

## Previous Lecture: Deadlocks

- A problem/situation with concurrency for both SW (e.g., Java threads) and HW (e.g., interconnection network in a multiprocessor based system) systems.
- Different ways to deal with:
  - ◆ Ignore (common in today's OS)
  - ◆ Prevent and avoid (same as prevent but needs a priori resource usage information)
  - ◆ Detect (periodically run detection to check for deadlock, interrupt/terminate some of the processes and/or preempt resources that are held)
- Practical Lesson: You need to make sure your application code is NOT DEADLOCKED! The OS will not help you out...
- Good idea to use/construct the Resource Allocation Graph even at the application level.
- Distributed systems and OS – the deadlock problem is further complicated...

## Chapter 9: Memory Management

- One of the most critical OS components
- Outline:
  - ◆ Background
  - ◆ Swapping
  - ◆ Contiguous Allocation
  - ◆ Paging
  - ◆ Segmentation
  - ◆ Segmentation with Paging

## Background

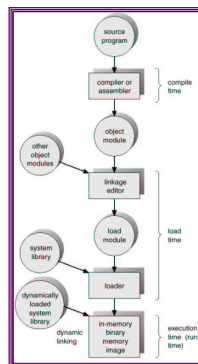
- Program must be brought into memory and placed within a process for it to be run.  
Assume that the program must fit in memory or the program itself needs to deal with when not. This is different from virtual memory (will be covering later) when this (i.e. fit or not in memory) will be managed by the OS!
- User programs go through several steps before being run. Let us review these steps....

## Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes. For example the MS-DOS .COM format executables...
- **Load time:** Must generate *relocatable code* (e.g., say from the beginning of this module rather than address 100) if memory location is not known at compile time.
- **Execution time:** Address binding delayed until run time, if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base and limit registers*), for efficiency reasons. Most general-purpose OS.

## Multi-step Processing of a User Program



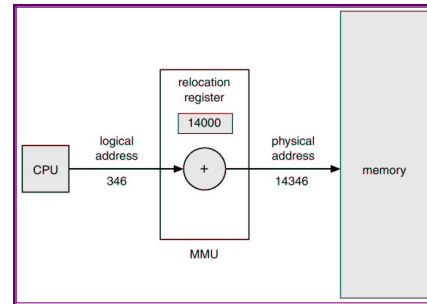
## Logical vs. Physical Address Space

- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management.
  - ◆ *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - ◆ *Physical address* – address seen by the memory unit.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

## Memory-Management Unit (MMU)

- Hardware device that maps virtual address to physical address.
- In a simple MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- There are many different methods for accomplishing such a mapping, as we will see later.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

## Dynamic relocation using a relocation register



## Dynamic Loading

- In our discussion so far we required that the entire data and program must be in physical memory before it starts executing.
- Dynamic Loading – does not require that in order to obtain better memory-space utilization.
- Idea: Routine is not loaded until it is called
  - ◆ Better memory-space utilization as unused routine is never loaded.
  - ◆ Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required implemented through program design. It is the responsibility of the user to take advantage of this method. OS may provide library routines to implement dynamic loading.

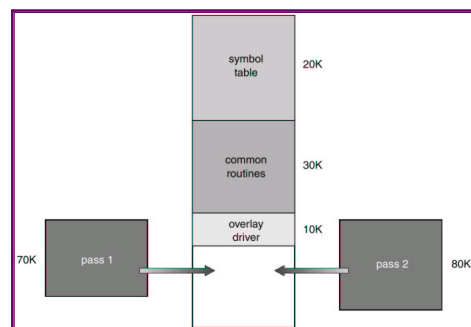
## Dynamic Linking

- Some OS supports static linking in which libraries are treated as any other object module
- Dynamic Linking: linking postponed until execution time.
- Small piece of code, *stub*, is included for each library reference in the image; used to locate the appropriate memory-resident library routine.
  - ◆ Stub replaces itself with the address of the routine, and executes the routine.
  - ◆ Operating system needed to check if routine is in processes' memory address. If routine is not in memory, the stub loads it into memory. That is, only first time dynamic linking incurs the overhead.
- Dynamic linking is particularly useful for libraries to avoid building in large pieces of static libraries in the executable image (i.e., as with the case of static libraries). That (may) affect both memory allocation and (instruction) caching performance.

## Overlays

- A process can be larger than the available amount of memory with overlays.
- Idea: Keep in memory only those instructions and data that are needed at any given time. Overload instructions that are no longer needed.
  - ◆ Needed/useful when process is larger than amount of memory allocated to it.
  - ◆ Implemented by user, no special support needed from operating system, programming design of overlay structure is complex; Require that the programmer has complete knowledge of the program, its code, and its data structures.
- Automatic techniques to run large programs in limited physical memory are certainly preferred.

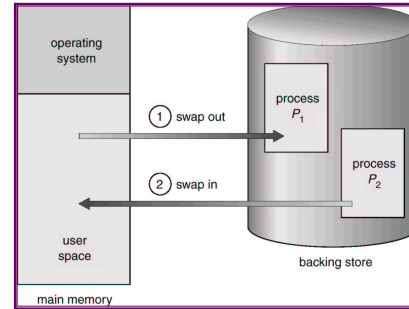
## Overlays for a Two-Pass Assembler



## Swapping

- A process needs to be in memory to be executed.
- Idea: A process can however be *swapped temporarily out of memory to a backing store*, and then brought back into memory for continued execution.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms; *lower-priority process is swapped out* so if *higher-priority process* arrives and wants service it can be *loaded* and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped. When a process is swapped back will be loaded into same memory (if not execution time binding is used)
- Modified versions of swapping are found on many systems, i.e., UNIX, Linux, and Windows.

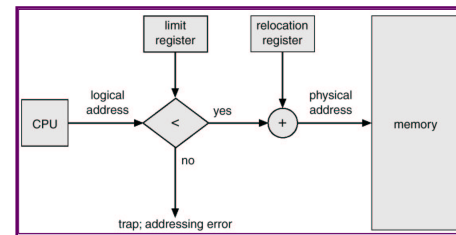
## Schematic View of Swapping



## Contiguous Allocation

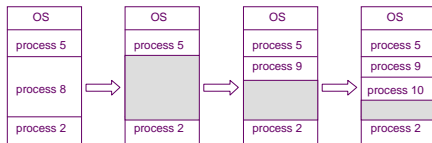
- Main memory usually divided into two partitions:
  - ◆ Resident operating system, usually held in low memory with interrupt vector. Since the interrupt vector is typically in low-memory it is more common to map the resident OS into low-memory.
  - ◆ User processes then held in high memory.
- ◆ Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data size dynamically (e.g., device driver code and other transient OS code is not kept in memory when not needed anymore).
- ◆ Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register. Every address is checked against these.
- Fixed-partition allocation: each partition may contain exactly one process; degree of multiprogramming bound to number of processes. IBM OS/360

## Hardware Support for Relocation and Limit Registers



## Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - ◆ *Hole* – block of available memory; holes of various size are scattered throughout memory.
  - ◆ When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - ◆ Operating system maintains information in a table about:
    - a) allocated partitions
    - b) free partitions (hole)



## Dynamic Storage-Allocation Problem

How to satisfy a request of size  $n$  from a list of free holes.

- **First-fit:** Allocate the *first* hole that is big enough.
- **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit:** Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

First-fit and best-fit performs better than worst-fit in terms of speed and storage utilization. The algorithms suffer from external fragmentation.

## Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous, rather it is small holes.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used. This because the physical memory is using fixed-size blocks... so if a process requests 1021 bytes it is more efficient to allocate 1024 (power of two rule)
- Reduce external fragmentation by compaction
  - ◆ Shuffle memory contents to place all free memory together in one large block.
  - ◆ Compaction is possible *only* if relocation is dynamic (by changing base registers and moving data and code), and is done at execution time.
  - ◆ I/O problem
    - ✓ Latch job in memory while it is involved in I/O.
    - ✓ Do I/O only into OS buffers.

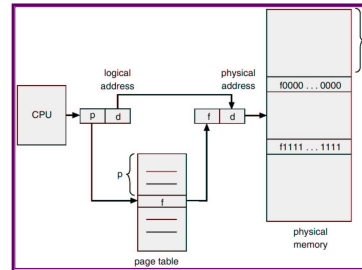
## Paging

- **Logical address space of a process can be noncontiguous;** process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called **pages**.
- Keep track of all free frames.
- To run a program of size  $n$  pages, need to find  $n$  free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation.

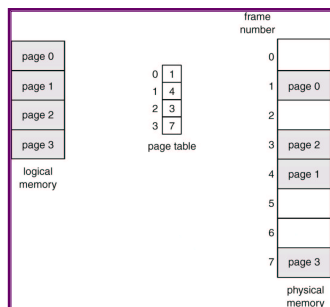
## Address Translation Scheme

- Address generated by CPU is divided into:
  - ◆ **Page number ( $p$ )** – used as an index into a **page table** which contains base address of each page in physical memory.
  - ◆ **Page offset ( $d$ )** – combined with base address to define the physical memory address that is sent to the memory unit.

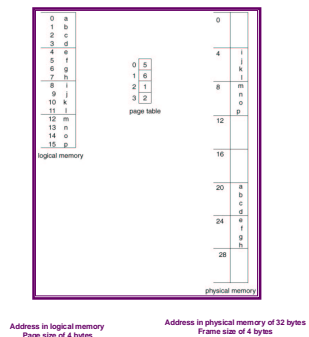
## Address Translation Architecture



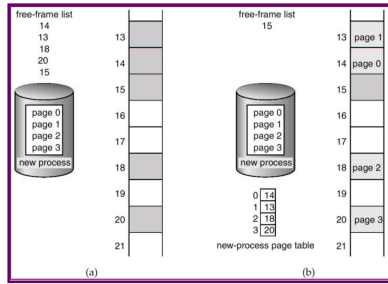
## Paging Example



## Paging Example



## Free Frames



Before allocation

After allocation

## Implementation of Page Table

- Page table is kept in main memory.
- Page-table base register (PTBR) points to the page table.
- Page-table length register (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)

## Associative Memory

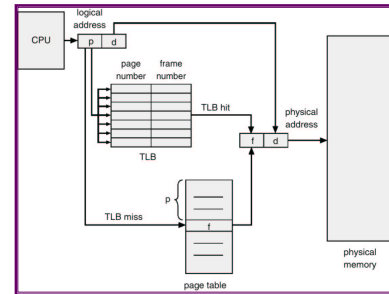
- Associative memory – parallel search

Page #	Frame #

Address translation ( $A'$ ,  $A''$ )

- If  $A'$  is in associative register, get frame # out.
- Otherwise get frame # from page table in memory

## Paging Hardware With TLB



## Effective Access Time

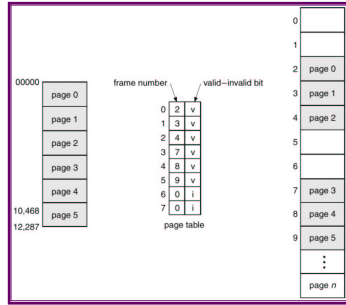
- Associative Lookup =  $\epsilon$  time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers.
- Hit ratio =  $\alpha$
- Effective Access Time (EAT)
 
$$EAT = (1 + \epsilon) \alpha + (2 + \epsilon)(1 - \alpha)$$

$$= 2 + \epsilon - \alpha$$
- THIS WOULD BE THE CASE IF TLB hit-time would be equal a page table memory access. HOWEVER TLB access is MUCH faster with current technology! It is a small (30-64 entries) SRAM based CAM (Content Addressable Memory) structure typically compared to accessing DRAM. Page table access may be rare enough so it is not in the cache...
- Effective Access Time (EAT) (assuming from DRAM)
 
$$EAT = (L_{tlb} + \epsilon) \alpha + (2L_{dram} + \epsilon)(1 - \alpha)$$
- $L_{dram} \sim 60\text{nsec}$ ,  $L_{tlb} \sim 2\text{-}3\text{ nsec}$ ,  $\alpha$  is 90%, insights?

## Memory Protection

- Memory protection implemented by associating protection bit with each frame.
- Valid-invalid bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page.
  - "invalid" indicates that the page is not in the process' logical address space.

## Valid (v) or Invalid (i) Bit In A Page Table



## Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

## Hierarchical Page Tables

- Break up the logical address space into multiple page tables.
- A simple technique is a two-level page table.

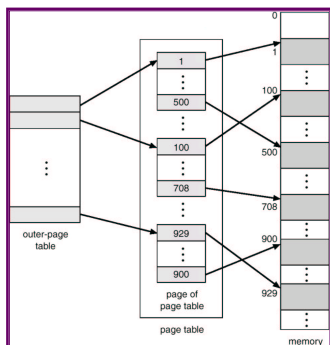
## Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - ✦ a page number consisting of 20 bits.
  - ✦ a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - ✦ a 10-bit page number.
  - ✦ a 10-bit page offset.
- Thus, a logical address is as follows:

page number		page offset
$p_1$	$p_2$	$d$
10	10	12

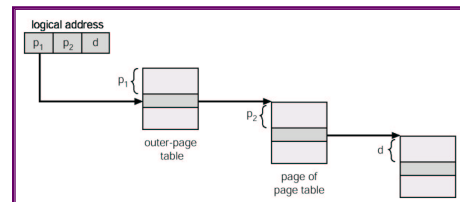
where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the outer page table.

## Two-Level Page-Table Scheme



## Address-Translation Scheme

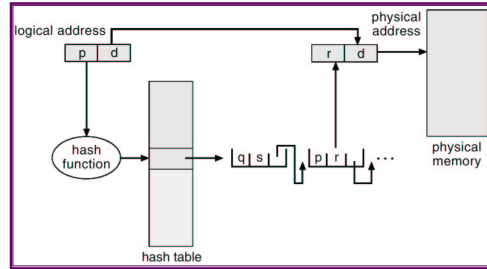
- Address-translation scheme for a two-level 32-bit paging architecture



## Hashed Page Tables

- Common in address spaces > 32 bits.
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

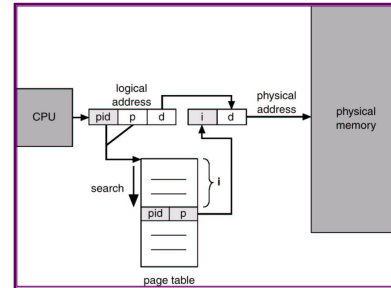
## Hashed Page Table



## Inverted Page Table

- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.

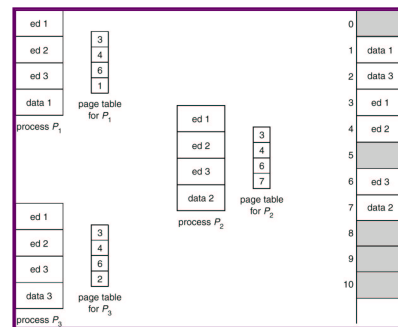
## Inverted Page Table Architecture



## Shared Pages

- Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.
- Private code and data**
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

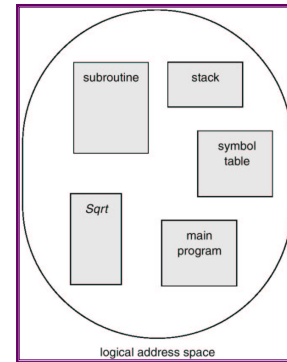
## Shared Pages Example



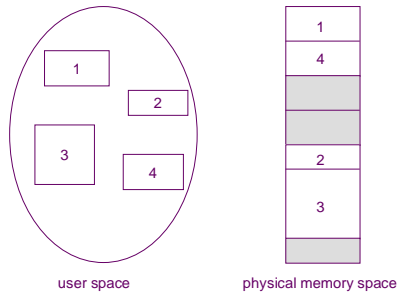
## Segmentation

- Memory-management scheme that supports user view of memory.
- A program is a collection of segments. A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays

## User's View of a Program



## Logical View of Segmentation



## Segmentation Architecture

- Logical address consists of a two tuple:
  - <segment-number, offset>
- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - ♦ base – contains the starting physical address where the segments reside in memory.
  - ♦ limit – specifies the length of the segment.
- *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 a program;
  - segment number  $s$  is legal if  $s < STLR$ .

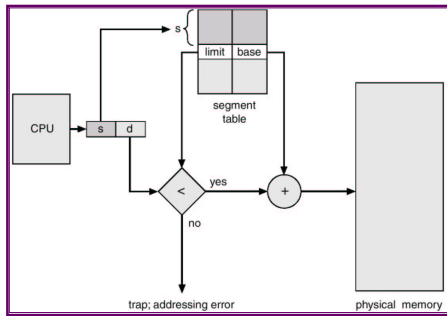
## Segmentation Architecture (Cont.)

- Relocation.
  - ♦ dynamic
  - ♦ by segment table
- Sharing.
  - ♦ shared segments
  - ♦ same segment number
- Allocation.
  - ♦ first fit/best fit
  - ♦ external fragmentation

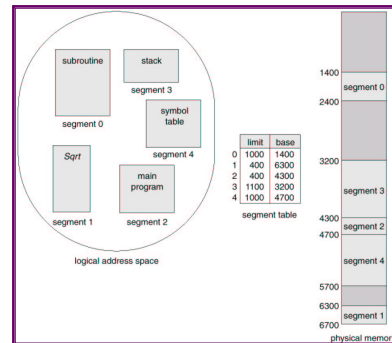
## Segmentation Architecture (Cont.)

- Protection. With each entry in segment table associate:
  - ♦ validation bit = 0  $\Rightarrow$  illegal segment
  - ♦ read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.
- A segmentation example is shown in the following diagram

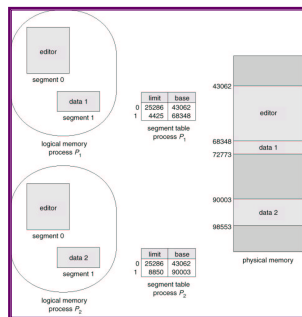
## Segmentation Hardware



## Example of Segmentation



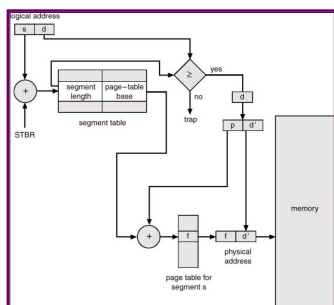
## Sharing of Segments



## Segmentation with Paging – MULTICS

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

## MULTICS Address Translation Scheme



## Segmentation with Paging – Intel 386

- As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.

## Intel 30386 Address Translation

