## Today

- Memory Management



## Review

- What does the MMU do?
  - Why is the MMU needed?
- How can we allocate memory to objects in the heap?



## Review: Memory Management

- Processor can only directly use data from registers
  - Need to move data closer (memory)
- Ideally, programmers want memory that is large, fast, & non volatile
- Memory hierarchy
  - Small amount of fast, expensive memory – cache
  - Some medium-speed, medium-price – main memory
  - Gigabytes of slow, cheap disk storage – swap/virtual memory
- Multiprogramming makes memory management trickier



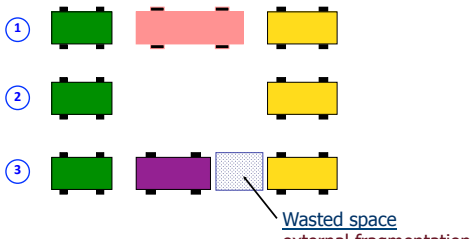
## Review: Memory Management Unit

- MMU = Hardware that maps virtual to physical addresses
- The user program deals with *logical* addresses
  - never sees the *real* physical addresses
- At runtime, relocation register added to every address generated by user process before being sent to memory



## Review: Variable Partitioning

Variable partitioning is the strategy of parking differently sized cars along a street with no marked parking space dividers.





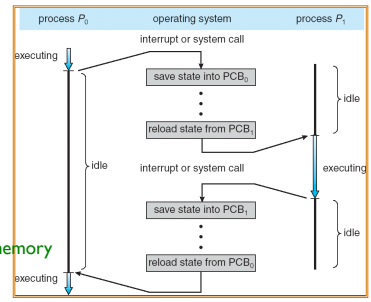

## Review: Dynamic Storage-Allocation Problem

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

- **First-fit**: Allocate the *first* hole that is big enough
- **Next-fit**: Allocate the *first* hole, but start at this position next time looking for a hole
- **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.

In general, sorting by hole size makes merging neighbors more expensive after memory is freed.

## CPU Switch From Process to Process




"state" = memory



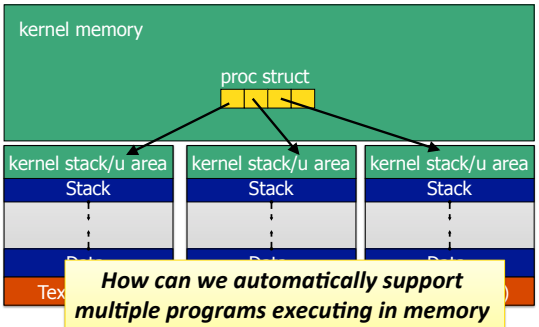
## Purpose of Memory

- Program must be brought into memory and placed within a process for it to be run
- Subdivide memory to accommodate multiple processes
- Memory needs to allocated efficiently to pack as many processes into memory as possible

***How do we manage this automatically?***



## Big Picture

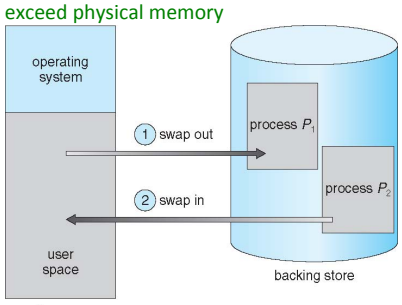



***How can we automatically support multiple programs executing in memory at the same time?***



## One Approach: Swapping

Problem: Total physical memory space of processes can exceed physical memory






## Swapping

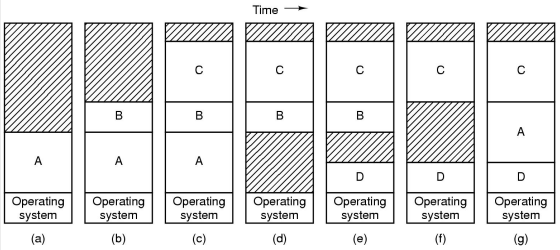
- A process can be
  1. swapped temporarily out of memory to a backing store
  2. then brought back into memory for continued execution
- Costs of swapping (may be part of context switch)
  - Swap time is dominated by transfer time
    - Total transfer time is directly proportional to the amount of memory swapped



## Swapping

- System maintains a ready queue of ready-to-run processes that have memory images on disk
- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method

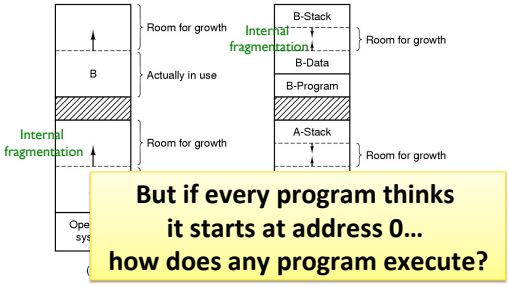
## Swapping




- Memory allocation changes as processes come into memory, leave memory
- Shaded regions are unused memory



## Swapping




**But if every program thinks it starts at address 0… how does any program execute?**

- Allocating space for growing data segment
- Allocating space for growing stack & data segment



## Fragmentation

- **Internal Fragmentation**
  - unused memory in a partition
- **External Fragmentation**
  - small holes in memory between allocated partitions
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Expensive, not often performed in practice
    - Not always possible; only if relocation is dynamic
  - Instead, use different allocation scheme to avoid



## Address Relocation

- When program loaded, absolute memory locations assigned
- A process may occupy different ***partitions***
  - Thus, different ***absolute*** memory locations during execution
- Separate logical/virtual and physical addresses
  - OS manages and translates physical
  - User programs deal with logical/relative

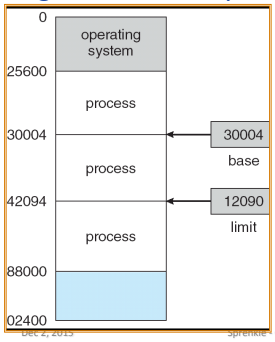


## MMU: Relocation and Protection

- Cannot be sure where program will be loaded in memory
  - Address locations of variables and code routines cannot be absolute
  - Must keep a program out of other processes' partitions
- Use *base* and ***limit*** values
  - Address locations added to base value to map to physical addr
  - Address locations larger than limit value is an error

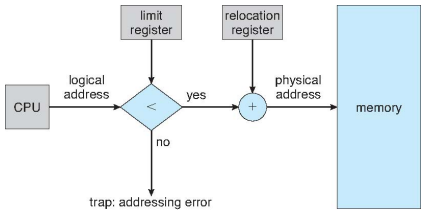


## Base and Limit Registers Define Logical Address Space




- Base/relocation register
  - Starting address for the process
- Limit register
  - Ending location of the process
- These values are set when the process is loaded and when the process is swapped in

## Hardware Support for Relocation and Limit Registers






## Other Memory Allocation Schemes

- Contiguous:
  - Partitioning - Fixed & Dynamic sizes
  - With or without swapping
- Non-contiguous
  - Simple paging
  - Simple segmentation
- Virtual Memory
  - Segmentation and/or Paging



Partitioning

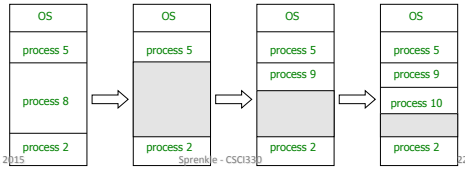
## CONTIGUOUS ALLOCATION



## Contiguous Allocation: Partitions

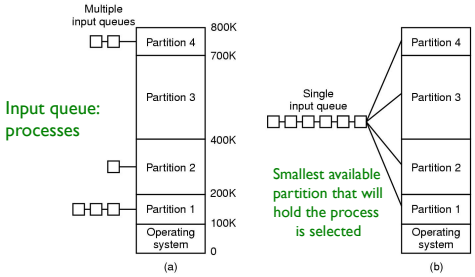
- Multiple-partition allocation
  - *Hole* – block of available memory
    - holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about: a) allocated partitions b)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

***Problem?***






## Partitions: Process Queues




Given either fixed or dynamic memory partitions, and either separate input queues for each partition or single input queue, **which will cause less fragmentation?**

## Placement Algorithm with Partitions

- Equal-size partitions (fixed)
  - because all partitions are of equal size, it does not matter which partition is used
- Unequal-size partitions (dynamic)
  - Competing goals:
    - Minimize waste (fragmentation)
    - Minimize overhead (time)
  - Dynamic storage allocation problem…

## Dynamic Storage Allocation Problem

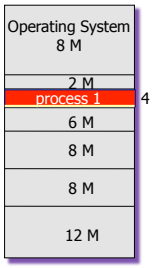
**How to satisfy a request of size *n* from a list of free holes**

- **First-fit**: Allocate the *first* hole that is big enough
- **Next-fit**: Allocate the *first* hole, but start at this position next time looking for a hole
- **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.



## Fixed Partitioning - Unequal Sizes

- Reduces internal fragmentation




Choosing partition sizes?
→ based on expected workload

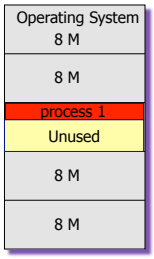


## Fixed Partitioning - Equal Size

- Problem: Main memory use is inefficient.
  - Internal Fragmentation - Part of partition unused

*Fixed, equal-sized partitions may not work well for an* **entire** *process.*

*How can we make them better?*





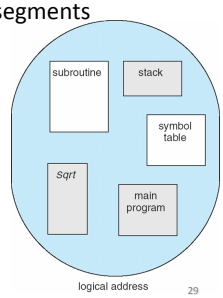

Segmentation

## NON-CONTIGUOUS ALLOCATION



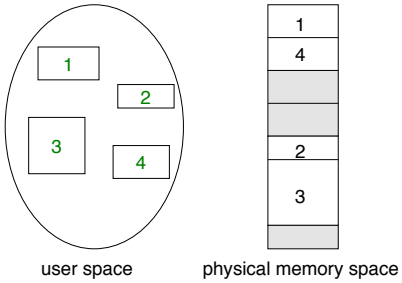
## Segmentation

- Supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit:
    - main program
    - Procedure
    - Object
    - …
  - Each has separate purpose






## Logical View of Segmentation




Since segments vary in length,
memory allocation is a dynamic storage allocation problem

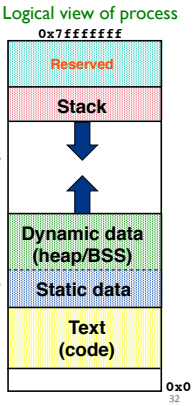
## Segmentation Architecture

- Logical address is a tuple:
  <segment-number, offset>,
- Segment table
  - maps two-dimensional physical addresses
  - each table entry has:
    - Segment-table **base** register (STBR) contains the starting physical address where the segments reside in memory
    - Segment-table length register (STLR) specifies the length of the segment
      - Acts as the **limit**



## VAS example (32-bit)

- The program uses virtual memory through its process' Virtual Address Space:
  - An addressable array of bytes…
  - Containing every instruction the process thread can execute…
  - And every piece of data those instructions can read/write…
    - i.e., read/write == load/store on memory
- **Partitioned into logical segments (regions) with distinct purpose and use.**
  - Every memory reference by a thread is interpreted in the context of its VAS.
    - Resolves to a location in machine memory






Paging

## NON-CONTIGUOUS ALLOCATION



## Paging Guiding Principle

- Idea: process is allocated physical memory whenever available
  - Avoids problem of variable-sized memory chunks
  - Avoids external fragmentation
    - Internal fragmentation only



## Paging

- Partition memory into small equal-size chunks
  - Chunks of memory are called ***frames***
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide each process into the same size chunks
  - Chunks of a process are called ***pages***
- Operating system maintains a ***page table*** for each process
  - contains the frame location for each process page
  - memory address = page number + offset

**Challenge: relocation**



## Using Pages

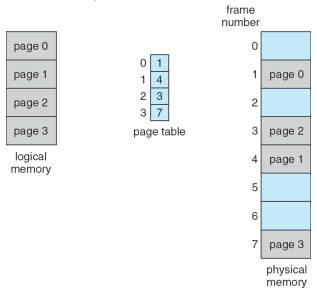
- Keep track of all free frames
- To run a program of size *N* pages, need to find *N* free frames and load program
- Set up a page table to translate logical to physical addresses
- Backing store likewise split into pages

## Address Translation Scheme

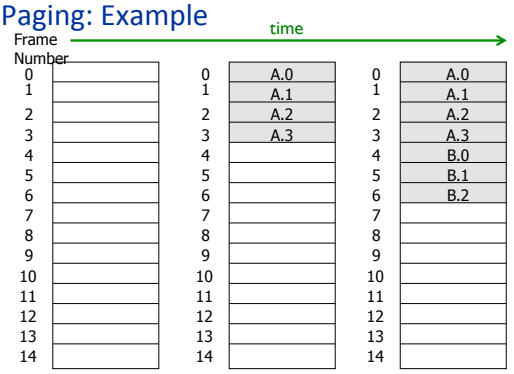
- Address generated by CPU is divided into:
  - ***Page number (p)*** – used as an index into a *page table* which contains base address of each page in physical memory
  - ***Page offset (d)*** – combined with base address to define the physical memory address that is sent to the memory unit



## Paging Model of Logical and Physical Memory

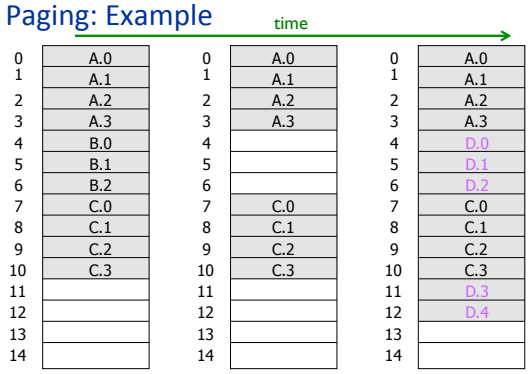



## Paging: Example



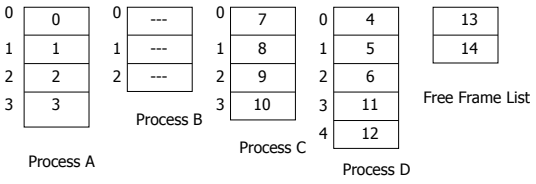



## Paging: Example






## Page Tables for Example




Within each process, the memory allocation is **not** necessarily in page order:

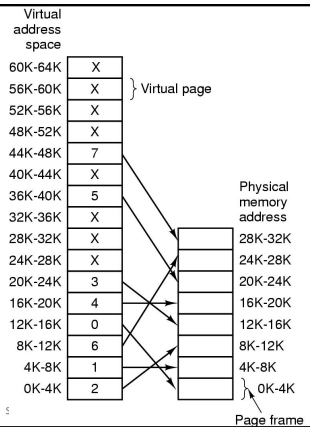
| 1 |
|---|
| 5 |
| 2 |
| 6 |



## Example Page Table

The relation between virtual addresses and physical memory addresses given by page table

Every memory access requires **2** lookups: one in the page table and one in memory



## TLBs – 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 |

- TLB: special, small, fast-lookup hardware cache
  - used to speed up paging



## Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- Valid-invalid bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process's logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use page-table length register (PTLR)
- Any violations result in a trap to the kernel

## Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page

Which page should we remove next?

- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon



## Optimal Page Replacement Algorithm

- Replace page needed at the farthest point in future
  - Optimal but unrealizable
- Estimate by …
  - logging page use on previous runs of process
  - although this is impractical



## Not Recently Used Page Replacement Algorithm

- Each page has Reference bit, Modified bit
  - bits are set when page is referenced, modified
- Pages are classified
  1. not referenced, not modified
  2. not referenced, modified
  3. referenced, not modified
  4. referenced, modified
- NRU removes page at random
  - from lowest-numbered, non-empty class



## TODO

- Project 5 due last day of class