Dynamic Domain Allocation

- A concept covered in a previous lecture
- We'll just review it here
- Domains are regions of memory made available to a process
  - Variable sized, usually any size requested
  - Each domain is contiguous in memory addresses
  - Domains have access permissions for the process
  - Potentially shared between processes
- Each process could have multiple domains
  - With different sizes and characteristics

Problems With Domains

- Not relocatable
  - Once a process has a domain, you can't easily move its contents elsewhere
- Not easily expandable
- Impossible to support applications with larger address spaces than physical memory
  - Also can't support several applications whose total needs are greater than physical memory
- Also subject to fragmentation

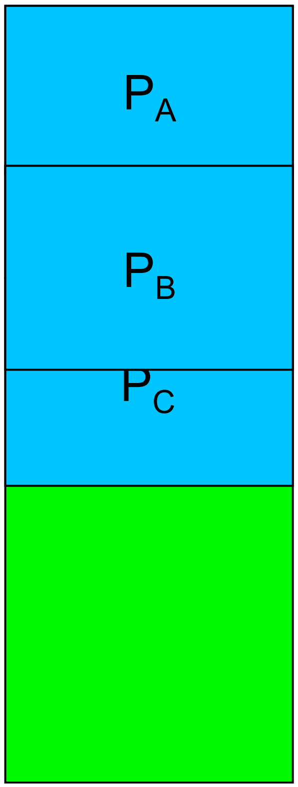
Relocation and Expansion

- Domains are tied to particular address ranges
  - At least during an execution
- Can't just move the contents of a domain to another set of addresses
  - All the pointers in the contents will be wrong
  - And generally you don't know which memory locations contain pointers
- Hard to expand because there may not be space "nearby"

The Expansion Problem

- Domains are allocated on request
- Processes may ask for new ones later
- But domains that have been given are fixed
  - Can't be moved somewhere else in memory
- Memory management system might have allocated all the space after a given domain
- In which case, it can't be expanded

Illustrating the Problem



Now Process B wants to expand its domain size

But if we do that, Process B steps on Process C's memory

We can't move C's domain out of the way

And we can't move B's domain to a free area

We're stuck, and must deny an expansion request that we have enough memory to handle

Address Spaces Bigger Than Physical Memory

- If a process needs that much memory, how could you possibly support it?
- Two possibilities:
  1. It's not going to use all the memory it's asked for, or at least not all simultaneously
  2. Maybe we can use something other than physical memory to store some of it
- Domains are not friendly to either option

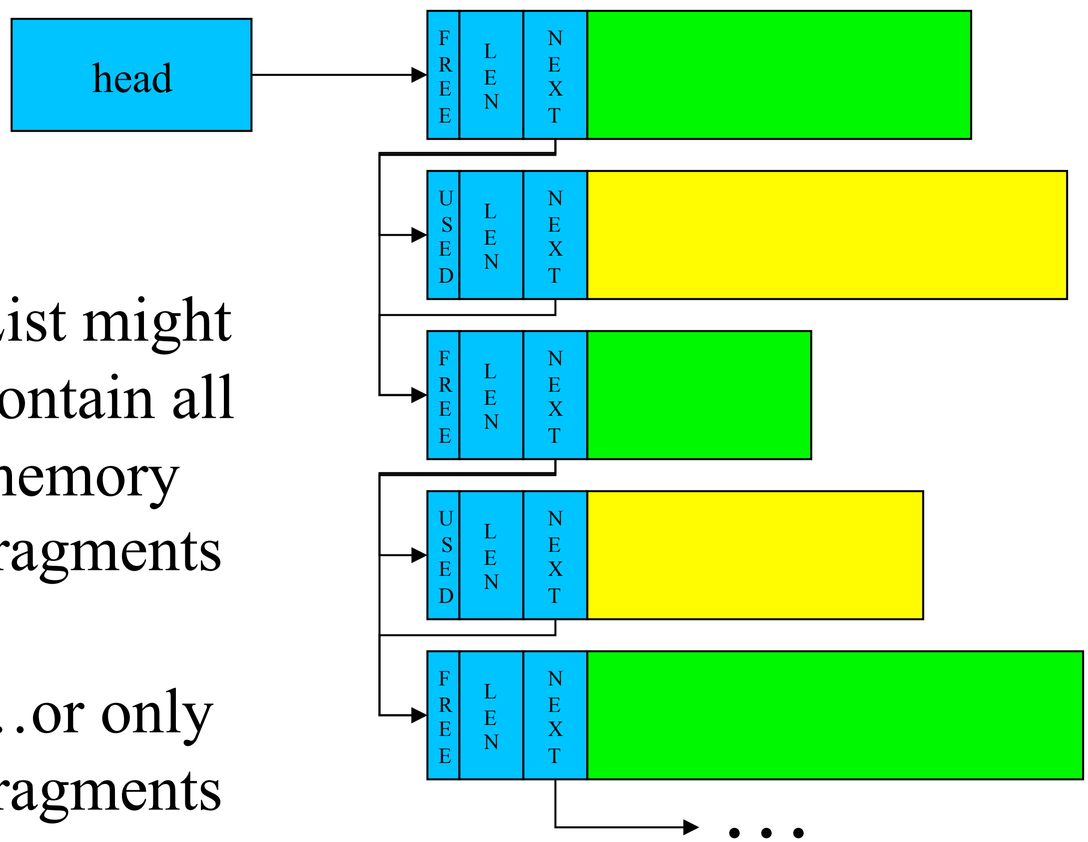
How To Keep Track of Variable Sized Domains?

- Start with one large “heap” of memory
- Maintain a *free list*
  - Systems data structure to keep track of pieces of unallocated memory
- When a process requests more memory:
  - Find a large enough chunk of memory
  - Carve off a piece of the requested size
  - Put the remainder back on a *free list*
- When a process frees memory
  - Put it back on the free list

Managing the Free List

- Fixed sized blocks are easy to track
  - A bit map indicating which blocks are free
- Variable chunks require more information
  - A linked list of descriptors, one per chunk
  - Each descriptor lists the size of the chunk and whether it is free
  - Each has a pointer to the next chunk on list
  - Descriptors often kept at front of each chunk
- Allocated memory may have descriptors too

The Free List



List might contain all memory fragments

…or only fragments that are free

Free Chunk Carving

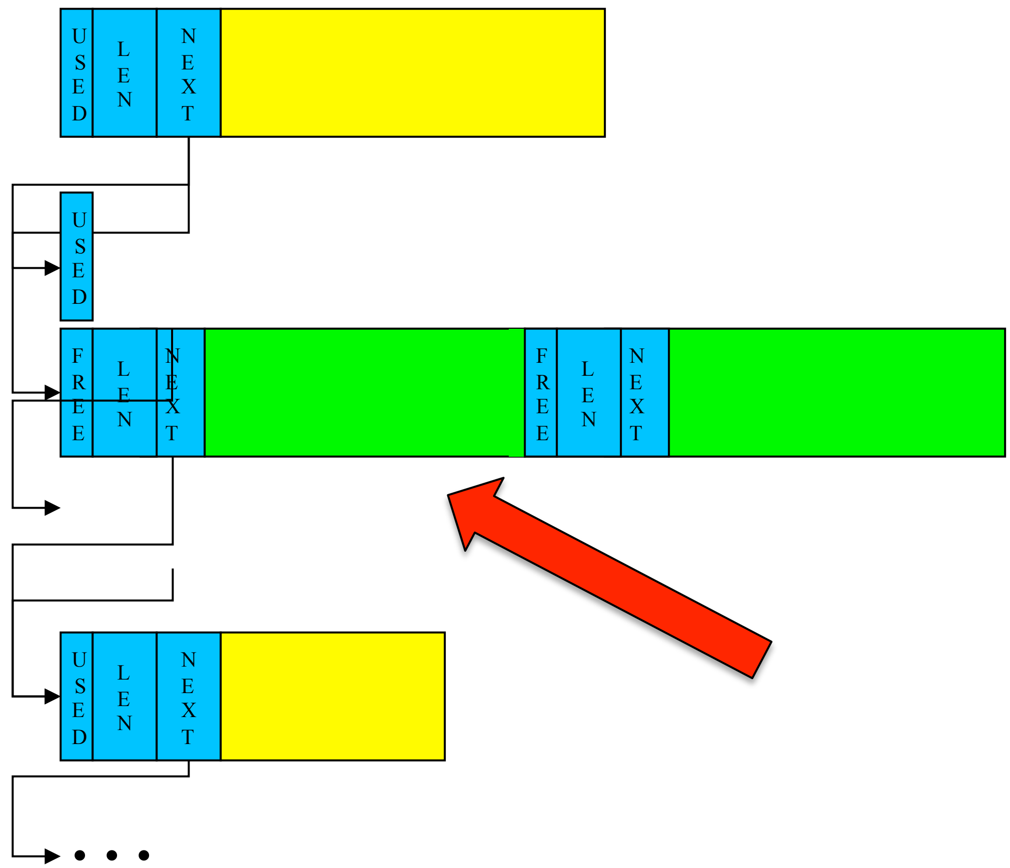
1. Find a large enough free chunk

2. Reduce its len to requested size

3.Create a new header for residual chunk

4. Insert the new chunk into the list

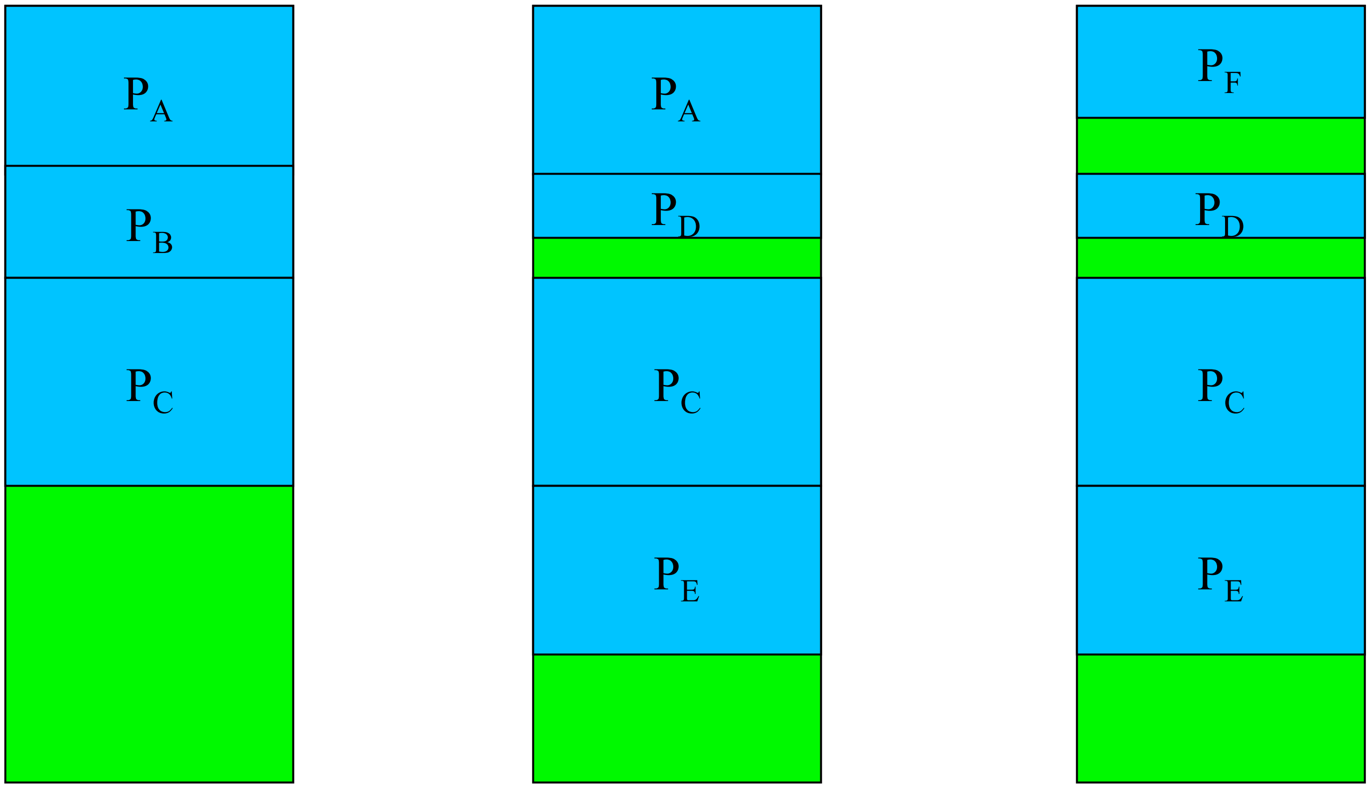
5. Mark the carved piece as in use



Variable Domain and Fragmentation

- Variable sized domains not as subject to internal fragmentation
  - Unless requestor asked for more than he will use
  - Which is actually pretty common
  - But at least memory manager gave him no more than he requested
- Unlike fixed sized partitions, though, subject to another kind of fragmentation
  - *External fragmentation*

External Fragmentation



We gradually build up small, unusable memory chunks scattered through memory

External Fragmentation: Causes and Effects

- Each allocation creates left-over chunks
  – Over time they become smaller and smaller
- The small left-over fragments are useless
  – They are too small to satisfy any request
  – A second form of fragmentation waste
- Solutions:
  – Try not to create tiny fragments
  – Try to recombine fragments into big chunks

How To Avoid Creating Small Fragments?

- Be smart about which free chunk of memory you use to satisfy a request
- But being smart costs time
- Some choices:
  - Best fit
  - Worst fit
  - First fit
  - Next fit

Best Fit

- Search for the "best fit" chunk
  - Smallest size greater than or equal to requested size
- Advantages:
  - Might find a perfect fit
- Disadvantages:
  - Have to search entire list every time
  - Quickly creates very small fragments

Worst Fit

- Search for the "worst fit" chunk
  - Largest size greater than or equal to requested size
- Advantages:
  - Tends to create very large fragments

    … for a while at least
- Disadvantages:
  - Still have to search entire list every time

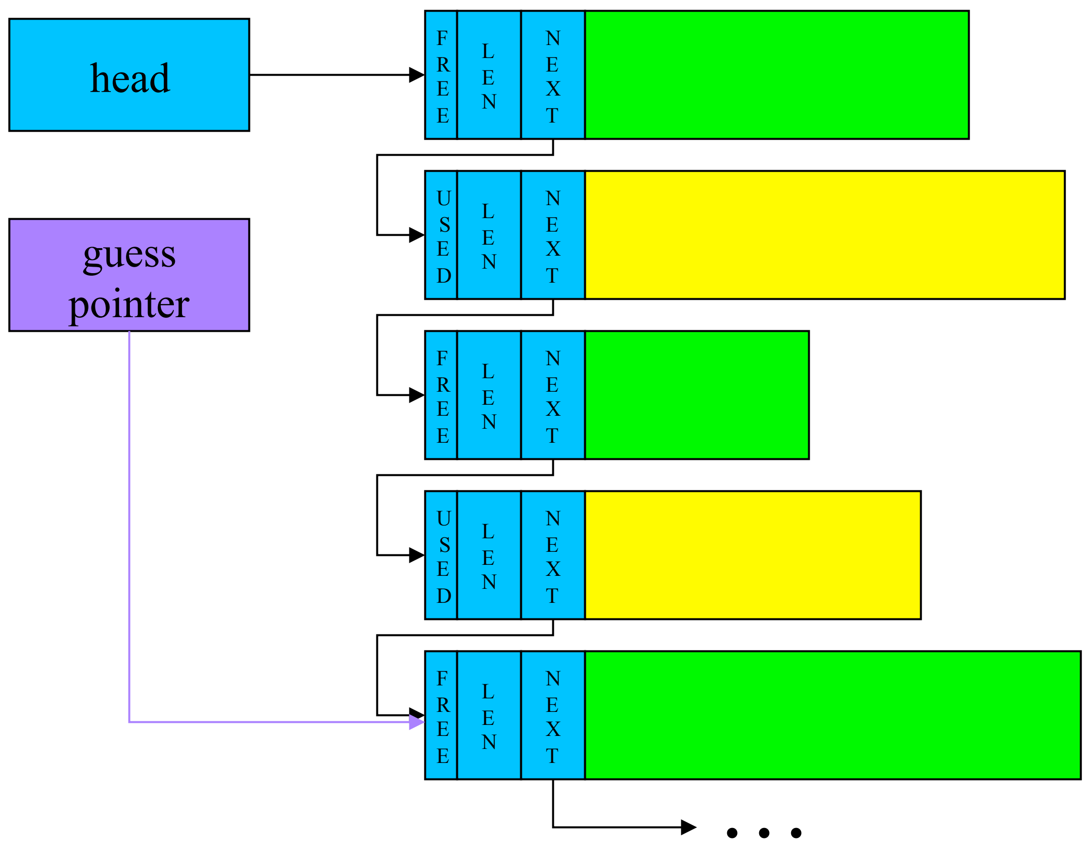
First Fit

- Take first chunk you find that is big enough
- Advantages:
  - Very short searches
  - Creates random sized fragments
- Disadvantages:
  - The first chunks quickly fragment
  - Searches become longer
  - Ultimately it fragments as badly as best fit

Next Fit

After each search, set guess pointer to chunk after the one we chose.

That is the point at which we will begin our next search.



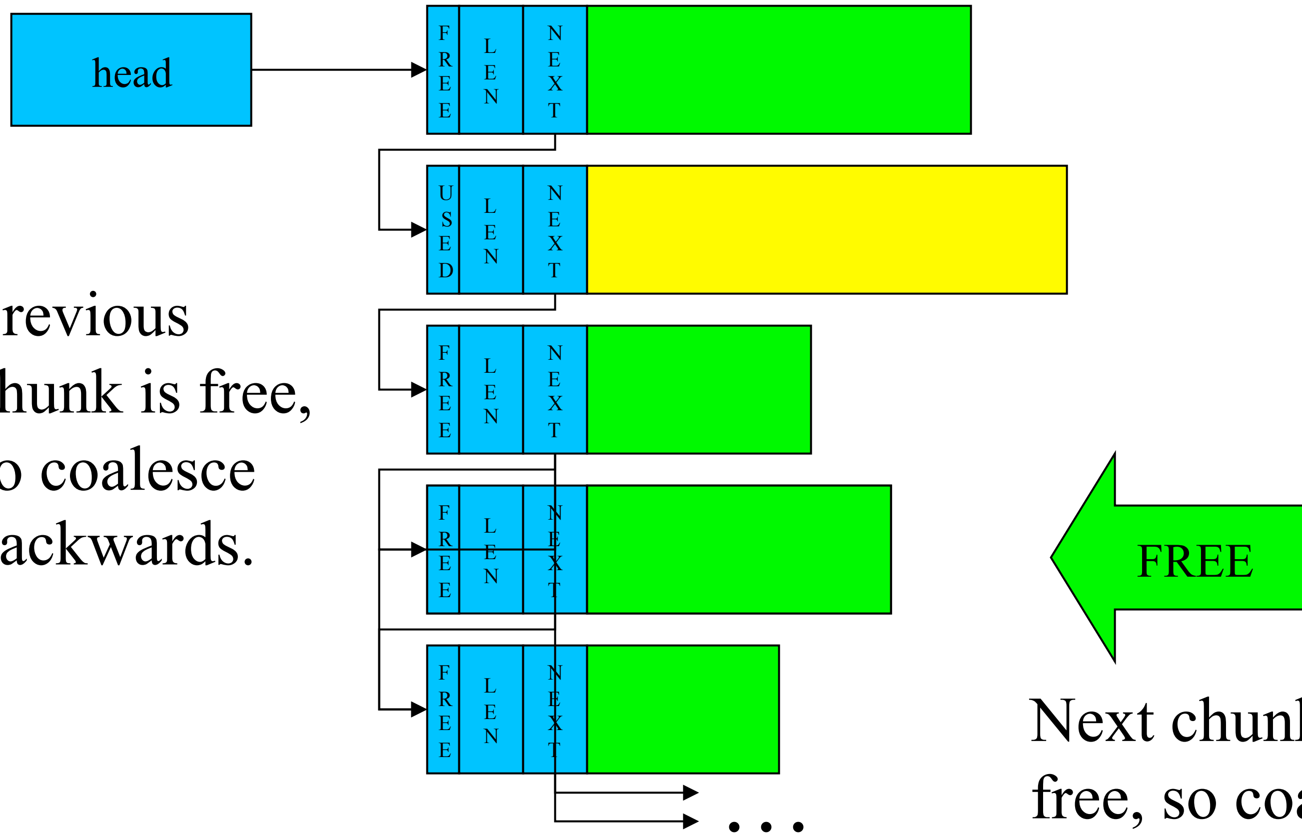
Next Fit Properties

- Some advantages of each approach
  - Short searches (maybe shorter than first fit)
  - Spreads out fragmentation (like worst fit)
- But more fragmentation than best fit
- Guess pointers are a general technique
  - Think of them as a lazy (non-coherent) cache
  - If they are right, they save a lot of time
  - If they are wrong, the algorithm still works
  - They can be used in a wide range of problems

Coalescing Domains

- All variable sized domain allocation algorithms have external fragmentation
  - Some get it faster, some spread it out
- We need a way to reassemble fragments
  - Check neighbors whenever a chunk is freed
  - Recombine free neighbors whenever possible
  - Free list can be designed to make this easier
    - E.g., where are the neighbors of this chunk?
- Counters forces of external fragmentation

Free Chunk Coalescing



Previous chunk is free, so coalesce backwards.

Next chunk is also free, so coalesce forwards.

Fragmentation and Coalescing

- Opposing processes that operate in parallel
  – Which of the two processes will dominate?
- What fraction of space is typically allocated?
  – Coalescing works better with more free space
- How fast is allocated memory turned over?
  – Chunks held for long time cannot be coalesced
- How variable are requested chunk sizes?
  – High variability increases fragmentation rate
- How long will the program execute?
  – Fragmentation, like rust, gets worse with time

Coalescing and Free List Implementation

- To coalesce, we must know whether the previous and next chunks are also free
- If the neighbors are guaranteed to be in the free list, we can look at them and see if they are free
- If allocated chunks are not in the free list, we must look at the free chunks before and after us
  - And see if they are our contiguous neighbors
  - This suggests that the free list must be maintained in address order

Variable Sized Domain Summary

- Eliminates internal fragmentation
  - Each chunk is custom made for requestor
- Implementation is more expensive
  - Long searches of complex free lists
  - Carving and coalescing
- External fragmentation is inevitable
  - Coalescing can counteract the fragmentation
- Must we choose the lesser of two evils?