Lecture 8 (Page replacement algorithms)

Page Replacement Algorithms:

When page fault occurs, OS should choose page to evict. If page was been modified need to to be rewritten to the disk, otherwise rewrite is unnecessary.

- Optimal algorithm
- Not recently used algorithm
- FIFO algorithm
- Second-chance algorithm
- Clock algorithm
- Least recently used (LRU) algorithm
- Working set algorithm
- WSClock algorithm

Optimal algorithm

Impossible to implement, easy to describe.

page fault occurs some set of pages is in memory → one of pages referenced to very next instruction, other may NOT be referenced until n instruction later → each page can be labeled with number of instruction which will executed before that page is first referenced → page with highest n is removed.

Problem: OS cannot determine next instruction to execute, it's possible only on the second run of algorithm.

Not recently used

Class 0: R = 0, M = 0

Class 1: R = 0, M = 1

Class 2: R = 1, M = 0

Class 3: R = 1, M = 1

Classes updates on every memory reference.

At page fault, NRU inspects pages and removes a page at random from **lowest-numbered nonempty class**

Advantages: easy to understand, efficient to implement, adequate performance

---

First-In First-Out

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: old page might be still useful

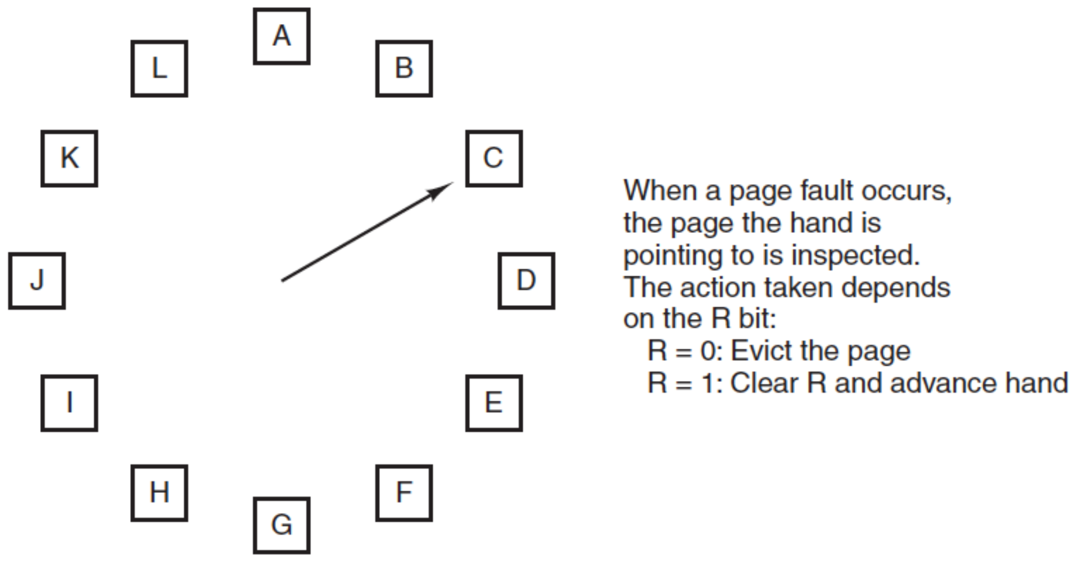
Second-chance algorithm

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

---

Clock Algorithm

- 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.



Least Recently Used (LRU)

When a page fault occurs, throw out the page that has been unused for the longest time.

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 end. The list must be updated on every memory reference

Hardware supply 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

Simulates LRU:

- NFU (not frequently used)
- Aging (modification of NFU)

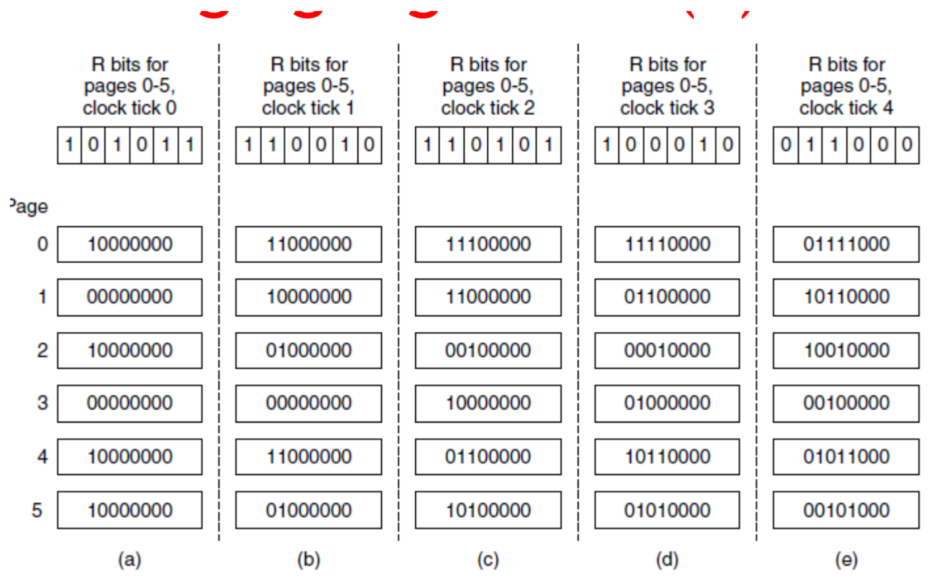
Not frequently used (NFU)

- 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

Aging algorithm

A small modification to NFU:

- 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



Page	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10010000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	(a)	(b)	(c)	(d)	(e)

Working Set Algorithm

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

Locality of reference - during any phase of execution, the process references only a relatively small fraction of its pages

Working set of process - the set of pages that a process currently used

Thrashing - program's behavior when page fault causing every few instruction (constant page faulkt occurs)

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.**

Prepaging - loading pages before letting the process run.

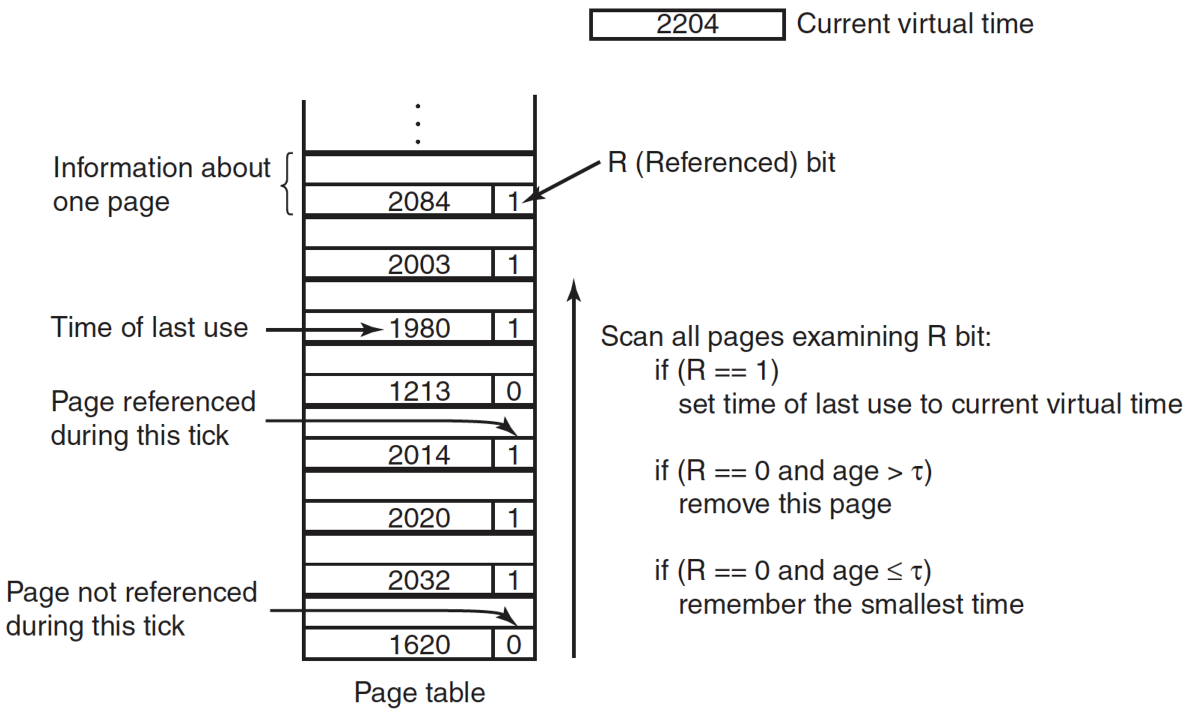
A possible page **replacement algorithm**: when a page fault occurs, find a page not in the working set and evict it.

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**)

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:
    1. If R ==1 ⇒ Time of last use = current virtual time

2. if R == 0 ⇒ If age = (current virtual time - time of last use) ≤ T ⇒ page in working set
3. if all pages in working set, the one with R == 0 and greatest age are **evicted**
4. in the worst case, when all R == 1, page choose at random



2204
Current virtual time
Information about one page
R (Referenced) bit
2084
1
2003
1
Time of last use
1980
1
1213
0
Page referenced during this tick
2014
1
2020
1
2032
1
Page not referenced during this tick
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

WSClock algorithm

The data structure is a circular, initially empty, list of page frames, as in the clock algorithm. 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.

At each page fault the page pointed to by the hand is examined first:

- if R == 1 ⇒ not ideal candidate to remove
- if R == 0 ⇒ hand advanced to the next page and algorithm repeats
- If R == 0 & the age > T & M==0 ⇒ the page frame is simply claimed and new page put there

- If M==1, 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

Two cases when hand return to 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 has 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)

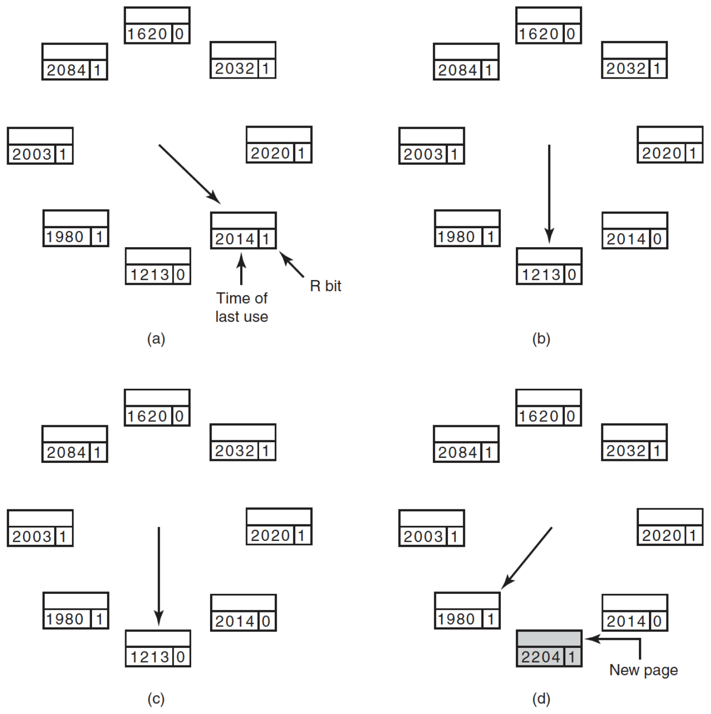
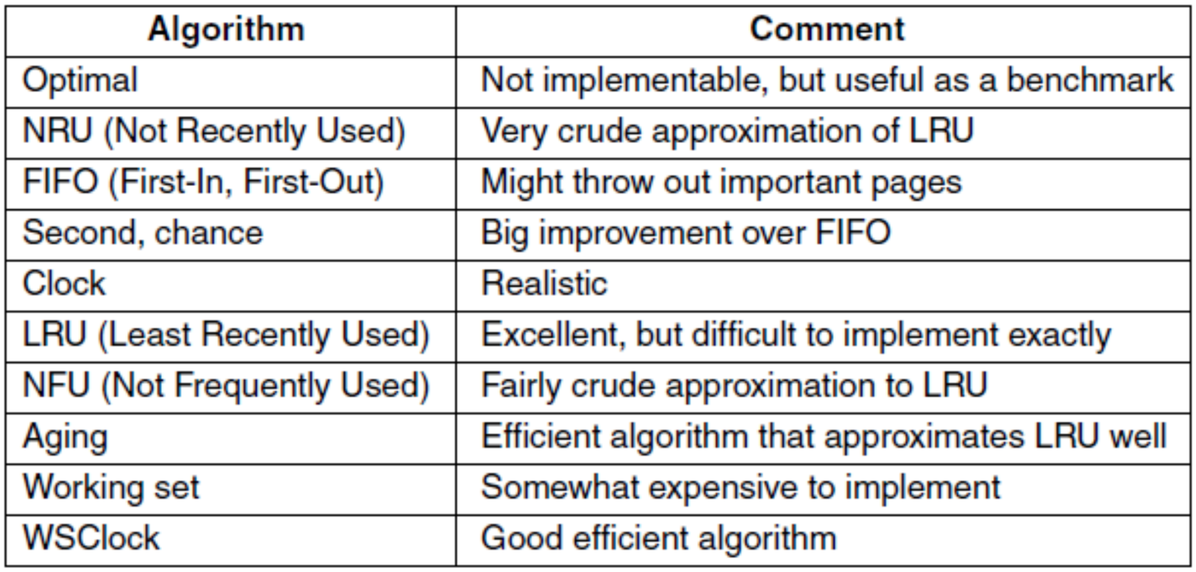


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when $R = 1$. (c) and (d) give an example of $R = 0$.



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

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 versus Global Allocation Policies

Local - The algorithm can try to find the least recently used page considering only the pages currently allocated to A. If a local algorithm is used and the working set grows, thrashing will result. If the working set shrinks, local algorithms waste memory

Global - The algorithm can try to find the least recently used page considering all the pages in memory. If a global algorithm is used, the system must continually decide how many page frames to assign to each process

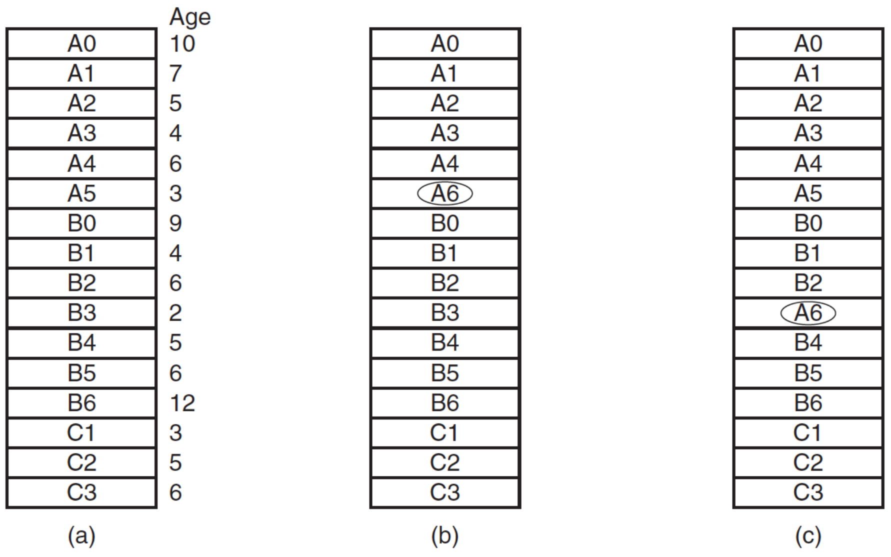

Age
A0 10
A1 7
A2 5
A3 4
A4 6
A5 3
B0 9
B1 4
B2 6
B3 2
B4 5
B5 6
B6 12
C1 3
C2 5
C3 6
(a)
A0 A1 A2 A3 A4 A6 B0 B1 B2 B3 B4 B5 B6 C1 C2 C3
(b)
A0 A1 A2 A3 A4 A5 B0 B1 B2 A6 B4 B5 B6 C1 C2 C3
(c)

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

Approaches to fix: aging bits & algorithm(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)

PFF (**Page Fault Frequency**) algorithm that tells when to increase or decrease a process' page allocation

Page fault rate - number of faults per second

FIFO - local(oldest in process) & global(oldest in memory); LRU - local(lru page in current process) & global(lru page in all memory). Working set or WSClock - local only

---

Load Control

System might thrash. PFF indicates that processes need more memory but no processes need less memory. Some of the processes may be swapped to the disk to free up all the pages they are holding. These pages are divided among thrashing processes.

Page size

Page size determines by OS. Determining the best page size requires balancing several competing factors and there is no overall optimum.

Internal fragmentation: situation when text, data, or stack segment will not fill an integral number of pages.

Small pages: pros(reduce integral fragmentation, easy to allocate small programs with phases) & cons(large page table, time for transfer small page == for large page, much valuable space on TLB)

Optimal page size: process size = s, page size = p, page entry size = e,

Number of pages per process = s/p, space per process = es/p

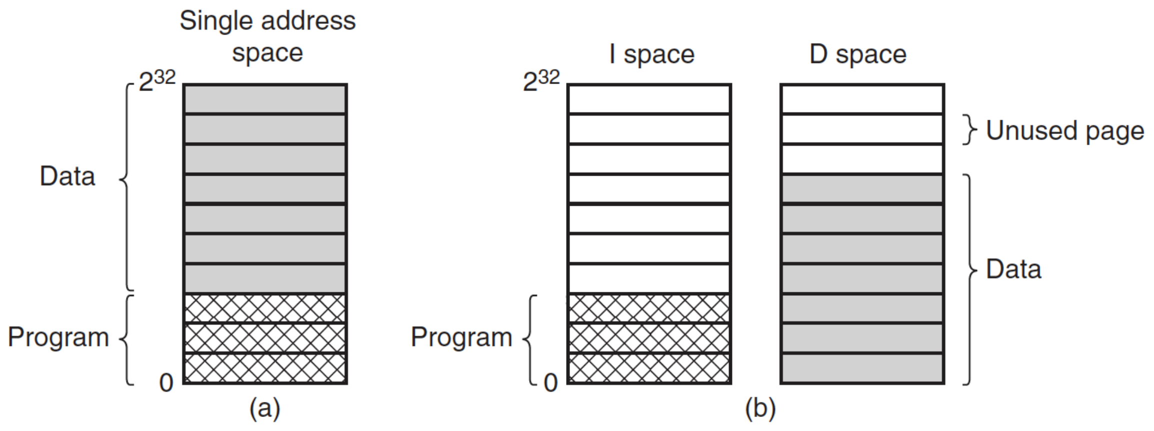
Wasted memory = p/2 - due to internal fragmentation

Overhead = se/p+p/2

Optimal page size p = sqrt(2se)

Separate Instruction and Data Spaces

If address space is to small when we can have separate address spaces for instructions (program text) and data, called I-space and D-space. Each one has its own page table, with its own mapping of virtual pages to physical page frames. Having I-space & D-space doubles available address space.



Shared pages

It is not efficient to have two copies of the same page in memory at the same time. Not all the pages are shareable.

Problem: scheduler can remove shareable page, which can cause page faults.

If separate I-space & D-space ⇒ shareable I-space & diff D-space.

Pages that are read-only, such as program text, can be shared, but for data pages sharing is more complicated.

Parent & 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.

Copy on write - pages which never modified don't need to be copy, only the data pages that modified needs to be copied.

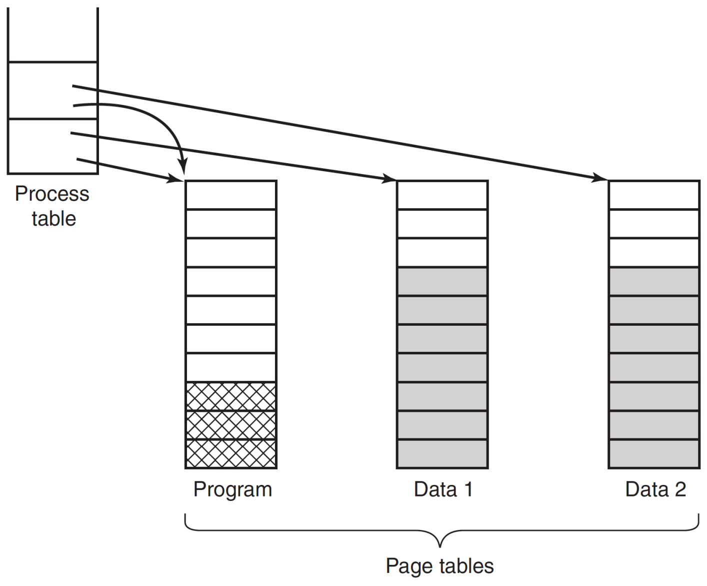


Figure 3-25. Two processes sharing the same program sharing its page tables.

Shared libraries

Undefined externals - functions called in the object files but not present there

Common programs use 20–50 MB worth of graphics and user interface functions. 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 loaded: program is loaded or function in them firstly called.

DLL - dynamic linked library - shared library in Windows

If two processes shares the same library, can occur problem of address(picture). Solutions: use copy on write and create new pages for each process sharing the library, relocating them on the fly as they are created (kill concept of library) & compile shared libraries with a special compiler flag telling the compiler not to produce any instructions that use absolute addresses. Instead only instructions using relative addresses are used.

Code that uses only relative offsets is called **position-independent code.**

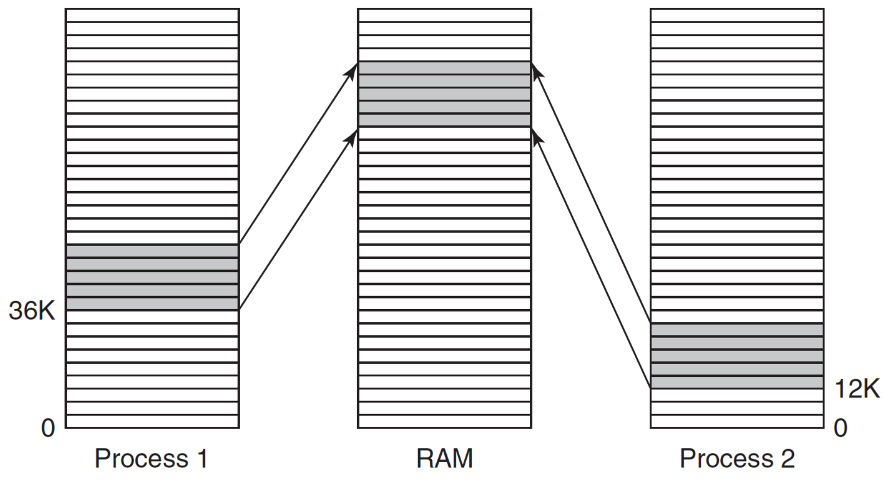


Figure 3-26. A shared library being used by two processes.

Mapped files

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. One page at the time. Process exit or explicitly unmaps the file ⇒ modified pages written back to disk. Mapped files provide an alternative model for I/O. Instead, of doing reads and writes, the file can be accessed as a big character array in memory. If two or more processes map onto the same file at the same time, they can communicate over shared memory.

Cleaning Policy

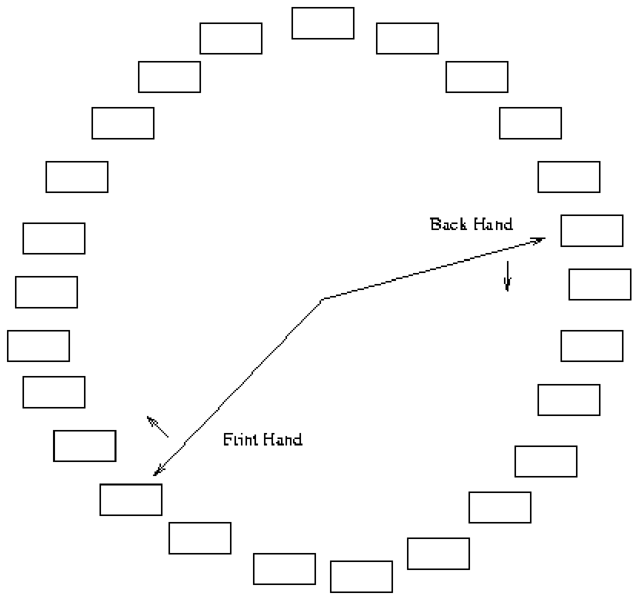
Paging works best on large amount of free pages.

Paging daemon - 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 starts selecting pages to evict using replacement algorithms. Paging daemons ensure that all free frames are clean.

One way to implement this cleaning policy is with a **two-handed clock:** front hand - paging daemon, back hand - for page replacement as in standard clock algorithm;

- if daemon tells that page is dirty(M==1)⇒page written on disk & front hand advanced
- if daemon tells that page is clean(M==0)⇒front hand advanced;

Probability of the back hand hitting a clean page is increased due to the work of the paging daemon.



Each Rectangle is a page frame. As the Front Hand advancees, it sets the referenced bit in the page frame to zero. As the back hand advances, if it comes to a page where the referenced bit is zero, it marks that page to be overwritten. When a page is referenced, its referenced bit is set to one.

Virtual Memory Interface

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.

Sharing of pages can be used to implement a high-performance **message-passing system**: if process controls their page map → message passed by having sending process unmaps pages containing message → receiving message will map them in → only page names copies.

Another advanced memory management technique is called **distributed shared memory:** n processes shares set of pages over network → process reference not mapped page → page fault → page fault handler send message to machine to unmaps the page & send it over network → page arrives → page mapped in → faulting instruction restarted.