Memory Management

Week 07 – Lecture 1

Paging

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, 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)

- 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)

- 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 Algorithm (1)

- 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 Algorithm (2)

- 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
  - and gives an adequate performance

First-In, First-Out (FIFO) Page Replacement Algorithm

- 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

Second-Chance Algorithm (1)

- 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 (Fig. 3-15):
  - If it is 0, the page is both old and unused, so it is replaced immediately
  - If it is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory

Second-Chance Algorithm (2)

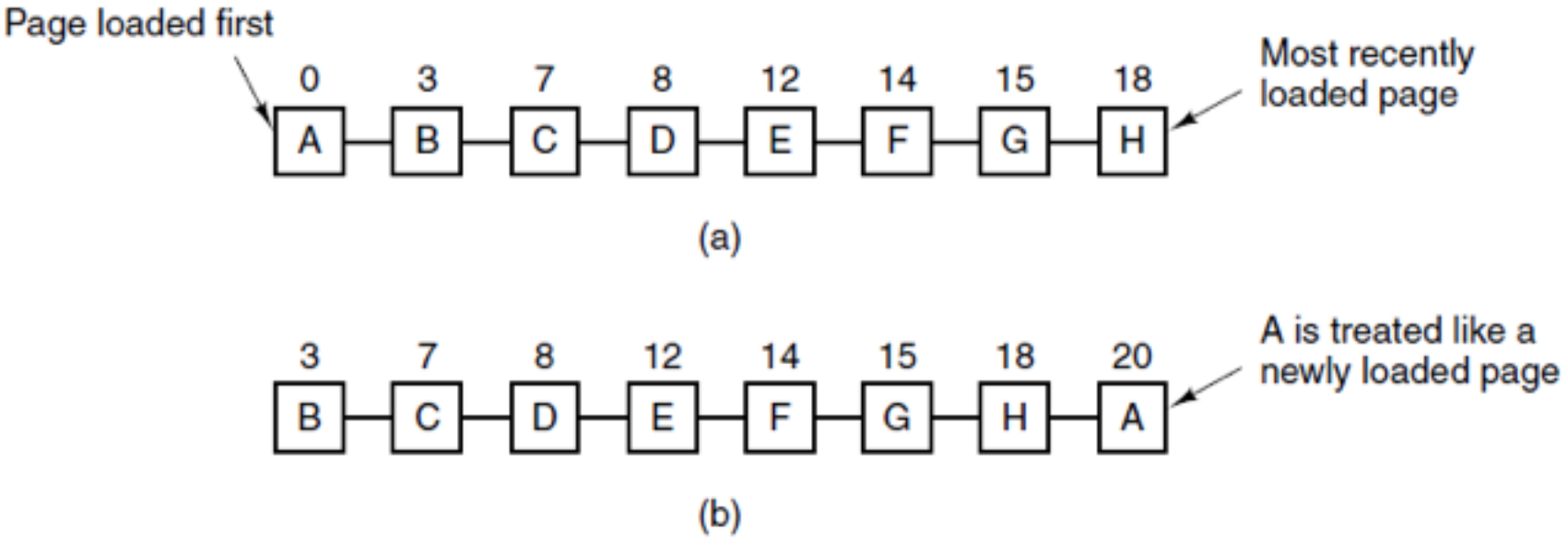
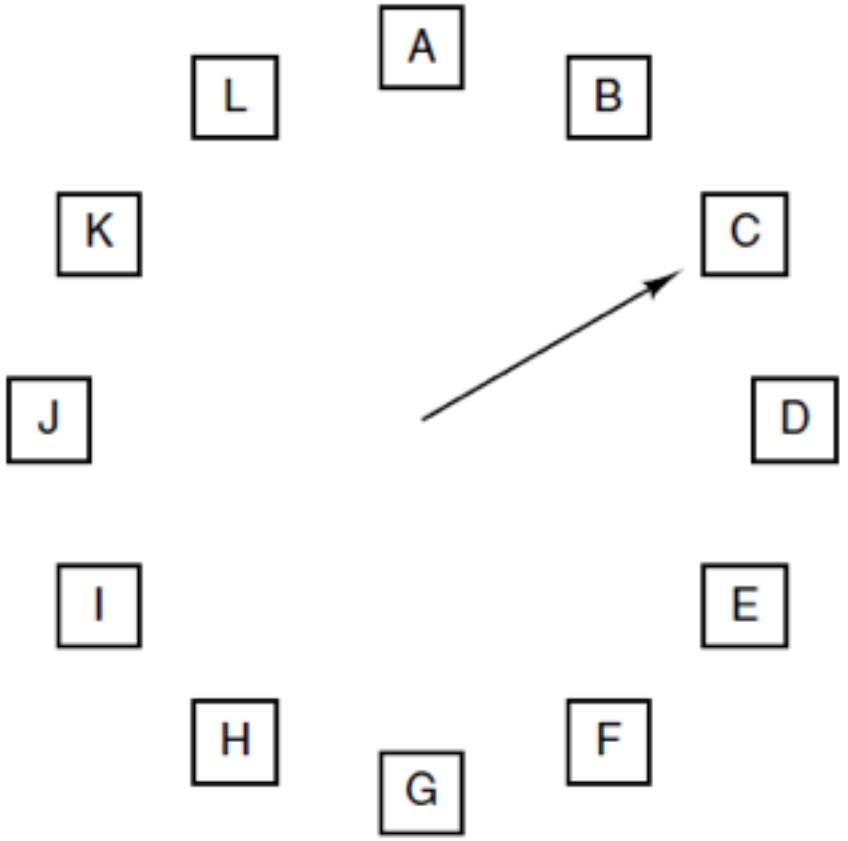


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its *R* bit set. The numbers above the pages are their load times.

Clock Page Replacement Algorithm (1)

- 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 Page Replacement Algorithm (2)



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

Figure 3-16. The clock page replacement algorithm.

Least Recently Used (LRU) Algorithm (1)

- The observation: pages that have been heavily used in the last few instructions will probably be heavily used again soon; pages that have not been used for a long time will not probably be used soon
- A realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time

Least Recently Used (LRU) Algorithm (2)

- It is 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: we need to find a page in the list, to delete it and then to move it to the front
  - This is 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
- However, it is possible to simulate it in software. Two examples of such algorithms are:
  - NFU (Not Frequently Used)
  - Aging (modification of NFU)

Not Frequently Used Algorithm (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 Algorithm (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
  - In this case 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)

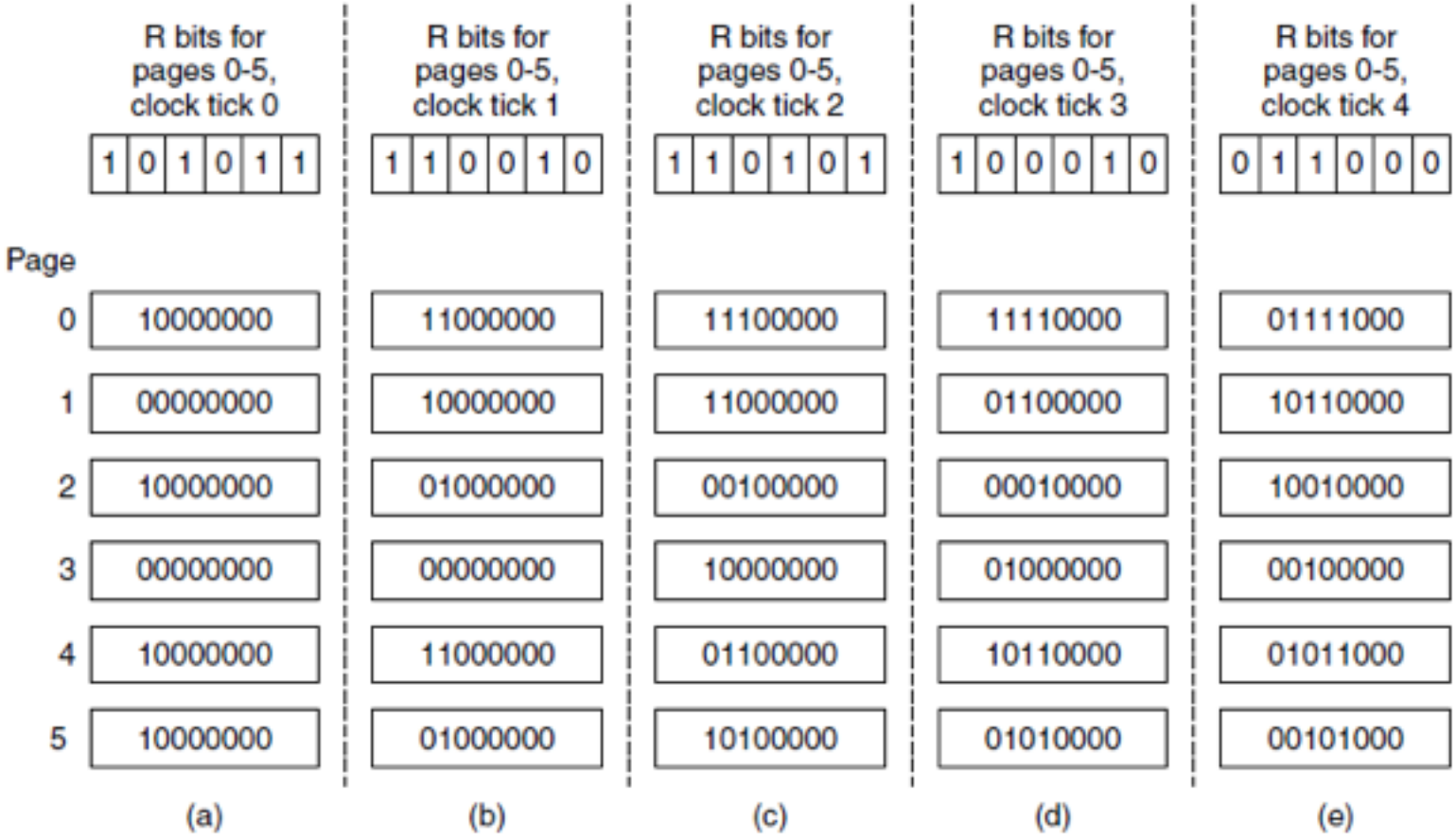


Figure 3-17. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

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
- Is 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)

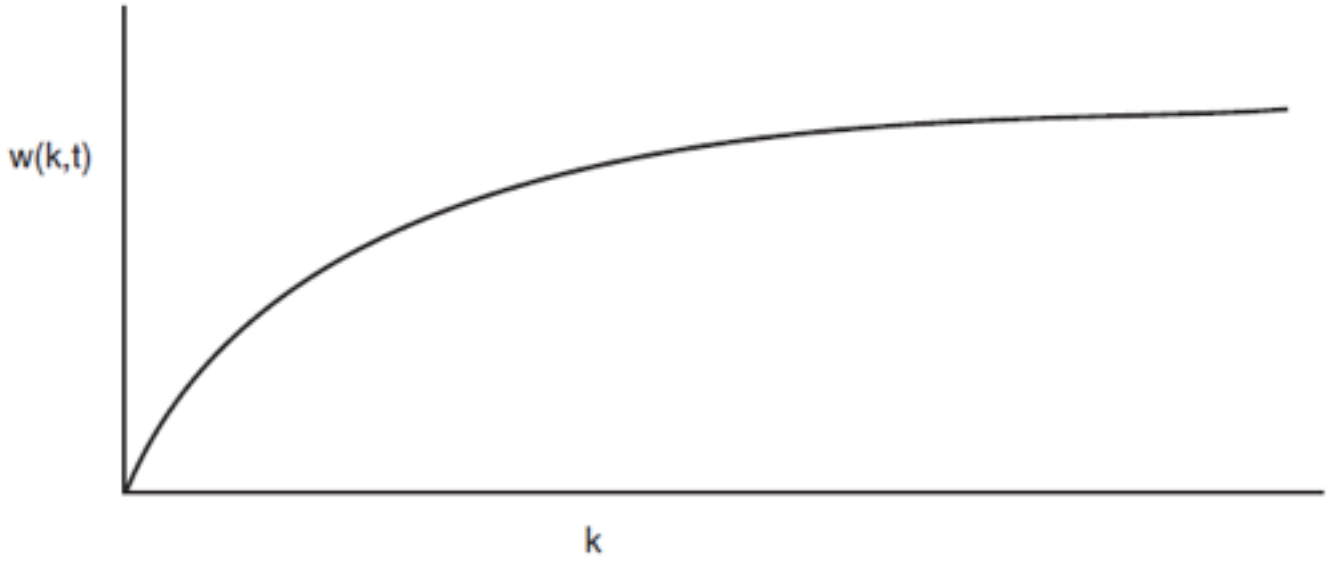


Figure 3-18. The working set is the set of pages used by the *k* most recent memory references. The function *w(k, t)* is the size of the working set at time *t*.

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 he 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)

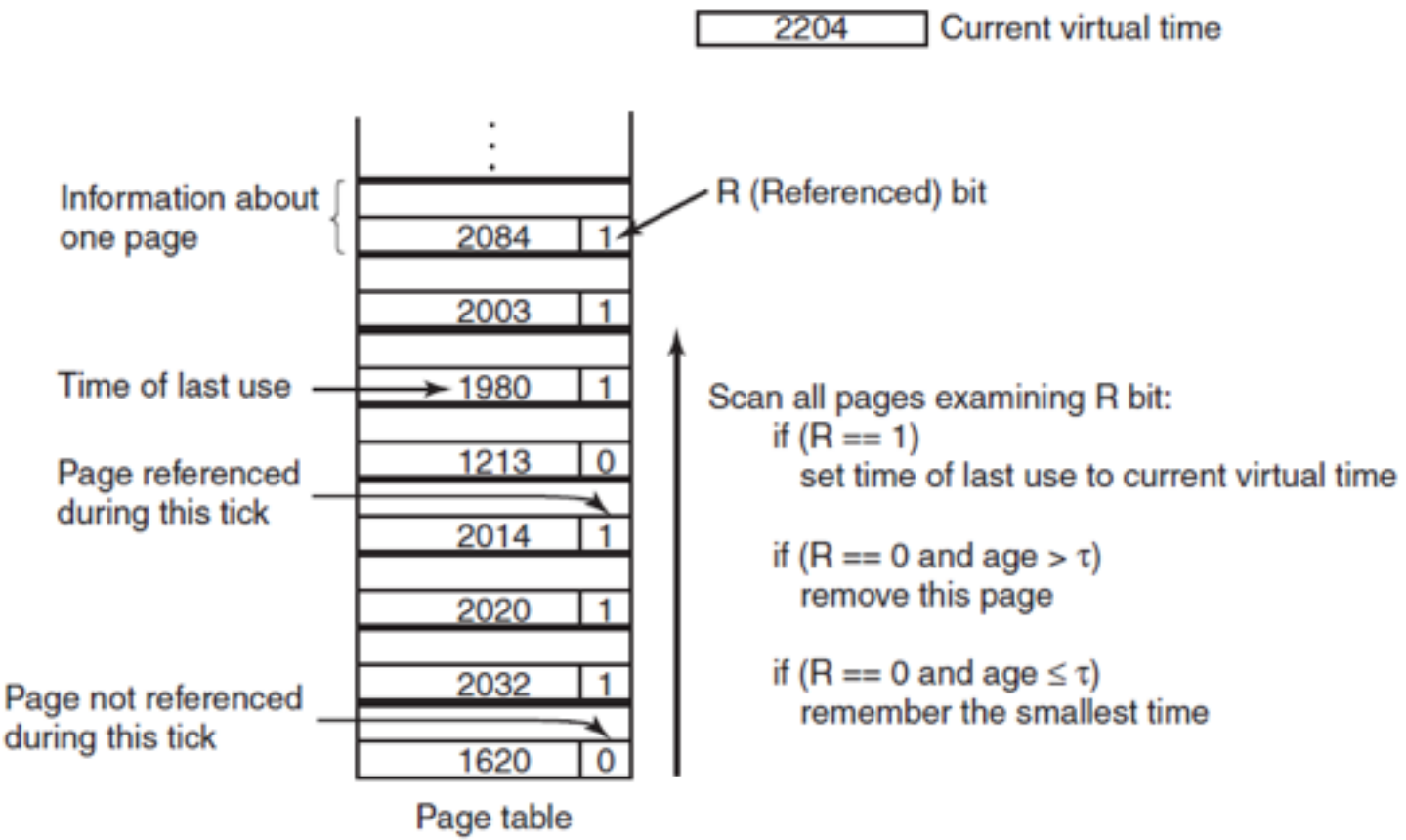


Figure 3-19. The working set algorithm.

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)

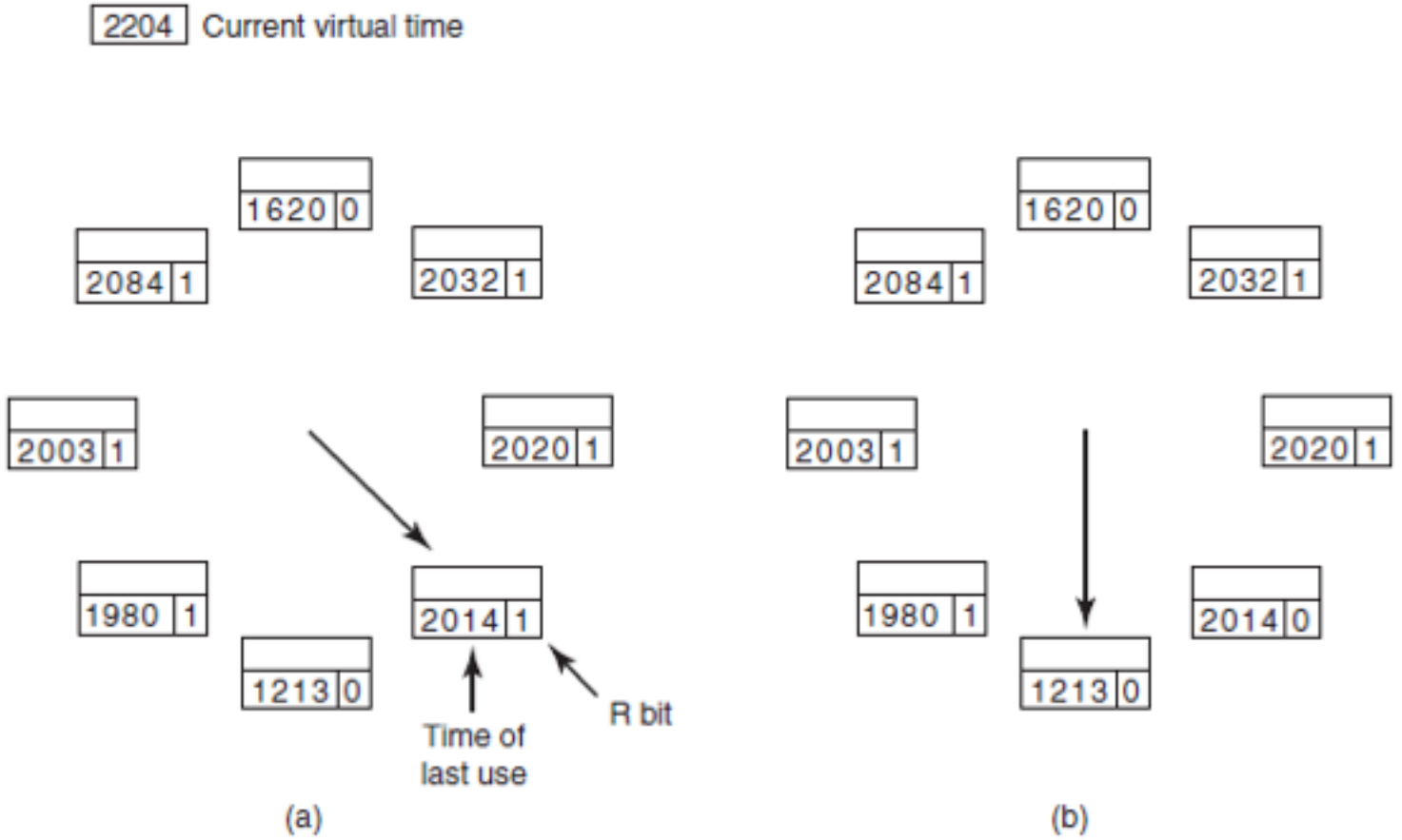


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when *R* = 1.

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)

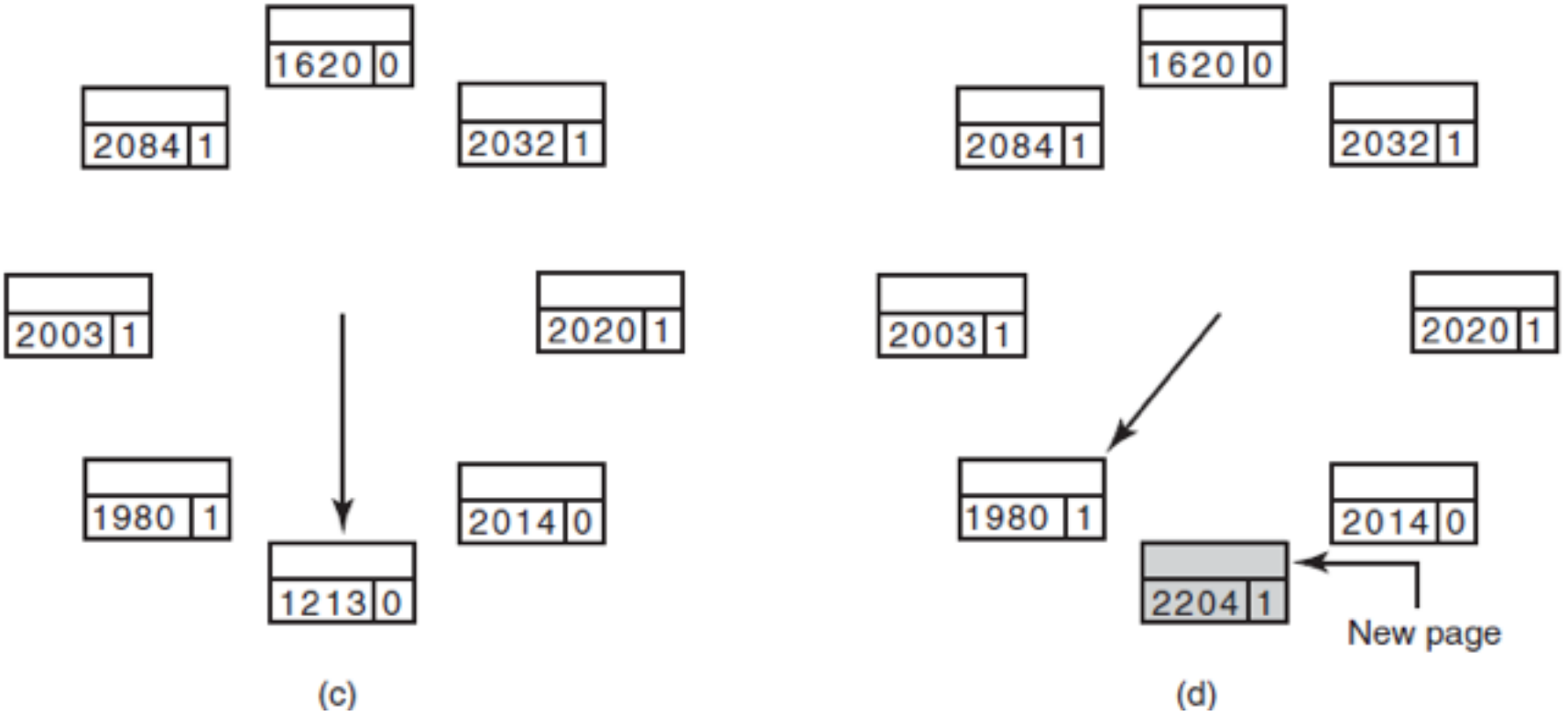


Figure 3-20. Operation of the WSClock algorithm. (c) and (d) give an example of $R = 0$.

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 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 of Page Replacement Algorithms

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. Page replacement algorithms discussed in the text.

End

Week 07 – Lecture 1

References

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