Memory Management

Week 08 – Lecture

Paging

Team

- Instructor
  - Giancarlo Succi
- Teaching Assistants
  - Nikita Lozhnikov (also Tutorial Instructor)
  - Manuel Rodriguez
  - Shokhista Ergasheva

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013
    Prentice-Hall, Inc.

  and customised for the needs of this course.
- Additional input for the slides are detailed later

Page Replacement Algorithms (1)

- Recap:
  - There are two status bits, *R* and *M*, associated with each page
  - *R* is set whenever the page is referenced
  - *M* is set when the page is modified
  - These bits must be updated on every memory reference, usually by hardware
  - Once a bit has been set to 1, it stays 1 until the operating system resets it

Page Replacement Algorithms (2)

- When a page fault occurs, the OS has to choose a page to evict (remove from memory) to make room for the incoming page
- If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date
- If the page has not been changed (e.g., it contains program text) no rewrite is needed

Page Replacement Algorithms (3)

- Optimal algorithm
- Not recently used algorithm
- First-in, first-out (FIFO) algorithm
- Second-chance algorithm
- Clock algorithm
- Least recently used (LRU) algorithm
- Working set algorithm
- WSClock algorithm

Optimal Algorithm (1/2)

- Easy to describe but impossible to actually implement:
  - At the moment that a page fault occurs, some set of pages is in memory
  - One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until *n* instructions later
  - Each page can be labeled with the number of instructions that will be executed before that page is first referenced
  - The page with the highest *n* should be removed

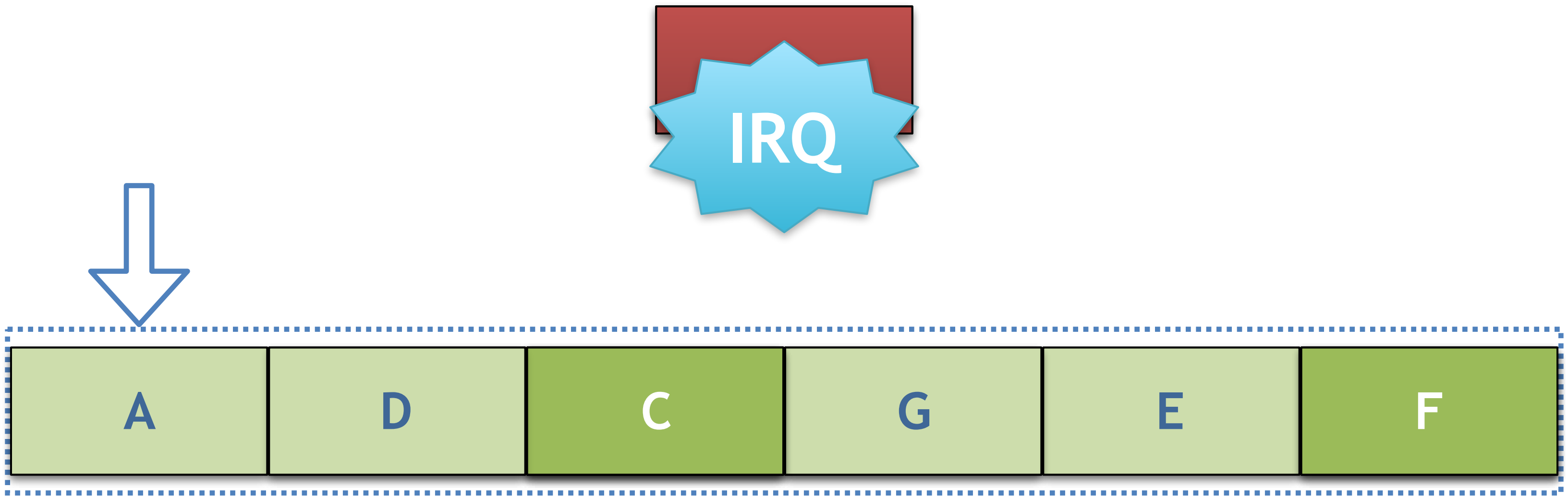
Optimal Algorithm (2/2)

- At the time of the page fault, the OS does not know when each of the pages will be referenced next
- We can run a program on a simulator and track all the page references so it would be possible to implement optimal page replacement algorithm during the second run
- **The result can only be used to compare the performance of real page replacement algorithms with the optimal one**

Not Recently Used (1/3)

- Categories of pages based on the current values of their *R* and *M* bits:
  - Class 0: not referenced, not modified.
  - Class 1: not referenced, modified.
  - Class 2: referenced, not modified.
  - Class 3: referenced, modified.
- At page fault, the NRU algorithm inspects pages and removes a page at random from the **lowest-numbered nonempty class**

Not Recently Used (2/3)



R = 0; M = 0	R = 0; M = 1	R = 1; M = 0	R = 1; M = 1

NRU algorithm: memory reference sequence

A-B-C-D-E-F-G-F(M)-C(M)-(IRQ)-C(R)-D

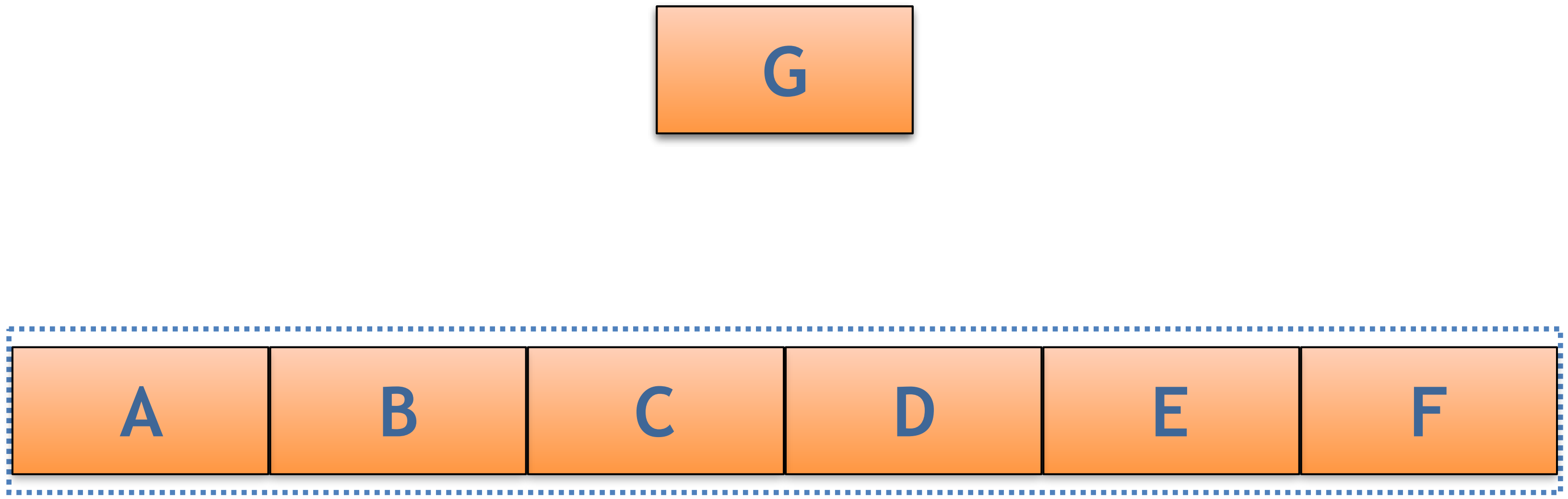
Not Recently Used Algorithm (3/3)

- Main idea:
  - it is better to remove a **modified page that has not been referenced in at least one clock tick** than a **clean page that is in heavy use**
- Advantages:
  - easy to understand
  - moderately efficient to implement
  - gives an adequate performance

First-In, First-Out (FIFO) (1/2)

- Low-overhead paging algorithm
- The OS maintains a list of all pages currently in memory, with the most recent arrival at the tail and the least recent arrival at the head
- On a page fault, the page at the head is removed and the new page added to the tail of the list
- Problem: the oldest page may still be useful

First-In, First-Out (FIFO) (2/2)

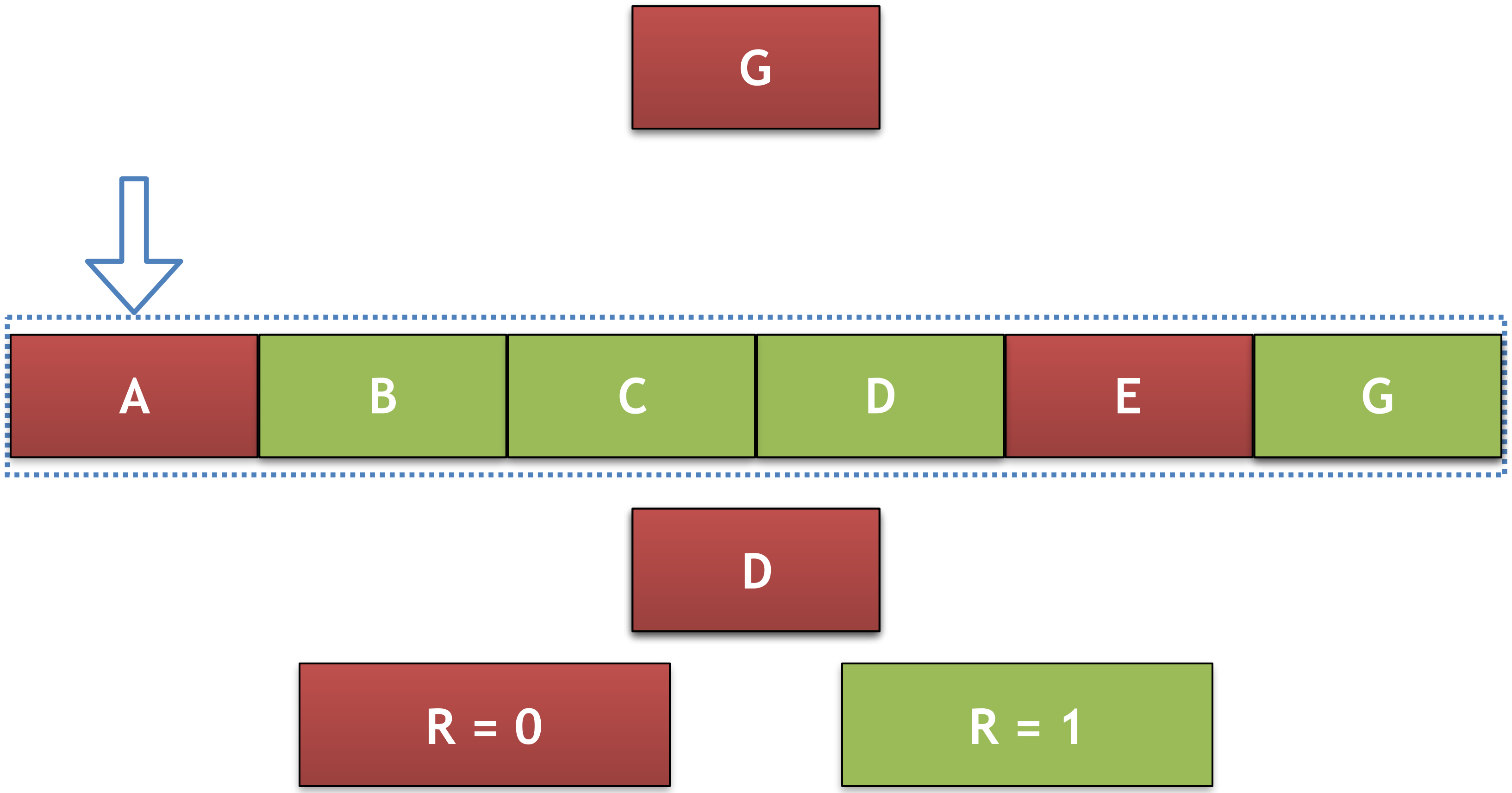


FIFO algorithm: memory reference sequence
A-B-C-D-E-F-G-A

Second-Chance Algorithm (1/2)

- We can modify FIFO to avoid the problem of throwing out a heavily used page. It can be done by inspecting the *R* bit of the oldest page:
  - 0: the page is both old and unused, so it is replaced immediately
  - 1: the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated

Second-Chance Algorithm (2/2)

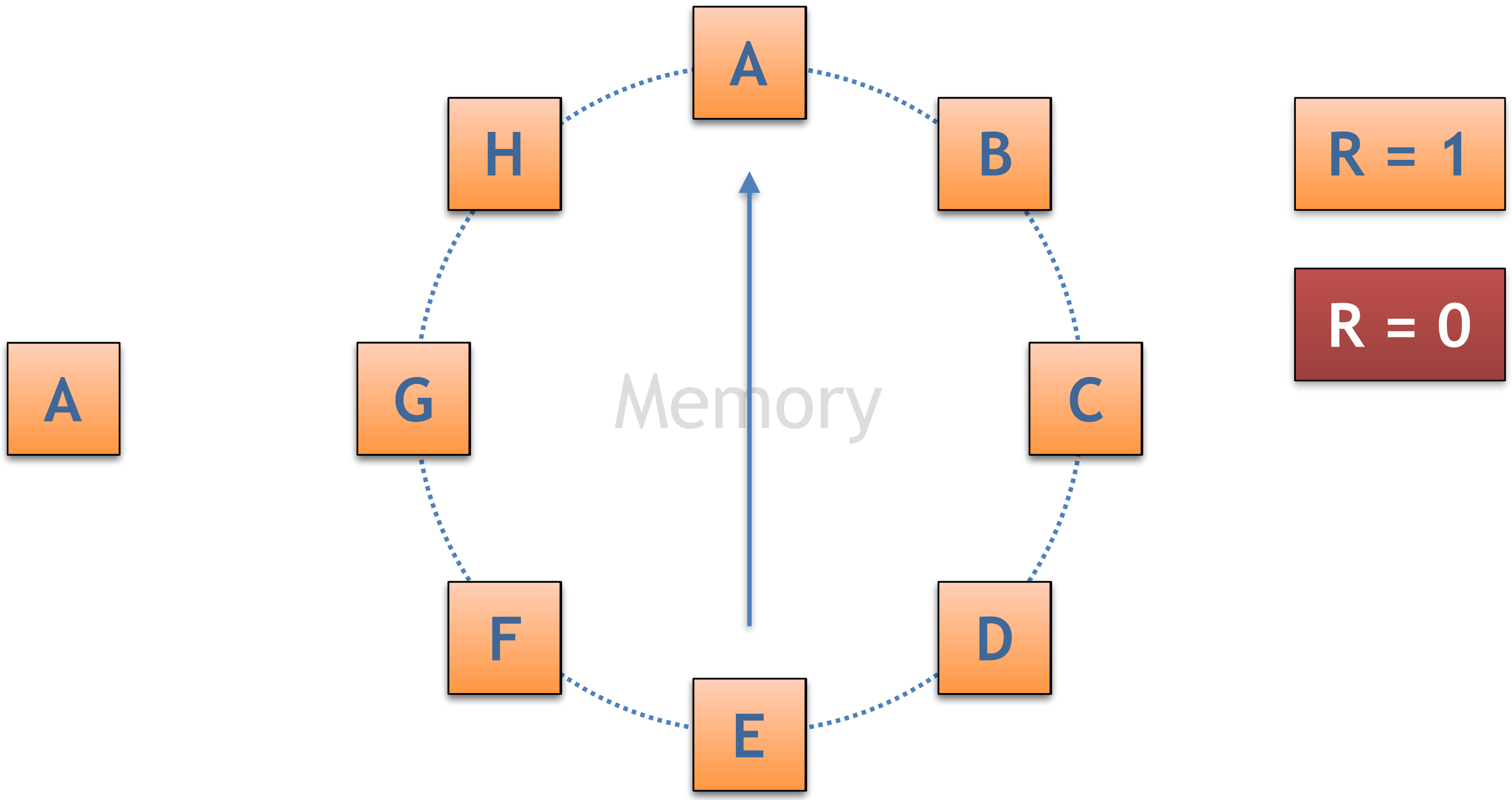


FIFO algorithm: memory reference sequence
A-B-C-D-E-F-G-A

Clock Algorithm (1/2)

- Since second chance algorithm is constantly moving pages around, it is not so effective. However, there is another approach (Fig. 3-15):
  - To keep pages on a circular list and to have a “clock hand” that points to the oldest page
  - When a page fault occurs, the page being pointed to by the hand is inspected
  - If its *R* bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position
  - If *R* is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page with *R* = 0 is found

Clock Algorithm (2/2)



Clock algorithm: memory reference sequence A-B-C-D-E-F-G-H-I-C-F-D-G-A

Least Recently Used (LRU) (1/2)

- Observation:
  - Pages that have been heavily used in the last few instructions will probably be heavily used again soon and vice versa
- A realizable algorithm:
  - When a page fault occurs, throw out the page that has been unused for the longest time

Least Recently Used (LRU) (2/2)

- Not cheap to implement:
  - It is necessary to maintain a linked list of all pages in memory, with the MRU page at the front and the LRU page at the rear
  - The list must be updated on every memory reference
  - A very time consuming operation, even in hardware

Hardware-Assisted Implementation of LRU

- Special 64-bit counter C is automatically incremented after each instruction
- Each page table entry must have a field large enough to contain the counter
- After each memory reference, the current value of C is stored in the page table entry for the page just referenced
- Page with the lowest counter value is the least recently used

Simulating LRU in Software

- Few, if any, machines have the required hardware to implement LRU
- It is possible to simulate it in software. Two examples are:
  - NFU (Not Frequently Used)
  - Aging (modification of NFU)

Not Frequently Used (1)

- NFU requires a software counter associated with each page, initially zero
- At each clock interrupt the OS scans all the pages in memory and updates the counter by adding *R* bit to it
- When a page fault occurs, the page with the lowest counter is chosen for replacement

Not Frequently Used (2)

- The problem example:
  - In a multipass compiler, pages that were heavily used during pass 1 may still have a high count during subsequent passes
  - If pass 1 has the longest execution time of all the passes, the pages containing the code for subsequent passes may always have lower counts
  - The OS will remove useful pages instead of pages no longer in use

Aging Algorithm (1)

- A small modification to NFU (Fig. 3-17):
  - The counters are each shifted right 1 bit before the *R* bit is added in
  - The *R* bit is added to the leftmost rather than the rightmost bit
- Another difference is that in aging the counters have a finite number of bits so it is not possible to distinguish pages that were referenced 9 or 1000 ticks ago
- In practice 8 bits is generally enough if a clock tick is around 20 msec

Aging Algorithm (2)

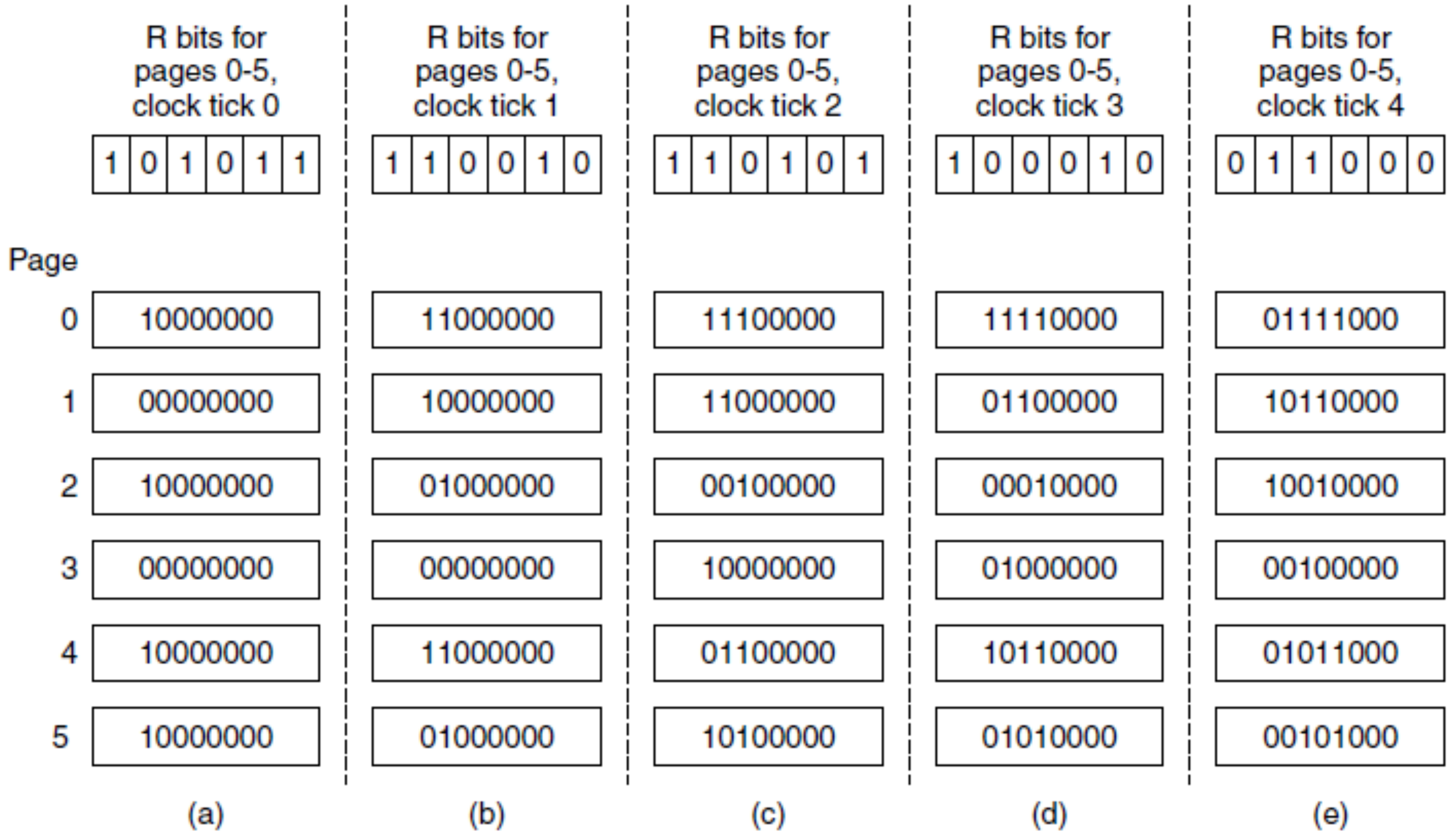


Figure 3-17. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

Working Set Algorithm (1)

- **Demand paging:**
  - Processes are started up with no pages in memory
  - As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the OS to bring in the page containing the first instruction
  - Other page faults for global variables and the stack usually follow quickly
  - After a while, the process has most of the pages it needs and settles down to run with relatively few page faults

Working Set Algorithm (2)

- **Locality of reference:**
  - During any phase of execution, the process references only a relatively small fraction of its pages
- **Working set of a process:**
  - The set of pages that a process is currently using
- **Thrashing:**
  - A program’s behavior when it is causing page faults every few instructions

Working Set Algorithm (3)

- Processes are often moved to disk to let others have a turn at the CPU
- The question is what needs to be done when the process is brought back
- It is possible to use demand paging, however, it is slow and wastes CPU time
- **Working set model (prepaging)**:
  - To keep track of each process' working set and make sure that it is in memory before letting the process run

Working Set Algorithm (4)

- At any instant of time *t* there exists a set consisting of all the pages used by the *k* most recent memory references (or page references)
- **The set *w(k, t)* is the working set**
- Because the *k = 1* most recent references must have used all the pages used by the *k > 1* most recent references, and possibly others, *w(k, t)* is a monotonically nondecreasing function of *k* (Fig. 3-18)

Working Set Algorithm (5)

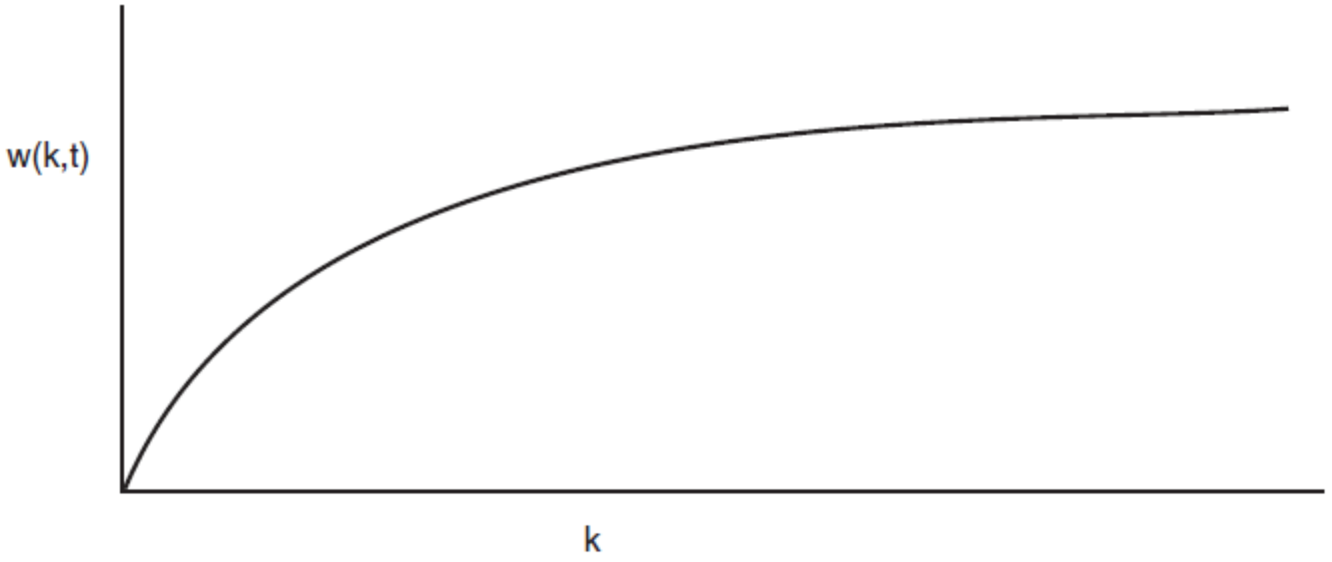


Figure 3-18. The working set is the set of pages used by the *k* most recent memory references. The function *w(k, t)* is the size of the working set at time *t*.

Working Set Algorithm (6)

- To implement the working set model, it is necessary for the OS to keep track of which pages are in the working set
- A possible page replacement algorithm: when a page fault occurs, find a page not in the working set and evict it

Working Set Algorithm (7)

- Various approximations are used to implement the algorithm. One idea is to use execution time instead of counting back memory references:
  - we could define a working set as the set of pages used during the past 100 msec of execution time
  - if a process starts running at time *T* and has had 40 msec of CPU time at real time *T* + 100 msec, for working set purposes its time is 40 msec (**process's current virtual time**)

Working Set Algorithm (8)

- The algorithm:
  - Each page table entry contains (at least) two key items of information: the approximate time the page was last used and the *R* bit
  - A periodic clock interrupt is assumed to cause software to run that clears the *R* bit on every clock tick
  - On every page fault, the page table is scanned to look for a suitable page to evict which may cause the next results:

Working Set Algorithm (9)

- As each entry is processed, the *R* bit is examined
- If it is 1, the current virtual time is written into the *Time of last use* field in the page table
- If *R* is 0, the page has not been referenced during the current clock tick. To see whether or not it should be removed, its age (the *current virtual time* minus its *Time of last use*) is computed and compared to τ
- If *R* is 0 but the age is less than or equal to τ, the page is still in the working set
- If all pages are in the working set, the one with *R* = 0 and with the greatest age is evicted
- In the worst case, all pages have *R* = 1, so one is chosen at random for removal, preferably a clean page, if one exists

Working Set Algorithm (10)

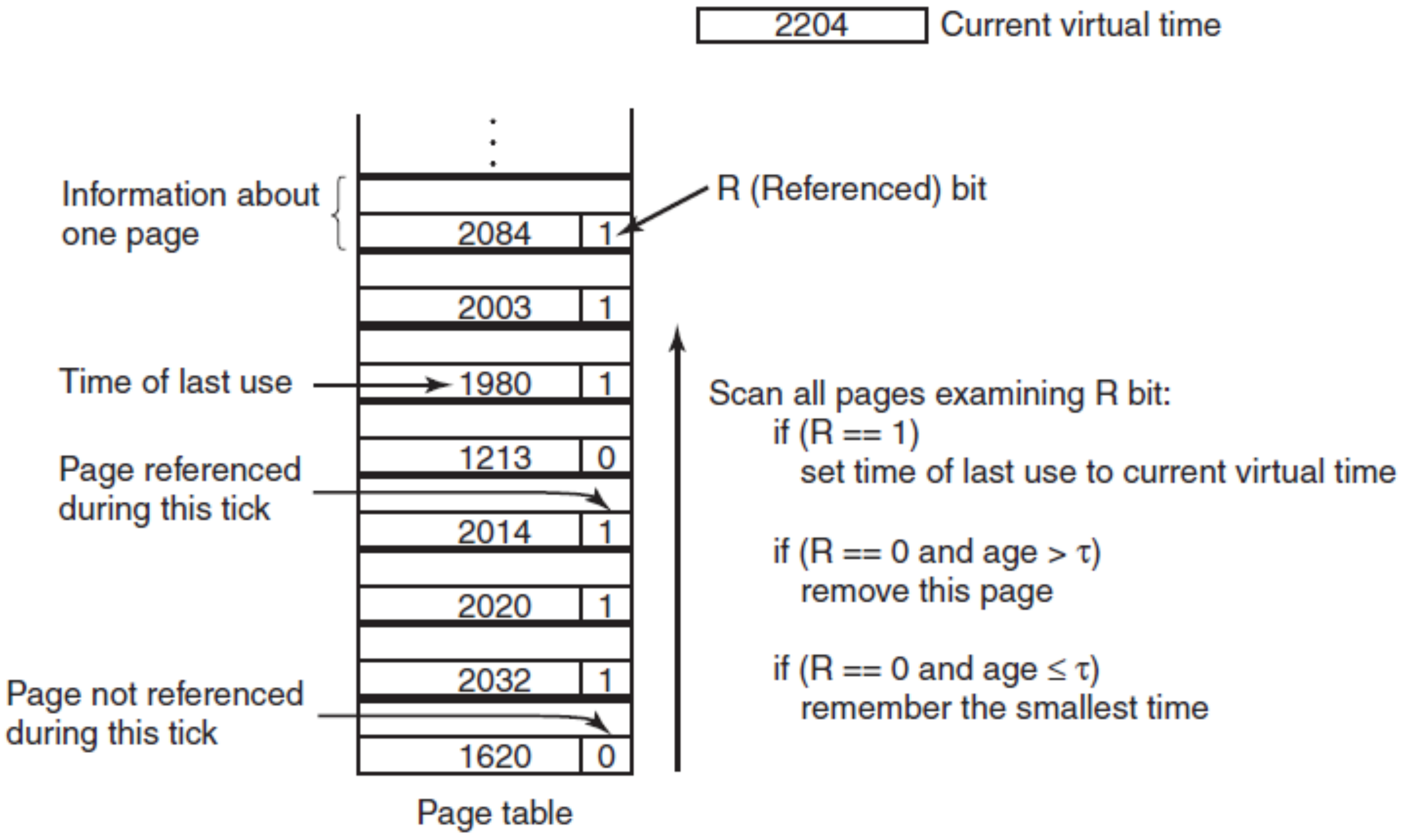

2204
Current virtual time
Information about one page
R (Referenced) bit
Time of last use
Page referenced during this tick
Page not referenced during this tick
2084 1
2003 1
1980 1
1213 0
2014 1
2020 1
2032 1
1620 0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

Figure 3-19. The working set algorithm.

WSClock Algorithm (1)

- An algorithm which is based on the clock algorithm but also uses the working set information
- Is widely used in practice since it is simple and provides good performance
- The data structure is a circular, initially empty, list of page frames, as in the clock algorithm (Fig. 3-20a). As the pages are added, they go into the list to form a ring
- Each entry contains the *Time of last use* field from the basic working set algorithm, as well as the *R* and *M* bits

WSClock Algorithm (2)

- At each page fault the page pointed to by the hand is examined first:
  - If the *R* = 1, the page has been used during the current tick so it is not an ideal candidate to remove
  - The *R* bit is then set to 0, the hand advanced to the next page, and the algorithm repeated for that page (Fig. 3-20b)

WSClock Algorithm (3)

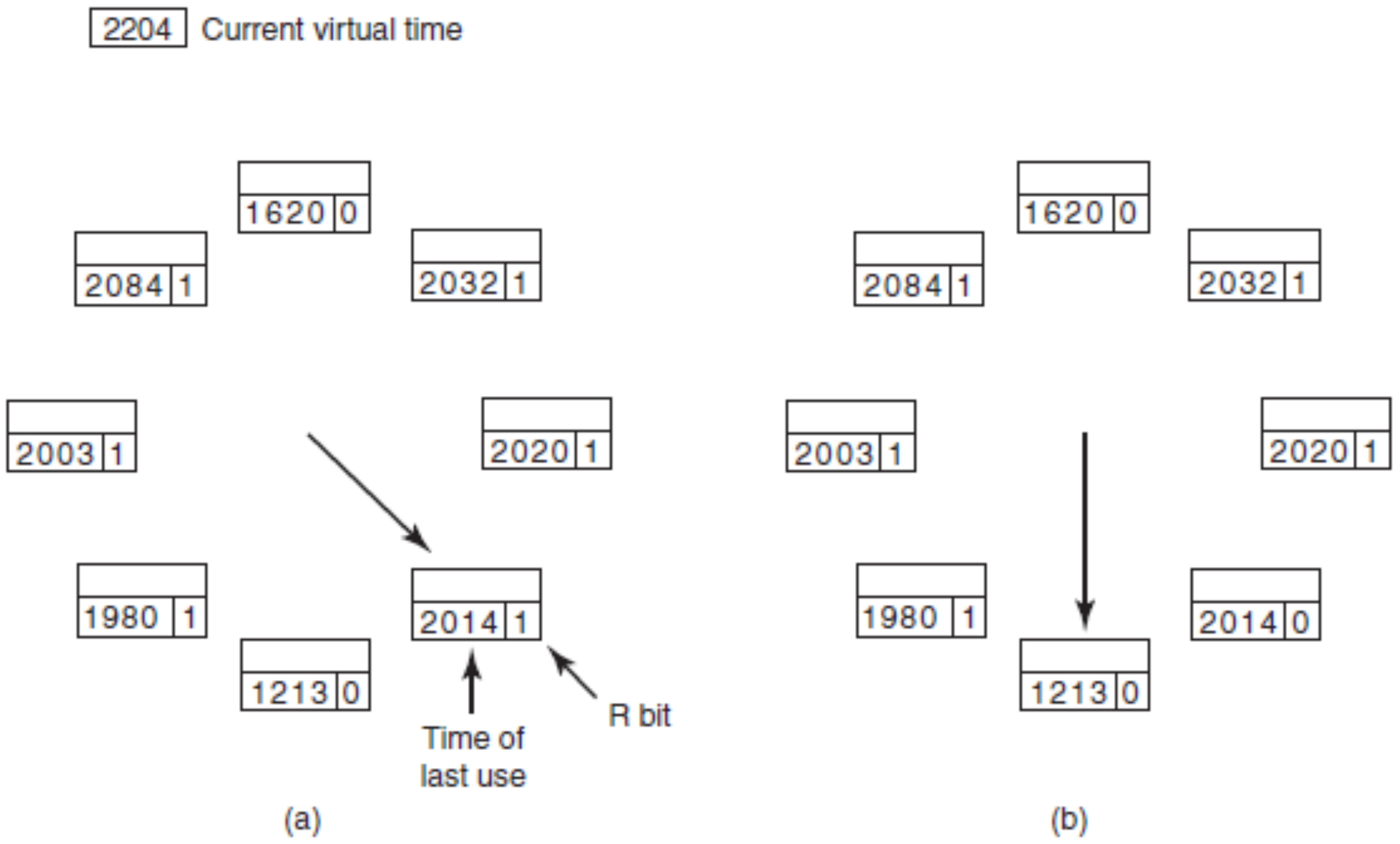


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when *R* = 1.

WSClock Algorithm (4)

- If $R = 0$ (Fig. 3-20c), the age is greater than τ and the page is clean, the page frame is simply claimed and the new page put there (Fig. 3-20d)
- If the page is dirty, it cannot be claimed immediately since no valid copy is present on disk
- To avoid a process switch, the write to disk is scheduled, but the hand is advanced and the algorithm continues with the next page

WSClock Algorithm (5)

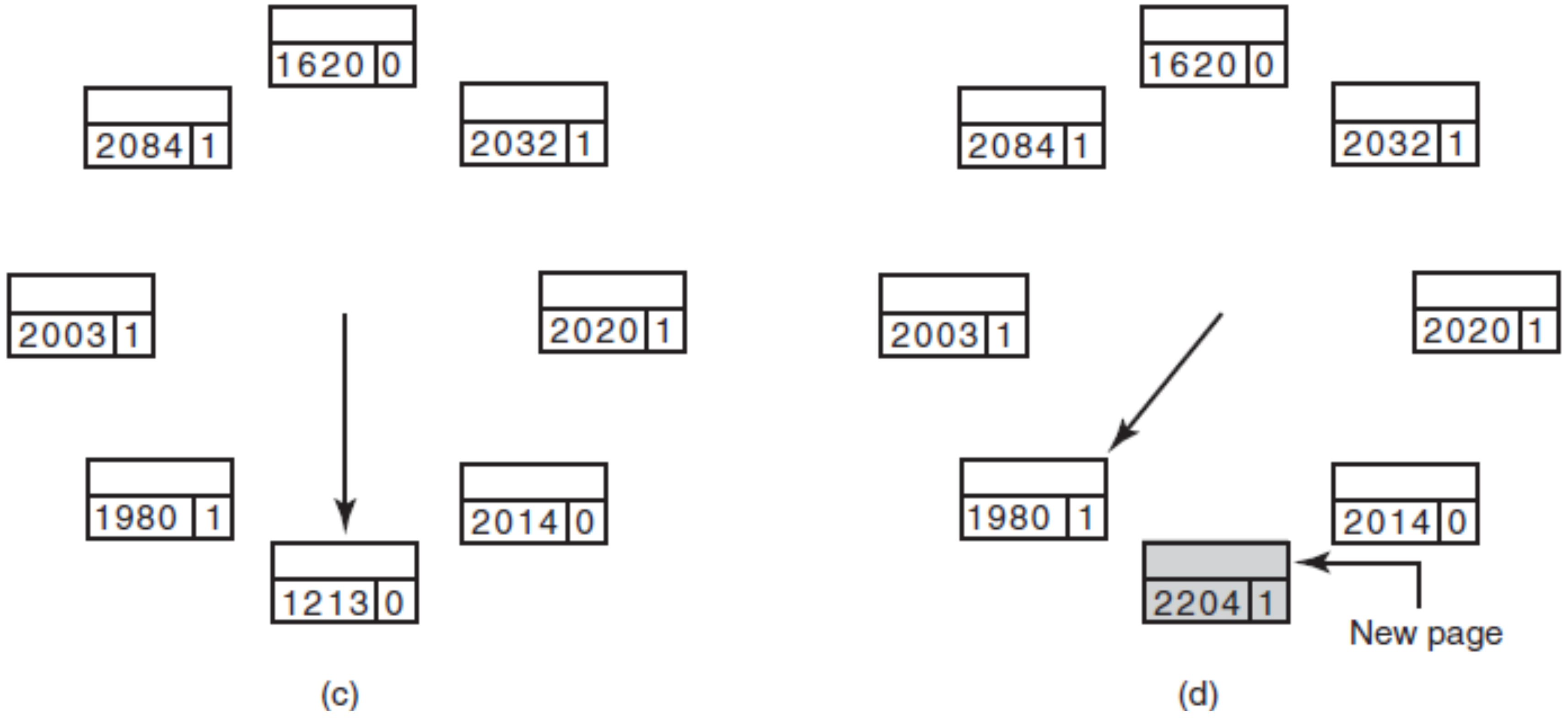
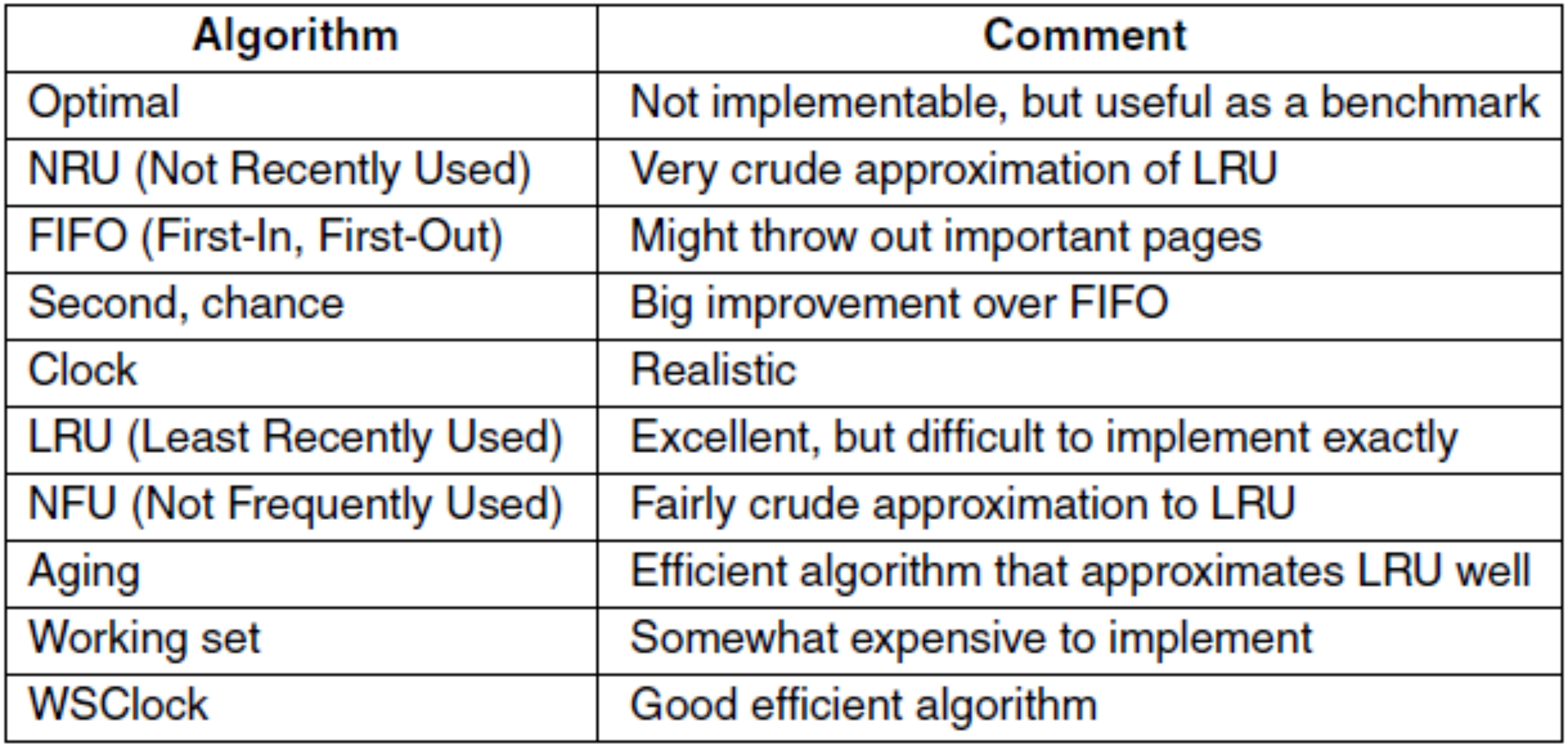


Figure 3-20. Operation of the WSClock algorithm. (c) and (d) give an example of *R* = 0.

WSClock Algorithm (6)

- There are two cases when the hand comes all the way around and back to its starting point:
  - **At least one write has been scheduled.** The hand just keeps moving, looking for a clean page. The first clean page encountered is evicted
  - **No writes have been scheduled.** The simplest thing to do is to claim any clean page and use it. If no clean pages exist, then the current page is chosen as the victim and written back to disk

Summary



Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second, chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Figure 3-21. Summary of page replacement algorithms

Design Issues For Paging Systems

- Local versus Global Allocation Policies
- Load Control
- Page Size
- Separate Instruction and Data Spaces
- Shared Pages
- Shared Libraries
- Mapped Files
- Cleaning Policy
- Virtual Memory Interface

Local vs. Global Allocation Policies (1)

- Example: three processes, A, B, and C, make up the set of runnable processes
- Suppose A gets a page fault. The algorithm can try to find the least recently used page considering either only the pages currently allocated to A or all the pages in memory
- Local Page Replacement: it looks only at A's pages, the page with the lowest age value is A5 (Fig. 3-22b)
- Global Page Replacement: the page with the lowest age value is removed without regard to whose page it is; page B3 will be chosen (Fig. 3-22c)

Local vs. Global Allocation Policies (2)

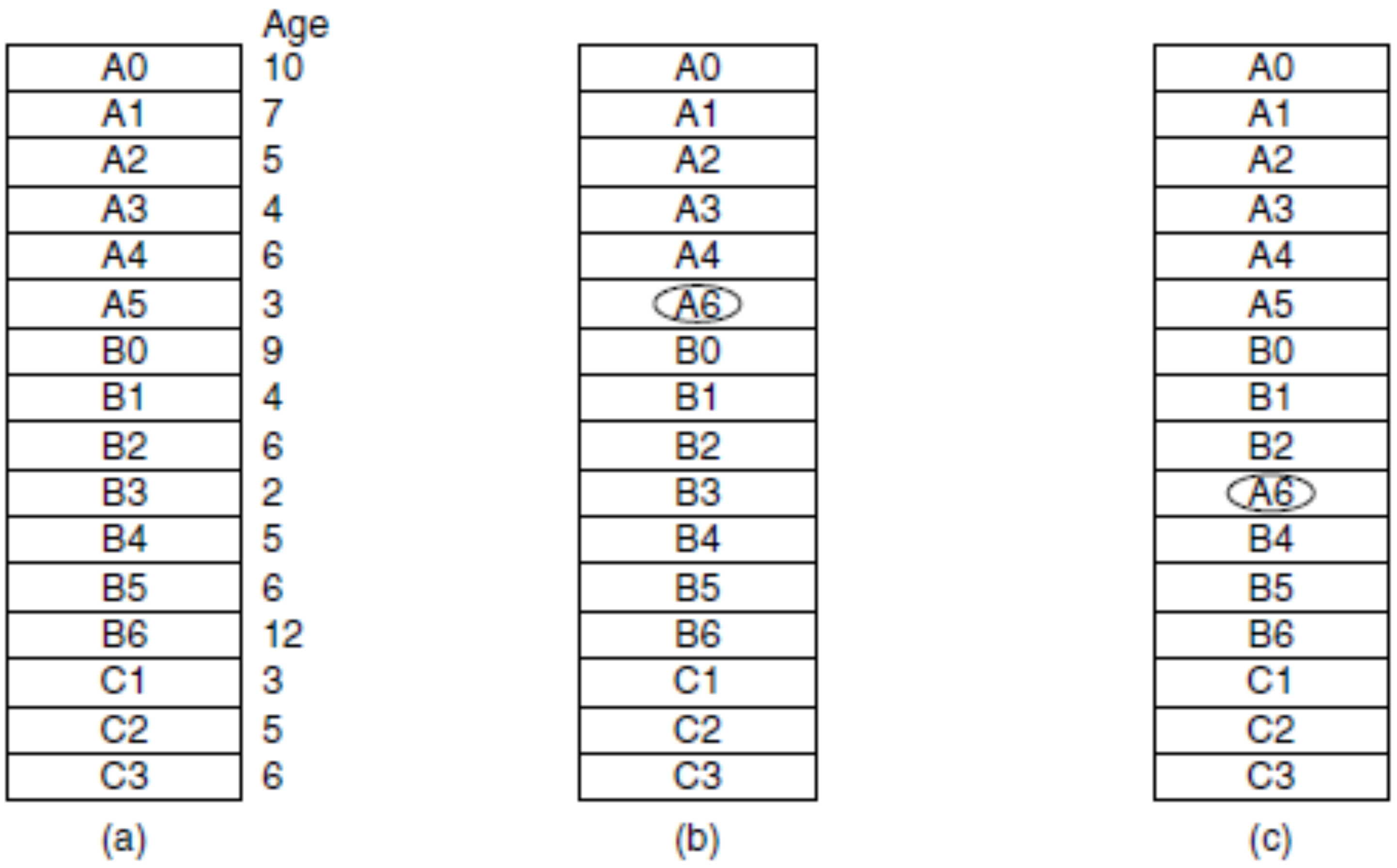


Figure 3-22. Local versus global page replacement.
(a) Original configuration. (b) Local page replacement.
(c) Global page replacement.

Local vs. Global Allocation Policies (3)

- If a local algorithm is used and the working set grows, thrashing will result. If the working set shrinks, local algorithms waste memory
- If a global algorithm is used, the system must continually decide how many page frames to assign to each process

Local vs. Global Allocation Policies (4)

- One way is to monitor the working set size as indicated by the aging bits, but this approach does not necessarily prevent thrashing
- Another approach is to have an algorithm for allocating page frames to processes:
  - to periodically determine the number of running processes and allocate each process an equal share, or
  - to allocate pages in proportion to each process' total size, with a 300-KB process getting 30 times the allotment of a 10-KB process

Local vs. Global Allocation Policies (5)

- If a global algorithm is used, the allocation has to be updated dynamically as the processes run
- One way to manage the allocation is to use the **PFF (Page Fault Frequency)** algorithm that tells when to increase or decrease a process' page allocation
- The assumption behind PFF is that the fault rate decreases as more pages are assigned (Fig. 3-23)

Local vs. Global Allocation Policies (6)

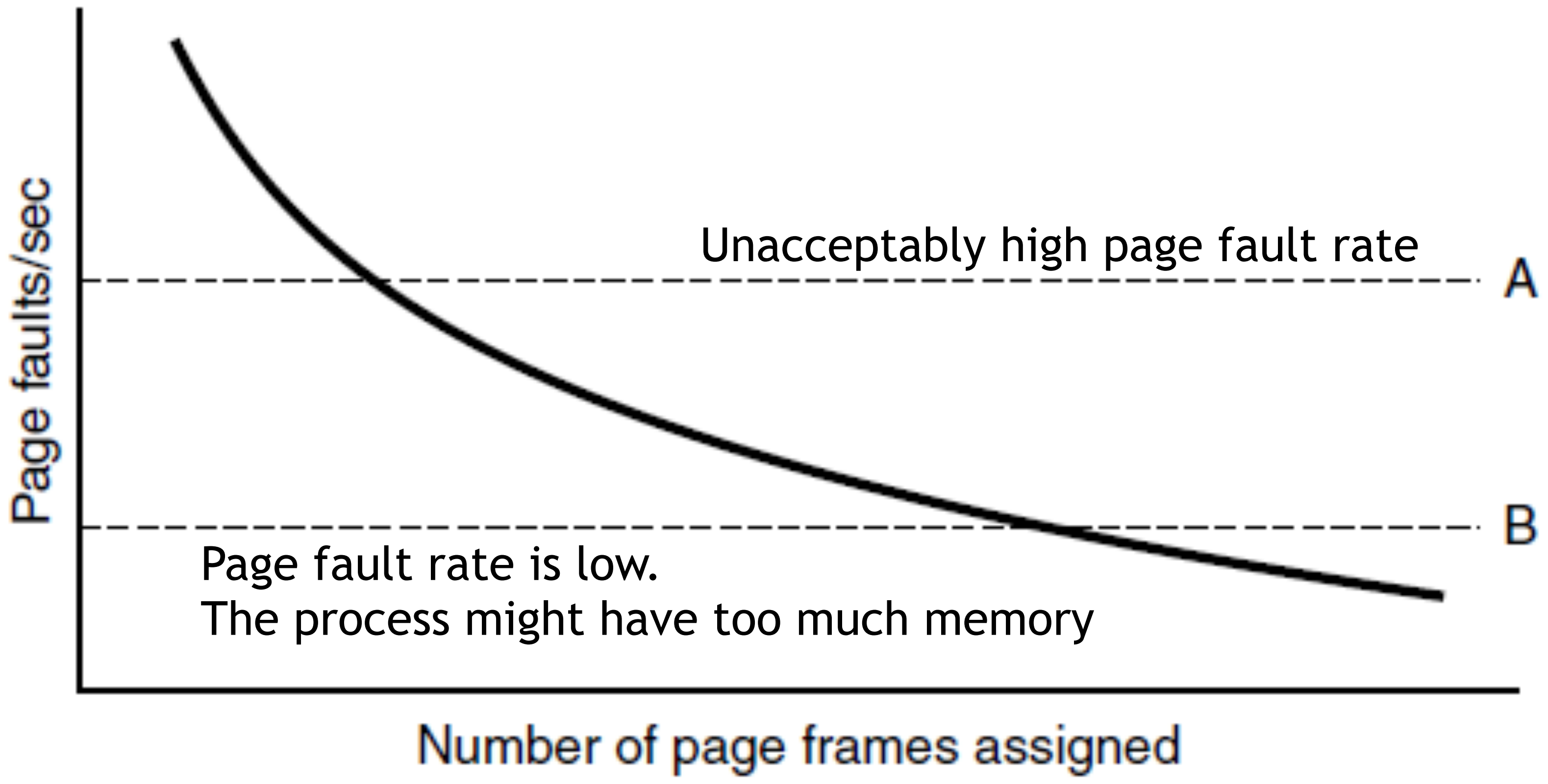


Figure 3-23. Page fault rate as a function of the number of page frames assigned.

Local vs. Global Allocation Policies (7)

- Some page replacement algorithms can work with either a local replacement policy or a global one
- For example, FIFO can replace the oldest page in all memory or the oldest page owned by the current process
- For other page replacement algorithms, only a local strategy makes sense. In particular, **the working set** and **WSClock** algorithms refer to some specific process and must be applied in that context

Load Control (1)

- Even with the best page replacement algorithm and optimal global allocation of page frames to processes, the system might thrash
- One symptom of this situation is that the PFF algorithm indicates that some processes need more memory but no processes need less memory
- In this case, there is no way to give more memory to those processes needing it without hurting some other processes. The only real solution is to temporarily get rid of some processes

Load Control (2)

- Some of the processes may be swapped to the disk to free up all the pages they are holding
- When a process is swapped out, its page frames divided up among other processes that are thrashing
- If the thrashing does not stop, another process has to be swapped out, and so on, until the thrashing stops

Load Control (3)

- However, when the number of processes in main memory is too low, the CPU may be idle for substantial periods of time
- Thus, we need to consider not only process size and paging rate when deciding which process to swap out, but also its characteristics, such as whether it is CPU bound or I/O bound, and what characteristics the remaining processes have

Page Size (1)

- The page size is a parameter that can be chosen by the OS which, for example, can always regard a pair of pages as one page with a double size
- Determining the best page size requires balancing several competing factors and there is no overall optimum

Page Size (2)

- **Small page size. Pros:**
  - It reduces **internal fragmentation** which is a situation when text, data, or stack segment will not fill an integral number of pages, thus leaving a part of a page empty
  - It will help to allocate small programs consisting of several phases. For example, with a 32-KB page size, a program consisting of eight sequential phases of 4 KB each will take 32 KB all the time. The smaller the page size is, the less memory will be wasted

Page Size (3)

- **Small page size. Cons:**
  - Having many pages means a large page table
  - Transferring a small page takes almost as much time as transferring a large page
  - Small pages use up much valuable space in the TLB. A program that uses 1 MB of memory with a working set of 64 KB would occupy at least 16 entries in the TLB if a page size is 4 KB. With 2-MB pages, a single TLB entry would be sufficient

Page Size (4)

- **Optimal page size:**
  - Let the average process size be ***s*** bytes and the page size be ***p*** bytes
  - Assume that each page entry requires ***e*** bytes
  - The approximate number of pages needed per process is ***s/p***, occupying ***s*e/p*** bytes of page table space
  - The wasted memory due to internal fragmentation is ***p/2***

Page Size (5)

- **Optimal page size** (cont.):
  - overhead = $se/p + p/2$
  - The first term is large when the page size is small. The second term is large when the page size is large. By taking the first derivative with respect to ***p*** and equating it to zero, we get the equation
  - $-se/p^2 + 1/2 = 0$
  - $\boldsymbol{p = (2se)^{1/2}}$

Separate Instruction and Data Spaces (1)

- A single address space usually holds both programs and data (Fig. 3-24a)
- If the address space is too small, it forces programmers to invent new ways to fit everything
- One solution is to have separate address spaces for instructions (program text) and data, called **I-space** and **D-space** (Fig. 3-24b)
- Each one has its own page table, with its own mapping of virtual pages to physical page frames

Separate Instruction and Data Spaces (2)

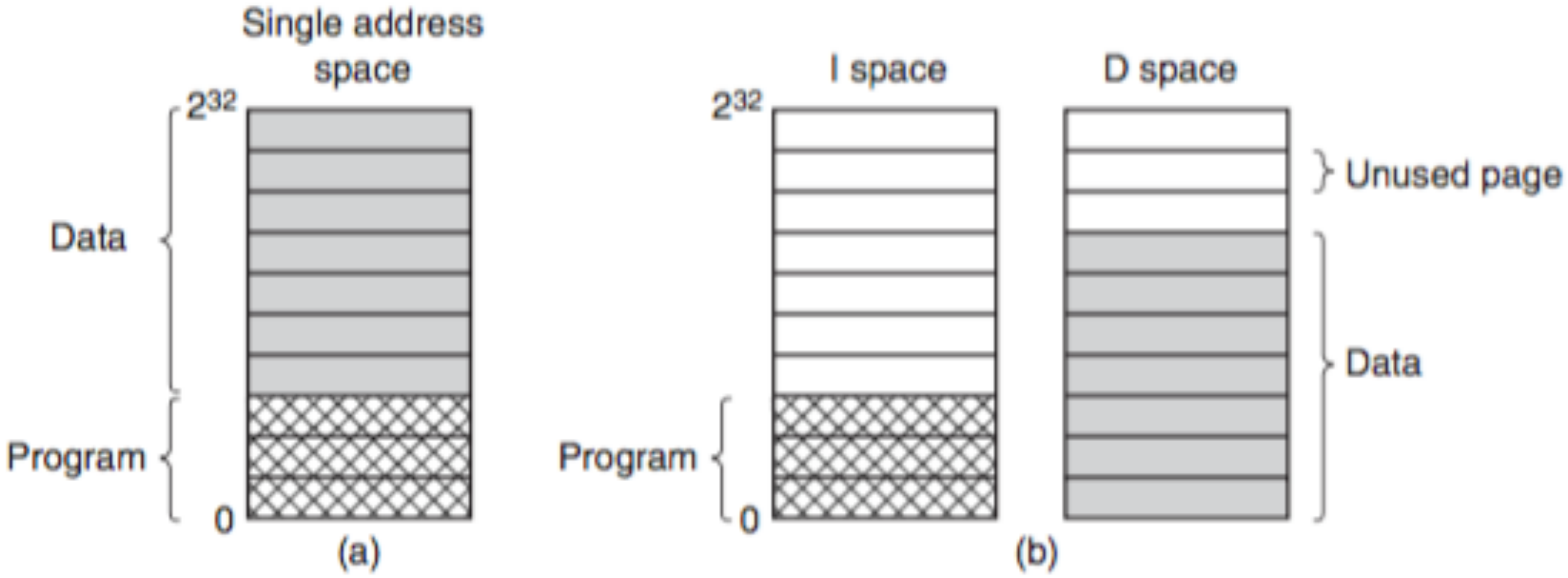


Figure 3-24. (a) One address space.
(b) Separate I and D spaces.

Separate Instruction and Data Spaces (3)

- When the hardware wants to fetch an instruction, it knows that it must use I-space and the I-space page table
- Similarly, data must go through the D-space page table
- Having separate I- and D-spaces does not introduce any special complications for the OS and it does double the available address space.
- Rather than for the normal address spaces, separate I- and D-spaces are used nowadays to divide the L1 cache

Shared Pages (1)

- In a large multiprogramming system, it is common for several users to be running the same program at the same time or a single user may be running several programs that use the same library
- It is not efficient to have two copies of the same page in memory at the same time
- Not all the pages are shareable. Pages that are read-only, such as program text, can be shared, but for data pages sharing is more complicated

Shared Pages (2)

- If separate I- and D-spaces are supported, it is relatively straightforward **to share programs** by having two or more processes use the same page table for their I-space but different page tables for their D-spaces (Fig. 3-25)
- The problem: if the scheduler decides to remove process A from memory, evicting all its pages will cause process B to generate a large number of page faults to bring them back in again

Shared Pages (3)

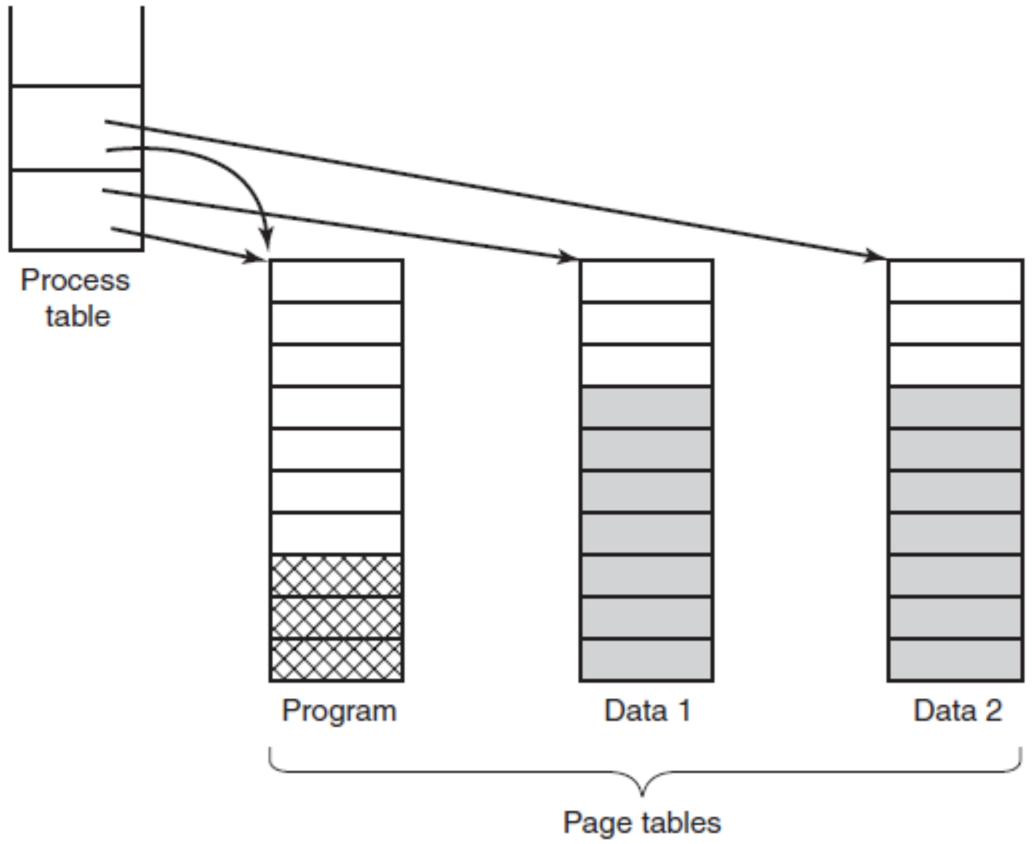


Figure 3-25. Two processes sharing the same program sharing its page table.

Shared Pages (4)

- When process A terminates, it is essential to be able to discover that the pages are still in use
- Searching all the page tables to see if a page is shared is usually too expensive, so special data structures are needed to keep track of shared pages

Shared Pages (5)

- Sharing data is trickier than sharing code, but it is not impossible:
  - In UNIX, after a *fork* system call, the parent and child are required to share both program text and data
  - No copying of pages is done at *fork* time
  - Each of the processes has **its own page table** and both of them point to the **same set of pages**

Shared Pages (6)

- Sharing data (cont.):
  - As soon as either process updates a memory word, the violation of the read-only protection causes a trap to the OS
  - A copy of the offending page is made, so that each process now has its own private copy
  - Both copies are now set to READ/WRITE, so subsequent writes to either copy proceed without trapping (**copy on write**)

Shared Libraries (1)

- In modern systems, there are many large libraries used by many processes, for example, multiple I/O and graphics libraries
- Consider traditional linking first:
  - When a program is linked, all the .o (object) files in the current directory are linked and libraries that are named in the command to the linker are scanned
  - Any functions called in the object files but not present there (e.g., printf) are called **undefined externals** and are sought in the libraries

Shared Libraries (2)

- Traditional linking (cont.):
  - Any functions called in the object files are included in the executable binary if they are found
  - Functions present in the libraries but not called are not included
  - When the linker is done, an executable binary file is written to the disk containing all the functions needed

Shared Libraries (3)

- Common programs use 20–50 MB worth of graphics and user interface functions
- Statically linking hundreds of programs with all these libraries would result in wasting a huge amount of space on the disk and space in RAM
- A common technique helping solve this problem is to use **shared libraries** (or **Dynamic Link Libraries** on Windows)

Shared Libraries (4)

- When a program is linked with shared libraries, the linker includes a **small stub routine** that binds to the called function at run time
- Shared libraries may be loaded either when the program is loaded or when functions in them are called for the first time
- The entire library is paged in into memory, page by page, as needed, so functions that are not called will not be brought into RAM

Shared Libraries (5)

- Another advantage is that it is possible to update a DLL without recompilation of the programs that call it
- However, there is one problem (Fig. 3-26):
  - When two different processes share the same library, it may be located at a different address in each process
  - Since the library is shared, relocation on the fly will not work

Shared Libraries (6)

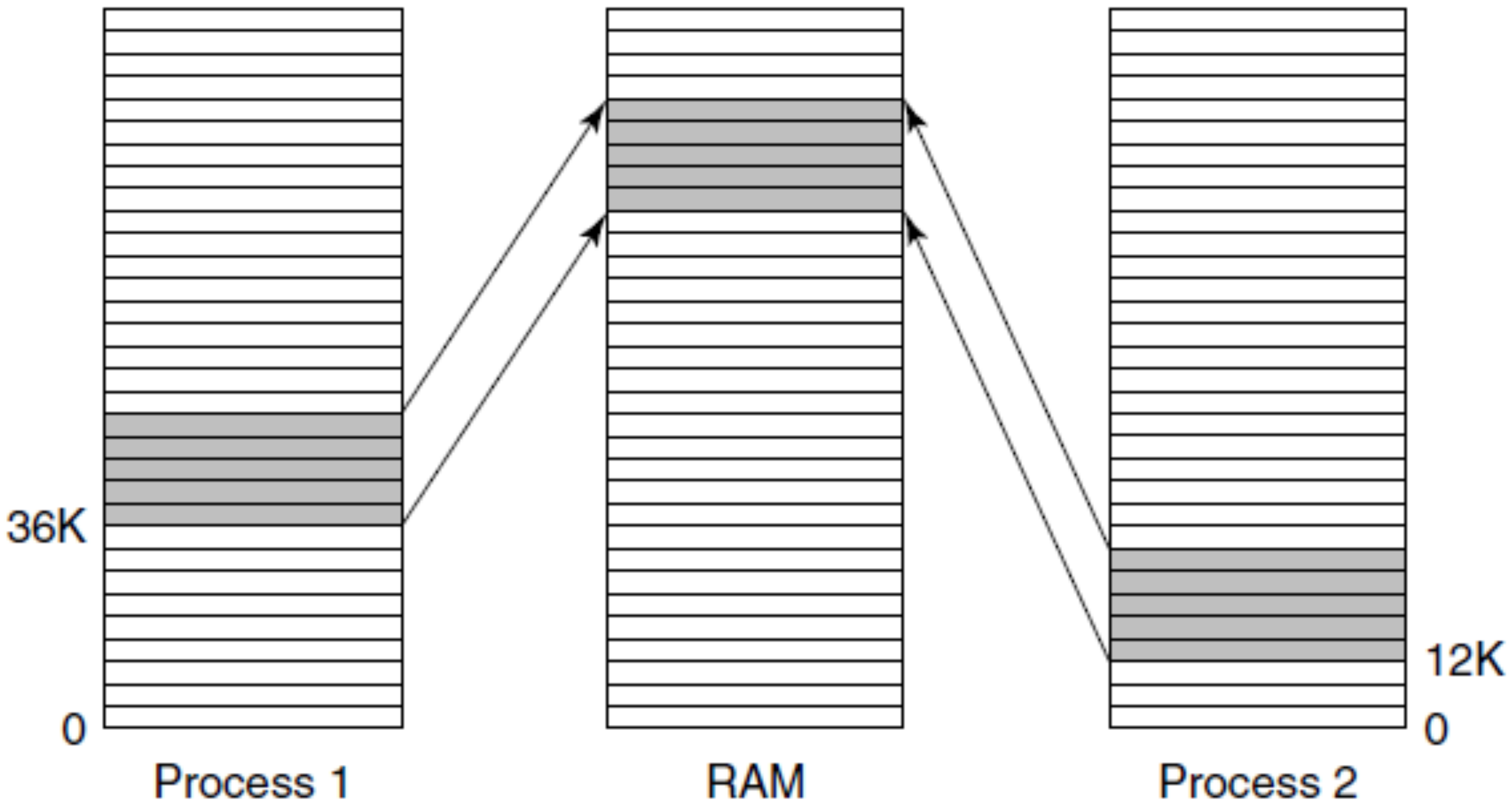


Figure 3-26. A shared library being used by two processes.

Shared Libraries (7)

- When the first function is called by process 2 (at address 12K), the jump instruction has to go to 12K+16, not 36K+16
- One way to solve the problem is to use copy on write and create new pages for each process sharing the library, but this scheme defeats the purpose of sharing the library
- A better solution is to compile shared libraries with a special compiler flag telling the compiler not to produce any instructions that use absolute addresses
- Code that uses only relative offsets is called **position-independent code**

Mapped Files (1)

- Shared libraries are a special case of a more general facility called **memory-mapped files**
- The idea is that a process can issue a system call to map a file onto a portion of its virtual address space
- As pages are touched, they are demand paged in one page at a time, using the disk file as the backing store
- When the process exits, or explicitly unmaps the file, all the modified pages are written back to the file on disk

Mapped Files (2)

- If two or more processes map onto the same file at the same time, they can communicate over shared memory
- Writes done by one process to the shared memory are immediately visible when the other one reads from the part of its virtual address spaced mapped onto the file
- If memory-mapped files are available, shared libraries can use this mechanism

Cleaning Policy (1)

- Paging works best when there is a large amount of free page frames available that can be claimed as page faults occur
- **Paging daemon** is a background process that sleeps most of the time, but is awakened periodically to inspect the state of memory
- If too few page frames are free, it begins selecting pages to evict using some page replacement algorithm

Cleaning Policy (2)

- The previous contents of the page are remembered
- In the event one of the evicted pages is needed again before its frame has been overwritten, it can be reclaimed by removing it from the pool of free page frames
- The paging daemon ensures that all the free frames are clean, so they need not be written to disk in a big hurry when they are required

Cleaning Policy (3)

- One way to implement this cleaning policy is with a two-handed clock:
  - The front hand is controlled by the paging daemon
  - When it points to a dirty page, that page is written back to disk and the front hand is advanced
  - When it points to a clean page, it is just advanced
  - The back hand is used for page replacement, as in the standard clock algorithm
  - Thus, the probability of the back hand hitting a clean page is increased due to the work of the paging daemon

Virtual Memory Interface (1)

- In some advanced systems, programmers have some control over the memory map and can use it in nontraditional ways to enhance program behavior
- If programmers can **name regions** of their memory, it may be possible for one process to give another process the name of a memory region so that process can also map it in

Virtual Memory Interface (2)

- Sharing of pages can also be used to implement a high-performance message-passing system:
  - Normally, when messages are passed, the data is copied from one address space to another
  - If processes can control their page map, a message can be passed by having the sending process unmap the page(s) containing the message
  - The receiving process will map them in
  - Only the page names have to be copied, instead of all the data

Virtual Memory Interface (3)

- Another advanced memory management technique is called **distributed shared memory:**
  - Multiple processes share a set of pages over a network, possibly, but not necessarily, as a single shared linear address space
  - When a process references a page that is not currently mapped in, it gets a page fault
  - The page fault handler locates the machine holding the page and sends it a message asking it to unmap the page and send it over the network
  - When the page arrives, it is mapped in and the faulting instruction is restarted

End

Week 08 – Lecture

References

- Tanenbaum & Bos, Modern Operating Systems: 4th edition, 2013 Prentice-Hall, Inc.