Operating System Principles: Memory Management

CS 111

Operating Systems

Peter Reiher

Outline

- What is memory management about?
- Memory management strategies:
  - Fixed partition strategies
  - Dynamic partitions
  - Buffer pools
  - Garbage collection
  - Memory compaction

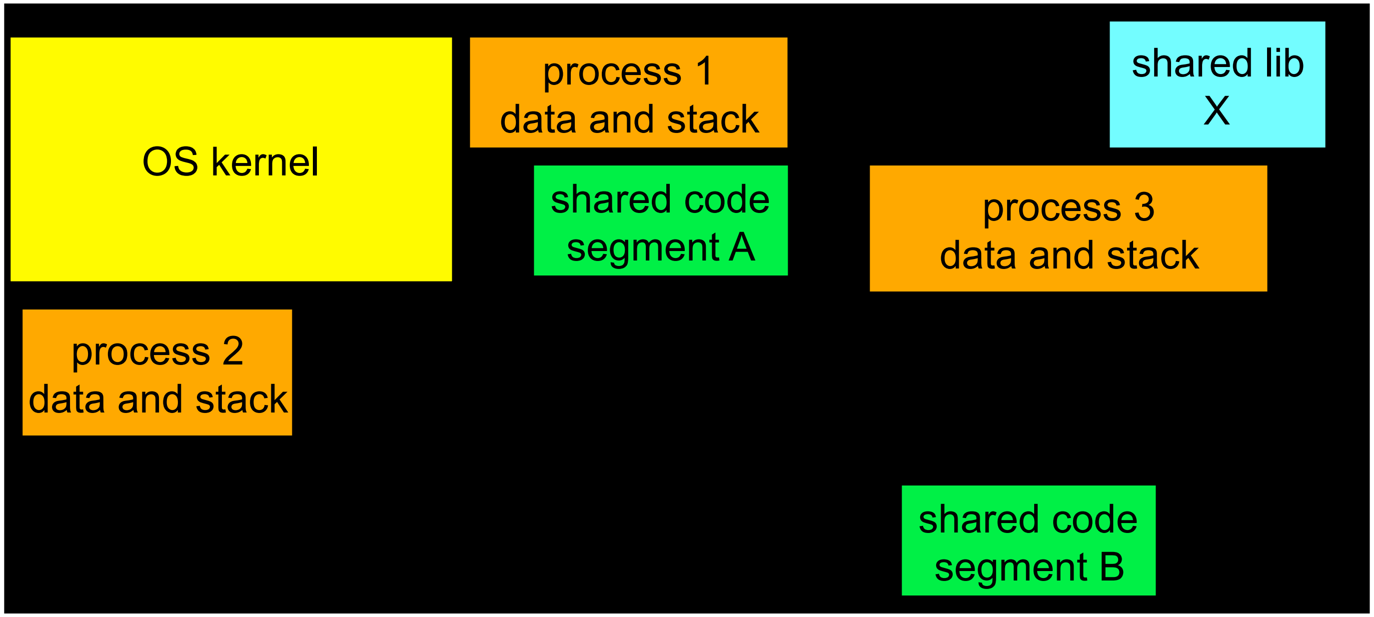
Memory Management

- Memory is one of the key assets used in computing
- In particular, memory abstractions that are usable from a running program
  - Which, in modern machines, typically means RAM
- We have a limited amount of it
- Lots of processes need to use it
- How do we manage it?

Memory Management Goals

1. Transparency
   - Process sees only its own address space
   - Process is unaware memory is being shared
2. Efficiency
   - High effective memory utilization
   - Low run-time cost for allocation/relocation
3. Protection and isolation
   - Private data will not be corrupted
   - Private data cannot be seen by other processes

Physical Memory Allocation

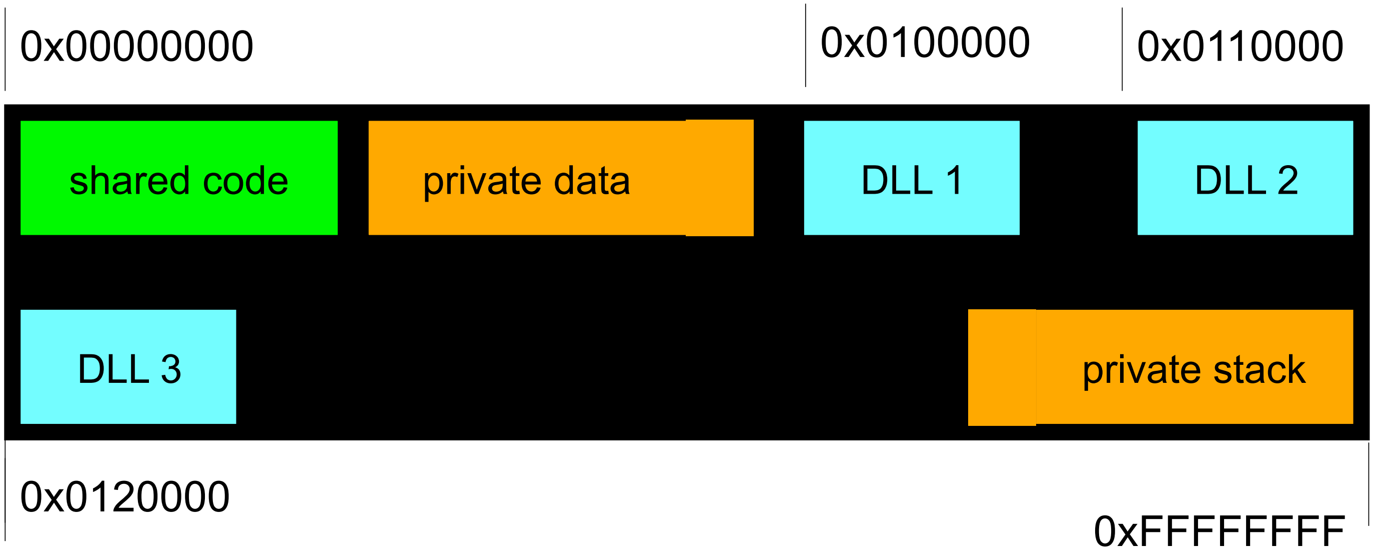


Physical memory is divided between the OS kernel, process private data, and shared code segments.

Physical and Virtual Addresses

- A cell of RAM has a particular physical address
- Years ago, that address was used by processes to name RAM locations
- Instead, we can have processes use virtual addresses
  - Which may not be the same as physical addresses
- More flexibility in memory management, but requires virtual to physical translation

A Linux Process' Virtual Address Space



All of these segments appear to be present in memory whenever the process runs.

Note this virtual address space contains no OS or other process segments

Aspects of the Memory Management Problem

- Most processes can’t perfectly predict how much memory they will use
- The processes expect to find their existing data when they need it where they left it
- The entire amount of data required by all processes may exceed amount of available physical memory
- Switching between processes must be fast
  - Can’t afford much delay for copying data
- The cost of memory management itself must not be too high

Memory Management Strategies

- Fixed partition allocations
- Dynamic partitions
- Relocation

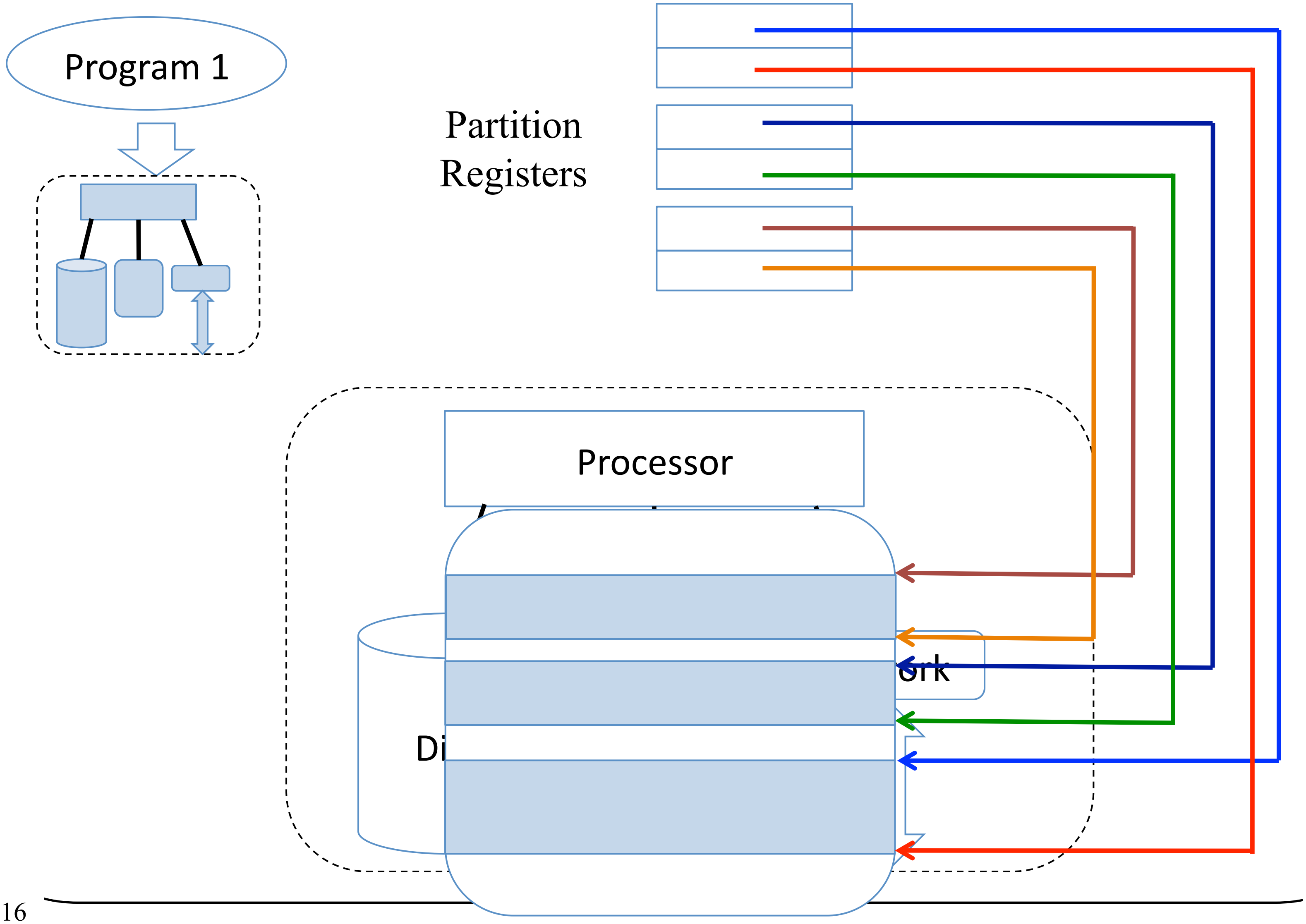
Fixed Partition Allocation

- Pre-allocate partitions for *n* processes
  - One or more per process
  - Reserving space for largest possible process
- Partitions come in one or a few set sizes
- Very easy to implement
  - Common in old batch processing systems
  - Allocation/deallocation very cheap and easy
- Well suited to well-known job mix

Memory Protection and Fixed Partitions

- Need to enforce partition boundaries
  - To prevent one process from accessing another's memory
- Could use hardware for this purpose
  - Special registers that contain the partition boundaries
  - Only accept addresses within the register values
- Basic scheme doesn't use virtual addresses

The Partition Concept



Problems With Fixed Partition Allocation

- Presumes you know how much memory will be used ahead of time
- Limits the number of processes supported to the total of their memory requirements
- Not great for sharing memory
- *Fragmentation* causes inefficient memory use

Fragmentation

- A problem for all memory management systems
  - Fixed partitions suffer it especially badly
- Based on processes not using all the memory they requested
- As a result, you can't provide memory for as many processes as you theoretically could

Fragmentation Example

Let’s say there are three processes, A, B, and C

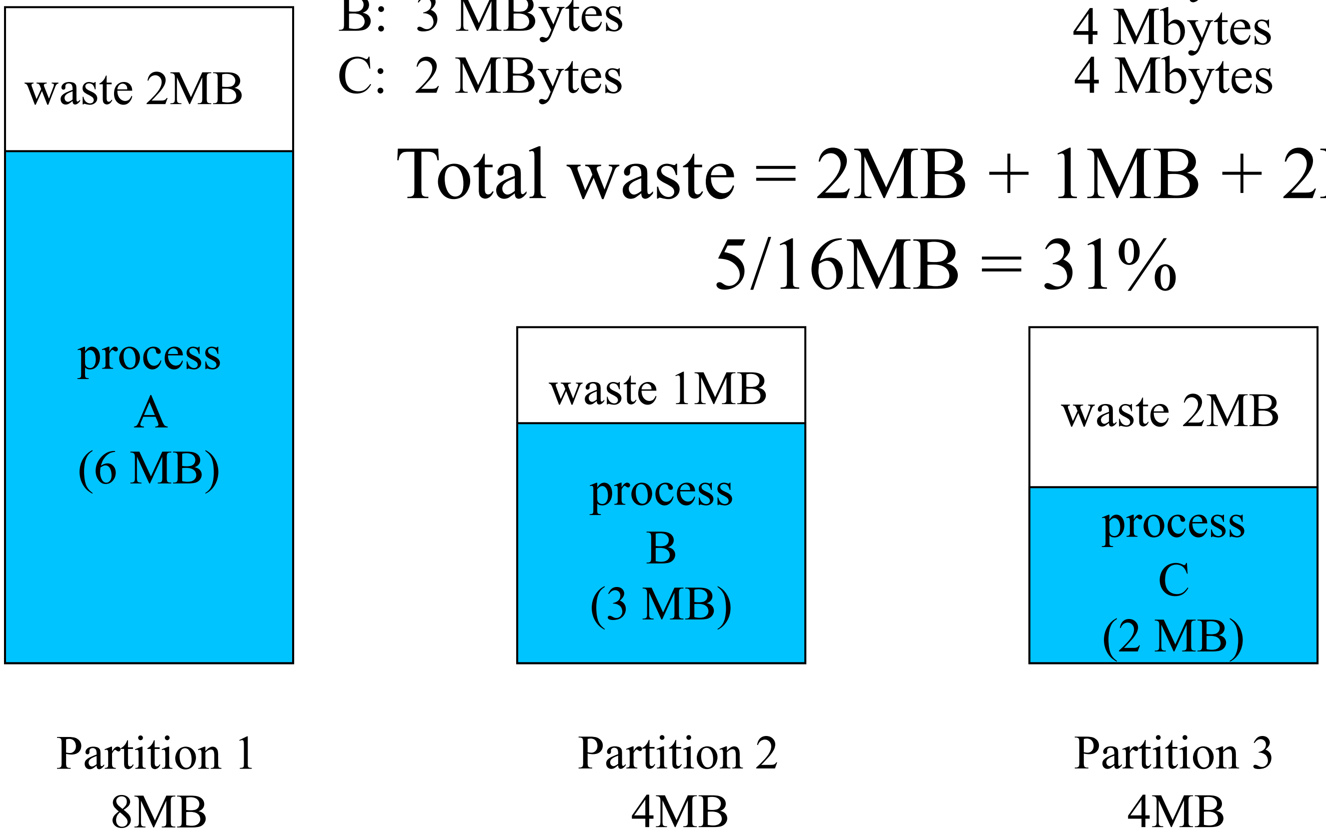
Their memory requirements:

A: 6 MBytes
B: 3 MBytes
C: 2 MBytes

Available partition sizes:

8 Mbytes
4 Mbytes
4 Mbytes

Total waste = 2MB + 1MB + 2MB = 5/16MB = 31%



waste 2MB

process
A
(6 MB)

Partition 1
8MB

waste 1MB

process
B
(3 MB)

Partition 2
4MB

waste 2MB

process
C
(2 MB)

Partition 3
4MB

Internal Fragmentation

- Fragmentation comes in two kinds:
  - Internal and external
- This is an example of *internal fragmentation*
  - We'll see external fragmentation later
- Wasted space in fixed sized blocks
  - The requestor was given more than he needed
  - The unused part is wasted and can't be used for others
- Internal fragmentation can occur whenever you force allocation in fixed-sized chunks

More on Internal Fragmentation

- Internal fragmentation is caused by a mismatch between
  - The chosen sizes of a fixed-sized blocks
  - The actual sizes that programs use
- Average waste: 50% of each block
- Overall waste reduced by multiple sizes
  - Suppose blocks come in sizes S1 and S2
  - Average waste = ((S1/2) + (S2 - S1)/2)/2

Summary of Fixed Partition Allocation

- Very simple
- Inflexible
- Subject to a lot of internal fragmentation
- Not used in many modern systems
  - But a possible option for special purpose systems, like embedded systems
  - Where we know exactly what our memory needs will be

Dynamic Partition Allocation

- Like fixed partitions, except
  - Variable sized, usually any size requested
  - Each partition is contiguous in memory addresses
  - Partitions have access permissions for the process
  - Potentially shared between processes
- Each process could have multiple partitions
  - With different sizes and characteristics

Problems With Dynamic Partitions

- Not relocatable
  - Once a process has a partition, 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

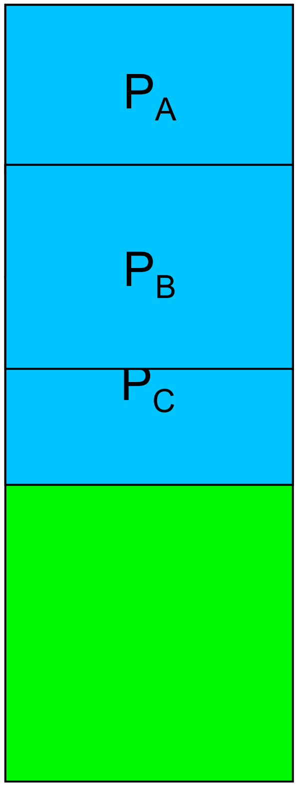
Relocation and Expansion

- Partitions are tied to particular address ranges
  - At least during an execution
- Can’t just move the contents of a partition 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

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

Illustrating the Problem



Now Process B wants to expand its partition size

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

We can’t move C’s partition out of the way

And we can’t move B’s partition to a free area

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

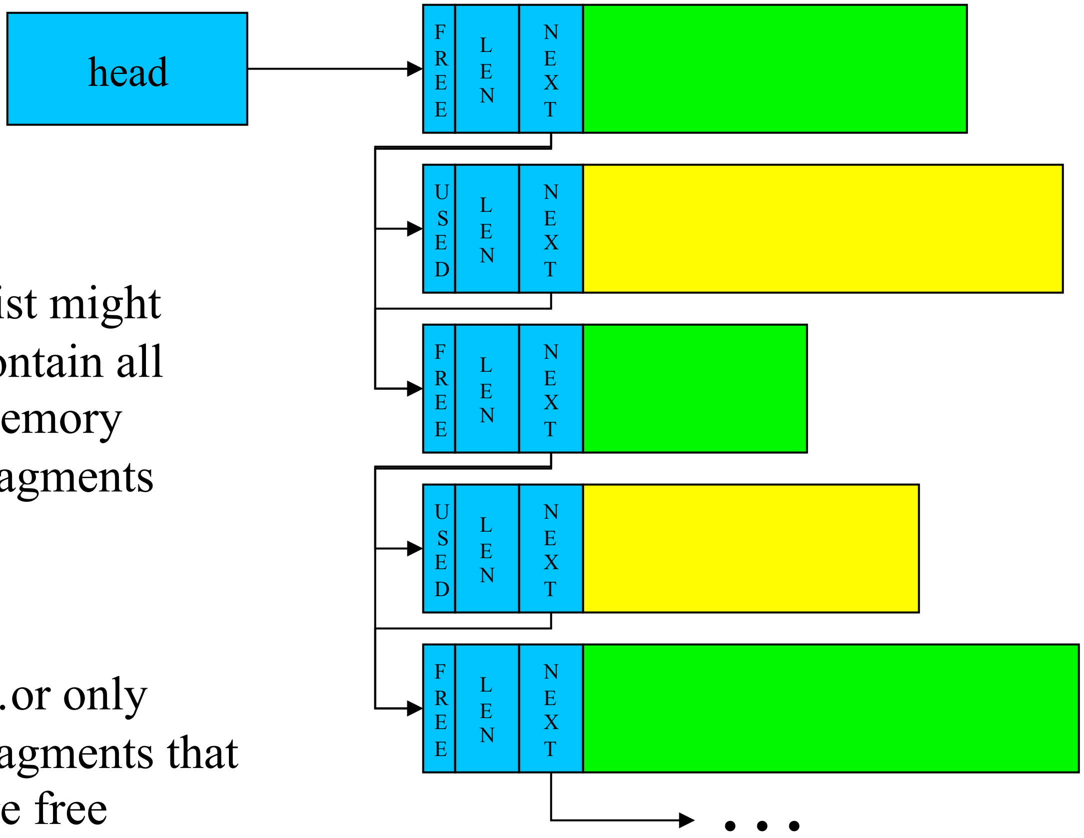
How To Keep Track of Variable Sized Partitions?

- 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

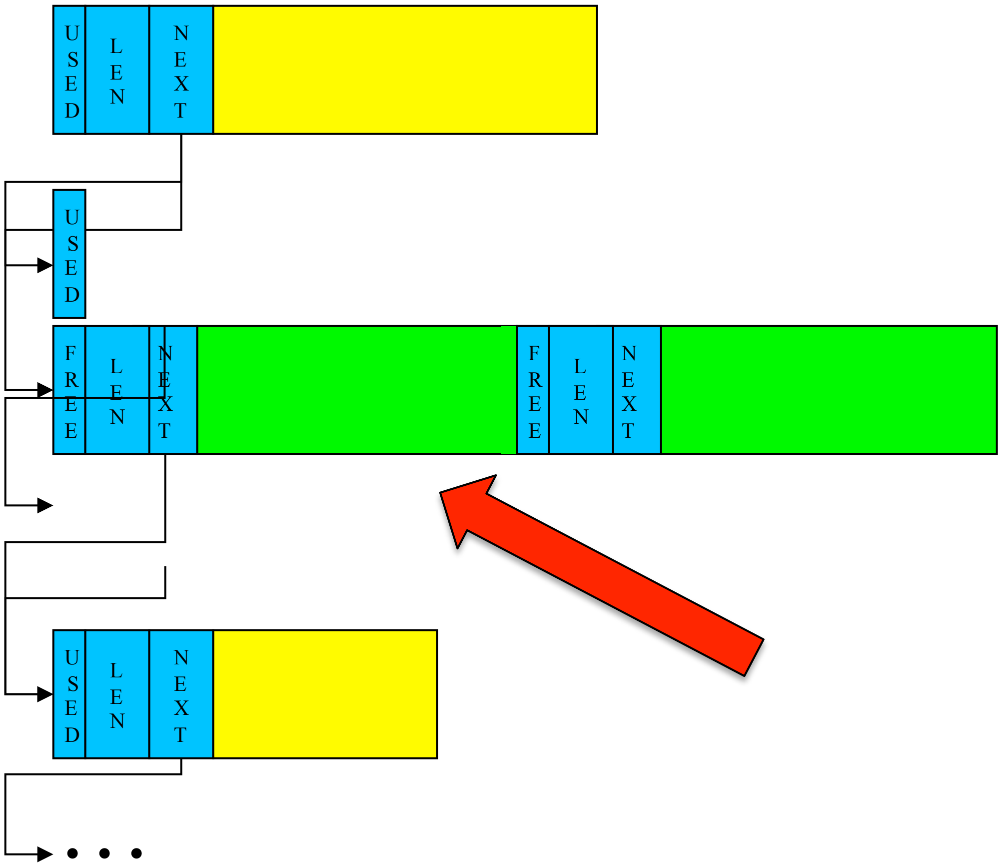
The Free List



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

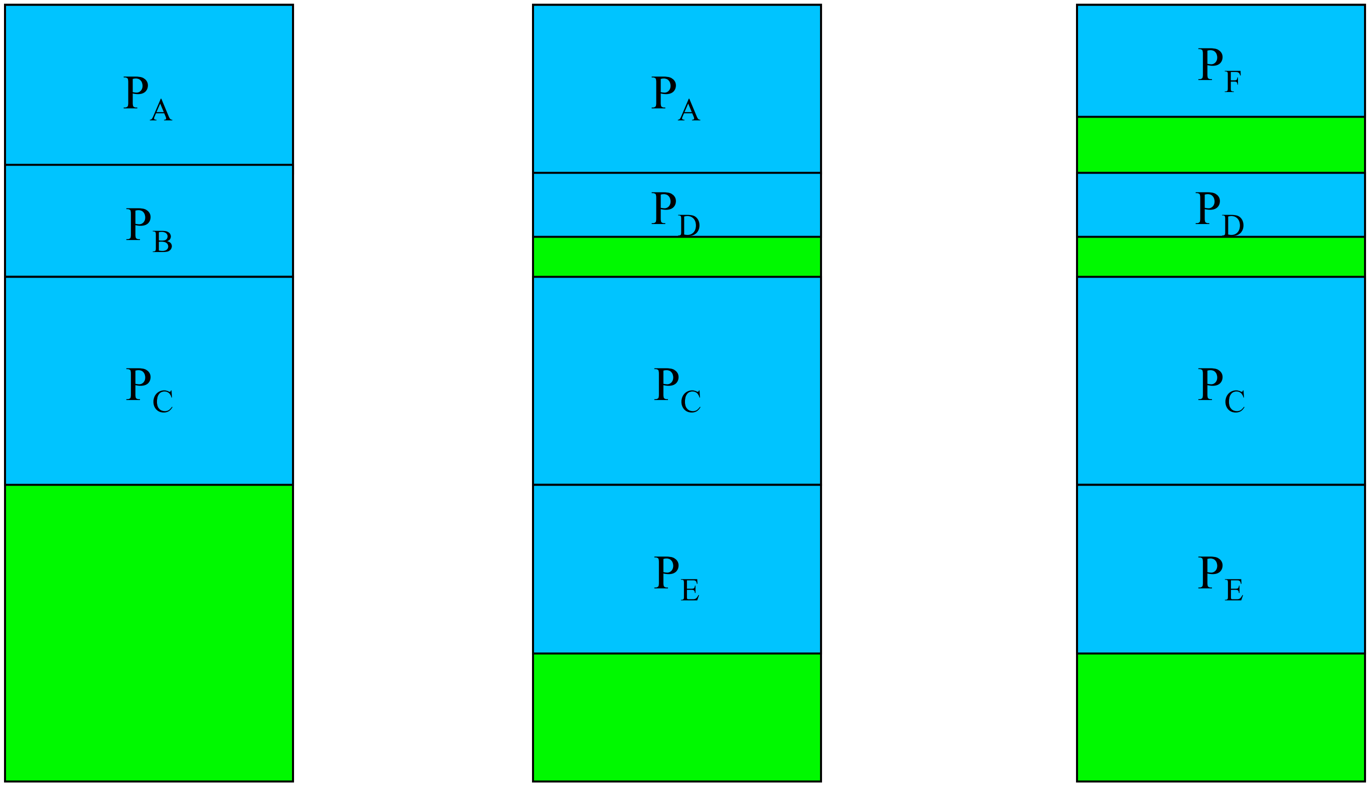
4. Insert the new chunk into the list

5. Mark the carved piece as in use

Variable Partitions and Fragmentation

- Variable sized partitions 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

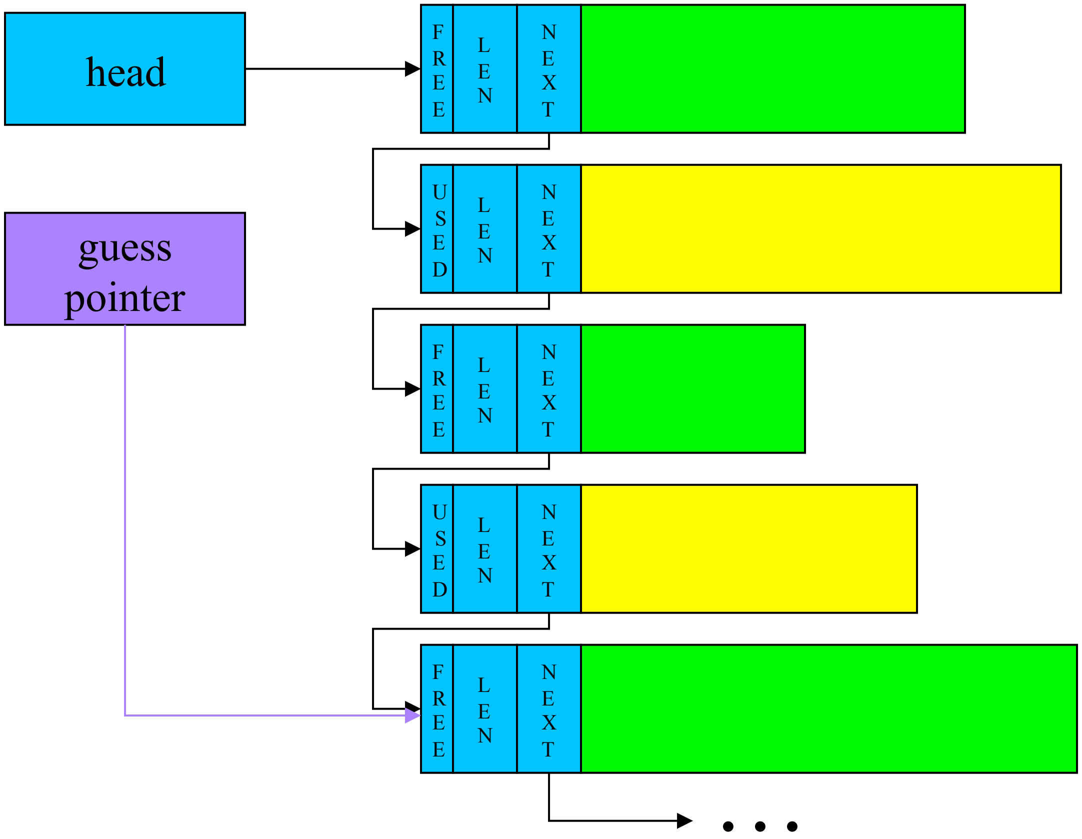
- Search for the “worst fit” chunk
  - Largest size greater than or equal to requested size
- Advantages:
  - Tends to create very large fragments

    … 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.

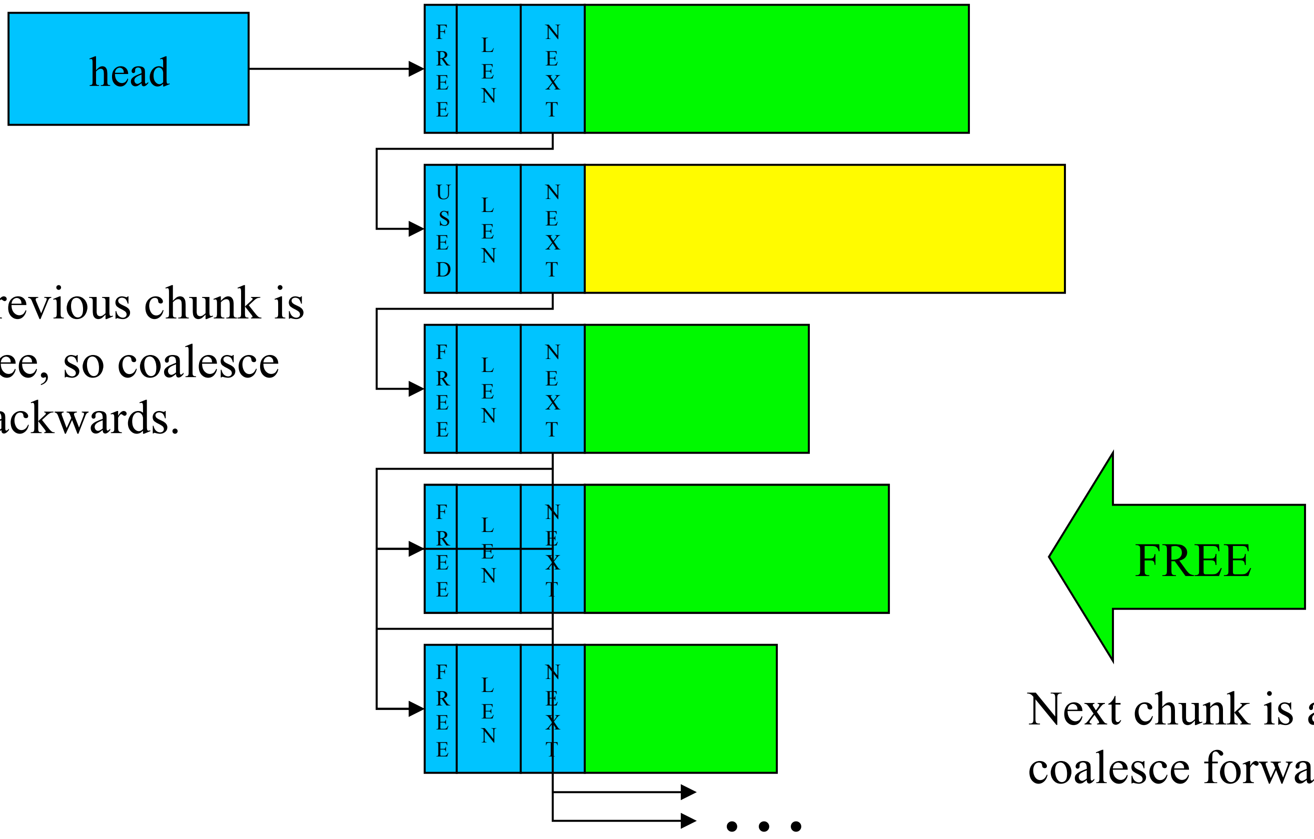
Next Fit Properties

- Tries to get advantages of both first and worst fit
  - Short searches (maybe shorter than first fit)
  - Spreads out fragmentation (like worst 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 Partitions

- All variable sized partition 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 Partition 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?

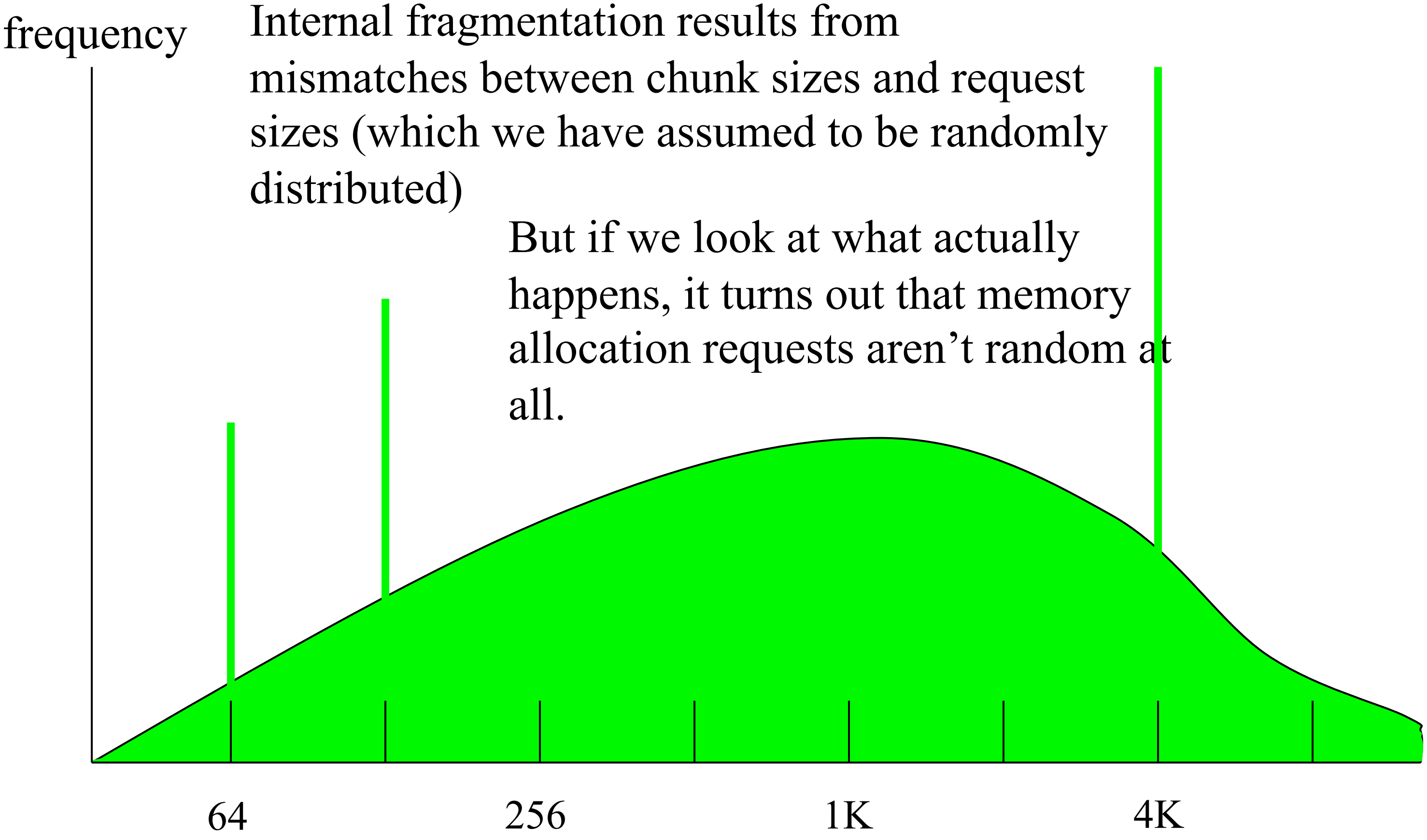
Another Option

- Fixed partition allocations result in internal fragmentation
  - Processes don’t use all of the fixed partition
- Dynamic partition allocations result in external fragmentation
  - The elements on the memory free list get smaller and less useful
- Can we strike a balance in between?

A Special Case for Fixed Allocations

Internal fragmentation results from mismatches between chunk sizes and request sizes (which we have assumed to be randomly distributed)

But if we look at what actually happens, it turns out that memory allocation requests aren’t random at all.



Why Aren’t Memory Request Sizes Randomly Distributed?

- In real systems, some sizes are requested much more often than others
- Many key services use fixed-size buffers
  - File systems (for disk I/O)
  - Network protocols (for packet assembly)
  - Standard request descriptors
- These account for much transient use
  - They are continuously allocated and freed
- OS might want to handle them specially

Buffer Pools

- If there are popular sizes,
  - Reserve special pools of fixed size buffers
  - Satisfy matching requests from those pools
- Benefit: improved efficiency
  - Much simpler than variable partition allocation
    - Eliminates searching, carving, coalescing
  - Reduces (or eliminates) external fragmentation
- But we must know how much to reserve
  - Too little, and the buffer pool will become a bottleneck
  - Too much, and we will have a lot of unused buffer space
- <u>Only</u> satisfy perfectly matching requests
  - Otherwise, back to internal fragmentation

How Are Buffer Pools Used?

- Process requests a piece of memory for a special purpose
  - E.g., to send a message
- System supplies one element from buffer pool
- Process uses it, completes, frees memory
  - Maybe explicitly
  - Maybe implicitly, based on how such buffers are used
    - E.g., sending the message will free the buffer “behind the process’ back” once the message is gone

Dynamically Sizing Buffer Pools

- If we run low on fixed sized buffers
  - Get more memory from the free list
  - Carve it up into more fixed sized buffers
- If our free buffer list gets too large
  - Return some buffers to the free list
- If the free list gets dangerously low
  - Ask each major service with a buffer pool to return space
- This can be tuned by a few parameters:
  - Low space (need more) threshold
  - High space (have too much) threshold
  - Nominal allocation (what we free down to)
- Resulting system is highly adaptive to changing loads

Lost Memory

- One problem with buffer pools is memory leaks
  - The process is done with the memory
  - But doesn't free it
- Also a problem when a process manages its own memory space
  - E.g., it allocates a big area and maintains its own free list
- Long running processes with memory leaks can waste huge amounts of memory

Garbage Collection

- One solution to memory leaks
- Don’t count on processes to release memory
- Monitor how much free memory we’ve got
- When we run low, start garbage collection
  - Search data space finding every object pointer
  - Note address/size of all accessible objects
  - Compute the compliment (what is inaccessible)
  - Add all inaccessible memory to the free list

How Do We Find All Accessible Memory?

- Object oriented languages often enable this
  - All object references are tagged
  - All object descriptors include size information
- It is often possible for system resources
  - Where all possible references are known
    - (E.g., we know who has which files open)
- How about for the general case?

General Garbage Collection

- Well, what would you need to do?
- Find all the pointers in allocated memory
- Determine “how much” each points to
- Determine what is and is not still pointed to
- Free what isn’t pointed to
- Why might that be difficult?

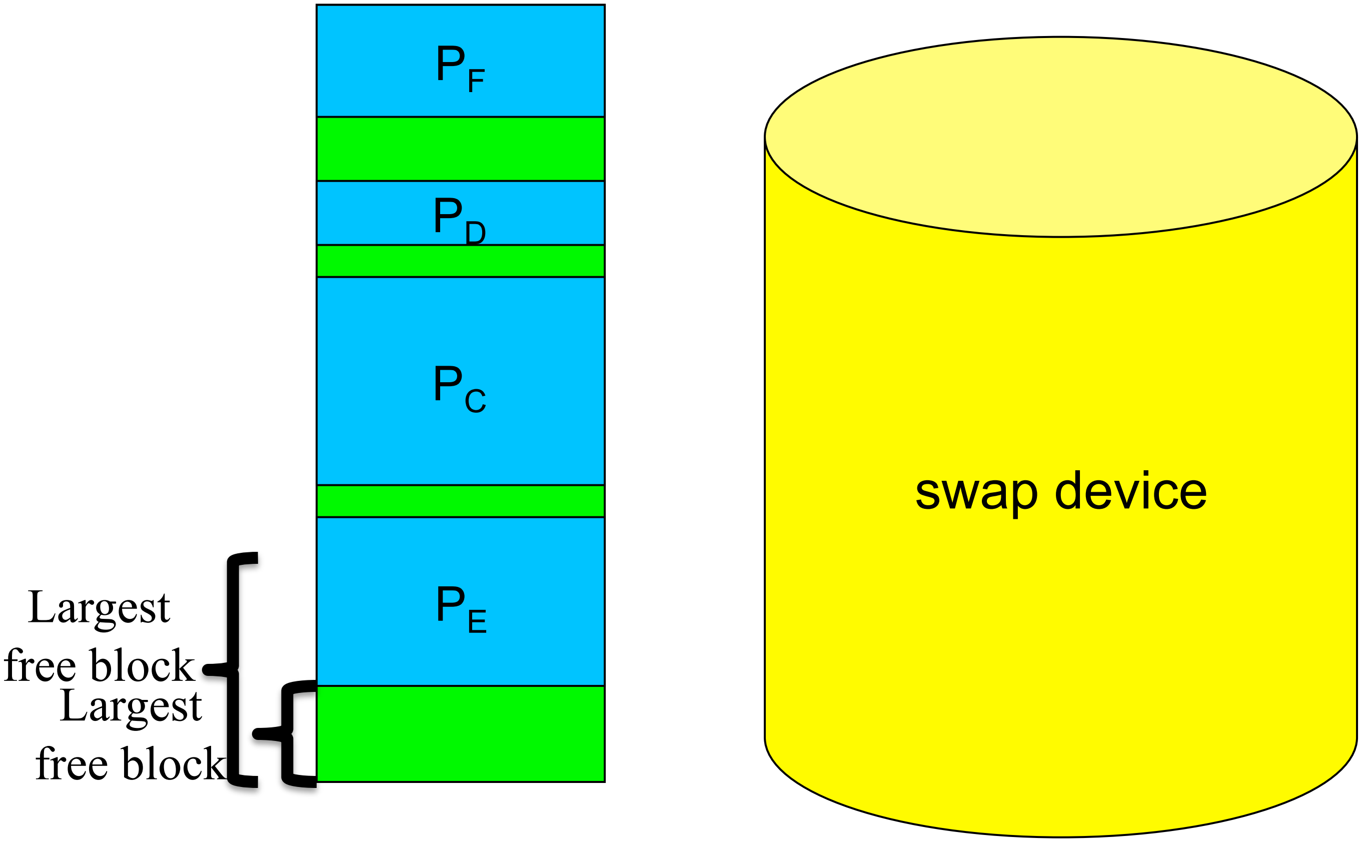
Problems With General Garbage Collection

- A location in the data or stack segments might <u>seem</u> to contain addresses, but ...
  - Are they truly pointers, or might they be other data types whose values happen to resemble addresses?
  - If pointers, are they themselves still accessible?
  - We might be able to infer this (recursively) for pointers in dynamically allocated structures …
  - But what about pointers in statically allocated (potentially global) areas?
- And how much is “pointed to,” one word or a million?

Compaction and Relocation

- Garbage collection is just another way to free memory
  - Doesn't greatly help or hurt fragmentation
- Ongoing activity can starve coalescing
  - Chunks reallocated before neighbors become free
- We could stop accepting new allocations
  - But resulting convoy on memory manager would trash throughput
- We need a way to rearrange active memory
  - Re-pack all processes in one end of memory
  - Create one big chunk of free space at other end

Memory Compaction



Now let's compact!

An obvious improvement!

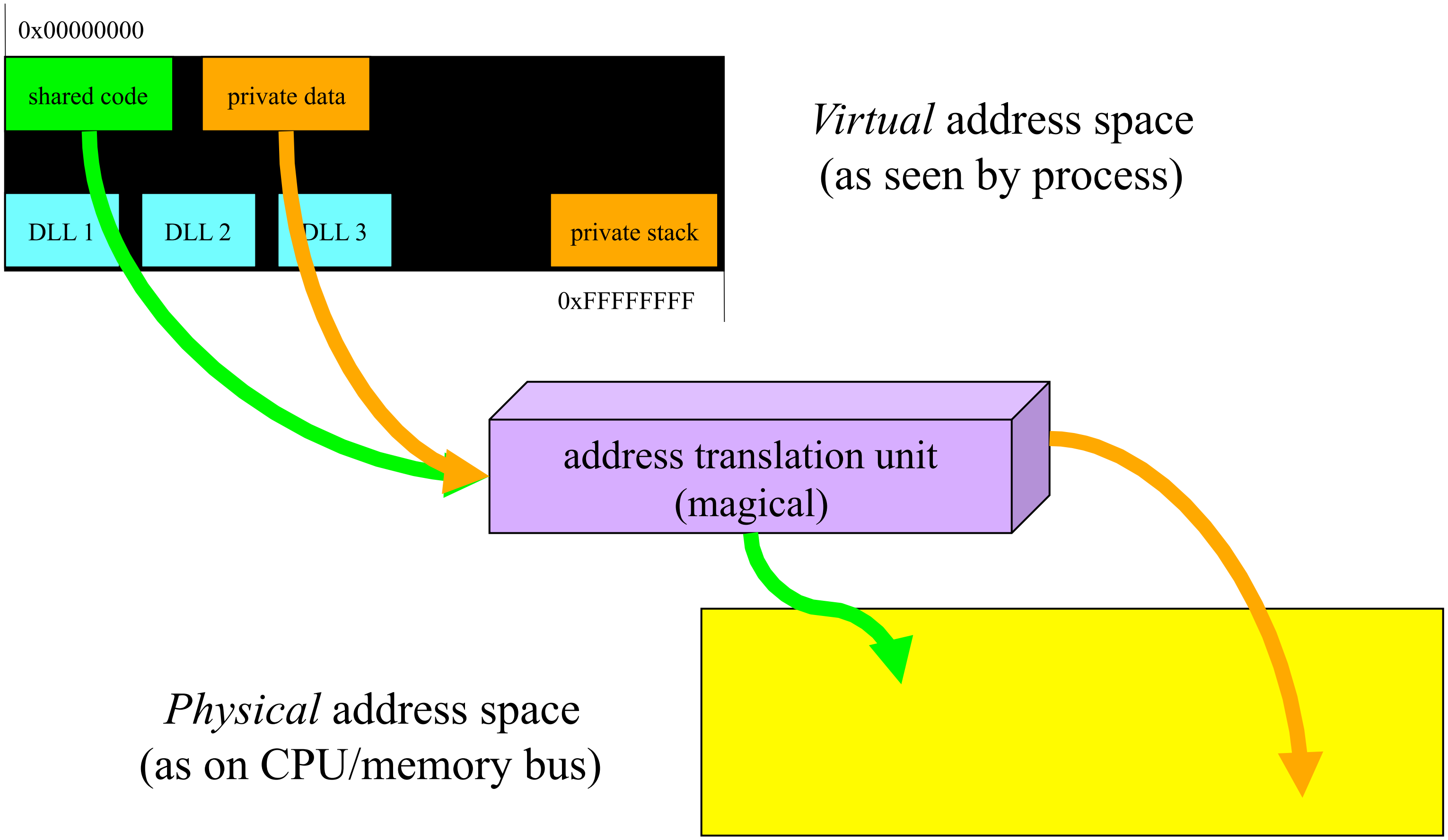
All This Requires Is Relocation . . .

- The ability to move a process
  - From region where it was initially loaded
  - Into a new and different region of memory
- What’s so hard about that?
- All addresses in the program will be wrong
  - References in the code segment
    - Calls and branches to other parts of the code
    - References to variables in the data segment
  - Plus new pointers created during execution
    - That point into data and stack segments

The Relocation Problem

- It is not generally feasible to re-relocate a process
  - Maybe we could relocate references to code
    - If we kept the relocation information around
  - But how can we relocate references to data?
    - Pointer values may have been changed
    - New pointers may have been created
- We could never find/fix all address references
  - Like the general case of garbage collection
- Can we make processes location independent?

Virtual Address Spaces



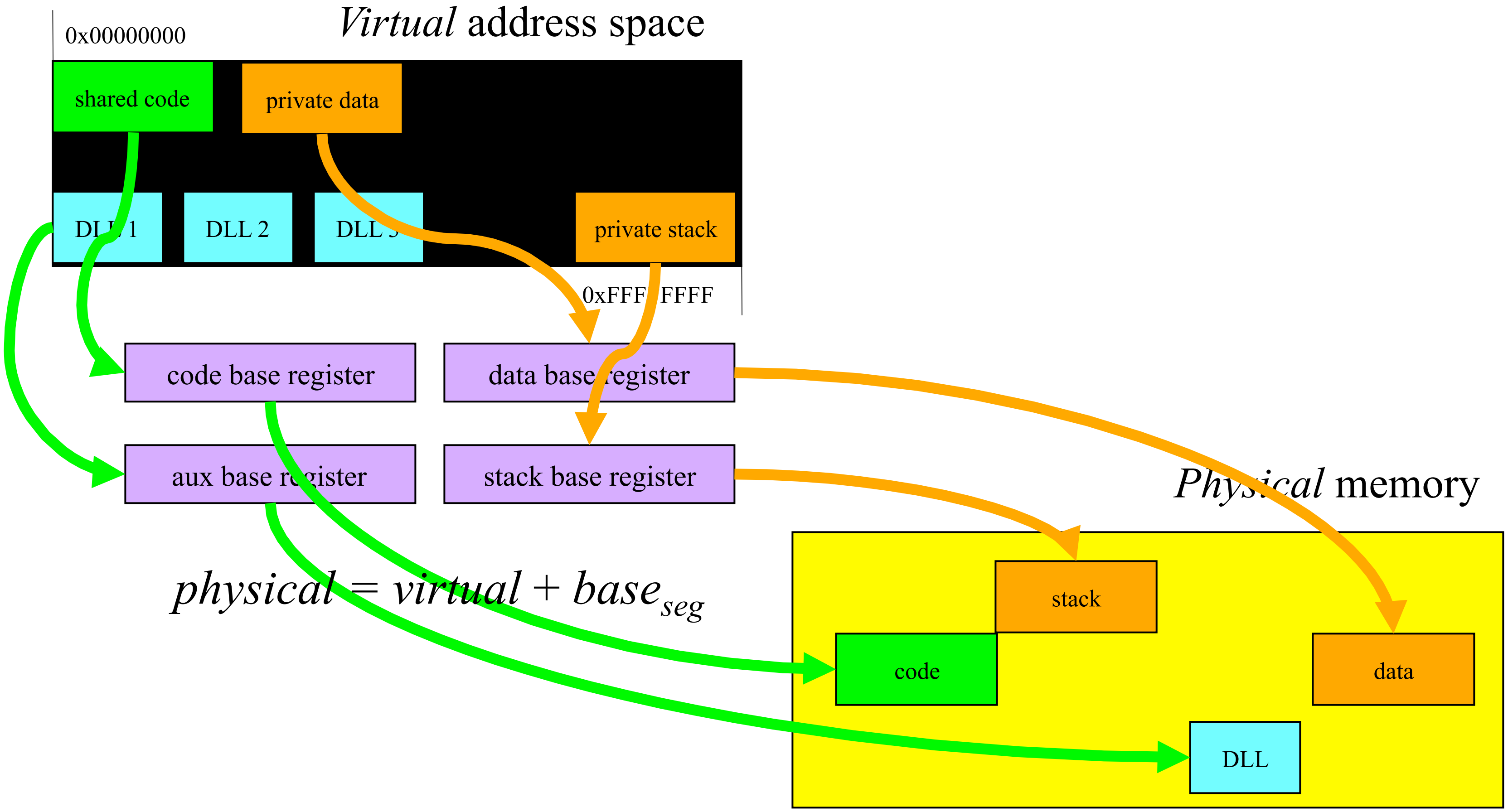
Virtual address space
(as seen by process)

Physical address space
(as on CPU/memory bus)

Memory Segment Relocation

- A natural model
  - Process address space is made up of multiple segments
  - Use the segment as the unit of relocation
  - Long tradition, from the IBM system 360 to Intel x86 architecture
- Computer has special relocation registers
  - They are called segment base registers
  - They point to the start (in physical memory) of each segment
  - CPU automatically adds base register to every address
- OS uses these to perform virtual address translation
  - Set base register to start of region where program is loaded
  - If program is moved, reset base registers to new location
  - Program works no matter where its segments are loaded

How Does Segment Relocation Work?

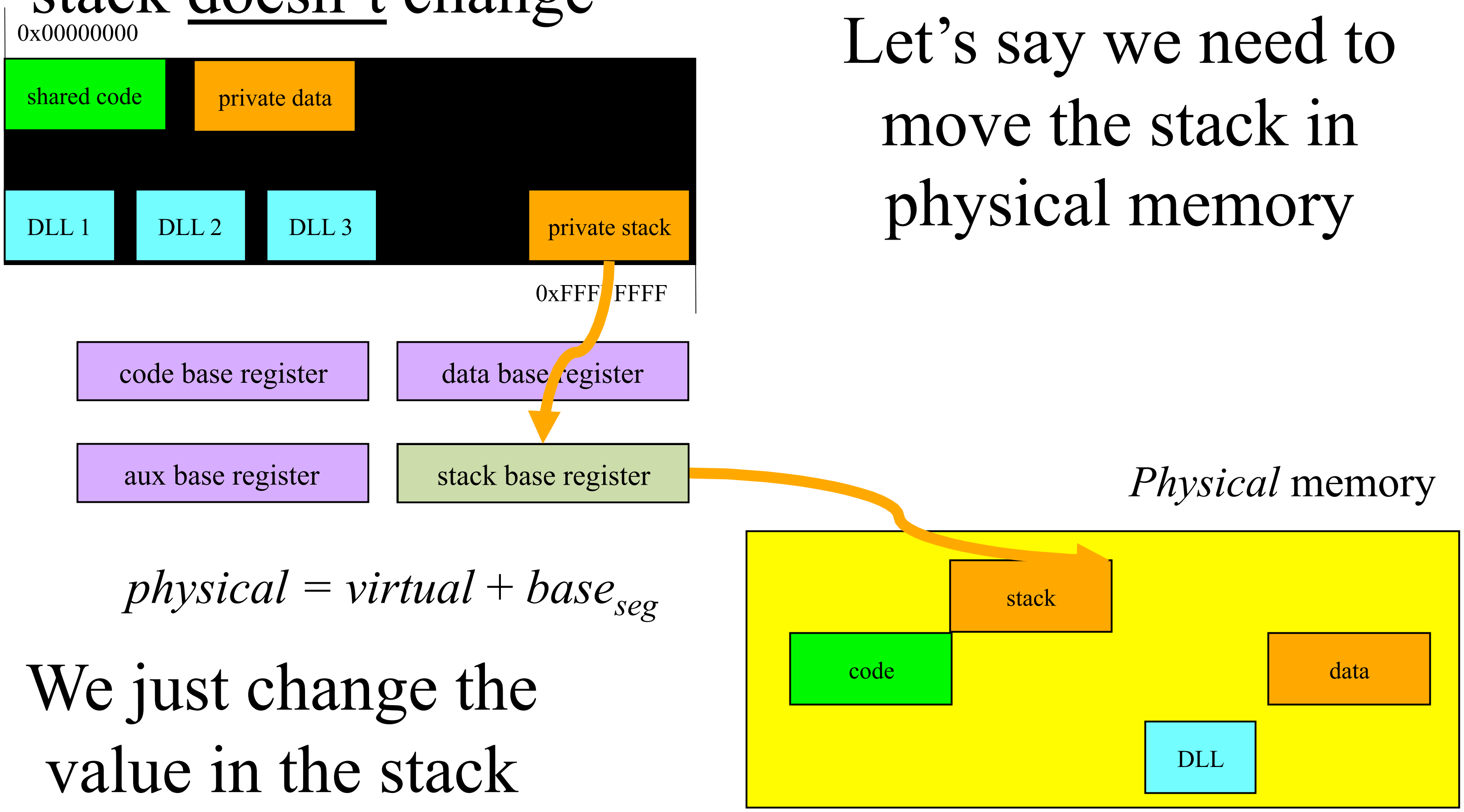


$physical = virtual + base_{seg}$

Relocating a Segment

The virtual address of the stack <u>doesn't</u> change

Let's say we need to move the stack in physical memory



$$physical = virtual + base_{seg}$$

We just change the value in the stack base register

Relocation and Safety

- A relocation mechanism (like base registers) is good
  - It solves the relocation problem
  - Enables us to move process segments in physical memory
  - Such relocation turns out to be insufficient
- We also need protection
  - Prevent process from reaching outside its allocated memory
    - E.g., by overrunning the end of a mapped segment
- Segments also need a length (or limit) register
  - Specifies maximum legal offset (from start of segment)
  - Any address greater than this is illegal (in the hole)
  - CPU should report it via a <u>segmentation</u> exception (trap)

How Much of Our Problem Does Relocation Solve?

- We can use variable sized partitions
  - Cutting down on internal fragmentation
- We can move partitions around
  - Which helps coalescing be more effective
  - But still requires contiguous chunks of data for segments
  - So external fragmentation is still a problem
- We need to get rid of the requirement of contiguous segments