Lecture 14: Dynamic Memory

Instructor: Mitch Neilsen

Office: N219D

Quote of the Day

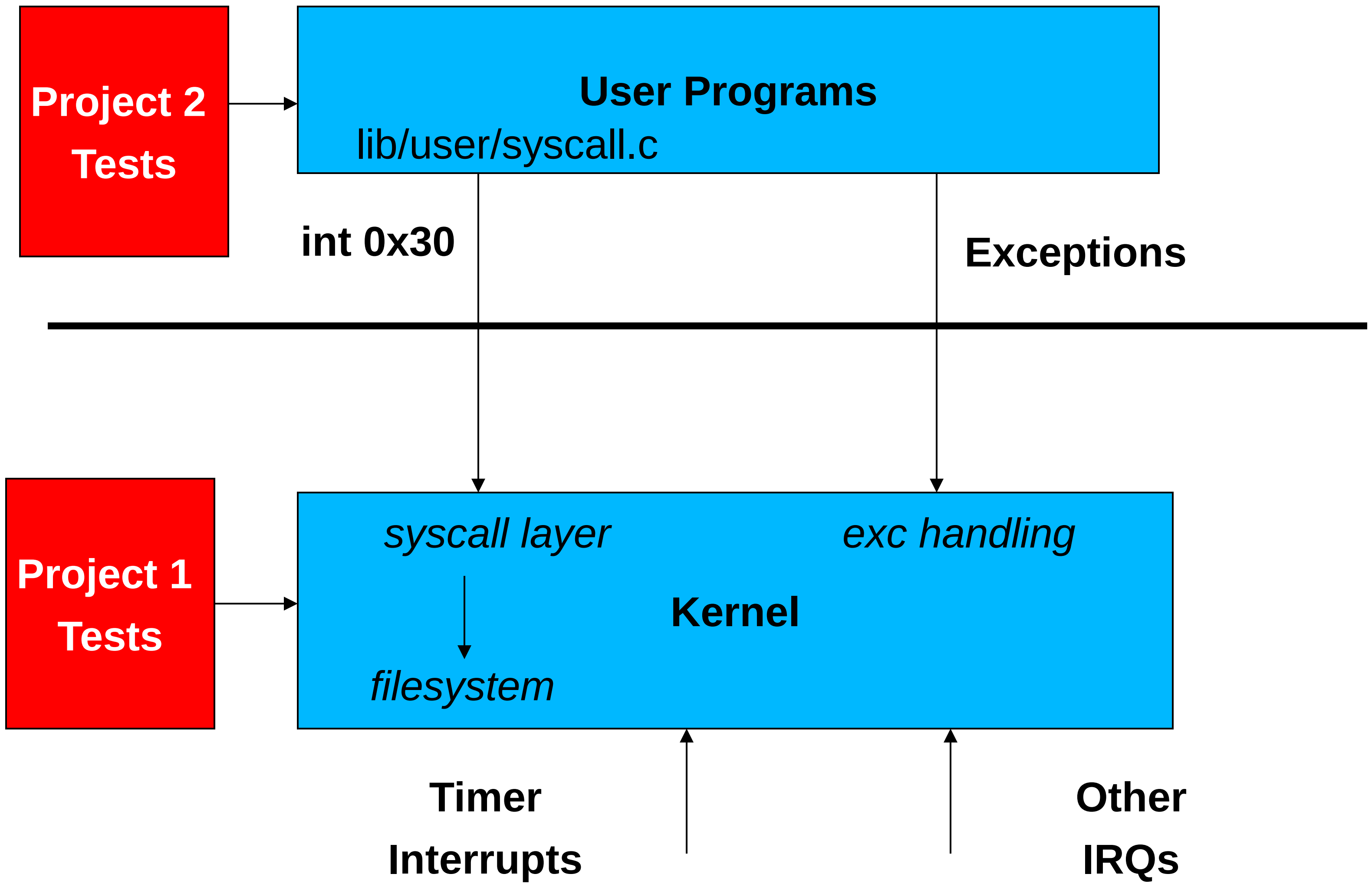
"I have not failed. I've just found 10,000 ways that won't work."

-- Thomas Edison

Project 2

- System Call: void halt(void)
- TYPO: `power_off()` should be `shutdown_power_off()`.

Project 1 and Project 2



When does a process need to access Operating System functionality?

- Here are several examples:
  - Reading a file. The OS must perform the file system operations required to read the data off of disk.
  - Creating a child process. The OS must set stuff up for the child process.

How do processes invoke OS functionality?

- By making a system call.
  - Conceptually, processes call a subroutine that goes off and performs the required functionality. But OS must execute in supervisor mode, which allows it to do things like manipulate the disk directly.
  - To switch from normal user mode to supervisor mode, most machines provide a system call instruction.
    - **This instruction causes an exception to take place.**
    - **The hardware switches from user mode to supervisor mode and invokes the exception handler inside the operating system.**
  - There is typically some kind of convention that the process uses to interact with the OS.

Let's do an example - Open() system call

- System calls typically start out with a normal subroutine call.

```
int handle = open("sample.txt");
```

- **open()** executes a syscall instruction, which generates a system call exception 0x30.

```
syscall1 (SYS_OPEN, file);
```

- By convention, the Open subroutine puts a number on the stack to indicate which routine (SYS_OPEN = 6) should be invoked.
  - Inside the exception handler the OS looks at the stack to figure out what system call it should perform.
- The Open system call also takes a parameter. By convention, the compiler also puts this (e.g., a pointer to the filename) on the stack.
  - More conventions: return values are put into the %EAX register.
- Inside the exception handler, the OS figures out what action to take, performs the action, then returns back to the user program.

SYS_OPEN is defined in lib/syscall-nr.h

```
#ifndef __LIB_SYSCALL_NR_H
#define __LIB_SYSCALL_NR_H
/* System call numbers. */
enum
  {
    /* Projects 2 and later. */
    SYS_HALT,                   /* Halt the operating system. */
    SYS_EXIT,                   /* Terminate this process. */
    SYS_EXEC,                   /* Start another process. */
    SYS_WAIT,                   /* Wait for a child process to die. */
    SYS_CREATE,                 /* Create a file. */
    SYS_REMOVE,                 /* Delete a file. */
    SYS_OPEN,                   /* Open a file. */
…
```

Thus, SYS_OPEN = 6.

Open( ) system call details

In pintos/src/lib/user/syscall.c:

```
int
open (const char *file)
{
  return syscall1 (SYS_OPEN, file);
}
.. (and above)..

/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an `int'. */
#define syscall1(NUMBER, ARG0)                                     \
    ({                                                             \
      int retval;                                                  \
      asm volatile                                                 \
        ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
           : "=a" (retval)                                         \
           : [number] "i" (NUMBER),                                \
             [arg0] "g" (ARG0)                                     \
           : "memory");                                            \
      retval;                                                      \
```

Initialize syscall handler

```
void
syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
  lock_init (&fs_lock);
}
```

Add sys_open to syscall_handler

```
static void
syscall_handler (struct intr_frame *f)
{
  typedef int syscall_function (int, int, int);

  /* A system call. */
  struct syscall
  {
      size_t arg_cnt;             /* Number of arguments. */
      syscall_function *func;     /* Implementation. */
  };

  /* Table of system calls. */
  static const struct syscall syscall_table[] =
  {
      {0, (syscall_function *) sys_halt},
      {1, (syscall_function *) sys_exit},
      {1, (syscall_function *) sys_exec},
      {1, (syscall_function *) sys_wait},
      {2, (syscall_function *) sys_create},
      {1, (syscall_function *) sys_remove},
      {1, (syscall_function *) sys_open},
      …
  }
```

Add sys_open to syscall_handler

```
  const struct syscall *sc;
  unsigned call_nr;
  int args[3];
  /* Get the system call. */
  copy_in (&call_nr, f->esp, sizeof call_nr);
  if (call_nr >= sizeof syscall_table / sizeof *syscall_table)
      thread_exit ();
  sc = syscall_table + call_nr;

  /* Get the system call arguments. */
  ASSERT (sc->arg_cnt <= sizeof args / sizeof *args);
  memset (args, 0, sizeof args);
  copy_in (args, (uint32_t *) f->esp + 1, sizeof *args * sc->arg_cnt);

  /* Execute the system call,
     and set the return value. */
  f->eax = sc->func (args[0], args[1], args[2]);
}
```

Add sys_open function to syscall.c

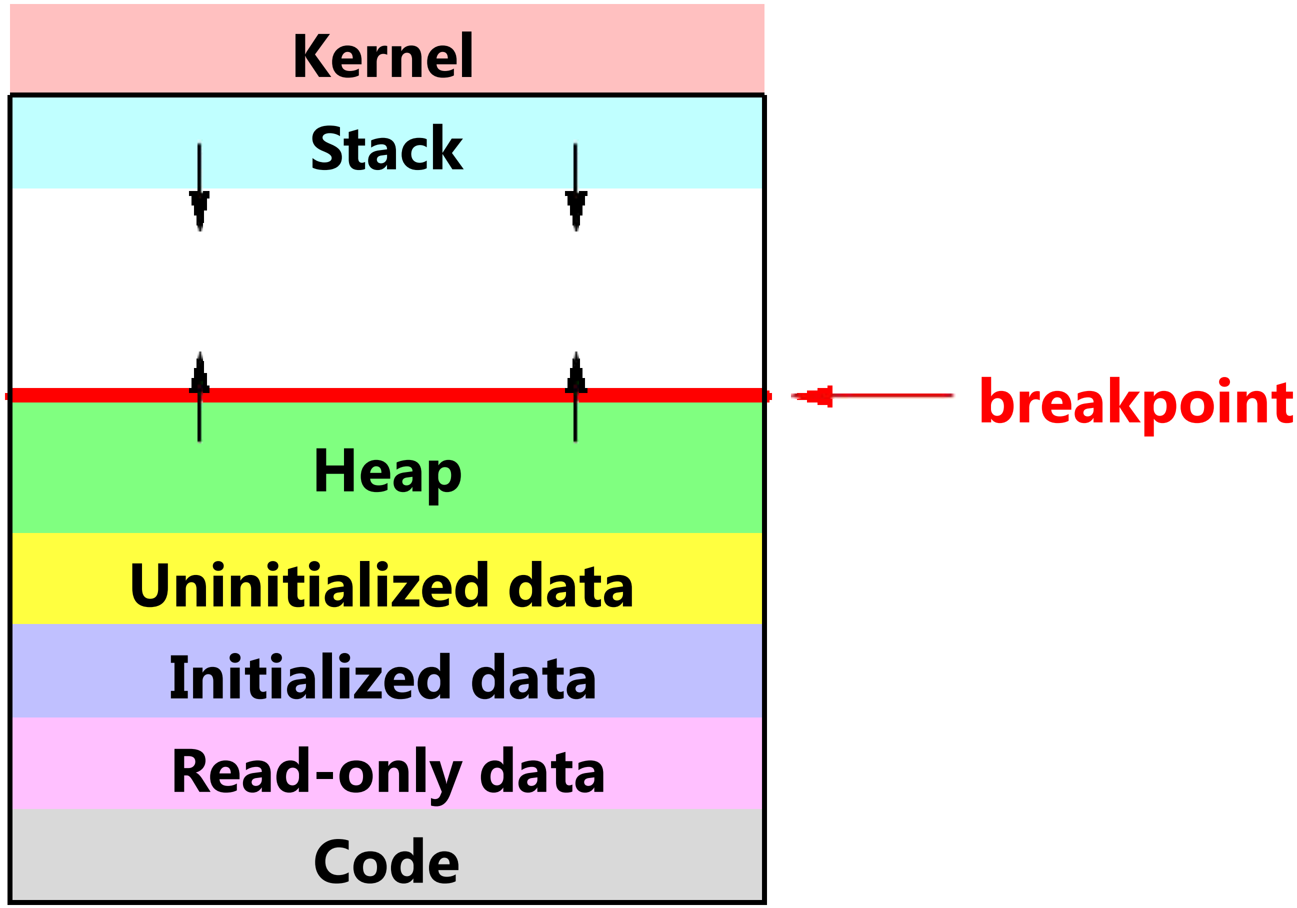
```
sys_open (const char *ufile)
{
  char *kfile = copy_in_string (ufile);
  struct file_descriptor *fd;
  int handle = -1;

  fd = malloc (sizeof *fd);
  if (fd != NULL)
  {
      lock_acquire (&fs_lock);
      fd->file = filesys_open (kfile);
      if (fd->file != NULL)
      {
          struct thread *cur = thread_current ();
          handle = fd->handle = cur->next_handle++;
          list_push_front (&cur->fds, &fd->elem);
      }
      else
          free (fd);
      lock_release (&fs_lock);
  }
  palloc_free_page (kfile);
  return handle;
}
```

```c
/* Creates a copy of user string US in kernel memory and returns it
   as a page that must be freed with palloc_free_page().
   Truncates the string at PGSIZE bytes in size. */
static char *
copy_in_string (const char *us)
{
  char *ks;
  size_t length;
  ks = palloc_get_page (PAL_ASSERT | PAL_ZERO);
  if (ks == NULL)
      thread_exit ();
  for (length = 0; length < PGSIZE; length++)
  {
      if (us >= (char *) PHYS_BASE || !get_user (ks + length, us++))
      {
          palloc_free_page (ks);
          thread_exit ();
      }
      if (ks[length] == '\0')
          return ks;
  }
  ks[PGSIZE - 1] = '\0';
  return ks;
}
```

Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- **Memory-Mapped Files**
- **Allocating Kernel Memory**
- **Other Considerations**
- **Operating-System Examples**

Recall typical virtual address space

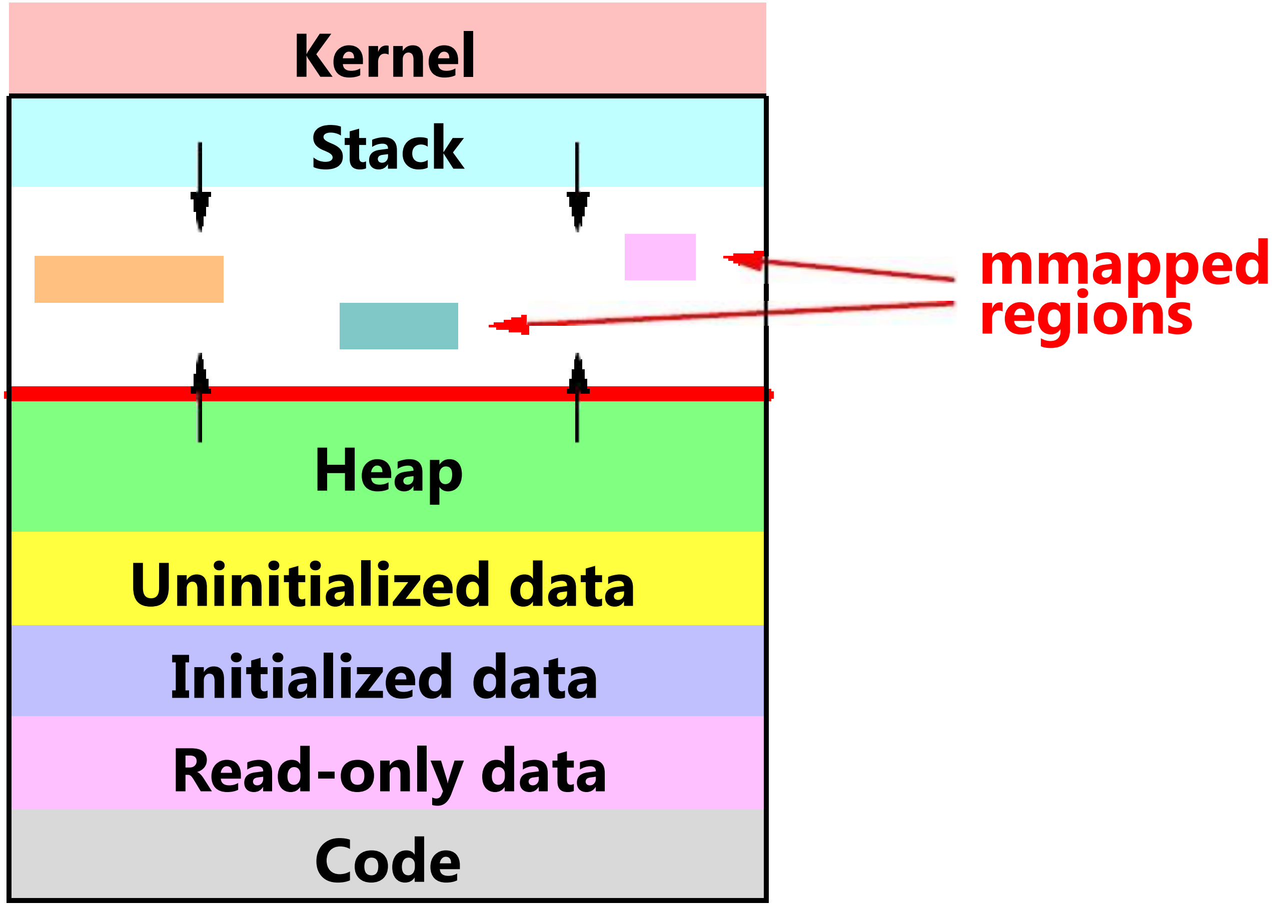


- **Dynamically allocated memory goes in heap**
- **Top of heap called *breakpoint***
  - Addresses between breakpoint and stack all invalid

Early VM system calls

- **OS keeps "Breakpoint" – top of heap**
  - Memory regions between breakpoint & stack fault on access
- char *brk (const char *addr);
  - Set and return new value of breakpoint
- char *sbrk (int incr);
  - Increment value of the breakpoint & return old value
- **Can implement** malloc **in terms of** sbrk
  - But hard to "give back" physical memory to system

Memory mapped files



- Other memory objects between heap and stack

mmap **system call**

- void *mmap (void *addr, size t len, int prot,
  int flags, int fd, off t_offset)
  - Map file specified by fd at virtual address addr
  - If addr is NULL, let kernel choose the address
- prot **– protection of region**
  - OR of PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE
- flags
  - MAP_ANON – anonymous memory (fd should be -1)
  - MAP_PRIVATE – modifications are private
  - MAP_SHARED – modifications seen by everyone

More VM system calls

- int msync(void *addr, size_t len, int flags);
  - Flush changes of mmapped file to backing store
- int munmap(void *addr, size_t len)
  - Removes memory-mapped object
- int mprotect(void *addr, size_t len, int prot)
  - Changes protection on pages
- int mincore(void *addr, size_t len, char *vec)
  - Returns in vec which pages are present

Exposing page faults

```
struct sigaction {
    union {                          /* signal handler */
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);
    };
    sigset_t sa_mask;            /* signal mask to apply */
    int sa_flags;
};

int sigaction (int sig, const struct sigaction *act,
                              struct sigaction *oact)
```

- **Can specify function to run on** SIGSEGV
  (Unix signal raised on invalid memory access)

Example: OpenBSD/i386 siginfo

```
struct   sigcontext {
   int sc_gs; int sc_fs; int sc_es; int sc_ds;
   int sc_edi; int sc_esi; int sc_ebp; int sc_ebx;
   int sc_edx; int sc_ecx; int sc_eax;

   int sc_eip; int sc_cs; /* instruction pointer */
   int sc_eflags;         /* condition codes, etc. */
   int sc_esp; int sc_ss; /* stack pointer */

   int sc_onstack;        /* sigstack state to restore */
   int sc_mask;           /* signal mask to restore */

   int sc_trapno;
   int sc_err;
};
```

VM tricks at user level

- **Combination of** mprotect**/**sigaction **very powerful**
  - Can use OS VM tricks in user-level programs [Appel]
  - E.g., fault, unprotect page, return from signal handler
- **Technique used in object-oriented databases**
  - Bring in objects on demand
  - Keep track of which objects may be dirty
  - Manage memory as a cache for much larger object DB
- **Other interesting applications**
  - Useful for some garbage collection algorithms
  - Snapshot processes (copy on write)

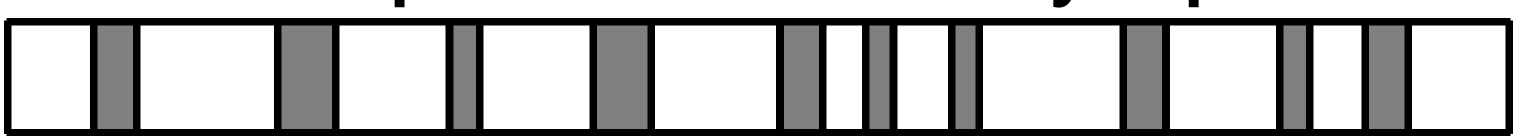
Dynamic memory allocation

- **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
- **Today: how to implement it**
  - Lecture draws on [Wilson] (good survey from 1995)
- **Some interesting facts:**
  - Two or three line code change can have huge, non-obvious impact on how well an allocator works (examples to come)
  - Proven: impossible to construct an "always good" allocator
  - Surprising result: after 35 years, memory management still poorly understood

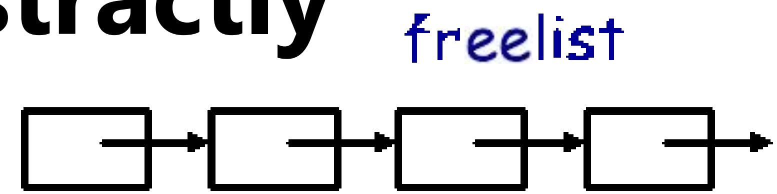
Why is it hard?

- Satisfy arbitrary set of allocation and free's.
- Easy without free: set a pointer to the beginning of some big chunk of memory ("heap") and increment on each allocation:

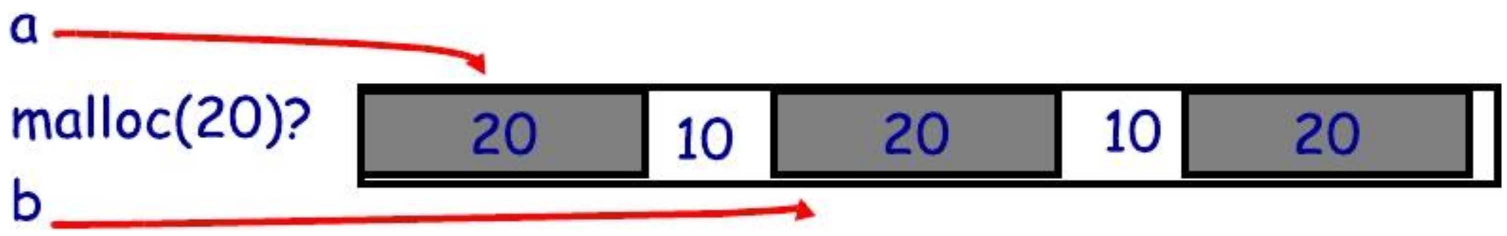
- Problem: free creates holes ("fragmentation") Result? Lots of free space but cannot satisfy request!



More abstractly



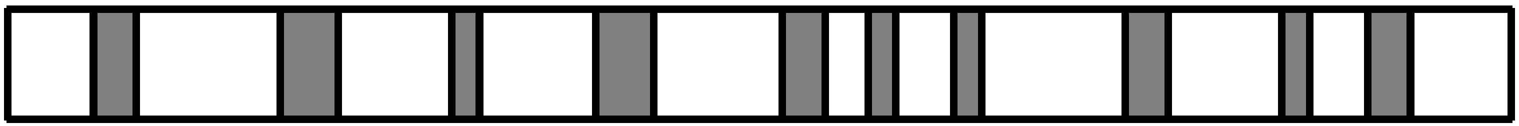
- **What an allocator must do:**
  - Track which parts of memory in use, which parts are free
  - Ideal: no wasted space, no time overhead
- **What the allocator cannot do:**
  - Control order of the number and size of requested blocks
  - Change user ptrs ==⇒ (bad) placement decisions permanent



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

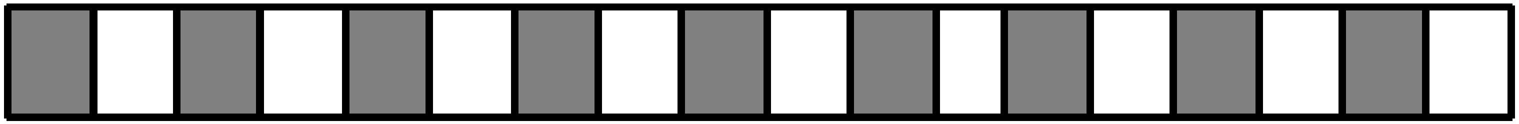
What is fragmentation really?

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



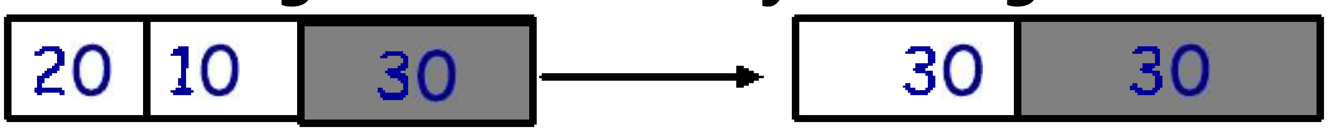
  - If they die at the same time, then no fragmentation:

  - Different sizes: If all requests the same size, then no fragmentation (that’s why no external fragmentation w. 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 chose any larger block to split
  - One way: chose block with smallest remainder (best fit)
- **Coalescing free blocks to yield larger blocks**



  - Freedom: when to coalesce (deferring can be good) 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: $M \cdot (n_{\max} / n_{\min})$
    (only ever uses a memory location for a single size)
  - Good allocator: $\sim M \cdot \log(n_{\max} / n_{\min})$

Pathological examples

- **Given allocation of 7 20-byte chunks**

  - What's a bad stream of frees and then allocates?

- **Given a 128-byte limit on malloced 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" = ~20% fragmentation under many workloads

Pathological examples

- **Given allocation of 7 20-byte chunks**

  - What’s a bad stream of frees and then allocates?
  - **Free every other chunk, then alloc 21 bytes**
- **Given a 128-byte limit on malloced 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” = ~20% fragmentation under many workloads

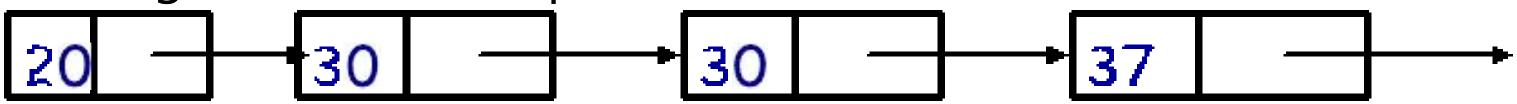
Pathological examples

- **Given allocation of 7 20-byte chunks**

  - What’s a bad stream of frees and then allocates?
  - **Free every other chunk, then alloc 21 bytes**
- **Given a 128-byte limit on malloced 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” = ~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 pointers to next



  - Code: Search freelist for block closest in size to the request. (Exact match is ideal)
  - During free (usually) coalesce adjacent blocks
- **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 ns, then try to allocate an $n + 1$**
- **Example: start with 100 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 (though the way real programs behave suggest it easily could)**

First fit

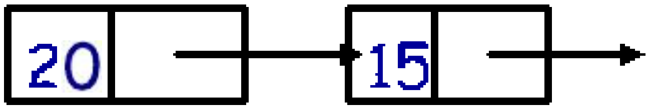
- **Strategy: pick the first block that fits**
  - Data structure: free list, sorted lifo, fifo, or by address
  - Code: scan list, take the first one
- **LIFO: put free object on front of list.**
  - Simple, but causes higher fragmentation
  - Potentially good for cache locality
- **Address sort: order 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 on 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

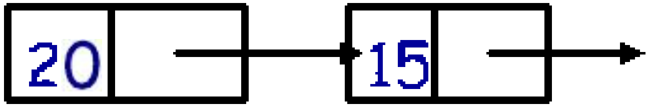
First fit: Nuances

- **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 ⇒ first fit wins**

First/best fit: weird parallels

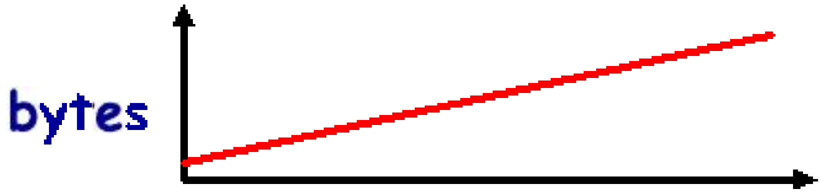
- **Both seem to perform roughly equivalently**
- **In fact the placement decisions of both are roughly identical under both randomized and real workloads!**
  - No one knows why
  - Pretty strange since they seem pretty different
- **Possible explanations:**
  - First fit like best fit because over time its free list becomes sorted by size: the beginning of the free list accumulates small objects and so fits tend to be close to best
  - Both have implicit “open space heuristic” try not to cut into large open spaces: large blocks at end only used when have to be (e.g., first fit: skips over all smaller blocks)

Some worse ideas

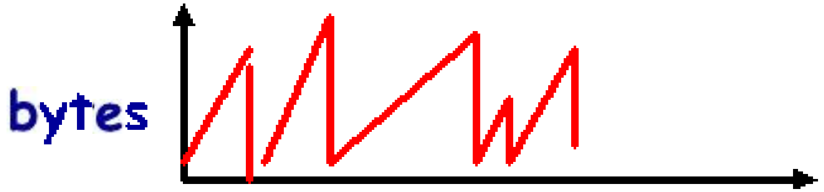
- **Worst-fit:**
  - Strategy: fight against sawdust by splitting blocks to maximize leftover size
  - In real life seems to ensure that no large blocks around
- **Next fit:**
  - Strategy: use first fit, but remember where we found the last thing and start searching from there
  - Seems like a good 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

Known patterns of real programs

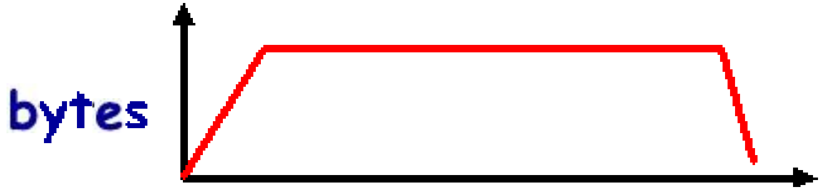
- **So far we’ve treated programs as black boxes.**
- **Most real programs exhibit 1 or 2 (or all 3) of the following patterns of alloc/dealloc:**
  - *Ramps*: accumulate data monotonically over time



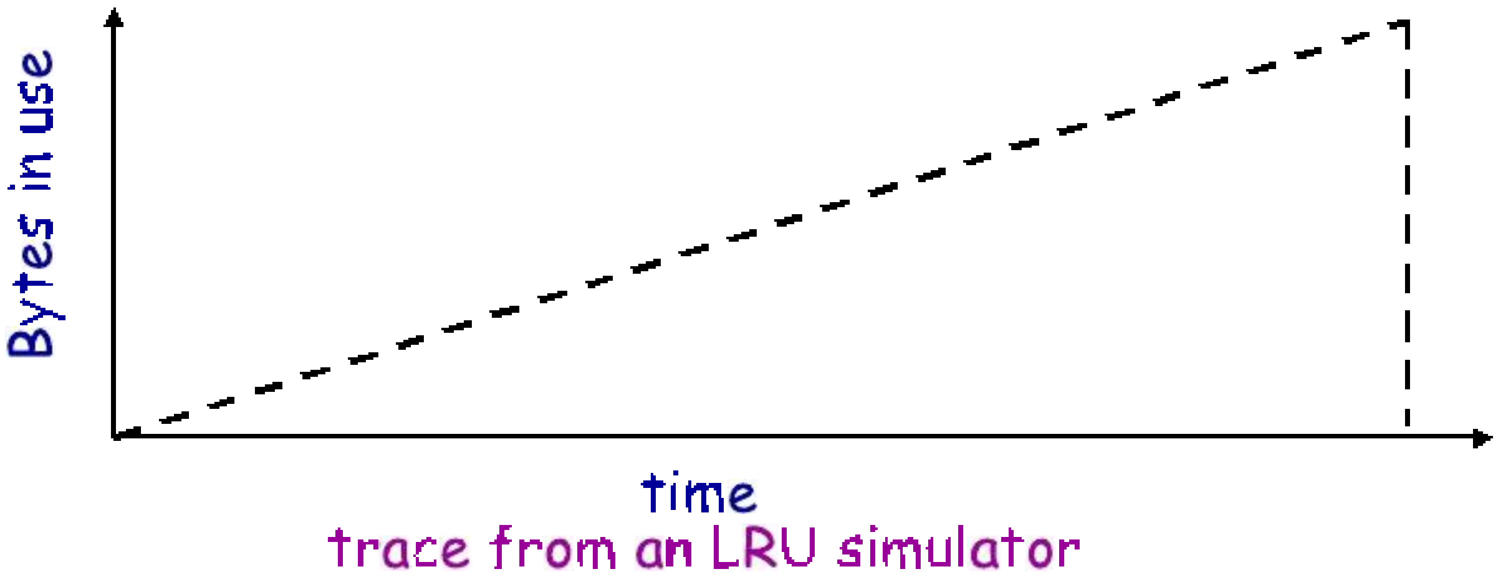
  - *Peaks*: allocate many objects, use briefly, then free all



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



Pattern 1: ramps



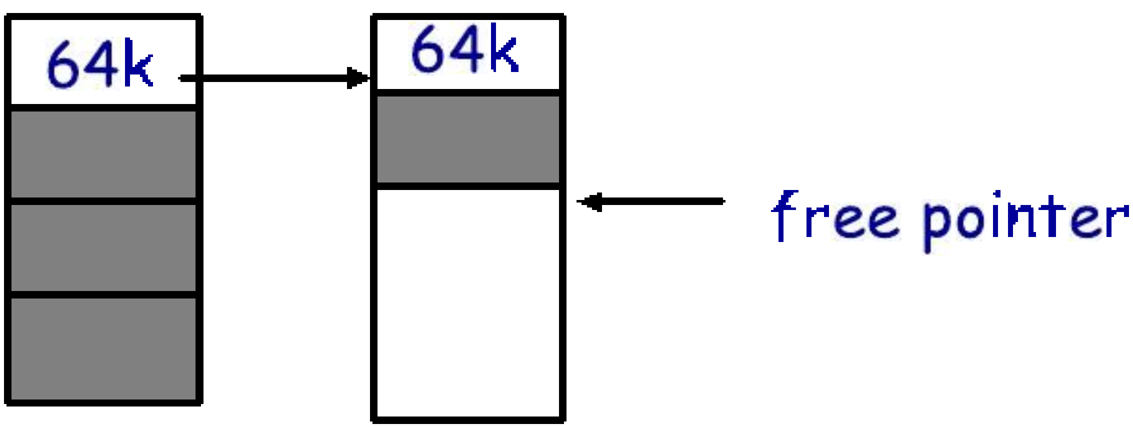
- **In a practical sense: ramp = no free!**
  - Implication for fragmentation?
  - What happens if you evaluate allocator with ramp programs only?

Pattern 2: peaks

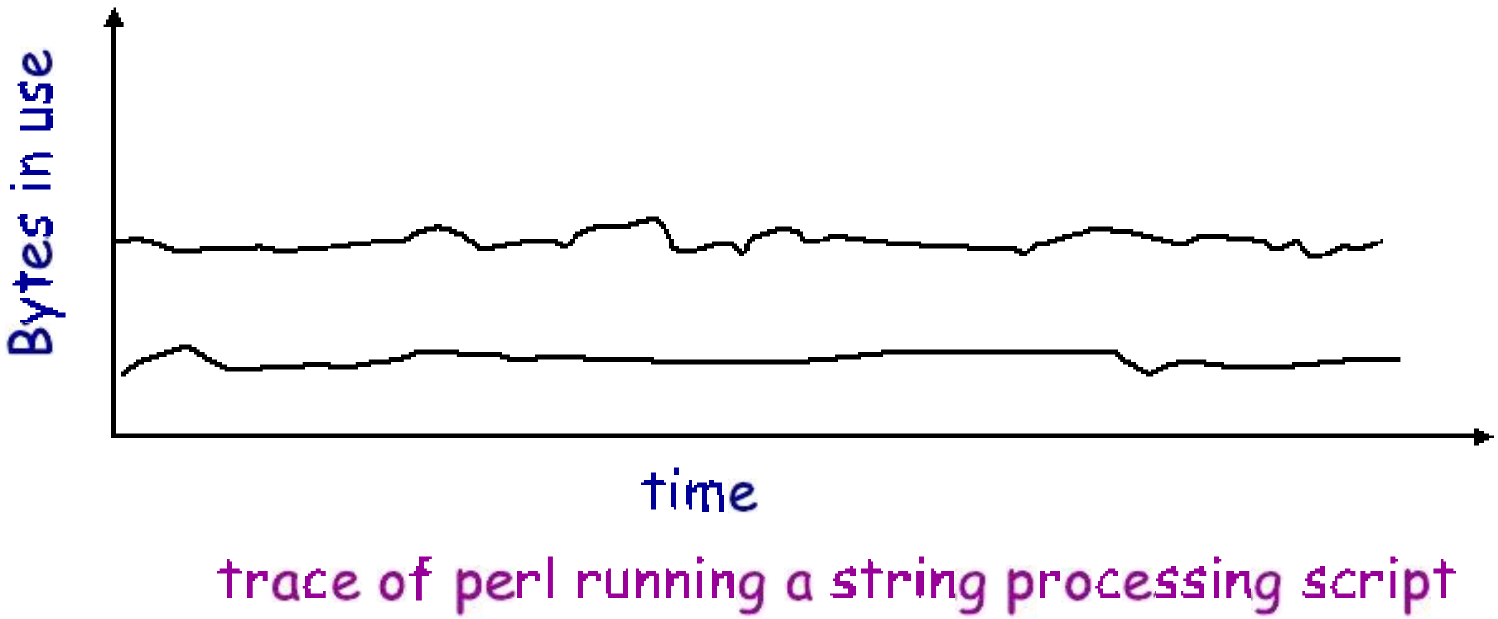
- **Peaks: allocate many objects, use briefly, then free all**
  - Fragmentation a real danger
  - What happens if peak allocated from contiguous memory?
  - Interleave peak & ramp? Interleave two different peaks?

Exploiting peaks

- **Peak phases: alloc a lot, then free everything**
  - So have new allocation interface: alloc as before, but only support free of everything
  - 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



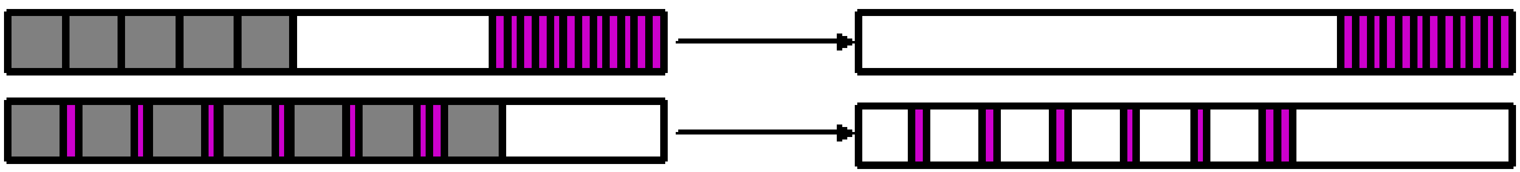
Pattern 3: Plateaus



- **Plateaus: allocate many objects, use for a long time**
  - What happens if overlap with peak or different plateau?

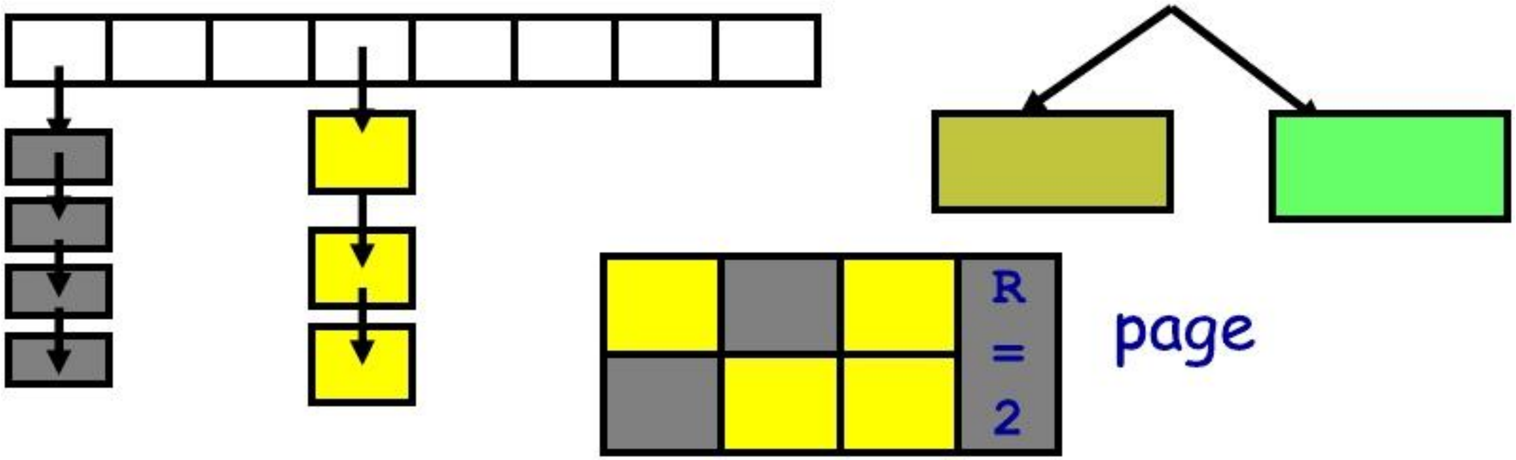
Fighting fragmentation

- **Segregation = reduced fragmentation:**
  - Allocated at same time ~ freed at same time
  - Different type ~ freed at different time



- **Implementation observations:**
  - Programs allocate small number of different sizes
  - Fragmentation at peak use more important than at low
  - Most allocations small (< 10 words)
  - Work done with allocated memory increases with size
  - Implications?

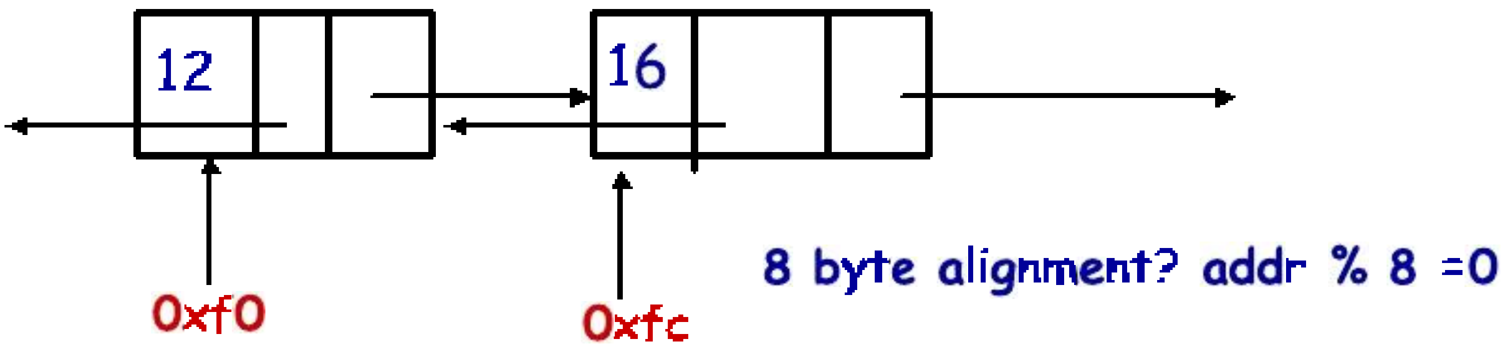
Simple, fast segregated free lists



- **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**

Typical space overheads

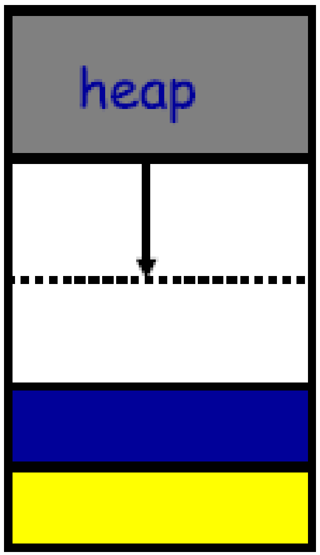
- **Free list bookkeeping + alignment determine minimum allocatable size:**
  - Store size of block
  - Pointers to next and previous freelist element



  - Machine enforced overhead: alignment. Allocator doesn't know type. Must align memory to conservative boundary
  - Minimum allocation unit? Space overhead when allocated?

Getting more space from OS

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



sbrk(4096)

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

- **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

Faults + resumption = power

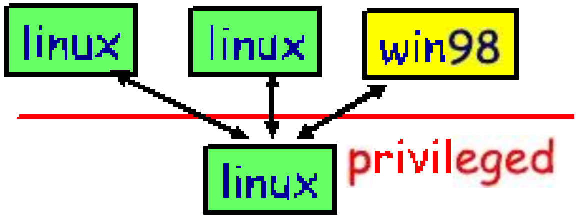
- **Resuming after fault lets us emulate many things**
  - "every problem can be solved with layer of indirection"
- **Example: sub-page protection**
- **To protect sub-page region in paging system:**

  - Set entire page to weakest permission; record in PT

  - Any access that violates perm will cause an access fault
  - Fault handler checks if page special, and if so, if access allowed. Continue or raise error, as appropriate

More fault resumption examples

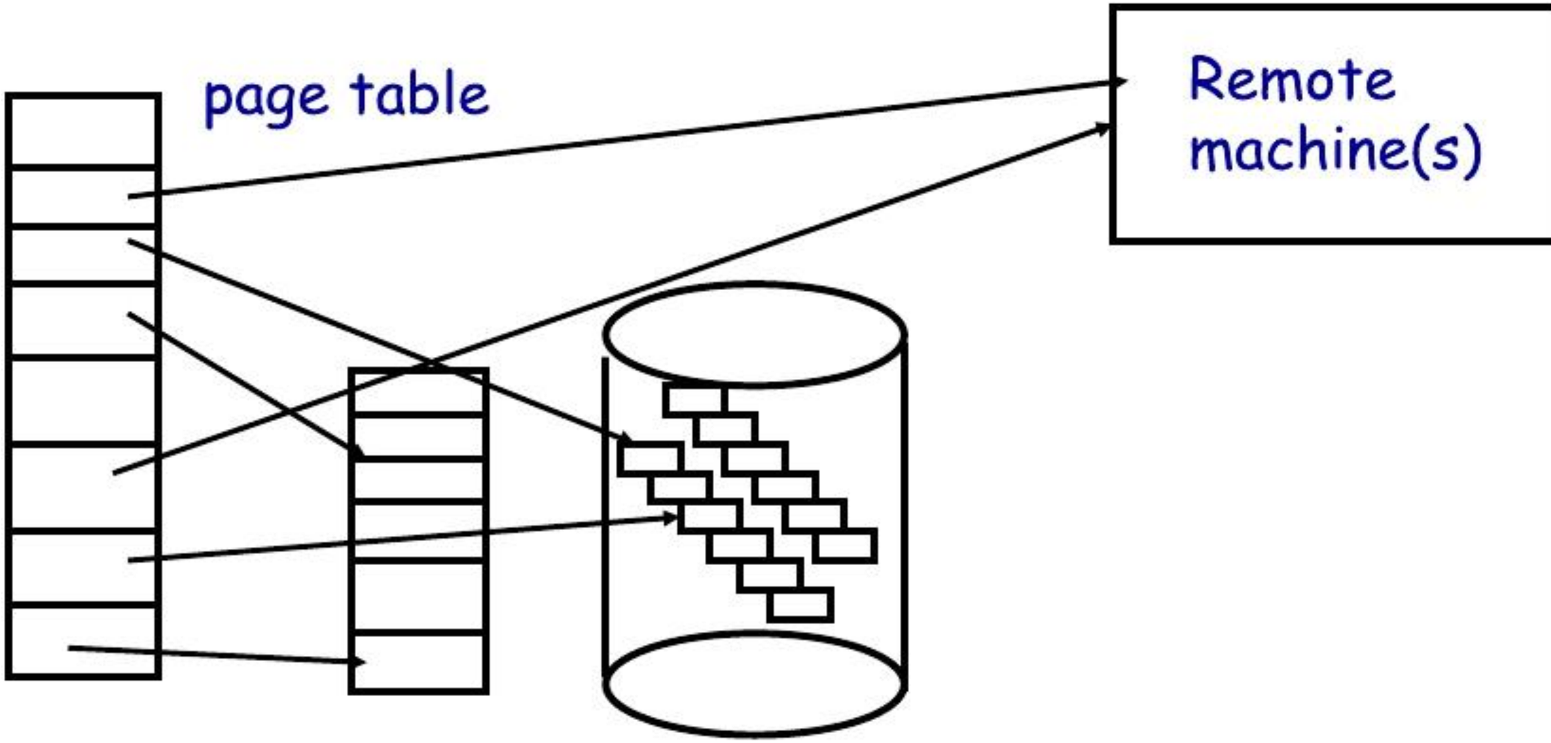
- **Emulate accessed bits:**
  - Set page permissions to “invalid”.
  - On any access will get a fault: Mark as accessed
- **Avoid save/restore of FP registers**
  - Make first FP operation fault to detect usage
- **Emulate non-existent instructions:**
  - Give inst an illegal opcode; OS fault handler 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 op allowed, do it, otherwise kill.
  - IBM’s VM/370. VMware (sort of)



Not just for kernels

- **User-level code can resume after faults, too**
- 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 w. 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

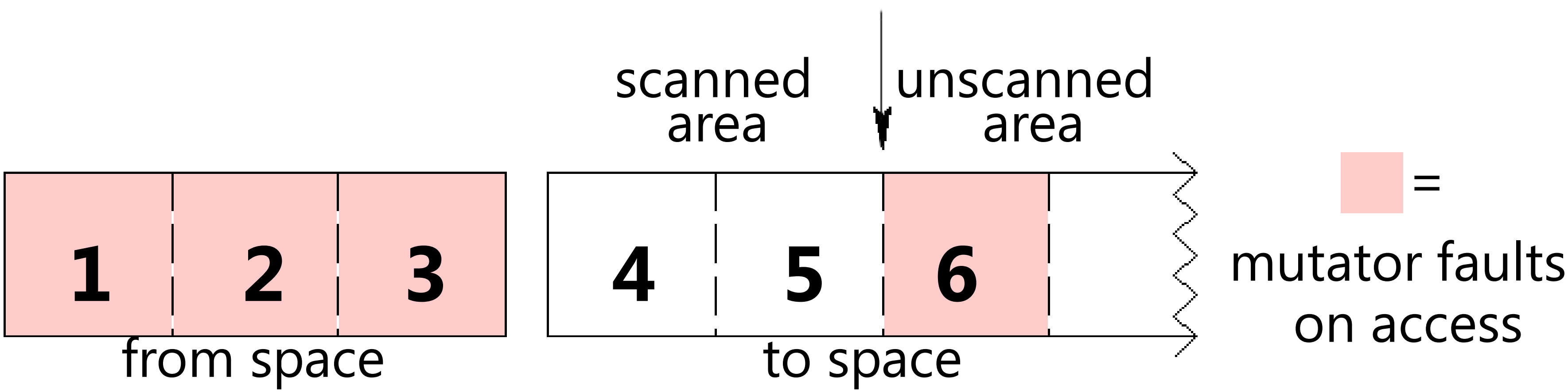
- **Idea: Objects that persist across program invocations**
  - E.g., object-oriented database; useful for CAD/CAM type apps
- **Achieve by memory-mapping a file**
- **But only write changes to file at end if commit**
  - Use dirty bits to detect which pages must be written out
  - Or emulate dirty bits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

Garbage collection

- **In safe languages, run time knows about all pointers**
  - So can move an object if you change all the pointers
- **What memory locations might a program access?**
  - Any 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
- **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



Heap overflow detection

- **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**

```
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;
}
```

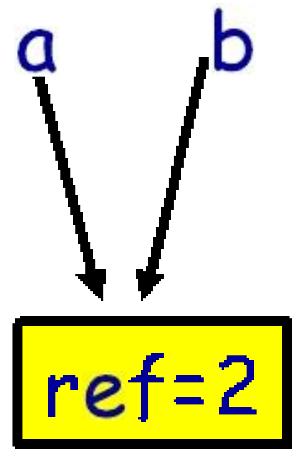
- **But would be even faster to eliminate lines 1 & 2!**

Heap overflow detection 2

- **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

- **Seemingly simpler GC scheme:**
  - Each object has "ref count" of pointers to it
  - Increment when pointer set to it
  - Decremented when pointer killed (C++ destructors handy for such "smart pointers")

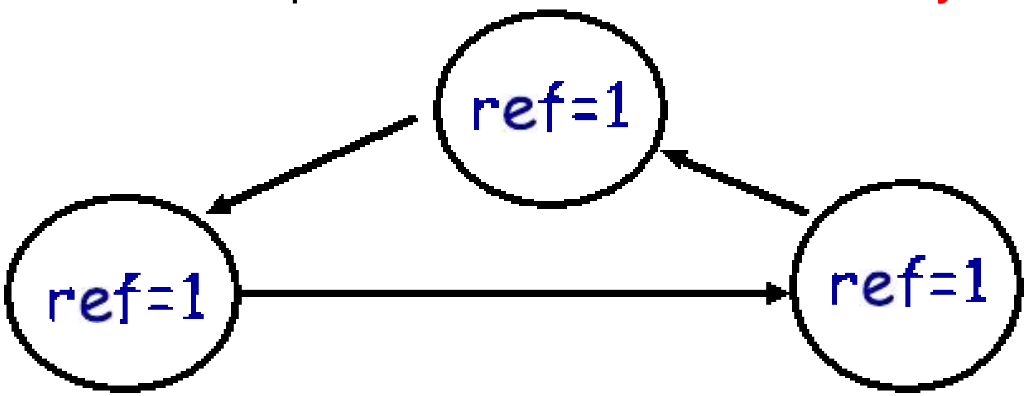



```
void foo(bar c) {
    bar a, b;
    a = c;        ........................ c->refcnt++;
    b = a;        ........................ a->refcnt++;
    a = 0;        ........................ c->refcnt--;
    return;       ........................ b->refcnt--;
}
```

  - ref count == 0? Free object
- **Works well for hierarchical data structures**
  - E.g., pages of physical memory

Reference counting pros/cons

- **Circular data structures always have ref count > 0**
  - No external pointers means lost memory



- **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 reference counting, must write ref count each time

Summary

- Read Ch. 1-8
- Processes and Threads (Ch. 4)
- Process Scheduling (Ch. 5)
- Synchronization (Ch. 6)
- Deadlock (Ch. 7)
- Memory Management (Ch. 8)
- Virtual Memory (Ch. 9)
- Project #2 – System Calls and User-Level Processes