CSC 539: Operating Systems Structure and Design

Spring 2006

Memory management

- swapping
- contiguous allocation
- paging
- segmentation
- segmentation with paging

Memory management

memory management and CPU scheduling are perhaps the two most central tasks of an OS

- memory consists of a large array of words, each with its own address
- both program and data are stored in memory

recall: CPU fetch-execute cycle

1. fetch next instruction from memory (based on program counter)
2. decode the instruction
3. possibly fetch operands from memory
4. execute the instruction
5. store the result in memory

note: memory is given an address and it returns a value

- it does not distinguish between instructions and data
- it does not care how the address was arrived at

Address binding

users do not (cannot?) think about real addresses

- user processes go through several steps before execution (e.g., compilation, loading, linking, …)
- addresses may be represented in different ways during these steps (symbol, relocatable addr, …)

instructions/data may be bound to real addresses:

at compile time:

compiler substitutes actual addresses in executable

used in MS-DOS (why?)

at load time:

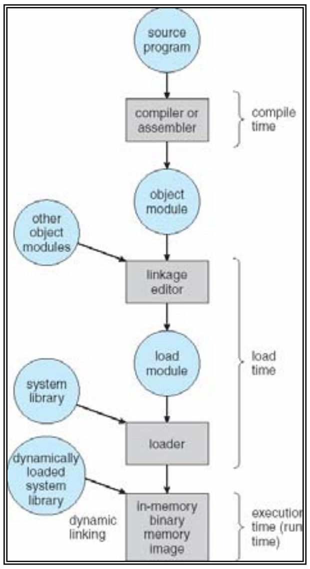
compiler generates relocatable code, final binding is delayed until load time

once loaded in memory and process starts, bindings are locked in – not generally used (why?)

at execution time:

if process can be moved during its execution, then binding must be delayed until run time

requires special hardware support – used in most general-purpose OS's (why?)



Logical vs. physical addresses

the concept of a logical address space that is mapped to a physical address space is central to memory management

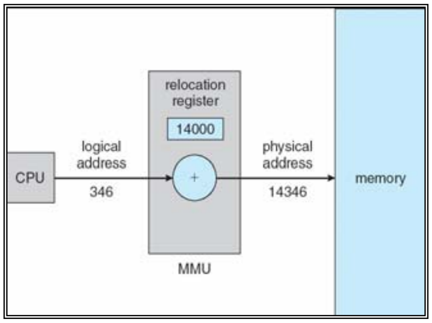
- CPU works in a logical (virtual) address space

  users, processes, instruction operands, program counter refer to logical addresses
- memory is a physical address space

  real memory accesses require real addresses

Memory Management Unit (MMU) maps logical addresses to physical addresses

- implemented in hardware
- for example, might have a relocation register that provides the base address for a process' block of physical memory



Swapping

a process needs to be in memory to be executed

- however, it may be temporarily swapped out of memory into a backing store
- the process can later be restarted by swapping it back into memory
- backing store is generally a fast disk – speed is important!

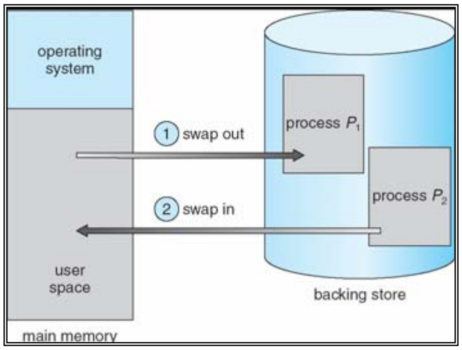
swapping example:

when a process blocks or timeouts,

- swap it out to disk
- swap in a new (ready) process from disk to memory
- dispatch the new process

another example: *roll out, roll in*

- in a priority-based system, lower-priority process may be swapped out so higher-priority process can be loaded and executed



To swap or not to swap

few operating systems use standard swapping

- the context-switch time is generally too costly
- special care must be taken, e.g., can't swap out a process waiting on I/O

some operating systems use variations on swapping

- Windows 3.1: if new process is loaded but insufficient memory, swap out old
  however, user decides when to swap out & swap in processes
- some versions of UNIX will use paging if system load is very high

swapping may be used in combination with other approaches (e.g., segmentation – later)

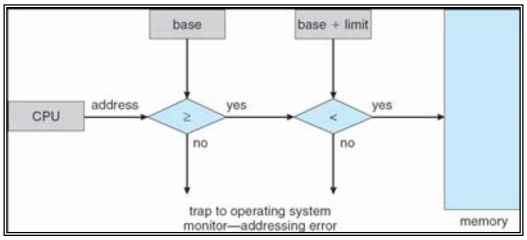
Contiguous memory allocation

main memory must accommodate both the OS and user processes

- usually divided into two partitions:
  OS & interrupt vector in low memory, user processes in high memory
- *contiguous memory allocation* places each process in a single, contiguous section of memory

OPTION 1: single partition (all processes share same memory space)

- relocation-register scheme used to protect user processes from each other, and from changing OS code and data
- relocation register contains smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register

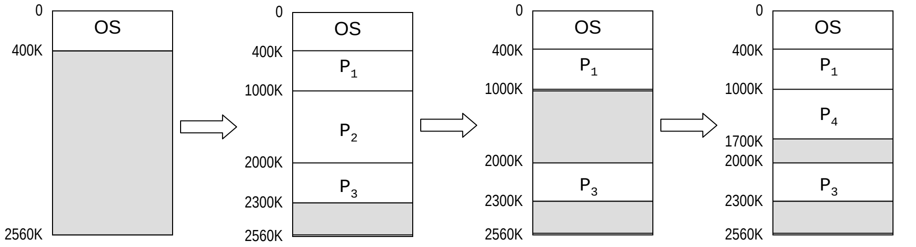


Contiguous memory allocation (cont.)

OPTION 2: multiple partitions (each process has its own partition)

- a hole is a block of available memory
- when a process arrives, it is allocated memory from a sufficiently large hole
- OS maintains info about allocated and free partitions (holes)

process	memory
P_1	600K
P_2	1000K
P_3	300K
P_4	700K
P_5	500K

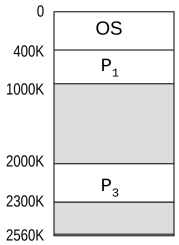


Dynamic storage allocation problem

as processes free up holes in memory and new processes request memory, how do you allocate partitions?

- first-fit: allocate the first hole that is big enough
- best-fit: allocate the smallest hole that is big enough
  must search the entire list of holes unless ordered by size
  produces the smallest leftover hole
- worst-fit: allocate the largest hole available
  also must search the entire list of holes
  produces the largest leftover hole

process	memory
P_1	600K
P_2	1000K
P_3	300K
P_4	200K
P_5	100K



first-fit is fast

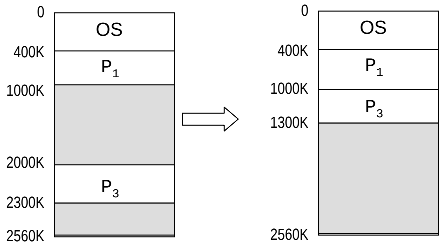
best-fit tends to work better w.r.t. storage utilization

Fragmentation

with dynamic storage allocation, (external) fragmentation can be a problem

- there may be many small holes, none big enough to fit a process
  50% rule: in practice, given N allocated blocks, expect N/2 blocks lost to fragmentation
- compaction is needed to coalesce holes (similar to defragging a hard drive)
  note: compaction is possible only if bindings are done at execution time

process	memory
P_1	600K
P_2	1000K
P_3	300K
P_4	1200K



internal fragmentation can occur as well

- allocated memory may be slightly larger than requested memory
- e.g., suppose memory is allocated in blocks of 4K
  → a 1K process will waste 3K of space in its partition, as will a 5K process

Paging

paging is one solution to the external fragmentation problem

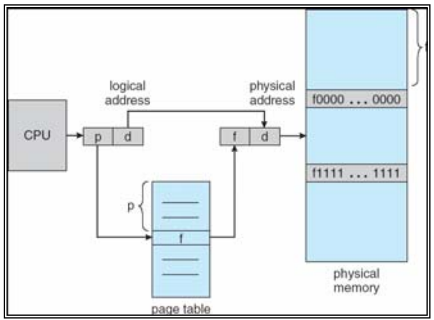
- noncontiguous storage allocation – a process is allocated physical memory wherever it is available

  divide the physical memory into fixed-size blocks called *frames*
  (size is a power of 2, generally between 512 and 8K bytes)
  divide logical memory into blocks of the same size called *pages*

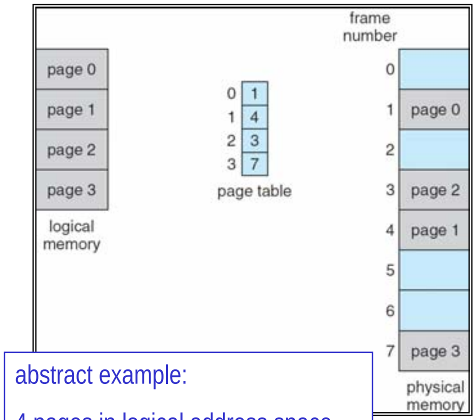
- to run a process of size n pages, need to find n free frames and load

every logical address generated by the CPU is divided into:

- page number (p): used as an index into a page table containing the base address of each page in physical memory
- page offset (d): combined with base address to determine the physical memory address



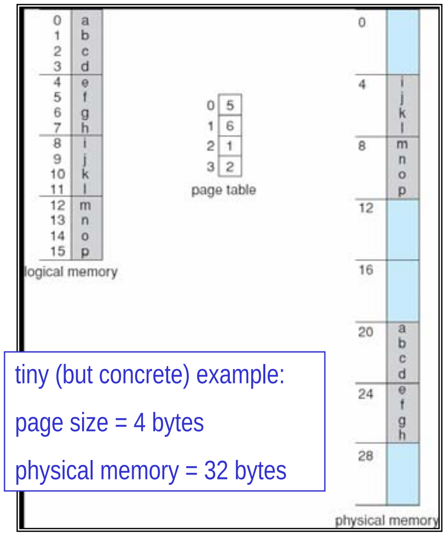
Paging examples



abstract example:

4 pages in logical address space

page table maps to physical frames



tiny (but concrete) example:

page size = 4 bytes

physical memory = 32 bytes

Hardware defines page/frame size

EXAMPLE: logical address is 32 bits (address space 2^{32} = 4 GB)

10 bits are used for page number → 2^{10} = 1024 pages
22 bits are used for offset → 2^{22} = 4 MB of addresses per page

0010001001	0000110110011001110111
p	d

more, smaller pages are obtained by increasing p & decreasing d

12 bits are used for page number → 2^{12} = 4K pages
20 bits are used for offset → 2^{20} = 1 MB of addresses per page

001000100111	00001101100110011101
p	d

What paging gives us

- there is no external fragmentation
- internal fragmentation is minimized
  all but the last frame allocated to a process will be full (on avg, will waste ½ a frame)
- the page table is used to dynamically convert logical addresses into physical addresses
- small frames reduce fragmentation, but increase the size of the page table

Frame table

OS needs to keep track of allocated and free frames in memory

- system-wide frame table keeps this information
- when a new process arrives, it is allocated frames that are free (if there are not enough then the process must wait)

Frame number	Alloc/free	Owner	Page in process
0	alloc	6	23
1	free	-	-
2	free	-	-
3	alloc	3	6

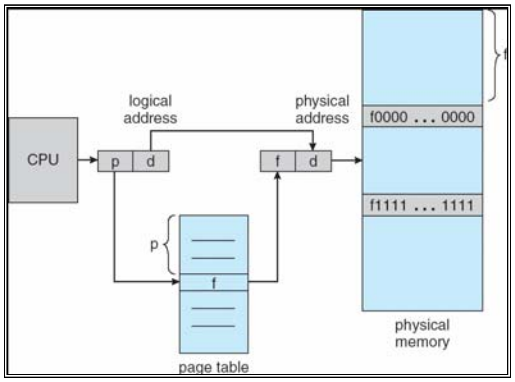
Structure of a page table

each process has its own page table, stored in main memory (PCB)

- Page Table Base Register (PTBR) points to the page table for that process
- PTBR changed by the OS whenever a process is dispatched, or context-switch

note: every data/instruction access now requires 2 memory accesses

1. one access to get the page table entry, so can construct physical address
2. one more to get the data/instruction stored at that address



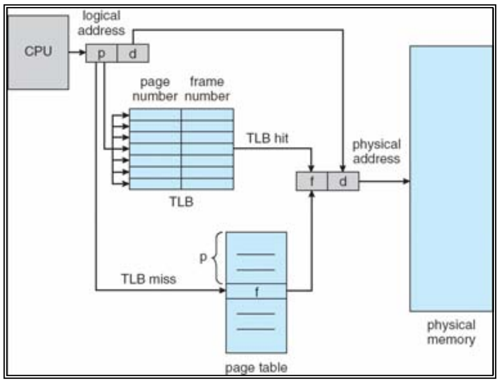
Translation look-aside buffer

can reduce to a single memory access using an *associative array*

- Translation Look-aside Buffer (TLB) provides constant-time access to frame number given page number (using a parallel search)

- TLB retains the most recent references to the page table
- MMU first checks to see if the page in the logical address is in the TLB, if not then must access page table
- *note similarity to caching*
- suppose memory access = 100ns,
  TLB access = 10 ns

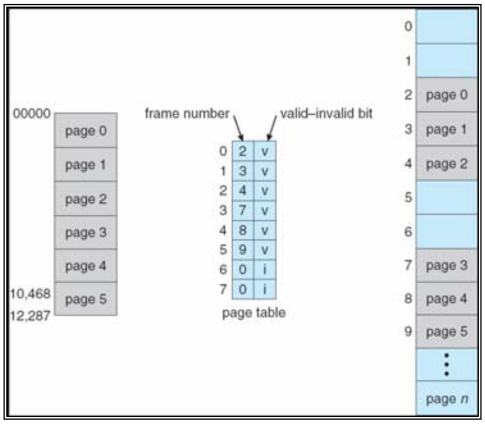
  if 80% of references are in TLB, then effective memory reference time is
  (210 * 0.20 + 110 * 0.80) = 130 ns



Memory protection with paging

can associate a protection bit with each frame

- *valid* indicates that the page in that frame is in the process' logical address space
  → legal to access
- *invalid* indicates that the page is not in the process' logical address space
  → not legal to access



Handling large address spaces

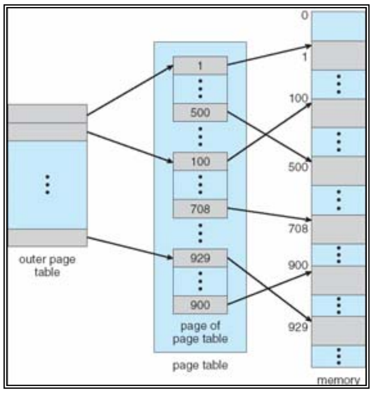
suppose page size is 4KB, address space uses 32 bits (references 4 G)

requires a page table with 1 M entries, each entry is 4 bytes → 4 MB of main memory

multi-level paging: break the table into pieces, have another table tell where the pieces are

page number(s)		page offset
p_1	p_2	d

p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.



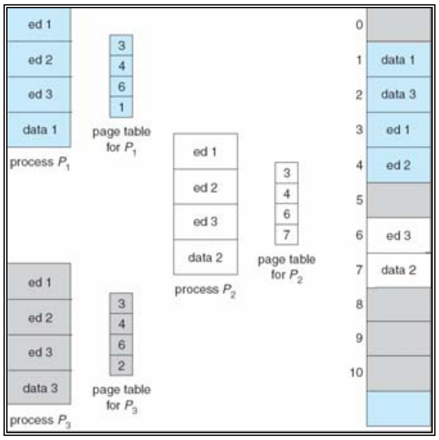
hashed page table: store page table info in hash table for faster access & less waste

hash key is page number; entry stores the frame number

Shared pages

pages can be shared among processes

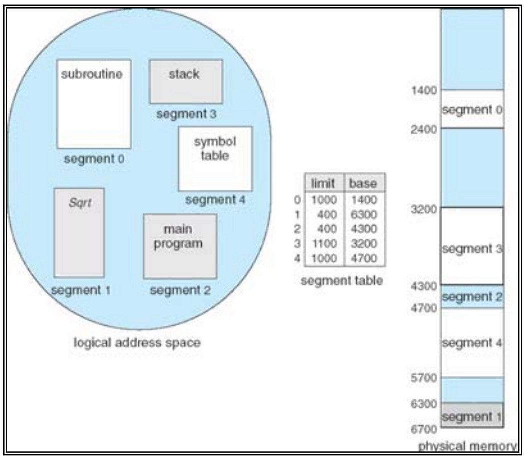
- read-only reentrant code can be shared without fear of interference
- a frame in main memory can be referenced by several different page tables
- OS must ensure that a frame cannot be removed from memory if any active processes are referencing it



Segmentation

segmentation is similar to paging, except that partitioning is based on the user's view of memory

- user sees a program as a collection of segments
  e.g., main program, function, object, global variables, stack, symbol table, …



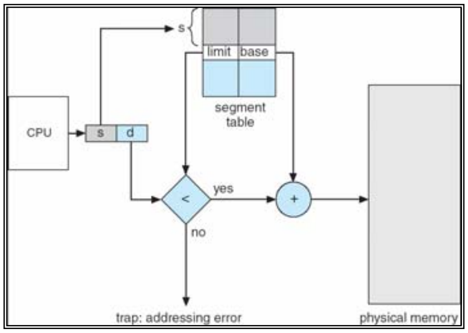
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segmentation architecture

the compiler creates segments for main, functions, globals, …

- each segment is stored separately in memory
- must have hardware support to map logical address (segment number + offset) to physical address

- segment table maps segment number to base address of segment in memory
- segment table base register (STBR) points to the segment table's location in memory
- segment table length register (STLR) indicates number of segments used by program



Advantages of segmentation

view of memory is the user's view

segments are protected from one another

- each segment contains one type of information (e.g., instructions, stack, …)

sharing segments is logical and easy

- if all the instructions are in one segment and all data in another, the instruction segment can be shared freely by different processes (each with own data)

as with paging, can implement protection using a valid/invalid bit

DISADVANTAGE: unlike paging, external fragmentation can be a problem

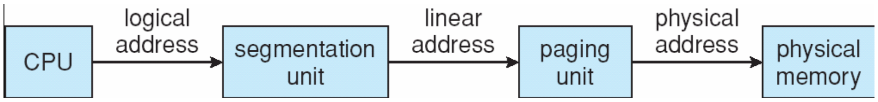
Segmentation with paging

modern computers use a combination of segmentation and paging

- solve the external fragmentation problem by paging the segments
- segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for the segment

example: Intel Pentium

- utilizes both pure segmentation and segmentation with paging
- the segmentation and paging units act in sequence to map a logical address to physical memory

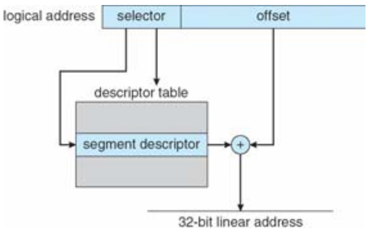


Pentium segmentation & paging

segmentation

- max segment size = 4 GB
- max # of segments per process = 16 KB

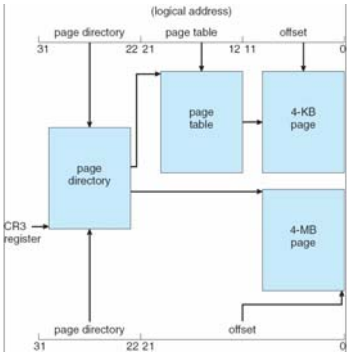
- logical address is 48 bits long
  16-bit selector identifies the segment
  32-bit offset specifies location in segment



paging

- page size = 4 KB or 4MB

- utilizes 2-level paging scheme
  first 10 bits index a page directory
  next 10 bits index corresponding page table
  remaining 12 bits specify offset in page

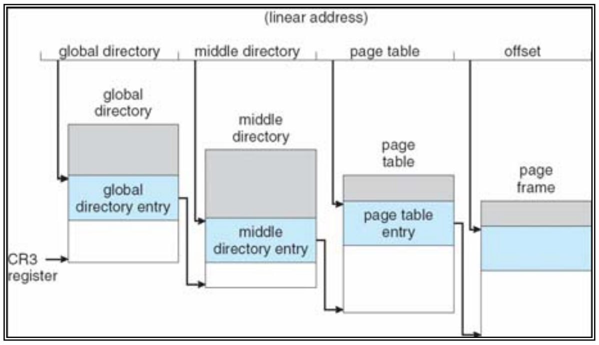


Linux on Pentium Systems

Linux is designed to work on a variety of processors

- since some processors don't support segmentation, Linux doesn't rely on it
- on the Pentium, Linux uses only 6 segments
  kernel code, kernel data, user code, user data, task-state, local-descriptor table

- since some processors support 64-bit addresses, 2-level paging is not sufficient
- in general, Linux uses 3-level paging
  but uses 0 bits for middle table if only 32-bits are supported (as with Pentium)


(linear address)
global directory
middle directory
page table
offset
global directory
global directory entry
middle directory
middle directory entry
page table
page table entry
page frame
CR3 register