, each entry of the hash table can contain frame numbers for several pages at a time (say 16 pages).
- Clustered page tables are useful for sparse address spaces.

6.3 Inverted Page Table

- To avoid large page table sizes, inverted page tables are used.
- An inverted page table has **one entry per frame** of physical memory used. Each entry consists of the virtual address of the page that is using that physical location, and information about which process is using that frame (the PID or the ASID).

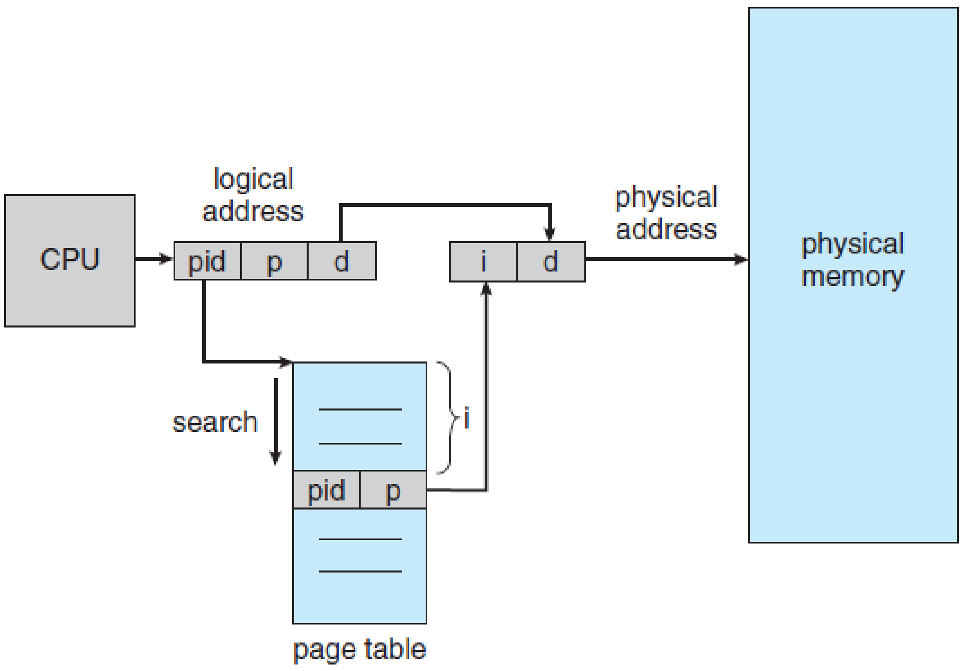

Figure 9: Inverted Page Table

- The below example is for the IBM RT architecture.
- Each virtual address consists of the tuple ⟨ process id, page number, offset ⟩.
- Each page table entry consists of the tuple ⟨ process id, page number ⟩. The index i of the page table is the physical address, that is transferred to the actual memory address along with the offset generated by CPU.
- Searching takes time in inverted page tables because lookup happens through virtual address but the table is sorted in order of physical addresses. Hash tables are used to reduce lookup time but they cause an extra memory reference step.
- Shared pages are hard to implement in inverted page tables, because shared pages mean that multiple virtual addresses (one per process that uses the shared page) map to the same physical address. A reference to a virtual address that is not mapped in the inverted page table will cause a page fault.

7 Demand Paging

- Instead of loading all the pages of a process into the memory at once, pages can be loaded only as and when **needed**. This is known as demand paging.
- Pages are loaded into memory only when the program requests them.
- The **lazy swapper** (or pager, as it deals with swapping individual pages) brings in pages from disk into main memory only when they are requested, instead of bringing in all the pages associated with the process all at once.
- When a process is to be swapped in, the pager guesses which pages are to be brought in before the process is swapped out.
- In order to distinguish the pages that are in disk and memory, the valid-invalid bit is used. If the bit is set to valid, then that page is a valid address for the process *and* it is in memory. If it is invalid then either the page is an invalid address or it is a valid address but it is not in memory.
- If the initial guess is correct then the process will run as if all pages were brought in, and no page faults occur.
- A **page fault** occurs when a process tries to access the address in a page that was not brought into the main memory. Such pages have entries in page table where they are marked *invalid*.
- An attempt to access such a page leads to a trap in the OS, and the trap is the result of the OS failing to bring in the desired pages.
- Handling page faults in demand paging follows the below sequence of steps:
  1. When a page fault occurs, the internal table (inside the PCB of that process) is checked to see if the reference was valid or not.
  2. If it was an invalid reference then the process is terminated.
  3. In case of a valid reference, the page that is not in main memory is paged in. First a free frame is located, a disk operation is scheduled to read the requested page into that located frame.
  4. Once the disk operation is done, the table inside the PCB and the page table are both updated to reflect this change. The instruction that was interrupted by the trap can now be restarted safely.
- In **pure demand paging**, a process is started with **none** of its pages loaded in. One by one, the process goes through multiple page faults until all the required pages are in memory and the process can run to completion.
- For each instruction that is executed, many pages can be accessed at the same time (a single instruction page and multiple data pages). The resulting number of page faults that pure demand paging would cause in this case would be unacceptable in terms of performance.
- Whenever page fault occurs, the current state of the currently executing instruction (registers, PC, condition code) must be saved so that the instruction can be restarted successfully.
- This can be safely done for normal instructions that only modify one location at a time. The difficulty arises when instructions modify more than one location at a time (for example the `MVC` instruction in IBM System 360 architecture, that moves 256 bytes from one location to another (maybe overlapping) location).
- If either source or destination straddles a page boundary, then page fault will occur in the middle of the instruction. If the source and dest overlap then the instruction cannot be restarted as the source may have been modified.
- A simple solution to this is to have the instruction try to access both ends of both blocks before any moving of data. If any page fault is to happen, it will happen at this point, hence the original data is still safe and the instruction can be restarted.
- Another solution is the use of temporary registers to store the old value that can be later written to the right location. In case of a page fault, the old values are overwritten into the original location and the instruction can be restarted.

7.1 Performance of Demand Paging

- The effective access time of demand paging is given by

$$EAT = (1 - p) \times t_m + p \times t_{pf} \tag{1}$$

  Where p is the probability of a page fault, t_m is the average memory access time, and t_{pf} is the time needed to service a page fault.

- The page fault time t_{pf} is the time needed to carry out the following steps:
  1. Trap to the OS
  2. Save current process P_1 state, registers
  3. Check if access was legal or not. If it was legal then find location of that page on the disk. Allocate a disk read operation to a free frame.
  4. Wait for the disk read to take place. This includes the latency of the disk, and the transfer time from disk to memory.
  5. While the above is happening, allocate some other process P_2 to the CPU
  6. When disk read is done, an interrupt is received from the disk, and the currently running process P_2 is saved. If the interrupt is from the disk, then the page tables and process tables are updated. Wait for CPU to be allocated to P_1 again.
  7. Restore the state of P_1 and resume the interrupted instruction.

- Demand paging is also affected by how the **swap space** is handled. As swap space is faster, it is better to load pages from swap space than from the actual file system.

- One option is to load pages for the first time from the file system, but when they are replaced they are sent to the swap space.

- Another option for binary files specifically, is to use the file system as a backing store, and when the page is to be replaced, it can just be overwritten in the same location in memory. For pages not associated with a file (like process stack/heap, aka **anonymous memory**), swap space has to be used. This approach is used on Solaris and BSD UNIX.

8 Copy on Write

- This is an approach similar to shared pages, that is used when a child process is created using the `fork()` call in the parent.

- The traditional approach is to make the child as an exact full copy of the parent address space. But since many child processes anyway erase the parent copy by using the `exec()` call, this copying is not necessary.

- In copy on write, initially the child and parent share the same pages, and these pages are marked as Copy on write pages.

- If one of the process wishes to modify such a page, then the OS creates a copy of the page, maps it to that process address space, and then allocates that page to the process that modified.

- Only pages that can be modified are marked as CoW. Non modifiable pages (like code) are still shared by parent and child.

- Free pages are allocated to the process that modifies, from a **pool** of free pages. This pool of free pages is used when stack/heap for a process has to expand, or CoW needs to take place.

- OSes uses **zero fill on demand** pages that have been filled with 0 before allocating (to clear out the previous content).

- The `vfork()` system call in UNIX is used to create a new process without creating a copy of the parent process' memory. The child shares all memory with parent until it issues an `exec()` call, then the copy is done and the appropriate code is loaded onto the process memory.

- As `vfork()` uses CoW, it must be used with caution so that the child does not modify the parent's address space. It is intended to be used when child calls `exec()` immediately after creation.

9 Page Replacement

- Page replacement is used to handle the case when there are no free frames in the memory, but a new page is to be loaded from the disk.
- With page replacement, the sequence of steps in handling a page fault changes to:
  1. When a page fault occurs, the internal table (inside the PCB of that process) is checked to see if the reference was valid or not.
  2. If it was an invalid reference then the process is terminated.
  3. In case of a valid reference, the page that is not in main memory is paged in. If a free frame is found, then it is used. If no free frame is found, a page-replacement algorithm selects a **victim frame**. The victim frame is swapped out and the new page is loaded in its place.
  4. Once the disk operation is done, the table inside the PCB and the page table are both updated to reflect this change. The instruction that was interrupted by the trap can now be restarted safely.

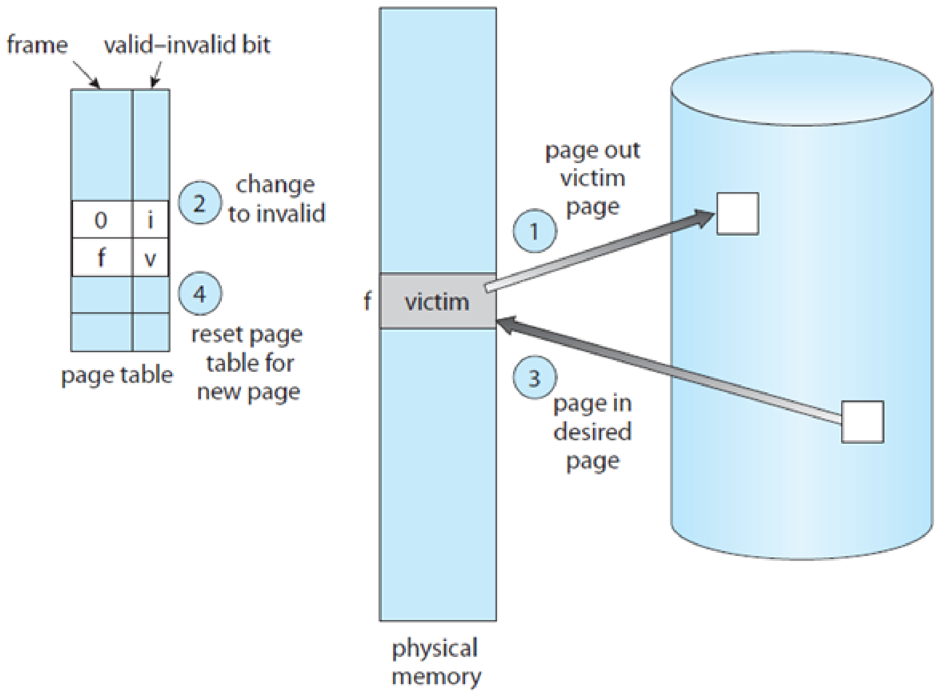

Figure 10: Page Replacement

- Every time page replacement takes place, there are 2 disk operations that take place. This doubles the page fault service time t_{pf} in case replacement must take place.
- The number of disk operations involved in page replacement can be reduced by the use of a **modify bit** (or a *dirty bit*) in the page table.
- If the modify bit is set, then the page has been modified since it was loaded from disk, and if the bit is not set then it is not modified since it was loaded from the disk.
- If the modify bit of a victim page is not set, then it does not have to be written back to the disk (the new page can simply overwrite on the existing page).
- Page replacement (along with demand paging) is the crucial link between virtual and physical memory.

- eg: If a process needs 20 pages to run, then without demand paging all 20 pages must be in the physical memory. But with demand paging, the process can take up only 10 pages and the rest of the pages can be loaded only when they are needed using some page replacement algorithm.
- This means that a larger virtual memory space can be given to the process, while the actual physical memory size is still the same.
- The components of the paging system are a **frame-allocation** algorithm (with multiple processes in memory, how many frames does each one get) and a **page replacement** algorithm (if no free frame is available, which frame is selected as victim and swapped out of memory).
- Performance of page replacement algos is tested on a reference string of address (which can be converted into a reference string of page numbers) and a fixed number of frames, and counting the number of page faults. Less page faults mean better performance.

9.1 FIFO Page Replacement

- Out of all pages in the memory, the page that was bought in first is swapped out.
- The below example considers a memory of size 3 frames.

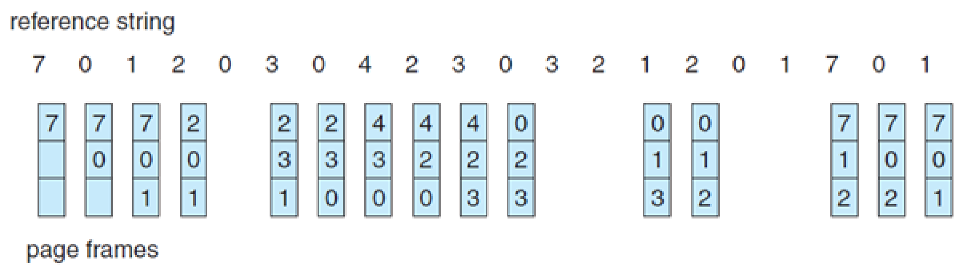

Figure 11: FIFO Page Replacement

- FIFO is simple and easy to implement, but causes sub-optimal performance (if the oldest page happens to be one that is constantly in use, then replacing it can cause immediate page fault and one more replacement).
- FIFO also suffers from **Belady's Anomaly**, which is an experimental result. The observation is that increasing the number of frames in memory does not reduce the number of page faults.

9.2 Optimal Page Replacement Algorithm (OPT)

- The page that *will not be used* for the longest time is replaced.

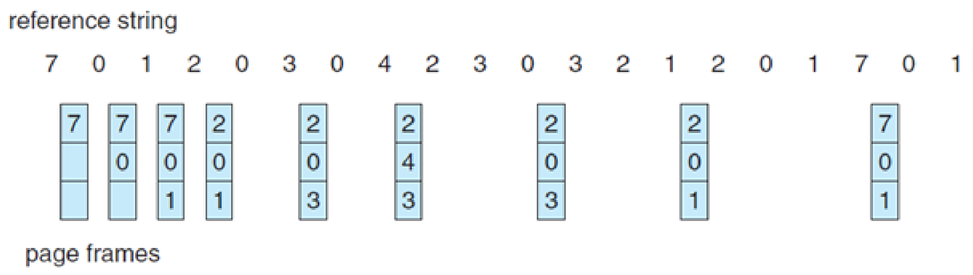

Figure 12: Optimal Page Replacement

- OPT offers excellent performance (better than FIFO) but is hard to implement as it requires future knowledge of what pages be used in the reference string.
- OPT also does not suffer from Belady's Anomaly

9.3 LRU Page Replacement

- The page that *has not been used* for the longest time is replaced.

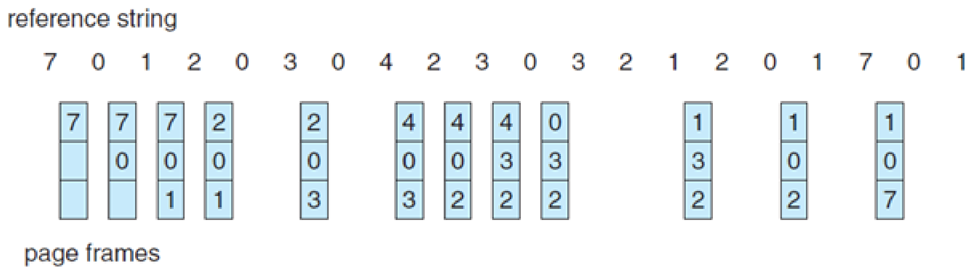

Figure 13: LRU Page Replacement

- LRU is the exact reverse logic of OPT algorithm.
- Let S be a reference string and S^r be the reverse of S. The page fault rate on a sS using LRU would be the same as the page fault rate S^r using OPT.
- LRU can be implemented using
  - **Counters**:
    1. Every page table entry has a time-of-use field and there is a single clock register attached to it.
    2. Whenever a reference is made to a page, the time of use field of that page table entry is assigned the current clock register value.
    3. For every page replacement, the entire page table is searched and the page with the smallest time value is replaced.
    4. Clock overflow, changes to page table (because of scheduling) and the number of memory writes (one for changing time of use in the page table) are to be considered.
  - **Stack**:
    1. A stack of page numbers is maintained.
    2. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom
    3. Since page must be removed from the middle of the stack, the stack is implemented as a doubly-linked list, with a head and a tail pointer
    4. Every update to the stack requires changing at most 6 pointers and getting the LRU page is easy (tail pointer).

10 Frame Allocation

- The simplest strategy for frame allocation is to put all the free frames in memory in the free-frame list, and all processes can share from this pool.
- The OS can also be allowed to use the free frames for buffers and tables, but a certain number of frames are always reserved as free frames, so that if replacement must happen, then a free frame is always available.

10.1 Minimum Number of Frames

- Every process must be allocated a certain minimum number of frames when it is loaded. This is primarily done for performance reasons, so that memory references do not result in page faults.
- If there are n levels of possible memory references, then $n+1$ pages must be loaded into the memory at minimum.

- The number of levels of indirection must be restricted to ensure that the physical memory is not filled up entirely by one process' pages.
- The maximum number of frames is dictated by the amount of physical memory, and the miniumum number of frames is defined by the architecture (the number of indirect references dictates this).

10.2 Allocation Algorithms

- In **equal allocation**, the m available frames are distributed among the n processes in equal proportion. Each process gets m/n frames of memory.
- This results in unused memory if the number of frames allocated is too large for the process size.
- In **proportional allocation**, each process is allocated frames according to its size. If the size of process p_i is s_i, then the total size of all the processes is

  $$S = \sum_{i=1}^{n} s_i$$

  and the allocation for each process is

  $$a_i = \frac{s_i}{S} \times m$$

  where m is the total number of available frames. a_i is rounded up to the nearest integer so that it is greater than the minimum number of frames needed per process according to the architecture.
- Allocation varies according to the degree of multiprogramming in the system. In case of increased multiprogramming, processes will lose some frames to accommodate new processes that enter the system.
- Both equal and proportional allocation treat high priority processes and low priority processes as equals. The proportional allocation scheme can be modified to use priorities of processes instead of their relative size.

10.3 Global v Local Allocation

- All page replacement algorithms can be classified into global (victim frame is selected from the entire memory) and local (victim frame is selected from frames that are allocated only to that process).
- In a priority-based global allocation system, the high priority processes can select frames that are allocated to lower priority processes for replacement. But in this scheme a process does not have full control on its page fault rate as other processes are also involved in selecting frames for replacement.
- In local allocation however, the process' own less used pages are not available to it, hence causing lower system throughput. Hence global allocation is preferred.

10.4 UMA vs NUMA

- Memory access on NUMA systems can depend on whether the memory to be accessed is on the same board as the CPU who is accessing it, or on a different board.
- Allocation algorithms that take NUMA into account offer better performance on NUMA systems compared to algorithms that do not.
- The algorithmic changes consist of having the scheduler track the last CPU on which each process ran. If the scheduler tries to schedule each process onto its previous CPU, and the memory-management system tries to allocate frames for the process close to the CPU on which it is being scheduled, then improved cache hits and decreased memory access times will result.
- A process with many *threads* can end up with those threads scheduled on different boards.
- In Solaris, there are **lgroups** (Latency groups) of CPU and memory that are close to each other. The scheduler tries to schedule all the threads of a process within the same (or nearest) lgroup and allocation happens within that lgroup.

11 Thrashing

- A process is said to be thrashing if it spends more time on paging activities (like page replacement) than on actual execution.
- Given any process that does not have enough pages, and all of the pages are in active use. A page fault occurs and one of the currently active pages is replaced, which causes another page fault. This causes a never ending sequence of page faults.

11.1 Cause of Thrashing

- If the CPU utilization falls below a certain extent, the OS loads new processes into memory.
- Considering a global allocation system, the processes replace frames without regard for which process they belong to. Thus the processes that lose frames start to encounter page faults, thus selecting frames from other processes to be replaced, causing those processes to encounter page faults, and so on.
- As more and more page faults occur, more and more page in requests are made to the disk, thus processes wait for the disk in a queue. While they wait, no work is being done, hence the CPU utilization falls and the OS loads in more processes.
- As these processes wait for paging to occur, throughput of the system falls, and **thrashing** starts.

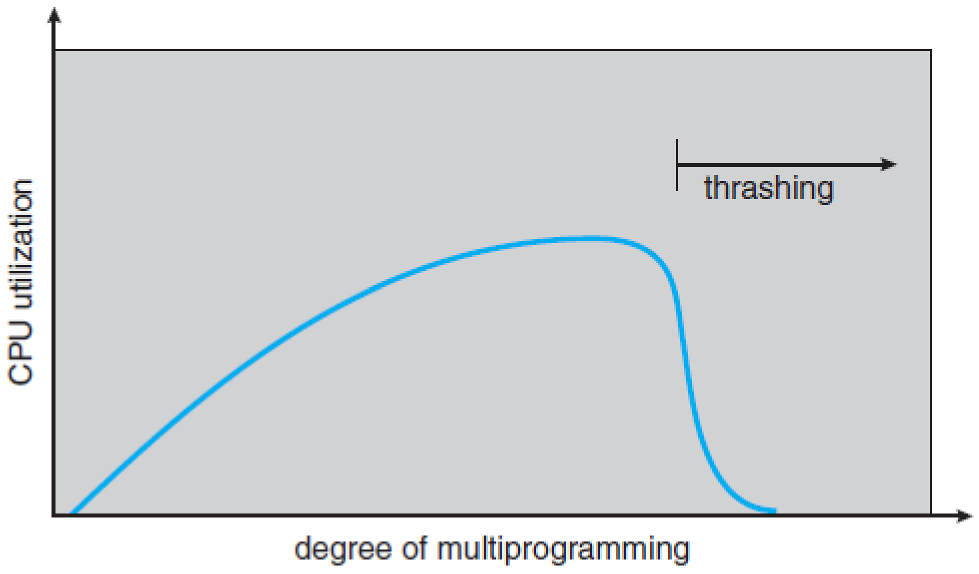

Figure 14: Thrashing

- The effects of thrashing can be limited using a **local replacement** or a **priority replacement** algorithm.
- With local replacement, one process that is thrashing cannot cause cause other processes to thrash by stealing their frames and replacing them.
- But if a process is thrashing, it occupies its place on the disk queue indefinitely, hence the average page fault service t_{pf} increases even for processes that are not thrashing.
- The locality assumption is used to develop the **working-set model** that is used to solve this issue.
- The assumption of locality states that a process uses a group of pages that are active all at once (this group is called a locality, and multiple localities can be overlapping), and processes can move between localities.

- Caching is another technique that makes use of the principle of locality.
- If all the frames are allocated for one locality, then the process will first fault until all the pages of that locality are loaded in memory, and then it will not fault until it switches localities.
- If frames are allocated less than the process' locality size, then it will thrash as all the pages it is actively using cannot fit in memory.

11.2 Working Set Model

- A parameter Δ, called the *working-set window* is used in this model. The set of Δ most recently used pages is the working set.
- The current working set at time t is the set of Δ most recent pages that were referenced before that time t. The working set can be thought of as an estimate to the locality of the process.
- The accuracy of the working set depends on the value of Δ. If it is too small then it will not cover the entire locality, and if it is too large then it overlaps many localities.
- The total page demand D is calculated as

$$D = \sum_{i=1}^{n} WSS_i$$

- If $D > m$ then thrashing will occur as the demand is more than the number of free frames.
- The OS monitors the working set of each process, and allocates as many frames as the working set size to that process. If enough extra frames are available then another process can be initiated.
- If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are written out (swapped), and its frames are reallocated to other processes. The suspended process can be restarted later.
- The working-set model can be approximated with a fixed-interval timer interrupt and a reference bit.
- For example, let Δ equals 10,000 references and that timer interrupt occurs every 5,000 references.
- When a timer interrupt happens, the reference bit values are copied and cleared for each page. Thus, if a page fault occurs, the current reference bit and two in-memory bits can be examined to determine whether a page was used within the last 10,000 to 15,000 references.
- If it was used, at least one of these bits will be on. If it has not been used, these bits will be off. Pages with at least one bit on will be considered to be in the working set.

11.3 Page Fault Frequency Model

- Thrashing leads to an increase in the page fault rate, and the PFF model is a method to control the page fault rate.
- Thresholds are set for max and min page fault rate. If the page fault rate exceeds the upper bound, then more frames are allocated to that process. If page fault rate falls below the lower bound, a frame is removed from that process address space.
- If the page-fault rate increases and no free frames are available, a process must be selected and swapped out to backing store. The freed frames are then distributed to processes with high page-fault rates

12 Case Studies in Memory Management

12.1 Intel IA-32 Architecture

12.1.1 Segmentation Hardware

- There are 16K segments per process, and each segment is allowed to be as large as 4 GB.
- The logical memory of a process consists of these 16K segments divided into two partitions. One partition is private only to that process, and its segment table is called the **Local Descriptor Table** (LDT). The other one is meant to be shared between all processes, and its segment table is called the **Global Descriptor Table** (GDT).
- Each entry in the LDT and GDT is 64 bits long, with detailed info about that segment.
- The *selector* is a 16 bit portion of the table entry that consists of a 13-bit segment number, one bit that indicates GDT or LDT, and 2 protection bits. The offset is a 32-bit number that specifies the location of the byte within the segment.
- The machine has 6 segment registers (allowing six segments to be addressed at any one time by a process), and 6 microprogram registers to hold the corresponding descriptors from either the LDT or GDT. This cache lets the processor avoid having to read the descriptor from memory for every memory reference.
- The segment register points to the appropriate entry in the LDT/GDT, and the base+limit info from the descriptor is used to generate a 32-bit linear address.

12.1.2 Paging Hardware

- Intel IA-32 architecture supports page sizes of 4KB or 4 MB.
- For 4KB pages, a 2-level hierarchical page table is used. The 32 bit virtual address generated above is converted into a 10, 10, 12 bit split. The first 2 groups of 10 bits are the page numbers in the lower level of the page table, and the last 12 bits are the page offset.
- The outermost level of the page table hierarchy is called the **page directory**.
- For 4MB pages, the leftmost 10 bits are the page number and the remaining 22 bits are the page offset. The middle level of the hierarchy is bypassed altogether.
- The page size is mentioned in the `Page_Size` field of the page directory. (1 for 4MB, 0 for 4KB).
- The page tables can be swapped to and from disk. The page directory maintains a valid-invalid bit to check if the inner page table referred to is in memory or not.
- Under the **page-address extension**(PAE) scheme, the two level hierarchy is split into 3 levels, and the page-directory and page-table entries are now 64 bits long (instead of 32 without PAE).
- Now the base address of page tables and page frames is expanded to 24 bits (instead of 20) and this combined with the 12-bit offset leads to a total of 2^{36} bits or 64GB of total addressable memory.
- With the three level hierarchy, page sizes supported are 4 kB and 2 MB.

12.1.3 Intel x64 architecture

- Instead of using all 64 bytes for memory addressing, only 48 bytes are used to address.
- 4 levels of paging hierarchy provide support for page sizes of 4kB, 2MB or 1 GB.
- Due to the use of PAE, the 48 byte address can be used to address 2^{52} bytes or 4096 TB of physical memory.

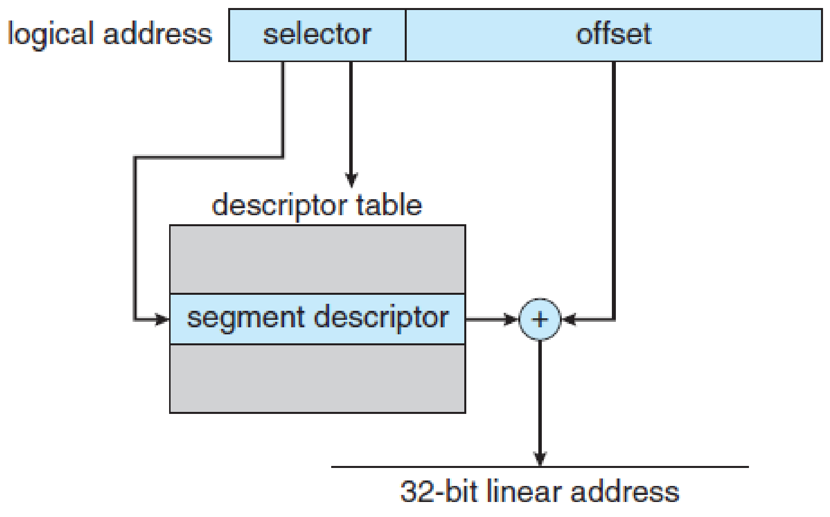

Figure 15: Segmentation hardware of Intel IA-32

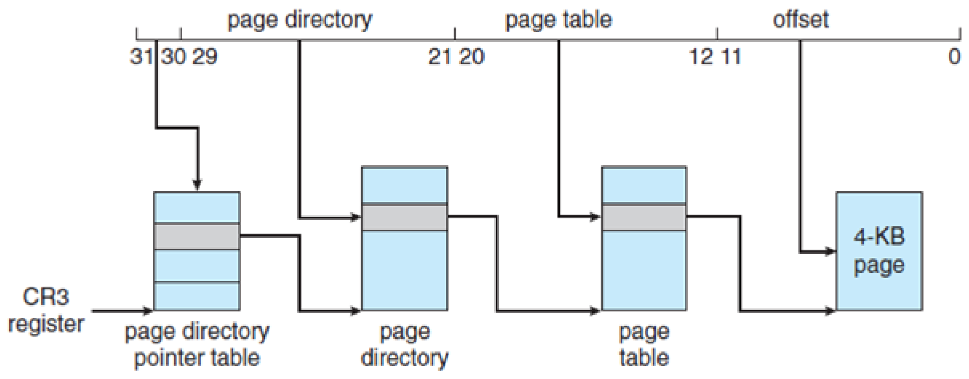

Figure 16: PAE in Intel IA-32

12.2 Windows

- Windows uses demand paging in conjunction with **clustering** to implement virtual memory management.
- In clustering, if a page fault occurs, then the pages that are close by to the faulted page (in a cluster) are all brought in (instead of only the faulted page). This takes advantage of the principle of locality.
- When a process is created, it is assigned a **working set maximum** (largest possible number of pages in memory for that process) and a **working set minimum** (min number of pages that is guaranteed to have in memory for that process) .
- If a process that is already at the working set maximum undergoes a page fault, local LRU replacement is used to load the new page.
- If the amount of free memory falls below a threshold, then all processes undergo **automatic working set trimming**. For each process, if it has been allocated more pages than the working set minimum, the extra pages are removed.
- Processes that are already at their working set minimum can be allocated some pages if available.
- Windows performs working set trimming on user as well as system processes

12.3 Solaris

- Solaris maintains a `lotsfree` variable that represents the threshold for number of free pages in memory. Normally it is set to 1/64 of total physical memory size.
- If number of free frames falls below `lotsfree`, a process called **pageout** begins. A check for this is done 4 times per second.
- The **reference bit** is a bit in the page table that is set every time that page is referenced in the process.
- There are two scanners involved in the pageout process. The first scanner scans all the pages and sets the reference bit to 0 for all. The second scanner follows the first scanner and checks for all the pages that still have 0 reference bit (between the first and second scanners processes are still running and they might make references to some pages hence setting those bits to 1).
- If the frame with reference bit 0 has been modified, then the second scanner also writes it to the disk. Then this frame is added to the free frame list.
- The `scanrate` parameter is used to control the rate of scanning. When the amount of free memory falls below `lotsfree` then scanning starts at the rate of `slowscan`, progressing up to a maximum possible rate of `fastscan`.
- The `handspread` parameter controls the distance between the two scanners (in terms of number of pages).
- If free memory falls below the value of `desfree`, pageout will run a hundred times per second with the intention of keeping at least `desfree` free memory available.
- Enhancements to this algorithm in recent releases include ignoring shared pages for scanning, and giving priority to process files over normal memory files, also called **priority paging**.

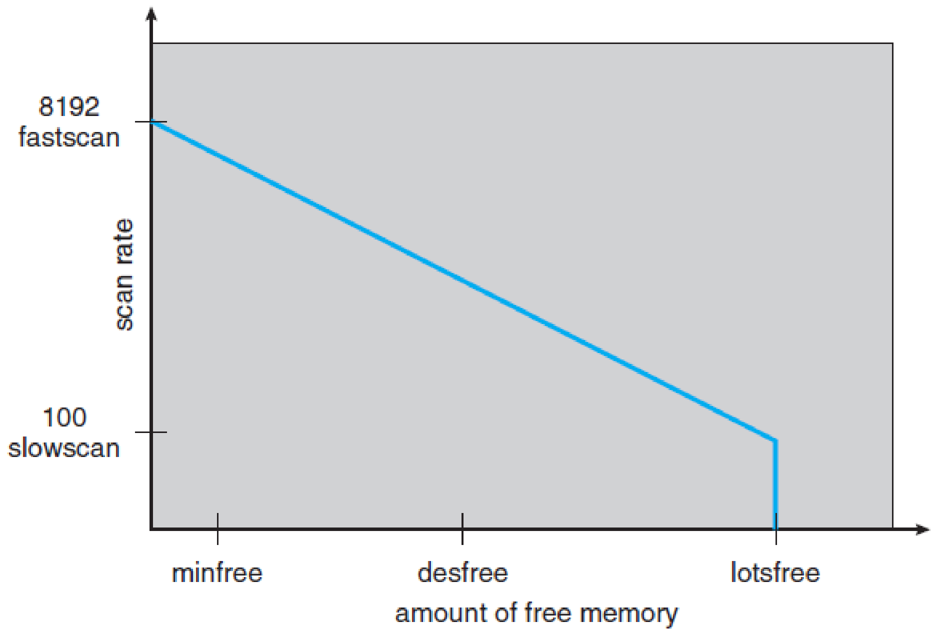

Figure 17: Solaris page scanning rate