Chapter 10: Virtual Memory

1

Chapter Topics

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations

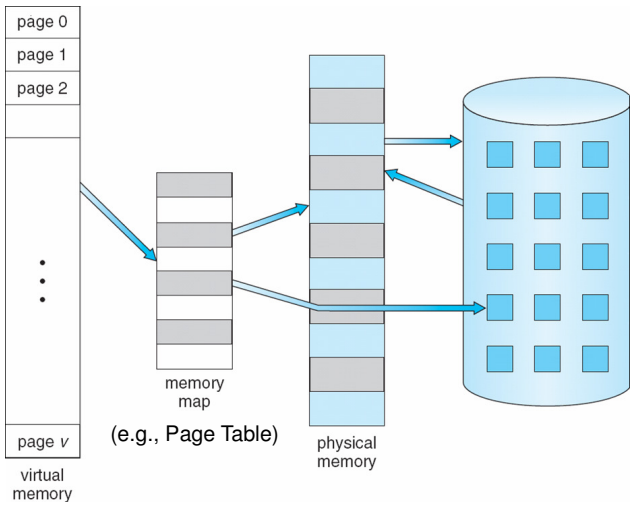
2

Background

- Virtual Memory (VM) of a process means the Logical Address Space of the process.
- **Virtual memory** is a technique that allows the execution of processes that are not completely in memory.
  - Separation of logical address space from physical address space.
  - Logical address space can therefore be much larger than physical address space. ⇒ simplifying the programming task.
  - Allows system libraries to be shared by several processes through mapping of them into the virtual address space of each process.
  - Speeding up process creation because pages (in memory) can be shared between the parent and child processes.
- Virtual memory can be implemented via:
  - Demand paging: A page is loaded into memory only when it is needed (i.e., referenced) during the program execution.
  - Prepaging (i.e., Prefetching of pages): A page (that is expected to be needed) is loaded into memory before it is actually needed during the program execution.

3

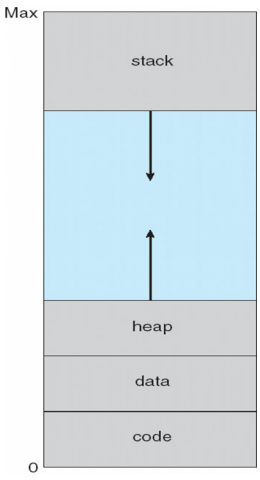
Virtual Memory That is Larger Than Physical Memory



4

Virtual Memory (i.e., Logical Address Space) of a Process

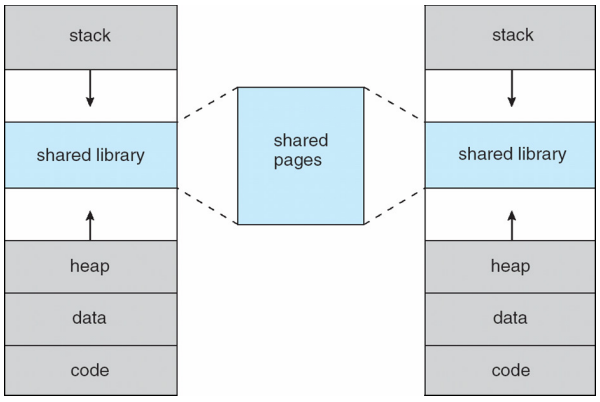
- Logical view of how a process is stored in memory.



5

Shared Library Using Virtual Memory

- Each process considers the shared libraries to be part of its virtual address space, but the actual pages (of each shared library) in the physical memory are shared by all the processes.



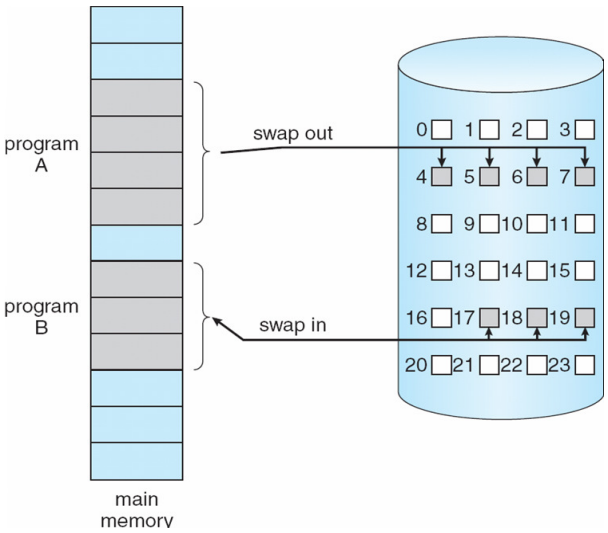
6

Demand Paging

- A page is loaded into memory only when it is needed during the program execution.
  - Less main memory space is needed for each process.
  - Faster response.
  - More processes can share main memory.
- Instead of swapping the entire process into memory (by using a swapper), we use a *pager* (i.e., a paging device or a system) which swaps a page into memory only when that page is needed (or prefetched).

7

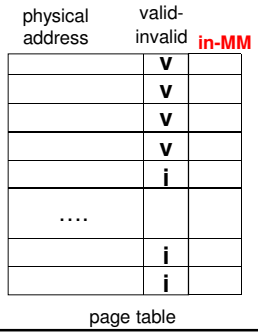
Transfer of Pages of a Process to Contiguous Disk Space Can Reduce the Transfer Time



8

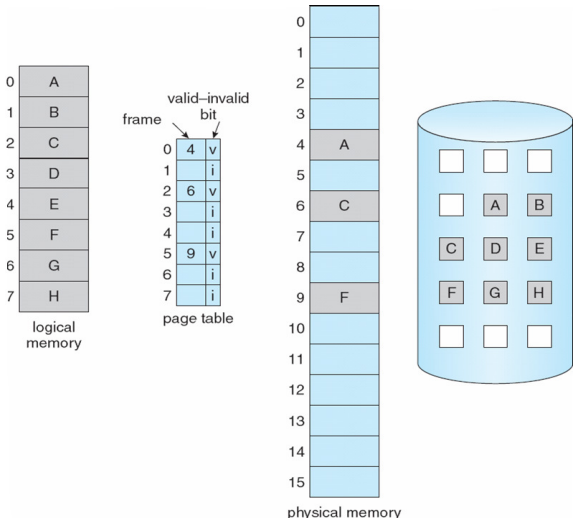
Valid-Invalid Bit in the Page Table

- A valid–invalid bit is associated with each entry in the page table.
  - **Valid (v)**: the page is in the logical space of the process (i.e., legal) and in memory.
  - **Invalid (i)**: the page is not legal or not in memory.
- Note: It is more clear to indicate whether a page is in main memory or not by using a separate bit (in addition to the valid-invalid bit) in the page table.



9

Page Table When Some Pages are Not in Main Memory



10

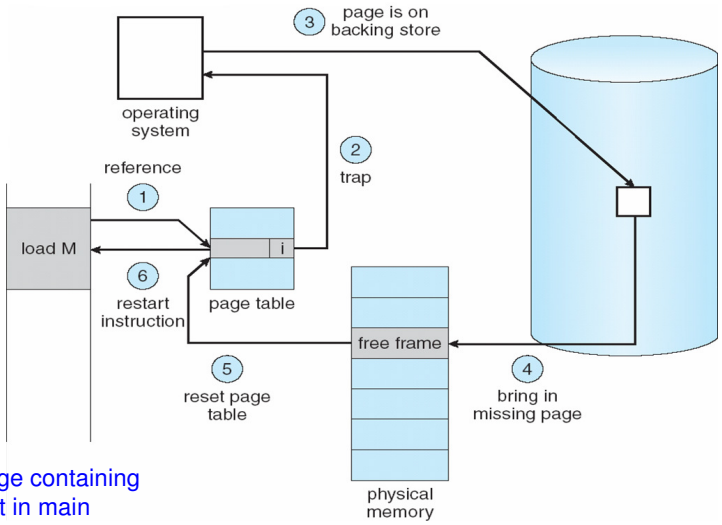
Page Fault

- Access to a page that is not in the main memory (or a page marked invalid) causes a page fault, a trap to the OS. The procedure for handling page fault is as follows:

1. OS checks an internal table (usually kept with the PCB) for this process to determine whether the reference was a valid or invalid memory access.
   - If invalid reference (i.e., reference an illegal page) ⇒ abort the process
   - If valid reference ⇒ fetch the page
2. Find a free frame in memory.
3. Read the demanded page from disk into the free frame.
4. Modify the page table entry of the page (by setting the valid-invalid bit to *v*).
5. Restart the instruction that caused the page fault.

11

Steps in Handling a Page Fault



The page containing M is not in main memory.

12

Page Fault (cont'd)

- A page fault may occur at any memory reference:
  - If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again (after the page containing the instruction is loaded into memory).
  - If the page fault occurs while fetching an operand of an instruction, after the page containing the operand is loaded into memory, we must fetch and decode the instruction again then fetch the operand.
    - For example, ADD A, B, C (for C ← A + B) may cause a page fault on the page containing C when we try to store the result (of A + B) in C.

13

Performance of Demand Paging

- Page fault rate ($0 \le p \le 1.0$) means the probability of a page fault ⇒ page hit rate is (1- p)
  - if p = 0, no page fault
  - if p = 1, every reference causes a page fault.
- Effective Access Time (EAT) of a referenced word:

  EAT = (1 – p) x memory access time
  + p x {service the page fault interrupt
  + swap out a page (if necessary)
  + swap in the demanded page
  + memory access time
  + restart overhead}

14

EAT Example

- If memory access time for a word is 200 nanoseconds, and average page-fault service time is 8 milliseconds

  EAT = (1 – p) x 200 nsec + p x 8 msec
  = (1 – p) x 200 + p x 8,000,000 nsec
  = 200 + p x 7,999,800 nsec

- If one out of 1000 references causes a page fault (i.e., p=0.001), then EAT = 8.2 microseconds.
  - This is a slowdown by a factor of 40!
  - Page fault also causes a context switching to another process (selected by the CPU scheduler).

15

Swapping Pages of a File

- Disk I/O to the swap space (allocated on a disk) is faster than that to the file system, because
  - Swap space is usually allocated in much larger blocks, and a separate **swap-space storage manager**, which is much simpler than typical file management system, is used to allocate and deallocate disk blocks from the raw partition.
- Alternatives for swapping pages of a file:
  - Copying the entire file into the swap space and then performing demand paging from the swap space.
  - Demand paging from the file system initially, but writing (i.e., transferring) the pages to the swap space as they are replaced from memory (in order to bring in other pages).
  - To save the swap space, for read-only files, when their pages are replaced, the page frames containing them could be simply overwritten (with new pages being fetched) because they are never modified.

16

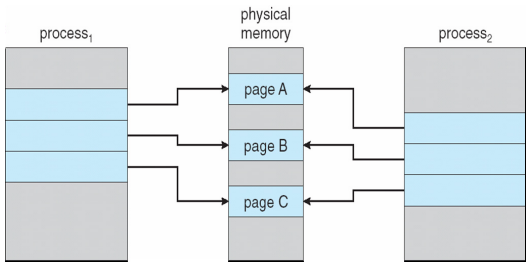
Copy-on-Write

- Traditionally, **`fork()`** system call creates a child process which duplicates the pages of the parent process.
- The copy-on-write (COW) allows both parent and child processes to initially share the same physical pages in memory.
  - These shared pages are marked as copy-on-write pages, meaning that if either process tries to modify a shared page, a copy of the shared page is created in memory and then modified.
- COW allows more efficient creation of a process as only the pages to be modified are copied later.

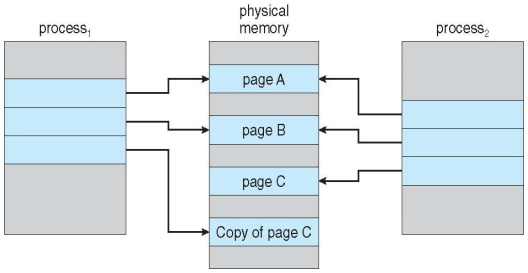
17

Copy-on-Write Example

Before process$_1$ modifies page C



After process$_1$ modifies page C



18

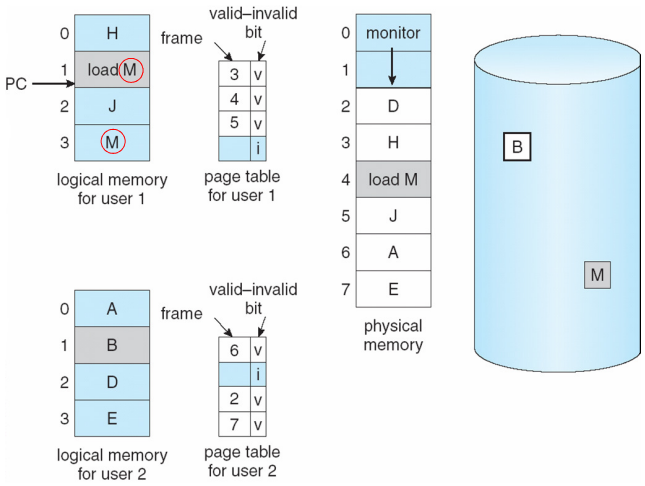
Page Replacement

- When page fault happens, we need to find a free page frame in the memory. If there is no free page frame, we can select a virtual page stored in a frame for replacement.
  - The selected virtual page (stored in a frame) is swapped out (or discarded if it was not modified), and the newly referenced virtual page is fetched into that page frame.
- To avoid swapping out every virtual page to be replaced, we can associate a *modify bit* (aka *modified bit* or *dirty bit*) with each virtual page in the memory. If a virtual page is modified (i.e., written) while it has been in the memory, its modify bit is set to one.
  - Only modified virtual pages are written to disk when they are replaced.

19

Need for Page Replacement

- Virtual page 3 (containing M) of Process 1 is referenced, which causes a page fault. But, there is no free frame in memory.

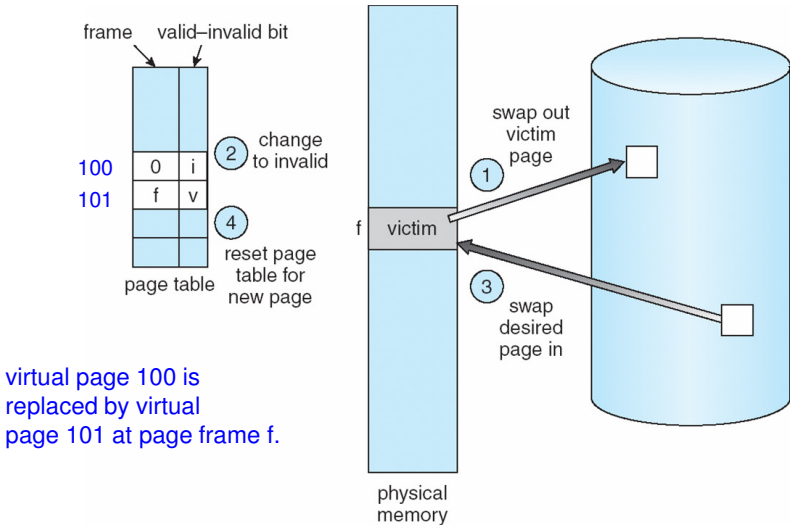


20

Basic Page Replacement

1. Find the location of the desired page on disk.
2. Find a free frame:
   (a) If there is a free frame, use it.
   (b) If there is no free frame, use a page replacement algorithm to select a victim page (stored in a frame). If this victim page has been modified, it should be written to disk.
3. Read the desired page from disk into the selected frame and update the page table.
4. Resume the process (by restarting the instruction which caused the page fault).

Page Replacement



virtual page 100 is replaced by virtual page 101 at page frame f.

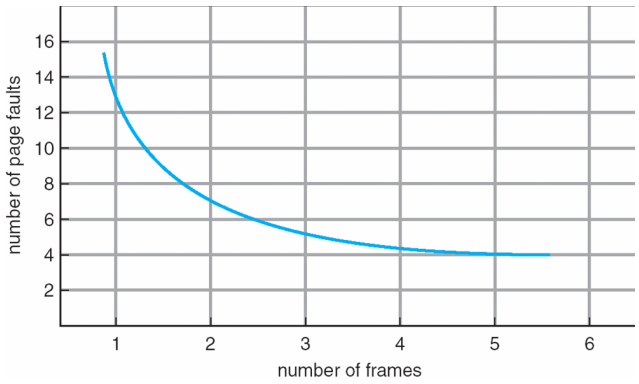
Page Replacement Algorithms

- A good page replacement algorithm is the one that results in a low page fault rate (for all the processes).
- We can evaluate a page replacement algorithm by running it on a particular reference string (i.e., sequence of referenced pages) and counting the number of page faults on that string.
  - For example, a reference string could be the following sequence of referenced virtual page numbers: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

23

Page Faults versus the Number of Frames

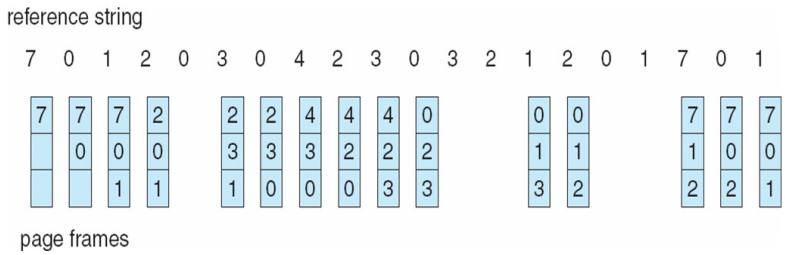
- In general, as the number of page frames allocated to a process increases, the number of page faults decreases to some minimal level.



24

FIFO Page Replacement

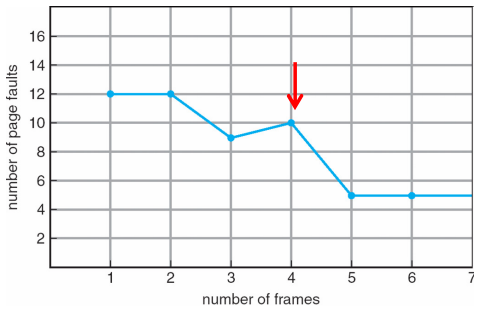
- The FIFO replacement algorithm selects the virtual page that has been in main memory for the longest time.
- A FIFO queue can be used for implementation.
  - Newly fetched page is place at the tail of the FIFO queue.
  - When a page replacement is needed, the page at the head of the FIFO queue is selected.



Note: FIFO queue is not shown here.

FIFO Page Replacement (cont'd)

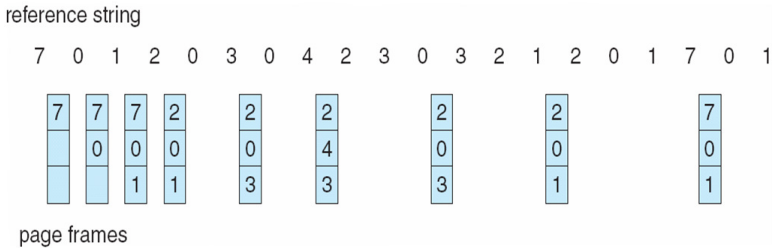
- For a reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- **Belady's Anomaly**: For some page replacement algorithms, the page fault rate may increase as the number of allocated frames increases.
  - This happens when the replacement algorithm cannot fully utilize the additional frame allocated (for certain reference string).

Optimal Page Replacement Algorithm (denoted by OPT or MIN)

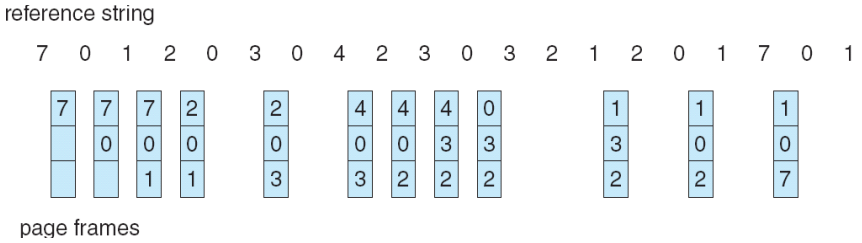
- OPT replaces the page that will not be used (i.e., referenced) for the longest period of time (in the future).
- OPT results in the minimum page fault rate, but requires the future reference string in advance.
  - So, OPT is not a feasible algorithm.
  - OPT is used as a reference in evaluating the performance of other feasible replacement algorithms.



27

Least Recently Used (LRU) Page Replacement Algorithm

- LRU page replacement algorithm select the virtual page that has not been used (i.e., referenced) for the longest period of time.



28

Implementation of Least Recently Used (LRU) Algorithm

- Implementation of LRU using counters.
  - We associate a *time-of-use field* with each page table entry.
  - A logical clock (i.e., a counter) is maintained in the system and the logical clock is incremented for every memory reference (as a new time-stamp value).
  - Whenever a virtual page is referenced, the value of the logical clock is copied to the time-of-use field of the page.
    - So, instead of strong the actual reference time, we store the counter value.
  - When a page needs to be replaced, we replace the page with smallest time-of-use value.

29

Implementation of Least Recently Used (LRU) Algorithm (cont'd)

- Implementation using a stack (called LRU stack).
  - Maintains a stack storing the virtual page numbers (i.e., ids).
  - When a virtual page is referenced, its number is moved to the top (entry) of the stack, pushing down the intermediate entries in the stack.
  - So if a virtual page in main memory is not referenced, its number in the LRU stack is moved down by one entry each time some other virtual page is referenced.
  - When a page replacement is needed, the virtual page whose number is stored at the bottom (entry) of the stack is selected.

30

LRU Approximation Algorithm Using Reference Bits

- Using a reference bit:
  - A reference bit (initially = 0) is associated with each virtual page (e.g., in each page table entry).
  - When a virtual page is referenced, its reference bit is set to 1.
  - Replace a virtual page whose reference bit is 0 (if one exists).
  - But we do not know the exact order the pages were used (i.e., referenced).

31

LRU Approximation Algorithm Using Reference Bits (cont'd)

- Additional-Reference-Bits Algorithm:
  - Each virtual page is associated with a reference bit.
  - We keep additional 8-bit byte for each page (in a table) in memory.
  - At regular time intervals (for example, every 100 msec), the OS shifts the reference bit of each virtual page into the leftmost bit of its 8-bit byte — shifting the other bits to right by one bit — and then reset the reference bit to 0.
    - This 8-bit byte of each page represents the use of the page for the last eight time intervals.
  - If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number (in its 8-bit byte) is the LRU page (among the ones whose current reference bit is 0).

32

Second-Chance Algorithm

- Second-chance algorithm is based on the FIFO replacement algorithm.
  - Each virtual page is associated with a reference bit. When a virtual page is referenced, its reference bit is set to 1.
  - When a page replacement is required, we inspect the reference bit of the page at the head of the FIFO queue.
  - If the reference bit is 0, we replace the page.
  - If the reference bit is 1, we give the page a second chance:
    - Set the reference bit to 0.
    - Move the page to the tail of the FIFO queue.
    - Select the next page (at the head the FIFO queue) and follow the same rule described above.

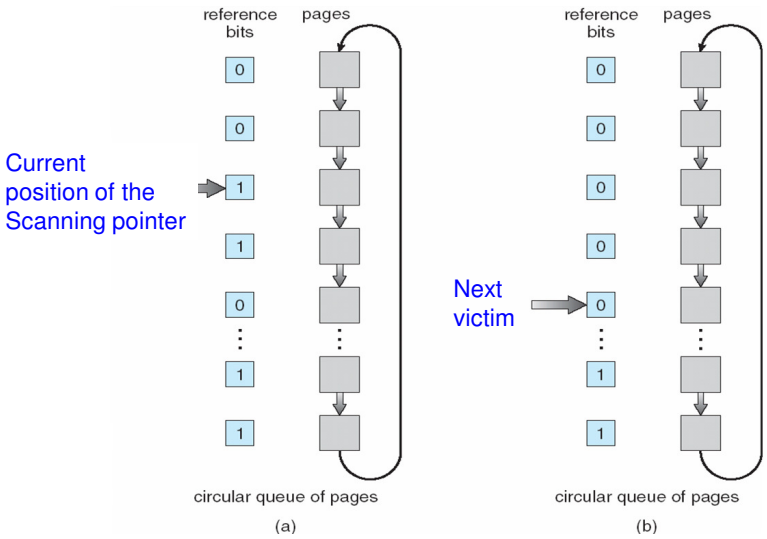
33

Clock (Page Replacement) Algorithm

- Clock algorithm is an implementation of the Second Chance algorithm by using a (logical) circular queue with *N* entries, where *N* is the number of virtual pages the process is allowed to keep in the main memory.
- Each entry can store a virtual page number, and each virtual page is associated with a reference bit (initialized to 0).
  - When a virtual page is referenced, its reference bit is set to 1.
  - A scanning pointer references the next entry to be checked for possible replacement, and initially it points to the entry whose number is 0.
  - On a page fault, the entries in the queue are scanned in a cyclic manner, starting from the entry currently referenced by the scanning pointer. If a the reference bit of the virtual page stored in the entry (pointed to by the scanning pointer) is 1, it is skipped (i.e., not replaced) but its reference bit is reset to 0.
  - The first virtual page number (in the cyclic scanning order) whose reference bit is 0 would be replaced.

34

Clock (Page Replacement) Algorithm (cont'd)



35

Counting Algorithms

- We keep a counter for each page to count the number of references that have been made to the page.
- **Least Frequently Used (LFU) Algorithm**: replaces the page with the smallest reference count.
  - LFU is based on the argument that the pages with small reference counts were brought into memory recently, so they have a good chance to be referenced in the near future.
- **Most Frequently Used (MFU) Algorithm**: replaces the page with the largest reference count.
  - Disfavors recently referenced pages.

36

Page-Buffering Algorithms

Page-buffering algorithms are often used in addition to a specific page replacement algorithm.

- **Keep a pool of free frames**. When a page fault occurs, a victim page is selected for replacement. However, the desired page is read into a free frame from the pool before the victim page is written out.
  - This procedure allows the process to restart as soon as possible. When the victim page is written out later, its frame is added to the pool of free frames.
- Keep a pool of free frames but **remember which page is still stored in each free frame.** The page still stored in a free frame can be used directly (if it is needed) before that frame is loaded with another page.
- **Maintain a list of modified pages.** Whenever the paging device is idle, a modified page is selected and written to disk. Its *modified-bit* (aka *dirty bit*) is then reset.
  - Later, if this page is selected for replacement, it doesn't need to be written out (i.e., transferred) to disk.

37

Allocation of Frames to Processes

- Each process needs a *minimum* number of frames.
  - As the number of frames allocated to each process decreases, the page-fault rate increases, slowing down the process execution.
- When a page fault occurs during the execution of an instruction, the instruction must be restarted. Thus, we must have enough frames to hold all the different pages that any single instruction can reference. For example,
  - if an instruction references one memory address, we need at least one frame for the instruction and one frame for the memory reference.
  - if one-level indirect addressing is allowed (for example, a `load` instruction on page 16 refers to an address on page 20, which is an indirect reference to page 23), then at least 3 frames are required per process.
- Two major memory allocation schemes:
  - Fixed allocation: each process is allocated a fixed number of frames.
  - Variable allocation: each process is allocated a variable number of frames.

38

Fixed Allocation of Memory Frames

- Equal allocation: Every process is allocated the same number of frames.
  - For example, if there are 93 frames and 5 processes, each process is allocated 18 frames. The 3 leftover frames can be used as a pool of free frames.
- Proportional allocation: Allocate frames to a process according to its (relative) virtual memory size.
  - The priority of a process, in addition to its size, can be also considered in this scheme.

39

Global vs. Local Page Replacement

1. **Local replacement**: replaces one of the virtual pages (in main memory) of the process that caused the page fault.
   - Local replacement doesn't allow a process to utilize less-used frames allocated to some other processes.
2. **Global replacement**: replaces a virtual page (in main memory) that belongs to any process.
   - For example, if a process requires a page replacement, we replace one of its own pages or a page of any lower-priority process.
   - One problem with the global replacement is that a process cannot control its own page-fault rate, so its execution times could be quite different in different runs.
   - To use global replacement, variable allocation should be used.

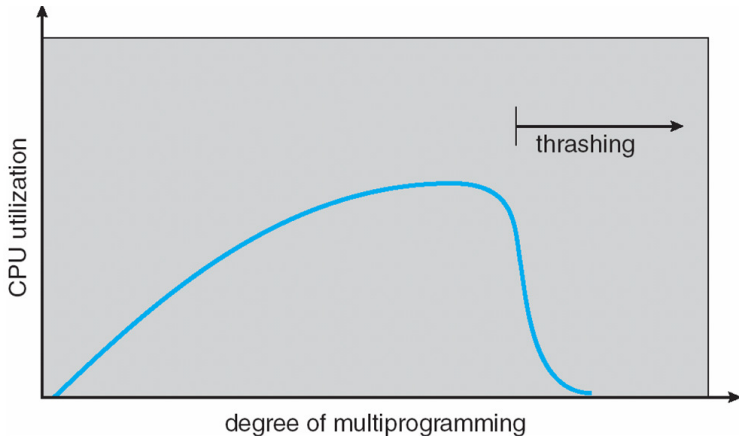
40

Thrashing

- If a process does not have an enough number of frames to store all the pages in active use, its page-fault rate is very high and it may replace a page that will be needed again right away. This high paging activity is called thrashing.
  - A process is thrashing if it is spending more time swapping pages (in and out) than executing.
- Thrashing happens when the *degree of multiprogramming* (i.e, the number of processes sharing the main memory) is too high, such that some processes don't have enough pages in main memory.
  - This leads to a low CPU utilization due to frequent page faults (thereby frequent context switching).

41

Thrashing (cont'd)



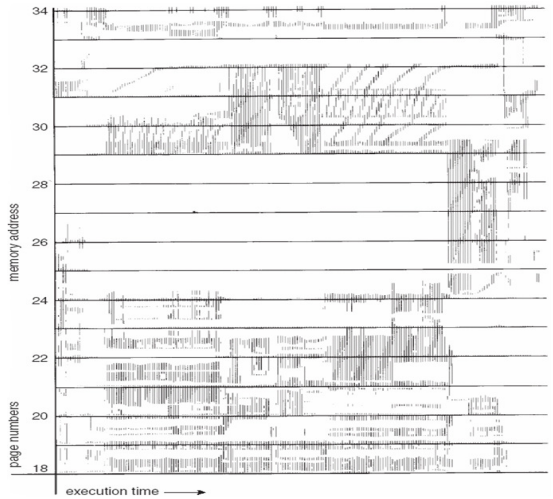
42

Locality Model of a Process

- A ***locality*** *of a process* is a set of its pages that are actively used together.
- The locality model: As a process executes, it moves from one locality to another locality.
  - Generally, a process has several different localities, which may overlap.
  - We also use the term ***Locality of Reference*** to represent the fact that the reference pattern of a typical process is not random (in its logical address space) during a short execution time, instead it is localized.

43

Locality in a Reference Pattern (or simply, Locality of Reference)



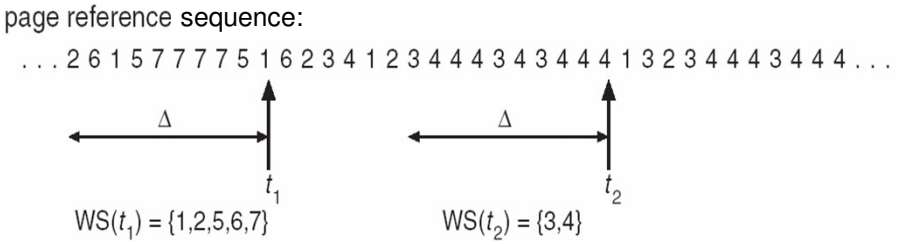
44

Working-Set Model

- For a process, the set of pages referenced during a time interval Δ is called the working-set of the process during Δ.
- Δ is called **working-set window**, which is a fixed time interval or a fixed number of (page) references (made by the process).
  For example, Δ =10,000 references.
  - If Δ is adequate, the working-set is a good approximation of the locality (at that time).
  - If Δ is too small, it may not encompass the entire locality.
  - If Δ is too large, it may encompass multiple localities.
- In a system using the working-set model, we keep track of the working-set of each process.

45

Example Working-Sets



Δ = 10 references

46

Working-Set Model (cont'd)

- If the current (i.e., most recent) working-set of a process is kept in memory, the page-fault rate of the process would be low.
- WSS_i : working-set size of Process i
  = total number of pages in the working-set of Process i.
- $D = \Sigma_i WSS_i$: total demand of all the processes for frames.
  - Suppose that m is the total number of frames in main memory. If $D > m$, thrashing will occur, because some process will not have an enough number of frames.
    - In this case, OS may suspend a process, so that its pages are swapped out and the freed frames are allocated to other processes.

47

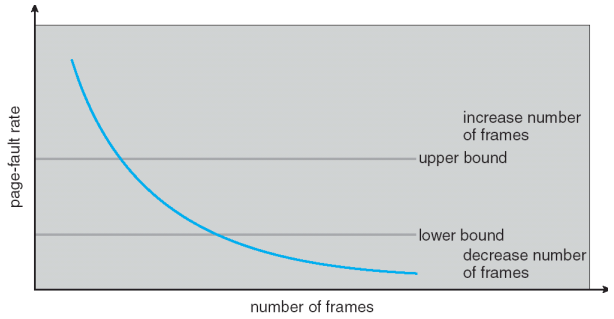
Keeping Track of the Working-Set

- Keeping track of the working-set is difficult because the working-set window is a moving window.
- We can approximate the working-set using a timer and a reference bit for each page.
- Example: Δ = 10,000 references
  - A timer interrupts at every 5000 references.
  - Whenever a timer interrupts, copy (i.e., save) all reference bits and reset them to 0.
  - Keep in memory the last 2 values of the reference bit of each page.
  - If one of the last two values of the reference bit of a page is 1, the page is included in the working set.
- To increase the accuracy of working set, we can set the timer to interrupt more frequently and save more previous values of the reference bit of each page.
  - For example, a timer interrupts at every 1000 references, and we save last 10 values of the reference bit of each page. Then, we can update the working set of the process more frequently (i.e., at every 1000 references).

48

Avoid Thrashing Using Page-Fault Frequency (PFF) of a Process

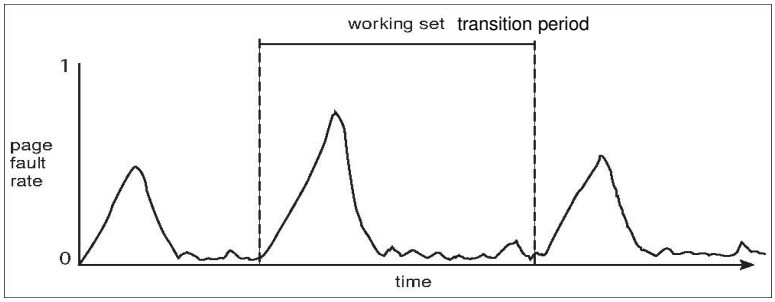
- Instead of using the working-set of each process to control the number of frames allocated to it, we can use its page-fault rate.
- We can set upper and lower bounds on the page-fault rate for a process:
  - If the actual page-fault rate exceeds the upper limit, we allocate another frame to the process.
  - If actual page-fault falls below the lower limit, we remove a frame from the process.



49

Relationship between the Transition in Working-Set and Page-Fault Rates

- The page-fault rate starts a peak when a process begins the demand-paging a new locality. However, once the new locality is brought into memory, the page-fault-rate falls.
- The span of the time between the start of one peak in page-fault rate and the start of the next peak represents the transition from one working-set to another.



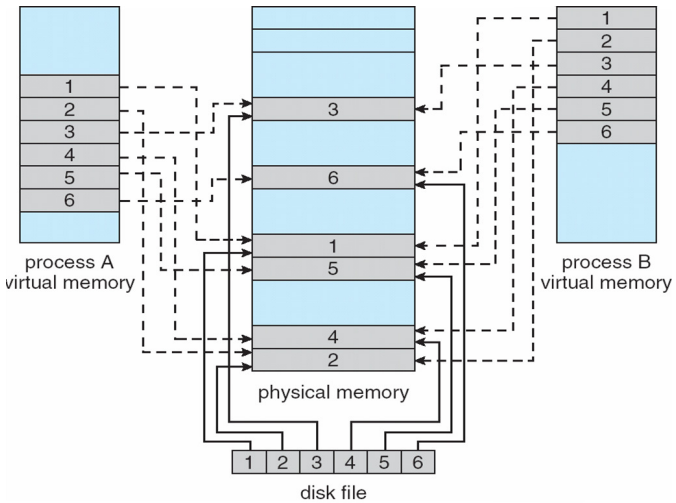
50

Memory-Mapped Files

- Memory-mapping a file allows the file I/O to be treated as routine memory accesses by mapping a file to pages in the virtual memory.
- Simplifies file access by handling file I/O through memory access rather than **`read()`** and **`write()`** system calls, so that the file access becomes much faster.
- Initially each page-sized portion of the file is read from the file system into a physical page (based on demand paging). Subsequent reads and writes on the pages are treated as ordinary memory accesses for data words, without involving file accesses.
  - When the file is closed, memory mapped pages of the file that have been modified will be written back to disk and removed from the virtual memory of the process.
- In some systems (e.g., Windows systems), multiple processes are allowed to map the same file into their virtual address spaces, so the pages of the file can be shared between processes.

51

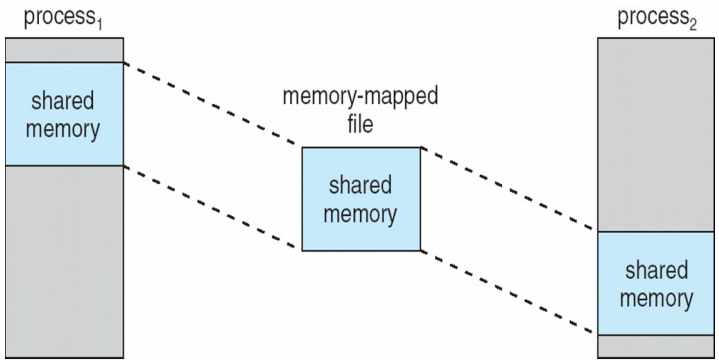
Memory Mapped Files



52

Shared Memory in Windows Using a Memory-Mapped File

- Actually, shared memory can be implemented by creating a memory-mapped file.



53

Prepaging (aka Prefetching of Pages)

- We can prepage (i.e., prefetch) all or some of the pages a process may need, before they are referenced (i.e., demanded) by the process.
  - Some OSs (e.g, Solaris) prepage all the pages of small files.
  - In a system using the working-set model, we keep track of the working-set of each process.
    - If a process is suspended (due to an I/O wait or lack of free frames for some other processes) and swapped out, we remember the working-set of the process.
    - When the process is resumed, we bring back the entire working-set of the process into memory before restarting the process.
- Prepaging is to reduce the large number of page faults that occurs when a process is started or a swapped-out process is resumed.
- But if some prepaged pages are not used, their prepaging time and memory space are wasted.

54

One Block Look-ahead (OBL) Prefetching Algorithm

- When a process has a page fault on a virtual page (number) *i* ; that means, the process reference a word in virtual page *i* and page *i* is not in main memory:
  - Virtual page *i* is fetched from disk to main memory, and
  - virtual page (*i* +1) is prefetched from disk to main memory (if it is not in main memory), expecting page (*i* +1) will be referenced by the process in the near future (based on the locality of reference).
- How OBL can be implemented with the LRU page replacement algorithm?
  - When OBL is used together with LRU replacement algorithm, the demanded page number is placed at the top of the LRU stack, whereas the prefetched page number is placed at the bottom of the LRU stack.
  - If page replacement is required for the demanded page or the prefetched page, the LRU page is replaced (for each).

55

Allocating Kernel Memory

- Allocating kernel memory (to kernel processes) is treated differently from allocating user memory (to user processes).
- Kernel memory is often allocated from a free-memory pool, different from the list of free frames used for user processes, because
  - The kernel data structures are fixed, but some of which are less or larger than a page size.
  - Some hardware devices interact directly with physical memory and may require contiguous memory space (as a buffer).
  - Many OSs do not subject kernel code and data to the paging system.

56

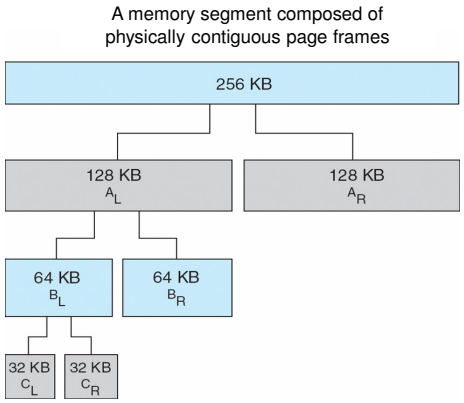
Buddy System

- Allocates memory from a fixed-size segment consisting of physically contiguous page frames. Memory is allocated from this segment using a **power-of-2 allocator:**
  - Satisfies requests in units that are sized as a power of 2: 4 KB, 8 KB, 16 KB, etc.
  - If a requested memory size is not in a power of 2, it is rounded up to the next higher power of 2.
  - If a requested memory size is smaller than the half an available memory segment, the segment is divide into two buddies (each one's size is the next lower power of 2), then if necessary, one of the two buddies is divided into two buddies, and so on.
  - If two buddies become free, they can be merged to create a larger memory segment.

57

Buddy System (cont'd)

- A drawback of buddy system is that rounding up to the next higher power of 2 is very likely to cause an internal fragmentation within the allocated segment.
  - For example, 64 KB would be allocated when 33 KB is requested.



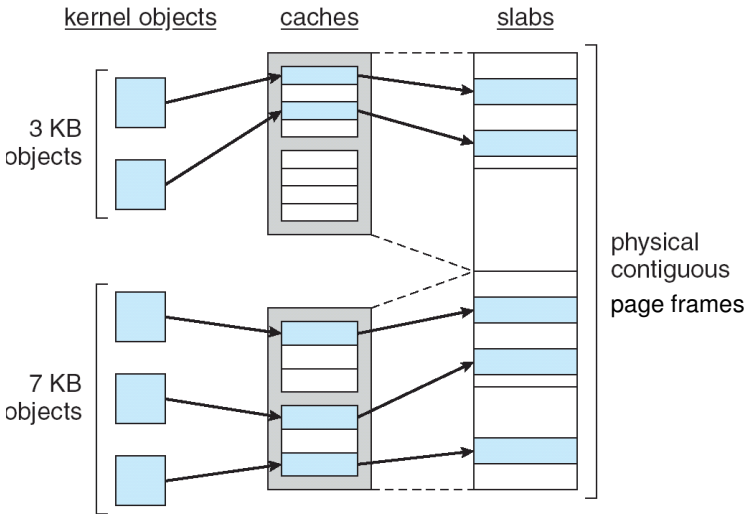
58

Slab Allocation

- A **slab** is made up of one or more physically contiguous page frames.
- A **cache** consists of one or more slabs.
- There is **a single cache for each unique kernel data structure**:
  - For example, a separate cache for the data structure of process descriptors (aka PCBs), a separate cache for semaphores, a separate cache for file control blocks, etc.
- Each cache is populated with **objects** that are instances of the kernel data structure the cache represents.
  - For example, the cache for semaphores stores the semaphore objects.
- The slab allocation algorithm uses caches to store kernel objects. When a cache is created, it is filled with objects marked as ***free.***
- When a new object for a kernel data structure is needed, any free object in the corresponding cache is assigned, and the assigned object is marked as ***used.***

59

Slab Allocation (cont'd)



A slab may store multiple kernel objects (depends on the slab size and the object size).

60

Slab Allocation (cont'd)

- In Linux, a slab may be in one of three possible states:
  - Full: all objects in the slab are marked as used.
  - Empty: all objects in the slab are marked as free.
  - Partial: the slab contains both used and free objects.
- The slab allocator first attempts to satisfy the memory request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slab is available, a new slab is allocated and assigned to the cache.
- Two benefits of slab allocation:
  - **No big internal fragmentation because each slab is divided into objects of specific size.**
  - **Memory requests are satisfied quickly because free objects are created in advance,** and the objects released by kernel are marked as free objects.

61

Page Size Selection

- Page size selection is based on the following factors:
  - Internal fragmentation of the last page of each process: on average, half of the last page is empty.
  - Page table size: if the page size is doubled, the page table size is reduced to a half.
  - Disk I/O overhead to transfer a page: the sum of *seek time* and *rotational latency* is a fixed overhead of a disk drive for each page access, and it is much bigger than the one page transfer time between a disk drive and memory.
  - Locality of a process: a larger page can capture more (spatial) locality by storing more adjacent words.
  - Disk sector size (e.g., 0.5 KB, 1 KB, 2 KB, 4 KB): a sector is the minimum storage unit on a disk drive.
- The trend is using larger page sizes.

62

TLB Reach

- TLB Reach: The amount of memory accessible via the TLB.
  - TLB Reach = (the number of entries in TLB) x (Page Size)
- Ideally, the working set of a process is stored in the TLB.
  - Because the page table in memory should be looked up at each TLB miss (i.e., referenced virtual page number and its memory frame number are not in TLB).
- How to increase the TLB Reach:
  - Increase the page size: This may lead to an increase in internal fragmentation because not all applications require a large page size.
  - Provide multiple page sizes: This allows applications that require larger page sizes the opportunity to use them without increasing the internal fragmentation.
    - For example, IA-32 architecture supports 4 KB and 4 MB pages; and x86-64 architecture supports 4 KB, 2 MB, and 1 GB pages.
    - In this case, we need to store the page size in each entry of the TLB, and the TLB should be managed by software.

63

Program Structure

- Program data structure:
  - `int[128,128] data;`
  - Assume the array `data` is stored in **row-major order** (e.g., in C/C++, Python), and the page size is 128 words, such that each row is stored in a page.
- Program 1:

```
for (j = 0; j <128; j++)
     for (i = 0; i < 128; i++)
          data[i,j] = 0;
```

  If OS allocates fewer than 128 page frames to this program, then its execution will result in 128 x 128 = 16,384 page faults.

- Program 2:

```
for (i = 0; i < 128; i++)
     for (j = 0; j < 128; j++)
          data[i,j] = 0;
```

  The execution of this program will result in 128 page faults.

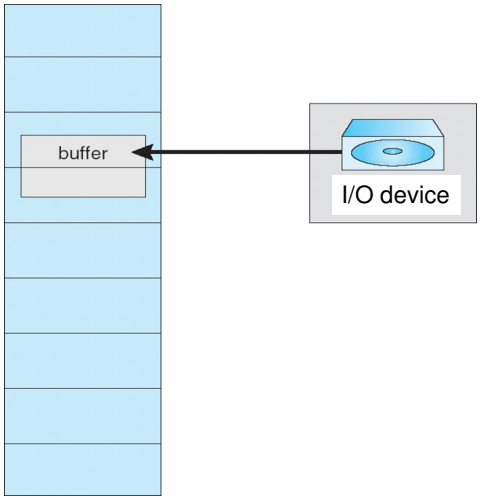
64

Locking Pages in Memory

- When demand paging is used, sometimes we need to allow some pages to be locked in memory so that they will not be replaced.
- Generally, an I/O device controller (e.g, USB storage device controller) is given the number of bytes to be transferred and the memory address for the buffer.
  - Page frames that are used for reading data from an I/O device must be locked, so that they will not be selected for replacement by a page replacement algorithm (before they are used by its application program).
- A **lock bit** is associated with each frame.
  - If the lock bit is set for a frame, the page stored in the frame cannot be selected for replacement.

65

The Reason Why Frames Used For I/O Must be Locked in Memory



66

Operating System Examples

- Windows
- Solaris

67

Windows

- Uses demand paging with **clustering**.
  - Clustering handles page faults by bringing in not only the faulting (i.e., missing) page but also several pages following the faulting page.
- When a process is created, it is assigned a **working-set minimum** and a **working-set maximum.**
  - The working-set minimum is the minimum number of pages the process is guaranteed to have in memory.
  - The working-set maximum is the maximum number of pages the process can have in memory.
  - **If a process that is at its working-set maximum incurs a page fault, it must replace a page using a local replacement policy (e.g., local LRU).**
- When the amount of free memory in the system falls below a threshold, **automatic working-set trimming** is performed to restore the amount of free memory.
  - Automatic working-set trimming removes (i.e., swaps out) the pages of processes that have pages in excess of their working-set minimum.

68

Solaris

- Maintains a list of free page frames to be allocated to the processes (when they have page faults).
- Uses three different threshold of free memory:
  - *lotsfree*: threshold to begin the *pageout* process, which swaps out the pages that were not referenced recently.
    - Four times per second, the kernel checks whether the amount of free memory is less than *lotsfee.* If yes, pageout is performed.
    - *lotsfee* is usually set to the greater of 512 KB or 1/64 of the total memory.
  - *Desfree* (desired amount of free memory): threshold to increase the rate of pageout to 100 times per second.
    - Its default value is 256 KB or lostfree/2, whichever is greater.
    - If the pageout process cannnot keep the amount of free memory above *desfree* for a 30-second average, the kernel begins swapping out processes.
  - *minfree*: threshold to begin pageout for every page fault.
    - Its default value is 128 KB or desfree/2, whichever is greater.

69

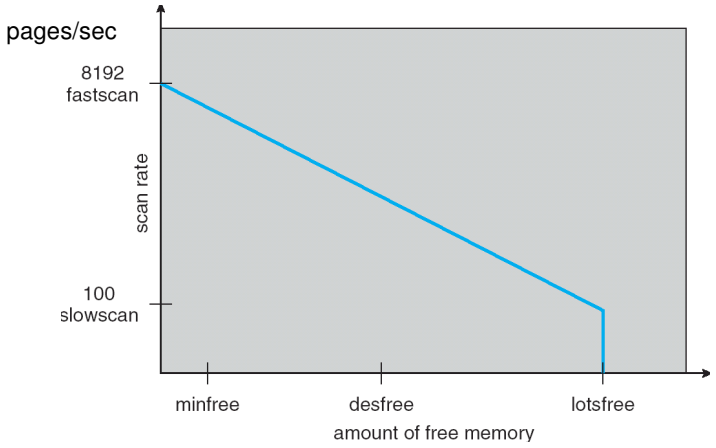
Solaris (cont'd)

- The **pageout process** swaps out the pages that have not been referenced recently.
- The pageout process is similar to the *clock* replacement algorithm but it uses two hands (i.e, pointers) while scanning pages.
  - The *front hand* of the clock scans all pages in memory, setting their reference bits to 0.
  - Later, the *back hand* of the clock examines the reference bits of the pages. If the reference bit of a page is still 0, swaps out the page and appends the page frame to the free frame list.
  - The distance between the two hands of the clock (in terms of a number of pages) is determined by a system parameter, named *handspread.* For example, *handspread* =1000 pages.
  - *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* (100 pages/sec) to *fastscan* (a half of all pages per second, with maximum 8192 pages/sec).

70

Page Scanrate of Solaris

- A higher page scanrate is used as the amount of free memory decreases (below *lotsfree*), in addition to the higher rate of pageout process.



71