This chapter discusses **virtual memory**, which maps a potentially larger logical address space onto a smaller physical memory. The key idea is to enable computer systems to execute partially loaded programs, which can run extremely large processes and frees application programmers from concerns of the amount of memory available in a computer system when writing programs.

Virtual Memory

- The separation of user *logical memory space* from *physical memory*.
- It allows part of user program (code and data) to reside in memory for execution. The argument for this is that in many cases there is no need to load the entire program into physical memory since some of it might never be used such as routines handling errors and large data structures.
- **Benefit:** (1) users or programmers are no longer constrained by the amount of physical memory available since each process occupies less amount of memory; (2) each user program takes less physical memory, which increases the degree of multiprogramming, and thus improves the CPU utilization; (3) less I/O needed to load or swap processes.
- **Complexity:** there are two fundamental issues to be addressed in order for this to work, (1) how much memory be allocated to a process, i.e., number of pages allocated to a process in demand paging. Recall the memory requirement of a process during different stage of execution is a variable; (2) how to handle page fault, i.e., when the memory accessed is not brought into physical memory yet.

Demand Paging

- **Pure Demand paging**: it brings a page into memory only if this page is actually referenced (i.e., any address within the page is referenced). In another word, pages not referenced during execution will never be brought into the memory.
- **Page fault** has to be handled under a demand paging scheme. The performance of the demand page scheme is primarily determined by the **page-fault rate**.
- **Free-frame list** is a pool of free frames that OS maintains to satisfy frame requests from users. OS typically allocates free frames using a technique known as **zero-fill-on-demand** -- the content of the frames gets *zeroed-out* before being allocated or reallocated, eliminating the possibility that old content be reused.

Page Replacement Policies

- The page replacement policies discussed here are under the assumption that each process is allocated with a **fixed** number of frames by using a **local replacement policy**. When a page fault occurs and if all frames allocated to a process are occupied, one of the frames is selected as a **victim** to be replaced. The difference in different page replacement policies lies in which page is selected as the ***victim***.
- **FIFO replacement**: it replaces the page that was brought into the memory the earliest. This suffers from the **Belady's Anomaly**, in which a higher page fault rate might occur when extra frames are allocated to a process. The cost of

implementing FIFO policy is low as it only needs to record the time that a page is brought into the memory, requiring no update each time memory is referenced.

- **Optimal (OPT) replacement:** it replaces a page that will not be used for the longest period of time in the future. This guarantees the minimum page-fault rate given a fixed number of frames; however, this is practically infeasible as it requires future knowledge of page reference, so it is only used as a benchmark for comparison.
- **LRU replacement**: it replaces the page that has not been used for the longest period of time (by examining the past). This is an approximation of OPT under the **locality model**, which states the (most) recently referenced memory locations will likely be accessed again in the near future. This can be implemented by a **counter** or **stack** algorithm, which requires hardware assistance and updates (software manipulation) for every memory reference. As the update can be very frequent (for each memory reference), LRU is usually too expensive to implement in practice.
- **Second-chance algorithm:** One bit used as a *reference bit* for each page that is set to 1 with a reference to this page. FIFO replacement order is followed for each page with the reference bit (0). If the reference bit is 1, reset to 0 (second chance). It basically finds the next page with reference bit = 0 according to FIFO order. A **Clock** algorithm can be used to implement the second-chance algorithm. This is an approximation to LRU under a coarse-granularity. Noticing when all the reference bit is 1, this performs the same as FIFO replacement after giving each page a second chance.

The Working-Set Model

- **Assumption**: a working-set model is based on ***memory reference locality*** in that memory access or subsequent memory access of a process tends to stay in the same set of pages - locality. A process migrates from one locality to another, e.g., operating on a different set of data or calling a subroutine over the course of its execution. The number of frames required or allocated to a process is a variable.
- The **working-set model** keeps tracks of the set of pages referenced along a given time line, which attempts to capture the minimum number of pages needed for a process (i.e., the current locality) during the course of a process execution.
- **Working-set window**: this specifies a window size (usually fixed) in term of the number of memory references (or instructions) it will keep track of. The selection of the working-set window size is non-trivial as it cannot be too small (does not capture the current locality), nor can it be too big (beyond the current locality).
- **The basic idea:** suppose we have selected an adequate working-set window size, the working-set or WSS (working-set for a process) would roughly capture the current locality, i.e., the minimum number of pages the process needs at the moment. This allows the operating system to dynamically adjust the number of frames allocated to each process based on the current working-set.
- **Thrashing:** since WSS_i is a good approximation of the number of pages needed for a process i at a given time, the total amount of memory needed (D) at the time

by all processes is the sum of all WSS_i. If D > m (the actual physical memory size), thrashing will occur, in which at least one process is in short of memory, thus its pages are swapped in and out frequently due to high page faults.

- **The difficulty:** This requires keeping track of the working set for each process. The working-set window is a moving window with each memory reference. A page is in the working-set if it has been referenced anywhere in this working-set window. This has too much overhead, esp. with a large working-set window. Note that for each memory reference by any process, there must be an update for its working-set (even if there is no change). An approximation can be used by adjusting a time interval and a reference bit.

Other Design Issues

- **Pre-paging** can be used at the start of a process execution to preload certain number of pages into the memory to reduce the initial high page fault rate. This is useful when the cost of the saved page faults is greater than the cost of pre-paging unnecessary pages, i.e., the pages that will not be used.
- **Page size**: there is a conflicting set of factors that affects the selection of page size, internal fragmentation, resolution, page table size, I/O overhead, reference locality. For instance, larger page size results in more internal fragmentation and less resolution; smaller page size results in large page table.
- **TLB reach**: the amount of memory that the TLB can access, determined by the number of entries in TLB and page size. This calls for multiple page sizes in some cases. This determines the amount of memory that can be directly accessed by TLB (without the need to access the page table in the memory for address translation). Ideally, the working set of each process should be stored in the TLB for fast access.