Operating Systems Principles
Memory Management

Mark Kampe
(markk@cs.ucla.edu)

Memory Management

Memory Management

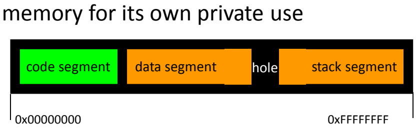
1. allocate/assign physical memory to processes
   - explicit requests: malloc (sbrk)
   - implicit: program loading, stack extension
2. manage the virtual address space
   - instantiate virtual address space on context switch
   - extend or reduce it on demand
3. manage migration to/from secondary storage
   - optimize use of main storage
   - minimize overhead (waste, migrations)

Memory Management Goals

1. transparency
   - process sees only its own virtual 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

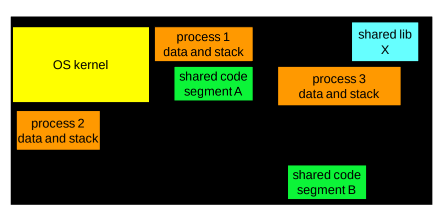
Linux process virtual address space

a process's virtual address space is made up of all of the memory locations that the process can address, and looks as if the process had all of memory for its own private use



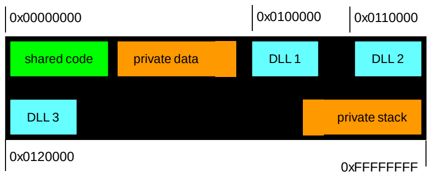
Code segment loaded near beginning of address space
Data segment starts at page boundary after code segment
Stack segment starts at high end of address space

Physical Memory Allocation



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

Linux Process – virtual address space



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

(code segments)

- program code
  - allocated when program loaded
  - initialized with contents of load module
- shared and Dynamically Loadable Libraries
  - mapped in at exec time or when needed
- all are read-only and fixed size
  - somehow shared by multiple processes
  - shared code must be read only

(implementing: code segments)

- program loader
  - ask for memory (size and virtual location)
  - copy code from load module into memory
- run-time loader
  - request DLL be mapped (location and size)
  - edit PLT pointers from program to DLL
- memory manager
  - allocates memory, maps into process

(data/stack segments)

- they are process-private, read/write
- initialized data
  - allocated when program loaded
  - initialized from load module
- data segment expansion/contraction
  - requested via system calls (e.g. sbrk)
  - only added/truncated part is affected
- process stack
  - allocated and grown automatically on demand

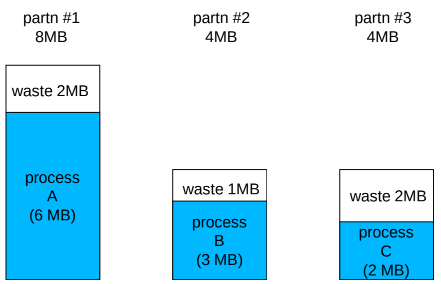
(implementing: data/stack)

- program loader
  - ask for memory (location and size)
  - copy data from load module into memory
  - zero the uninitialized data
- memory manager
  - invoked for allocations and stack extensions
  - allocates and deallocates memory
  - adjusts process address space accordingly

Fixed Partition Memory Allocation

- pre-allocate partitions for n processes
  - reserving space for largest possible process
- very easy to implement
  - common in old batch processing systems
- well suited to well-known job mix
  - must reconfigure system for larger processes
- likely to use memory inefficiently
  - large internal fragmentation losses
  - swapping results in convoys on partitions

Internal Fragmentation



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

(Internal Fragmentation)

- wasted space in fixed sized blocks
- caused by a mis-match between
  - the chosen sizes of a fixed-sized blocks
  - the actual sizes that programs request
- 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

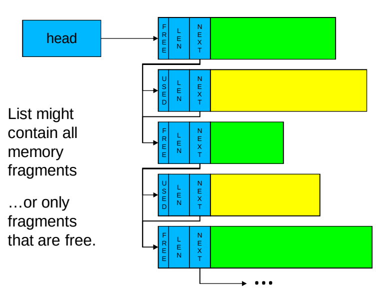
Stack vs Heap Allocation

- stack allocation
  - compiler manages space (locals, call info)
  - data is valid until stack frame is popped
  - OS automatically extends stack segment
- heap allocation
  - explicitly allocated by application (malloc/new)
  - data is valid until free/delete (or G.C.)
  - heap space managed by user-mode library
  - data segment size adjusted by system call

Variable Partition Allocation

- start with one large "heap" of 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 the free list
- when a process frees memory
  - put it back on the free list
- eliminates internal fragmentation losses

Variable Partition Free List

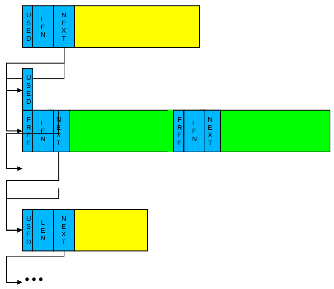


(Free lists: keeping track of it all)

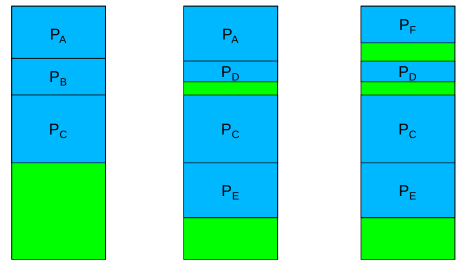
- fixed sized blocks are easy to track
  - a bit map indicating which blocks are free
- variable chunks require more information
  - a linked list of descriptors, one per chunk
  - each lists size of chunk, whether it is free
  - each has pointer to next chunk on list
  - descriptors often at front of each chunk
- allocated memory may have descriptors too

Free Chunk Carving

1. find large enough free chunk
2. reduce len to requested size
3. new header for residual chunk
4. insert new chunk into list
5. mark carved piece as in use



External Fragmentation



(External/Global Fragmentation)

- each allocation creates left-overs
  - 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

Which chunk: best fit

- search for the "best fit" chunk
  - smallest size greater/equal to requested size
- advantages:
  - might find a perfect fit
- disadvantages:
  - have to search entire list every time
  - quickly creates very small fragments

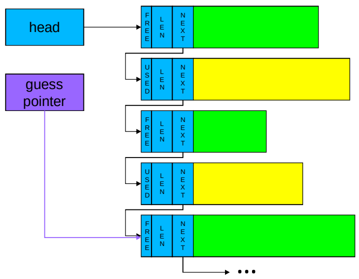
Which chunk: worst fit

- search for the "worst fit" chunk
  - largest size greater/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

Which chunk: first fit

- take first chunk 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

Which Chunk: 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 ... guess pointers)

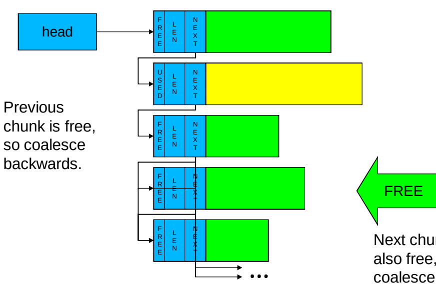
- the best of both worlds
  - short searches (maybe shorter than first fit)
  - spreads out fragmentation (like worst fit)
- guess pointers are a general technique
  - think of them as a lazy (non-coherent) cache
  - if they are right, they save a lot of time
  - if they are wrong, the algorithm still works
  - they can be used in a wide range of problems

Coalescing – de-fragmentation

- all VP algorithms have ext fragmentation
  - some get it faster, some spread it out
- we need a way to reassemble fragments
  - check neighbors when ever 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.

Free list must support coalescing

- coalescing happens at free time
  - when freeing a region, check its neighbors
  - if either neighbor is free, recombine them
- must be easy to find both adjacent regions
  - e.g. doubly linked list of region descriptors
  - other coupling between neighbors
    - (e.g. buddy system, where all regions are paired)

Coalescing vs. Fragmentation

- opposing processes 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

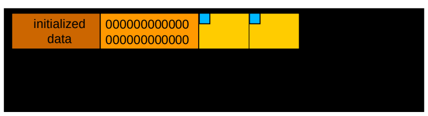
Fixed vs Variable Partition

- Fixed partition allocation
  - allocation and free lists are trivial
  - internal fragmentation is inevitable
    - average 50% (unless we have multiple sizes)
- Variable partition allocation
  - allocation is complex and expensive
    - long searches of complex free lists
  - eliminates internal fragmentation
  - external fragmentation is inevitable
    - can be managed by (complex) coalescing

User-mode memory allocation

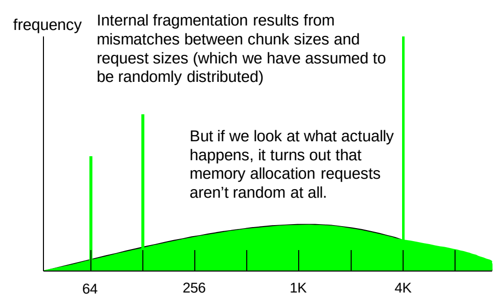
- use OS to get memory for process
  - e.g. sbrk system call to extend data segment
- UNIX malloc (user mode allocation)
  - variable partition, first fit-allocation
  - go back to OS to get more if heap is empty
- UNIX mfree (return memory when done)
  - return memory to free list
  - coalescing of contiguous free chunks

managing process private data



1. loader allocates space for, and copies initialized data from load module.
2. loader allocates space for, and zeroes uninitialized data from load module.
3. after it starts, program uses sbrk to extend the process data segment and then puts the newly created chunk on the free list
4. Free space "heap" is eventually consumed program uses sbrk to further extend the process data segment and then puts the newly created chunk on the free list

Memory Allocation Requests



(memory allocation requests)

- memory requests are not well distributed
  - some sizes are used much more 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

Special Buffer Pools

- if there are popular sizes
  - reserve special pools of fixed size buffers
  - allocate/free matching requests from those pools
- benefit: improved efficiency
  - much simpler than variable partition allocation
  - reduces (or eliminates) external fragmentation
- but ... we must know how much to reserve
  - too little: buffer pool will become a bottleneck
  - too much: we will have a lot of idle space

Balancing Space for Buffer Pools

- many different special purpose pools
  - demand for each changes continuously
  - memory needs to migrate between them
- sounds like dynamic a equilibrium
  - managed free space margins
    - maximum allowable free space per service
    - graceful handling of changing loads
  - claw-back call-back
    - OS requests services to free all available memory
    - prompt handling of emergencies

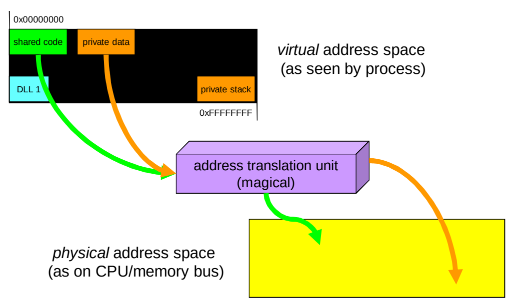
Buffer Pools – Slab Allocation

- requests are not merely for common sizes
  - they are often for the same data structure
  - or even assemblies of data structures
- initializing and demolition are expensive
  - many fields and much structure are constant
- stop destroying and reinitializing
  - recycle data structures (or assemblies)
  - only reinitialize the fields that must be changed
  - only disassemble to give up the space

The Need for Dynamic Relocation

- there are a few reasons to move a process
  - needs a larger chunk of memory
  - swapped out, swapped back in to a new location
  - to compact fragmented free space
- all addresses in the program will be wrong
  - references in the code, pointers in the data
- it is not feasible to re-linkage edit the program
  - new pointers have been created during run-time

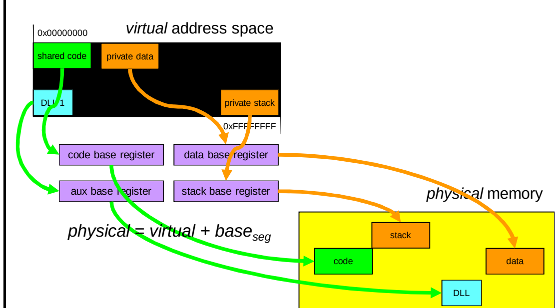
Virtual Address Translation



Segment Relocation

- a natural unit of allocation and relocation
  - process address space made up of segments
  - each segment is contiguous w/no holes
- CPU has segment base registers
  - point to (physical memory) base of each segment
  - CPU automatically relocates all references
- OS uses for virtual address translation
  - set base to region where segment is loaded
  - efficient: CPU can relocate every reference
  - transparent: any segment can move anywhere

Segment Relocation



Privacy and Protection

- confine process to its own address space
  - associate a length (or limit) with each segment
  - CPU verifies all offsets are within range
  - generates addressing exception if not
- protecting read-only segments
  - associate read/write access with each segment
  - CPU ensures integrity of read-only segments
- segmentation register update is privileged
  - only kernel-mode code can do this

assignments

- reading for the next lecture (moderately long)
  - Arpaci ch 18 … Introduction to Paging
  - Arpaci ch 19 … Translation Look-Aside Buffers
  - Arpaci ch 20 … Advanced Page Tables
  - Arpaci ch 21 … Swapping Mechanisms
  - Arpaci ch 22 … Swapping Policies
  - Working Sets and replacement algorithms

Supplementary Slides

Garbage Collection

- programmers often forget to free memory
- garbage collection is alternative to freeing
  - applications allocate objects, never free them
- when we run out, 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

Finding all *accessible* data

- 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)
- in general, however it is impossible
  - most languages do not support it

Diagnostic Free lists

- common mistakes w/dynamic memory
  - memory leaks (allocate it and never free it)
  - overruns (use more than you allocated)
  - clobbers (keep on using it after you free it)
- free list can help to catch these problems
  - all chunks in list (whether allocated or free)
  - record of who last allocated each chunk
  - guard zones at beginning and end of chunks

Other Dynamic Memory Advice

- uninitialized pointers, forget to allocate
  - some OS leave page 0 un-mapped
- returning pointers to local variables
  - “gcc –ansi –pedantic –wall” will catch these
- continued use, multiple frees
  - null pointers after freeing the memory
  - avoid keeping multiple pointers to an object
- buffer over-runs
  - use newer APIs with length parameters

Diagnostic Free List

- standard chunk header
  - free bit, chunk length, next chunk pointer
- allocation audit info
  - tracing down the source of memory leaks
- guard zones
  - detect application buffer over-runs
- zero memory when it is freed
  - detect continued use after chunk is freed

UNIX stack space management

Data segment starts at page boundary after code segment
Stack segment starts at high end of address space
Unix extends stack automatically as program needs more.

Data segment grows up; Stack segment grows down
Both grow towards the hole in the middle. They are not allowed to meet.