Silberschatz, et al.
Topics based on Chapter 9

Memory Management

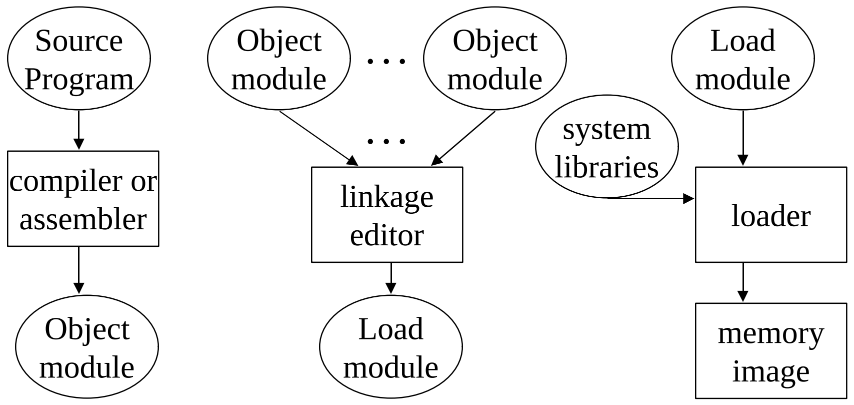
Memory Management

- Goal: permit different processes to <u>share</u> memory--effectively keep several in memory at the same time
- Eventual meta-goal: users develop programs in what appears to be their own infinitely-large address space (i.e., starting at address 0 and extending without limit)

Memory Management

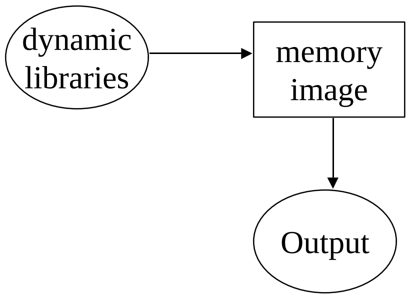
- Initially we assume that the entire program must be in physical memory before it can be executed.
  - How can we avoid including unnecessary routines? Programs consist of modules written by several people who may not [be able to] communicate their intentions to one another.
- Reality:
  - primary memory has faster access time but limited size; secondary memory is slower but much cheaper.
  - program and data must be in primary memory to be referenced by CPU directly

Multistep Processing of User Program



Compile time **----------------Load time---------------**

Multistep Processing of User Program



Run Time (execution time)

Multistep Processing of User Program

- **Binding**: associate location with object in program.
- For example, changing addresses in a user's program from logical addresses to real ones
- More abstractly, mapping one address space to another
- Many things can be bound in programming languages; we are concentrating on memory addresses here.

Binding

- Typically
  - compiler binds symbolic names (e.g., variable names) to relocatable addresses (i.e., relative to the start of the module)
  - linkage editor may further modify relocatable addresses (e.g., relative to a larger unit than a single module)
  - loader binds relocatable addresses to absolute addresses
- Actually, address binding can be done at any point in a design

When should binding occur?

- binding at *compile time*
  - generates absolute code. Must know at compile time where the process (or object) will reside in memory. Example: *0 in C. Limits complexity of system.
- binding at *load time*
  - converts compiler's relocatable addresses into absolute addresses at load time. The most common case. The program cannot be moved during execution.
- binding at *run time*
  - process can be moved during its execution from one memory segment to another. Requires hardware assistance (discussed later). Run-time overhead results from movement of process.

When should loading occur?

- Recall that loading moves objects into memory
- Load before execution
  - load all routines before runtime starts
  - straightforward scheme
- Load during execution--**Dynamic loading**
  - loads routines on first use
  - note that unused routines (<u>ones that are not invoked</u>) are not loaded
  - Implement as follows: on call to routine, check if the routine is in memory. If not, load it.

When should linking occur?

- Recall that linking resolves references among objects.
- Standard implementation: link before execution (hence all references to library routines have been resolved before execution begins). Called **static linking**.
- Link during execution: **dynamic linking**
  - *memory resident* library routines
  - every process uses the <u>same</u> copy of the library routines
  - hence linking is deferred to execution time, but loading is not necessarily deferred

Dynamic Linking

- Implementation of dynamic linking
  - library routines are not present in executable image. Instead *stubs* are present.
  - stub: small piece of code that indicates how to locate the appropriate memory-resident library routine (or how to load it if it is not already memory-resident)
  - first time that a routine is invoked, stub locates (and possibly loads) routine and then replaces itself with the address of the memory-resident library routine

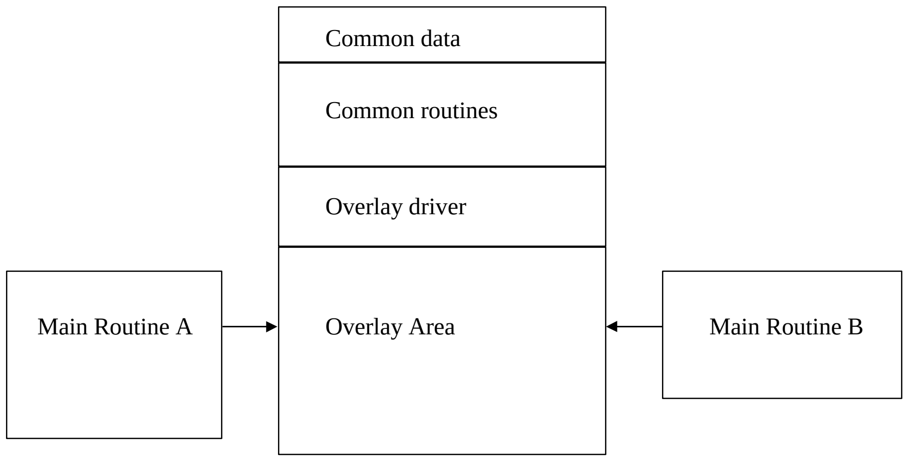
Dynamic Linking

- Also known as *shared libraries*
- Savings of overall memory (one copy of library routine) and of disk space (library routines are not in executable images).
- Expense: first use is more expensive
- Problem: incompatible versions
  - Can retain version number to distinguish incompatible versions of library. Alternative is to require upward compatibility in library routines.
  - If there are different versions, then you can have multiple versions of routine in memory at same time, counteracting a bit of the memory savings.
- Example: SUN OS's shared libraries

Overlays

- So far, the entire program and data of process must be in physical memory during execution.
- Ad hoc mechanism for permitting process to be larger than the amount of memory allocated to it: *overlays*
- In effect keeps only those instructions and data in memory that are in current use
- Needed instructions and data replace those no longer in use

Overlays Example



Overlays

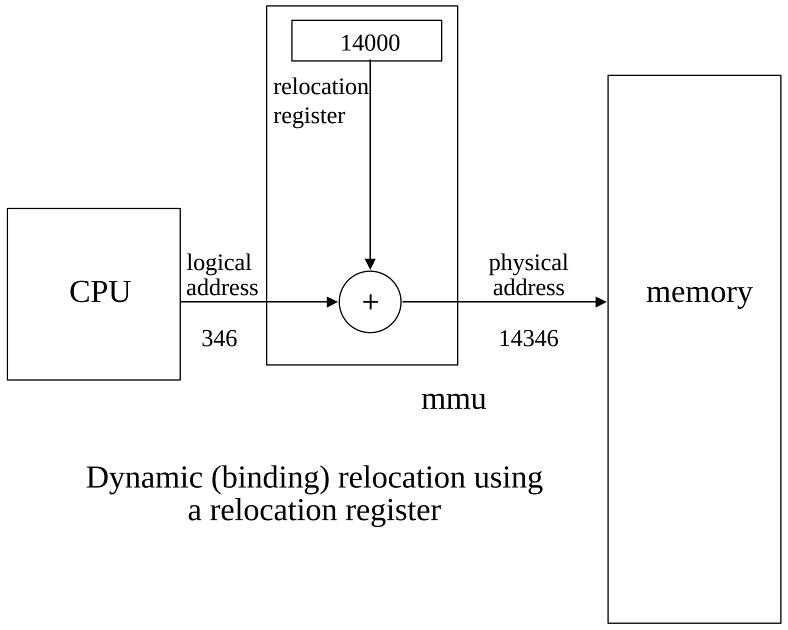
- Overlays do not require special hardware support--can be managed by programmer
- Programmer must structure program appropriately, which may be a difficulty
- Very common solution in early days of computers. Now, probably dynamic loading and binding are more flexible
- Example: Fortran common

Logical versus Physical Address Space

- **logical address**: generated by the CPU (logical address space)
- **physical address**: loaded into the *memory address register* of the memory (physical address space)
- compile-time and load-time address binding: logical and physical addresses are the same
- execution-time address binding: logical and physical addresses may differ
  - in this case, logical address referred to as **virtual address**

Mapping from Virtual to Physical Addresses

- Run-time mapping from virtual to physical address handled by the *Memory Management Unit* (*MMU*), a hardware device
- Simple MMU scheme
  - relocation register containing start position of process in memory
  - value in relocation register is added to every address generated by a user process when it sent to memory



Dynamic (binding) relocation using a relocation register

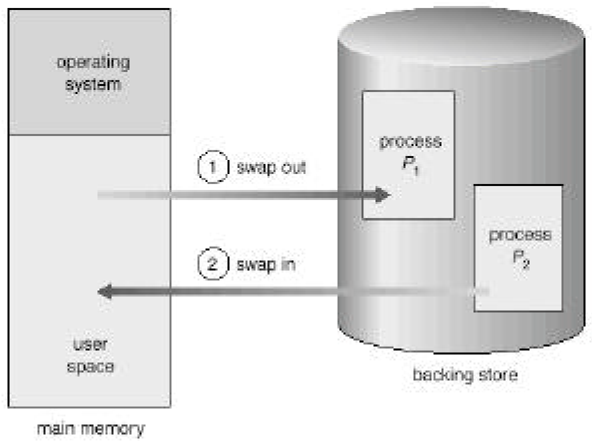
Logical Address Space versus Physical Address Space

- User programs only see the logical address space, in range 0 to *max*
- Physical memory operates in the physical address space, addresses in the range R+0 to R+*max*
- This distinction between logical and physical address spaces is a key one for memory management schemes.

Swapping

- What: temporarily move inactive process to *backing store* (e.g., fast disk). At some later time, return it to main memory for continued execution.
- Why: permit other processes to use memory resources (hence each process can be bigger)
- Who: decision of what process to swap made by medium-term scheduler

Schematic view of Swapping



Swapping

- Some possibilities of <u>when</u> to swap
  - if you have 3 processes, start to swap one out when its quantum expires while two is executing. Goal is to have third process in place when two's quantum expires (i.e., overlap computation with disk i/o)
  - context switch time is very high if you can't achieve this
- Another option: *roll out* lower priority process in favor of higher priority process. *Roll in* the lower priority process when the higher priority one finishes

Swapping

- If you have static address binding (i.e., compile or load time binding) have to swap process back into same memory space. *Why?*
- If you have execution-time address binding, then you can swap the process back into a different memory space.
- Disk is slow and the transfer time needed is proportional to the size of the process, so it is useful if processes can specify the parts of allocated memory that are unused to avoid having to transfer.

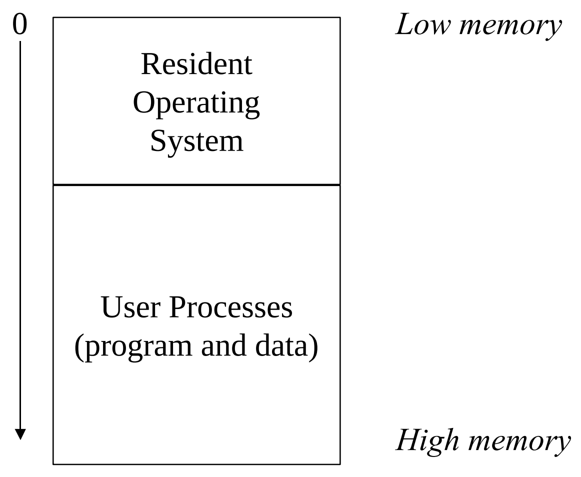
Swapping

- Process cannot be swapped until completely idle. Example of a problem: overlapped DMA input/output. (This requires that you have buffer space allocated in memory when the i/o request comes back)
- Note that in general swapping in this form (i.e., with this large sized granularity) is not very common now.

Contiguous Allocation

- Divide memory into *partitions*. Initially consider two partitions--one for the resident operating system and one for a user process.
- Where should the operating system go--low memory or high memory?
- Frequently put the operating system in low memory because this is where the interrupt vector is located. Also this permits the user partition to be expanded without running into the operating system (a factor when we have more than one partition or if we run the same binaries on different system configurations).

Memory Partitions



Single Partition Allocation

- Initial location of the user's process in memory is *not* 0
- The relocation register (base register) points to the first location in the user's partition. User's logical addresses are adjusted by the hardware to produce the physical address. (Address binding delayed until execution time.)
- Relocation register value is static during program execution, hence all of the OS must be present (it might be used). Otherwise have to relocate user code/data "on the fly"! In other words we cannot have transient OS code.

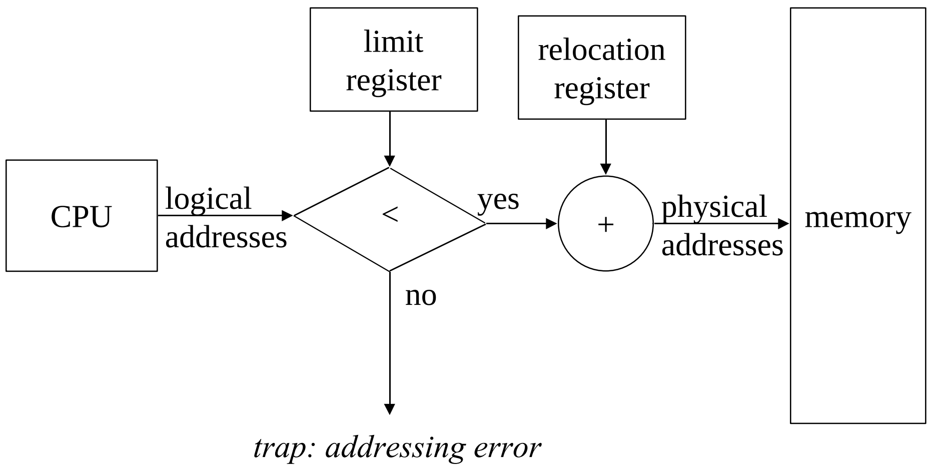
Single Partition Allocation

- How about memory references passed from the user process to the OS (for example, blocks of memory passed as an argument to a I/O routine)?
- The address must be translated from user's logical address space to the physical address space. Other arguments don't get translated (e.g., counts).
- Hence OS software has to handle these translations.

Limit Register

- How do we protect the OS from accidental or intentional interference from user processes?
- Add a *limit register* to the address mapping scheme

Limit Register



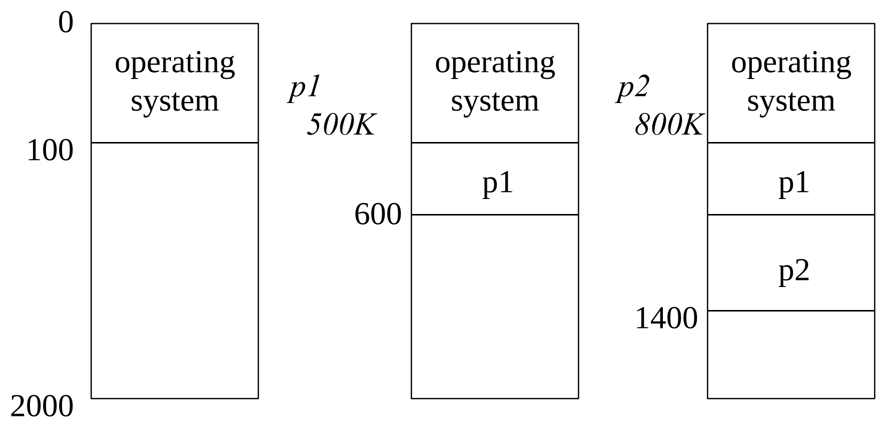
Multiple-Partition Allocation

- Goal: allocate memory to *multiple* processes (which permits rapid switches, for example)
- Simple scheme: fixed-size partition
  - memory divided into several partitions of fixed size
  - each partition holds one process
  - partition becomes free when process terminates; another process picked from the ready queue gets the free partition
  - number of partitions bounds the degree of multiprogramming
  - originally used in the IBM OS/360 operating system (MFT)
  - No longer used

Multiple-Partition Allocation Dynamic Partition

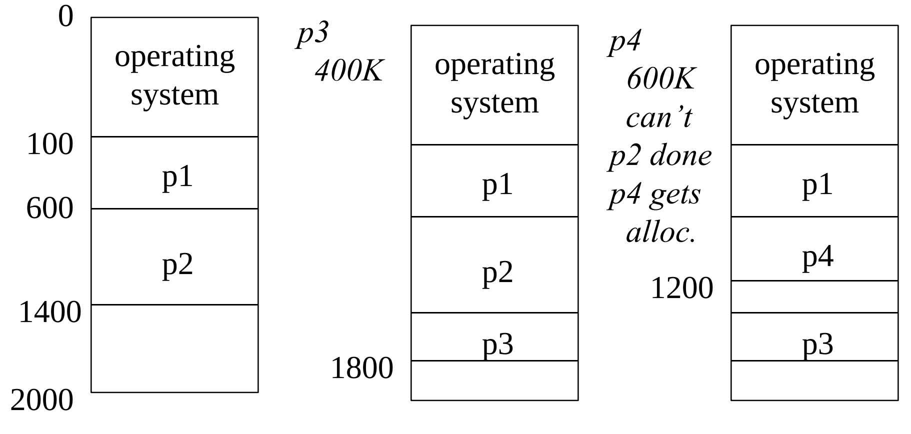
- Memory is partitioned dynamically
  - **Hole**: block of available memory
  - Holes of various size are scattered throughout memory
- Process still must occupy contiguous memory
- OS keeps a table listing which parts of memory are available
  - Allocated partitions
  - Free partitions (hole)
- When a process arrives, the OS searches for a part of memory that is large enough to hold the process. Allocates only the amount of needed memory.

Multiple-Partition Allocation Dynamic Partition



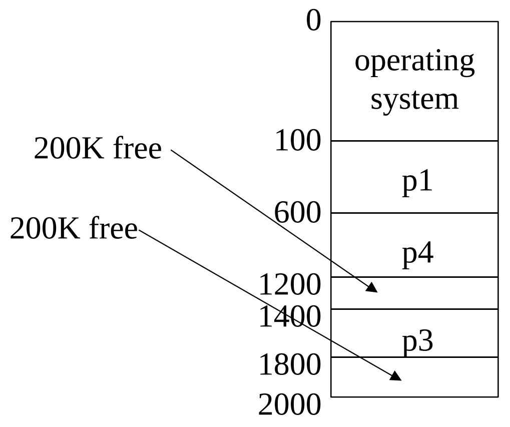
CPSC 410--Richard Furuta 2/17/00 33

Multiple-Partition Allocation Dynamic Partition



CPSC 410--Richard Furuta 2/17/00 34

Multiple-Partition Allocation Dynamic Partition



p5 requests 300 K but can’t obtain it since there is no large enough contiguous block free. Note that there *is* 400 K free in the system though...

Multiple-Partition Allocation Dynamic Partition

- This is an example of *external fragmentation*--sufficient amount of free memory to satisfy request but not in a contiguous block.
- We used a *first fit* algorithm this time to decide where to allocate space--what are some strategies for finding a free hole to fill?

Multiple-Partition Allocation Dynamic Partition

- *first fit* algorithm: allocate the *first* hole that is big enough. Searching can start either at beginning of set of holes or where the previous first-fit search ended. We quit when we find a free hole that is large enough.
- *best fit*: allocate the *smallest* hole that is big enough. Must search entire list to find it if you don't keep free list ordered by size.
- *worst fit*: allocate the *largest* hole. Again may need to search entire free list if not ordered. Produces the largest leftover hole, which may be less likely to create external fragmentation.

Multiple-Partition Allocation Dynamic Partition

- Simulation shows that first-fit and best-fit are better than worst-fit for time and storage use.
- First-fit is faster than best-fit
- First-fit and best-fit are similar in storage use.
- 50% rule--up to 1/3 of memory is lost to external fragmentation in first-fit (*N* allocated, 1/2 *N* lost)

Multiple-Partition Allocation Dynamic Partition

- General comments:
  - memory protection is necessary to prevent state interactions. This is effected by the limit register.
  - base registers are required to point to the current partition
- In general, blocks are allocated in some quantum (e.g., power or 2). No point in leaving space free if you can't address it or if it is too small to be of any use at all. Also there is an expense in keeping track of free space (free list; traversing list; etc.).
- This results in lost space--allocated but not required by process
- **Internal fragmentation**: difference between required memory and allocated memory.
- Internal fragmentation also results from estimation error and management overhead.

External Fragmentation

- External fragmentation can be controlled with compaction.
  - requires dynamic address binding (have to move pieces around)
  - can be quite expensive in time
  - some schemes try to control expense by only doing certain kinds of coalescing--e.g., on power of 2 boundary. (Topic of a data structures class.)
  - OS approach can also be to roll out/roll in all processes, returning processes to new addresses--no additional code required!

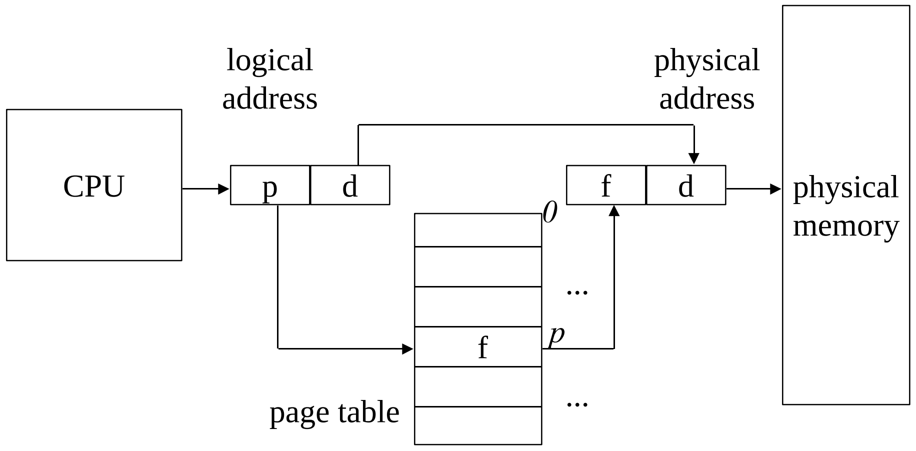
Paging

- Permit a process' memory to be non-contiguous
- Allocate physical memory in fixed-size, relatively small pieces, called **frames**
  - allocate frames to process as needed
  - avoids external fragmentation
  - causes increased internal fragmentation but attempts to minimize it through the small-sized frames. (Question: what is the characterization of the amount of internal fragmentation?)

Paging

- Implementation:
  - Divide (user) logical memory into **pages**. The page is the same size as the frame.
  - Dynamically map between pages and frames
  - Hardware assistance is required to do this mapping

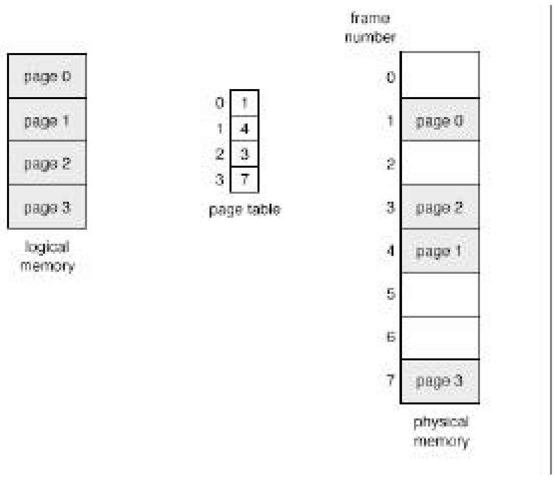
Paging
Hardware Assistance



Paging
Implementation

- Frames (and hence pages) are of fixed small size. Generally power of 2 large (why?).
- logical address of form (p d)
  - p, page number
  - d, offset within that page
- physical address of form (f d)
  - f, a frame number
  - d, offset within that frame
- Hardware maps from p to f; d copied across directly
- Q: how are f and d combined to get physical address?

Paging Example



CPSC 410--Richard Furuta 2/17/00 45

Frame Table

- How does the Operating System know what frames are in use in physical memory?
- Frame table
  - One entry per physical page frame
  - Indicates whether frame is free or allocated
  - If allocated, indicates to which page of which process or processes

CPSC 410--Richard Furuta 2/17/00 46

Paging
Fragmentation

- No external fragmentation since all frames/pages are the same size. Any page can be mapped to any frame.
- Internal fragmentation especially on last page of process. Average of 50% on last page.
- Smaller frames create less fragmentation but increase number of frames and size of page table.

Paging

- We still require that entire process fit into memory
- Each process has its own page table
- This means that process *cannot* address outside of its own address space
- Sharing frames (reentrant code) is possible
  - reentrant code--pure code, i.e., non self-modifying code that never changes during execution.
  - code frames can be shared among all processes
  - data separated out with one copy per process
  - process' page tables for code can be pointed to shared frame

Paging

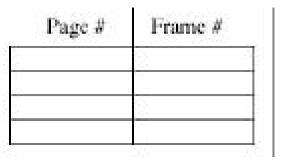
- Hardware support required (too slow to search)
  - simplest: put page table into dedicated high-speed registers
    - must keep page table reasonably small because of expense of registers (e.g., 256 entries)
    - but this severely limits the potential size of the process and increases expense of context switch!
  - Alternative: keep page table in memory with a page-table base register (*PTBR*) pointing to it
    - reduces context-switch time
    - doubles number of memory accesses

Paging

- Hardware support--continued
  - Associative registers, also called translation look-aside buffers (*TLB*s)
    - Associative registers: key and value
    - Keys are compared simultaneously and a corresponding value is returned
    - Fast but expensive. Size limited to between 8 and 2048 for example.
    - Use some strategy to decide which page table entries to put into associative memory.

Associative Registers

- Associative registers--parallel search
- Address translation (A’, A’’)
  - If A’ is in associative register, get frame #
  - Otherwise get frame # from page table in memory



Paging

- Hardware support (continued)
  - TLBs (continued)
    - on memory reference first check TLB. If match (*hit*) then use the value found
    - If no hit, then need to go to memory.
    - hit ratio: representation of how effective the process is.
    - What’s a strategy for loading the TLB? Perhaps cache entries on first access. Replace most recently used entry or use some form of rotation.

Effective Access Time

- Associative lookup = ε time units
- Assume that a memory cycle is 1 time unit
- Hit ratio = α
  - Percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Effective Access Time (EAT)

  EAT = $(1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha$

Memory protection

- Protection bits associated with frames; kept in page table
  - Read, write, read-only
  - Illegal operations result in hardware trap
- Valid/invalid bit
  - Illegal addresses, outside of process’ address space
  - Alternately: page-table length register (PTLR)

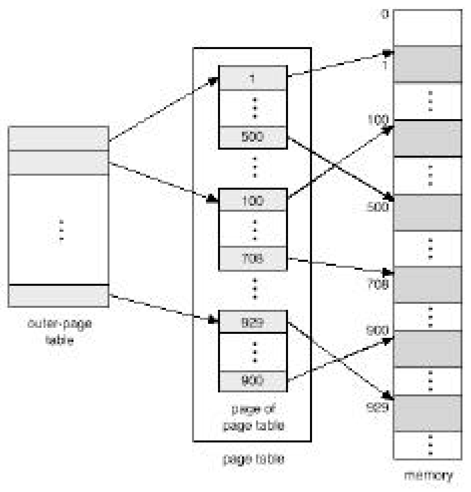
Paging

- Note that the OS also needs to keep track of which frames are being used--too expensive to search page tables to find a free frame
- Note that in these cases the logical address space is less than or equal to the physical address space in size. Later we will see the opposite case
- User's view is one contiguous space. Physical view is user's program scattered throughout physical memory.

Multilevel Paging

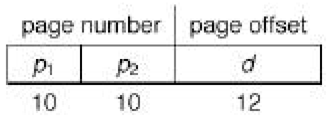
- Support a very large address space (say 2^{32} to 2^{64}) by using a two-level paging scheme.
- Here the page table itself is also paged.
- Otherwise the page table will become very large!
- Additional levels can also be introduced if necessary
- With caching, can decrease the requirement for additional memory references

Two-level Page table scheme



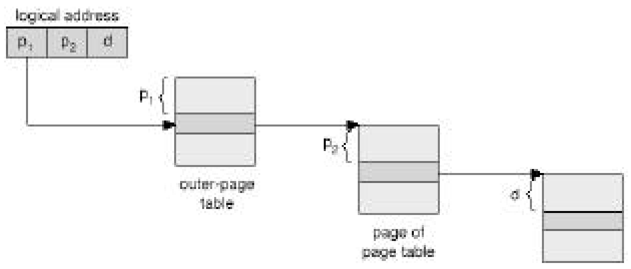
CPSC 410--Richard Furuta 2/17/00 57

Two-level paging example



32-bit machine with 4K page size

20 bit page number and 12 bit page offset

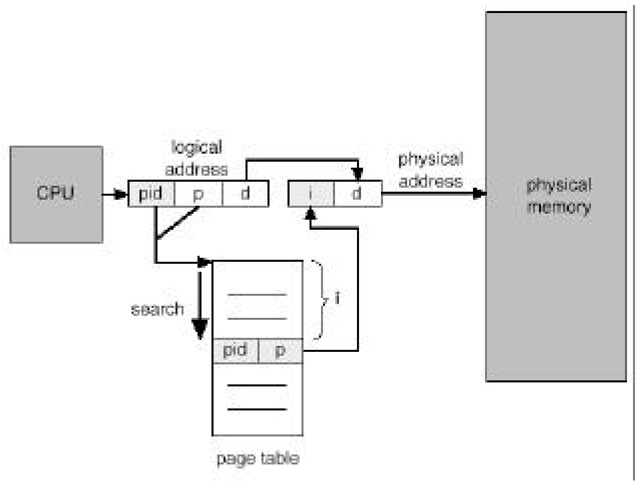


CPSC 410--Richard Furuta 2/17/00 58

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

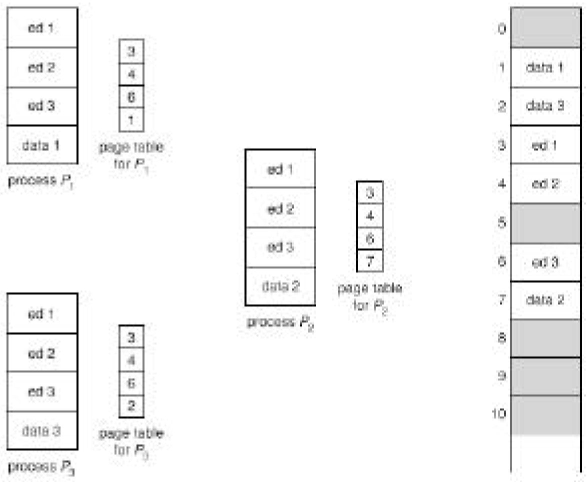
Inverted page table architecture



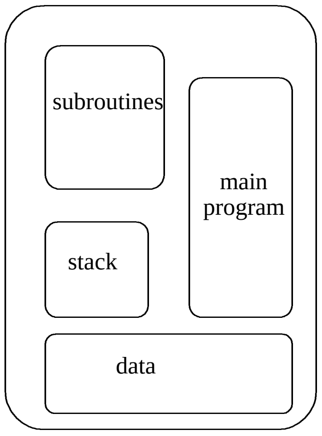
Shared pages

- 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

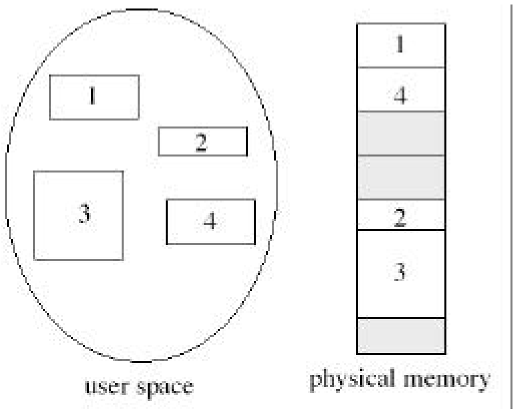


Segmentation



User's View of Memory

Segmentation



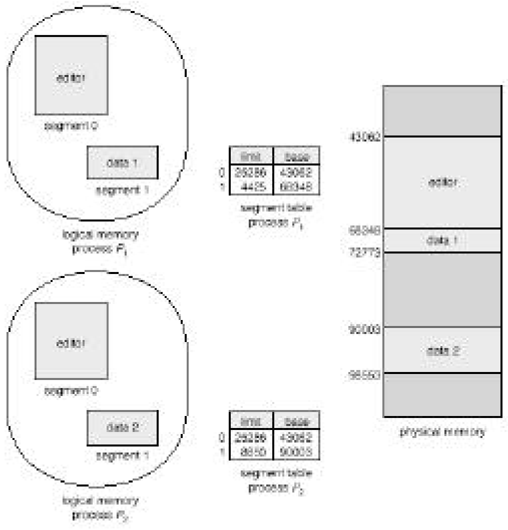
Segmentation

- Separating logical memory into different portions (segments) for different purposes
- Permits user’s partitioning of space to match the logical view
- Each segment has a name (e.g., a unique number) and a length.
- Logical address (from user’s point of view) is a segment number and an offset
- Physical address found by looking into a *segment table*

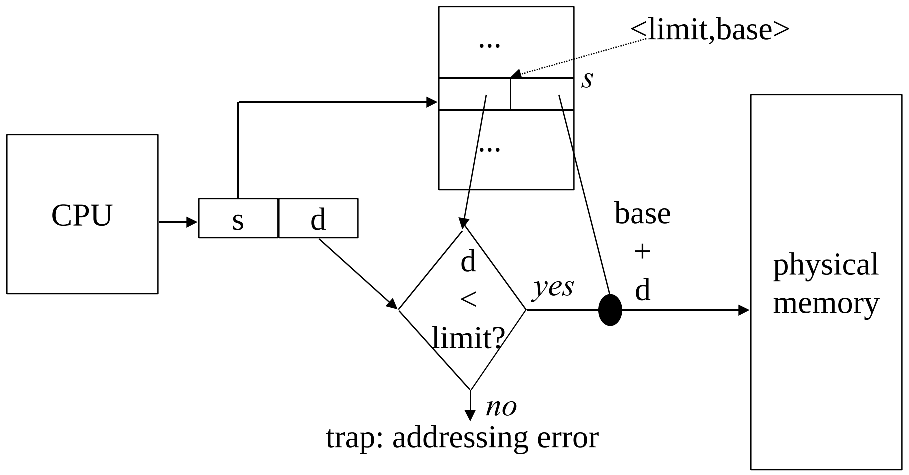
Segmentation segment table

- Segment table contains
  - limit
    - logical addresses must fall within the range 0..limit
    - error if it doesn’t (out of range)
    - provides memory access protection
  - base
    - added to offset to find physical address

Segmentation example



Segmentation Hardware



Segmentation

- Permits implementation of protections appropriate to segment's role
  - read-only for code
  - read/write for data
- But once again, since segments are variable size, we have the problem of external fragmentation.

Paged Segmentation

- Paging + Segmentation
- Aim: take the advantages of both paging and segmentation systems
- Implementation: treat each segment independently. Each segment has a page table.
- Logical address now consists of three parts
  - segment number
  - page number
  - offset

Paged Segmentation Implementation

- Segment table has base address for page table
- Look up <page number, offset> in page table
- Get physical address in return
- See examples in text for implementations in MULTICS and the Intel 386

Considerations in comparing memory-management strategies

- Hardware support
- Performance
- Fragmentation
- Relocation (when can we relocate)
- Swapping (and the cost)
- Sharing
- Protection