

INF333 - Operating Systems

Lecture X

Burak Arslan

ext-inf333@burakarslan.com 

Galatasaray Üniversitesi

Lecture X

2024-05-21

Course website

burakarslan.com/inf333 

Based On

cs212.stanford.edu 

OSC-10 Slides 

Dynamic memory allocation

Almost every useful program uses it

- ▶ Gives wonderful functionality benefits
 - ▶ Don't have to statically specify complex data structures
 - ▶ Can have data grow as a function of input size
 - ▶ Allows recursive procedures (stack growth)
- ▶ But, can have a huge impact on performance

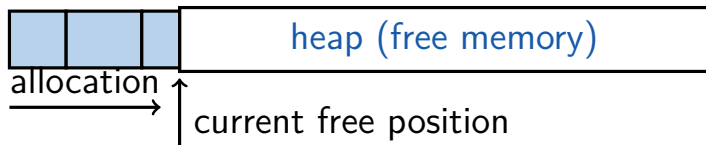
Dynamic memory allocation

Some interesting facts:

- ▶ Two or three line code change can have huge, non-obvious impact on how well allocator works (examples to come)
- ▶ Proven: impossible to construct an "always good" allocator
- ▶ Surprising result: memory management still poorly understood

Why is it hard?

- ▶ Satisfy arbitrary set of allocation and frees.
- ▶ Easy without free: set a pointer to the beginning of some big chunk of memory (“heap”) and increment on each allocation:

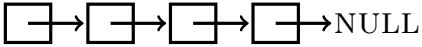


- ▶ Problem: free creates holes (“fragmentation”)
Result? Lots of free space but cannot satisfy request!



More abstractly

freelist

- ▶ What an allocator must do?  $\square \rightarrow \square \rightarrow \square \rightarrow \square \rightarrow \text{NULL}$
 - ▶ Track which parts of memory in use, which parts are free
 - ▶ Ideal: no wasted space, no time overhead
- ▶ What the allocator cannot do?
 - ▶ Control order of the number and size of requested blocks
 - ▶ Know the number, size, or lifetime of future allocations
 - ▶ Move allocated regions (bad placement decisions permanent)

`malloc(20)?`

20	10	20	10	20
----	----	----	----	----

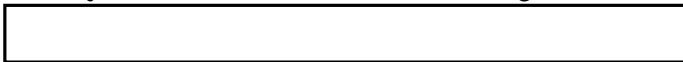
- ▶ The core fight: minimize fragmentation
 - ▶ App frees blocks in any order, creating holes in “heap”
 - ▶ Holes too small? cannot satisfy future requests

What is fragmentation really?

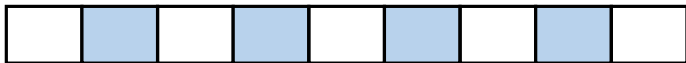
- ▶ Inability to use memory that is free
- ▶ Two factors required for fragmentation
 1. Different lifetimes—if adjacent objects die at different times, then fragmentation:



- ▶ If all objects die at the same time, then no fragmentation:

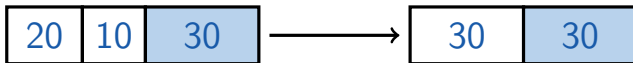


2. Different sizes: If all requests the same size, then no fragmentation (that's why no external fragmentation with paging):



Important decisions

- ▶ Placement choice: where in free memory to put a requested block?
 - ▶ Freedom: can select any memory in the heap
 - ▶ Ideal: put block where it won't cause fragmentation later (impossible in general: requires future knowledge)
- ▶ Split free blocks to satisfy smaller requests?
 - ▶ Fights internal fragmentation
 - ▶ Freedom: can choose any larger block to split
 - ▶ One way: choose block with smallest remainder (best fit)
- ▶ Coalescing free blocks to yield larger blocks



- ▶ Freedom: when to coalesce (deferring can save work)
- ▶ Fights external fragmentation

Impossible to “solve” fragmentation

- ▶ If you read allocation papers to find the best allocator
 - ▶ All discussions revolve around tradeoffs
 - ▶ The reason? There cannot be a best allocator
- ▶ Theoretical result:
 - ▶ For any possible allocation algorithm, there exist streams of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation.
- ▶ How much fragmentation should we tolerate?
 - ▶ Let M = bytes of live data, n_{\min} = smallest allocation, n_{\max} = largest
 - How much gross memory required?
 - ▶ Bad allocator: $M \cdot (n_{\max}/n_{\min})$
 - ▶ E.g., only ever use a memory location for a single size
 - ▶ E.g., make all allocations of size n_{\max} regardless of requested size
 - ▶ Good allocator: $\sim M \cdot \log(n_{\max}/n_{\min})$

Pathological examples

- ▶ Suppose heap currently has 7 20-byte chunks



- ▶ What's a bad stream of frees and then allocates?
- ▶ Given a 128-byte limit on `malloc()`'d space
 - ▶ What's a really bad combination of mallocs & frees?
- ▶ Next: two allocators (best fit, first fit) that, in practice, work pretty well
 - ▶ “pretty well” = $\sim 20\%$ fragmentation under many workloads

Pathological examples

- ▶ Suppose heap currently has 7 20-byte chunks

20	20	20	20	20	20	20
----	----	----	----	----	----	----

- ▶ What's a bad stream of frees and then allocates?
 - ▶ Free every other chunk, then alloc 21 bytes
- ▶ Given a 128-byte limit on `malloc()`'d space
 - ▶ What's a really bad combination of mallocs & frees?
- ▶ Next: two allocators (best fit, first fit) that, in practice, work pretty well
 - ▶ “pretty well” = $\sim 20\%$ fragmentation under many workloads

Pathological examples

- ▶ Suppose heap currently has 7 20-byte chunks

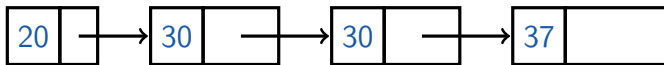
20	20	20	20	20	20	20
----	----	----	----	----	----	----

- ▶ What's a bad stream of frees and then allocates?
 - ▶ Free every other chunk, then alloc 21 bytes
- ▶ Given a 128-byte limit on `malloc()`'d space
 - ▶ What's a really bad combination of mallocs & frees?
 - ▶ Malloc 128 1-byte chunks, free every other
 - ▶ Malloc 32 2-byte chunks, free every other (1- & 2-byte) chunk
 - ▶ Malloc 16 4-byte chunks, free every other chunk...
- ▶ Next: two allocators (best fit, first fit) that, in practice, work pretty well
 - ▶ “pretty well” = $\sim 20\%$ fragmentation under many workloads

Best fit

- Strategy: minimize fragmentation by allocating space from block that leaves smallest fragment

- Data structure: heap is a list of free blocks, each has a header holding block size and a pointer to the next block



- Code: Search freelist for block closest in size to the request. (Exact match is ideal)
 - During free (usually) coalesce adjacent blocks
- Potential problem: Sawdust
 - Remainder so small that over time left with “sawdust” everywhere
 - Fortunately not a problem in practice

Best fit gone wrong

- ▶ Simple bad case: allocate n , m ($n < m$) in alternating orders, free all the n s, then try to allocate an $n + 1$
- ▶ Example: start with 99 bytes of memory

- ▶ alloc 19, 21, 19, 21, 19



- ▶ free 19, 19, 19:



- ▶ alloc 20? Fails! (wasted space = 57 bytes)

- ▶ However, doesn't seem to happen in practice

First fit

Strategy: pick the first block that fits

- ▶ Data structure: free list, sorted LIFO, FIFO, or by address
- ▶ Code: scan list, take the first one

LIFO: put free object in front of list.

- ▶ Simple, but causes higher fragmentation
- ▶ Potentially good for cache locality

Address sort: order free blocks by address

- ▶ Makes coalescing easy (just check if next block is free)
- ▶ Also preserves empty/idle space (locality good when paging)

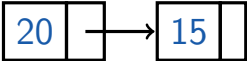
FIFO: put free object at end of list

- ▶ Gives similar fragmentation as address sort, but unclear why

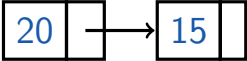
Subtle pathology: LIFO FF

- ▶ Storage management example of subtle impact of simple decisions
- ▶ LIFO first fit seems good:
 - ▶ Put object in front of list (cheap), hope same size used again (cheap + good locality)
- ▶ But, has big problems for simple allocation patterns:
 - ▶ E.g., repeatedly intermix short-lived $2n$ -byte allocations, with long-lived $(n + 1)$ -byte allocations
 - ▶ Each time large object freed, a small chunk will be quickly taken, leaving useless fragment. Pathological fragmentation

First fit: Nuances

- ▶ First fit sorted by address order, in practice:
 - ▶ Blocks at front preferentially split, ones at back only split when no larger one found before them
 - ▶ Result? Seems to roughly sort free list by size
 - ▶ So? Makes first fit operationally similar to best fit: a first fit of a sorted list = best fit!
- ▶ Problem: sawdust at beginning of the list
 - ▶ Sorting of list forces a large requests to skip over many small blocks. Need to use a scalable heap organization
- ▶ Suppose memory has free blocks: 
 - ▶ If allocation ops are 10 then 20, best fit wins
 - ▶ When is FF better than best fit?

First fit: Nuances

- ▶ First fit sorted by address order, in practice:
 - ▶ Blocks at front preferentially split, ones at back only split when no larger one found before them
 - ▶ Result? Seems to roughly sort free list by size
 - ▶ So? Makes first fit operationally similar to best fit: a first fit of a sorted list = best fit!
- ▶ Problem: sawdust at beginning of the list
 - ▶ Sorting of list forces a large requests to skip over many small blocks. Need to use a scalable heap organization
- ▶ Suppose memory has free blocks: 
 - ▶ If allocation ops are 10 then 20, best fit wins
 - ▶ When is FF better than best fit?
 - ▶ Suppose allocation ops are 8, 12, then 12 \implies first fit wins

Some worse ideas

Worst-fit:

- ▶ Strategy: fight against sawdust by splitting blocks to maximize leftover size
- ▶ In real life seems to ensure that no large blocks around

Next fit:

- ▶ Strategy: use first fit, but remember where we found the last thing and start searching from there
- ▶ Seems like a good idea, but tends to break down entire list

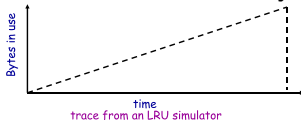
Buddy systems:

- ▶ Round up allocations to power of 2 to make management faster
- ▶ Result? Heavy internal fragmentation

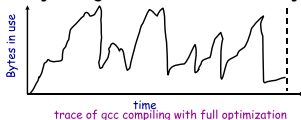
Exploiting program behavior

Known patterns of real programs

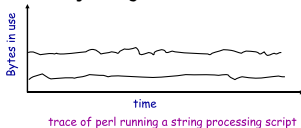
- ▶ So far we've treated programs as black boxes.
- ▶ Most real code exhibit one or more of the following:
 - ▶ *Ramps*: accumulate data monotonically over time



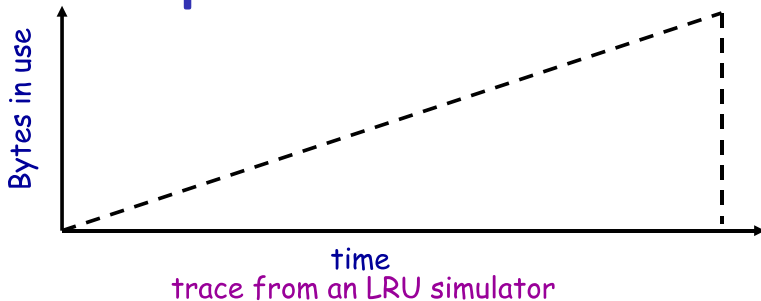
- ▶ *Peaks*: allocate many objects, use briefly, then free all



- ▶ *Plateaus*: allocate many objects, use for a long time



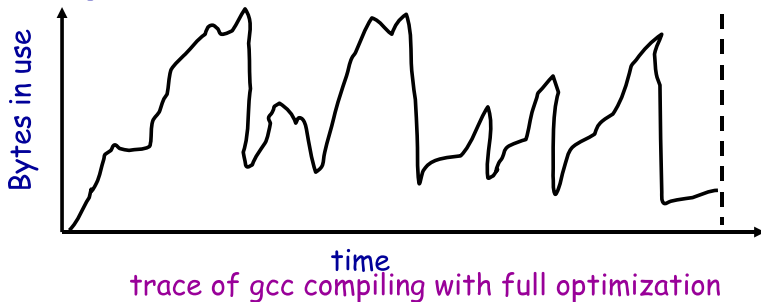
Pattern 1: ramps



In a practical sense: ramp == no calls to `free()`!

- ▶ Implication for fragmentation?
- ▶ What happens if you evaluate allocator with ramp programs only?

Pattern 2: peaks

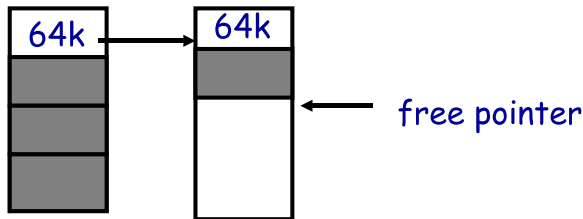


Peaks: allocate many objects, use briefly, then free all

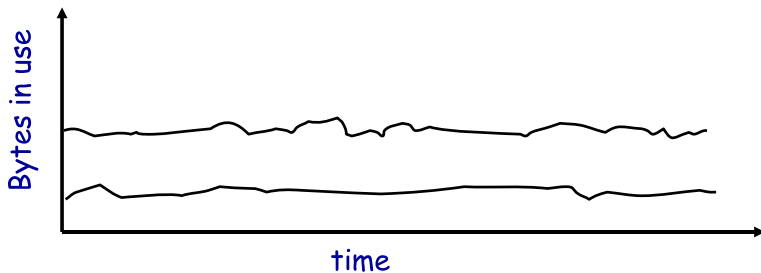
- ▶ Fragmentation a real danger
- ▶ What happens if peak allocated from contiguous memory?
- ▶ Interleave peak & ramp? Interleave two different peaks?

Exploiting peaks

- ▶ Peak phases: allocate a lot, then free everything
 - ▶ Change allocation interface: allocate as before, but only support free of everything all at once
 - ▶ Called “arena allocation”, “obstack” (object stack), or `alloca`/procedure call (by compiler people)
- ▶ Arena = a linked list of large chunks of memory
 - ▶ Advantages: `alloc` is a pointer increment, `free` is “free”
No wasted space for tags or list pointers



Pattern 3: Plateaus



trace of perl running a string processing script

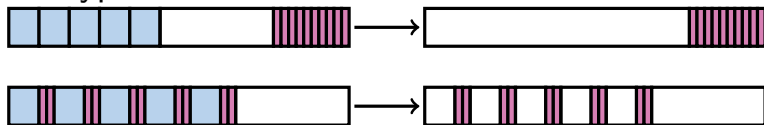
Plateaus: allocate many objects, use for a long time

- What happens if overlap with peak or different plateau?

Fighting fragmentation

Segregation = reduced fragmentation:

- ▶ Allocated at same time ~ freed at same time
- ▶ Different type ~ freed at different time



Implementation observations:

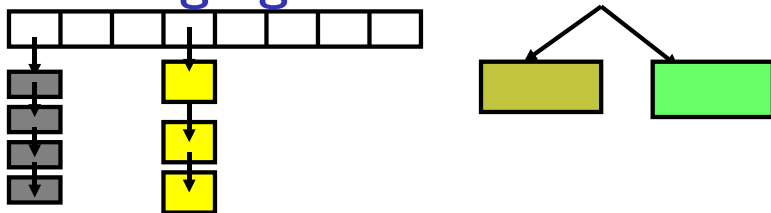
- ▶ Programs allocate a small number of different sizes
- ▶ Fragmentation at peak usage more important than at low usage
- ▶ Most allocations small (< 10 words)
- ▶ Work done with allocated memory increases with size
- ▶ Implications?

Allocator designs

Slab allocation [Bonwick]

- ▶ Kernel allocates many instances of same structures
 - ▶ E.g., a 1.7 kB `task_struct` for every process on system
- ▶ Often want contiguous *physical* memory (for DMA)
- ▶ Slab allocation optimizes for this case:
 - ▶ A **slab** is multiple pages of contiguous physical memory
 - ▶ A **cache** contains one or more slabs
 - ▶ Each cache stores only one kind of object (fixed size)
- ▶ Each slab is **full**, **empty**, or **partial**
- ▶ E.g., need new `task_struct`?
 - ▶ Look in the `task_struct` cache
 - ▶ If there is a partial slab, pick free `task_struct` in that
 - ▶ Else, use empty, or may need to allocate new slab for cache
- ▶ Advantages: speed, and no internal fragmentation

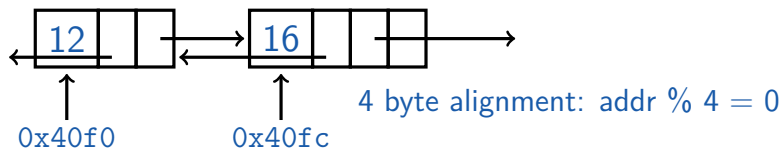
Simple, fast segregated free lists



- ▶ Array of free lists for small sizes, tree for larger
 - ▶ Place blocks of same size on same page
 - ▶ Have count of allocated blocks: if goes to zero, can return page
- ▶ Pro: segregate sizes, no size tag, fast small alloc
- ▶ Con: worst case waste: 1 page per size even w/o free, After pessimal free: waste 1 page per object
- ▶ TCMalloc [Ghemawat] [↗](#) is a well-documented malloc like this
 - ▶ Also uses “thread caching” to reduce coherence misses

Typical space overheads

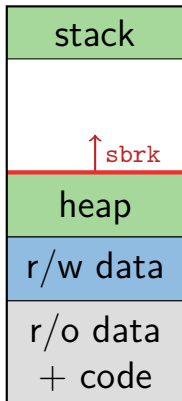
- ▶ Free list bookkeeping and alignment determine minimum allocatable size:
- ▶ If not implicit in page, must store size of block
- ▶ Must store pointers to next and previous freelist element



- ▶ Allocator doesn't know types
 - ▶ Must align memory to conservative boundary
- ▶ Minimum allocation unit? Space overhead when allocated?

Getting more space from OS I

- ▶ On Unix, can use `sbrk`
 - ▶ E.g., to activate a new zero-filled page:



```
/* add nbytes of valid virtual address space */  
void *get_free_space(size_t nbytes) {  
    void *p = sbrk(nbytes);  
    if (p == (void *) -1)  
        error("virtual memory exhausted");  
    return p;  
}
```


Getting more space from OS II

For large allocations, `sbrk` a bad idea

- ▶ May want to give memory back to OS
- ▶ Can't with `sbrk` unless big chunk last thing allocated
- ▶ So allocate large chunk using `mmap`'s `MAP_ANON`

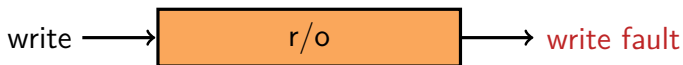
User-level MMU tricks

Faults + resumption = power

- ▶ Resuming after fault lets us emulate many things
 - ▶ “All problems in CS can be solved by another layer of indirection”
- ▶ Example: sub-page protection
- ▶ To protect sub-page region in paging system:



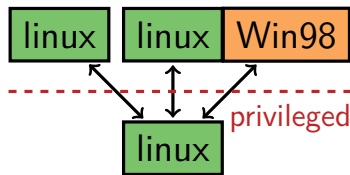
- ▶ Set entire page to most restrictive permission; record in PT



- ▶ Any access that violates permission will cause a fault
- ▶ Fault handler checks if page special, and if so, if access allowed
- ▶ Allowed? Emulate write (“tracing”), otherwise raise error

More fault resumption examples

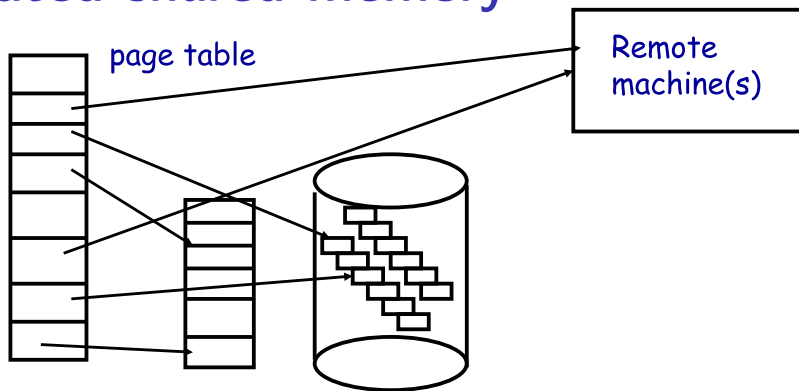
- ▶ Emulate accessed bits:
 - ▶ Set page permissions to “invalid”.
 - ▶ On any access will get a fault: Mark as accessed
- ▶ Avoid save/restore of floating point registers
 - ▶ Make first FP operation cause fault so as to detect usage
- ▶ Emulate non-existent instructions:
 - ▶ Give inst an illegal opcode; OS fault handler detects and emulates fake instruction
- ▶ Run OS on top of another OS!
 - ▶ Slam OS into normal process
 - ▶ When does something “privileged,” real OS gets woken up with a fault.
 - ▶ If operation is allowed, do it or emulate it; otherwise kill guest
 - ▶ IBM's VM/370. Vmware (sort of)



Not just for kernels

- ▶ User-level code can resume after faults, too. Recall:
 - ▶ `mprotect` – protects memory
 - ▶ `sigaction` – catches signal after page fault
 - ▶ Return from signal handler restarts faulting instruction
- ▶ Many applications detailed by [Appel & Li] [↗](#)
- ▶ Example: concurrent snapshotting of process
 - ▶ Mark all of process's memory read-only with `mprotect`
 - ▶ One thread starts writing all of memory to disk
 - ▶ Other thread keeps executing
 - ▶ On fault – write that page to disk, make writable, resume

Distributed shared memory



- ▶ Virtual memory allows us to go to memory or disk
 - ▶ But, can use the same idea to go anywhere! Even to another computer. Page across network rather than to disk. Faster, and allows network of workstations (NOW)

Persistent stores

- ▶ Idea: Objects that persist across program invocations
 - ▶ E.g., object-oriented database; useful for CAD/CAM type apps
- ▶ Achieve by memory-mapping a file
 - ▶ Write your own “malloc” for memory in a file
- ▶ But only write changes to file at end if commit
 - ▶ Use dirty bits to detect which pages must be written out
 - ▶ Or emulate dirty bits with *mprotect/sigaction* (using write faults)
- ▶ On 32-bit machine, store can be larger than memory
 - ▶ But single run of program won't access $> 4\text{GB}$ of objects
 - ▶ Keep mapping of 32-bit memory pointers \leftrightarrow 64-bit disk offsets
 - ▶ Use faults to bring in pages from disk as necessary
 - ▶ After reading page, translate pointers—known as *swizzling*

Garbage collection

Garbage collection

In safe languages, runtime knows about all pointers

- ▶ So can move an object if you change all the pointers

What memory locations might a program access?

- ▶ Any globals or objects whose pointers are currently in registers
- ▶ Recursively, any pointers in objects it might access
- ▶ Anything else is *unreachable*, or *garbage*; memory can be re-used

Garbage collection

Example: stop-and-copy garbage collection

- ▶ Memory full? Temporarily pause program, allocate new heap
- ▶ Copy all objects pointed to by registers into new heap
 - ▶ Mark old copied objects as copied, record new location
- ▶ Start scanning through new heap. For each pointer:
 - ▶ Copied already? Adjust pointer to new location
 - ▶ Not copied? Then copy it and adjust pointer
- ▶ Free old heap—program will never access it—and continue

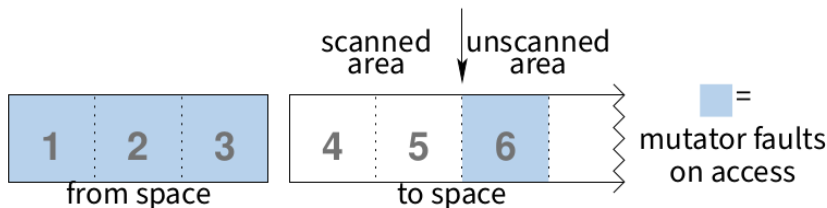
Concurrent garbage collection

Idea: Stop & copy, but without the stop

- ▶ *Mutator* thread runs program, *collector* concurrently does GC

When collector invoked:

- ▶ Protect from space & unscanned to space from mutator
- ▶ Copy objects in registers into *to space*, resume mutator
- ▶ All pointers in scanned *to space* point to *to space*
- ▶ If mutator accesses unscanned area, fault, scan page, resume



(See [Appel & Li] [↗](#).)

Heap overflow detection 1

- ▶ Many GCed languages need fast allocation
 - ▶ E.g., in lisp, constantly allocating cons cells
 - ▶ Allocation can be as often as every 50 instructions
- ▶ Fast allocation is just to bump a pointer

Heap overflow detection 2

```
char *next_free;  
char *heap_limit;  
  
void *alloc (unsigned size) {  
    if (next_free + size > heap_limit) /* 1 */  
        invoke_garbage_collector (); /* 2 */  
    char *ret = next_free;  
    next_free += size;  
    return ret;  
}
```

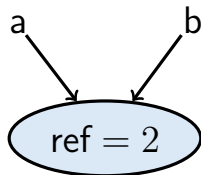
- But would be even faster to eliminate lines 1 & 2!

Heap overflow detection 3

- ▶ Mark page at end of heap inaccessible
 - ▶ `mprotect (heap_limit, PAGE_SIZE, PROT_NONE);`
- ▶ Program will allocate memory beyond end of heap
- ▶ Program will use memory and fault
 - ▶ Note: Depends on specifics of language
 - ▶ But many languages will touch allocated memory immediately
- ▶ Invoke garbage collector
 - ▶ Must now put just allocated object into new heap
- ▶ Note: requires more than just resumption
 - ▶ Faulting instruction must be resumed
 - ▶ But must resume with different target virtual address
 - ▶ Doable on most architectures since GC updates registers

Reference counting 1

- ▶ Seemingly simpler GC scheme:
 - ▶ Each object has “ref count” of pointers to it
 - ▶ Increment when pointer set to it
 - ▶ Decrement when pointer killed
(C++ destructors handy—c.f. `shared_ptr` ↗)



- ▶ Works well for hierarchical data structures
 - ▶ E.g., pages of physical memory

Reference counting 2

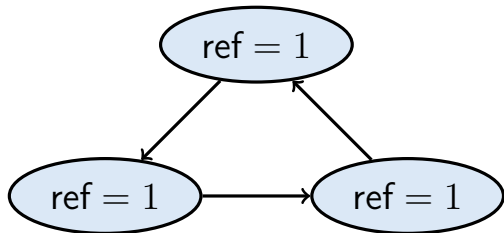
```
void foo(bar c) {  
    bar a, b;  
    a = c;           // c.refcnt++  
    b = a;           // a.refcnt++  
    a = nullptr;    // c.refcnt--  
    return;         // b.refcnt--  
}
```

- ref count == 0? Free object

Reference counting pros/cons 1

Circular data structures always have ref count > 0

- ▶ No external pointers means **lost memory**



- ▶ Python GC summary [↗](#)

Reference counting pros/cons 2

- ▶ Can do manually w/o PL support, but error-prone
- ▶ Potentially more efficient than real GC
 - ▶ No need to halt program to run collector
 - ▶ Avoids weird unpredictable latencies
- ▶ Potentially less efficient than real GC
 - ▶ With real GC, copying a pointer is cheap
 - ▶ With real GC, object destructors can run asynchronously
 - ▶ With refcounts, must update count each time & possibly take lock (but C++11's `std::move` can avoid overhead)
 - ▶ With refcounts, destructors can introduce unexpected latencies

Ownership types

- ▶ Another approach: avoid GC by exploiting type system
 - ▶ Use ownership types, which prohibit copies
- ▶ You can move a value into a new variable (e.g., copy pointer)
 - ▶ But then the original variable is no longer usable
- ▶ You can *borrow* a value by creating a pointer to it
 - ▶ But must prove pointer will not outlive borrowed value
 - ▶ And can't use original unless both are read-only (to avoid races)
- ▶ Ownership types available now in Rust [↗](#) language
 - ▶ First serious competitor to C/C++ for OSes, browser engines
- ▶ C++11 does something similar but weaker with unique types
 - ▶ `std::unique_ptr` [↗](#), `std::weak_ptr` [↗](#),...
 - ▶ Can `std::move` [↗](#) but not copy these