Memory Management

Chapter 3

Seyda Özer

Memory

- Paraphrase of Parkinson's Law, "*Programs expand to fill the memory available to hold them*."
- Average home computer nowadays has 10,000 times more memory than the IBM 7094, the largest computer in the world in the early 1960s

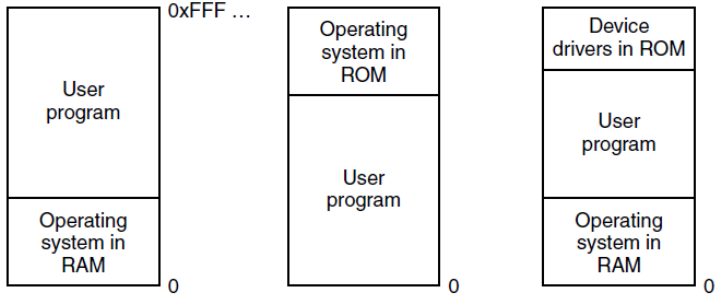
150,000
(32K words)

7090 worth $20 million in 2019

Memory Management

- Main memory is a resource
- Operating systems have to manage it
- Could have been simple: put the OS at the bottom, and the application gets the rest.
- Not so...

No Memory Abstraction



old main frames (60s)

some embedded systems

early MSDOS system (80s)

a and c: user programs can wipe out the OS mistakenly

Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist

Running Multiple Programs Without a Memory Abstraction

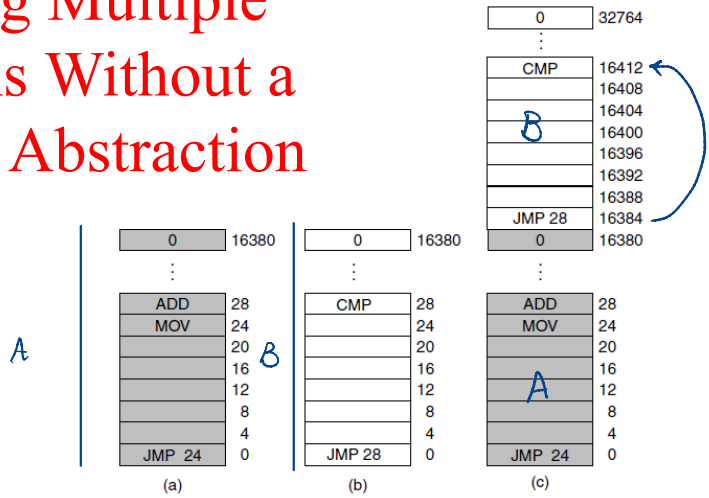


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

Static Relocation

- At program load time, the execution address is known
- Add the actual offset to the presumed location specified by the compiler and linker
- The cost of this — that is, the number of words that need to be relocated — varies depending on the hardware architecture and compiler.
- No protection

A Memory Abstraction: Address Spaces

- Multiprogramming and memory protection are not easy without abstraction
- An address space is the set of addresses that a process can use to address memory.
- Each process has its own address space, independent of those belonging to other process.
- Just as the process concept creates a kind of abstract CPU to run programs, the address space creates a kind of abstract memory for programs to live in.
- Two problems have to be solved: protection and relocation.

Base and Limit Registers

- Simple version of dynamic relocation
- Map each process' address space onto a different part of physical memory.
- The contents of the base register is added to all absolute memory references.
- Any memory reference larger than the limit register is invalid and causes a trap.

Disadvantage:

- every memory ref is an addition and comparison. Addition expensive, Why?
- Large programs that do not fit the memory!
- Solution: swapping and virtual memory

Base and Limit Registers

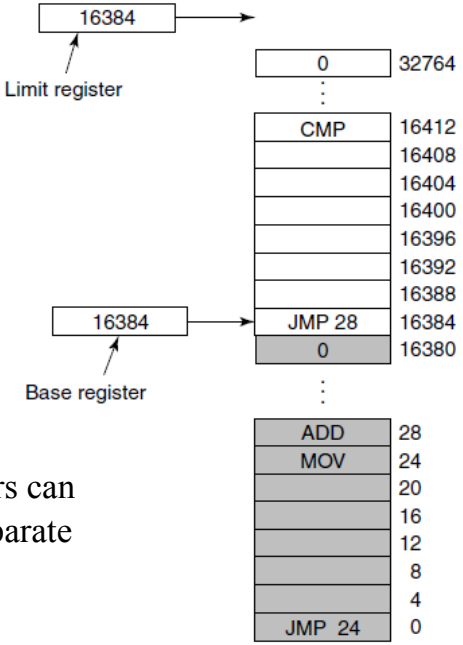


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

Swapping

- Physical memory is not large enough to hold all the processes
- Swapping: Write the entire program out to disk when it's blocked (another solution is virtual memory- partial programs)
- Read it all back in again, possibly at a different address, when it's ready to run.
- Can use this to compact memory (expensive)
- Apparently frees up the memory, but disk I/O consumes memory bandwidth.

Swapping (1)

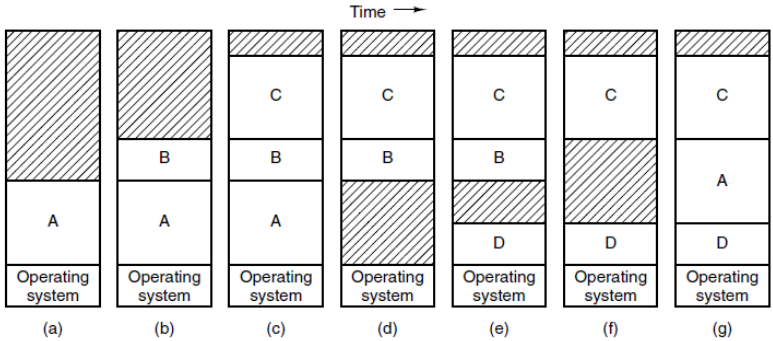


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory

how much memory should be allocated for a process?

- It is probably a good idea to allocate a little extra memory whenever a process is swapped in or moved,
- However, when swapping processes to disk, only the memory actually in use should be swapped;

Swapping (2)

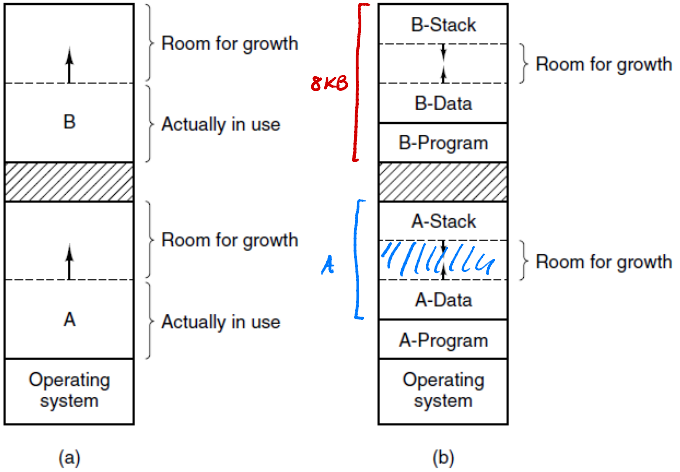


Figure 3-5. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

Memory Management

- When memory is assigned dynamically, the operating system must manage it.
- In general terms, there are two ways to keep track of memory usage:
  - bitmaps and
  - free lists.

4 KB ⇒ 4.2^{10} = 2^{12} 2^{20} 2^{8} = 256

Memory Management with Bitmaps

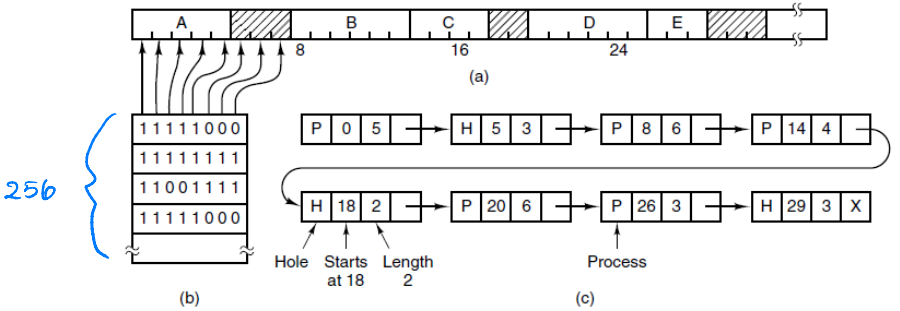


Figure 3-6. (a) A part of memory with five processes and three holes. The tickmarks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

Memory Management with Bitmaps

- Bitmaps: memory is divided into allocation units.
- Size of the allocation unit is a design issue: What is the tradeoff?
- Searching for K allocation unit is expensive.

Memory Management with Linked Lists

Figure 3-7. Four neighbor combinations for the terminating process, *X*.

Memory Management Algorithms

- First fit
- Next fit
- Best fit
- Worst fit
- Quick fit

First Fit

- Traverse the list until a large-enough block is found
- Allocate as much of it as is needed
- Return the rest to the free list
- When freeing memory, must merge adjacent blocks
- Obvious answer: order linked list by address
- Next fit: next search starts from the result of prev search

Best Fit

- First fit seems wasteful – there may be an exact match further down the list
- Solution: best fit
- Find the free block that's closest in size to the request
- Obvious data structure: order linked list by size
- Makes merging less efficient
- And best fit isn't more efficient! More wasted memory: why?

Worst Fit

- Best fit tends to leave small, useless blocks (fragmentation)
- What about worst fit — pick the largest block and sub-divide it?
- That's not very good, it turns out: first fit and best fit are better and are pretty close.
- Note: this is a simulation result based on trace data.
- For first fit, as much as 1/3 of memory may be unusable!
- Many other algorithms...

Better Implementation

- Keep the hole list and process list seperate
- Hole list can directly use the free memory (

Overlays

- There is a need to run programs that are too large to fit in memory.
- Solution adopted in the 1960s, programmers split programs into little pieces, called overlays

- Kept on the disk, swapped in and out of memory by the program itself

Virtual Memory

- Virtual memory: each program has its own address space, broken up into chunks called pages
- Convert a virtual address — the address as seen by the program — to a physical address
- Mapping is done by pages — group of (perhaps) 4K bytes
- The programs need not to be on contiguous areas of physical memory
- Eliminates all problems of allocation inefficiency, compaction, relocation, memory protection, and more

Paging (1)

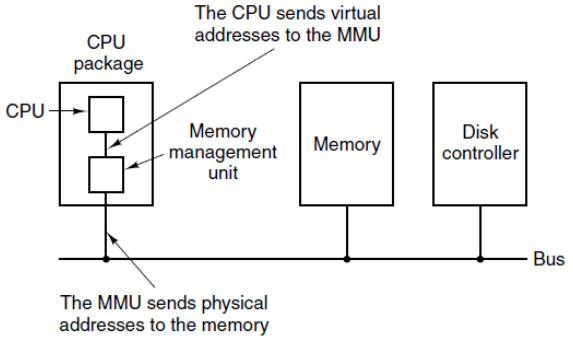


Figure 3-8. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was years ago.

Paging (2)

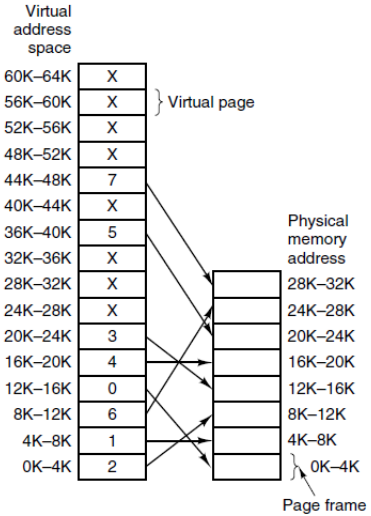


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the page table. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287

Virtual Memory – Paging

- Each program has its own address space, which is broken up into chunks called pages.
- The program-generated addresses are called virtual addresses and form the virtual address space.
- The virtual address space consists of fixed-size units called pages. The corresponding units in the physical memory are called page frames.
- Page sizes are powers of 2: we will see why

Paging (3)

- Present/absent bit keeps track of which pages are physically present in memory.

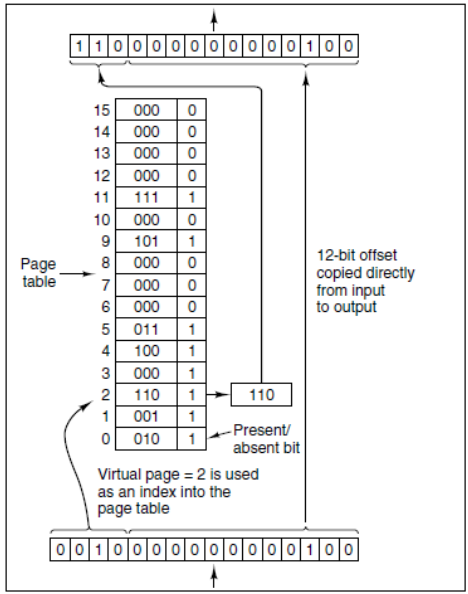


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

Implementing Virtual Memory

- Divide a virtual address A into $\langle V, O \rangle$, where V is the virtual page number and O is the offset within the page.
- The memory management unit (MMU) maps V into P, the physical page number and produces $\langle P, O \rangle$
- In its simplest form, this mapping is done by direct indexing into an MMU register bank M:

$$A \rightarrow \langle M[V], O \rangle$$

- This scheme can be faster than a base register; no additions are involved.

Structure of a Page Table Entry

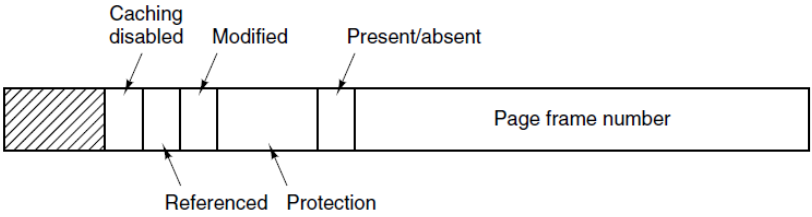
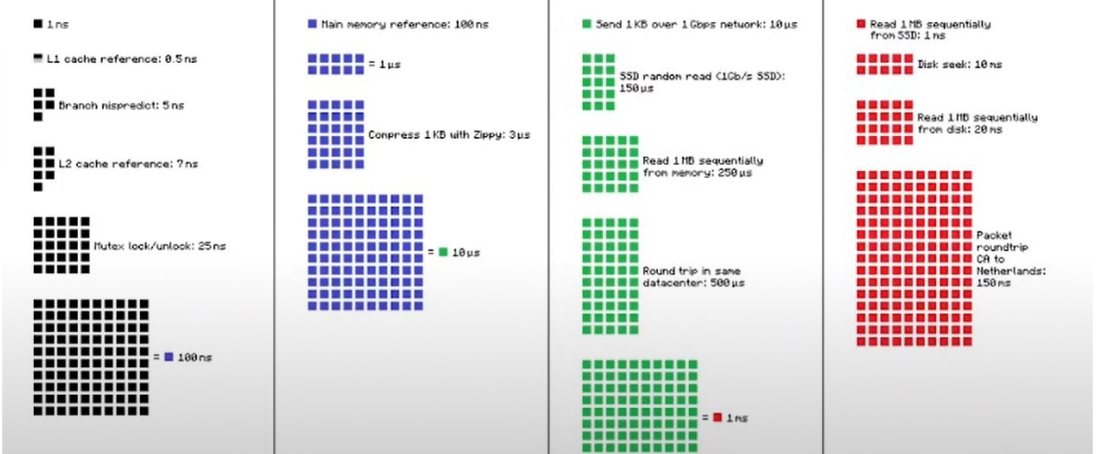


Figure 3-11. A typical page table entry.

Too Simple

- Earlier times this could work – page sizes were large relative to memory
- The PDP-11s on which Unix grew up had only 8 pages of address space — that many registers were affordable
- Modern computers easily have 1G of memory and pages of 4K bytes – or 256K possible pages
- We can't have that many MMU registers
- We need a separate page table for each process — we can't afford to copy that many MMU registers during process switches

Latency Numbers Every Programmer Should Know



Page Tables in RAM

- First-order solution: put the page table in RAM
- Still problematic:
  - It's slow — an extra RAM access for each memory reference
  - It's a lot of RAM for page tables
- Page tables do live in RAM, but there are optimizations

Speeding Up Paging

Major issues faced:

1.The mapping from virtual address to physical address must be fast.

2.If the virtual address space is large, the page table will be large.

Translation Lookaside Buffer

- Mappings from virtual address spaces are cached in the *Translation Lookaside Buffer* (TLB)
- The TLB is an associative memory — it maps a value, rather than an index, into another value
- When there's a TLB miss, an old entry is discarded, a memory lookup is done, and the new entry is inserted into the TLB
- This works because programs tend to exhibit locality of reference
- On many systems, the TLB is managed in hardware; on RISC systems, it is explicitly populated by the OS

Translation Lookaside Buffers

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

Figure 3-12. A TLB to speed up paging.

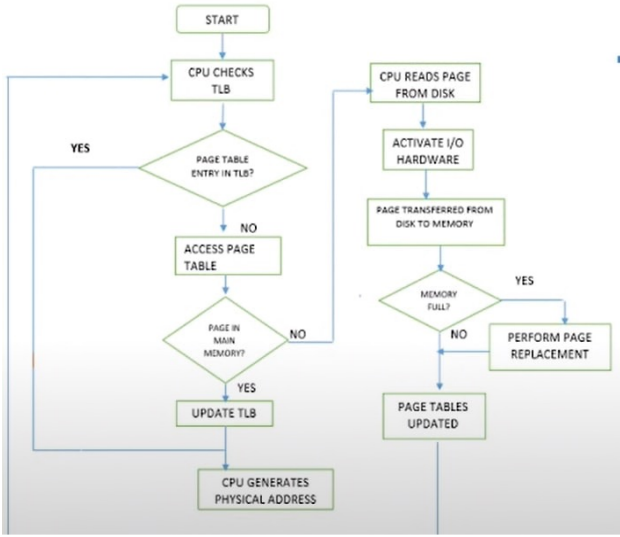
Software TLB Management

- Many RISC machines do nearly all of this page management in software.
- a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance

Software TLB Management

- A **soft miss** occurs when the page referenced is not in the TLB, but is in memory. A few nanoseconds.
- A **hard miss** occurs when the page itself is not in memory
- **A minor page fault**: the page may actually be in memory, but not in this process' page table, the page may have been brought in from disk by another process.
- **A major page fault**: if the page needs to be brought in from disk. Milliseconds
- **Segmentation fault**: the program simply accessed an invalid address

Software TLB Management



Typical TLB
These are typical performance levels of a TLB:[17]

size: 12 bits – 4,096 entries
hit time: 0.5 – 1 clock cycle
miss penalty: 10 – 100 clock cycles
miss rate: 0.01 – 1% (20–40% for sparse/graph applications)

If a TLB hit takes 1 clock cycle, a miss takes 30 clock cycles, and the miss rate is 1%, the effective memory cycle rate is an average of $(1+30) \times 0.99 + (1 + 30 + 30) \times 0.01 = 31.30$ (31.30 clock cycles per memory access).

Multilevel Page Tables

- Another problem is how to deal with very large virtual address spaces.
- The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time.
- In Fig. 3-13(a) we have a 32-bit virtual address that is partitioned into a 10-bit PT1 field, a 10-bit PT2 field, and a 12-bit Offset field.
- Since offsets are 12 bits, pages are 4 KB, and there are a total of 220 of them.

Multilevel Page Tables

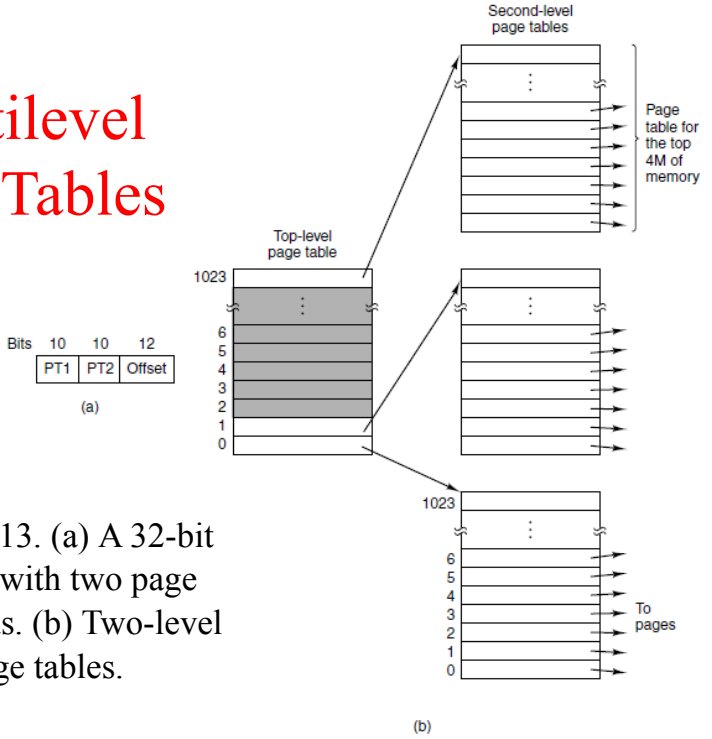


Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

Multilevel Page Tables

- Intel's 32 bit 80386 processor (in 1985) was able to address up to 4-GB of memory, using a two-level page table (4-KB page frames). $2^{10} \times 2^{10} \times 2^{12} = 2^{32}$
- Ten years later, the Pentium Pro, page map level 4, 512 entries in all tables
- $2^9 \times 2^9 \times 2^9 \times 2^9 \times 2^{12} = 2^{48}$ memory

Inverted Page Tables

- In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space.
- Inverted page tables save lots of space but virtual-to-physical translation becomes much harder.
- When process n references virtual page p, the hardware can no longer find the physical page by using p as an index into the page table.
- Instead, it must search the entire inverted page table for an entry (n, p).
- Solution: Use TLB and hash tables

Inverted Page Tables

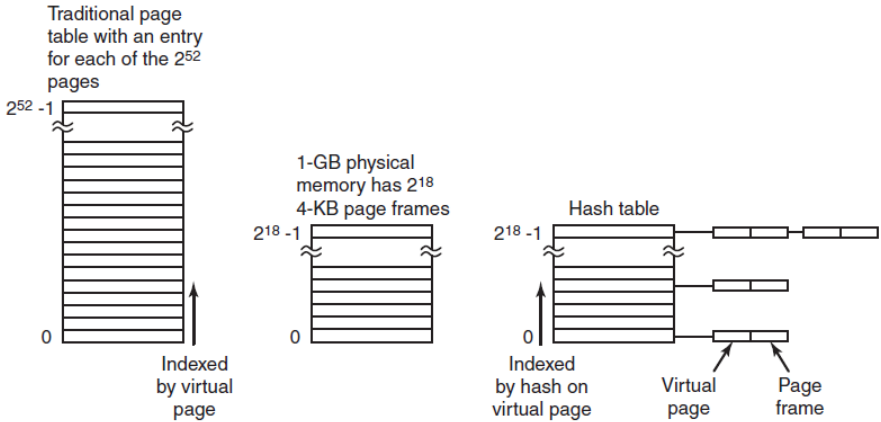


Figure 3-14. Comparison of a traditional page table with an inverted page table.

Page Replacement Algorithms

- Similar method with CPU cache, web server cache
- 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

Constraints

- Must be efficient — many paging decisions take place
- Must approximate the correct answer !
- Must be implementable on real hardware
- Usually, must work well on multitasking systems

Tools

- The OS has a few tools available to it
- The referenced bit — this page has been used recently
- The modified bit — discarding this page will be more expensive
- Clock interrupts
- Page fault interrupts
- Advice from the application

Optimal Algorithm

- Simple but impossible to implement
- The optimal page replacement algorithm says that the page that will not be referenced longest should be replaced.
- We can run algorithms on simulators, we would know the longest unreferences pages for the second run.
- In this way, it is possible to compare the performance of realizable algorithms with the best possible one.

Not Recently Used Algorithm

- At page fault, system inspects pages
- 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.

Not Recently Used (NRU)

- At process start time, reset all R and M bits
- On clock interrupts, clear R bits
- Classify pages by M and R:

	R	M
Class 0:	0	0
Class 1:	0	1
Class 2:	1	0
Class 3:	1	1

- On page fault, discard a random page from the lowest class

Resetting R and M

- Why do we reset R on clock interrupts?
  - –We want to know if a page has been used recently
- Why not reset M?
  - –M cannot be reset until the page has been written out to disk; an old copy will not be sufficient

Properties of NRU

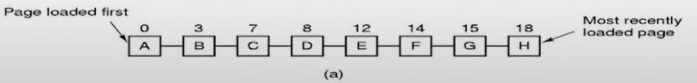
- Bias towards discarding unmodified pages
- But — better to discard a modified page that has not been used recently than one that is in use
- Simple algorithm; may give adequate performance on some systems
- Primarily useful for teaching

What's Interesting about NRU?

- It has the essential properties of any pagereplacement algorithm
- It looks for a (relatively) idle page
- It handles modified pages, but is biased against using them
- It is reasonably efficient

First In, First Out (FIFO)

- Forget about R and M
- When a page frame is needed, discard the oldest page
- Of course, the oldest page may still be busy, so it will come right back in
- FIFO is rarely used in this form



Second Chance FIFO

- Similar to pure FIFO, but the R bit is checked
- If R is set on an old page, clear the R bit but move it to the just-loaded end of the queue
- Approximates somehow LRU behavior
  - –Note the problem: it doesn't know if a page has been used recently or not
  - –It only approximates that if page faults are frequent

Second-Chance Algorithm

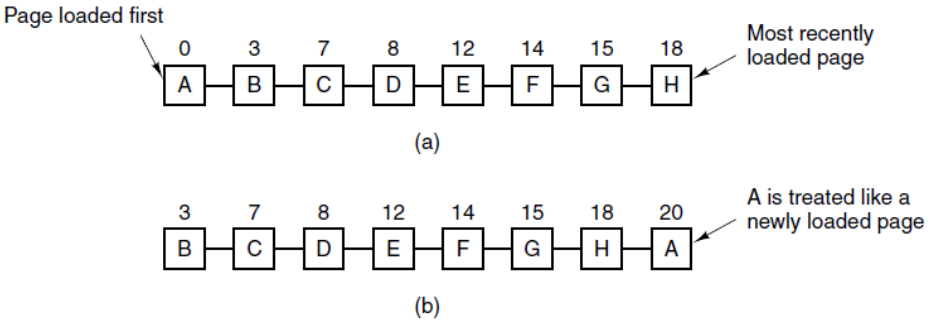


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.

Second Chance FIFO

- Loaded pages are ordered in a FIFO list: "oldest" page first in list
- When oldest page selected for replacement, the reference bit is inspected
  - If value is 0: replace page
  - If value is 1: page gets a "second chance":
    - Clear reference bit, set to 0
    - Move page to tail of FIFO list
    - Check next-oldest page
- A page given a second chance will move to the tail of the FIFO queue and becomes the youngest page
- Worst case:
  - All reference bits == 1, degenerates into pure FIFO

The Clock Algorithm

•FIFO can be implemented with a circular linked list; the list head is changed, rather than moving the page table entry → the clock algorithm

•The set of frames is considered as laid out like a circular buffer

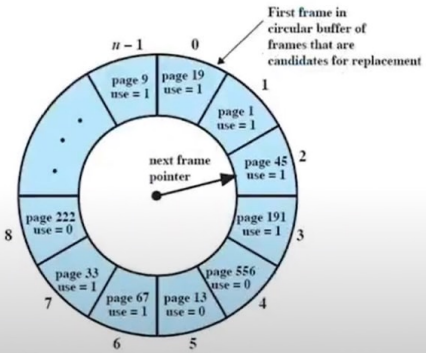
–contains a FIFO list of pages

–can be circulated in a round-robin fashion

•Each page has a reference bit

–when a page is first referenced (a page fault that leads to its load), the use bit of the frame is set to 1

•Each subsequent reference to this page again sets this bit to 1



Clock Policy

•Finding a page to replace

–The "next frame pointer" advances from frame to frame in this circular "frame buffer"

–As long as the "next frame pointer" encounters a page in a the frame with the reference bit set to 1, it is set it to 0 and moves on – it gives the page a second chance

–The first page encountered with reference bit == 0 is replaced

–Page is replaced, pointer is advanced to next page

Clock Page Replacement Algorithm

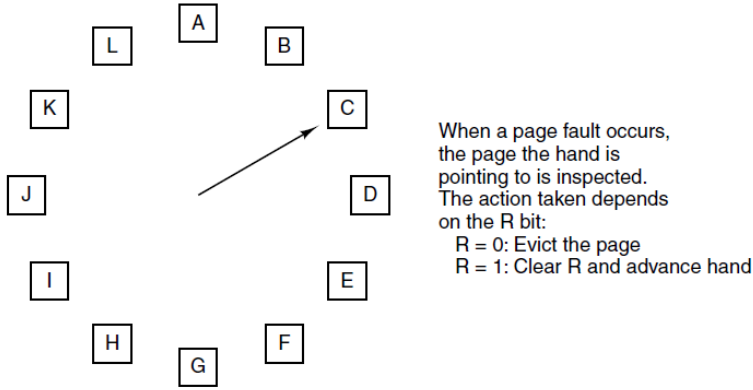


Figure 3-16. The clock page replacement algorithm.

Least Recently Used (LRU)

- Assumption: a page that hasn't been used recently is unlikely to be used soon
- But — how can this be implemented?
- More precisely, what data structure do we use to track this?

Hardware-Assisted LRU

- Have a 64-bit instruction counter
- On each memory reference, store the counter in a per-page frame field
- On a page fault, scan the page table for the lowest value
- Note: this is in the number of page frames

Software-Simulated LRU

- Have an array of counters, one per page
- At each clock tick, add the value of R to the counter
- Implements NFU — Not Frequently Used
- Problem: never forgets

NFU With Aging

- Shift each counter right (i.e., divide by 2) before each addition
- Add R bit to the high-order bit
- Recent references have more weight
- Pages referenced this clock tick and previous clock ticks are more important than those referenced only this time
- Note: every clock tick the R bit is updated

Simulating LRU in Software

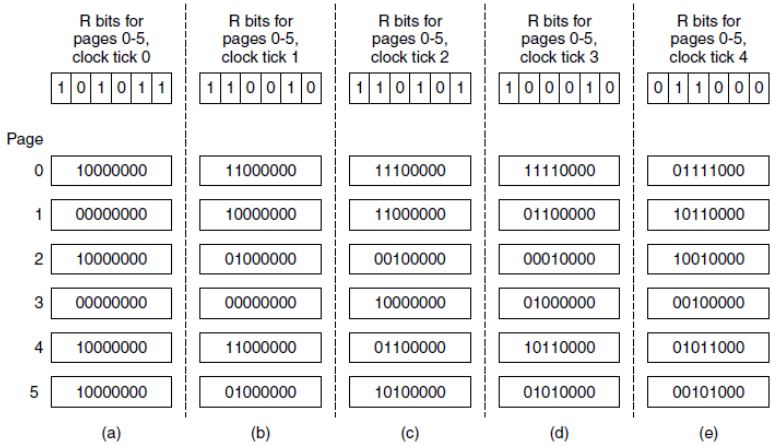


Figure 3-17. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

Frame Allocation to Processes

- Demand-paging will allocate frames as necessary
- When the free-frame list is exhausted, a page replacement algorithm will choose a page to be replaced

Thrashing is a bad thing!

•Thrashing is a situation where a process is repeatedly page faulting

–The process does not have enough frames to support the set of pages needed for fully executing an instruction

–It may have to replace a page that itself or another process may need again immediately

•A process is thrashing if it is spending more time on paging than execution

The Working Set

- At any time, a program is only using a small fraction of its pages (locality of reference)
- The set of pages in use at the moment is the working set
- At time t, the working set w(k,t) is the most recently referenced pages for the last k references
- Note that w is monotonically increasing as a function of k and it asymptotically approaches the total program size
- A program needs to have its working set in memory
- If there isn't enough memory to hold the entire working set, the program will thrash

Working Set Algorithm (1)

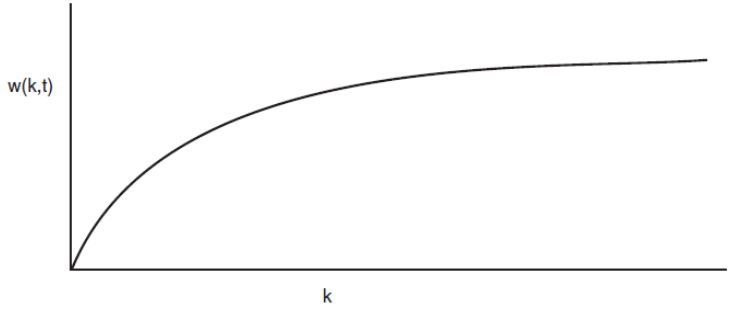


Figure 3-18. The working set is the set of pages used by the *k* most recent memory references. The function *w(k, t)* is the size of the working set at time *t*.

Using the Working Set

- Before running a process, make sure that the working set is in memory
- Because w is asymptotic, the exact choice of k isn't critical, as long as it's large enough
- The trick is to determine the working set

Approximating the Working Set

- Ideally, we would track the last memory references
- Instead, we track pages referenced during the last seconds
- For each page, keep a virtual clock field
- At each tick, update the clock field if R is set
- Pages not referenced during the last seconds may be discarded

Working Set Algorithm (2)

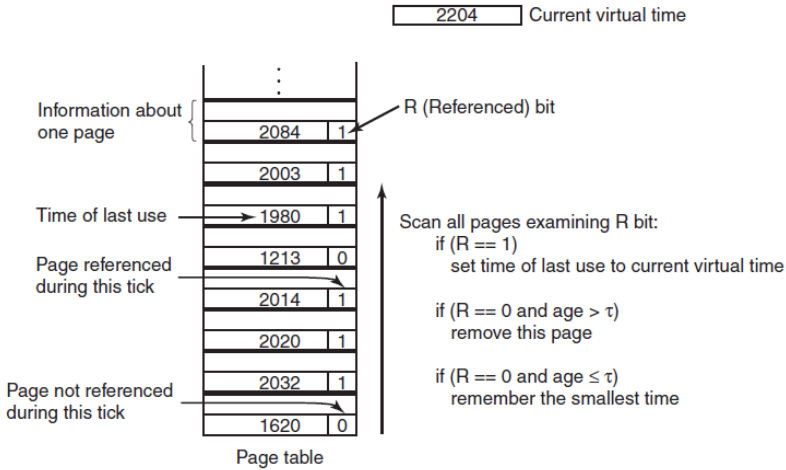


Figure 3-19. The working set algorithm.

WSClock

- Use the working set concept; use the clock algorithm's data structure
- Have a circular linked list of page frames
- At each page fault, check the page R and Time pointed to by the clock hand; if is R=1, the page is current and can't be discarded; advance the clock hand
- If R is 0, check the age T. If old enough , the page can be reused

WSClock Algorithm (1)

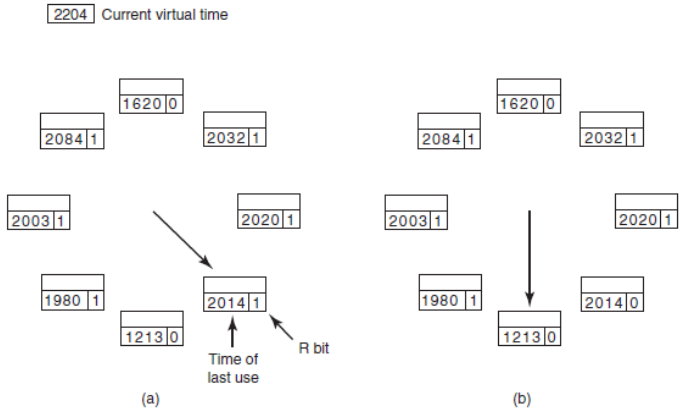


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when R = 1.

WSClock Algorithm (2)

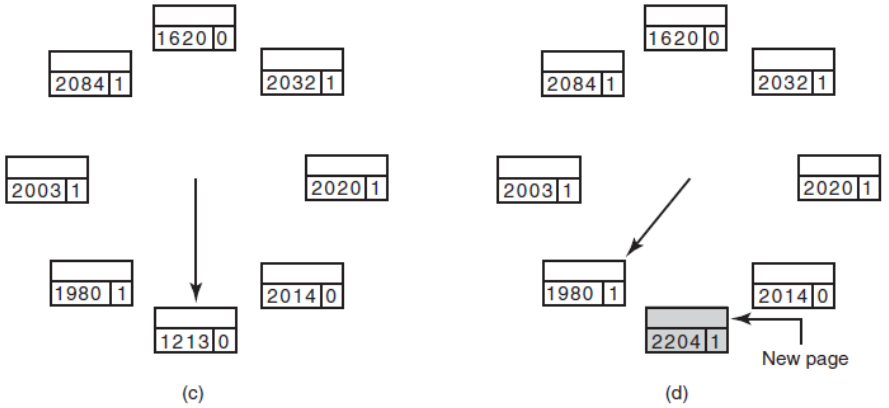


Figure 3-20. Operation of the WSClock algorithm. (c) and (d) give an example of R = 0.

Dealing with Modified Pages

- If W is 0, the page frame exists on disk and can be reused immediately
- If W = 1, schedule it to be written to disk and keep scanning — you might find a clean page

What Happens if the Clock Hand Circles?

- If some writes have been scheduled, keep looking; a write will complete eventually
- If no writes have been scheduled, all pages are in the working set; pick a random clean page to reuse

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

- NRU is never actually used
- Plain FIFO works poorly; second chance FIFO works reasonably well

Figure 3-21. Page replacement algorithms discussed in the text.

- NFU with aging is a good choice
- WSClock is efficient and gives good results

Design Issues

Local versus Global Allocation

- When process A has a page fault, where does the new page frame come from?
- More precisely, is one of A's pages reclaimed, or can a page frame be taken from another process?
- If another process, do we bias the selection in any fashion?
- If page replacement affects only the current process, we have local policy; if we look at all processes, we have a global allocation policy

Local versus Global Allocation Policies (1)

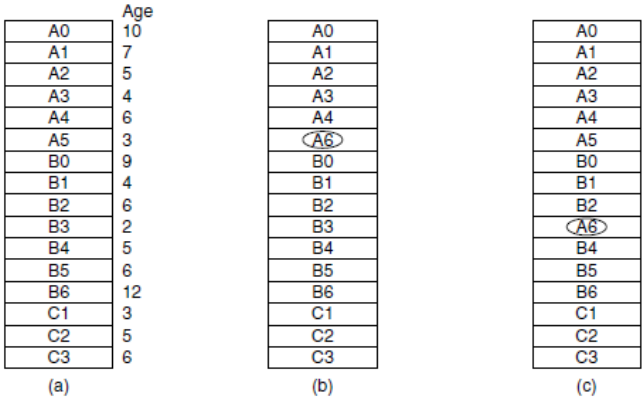


Figure 3-22. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

Choosing

- Global policies tend to work better
- If you use a local policy and the working set grows, you can get thrashing
- Similarly, if the working set shrinks, you waste memory
- With a global policy, though, you need to decide how much memory to allocate to each process

Memory Requirements Change

- Processes grow and shrink
- Working sets grow and shrink
- Allocations must be changed over time
- Monitor the page fault frequency (PFF) for each process
- A process with a high PFF gets a larger allocation; a process with a small PFF gets a smaller allocation

Local versus Global Allocation Policies (2)

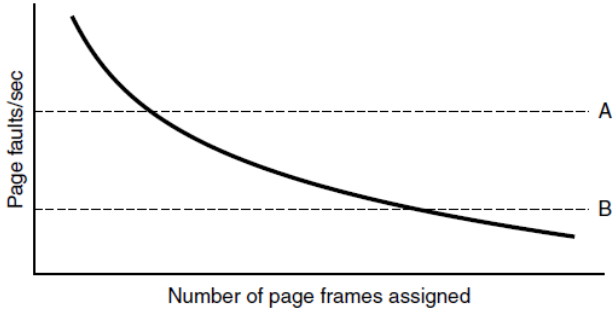
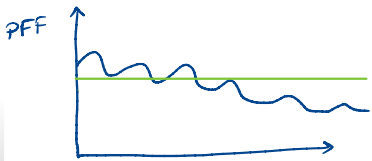


Figure 3-23. Page fault rate as a function of the number of page frames assigned.

Measuring PFF

- Count the number of page faults per second
- Accumulate this as a moving average, of the type we've seen several times before
- For many algorithms, including LRU, PFF goes down as memory allocation increases



$$P_t = \alpha \cdot P_{t-1} + (1-\alpha) P_{t-2}$$

Algorithms versus Allocation

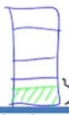
- Algorithms such as LRU and FIFO work with either local or global allocation policies
- Working set and WSclock are local-only
- There's no such thing as a working set for the entire system
- Must rely on allocation policy for global effects

Swapping

- If memory is not enough, trashing is unavoidable
- Write the entire program out to disk when it's blocked
- Read it all back in again, possibly at a different address, when it's ready to run
- Can use this to compact memory
- Apparently frees up the CPU, but disk I/O consumes memory bandwidth

Controlling Swapping

- Which processes should get swapped out?
- Do we look at priority? Size? History?
- Once processes are swapped out, when do they come back in?
- Need a two-level scheduler, one for ordinary CPU access and one for swapping out and in
- For this second scheduler, what are we optimizing for? CPU utilization? Throughput?



Page Size

- With large pages, we waste memory: on average, half of the last page isn't used
- With small pages, we use a lot of memory for page tables
- Call the average process size s and the page size p. Assume that each page table entry (and associated data structures) takes e bytes
- The overhead $o = \frac{s}{p}e + \frac{p}{2}$
- To optimize for memory use, differentiate and set to 0:

$$\frac{d_o}{d_p} = -\frac{se}{p^2} + \frac{1}{2} = 0$$

- Best size: $p = \sqrt{2se}$

Separate I and D spaces

- Most computers have a single address space that holds both programs and data
- If this address space is large enough, everything works fine. However, if it's too small, things become difficult
- Separate I and D spaces
- Rather than for the normal address spaces, they are now used to divide the L1 cache.

Separate Instruction and Data Spaces

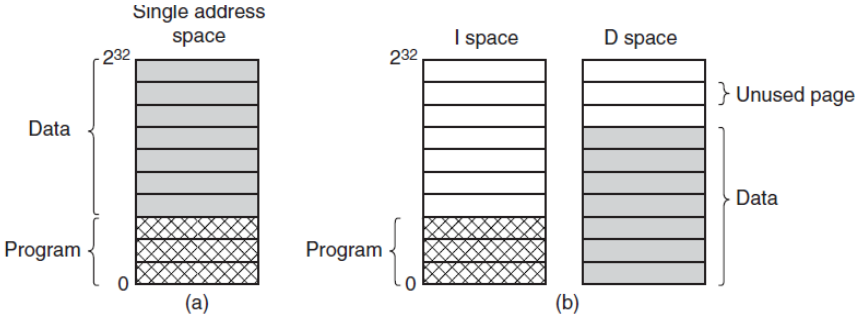


Figure 3-24. (a) One address space.
(b) Separate I and D spaces.

Shared pages

- Think about what happens after a fork…
- Pages that are read-only, such as program text, can be shared, but for data pages sharing is more complicated.
- If separate I- and D-spaces are supported, it is relatively straightforward to share programs

Shared Pages

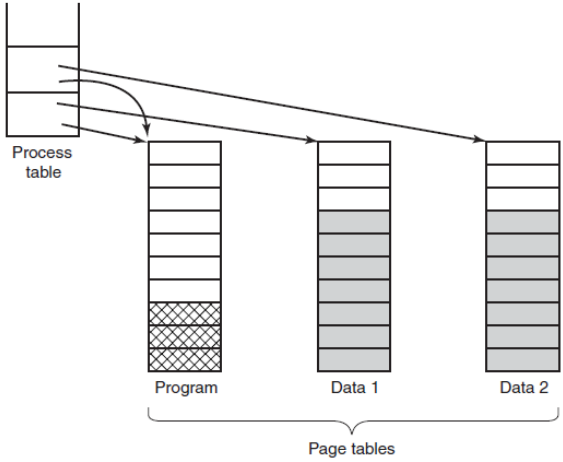


Figure 3-25. Two processes sharing the same program sharing its page table.

Shared pages

- Context switch overhead can be reduced by page-sharing
- Sharing data is trickier than sharing code, but it is not impossible.
- After a fork: Only the data pages that are actually modified need to be copied.
- This approach, called **copy on write**, improves performance by reducing copying

Shared Libraries

- Instead of statically binding libraries, shared libraries (DLL) can be used.
- Shared libraries are paged!
- Shared library routines are located at different memory addresses for each process (see figure)
- Solution: position independent code!

Shared Libraries

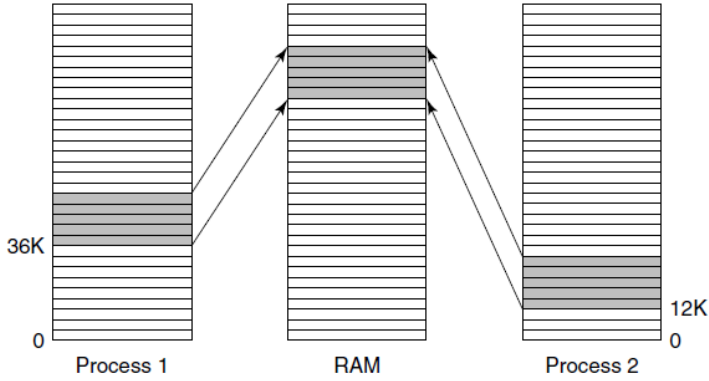


Figure 3-26. A shared library being used by two processes.

Mapped Files

•A process can issue a system call to map a file onto a portion of its virtual address space.

•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

•Shared libraries are really a special case of a more general memory-mapped files.

•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 when there is an abundant supply of free page frames

•paging systems generally have a background process, called the **paging daemon**, that sleeps most of the time

• If too few page frames are free, it begins selecting pages to evict using some page replacement algorithm.

Operating System Involvement with Paging

- There are four times when the operating system has paging-related work to do:
  - –process creation time: page table and space allocation
  - –process execution time: MMU reset, TLB flushed
  - –page fault time: find the page number, find available page
  - –process termination time: release the pages, table entries

Page Fault Handling (1)

1. The hardware traps to kernel, saving program counter on stack.
2. Assembly code routine started to save general registers and other volatile info
3. system discovers page fault has occurred, tries to discover which virtual page needed
4. Once virtual address caused fault is known, system checks to see if address valid and the protection consistent with access

Page Fault Handling (2)

5. If frame selected dirty, page is scheduled for transfer to disk, context switch takes place, suspending faulting process
6. As soon as frame clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
7. When disk interrupt indicates page has arrived, tables updated to reflect position, and frame marked as being in normal state.

Page Fault Handling (3)

8. Faulting instruction backed up to state it had when it began and program counter is reset
9. Faulting process is scheduled, operating system returns to routine that called it.
10. Routine reloads registers and other state information, returns to user space to continue execution

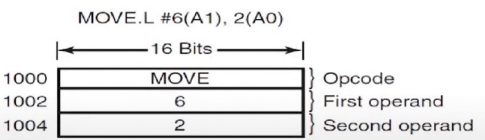
Instruction Backup

•In Fig. 3-27, an instruction starting at address 1000 that makes three memory references:

–the instruction word and

–two offsets for the operands.

•Depending on which of these three memory references caused the page fault, the program counter might be 1000, 1002, or 1004 at the time of the fault.



•Figure 3-27. An instruction causing a page fault.

Instruction Backup

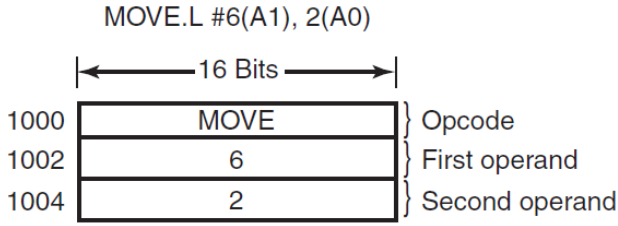


Figure 3-27. An instruction causing a page fault.

Locking Pages in Memory

- If a process starts an IO read and then blocks, then the page that holds the buffer can be replaced
- To prevent this, some pages might be locked or pinned.
- Or all IO can be done on kernel buffers

Backing Store

- Where do we store the pages on disk?
- The simplest algorithm is to have a special swap partition on the disk
- Disadvantage: wasted disk space
- Alternative: allocate disk space for each page when it is swapped out

Backing Store

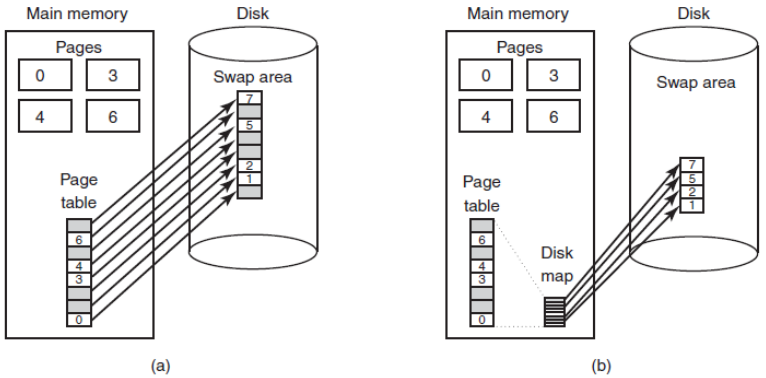


Figure 3-28. (a) Paging to a static swap area. (b) Backing up pages dynamically.

Separation of Policy and Mechanism (1)

- Split policy from mechanism
- Memory management system is divided into three parts
  1. A low-level MMU handler: assembly, machine dependent
  2. A page fault handler that is part of the kernel: machine independent, mechanism for paging.
  3. An external pager running in user space: policy, machine independent

Separation of Policy and Mechanism (2)

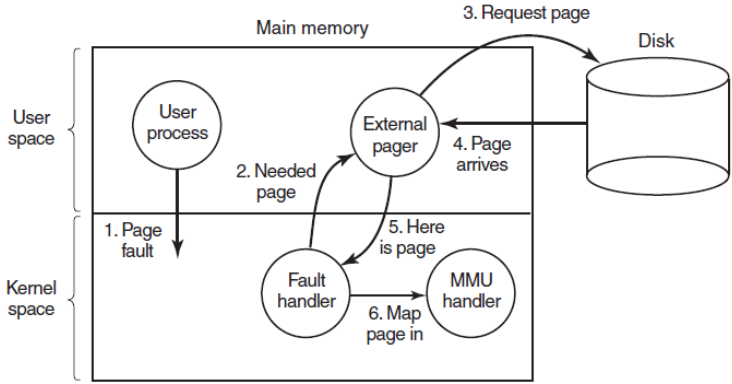


Figure 3-29. Page fault handling with an external pager.

Separation of Policy and Mechanism (3)

- **Split policy from mechanism**
- R and M bits are not accessible from the external pager
- Many switches to the kernel
- But
- This is very modular and easier to manage

Segmentation

- The virtual memory discussed so far is one-dimensional because the virtual addresses go from 0 to some maximum address, one address after another.
- For many problems, having two or more separate virtual address spaces may be much better than having only one.

Segmentation (1)

Examples of tables generated by compiler:

1. The source text being saved for the printed listing
2. The symbol table, names and attributes of variables.
3. The table containing integer and floating-point constants used.
4. The parse tree, syntactic analysis of the program.
5. The stack used for procedure calls within compiler.

Segmentation (2)

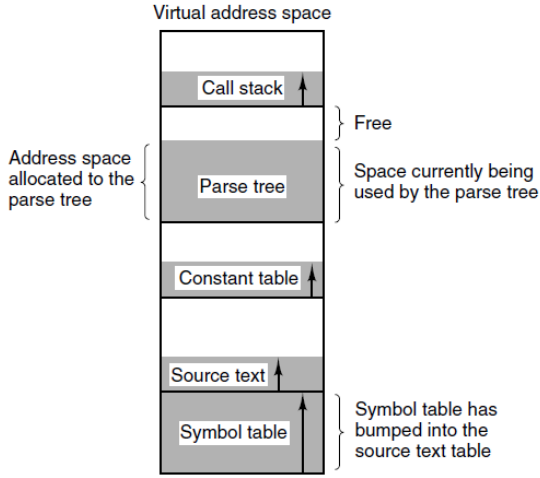


Figure 3-30. In a one-dimensional address space with growing tables, one table may bump into another.

Segmentation (3)

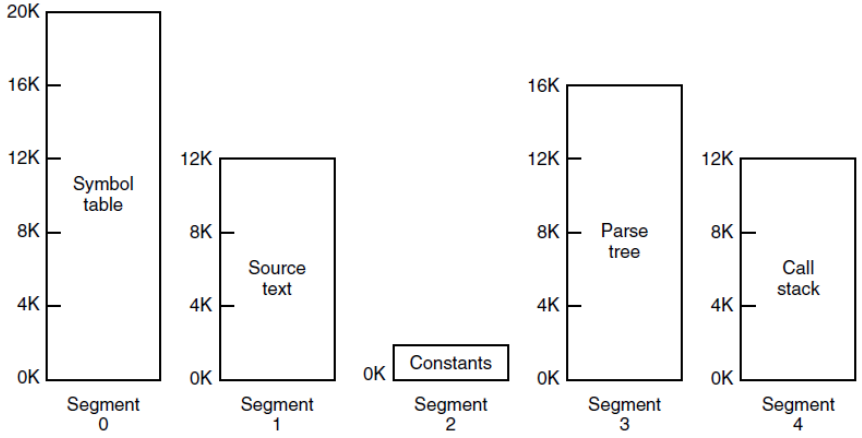


Figure 3-31. A segmented memory allows each table to grow or shrink independently of the other tables.

Segmentation (3)

- a segment is a logical entity, which the programmer is aware of and uses as a logical entity.
- A segment might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types.

Segmentation (3)

- If each procedure occupies a separate segment, with address 0 as its starting address, the linking of procedures compiled separately is greatly simplified
- With one dimentional memory, changing one procedure's size can affect the starting address of all the other (unrelated) procedures in the segment.

Segmentation (3)

- In a segmented system, a shared graphical library can be put in a segment and shared by multiple processes, eliminating the need for having it in every process' address space.

Segmentation (4)

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

Figure 3-32. Comparison of paging and segmentation

Implementation of Pure Segmentation

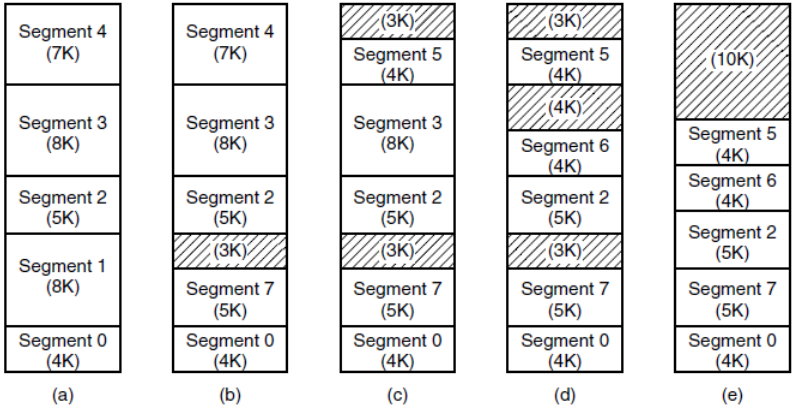


Figure 3-33. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

MULTICS (1)

- The MULTICS operating system was one of the most influential operating systems ever, having had a major influence on topics as disparate as UNIX, the x86 memory architecture, TLBs, and cloud computing.
- It was started as a research project at M.I.T. and went live in 1969.
- The last MULTICS system was shut down in 2000, a run of 31 years.

Segmentation with Paging: MULTICS (1)

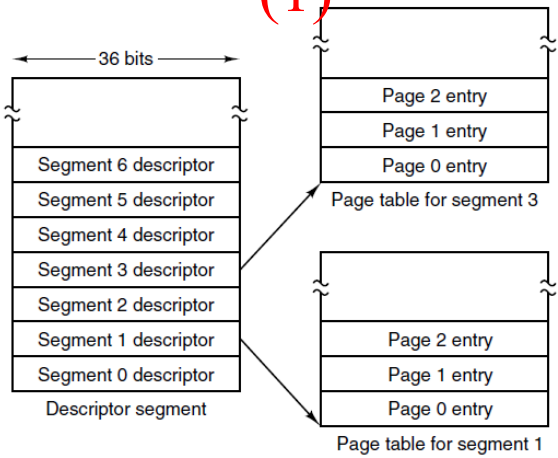


Figure 3-34. The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables.

Segmentation with Paging: MULTICS (2)

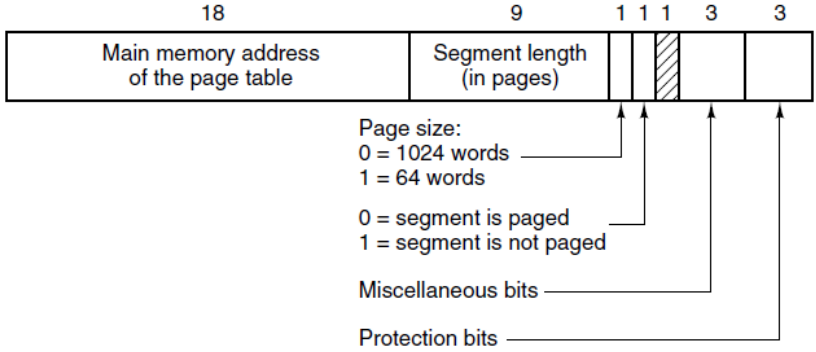


Figure 3-34. The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

Segmentation with Paging: MULTICS (3)

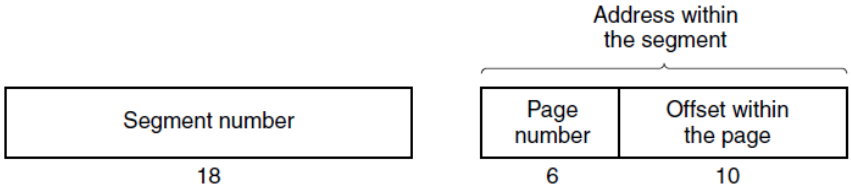


Figure 3-35. A 34-bit MULTICS virtual address.

Segmentation with Paging: MULTICS

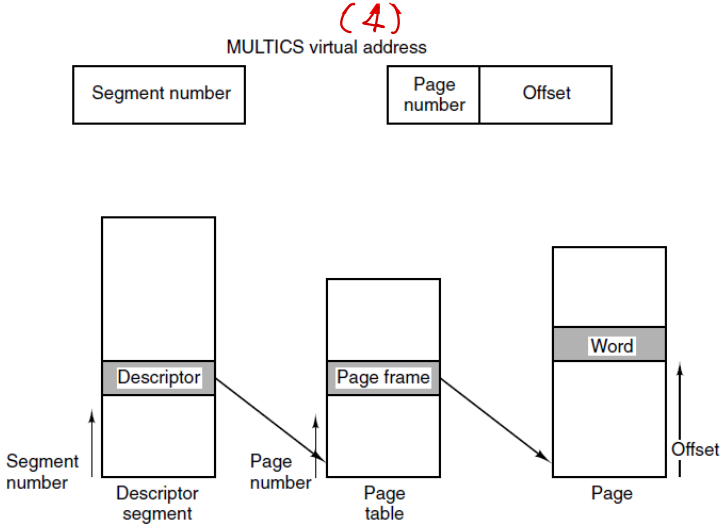


Figure 3-36. Conversion of a two-part MULTICS address into a main memory address.

Segmentation with Paging: MULTICS (5)

Comparison field

Segment number	Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Figure 3-37. A simplified version of the MULTICS TLB. The existence of two page sizes made the actual TLB more complicated.

Segmentation with Paging: The Intel x86 (1)

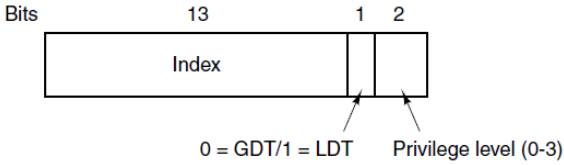


Figure 3-38. An x86 selector.

Segmentation with Paging: The Intel x86 (2)

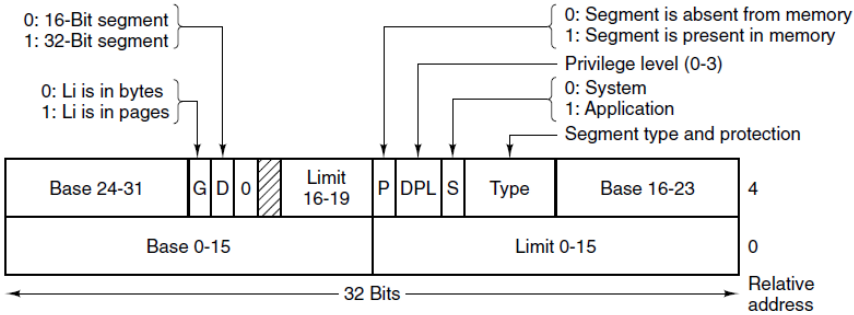


Figure 3-39. x86 code segment descriptor. Data segments differ slightly.

Segmentation with Paging: The Intel x86 (3)

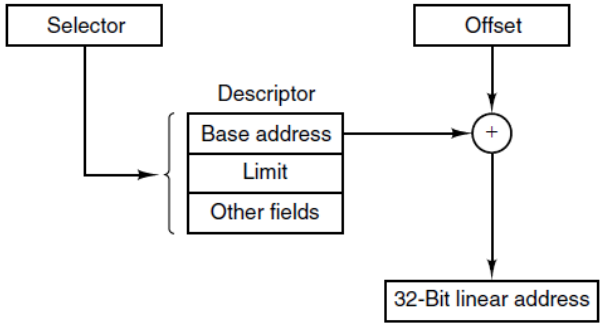


Figure 3-40. Conversion of a (selector, offset) pair to a linear address.

Segmentation with Paging: The Intel x86 (4)

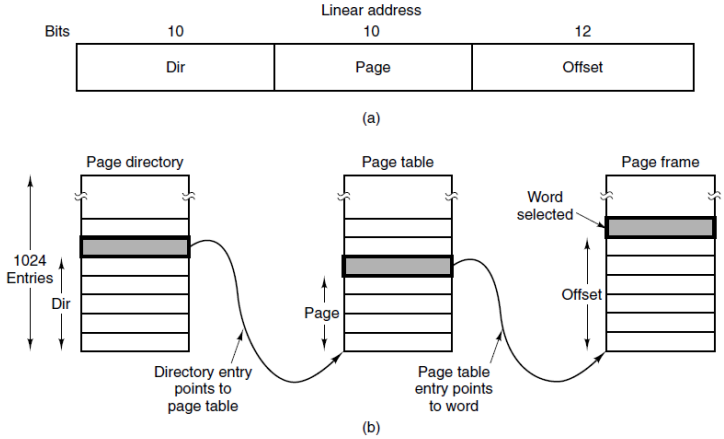


Figure 3-41. Mapping of a linear address onto a physical address.

End

Chapter 3