ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

OPERATING SYSTEMS & PARALLEL COMPUTING

Virtual Memory management

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Introduction to virtual memory

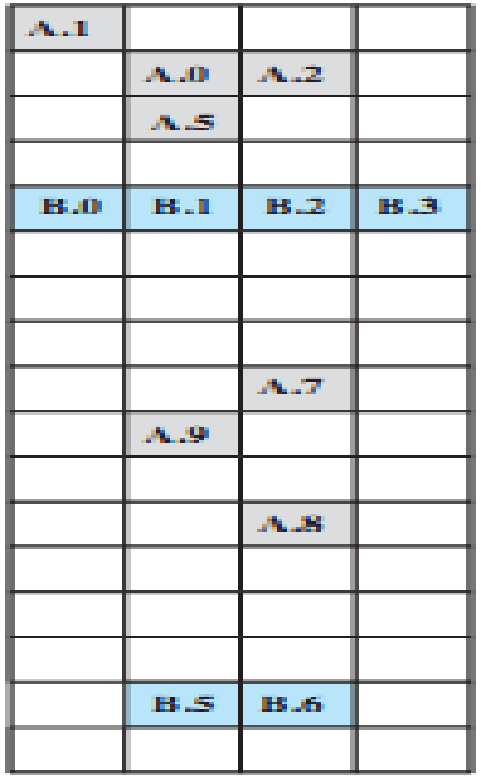
Background (1)

- Code needs to be in memory to execute, but entire program rarely used:
  - Error code, unusual routines, large data structures.
- Entire program code not needed at same time.
- Consider ability to execute partially-loaded program:
  - Program no longer constrained by limits of physical memory.
  - Each program takes less memory while running ->
    more programs run at the same time:
    - Increased CPU utilization and throughput with no increase in response time or turnaround time.
  - Less I/O needed to load or swap programs into memory -> each user program runs faster.

Background (2)

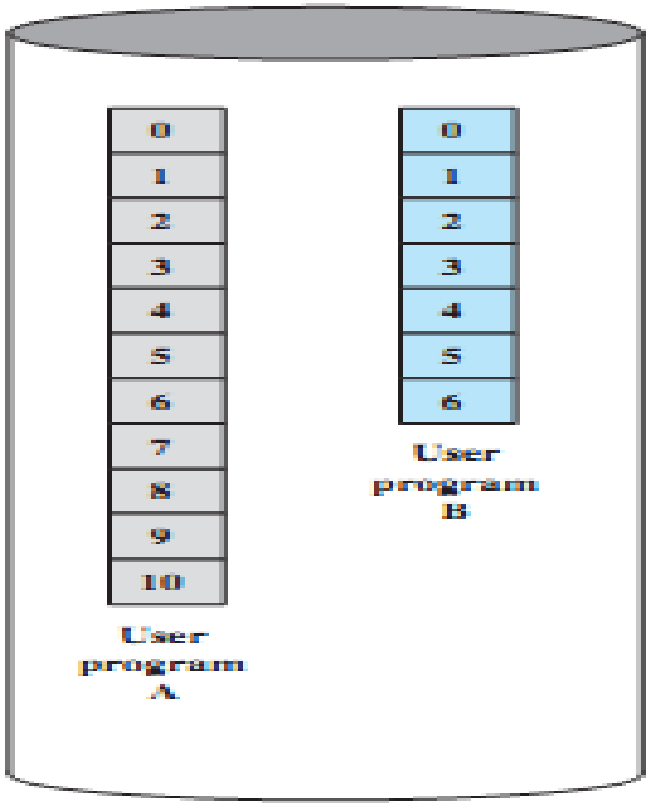
- **Virtual memory** – separation of user logical memory from physical memory:
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.
  - More programs running concurrently.
  - Less I/O needed to load or swap processes.

Virtual Memory Components



Main Memory

Main memory consists of a number of fixed-length frames, each equal to the size of a page. For a program to execute, some or all of its pages must be in main memory.



Disk

Secondary memory (disk) can hold many fixed-length pages. A user program consists of some number of pages. Pages for all programs plus the operating system are on disk, as are files.

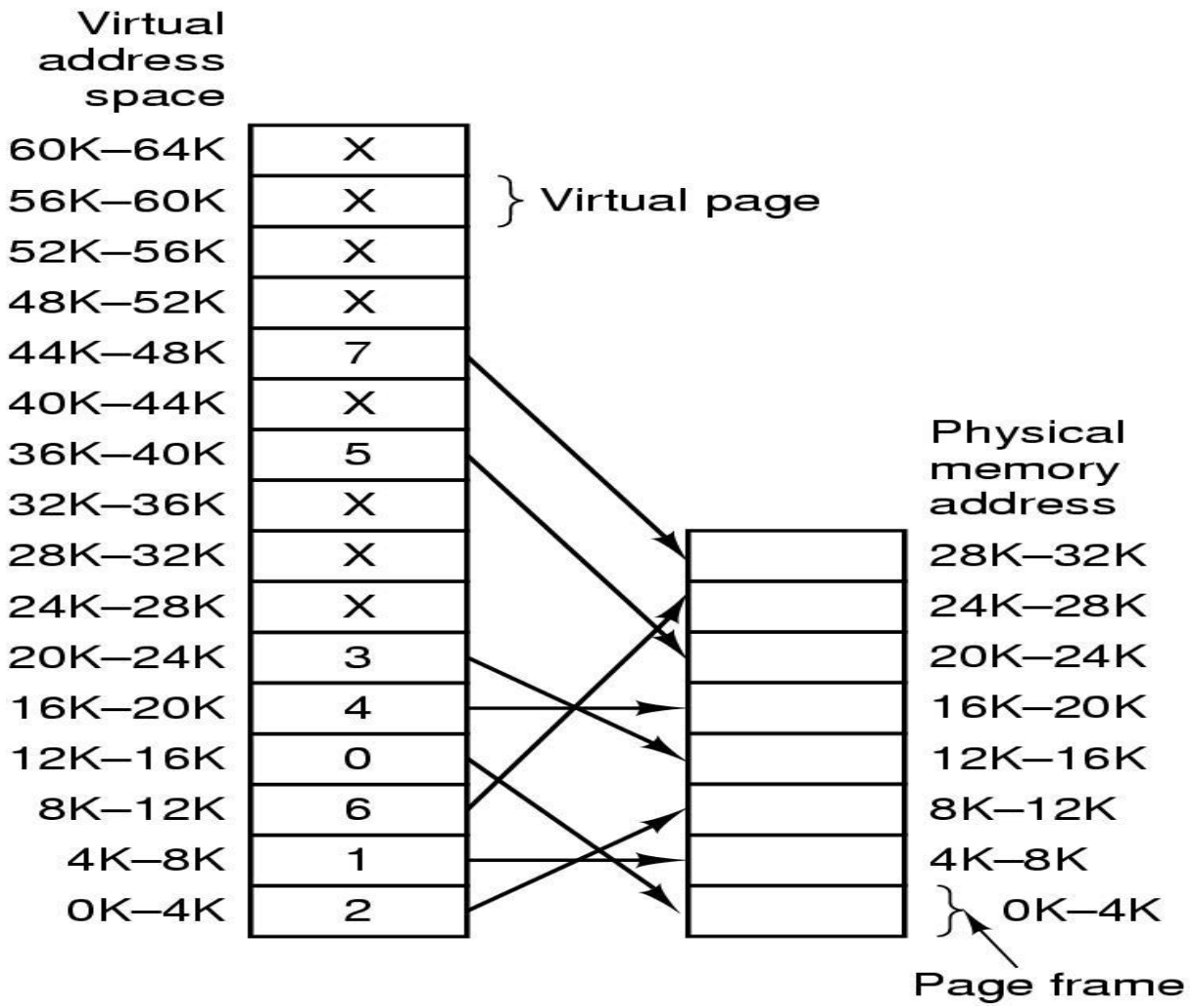
Background (3)

- **Virtual address space** – logical view of how process is stored in memory:
  - Usually start at address 0, contiguous addresses until end of space.
  - Meanwhile, physical memory organized in page frames.
  - MMU must map logical to physical.
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

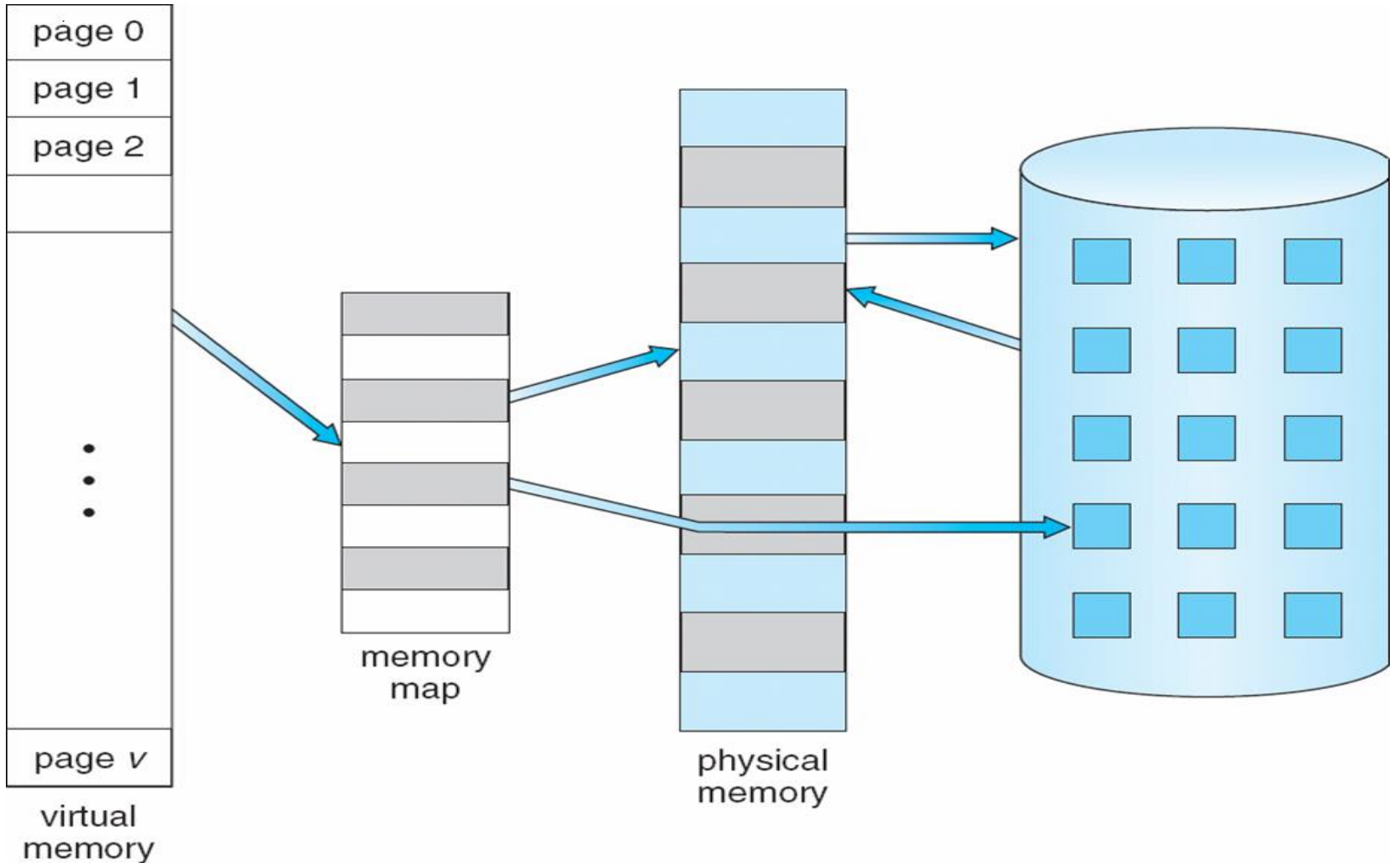
(4) Background

- Based on Paging/Segmentation, a process may be broken up into pieces (pages or segments) that do not need to be located contiguously in main memory.
- Based on the Locality Principle, all pieces of a process do not need to be loaded in main memory during execution; all addresses are virtual.
- The memory referenced by a virtual address is called virtual memory:
  - It is mainly maintained on secondary memory (disk).
  - pieces are brought into main memory only when needed.

Virtual Memory Example



Virtual Memory that is larger than Physical Memory



Advantages of Partial Loading

- More processes can be maintained in main memory:
  - only load in some of the pieces of each process.
  - with more processes in main memory, it is more likely that a process will be in the Ready state at any given time.
- A process can now execute even if it is larger than the main memory size:
  - it is even possible to use more bits for logical addresses than the bits needed for addressing the physical memory.

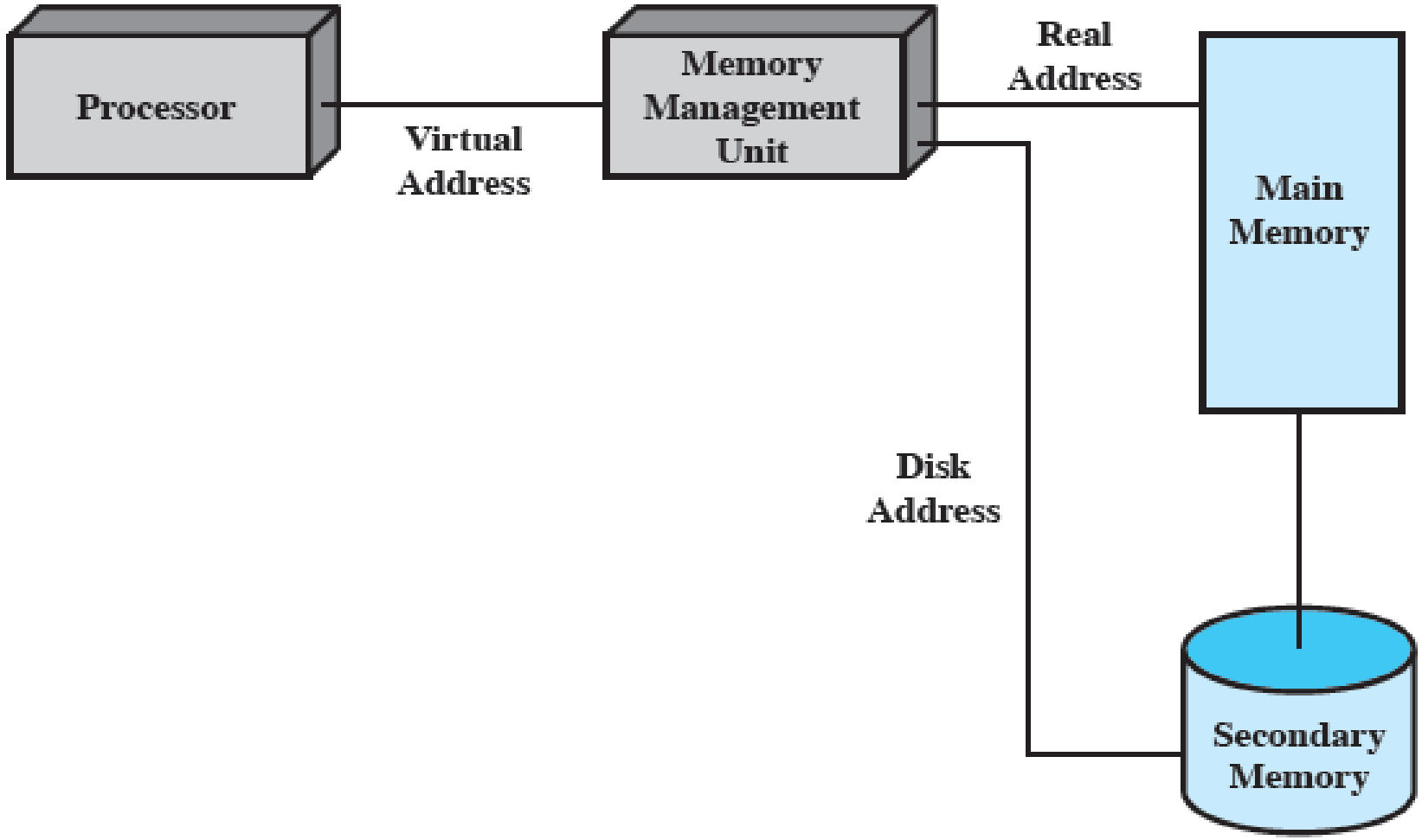
Virtual Memory: Large as you wish!

- Example:
  - Just 16 bits are needed to address a physical memory of 64KB.
  - Lets use a page size of 1KB so that 10 bits are needed for offsets within a page.
  - For the page number part of a logical address we may use a number of bits larger than 6, say 22 (a modest value!!), assuming a 32-bit address.

Support needed for Virtual Memory

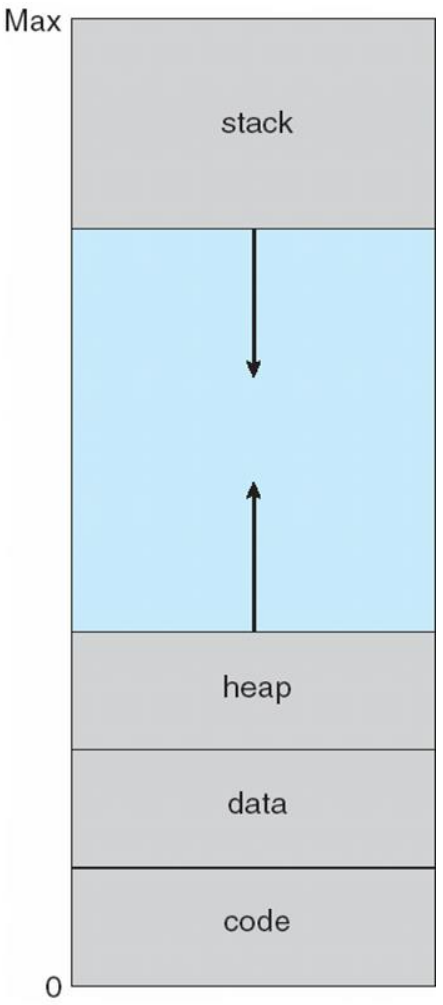
- Memory management hardware must support paging and/or segmentation.
- OS must be able to manage the movement of pages and/or segments between external memory and main memory, including placement and replacement of pages/segments.

Virtual Memory Addressing

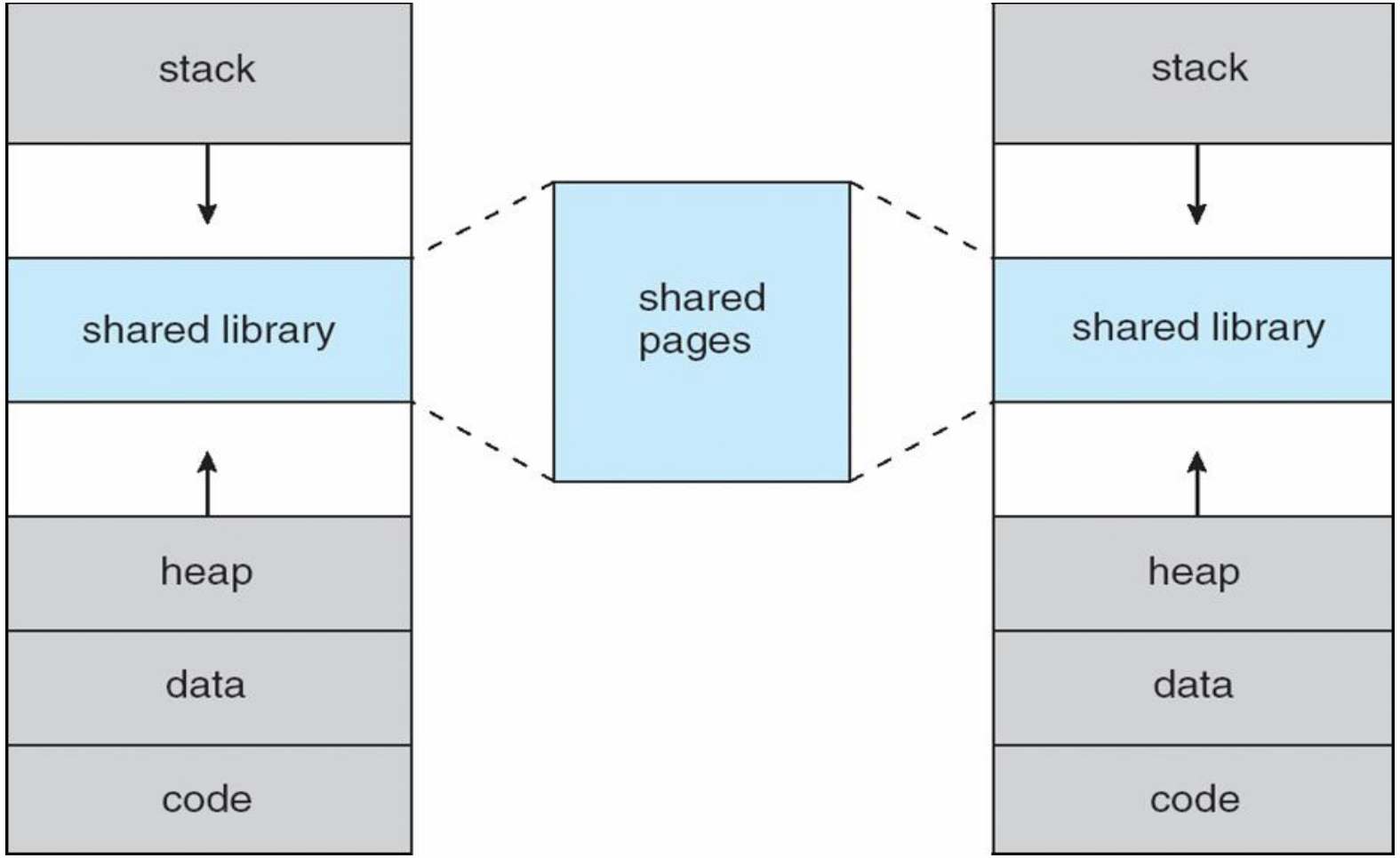


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”:
  - Maximizes address space use.
  - Unused address space between the two is hole.
  - No physical memory needed until heap or stack grows to a given new page.
- Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space.
- Shared memory by mapping pages read-write into virtual address space.
- Pages can be shared during `fork()`, speeding process creation.



Shared Library using Virtual Memory



Process Execution (1)

- The OS brings into main memory only a few pieces of the program (including its starting point).
- Each page/segment table entry has a valid-invalid bit that is set only if the corresponding piece is in main memory.
- The resident set is the portion of the process that is in main memory at some stage.

Process Execution (2)

- An interrupt (memory fault) is generated when the memory reference is on a piece that is not present in main memory.
- OS places the process in a Blocking state.
- OS issues a disk I/O Read request to bring into main memory the piece referenced to.
- Another process is dispatched to run while the disk I/O takes place.
- An interrupt is issued when disk I/O completes; this causes the OS to place the affected process back in the Ready state.

Demand Paging

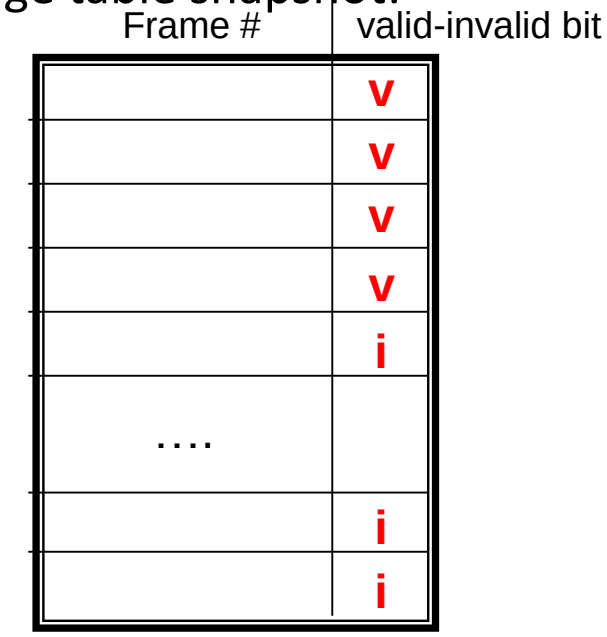
- Bring a page into memory only when it is needed:
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it:
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- Similar to paging system with swapping.
- Lazy swapper – never swaps a page into memory unless page will be needed; Swapper that deals with pages is a pager.

Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again.
- Instead, pager brings in only those pages into memory.
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging.
- If pages needed are already memory resident:
  - No difference from non demand-paging.
- If page needed and not memory resident:
  - Need to detect and load the page into memory from storage:
    - Without changing program behavior.
    - Without programmer needing to change code.

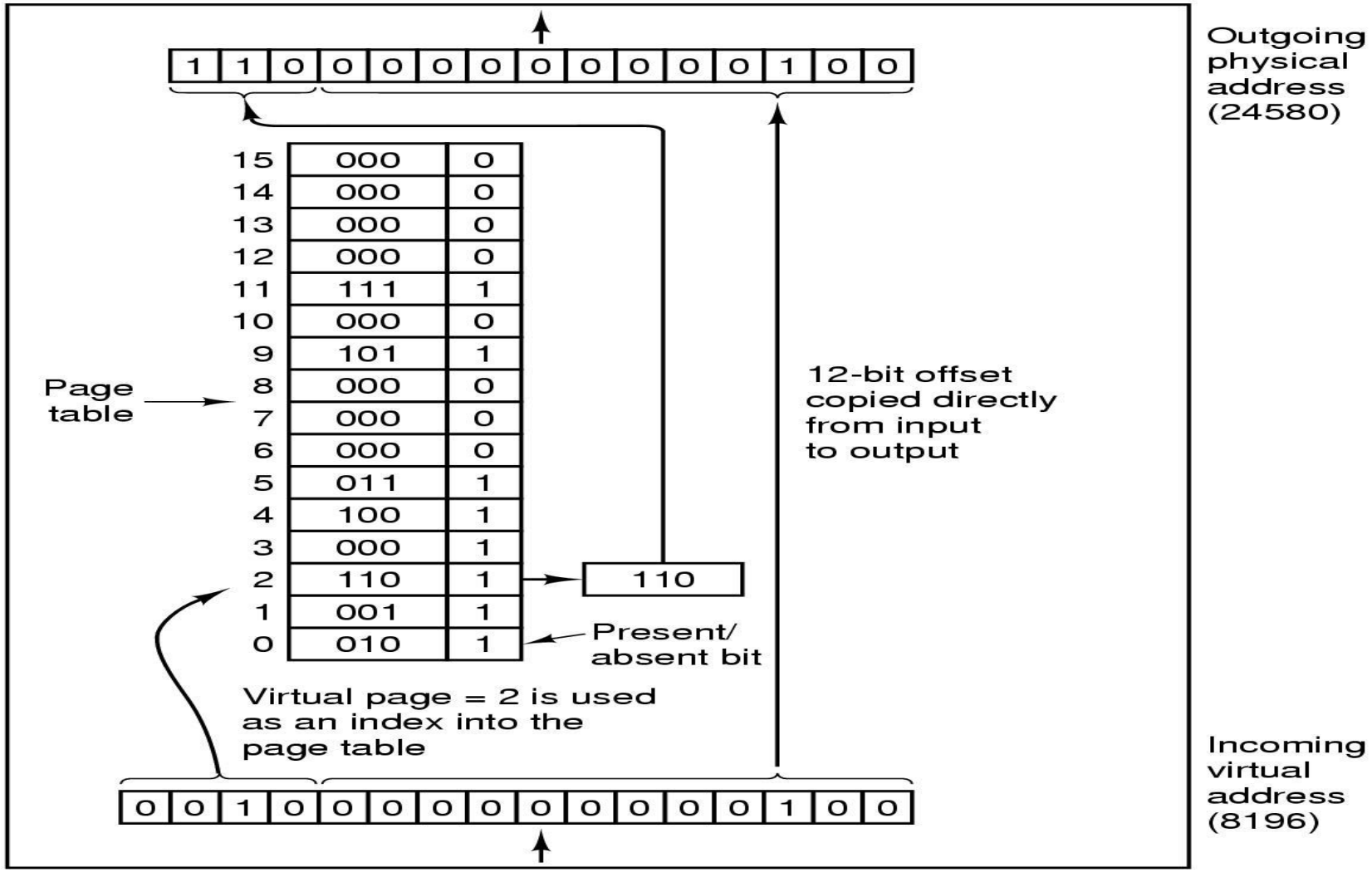
Valid-Invalid Bit

- With each page table entry, a valid–invalid (present-absent) bit is associated (**v** ⇒ in-memory, **i** ⇒ not-in-memory).
- Initially valid–invalid bit is set to **i** on all entries.
- Example of a page table snapshot:

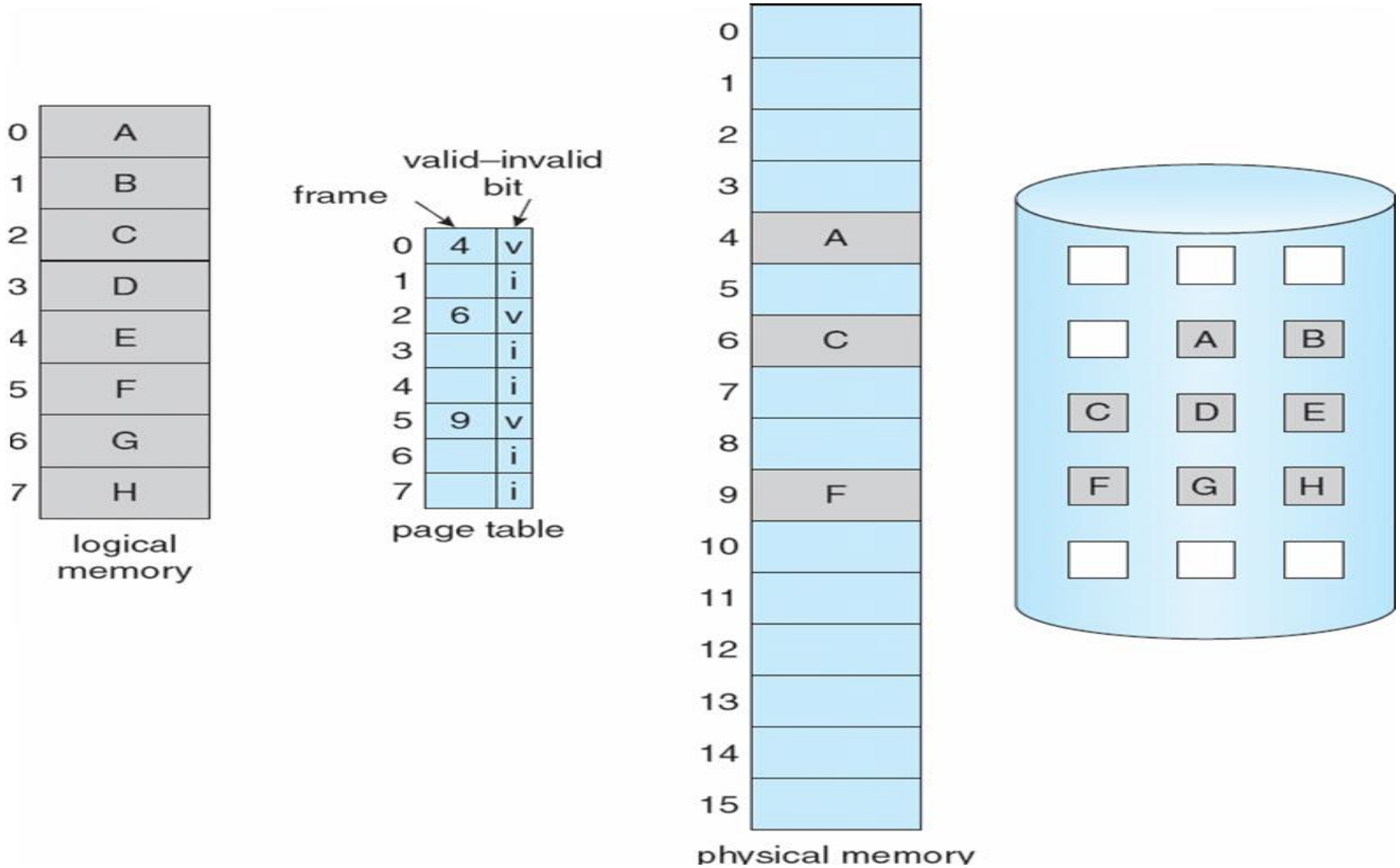


During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault.

Virtual Memory Mapping Example

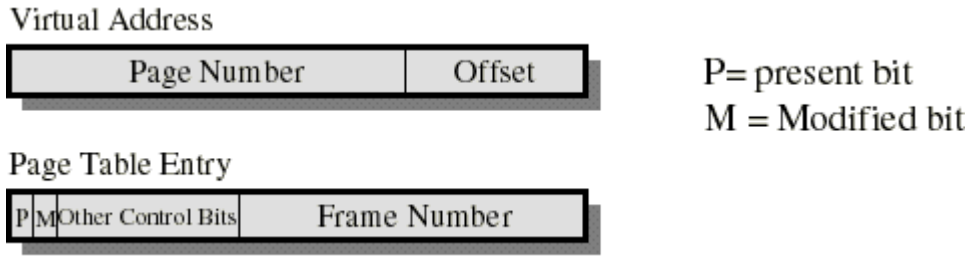


Page Table when some Pages are not in Main Memory



Dynamics of Demand Paging (1)

- Typically, each process has its own page table.

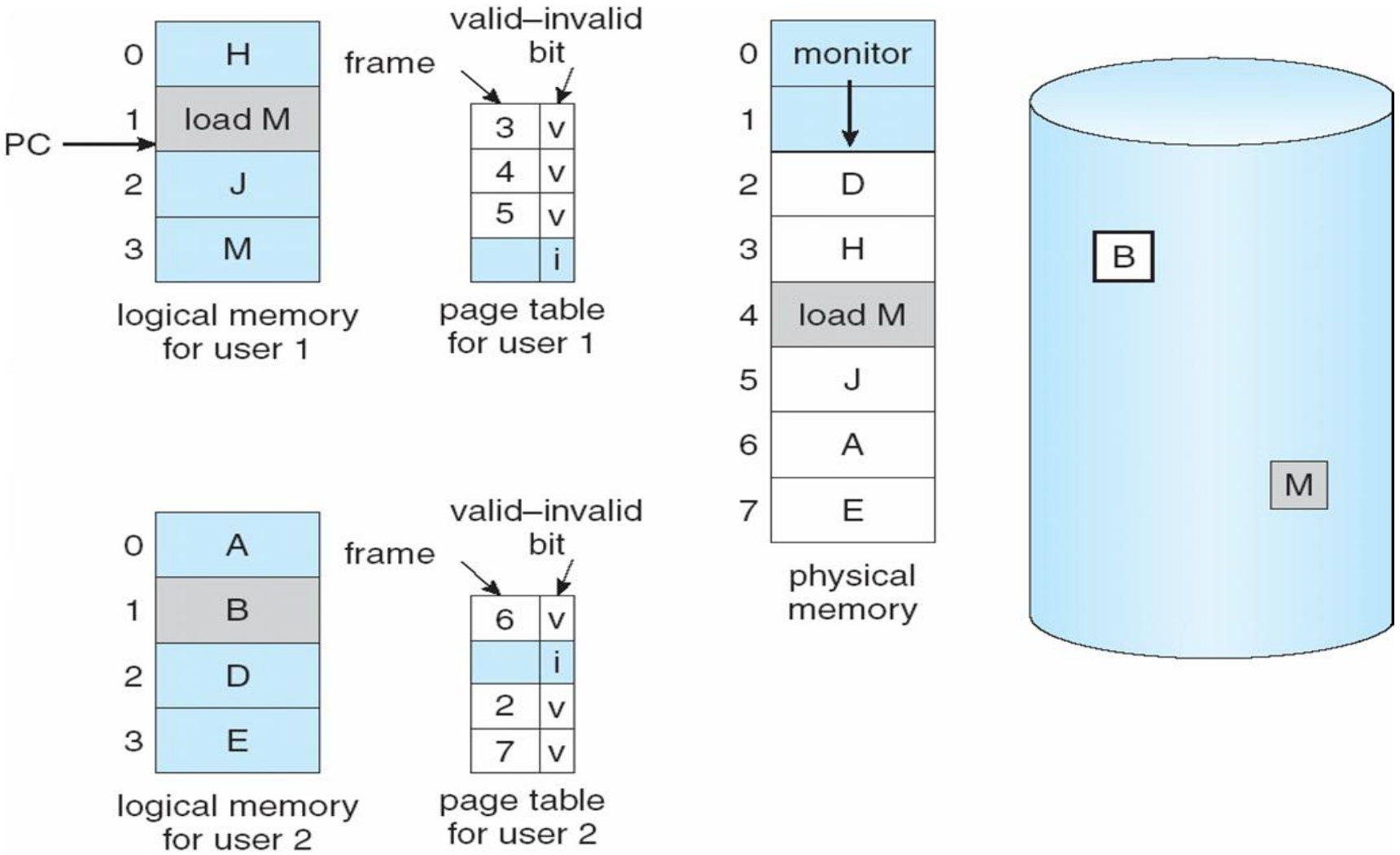


- Each page table entry contains a present (valid-invalid) bit to indicate whether the page is in main memory or not.
  - If it is in main memory, the entry contains the frame number of the corresponding page in main memory.
  - If it is not in main memory, the entry may contain the address of that page on disk or the page number may be used to index another table (often in the PCB) to obtain the address of that page on disk.

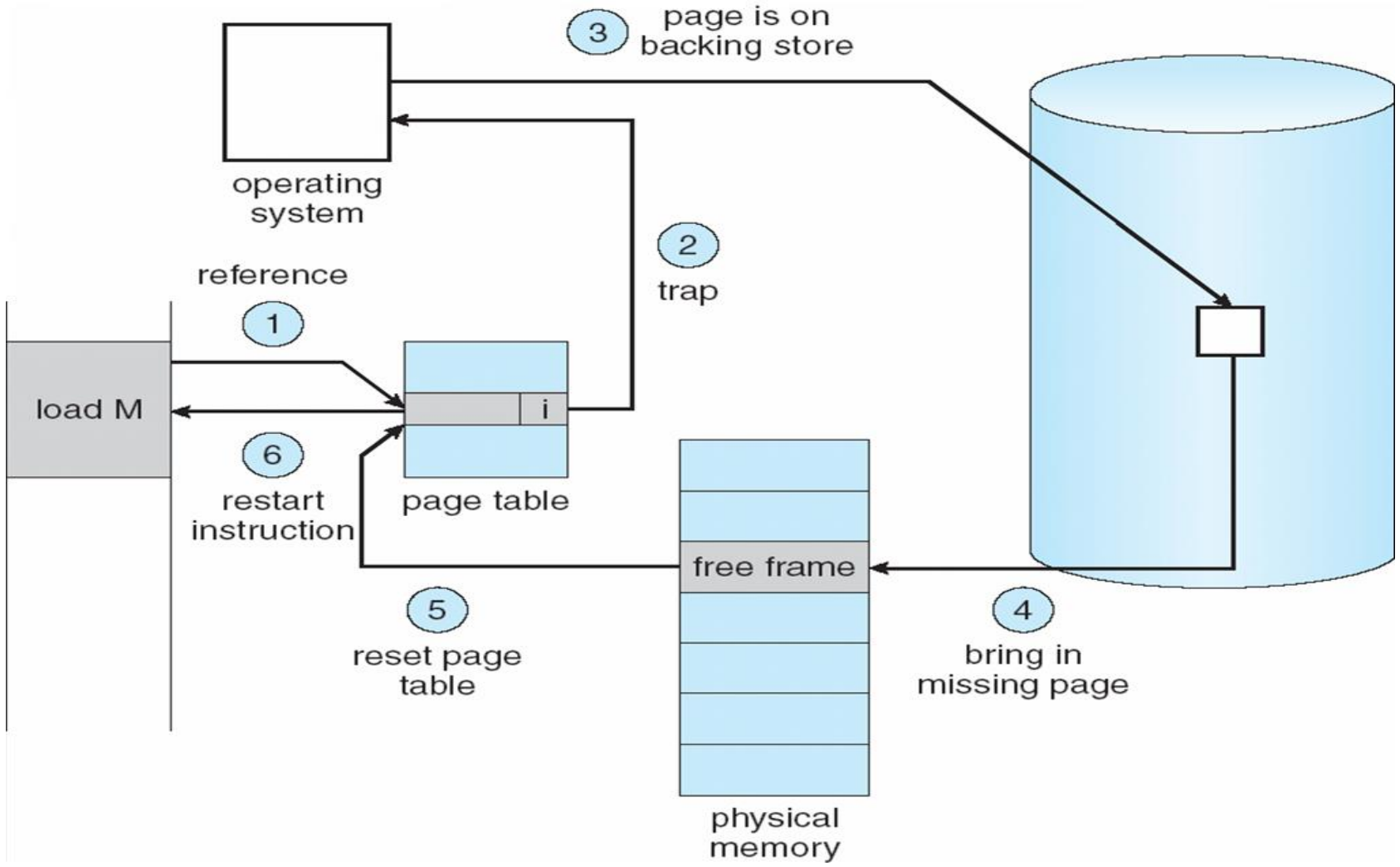
Dynamics of Demand Paging (2)

- A modified bit indicates if the page has been altered since it was last loaded into main memory:
  - If no change has been made, page does not have to be written to the disk when it needs to be swapped out.
- Other control bits may be present if protection is managed at the page level:
  - a read-only/read-write bit.
  - protection level bit: kernel page or user page (more bits are used when the processor supports more than 2 protection levels).

Need For Page Fault/Replacement



Steps in handling a Page Fault (1)



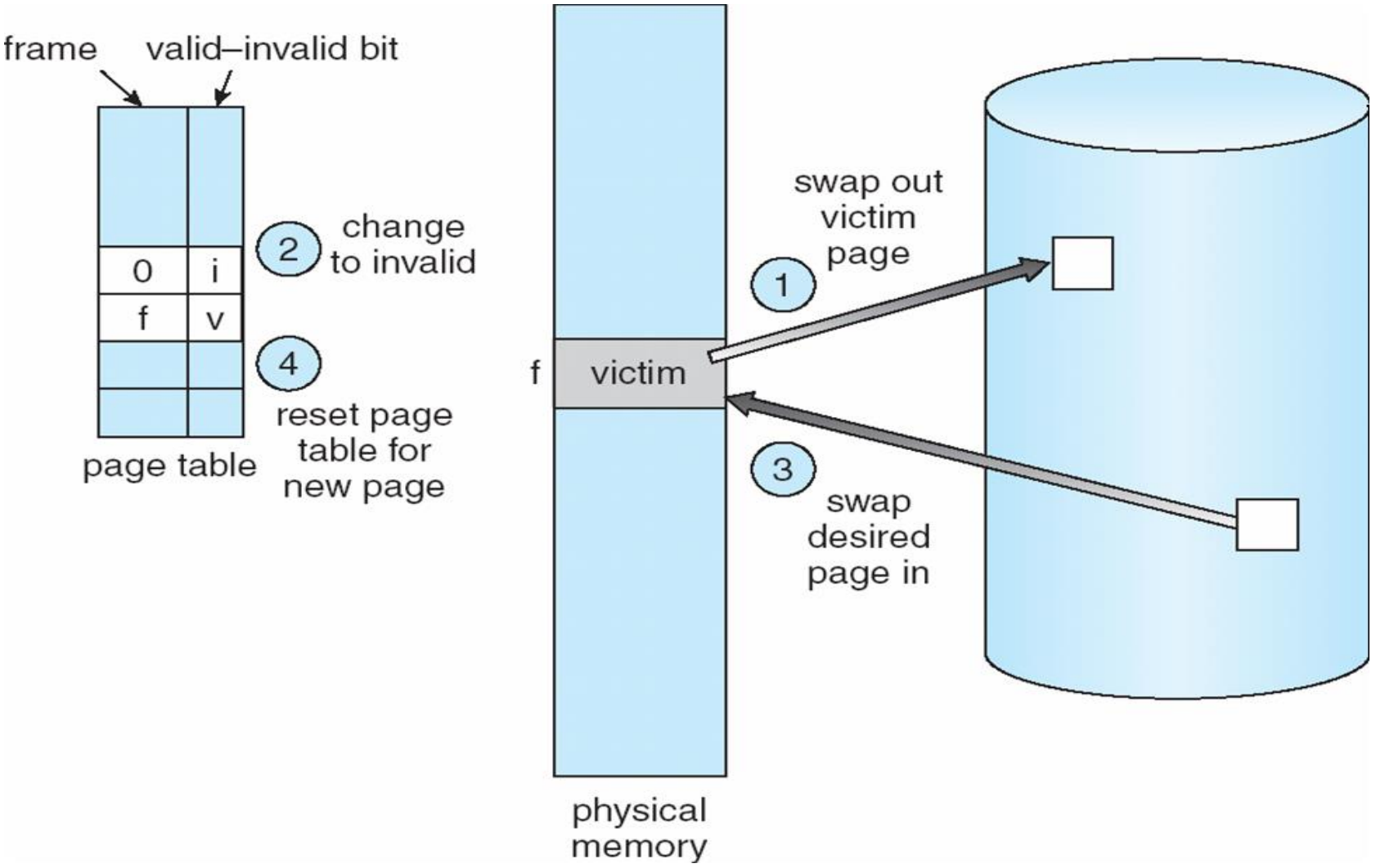
Steps in handling a Page Fault (2)

1. If there is ever a reference to a page not in memory, first reference will cause page fault.
2. Page fault is handled by the appropriate OS service routines.
3. Locate needed page on disk (in file or in backing store).
4. Swap page into free frame (assume available).
5. Reset page tables – valid-invalid bit = v.
6. Restart the instruction that caused the page fault.

What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out.
- Need page replacement algorithm.
- Performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.

Steps in handling a Page Replacement (1)



Steps in handling a Page Replacement (2)

1. Find the location of the desired page on disk.
2. Find a free frame:
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a victim page.
3. Bring the desired page into the (newly) free frame; update the page and frame tables.
4. Restart the process.

Comments on Page Replacement

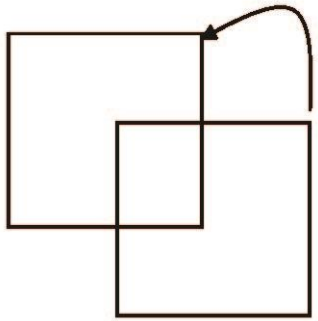
- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.
- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk.
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.

Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory:
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault.
  - And for every other process pages on first access.
  - This is **Pure demand paging**.
- Actually, a given instruction could access multiple pages -> multiple page faults:
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory.
  - Pain decreased because of locality of reference.
- Hardware support needed for demand paging:
  - Page table with valid/invalid bit.
  - Secondary memory (swap device with swap space).
  - Instruction restart.

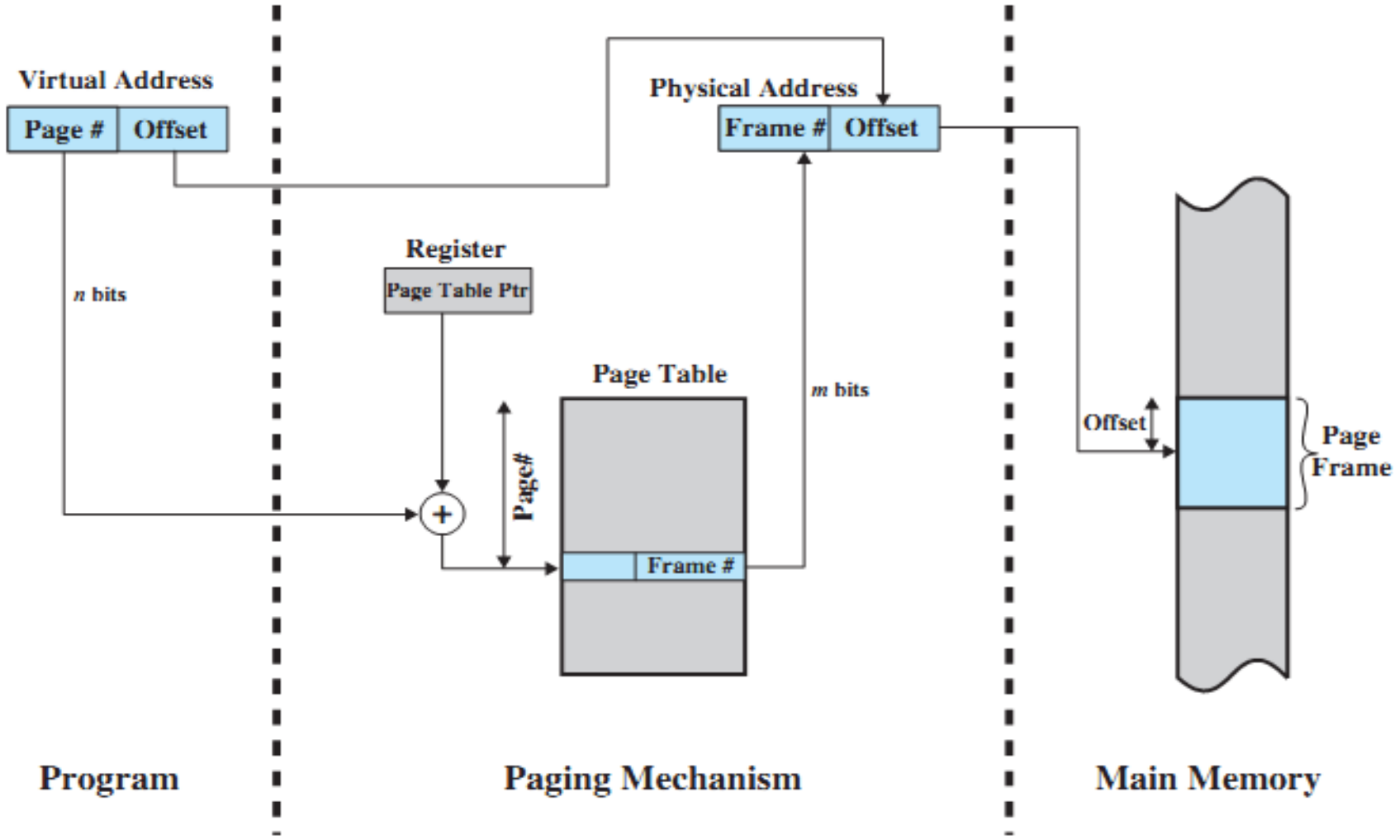
Instruction Restart

- Consider an instruction that could access several different locations:
  - block move

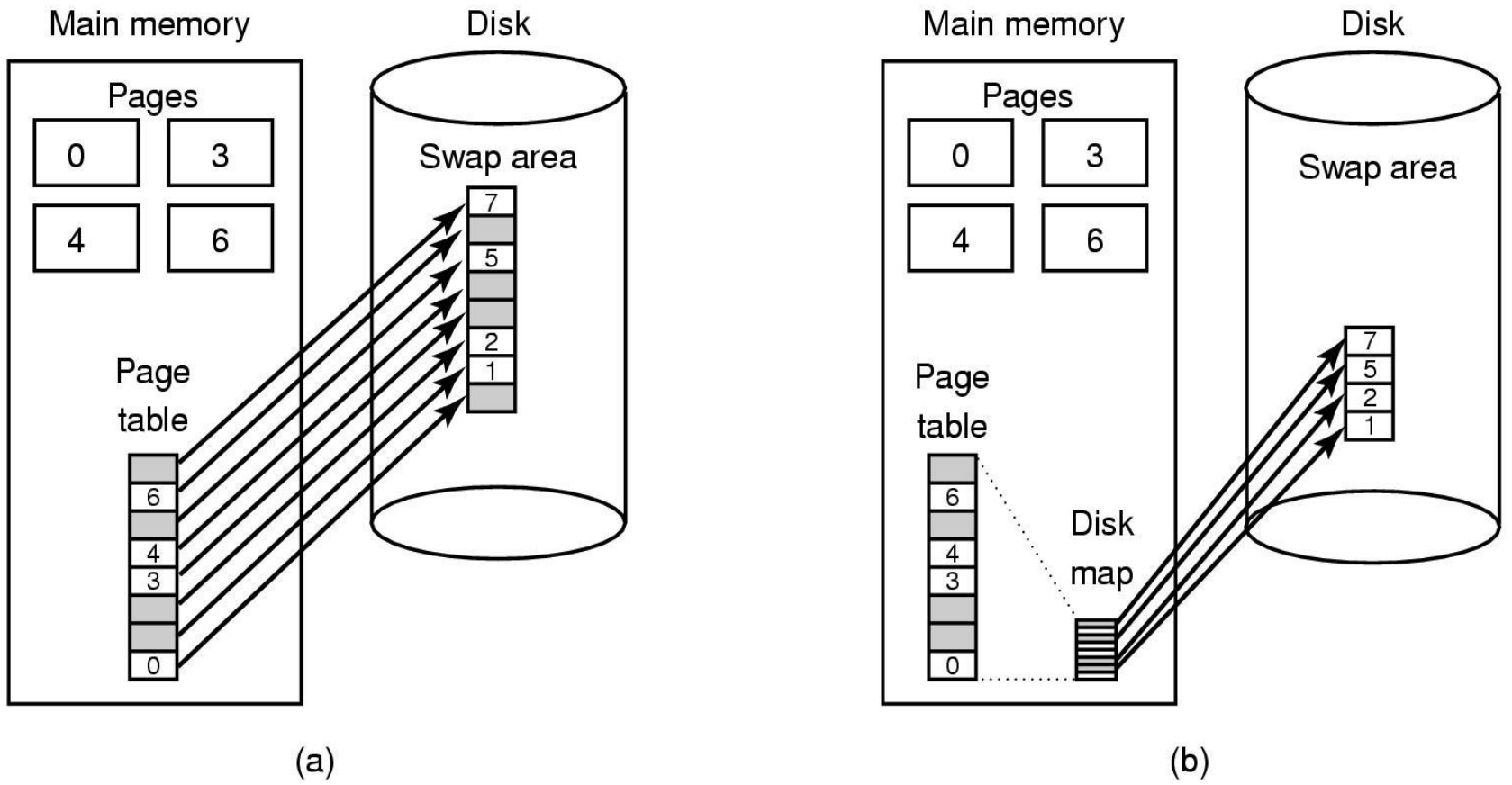


  - auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?

Address Translation in a Paging System



Backing/Swap Store



(a) Paging to static swap area. (b) Backing up pages dynamically.

Need for TLB

- Because the page table is in main memory, each virtual memory reference causes at least two physical memory accesses:
  - one to fetch the page table entry.
  - one to fetch the data.
- To overcome this problem a special cache is set up for page table entries, called the TLB (Translation Look-aside Buffer):
  - Contains page table entries that have been most recently used.
  - Works similar to main memory cache.

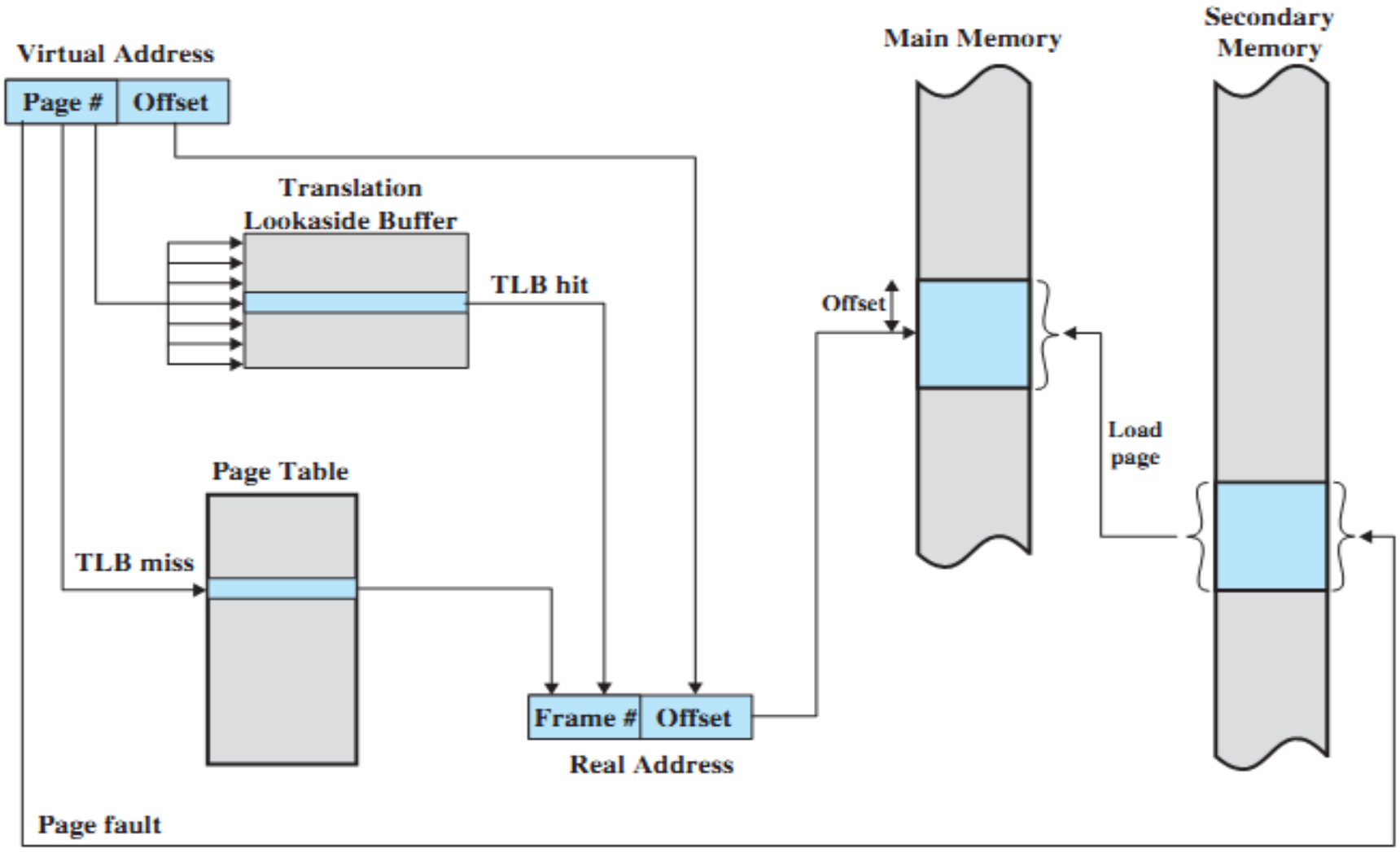
Example TLB

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

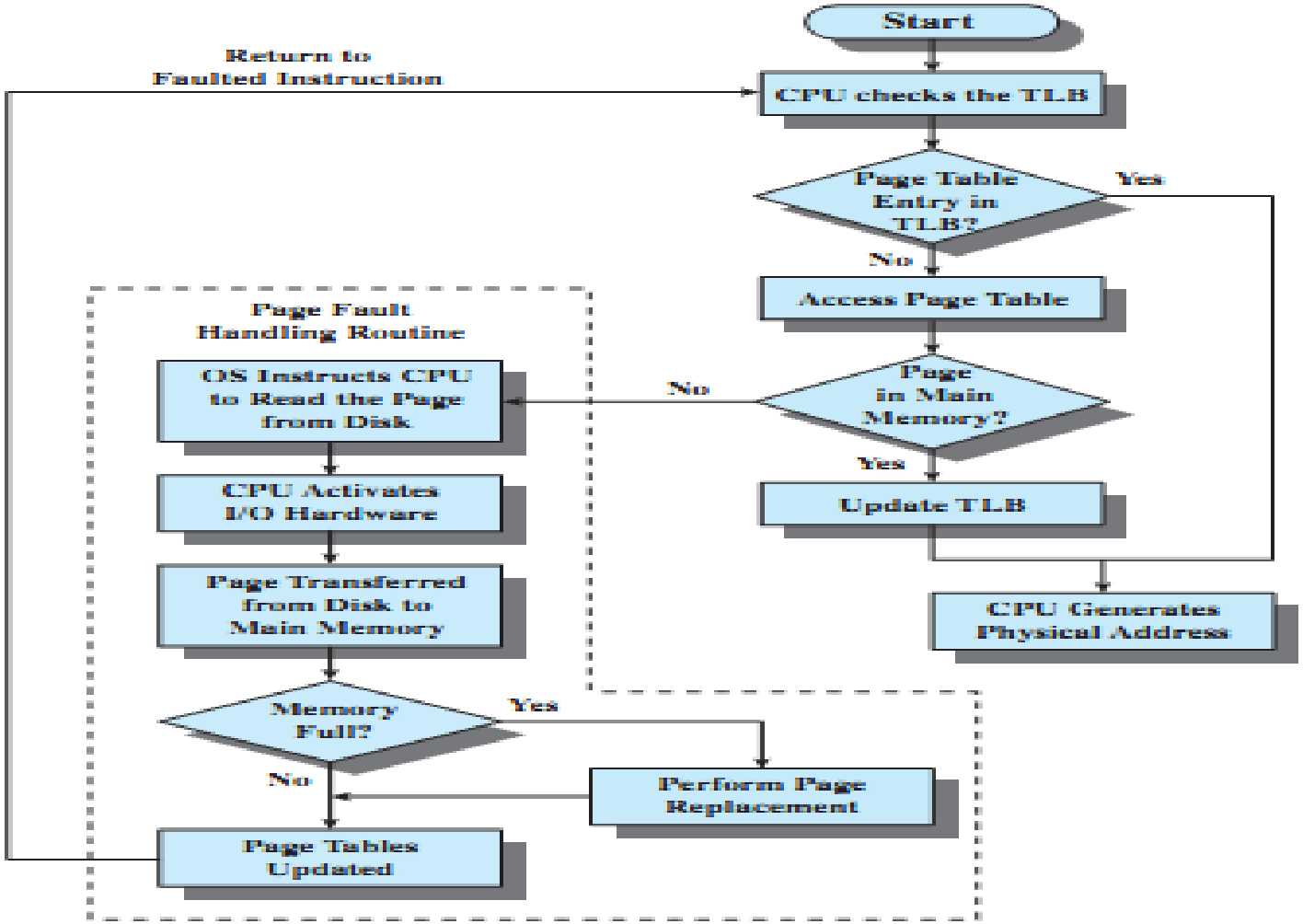
TLB Dynamics

- Given a logical address, the processor examines the TLB.
- If page table entry is present (a hit), the frame number is retrieved and the real (physical) address is formed.
- If page table entry is not found in the TLB (a miss), the page number is used to index the process page table:
  - if valid bit is set, then the corresponding frame is accessed.
  - if not, a page fault is issued to bring in the referenced page in main memory.
- The TLB is updated to include the new page entry.

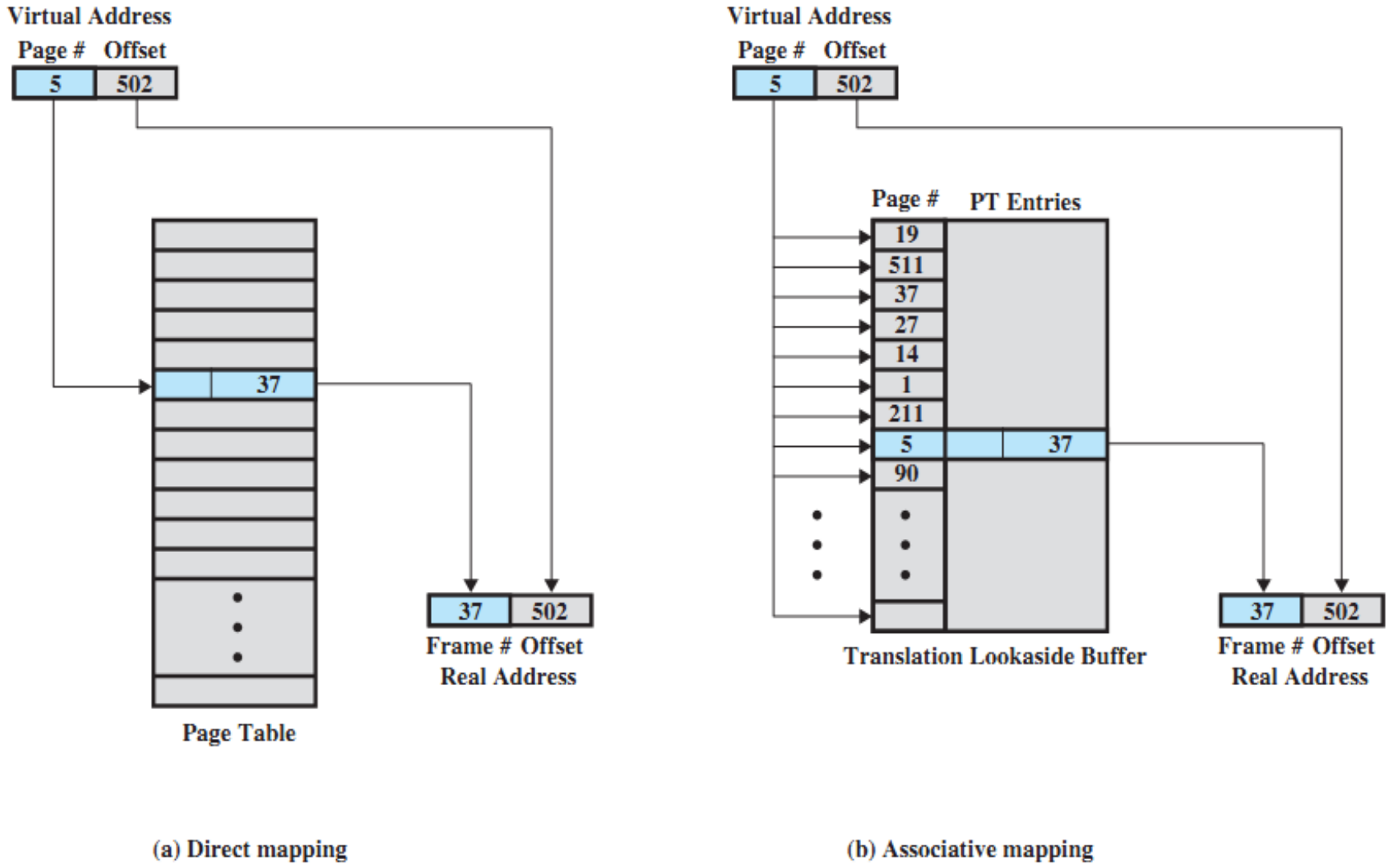
Use of a Translation Look-aside Buffer



Operation of Paging and TLB



Direct vs. Associative Lookup for Page Table Entries



TLB: further comments

- TLB use associative mapping hardware to simultaneously interrogates all TLB entries to find a match on page number.
- The TLB must be flushed each time a new process enters the Running state.
- The CPU uses two levels of cache on each virtual memory reference:
  - first the TLB: to convert the logical address to the physical address.
  - once the physical address is formed, the CPU then looks in the regular cache for the referenced word.

Stages in Demand Paging (worse case)

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine location of page on the disk.
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced.
    2. Wait for the device seek and/or latency time.
    3. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user.
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user.
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Performance of Demand Paging

- Three major activities:
  - Service the interrupt – careful coding means just several hundred instructions needed.
  - Read the page – lots of time..
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$, no page faults.
  - if $p = 1$, every reference is a fault.
- Effective Access Time (EAT) –

$$\begin{aligned} \text{EAT} = (1 - p) \times \text{memory access} \\ + p \times (\text{page fault overhead} \\ + [\text{swap page out}] \\ + \text{swap page in} \\ + \text{restart overhead}) \end{aligned}$$

Demand Paging Example

- Memory access time = 200 nanoseconds.
- Average page-fault service time = 8 milliseconds.
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

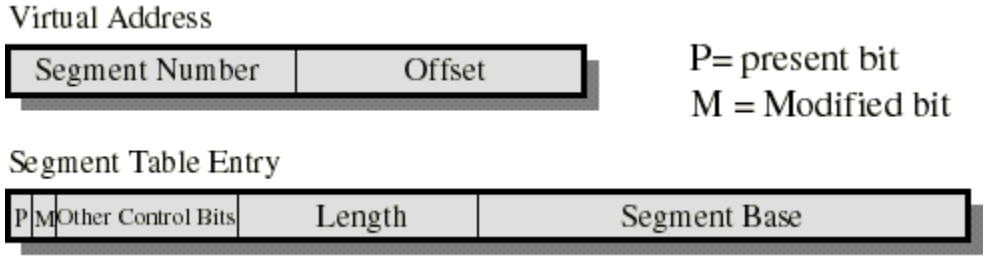
  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses.

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Segmentation and Paging Considerations

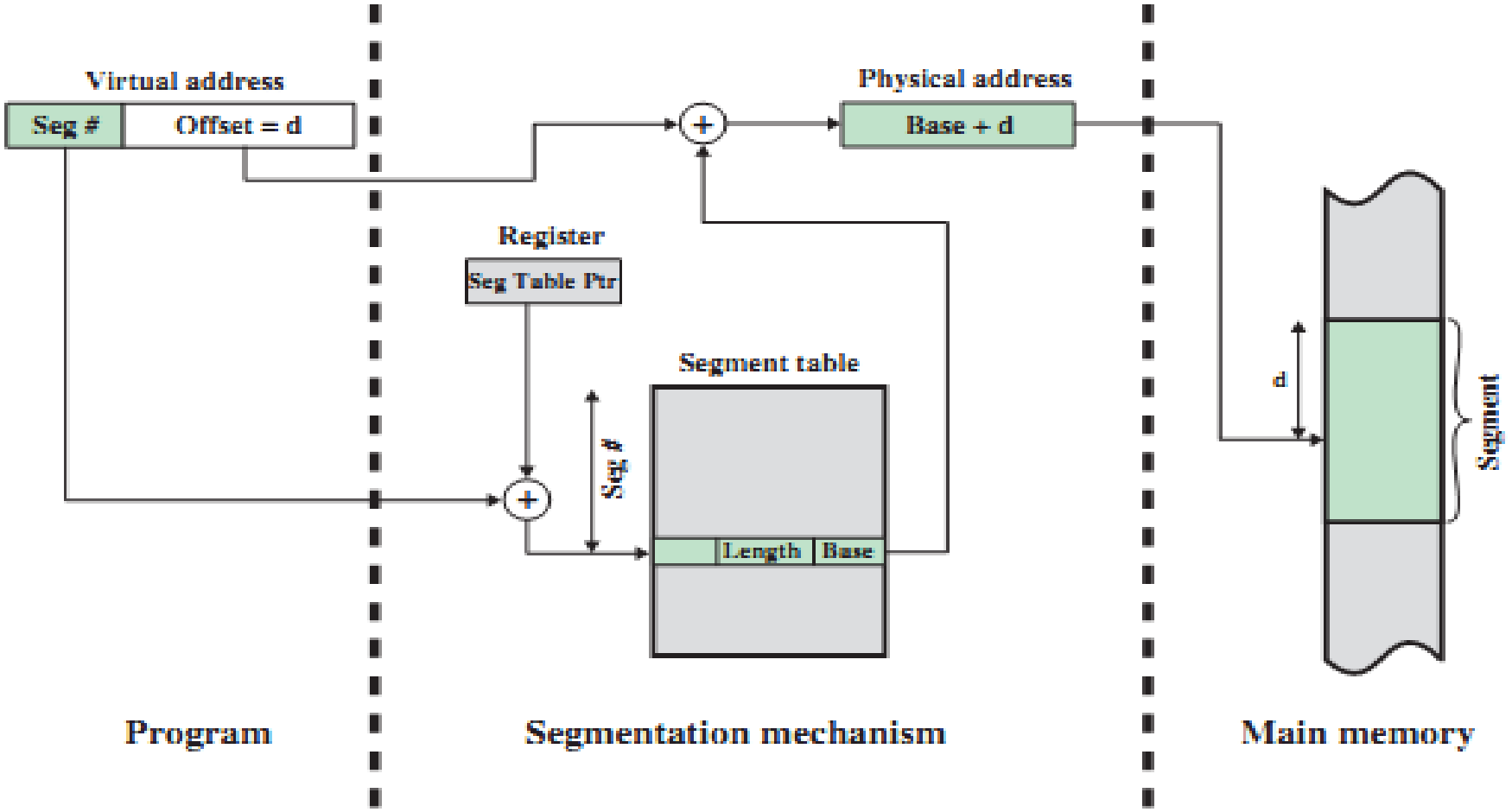
Dynamics of Segmentation

- Typically, each process has its own segment table.



- Similarly to paging, each segment table entry contains a present (valid-invalid) bit and a modified bit.
- If the segment is in main memory, the entry contains the starting address and the length of that segment.
- Other control bits may be present if protection and sharing is managed at the segment level.
- Logical to physical address translation is similar to paging except that the offset is added to the starting address (instead of appended).

Address Translation in a Segmentation System



Segmentation on Comments

- In each segment table entry, we have both the starting address and length of the segment; The segment can thus dynamically grow or shrink as needed.
- But variable length segments introduce external fragmentation and are more difficult to swap in and out.
- It is natural to provide protection and sharing at the segment level since segments are visible to the programmer (pages are not).
- Useful protection bits in segment table entry:
  - read-only/read-write bit
  - Kernel/User bit

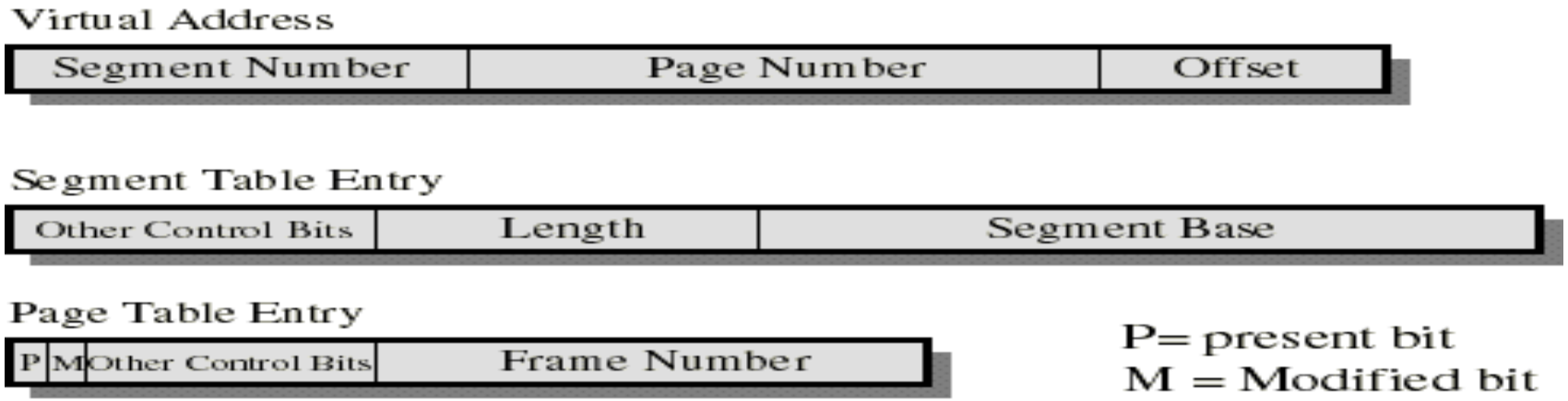
Comparison of Paging and Segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Combined Segmentation and Paging

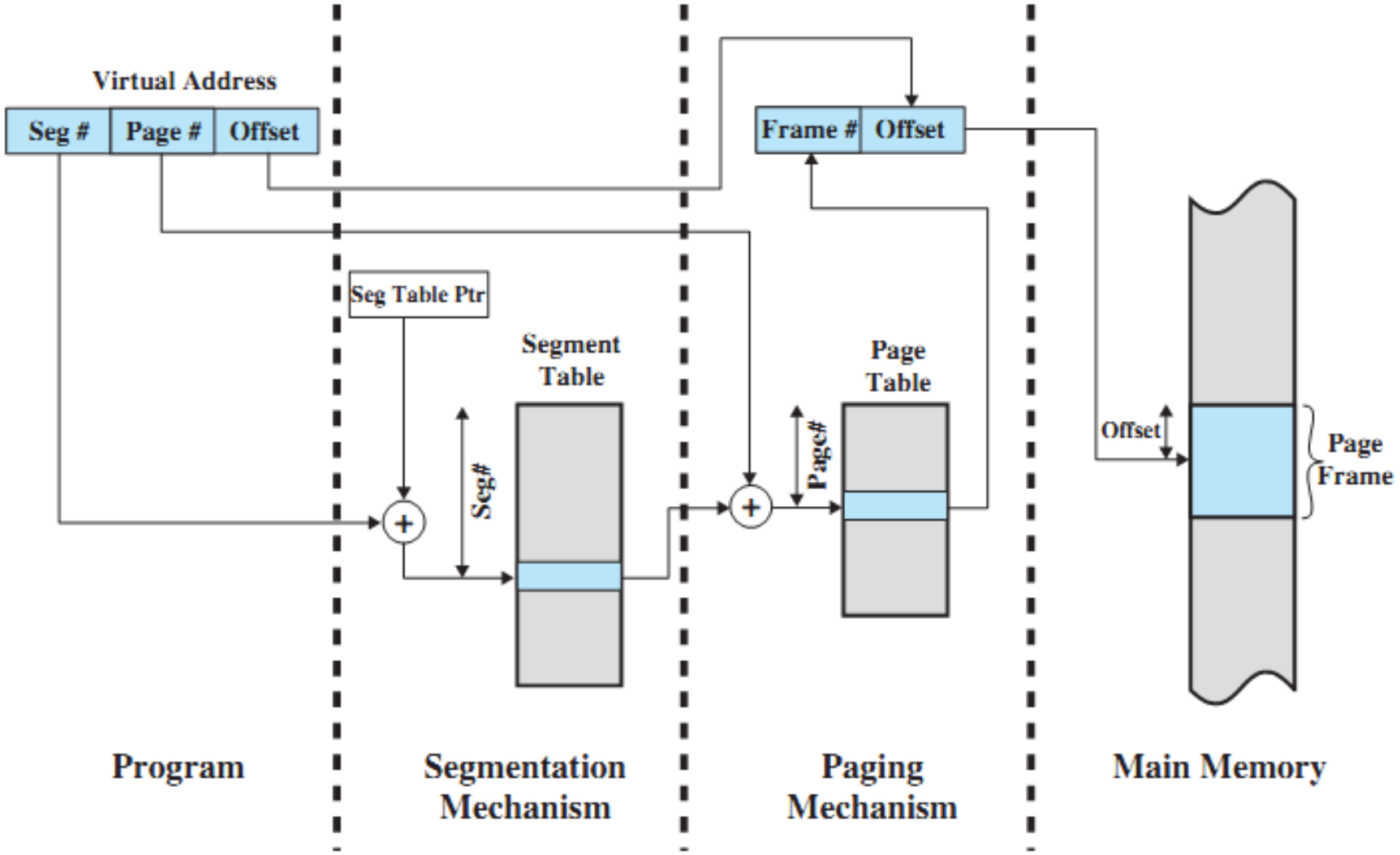
- To combine their advantages, some OSs page the segments.
- Several combinations exist – assume each process has:
  - one segment table.
  - several page tables: one page table per segment.
- The virtual address consists of:
  - a segment number: used to index the segment table who's entry gives the starting address of the page table for that segment.
  - a page number: used to index that page table to obtain the corresponding frame number.
  - an offset: used to locate the word within the frame.

Simple Combined Segmentation and Paging



- The Segment Base is the physical address of the page table of that segment.
- Present/modified bits are present only in page table entry.
- Protection and sharing info most naturally resides in segment table entry.
  - Ex: a read-only/read-write bit, a kernel/user bit...

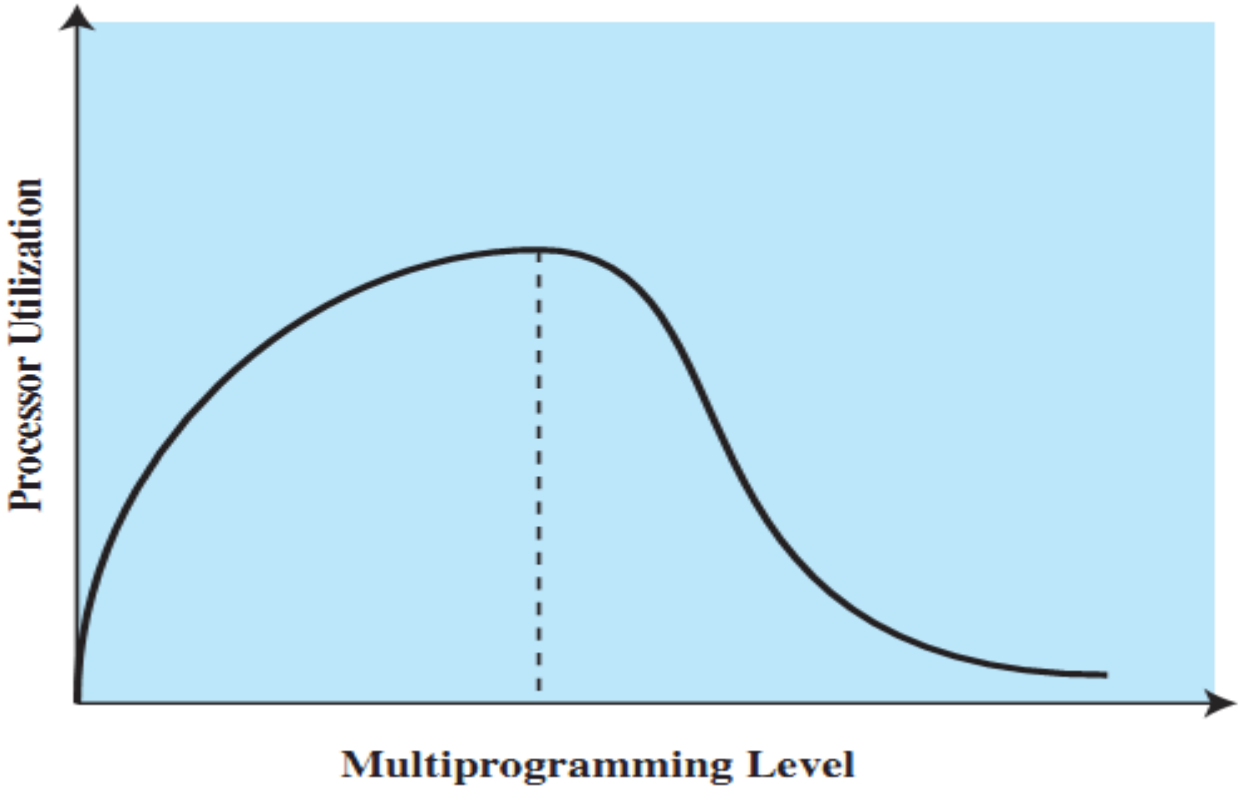
Address Translation in combined Segmentation/Paging



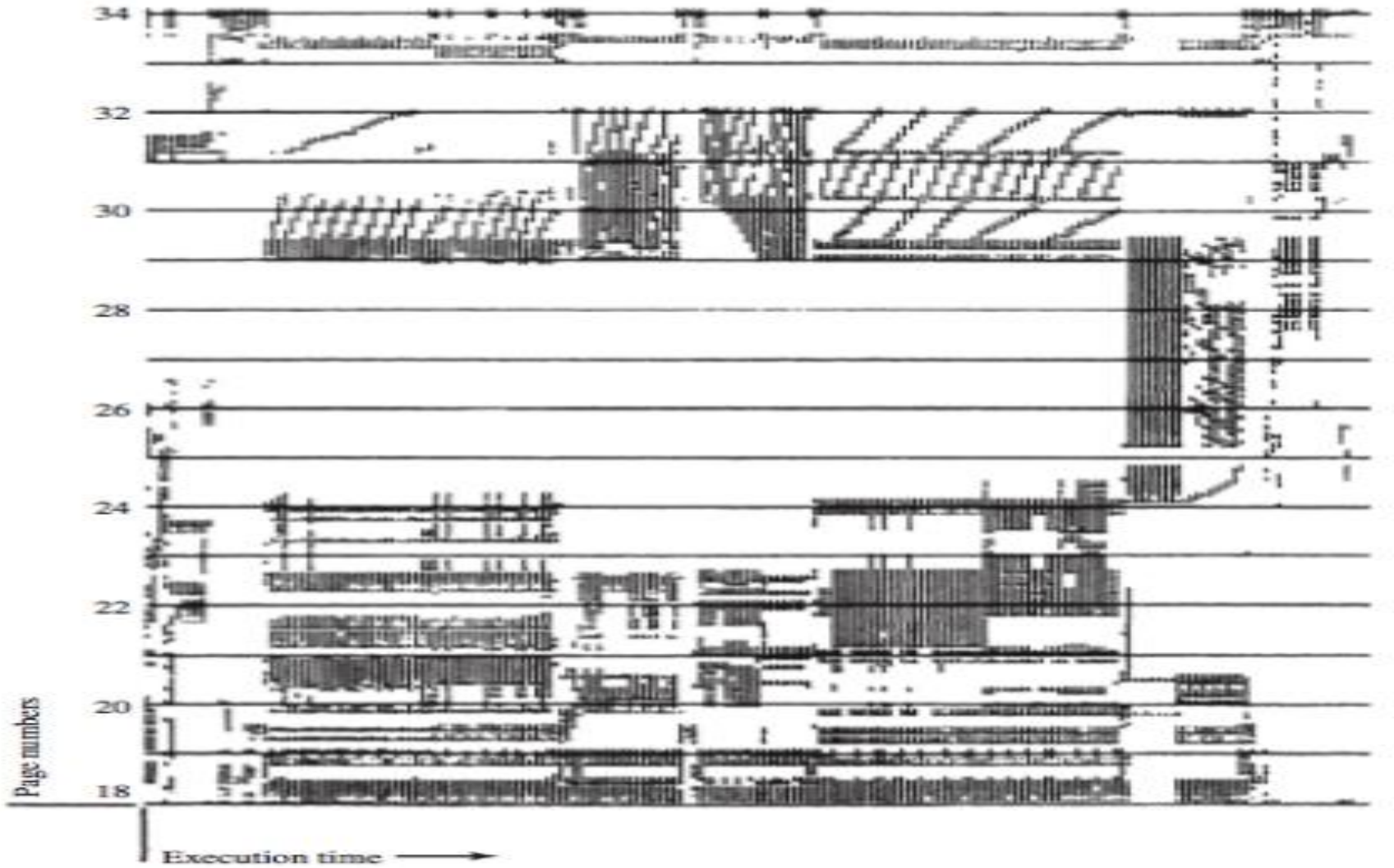
Paging Considerations

- Locality, VM and Thrashing
- Prepaging (Anticipatory Paging)
- Page size issue
- TLB reach
- Program structure
- I/O interlock
- Copy-on-Write
- Memory-Mapped Files

Degree of multiprogramming to be reached



Locality in a Memory-Reference Pattern



Locality and Virtual Memory

- Principle of locality of references: memory references within a process tend to cluster.
- Hence: only a few pieces of a process will be needed over a short period of time.
- Possible to make intelligent guesses about which pieces will be needed in the future.
- This suggests that virtual memory may work efficiently (i.e., thrashing should not occur too often).

Possibility of Thrashing (1)

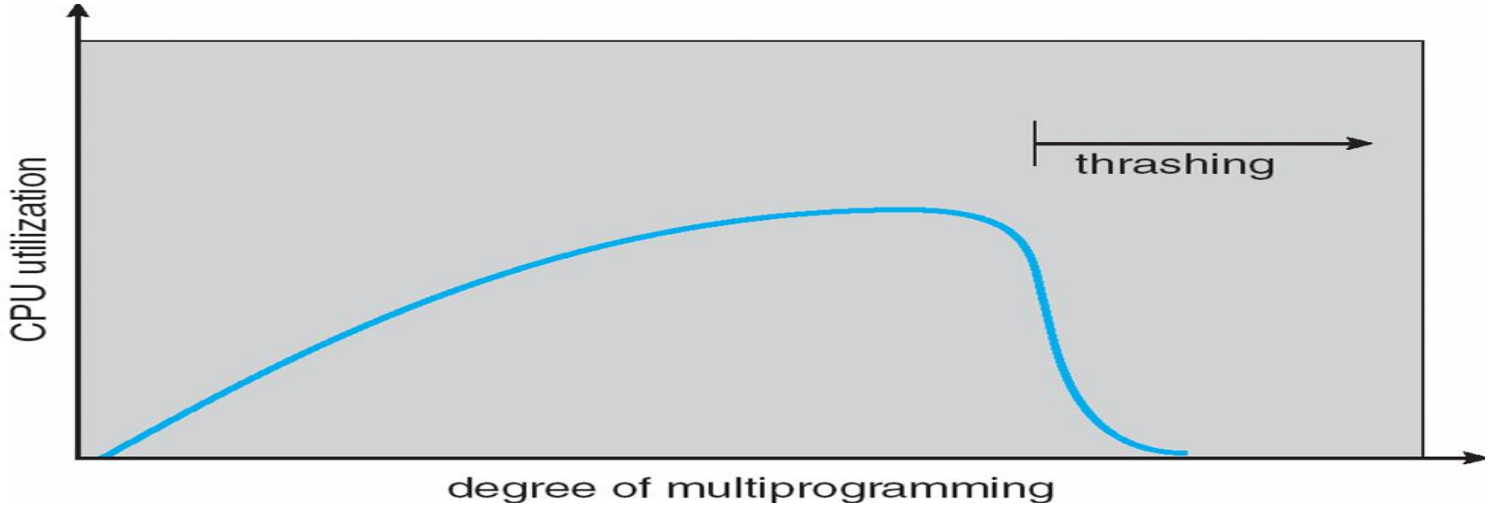
- If a process does not have “enough” pages, the page-fault rate is very high:
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system.
- **Thrashing** ≡ a process is busy swapping pages in and out.

Possibility of Thrashing (2)

- To accommodate as many processes as possible, only a few pieces of each process are maintained in main memory.
- But main memory may be full: when the OS brings one piece in, it must swap one piece out.
- The OS must not swap out a piece of a process just before that piece is needed.
- If it does this too often this leads to thrashing:
  - The processor spends most of its time swapping pieces rather than executing user instructions.

Locality and Thrashing

- Why does demand paging work?
  Locality model:
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality > total memory size



Prepaging

- Can help to reduce the large number of page faults that occurs at process startup or resumption.
- Prepage all or some of the pages a process will need, before they are referenced.
- But if prepaged pages are unused, I/O and memory was wasted.
- Assume *s* pages are prepaged and a fraction α of the pages are used:
  - Is cost of $s * \alpha$ saved pages faults greater or less than the cost of prepaging $s * (1- \alpha)$ unnecessary pages*?*
  - α near zero $\Rightarrow$ prepaging loses.

Page Size Issues

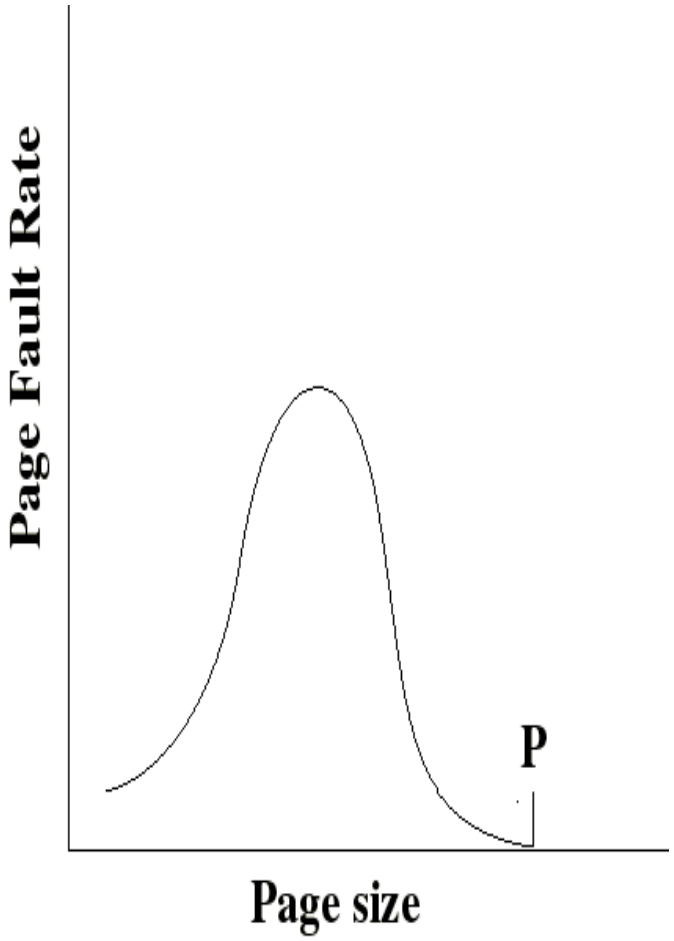
- Sometimes OS designers have a choice:
  - Especially if running on custom-built CPU.
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes).
- On average, growing over time.

The Page Size Issue (1)

- Page size is defined by hardware; exact size to use is a difficult question:
  - Large page size is good since for a small page size, more pages are required per process; More pages per process means larger page tables. Hence, a larger portion of page tables in virtual memory.
  - Large page size is good since disks are designed to efficiently transfer large blocks of data.
  - Larger page sizes means less pages in main memory; this increases the TLB hit ratio.
  - Small page size is good to minimize internal fragmentation.

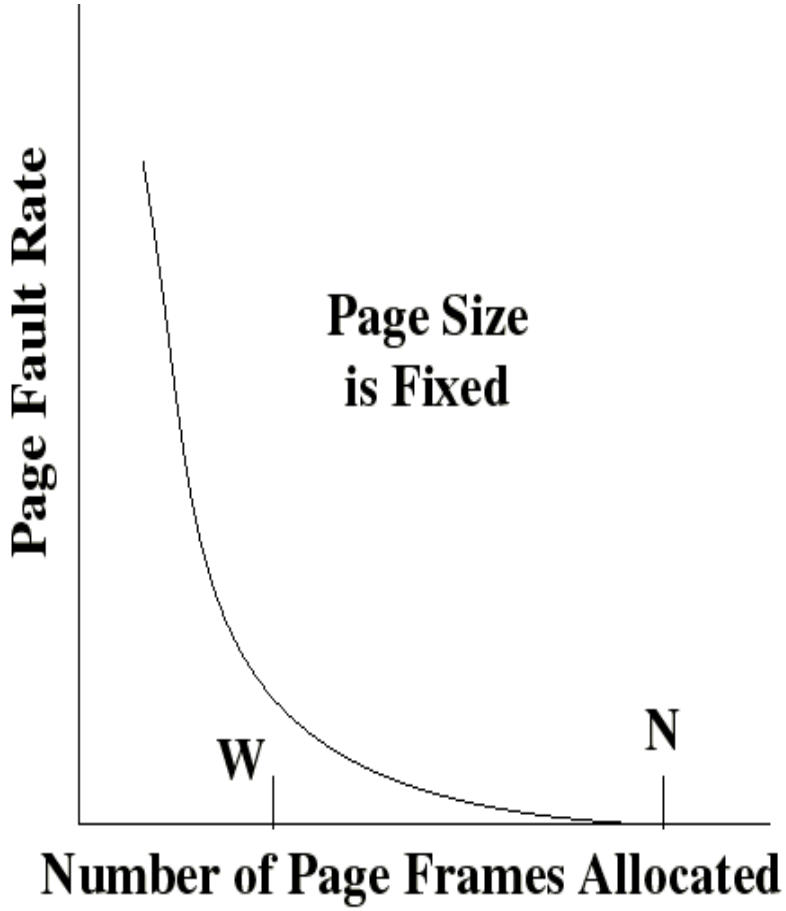
The Page Size Issue (2)

- With a very small page size, each page matches the code that is actually used: faults are low.
- Increased page size causes each page to contain more code that is not used. Page faults rise.
- Page faults decrease if we approach point P were the size of a page is equal to the size of the entire process.



The Page Size Issue (3)

- Page fault rate is also determined by the number of frames allocated per process.
- Page faults drops to a reasonable value when W frames are allocated.
- Drops to 0 when the number (N) of frames is such that a process is entirely in memory.



The Page Size Issue (4)

- Page sizes from 1KB to 4KB are most commonly used. Increase in page sizes is related to trend of increasing block sizes.
- But the issue is non trivial. Hence some processors supported multiple page sizes, for example:
  - Pentium supports 2 sizes: 4KB or 4MB
  - R4000 supports 7 sizes: 4KB to 16MB

Example Page Sizes

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit words
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
IBM POWER	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

TLB Reach

- The amount of memory accessible from the TLB.
- Ideally, working set of each process is stored in TLB:
  - Otherwise there is a high degree of page faults.
- TLB Reach = (TLB Size) x (Page Size)
- Increase the size of the TLB:
  - might be expensive.
- Increase the Page Size:
  - This may lead to an increase in internal fragmentation as 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 an increase in fragmentation.

Program Structure

- Program structure
  - **int A[][] = new int[1024][1024];**
  - Each row is stored in one page.
  - Program 1:

```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i,j] = 0;
```

    we have 1024 x 1024 page faults

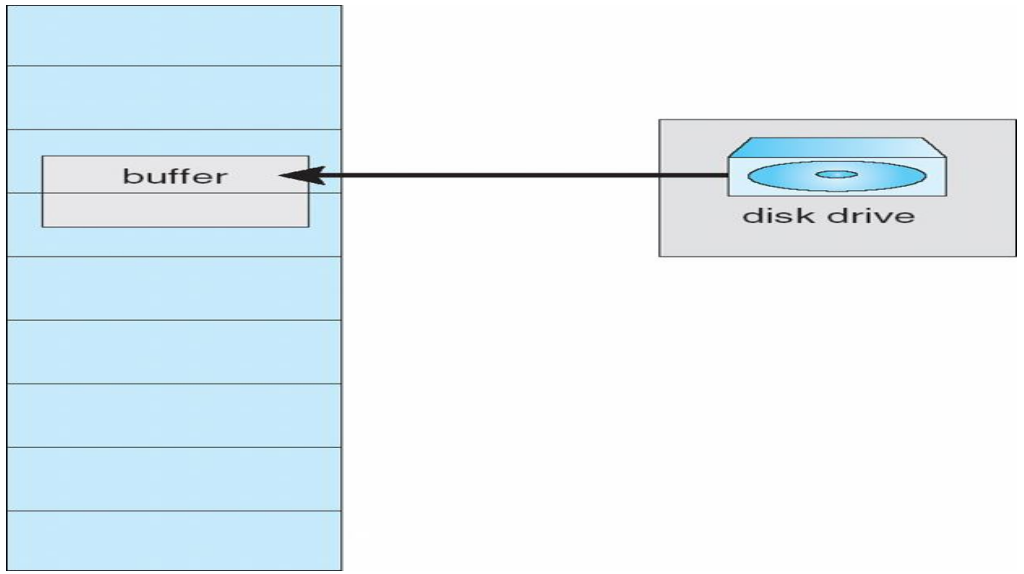
  - Program 2:

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i,j] = 0;
```

    we have 1024 page faults

I/O Interlock

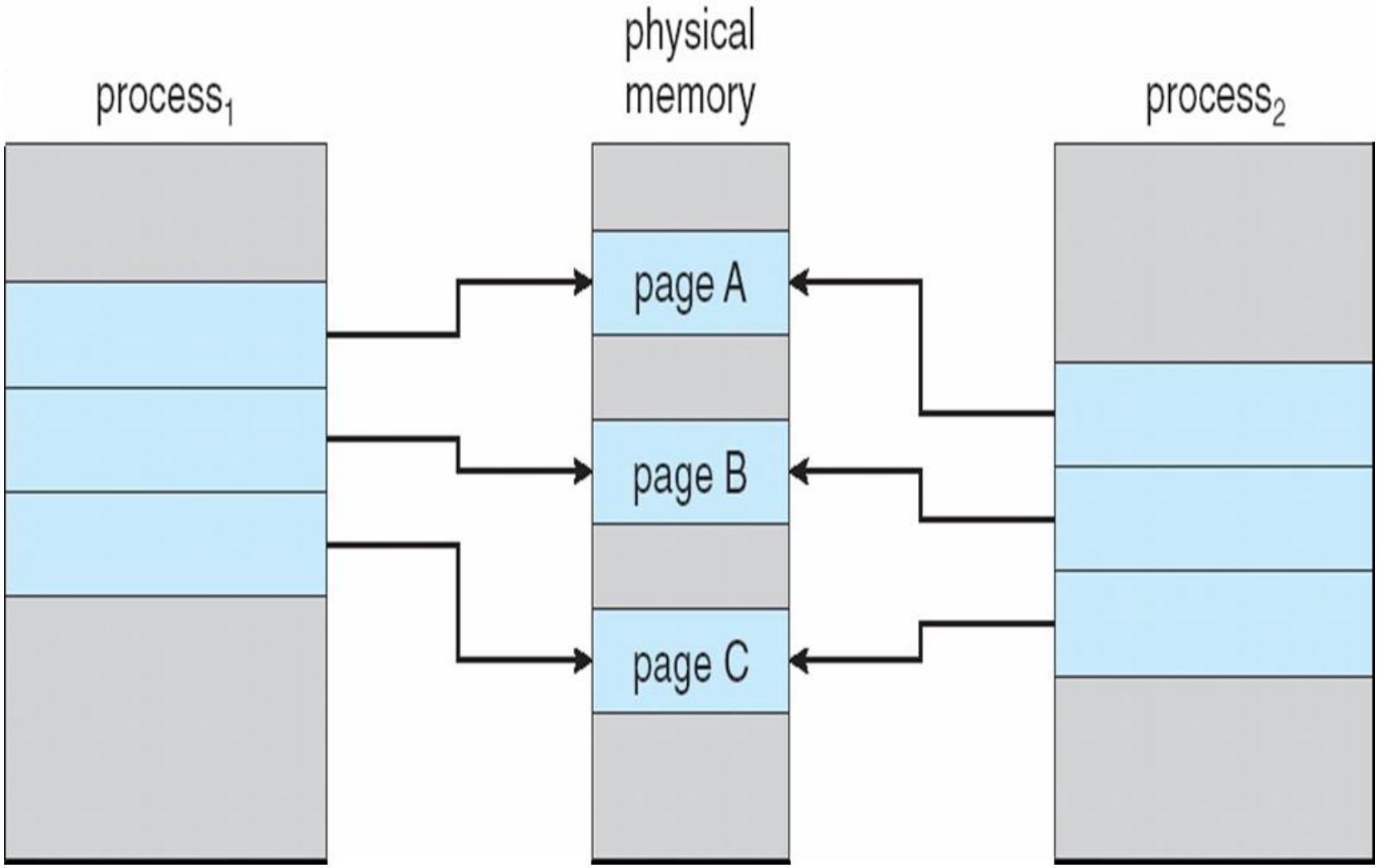
- **I/O Interlock** – Pages must sometimes be locked into memory.
- Consider I/O – Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.



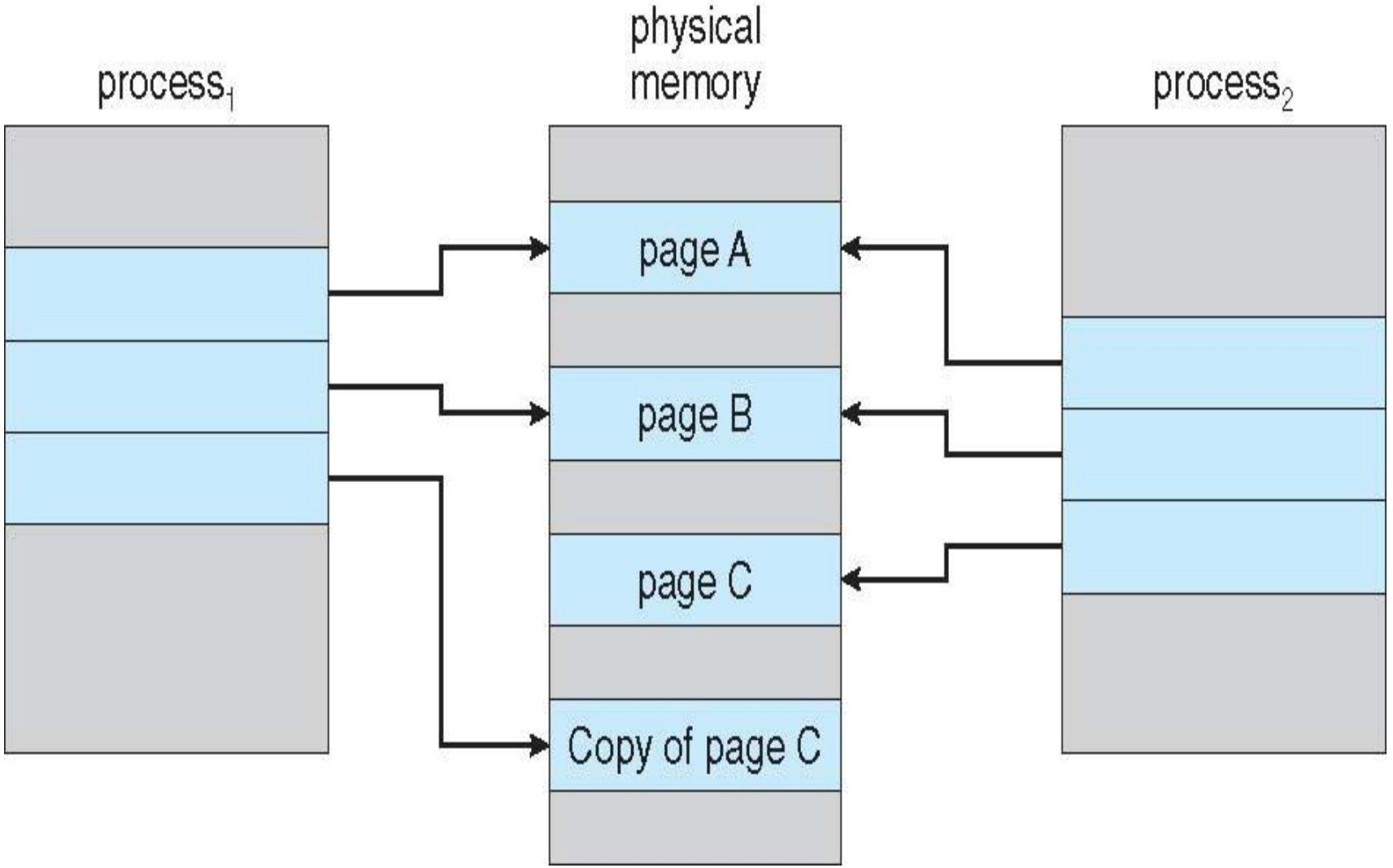
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory.
- If either process modifies a shared page, only then is the page copied.
- COW allows more efficient process creation as only modified pages are copied.
- In general, free pages are allocated from a pool of zero-fill-on-demand pages.

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



What Happens if there is no Free Frame?

- Used up by process pages.
- Also in demand from the kernel, I/O buffers, etc.
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out:
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

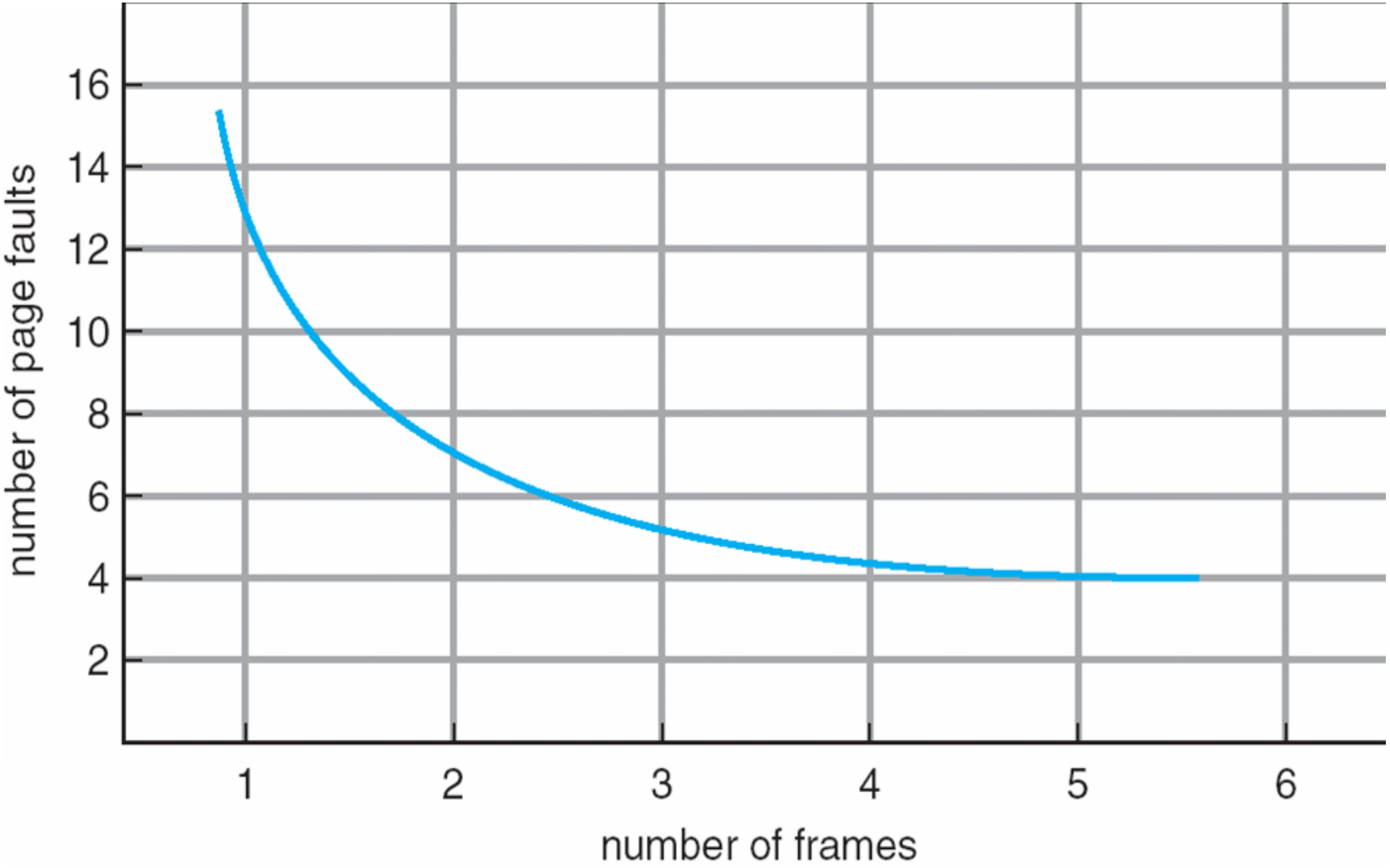
Page Replacement Algorithms

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Page Replacement Algorithms

- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults and page replacements on that string.
- In all our examples, we use a few recurring reference strings.

Graph of Page Faults vs. the Number of Frames

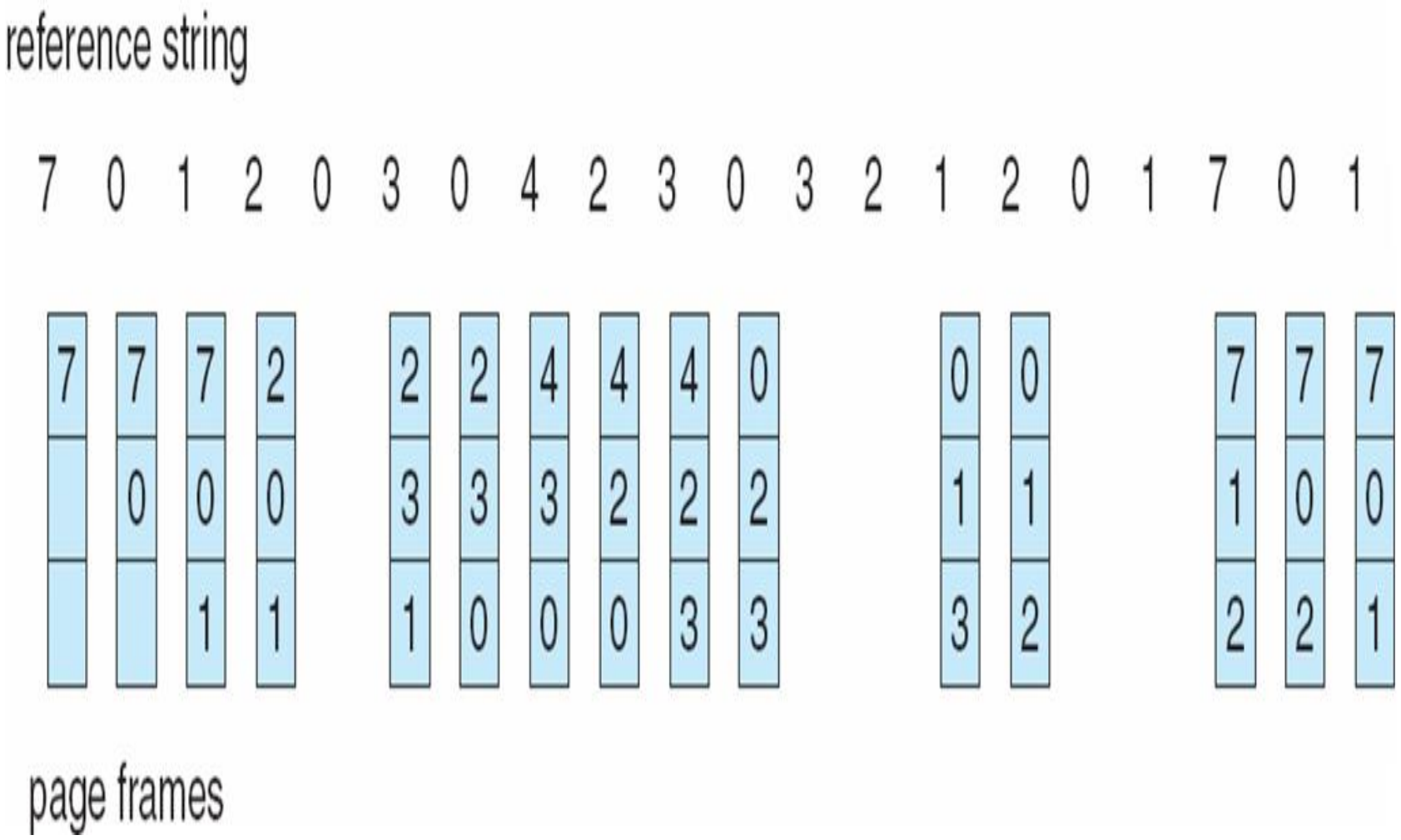


ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

The FIFO Policy

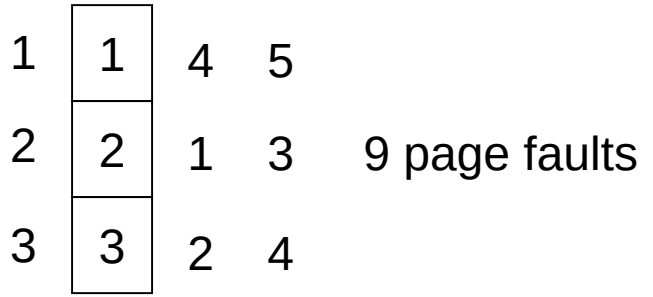
- Treats page frames allocated to a process as a circular buffer:
  - When the buffer is full, the oldest page is replaced. Hence first-in, first-out:
    - A frequently used page is often the oldest, so it will be repeatedly paged out by FIFO.
  - Simple to implement:
    - requires only a pointer that circles through the page frames of the process.

FIFO Page Replacement



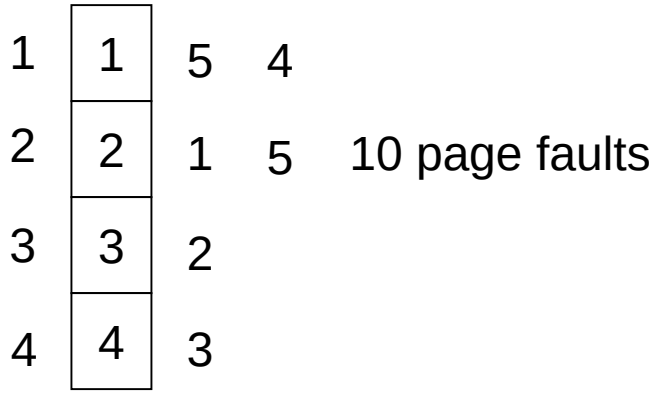
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process):



9 page faults

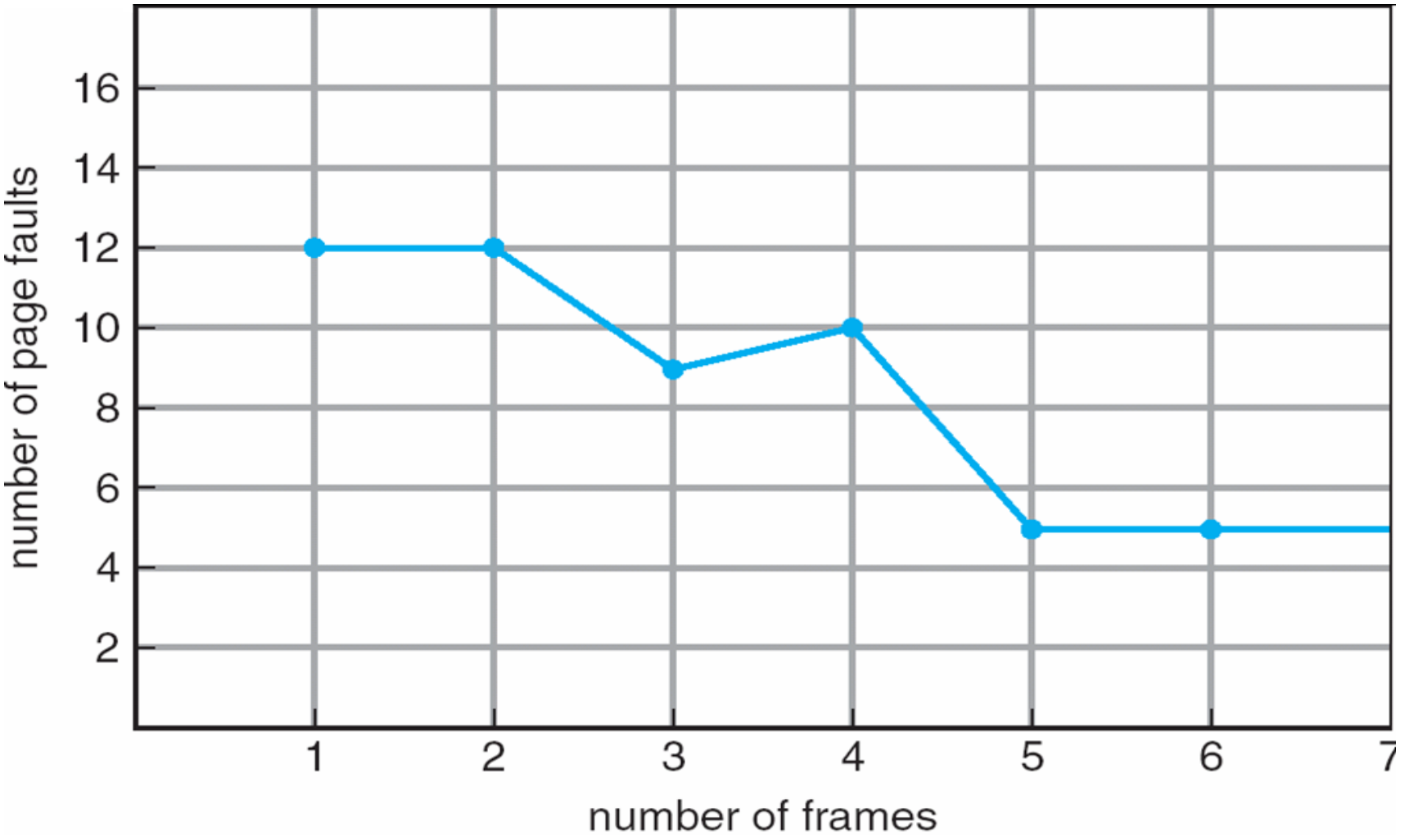
- 4 frames:



10 page faults

- FIFO Replacement manifests Belady's Anomaly:
  - more frames $\Rightarrow$ more page faults

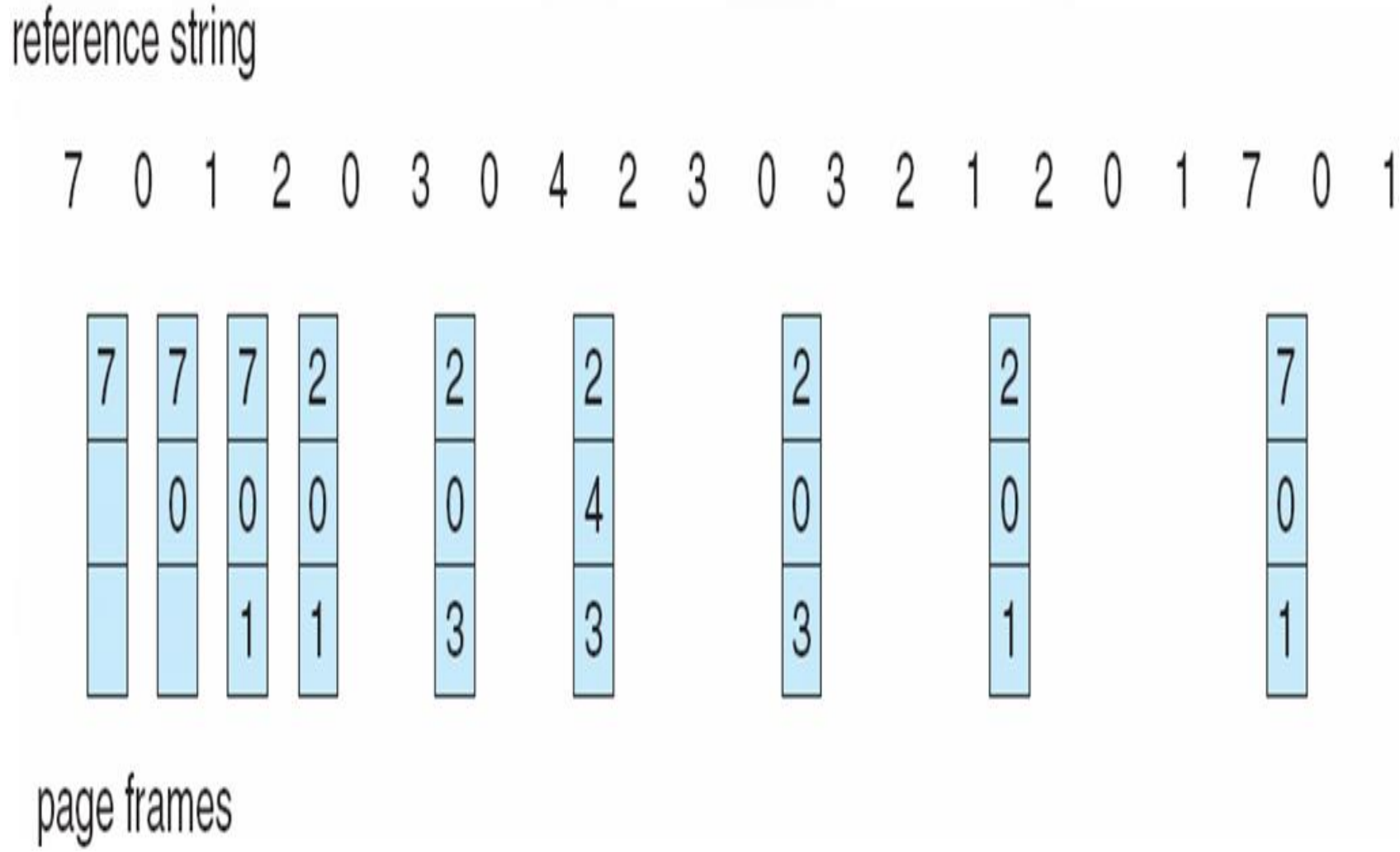
FIFO Illustrating Belady's Anomaly



Optimal Page Replacement

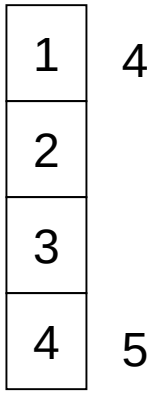
- The Optimal policy selects for replacement the page that will not be used for longest period of time.
- Impossible to implement (need to know the future) but serves as a standard to compare with the other algorithms we shall study.

Optimal Page Replacement



Optimal Algorithm

- Reference string : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 4 frames example



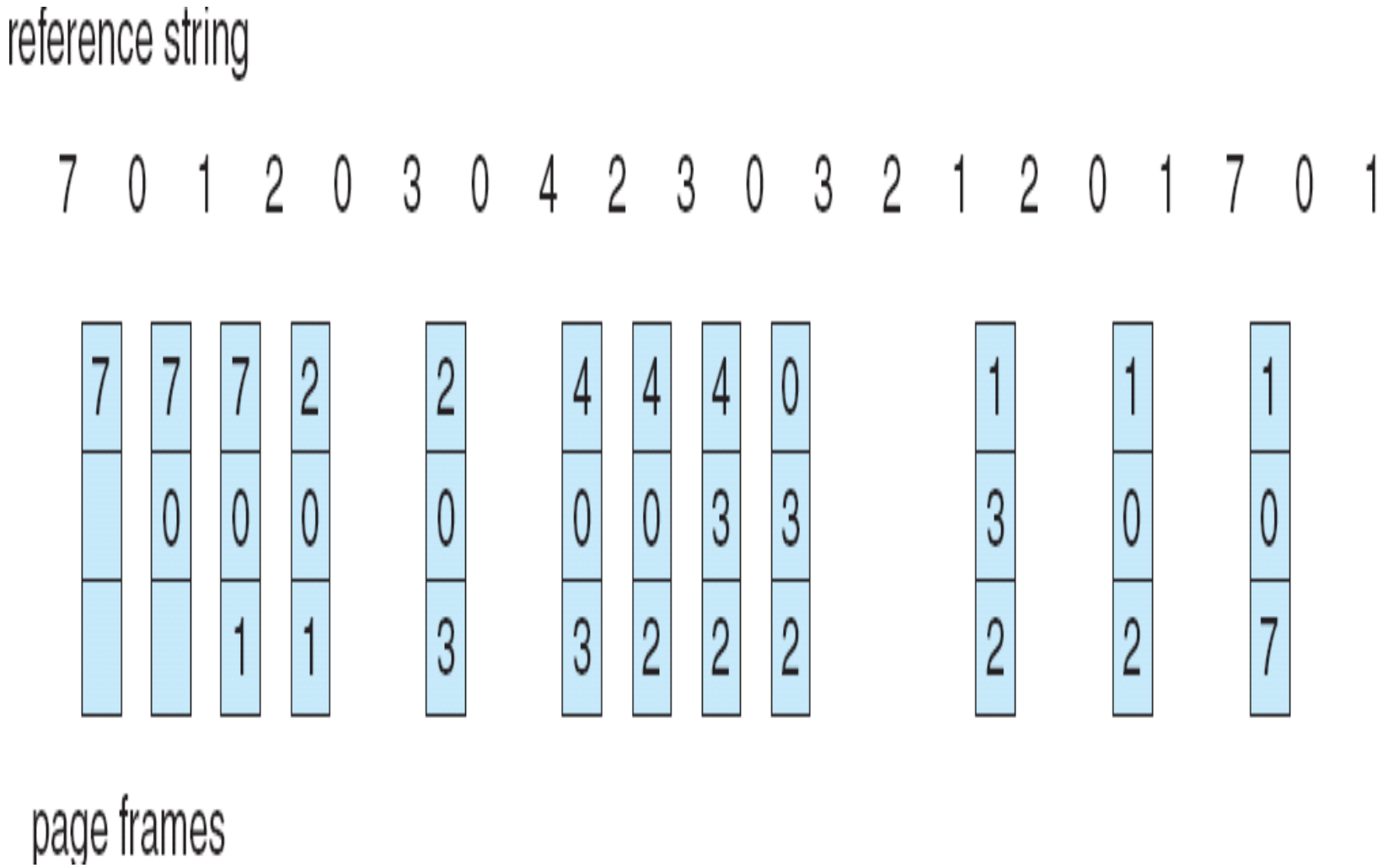
6 page faults

- How do you know future use? You don’t!
- Used for measuring how well your algorithm performs.

The LRU Policy

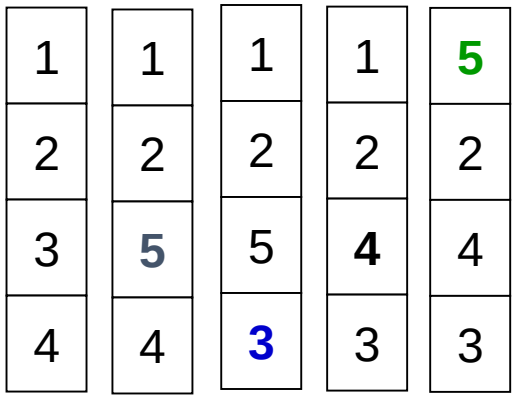
- Replaces the page that has not been referenced for the longest time:
  - By the principle of locality, this should be the page least likely to be referenced in the near future.
  - performs nearly as well as the optimal policy.

LRU Page Replacement



Least Recently Used (LRU) Algorithm

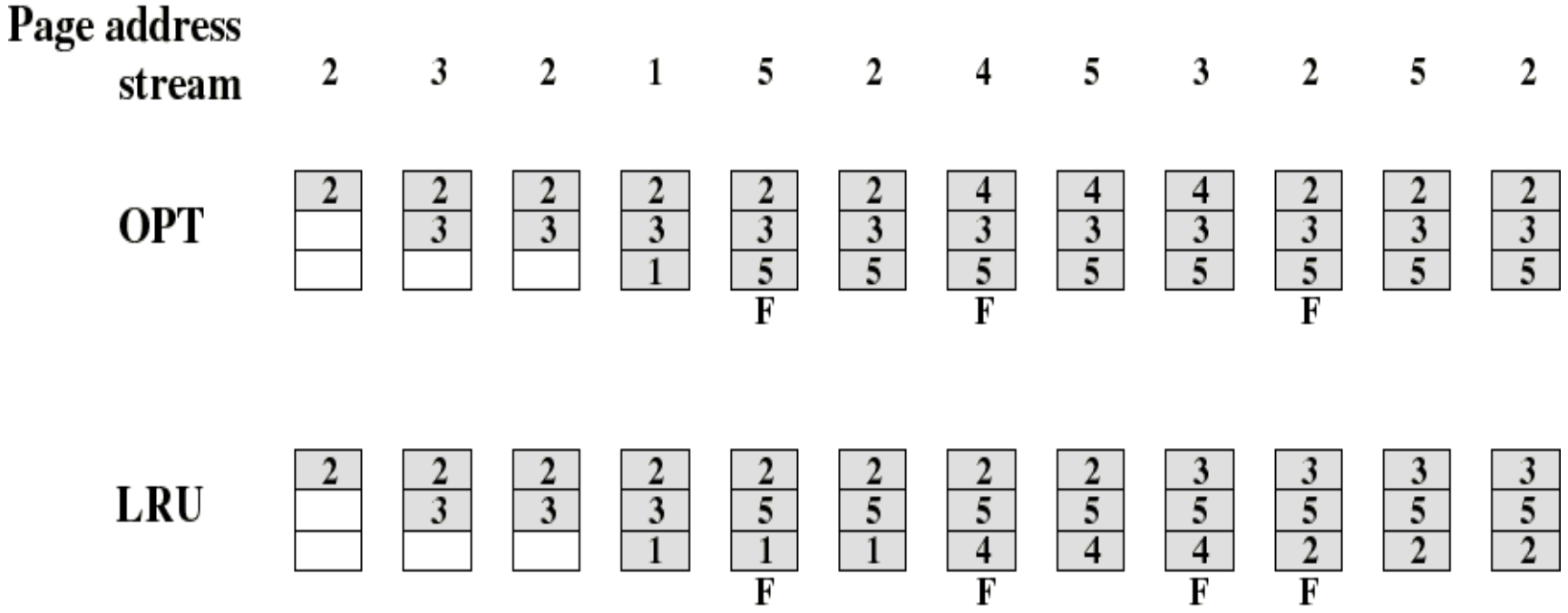
- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



8 page faults

Comparison of OPT with LRU

- Example: A process of 5 pages with an OS that fixes the resident set size to 3.



Comparison of FIFO with LRU

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
					F	F	F		F		F	F

- LRU recognizes that pages 2 and 5 are referenced more frequently than others but FIFO does not.

Implementation of the LRU Policy

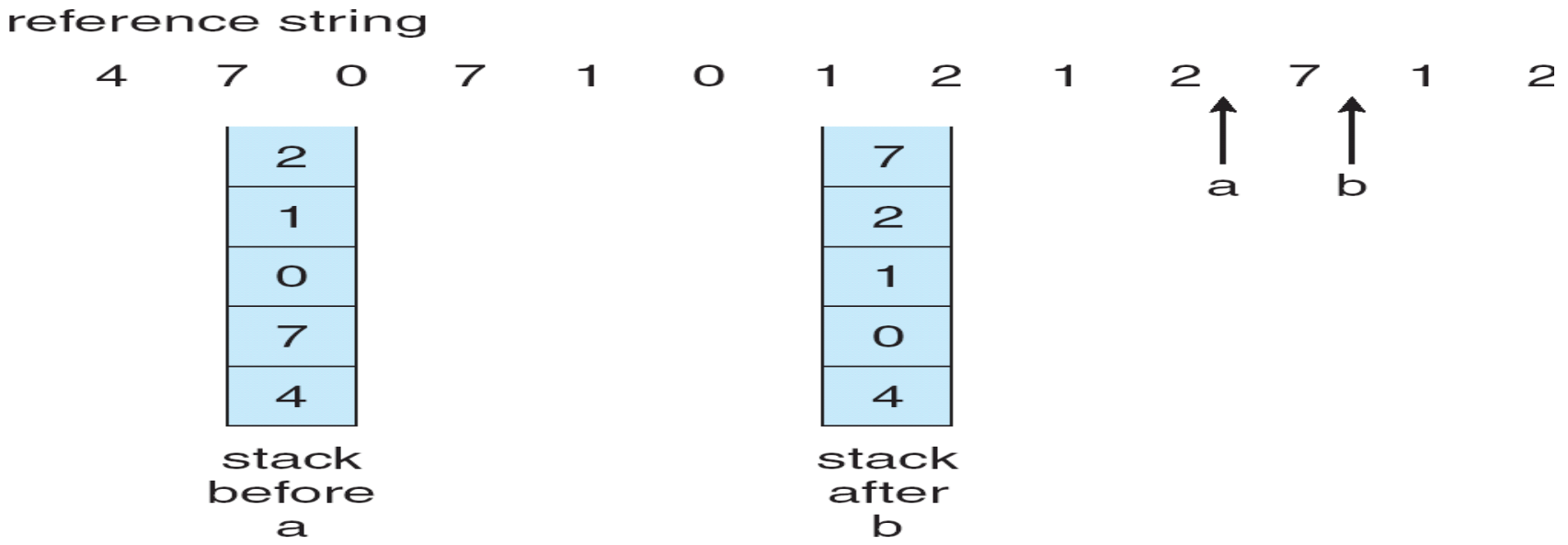
- Each page could be tagged (in the page table entry) with the time at each memory reference.
- The LRU page is the one with the smallest time value (needs to be searched at each page fault).
- This would require expensive hardware and a great deal of overhead.
- Consequently very few computer systems provide sufficient hardware support for true LRU replacement policy.
- Other algorithms are used instead.

LRU Implementations

- Counter implementation:
  - Every page entry has a counter; every time a page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which are to change.
- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement.

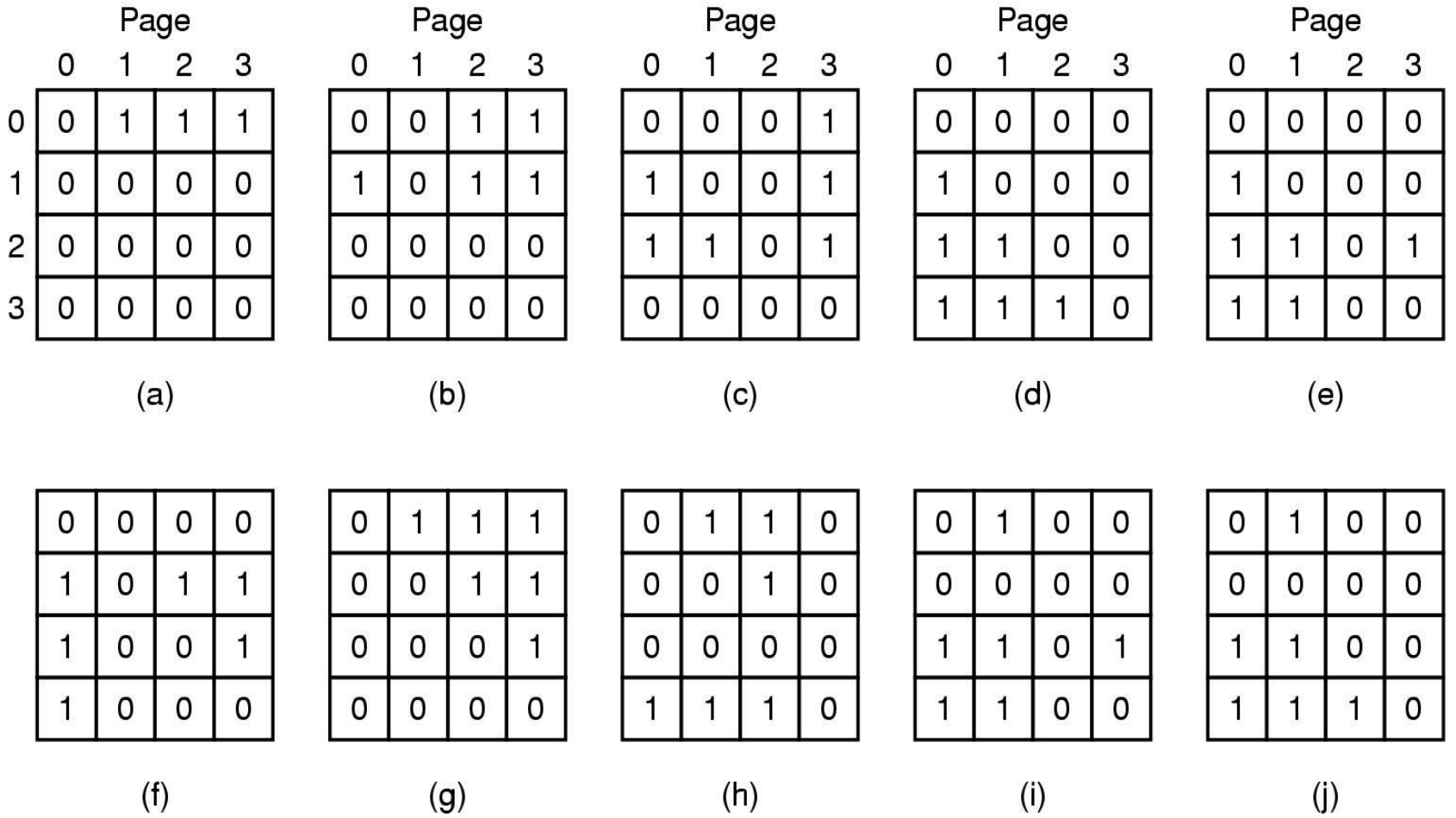
Use of a stack to implement LRU

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement – always take the bottom one.



Hardware Matrix LRU Implementation

Pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3



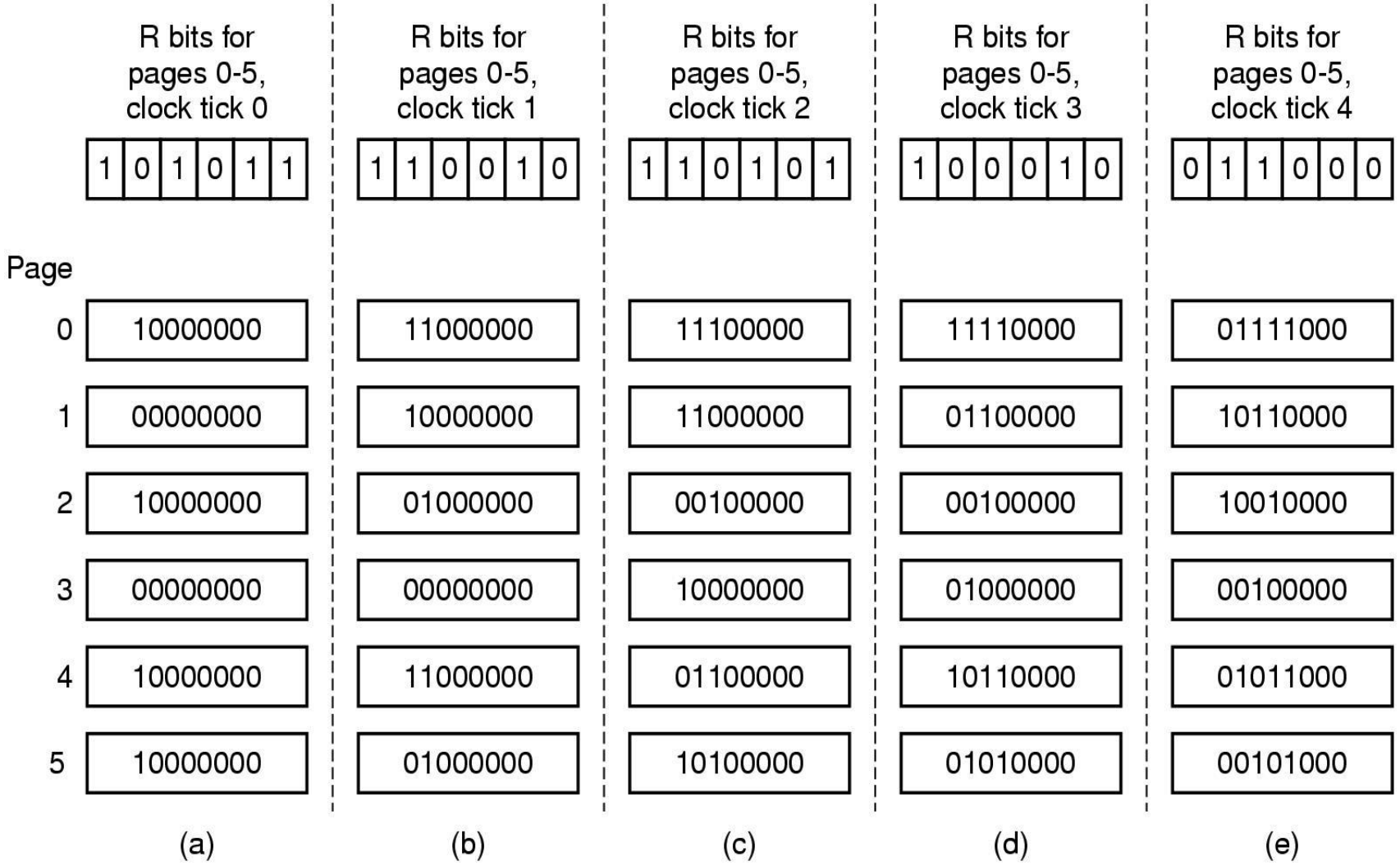
LRU Approximation Algorithms (1)

- Reference Bit:
  - With each page associate a bit, initially = 0
  - When page is referenced, bit is set to 1.
  - Replace the one which is 0 (if one exists) – we do not know the real order of use, however.

LRU Approximation Algorithms (2)

- Reference Byte:
  - Idea is to record reference bits at regular intervals; Keep a byte of reference bits for each page.
  - At regular intervals (say, every 20 ms), left shift the reference bit of each page into the high-order bit of the byte.
  - Each reference byte keeps the history of the page use (aging) for the last eight time intervals.
  - If we interpret the reference byte as an unsigned integer, the page with the lowest number is the LRU page.

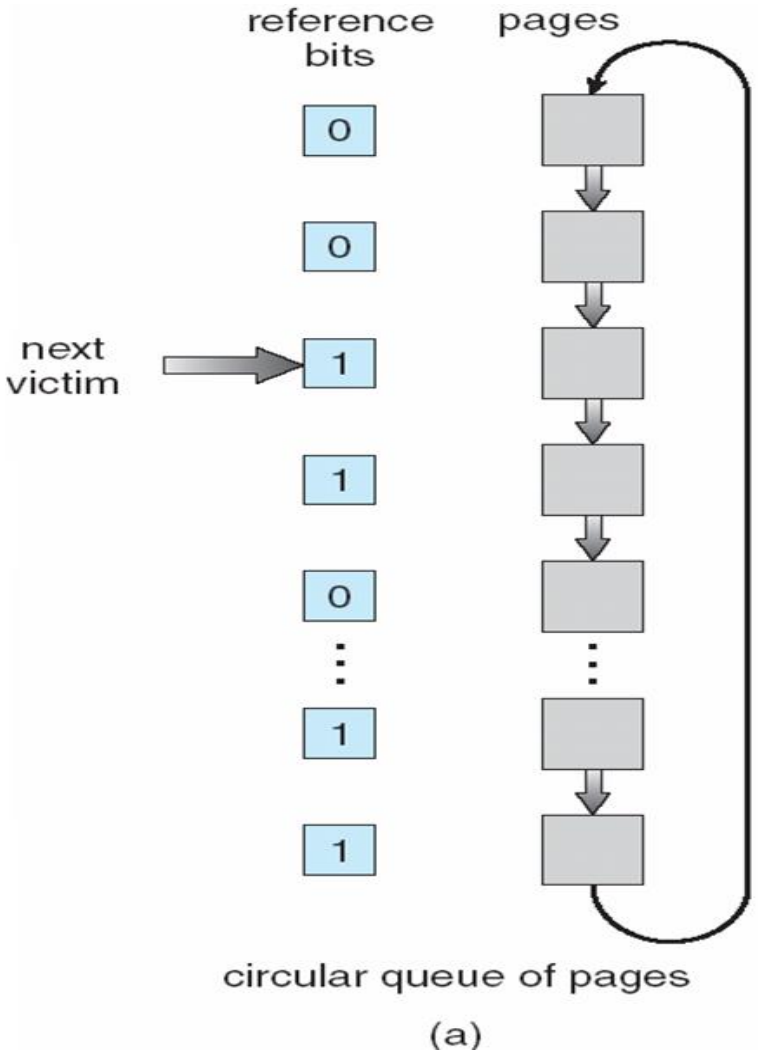
Reference Byte Example

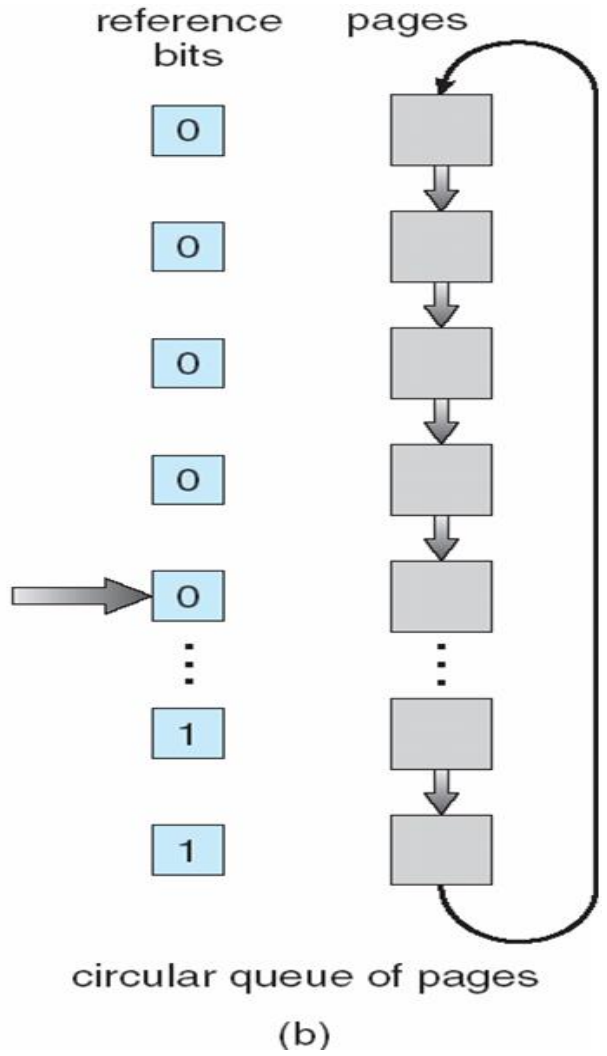


The Clock (Second Chance) Policy

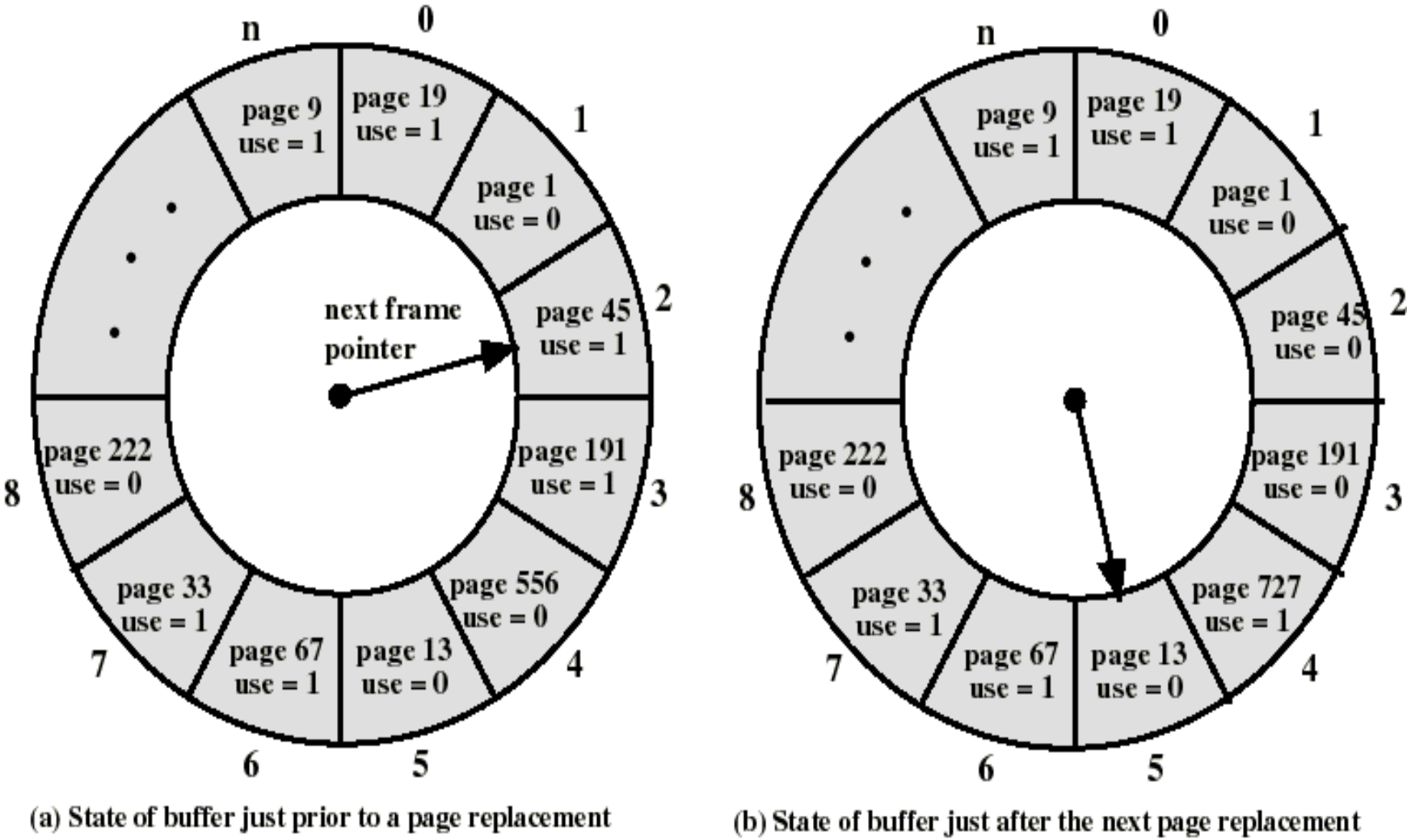
- The set of frames candidate for replacement is considered as a circular buffer.
- When a page is replaced, a pointer is set to point to the next frame in buffer.
- A reference bit for each frame is set to 1 whenever:
  - a page is first loaded into the frame.
  - the corresponding page is referenced.
- When it is time to replace a page, the first frame encountered with the reference bit set to 0 is replaced:
  - During the search for replacement, each reference bit set to 1 is changed to 0.

Clock Page-Replacement Algorithm





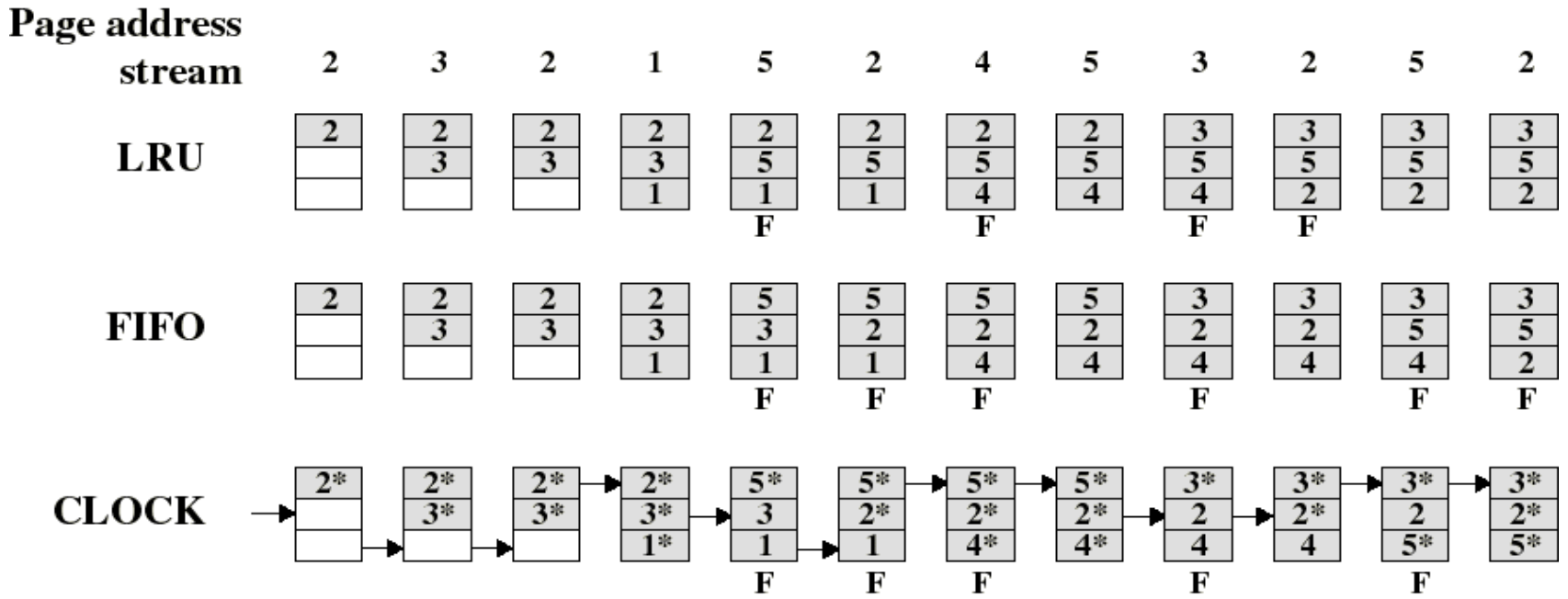
The Clock Policy: Another Example



(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement

Comparison of Clock with FIFO and LRU (1)

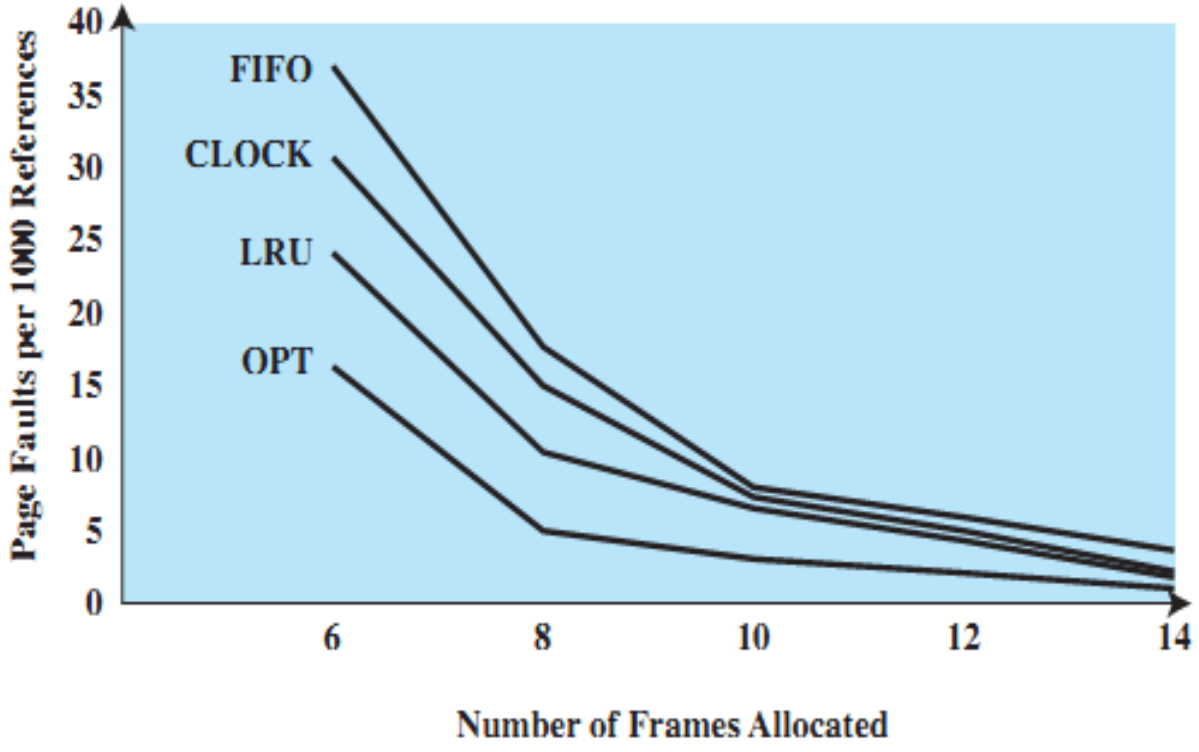


- Asterisk indicates that the corresponding use bit is set to 1.
- The arrow indicates the current position of the pointer.
- Note that the clock policy is adept at protecting frames 2 and 5 from replacement.

Comparison of Clock with FIFO and LRU (2)

- Numerical experiments tend to show that performance of Clock is close to that of LRU.
- Experiments have been performed when the number of frames allocated to each process is fixed and when pages local to the page-fault process are considered for replacement:
  - When few (6 to 8) frames are allocated per process, there is almost a factor of 2 of page faults between LRU and FIFO.
  - This factor reduces close to 1 when several (more than 12) frames are allocated. (But then more main memory is needed to support the same level of multiprogramming).

Fixed-Allocation, Local Page Replacement



Counting-based Algorithms

- Keep a counter of the number of references that have been made to each page.
- Two possibilities: Least/Most Frequently Used (LFU/MFU).
- LFU Algorithm: replaces page with smallest count; others were and will be used more.
- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Page Buffering (1)

- Pages to be replaced are kept in main memory for a while to guard against poorly performing replacement algorithms such as FIFO.
- Two lists of pointers are maintained: each entry points to a frame selected for replacement:
  - a free page list for frames that have not been modified since brought in (no need to swap out).
  - a modified page list for frames that have been modified (need to write them out).
- A frame to be replaced has a pointer added to the tail of one of the lists and the present bit is cleared in corresponding page table entry; but the page remains in the same memory frame.

Page Buffering (2)

- At each page fault the two lists are first examined to see if the needed page is still in main memory:
  - If it is, we just need to set the present bit in the corresponding page table entry (and remove the matching entry in the relevant page list).
  - If it is not, then the needed page is brought in, it is placed in the frame pointed by the head of the free frame list (overwriting the page that was there); the head of the free frame list is moved to the next entry.
  - (the frame number in the page table entry could be used to scan the two lists, or each list entry could contain the process id and page number of the occupied frame).
- The modified list also serves to write out modified pages in cluster (rather than individually).

Cleaning Policy (1)

- When should a modified page be written out to disk?
- Demand cleaning:
  - a page is written out only when it’s frame has been selected for replacement
    - but a process that suffers a page fault may have to wait for 2 page transfers.
- Pre-cleaning:
  - modified pages are written before their frames are needed so that they can be written out in batches:
    - but makes little sense to write out so many pages if the majority of them will be modified again before they are replaced.

Cleaning Policy (2)

- A good compromise can be achieved with page buffering:
  - recall that pages chosen for replacement are maintained either on a free (unmodified) list or on a modified list.
  - pages on the modified list can be periodically written out in batches and moved to the free list.
  - a good compromise since:
    - not all dirty pages are written out but only those chosen for replacement.
    - writing is done in batch.

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Virtual Memory Policies

Memory Management Software

- Memory management software depends on whether the hardware supports paging or segmentation or both.
- Pure segmentation systems are rare. Segments are usually paged -- memory management issues are then those of paging.
- We shall thus concentrate on issues associated with paging.
- To achieve good performance we need a low page fault rate.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Virtual Memory Policies

- Need to decide on:
  - Fetch policy
  - Placement policy
  - Replacement policy
  - Resident Set Management
  - Load Control

Fetch Policy

- Determines when a page should be brought into main memory. Two common policies:
  1. Demand Paging only brings pages into main memory when a reference is made to a location on the page (i.e., paging on demand only):
     - many page faults when process first starts but should decrease as more pages are brought in.
  2. Prepaging brings in pages whose use is anticipated:
     - locality of references suggest that it is more efficient to bring in pages that reside contiguously on the disk.
     - efficiency not definitely established: the extra pages brought in are “often” not referenced.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Placement Policy

- Determines where in real memory a process piece resides.
- For paging (and paged segmentation):
  - the hardware decides where to place the page:
    the chosen frame location is irrelevant since all memory frames are equivalent (not an issue).
- For pure segmentation systems:
  - first-fit, next fit... are possible choices (a real issue).

Replacement Policy

- Deals with the selection of a page in main memory to be replaced when a new page is brought in.
- This occurs whenever main memory is full (no free frame available).
- Occurs often since the OS tries to bring into main memory as many processes (pages) as it can to increase the multiprogramming level.

Replacement Policy

- Not all pages in main memory can be selected for replacement.
- Some frames are locked (cannot be paged out):
  - much of the kernel is held on locked frames as well as key control structures and I/O buffers.
- The OS might decide that the set of pages considered for replacement should be:
  - limited to those of the process that has suffered the page fault.
  - the set of all pages in unlocked frames.

Replacement Policy

- The decision for the set of pages to be considered for replacement is related to the resident set management strategy:
  - how many page frames are to be allocated to each process? We will discuss this later.
- No matter what is the set of pages considered for replacement, the replacement policy deals with algorithms that will choose the page within that set.

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Resident Set Management

- The OS must decide how many page frames to allocate to a process:
  - large page fault rate if too few frames are allocated.
  - low multiprogramming level if too many frames are allocated.

Resident Set Size

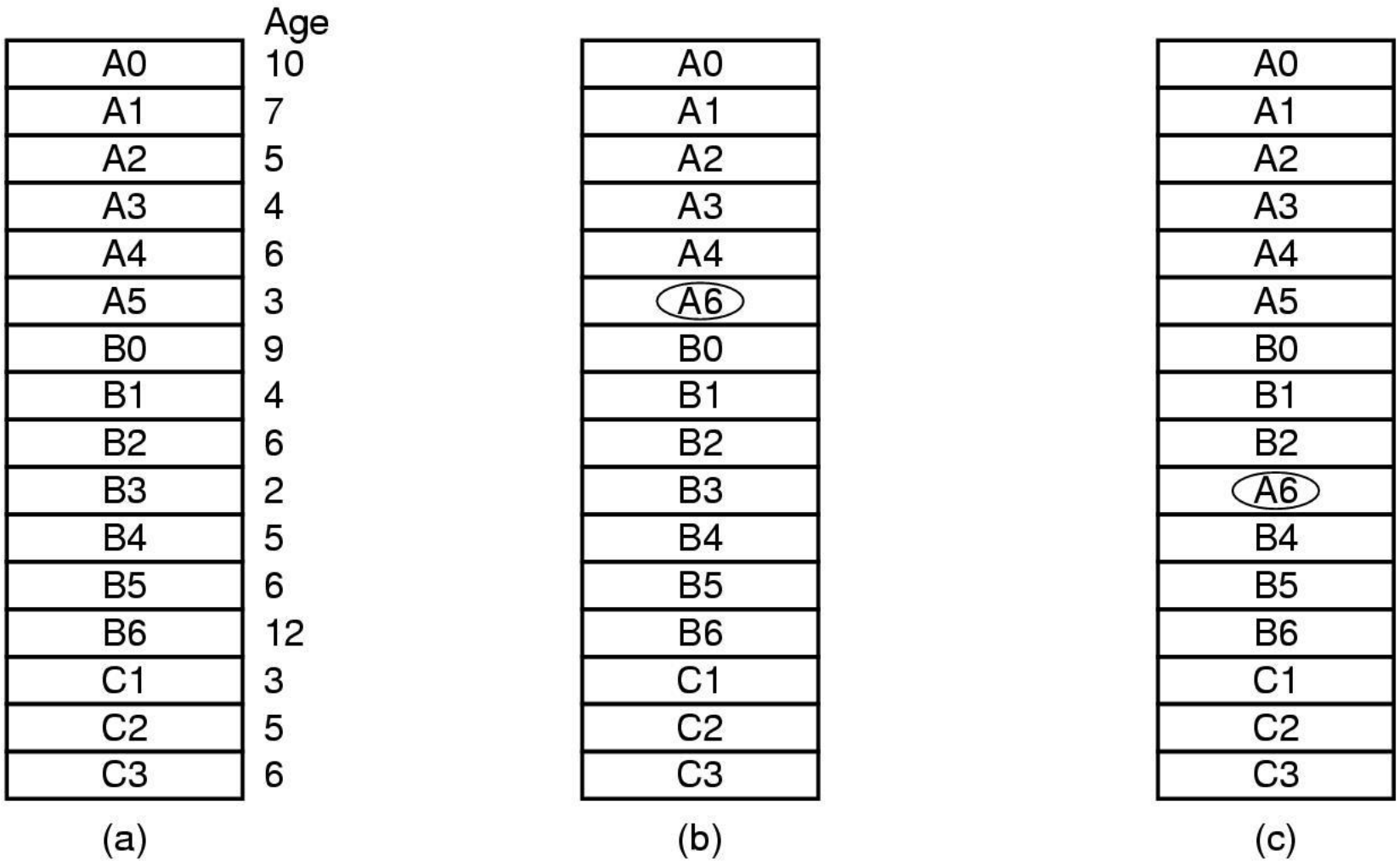
- Fixed-allocation policy:
  - allocates a fixed number of frames that remains constant over time:
    - the number is determined at load time and depends on the type of the application.
- Variable-allocation policy:
  - the number of frames allocated to a process may vary over time:
    - may increase if page fault rate is high.
    - may decrease if page fault rate is very low.
  - requires more OS overhead to assess behavior of active processes.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Replacement Scope

- Replacement scope is the set of frames to be considered for replacement when a page fault occurs.
- Local replacement policy:
  - each process selects from only its own set of allocated frames.
- Global replacement policy:
  - process selects a replacement frame from the set of all frames; one process can take a frame from another.
- We will consider the possible combinations of replacement scope and resident set size policy.

Local vs. Global Replacement Policies



ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Fixed allocation + Local scope

- Each process is allocated a fixed number of pages:
  - determined at load time; depends on application type.
- When a page fault occurs, page frames considered for replacement are local to the page-fault process:
  - the number of frames allocated is thus constant.
  - previous replacement algorithms can be used.
- Problem: difficult to determine ahead of time a good number for the allocated frames:
  - if too low: page fault rate will be high.
  - if too large: multiprogramming level will be too low.

Fixed Allocation: Equal/Proportional

- Equal allocation – for example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – allocate according to the size of process.

  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Fixed Allocation: Priority

- Use a proportional allocation scheme using priorities rather than size.
- If process P_i generates a page fault:
  - select for replacement one of its frames.
  - select for replacement a frame from a process with lower priority number.

Fixed allocation + Global scope

- Impossible to achieve:
  - if all unlocked frames are candidate for replacement, the number of frames allocated to a process will necessary vary over time.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Variable allocation + Global scope

- Simple to implement -- adopted by many OSs (like Unix SVR4).
- A list of free frames is maintained:
  - when a process issues a page fault, a free frame (from this list) is allocated to it.
  - Hence the number of frames allocated to a page fault process increases.
  - The choice for the process that will loose a frame is arbitrary; It's far from optimal.
- Page buffering can alleviate this problem since a page may be reclaimed if it is referenced again soon.

Variable allocation + Local scope

- May be the best combination (used by Windows NT).
- Allocate at load time a certain number of frames to a new process based on application type:
  - use either pre-paging or demand paging to fill up the allocation.
- When a page fault occurs, select the page to replace from the resident set of the process that suffered the page fault.
- Reevaluate periodically the allocation provided and increase or decrease it to improve the overall performance.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

The Working Set Strategy

- The working set strategy is a variable-allocation method with local scope based on the assumption of locality of references.
- The working set for a process at time t, W(D,t), is the set of pages that have been referenced in the last D virtual time units:
  - virtual time = time elapsed while the process was in execution.
  - D is a window of time.
  - W(D,t) is an approximation of the program's locality.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

The Working Set Strategy

- The working set of a process first grows when it starts executing.
- Then stabilizes by the principle of locality.
- It grows again when the process enters a new locality (transition period):
  - up to a point where the working set contains pages from two localities.
- Then decreases after a sufficient long time spent in the new locality.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

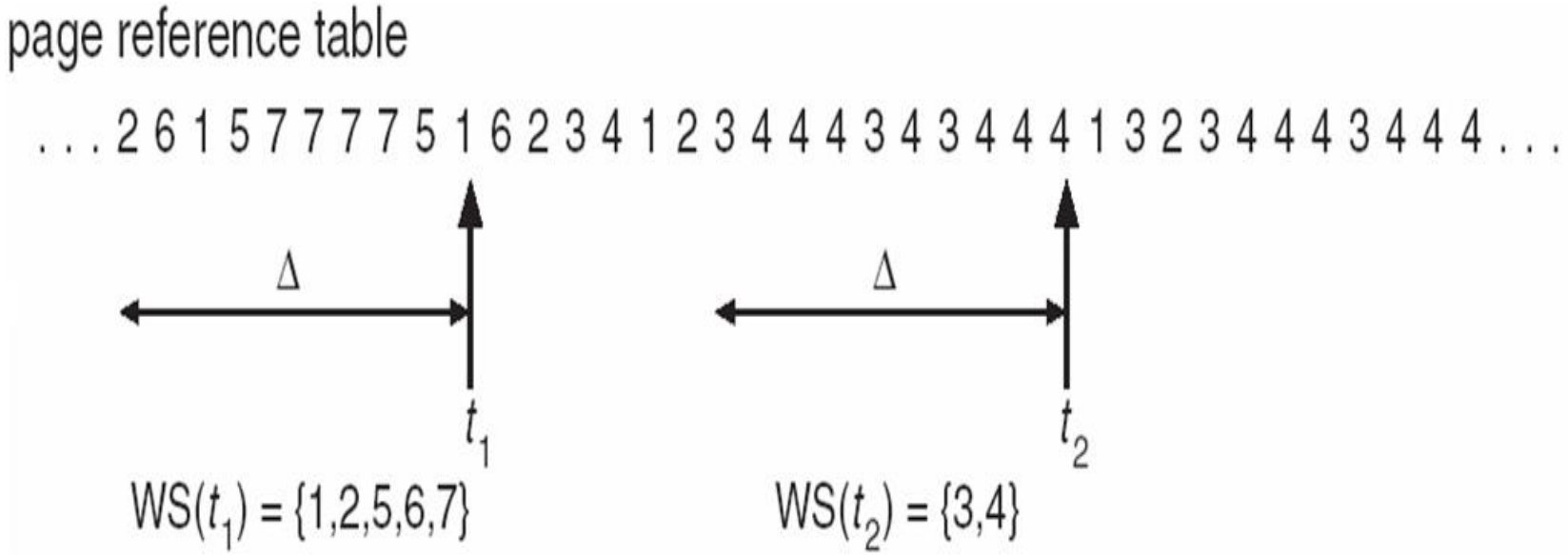
The Working Set Strategy

- The working set concept suggests the following strategy to determine the resident set size:
  - Monitor the working set for each process.
  - Periodically remove from the resident set of a process those pages that are not in the working set.
  - When the resident set of a process is smaller than its working set, allocate more frames to it:
    - If not enough free frames are available, suspend the process (until more frames are available), i.e., a process may execute only if its working set is in main memory.

Working-Set Model

- Δ = working-set window = a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ
  (varies in time):
  - if Δ too small it will not encompass entire locality.
  - if Δ too large it will encompass several localities.
  - if $\Delta = \infty \Rightarrow$ will encompass entire program.
- $D = \Sigma\ WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend one of the processes

Working-set model



Keeping Track of the Working Set

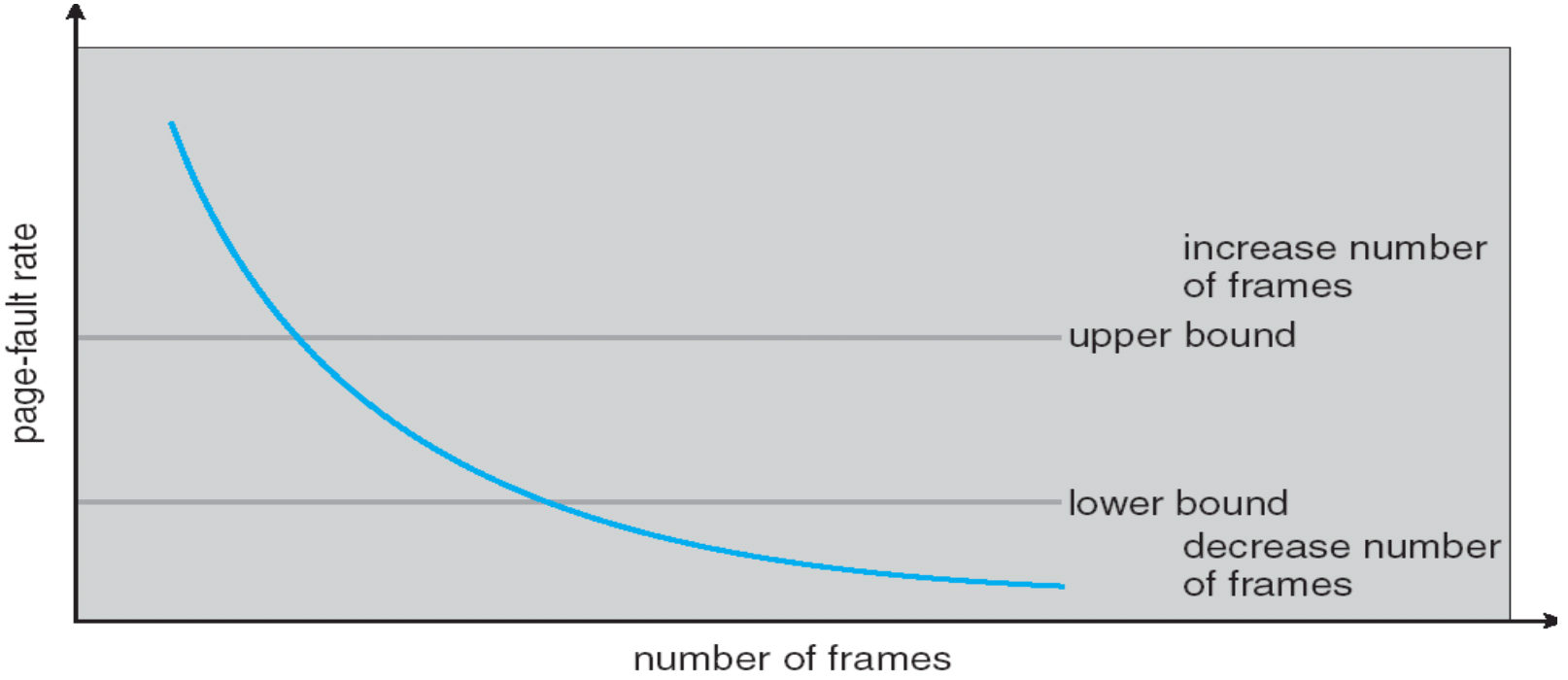
- Approximate with interval timer + a reference bit.
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts, copy and set the values of all reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units.

The Working Set Strategy

- Practical problems with this working set strategy:
  - measurement of the working set for each process is impractical:
    - necessary to time stamp the referenced page at every memory reference.
    - necessary to maintain a time-ordered queue of referenced pages for each process.
  - the optimal value for D is unknown and time varying.
- Solution: rather than monitor the working set, monitor the page fault rate!

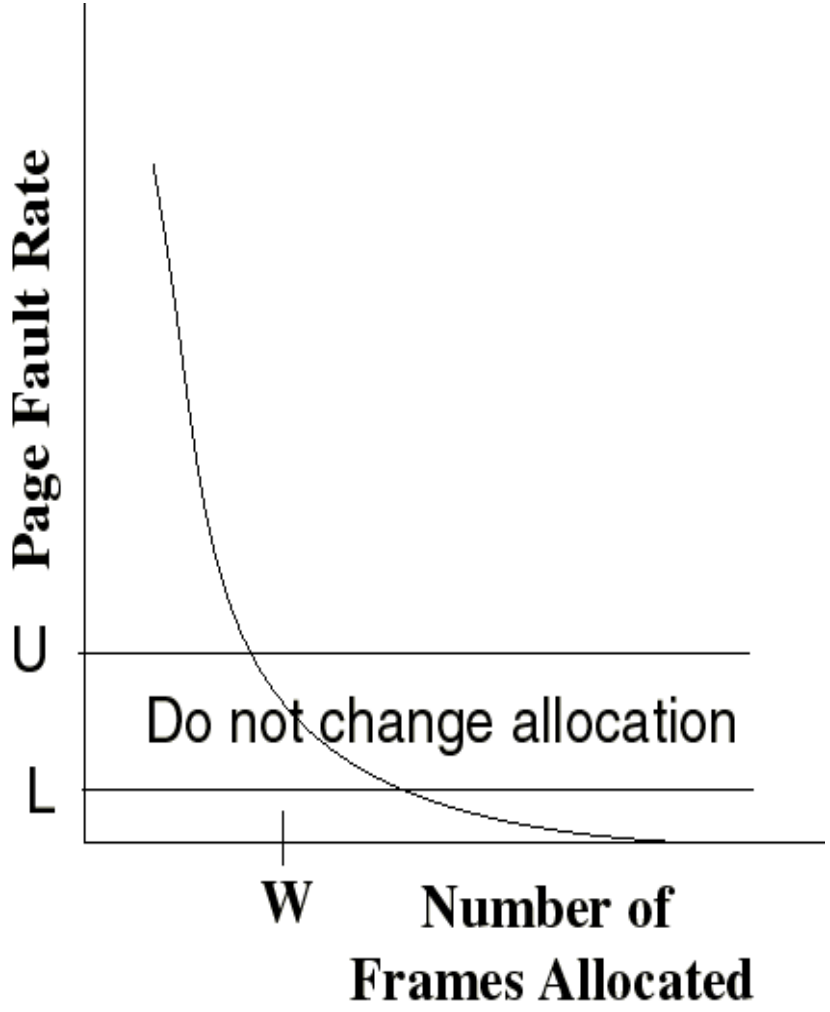
Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate:
  - If actual rate too low, process loses frame.
  - If actual rate too high, process gains frame.

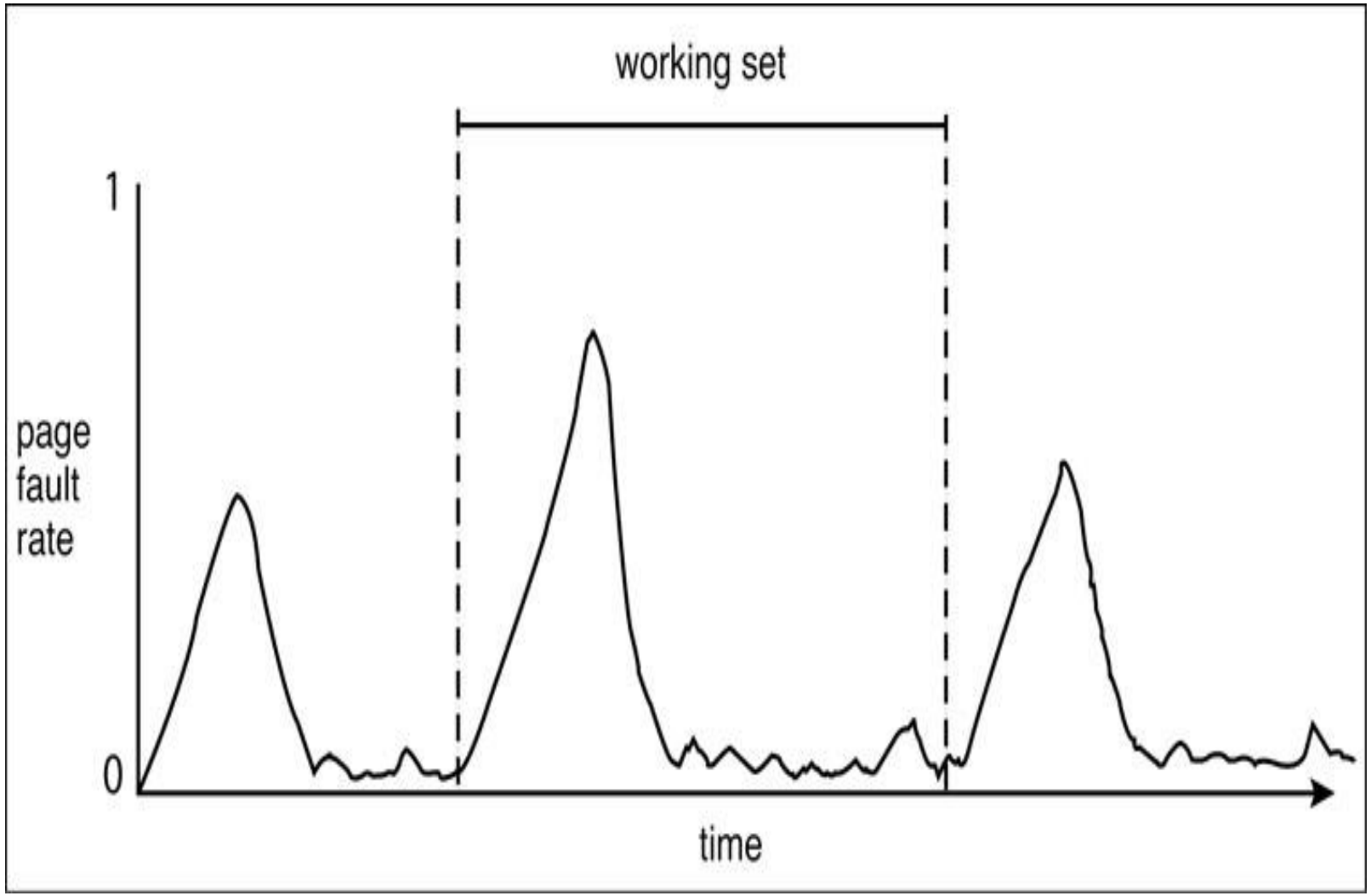


Page-Fault Frequency (PFF) Strategy

- Define an upper bound U and lower bound L for page fault rates.
- Allocate more frames to a process if fault rate is higher than U.
- Allocate less frames if fault rate is less than L.
- The resident set size should be close to the working set size W.
- We suspend the process if the PFF > U and no more free frames are available.

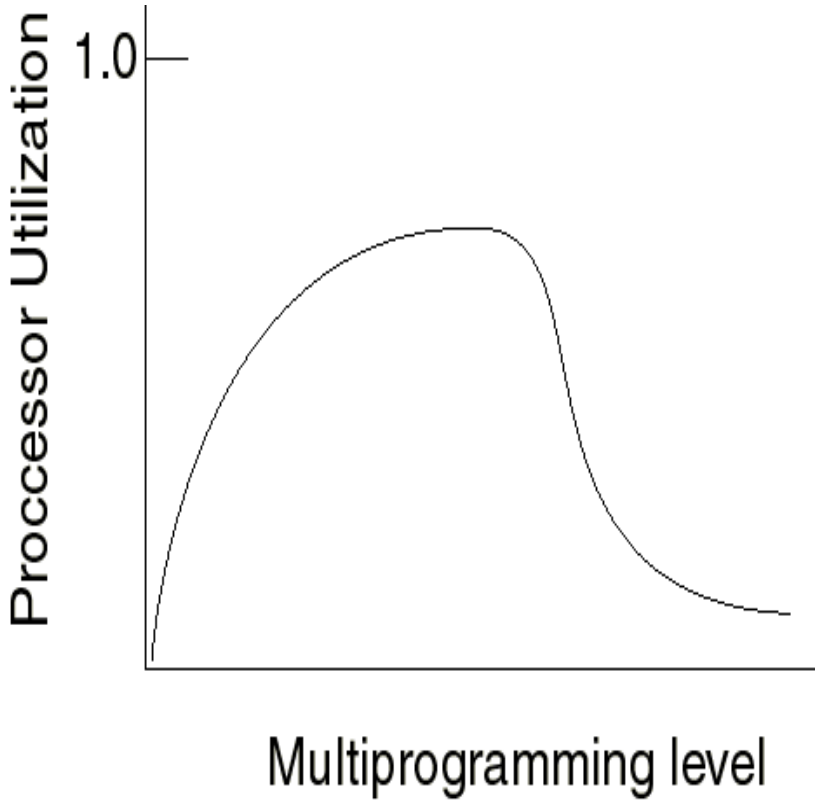


Working Sets and Page-Fault rates



Load Control

- Determines the number of processes that will be resident in main memory (i.e., the multiprogramming level):
  - Too few processes: often all processes will be blocked and the processor will be idle.
  - Too many processes: the resident size of each process will be too small and flurries of page faults will result: thrashing.



Load Control

- A working set or page fault frequency algorithm implicitly incorporates load control:
  - only those processes whose resident set is sufficiently large are allowed to execute.
- Another approach is to adjust explicitly the multiprogramming level so that the mean time between page faults equals the time to process a page fault:
  - performance studies indicate that this is the point where processor usage is at maximum.

Process Suspension

- Explicit load control requires that we sometimes swap out (suspend) processes.
- Possible victim selection criteria:
  - Faulting process
    - this process may not have its working set in main memory so it will be blocked anyway.
  - Last process activated
    - this process is least likely to have its working set resident.
  - Process with smallest resident set
    - this process requires the least future effort to reload.
  - Largest process
    - will yield the most free frames.

Windows XP (1)

- Uses demand paging with **clustering**.
- Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory.
- A process may be assigned as many pages up to its working set maximum.

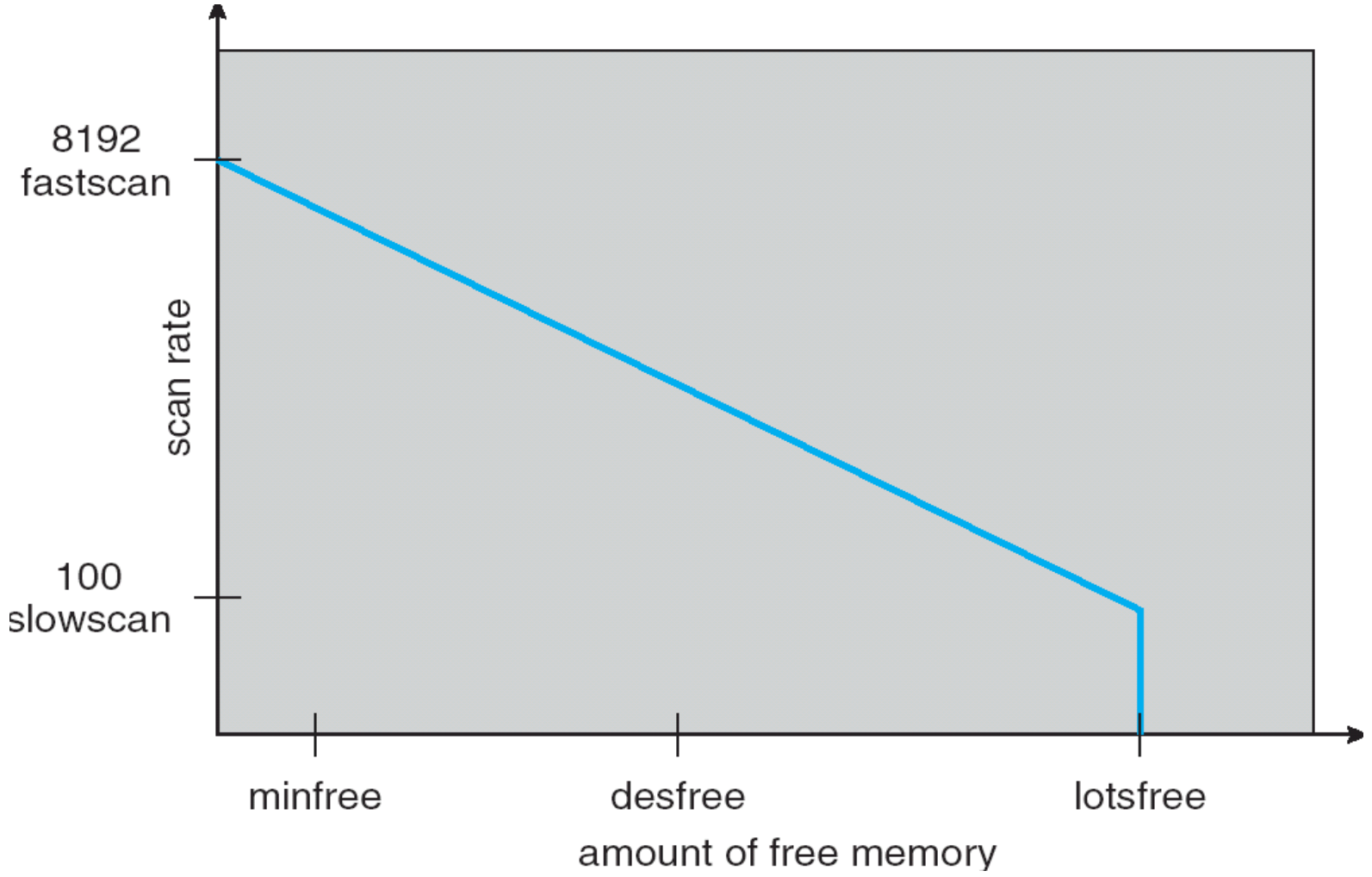
Windows XP (2)

- 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.
- Working set trimming removes pages from processes that have pages in excess of their working set minimum.

Solaris

- Maintains a list of free pages to assign faulting processes.
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging.
- *Desfree* – threshold parameter to increasing paging.
- *Minfree* – threshold parameter to being swapping.
- Paging is performed by *pageout* process.
- Pageout scans pages using modified clock algorithm.
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan.*
- Pageout is called more frequently depending upon the amount of free memory available.

Solaris 2 Page Scanner



SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY