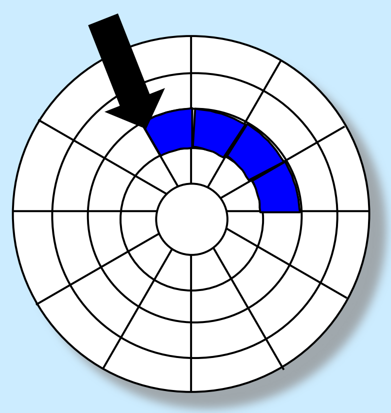
CS 33

Memory Hierarchy III

Reading a File on a Rotating Disk

- Suppose the data of a file are stored on consecutive disk sectors on one track
  - this is the best possible scenario for reading data quickly
    - » single seek required
    - » single rotational delay
    - » all sectors read in a single scan



Quiz 1

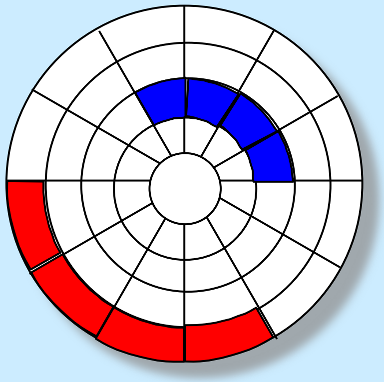
We have two files on the same (rotating) disk. The first file's data resides in consecutive sectors on one track, the second in consecutive sectors on another track. It takes a total of *t* seconds to read all of the first file then all of the second file.

Now suppose the files are read concurrently, perhaps a sector of the first, then a sector of the second, then the first, then the second, etc. Compared to reading them sequentially, this will take

a) less time

b) much more time

c) about the same amount of time (within a factor of 2)



Quiz 2

We have two files on the same solid-state disk. Each file's data resides in consecutive blocks. It takes a total of *t* seconds to read all of the first file then all of the second file.

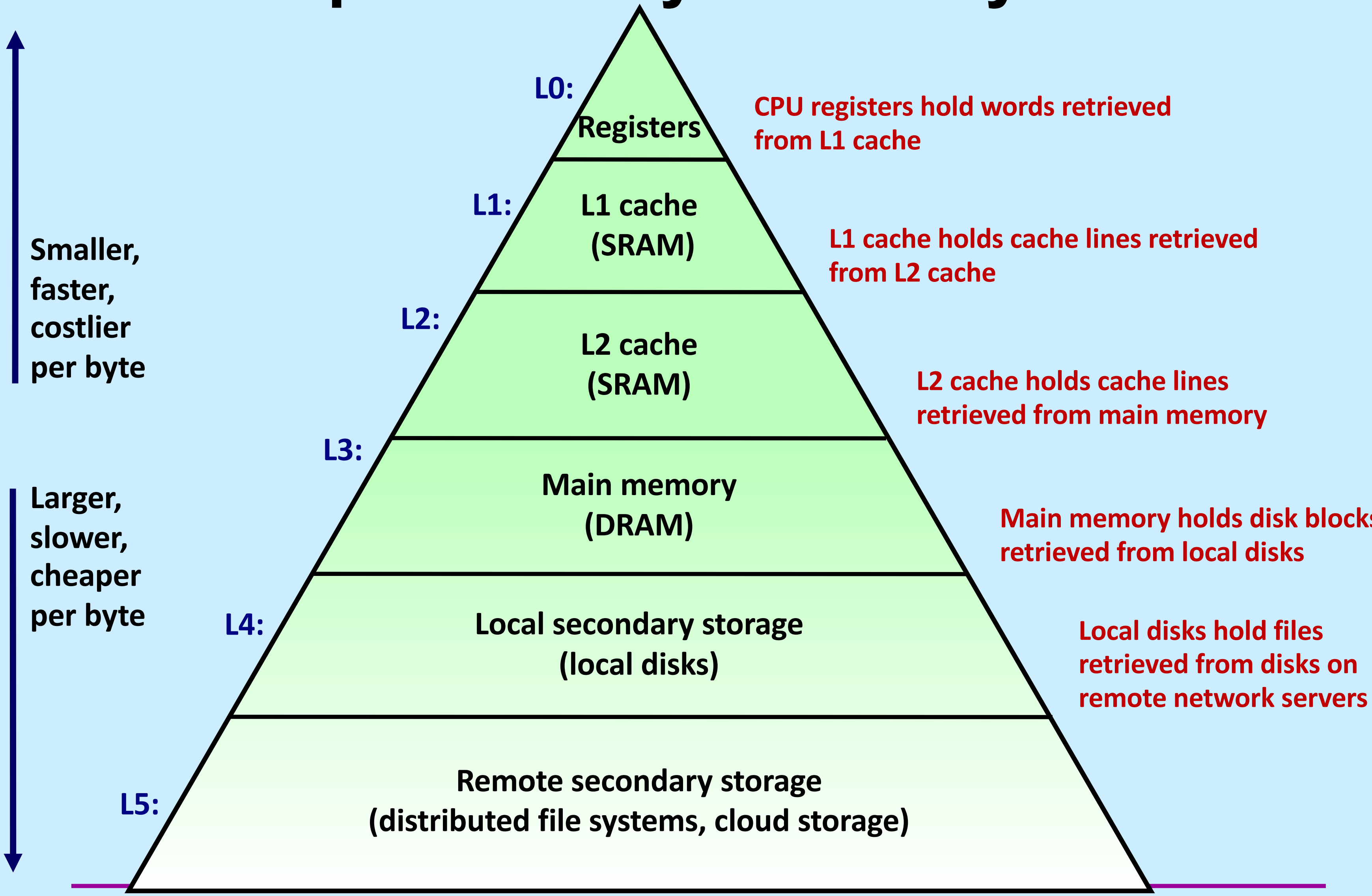
Now suppose the files are read concurrently, perhaps a block of the first, then a block of the second, then the first, then the second, etc. Compared to reading them sequentially, this will take

- **a) less time**
- **b) much more time**
- **c) about the same amount of time (within a factor of 2)**

Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - fast storage technologies cost more per byte, have less capacity, and require more power (heat!)
  - the gap between CPU and main memory speed is widening
  - well written programs tend to exhibit good locality
- These fundamental properties complement each other beautifully
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**

An Example Memory Hierarchy



Putting Things Into Perspective ...

- Reading from:
  - ... the L1 cache is like grabbing a piece of paper from your desk (3 seconds)
  - ... the L2 cache is picking up a book from a nearby shelf (14 seconds)
  - ... main system memory (DRAM) is taking a 4-minute walk down the hall to talk to a friend
  - ... a hard drive is like leaving the building to roam the earth for one year and three months

Disks Are Still Important

- Cheap
  – cost/byte less than SSDs
- (fairly) Reliable
  – data written to a disk is likely to be there next year
- Sometimes fast
  – data in consecutive sectors on a track can be read quickly
- Sometimes slow
  – data in randomly scattered sectors takes a long time to read

Abstraction to the Rescue

- **Programs don't deal with sectors, tracks, and cylinders**
- **Programs deal with *files***
  - **maze.c rather than an ordered collection of sectors**
  - **OS provides the implementation**

Implementation Problems

- Speed
  - use the hierarchy
    - » copy files into RAM, copy back when done
  - optimize layout
    - » put sectors of a file in consecutive locations
  - use parallelism
    - » spread file over multiple disks
    - » read multiple sectors at once

Implementation Problems

- Reliability
  - computer crashes
    - » what you thought was safely written to the file never made it to the disk — it's still in RAM, which is lost
    - » worse yet, some parts made it back to disk, some didn't
      - you don't know which is which
      - on-disk data structures might be totally trashed
  - disk crashes
    - » you had backed it up … yesterday
  - you screw up
    - » you accidentally delete the entire directory containing your shell 1 implementation

Implementation Problems

- Reliability solutions
  - computer crashes
    - » transaction-oriented file systems
    - » on-disk data structures always in well defined states
  - disk crashes
    - » files stored redundantly on multiple disks
  - you screw up
    - » file system automatically keeps "snapshots" of previous versions of files

CS 33

Linkers

gcc Steps

1) Compile
   - to start here, supply .c file
   - to stop here: `gcc -S` (produces .s file)
   - if not stopping here, gcc compiles directly into a .o file, bypassing the assembler
2) Assemble
   - to start here, supply .s file
   - to stop here: `gcc -c` (produces .o file)
3) Link
   - to start here, supply .o file

The Linker

- An executable program is one that is ready to be loaded into memory
- The linker (known as ld: /usr/bin/ld) creates such executables from:
  - object files produced by the compiler/assembler
  - collections of object files (known as libraries or archives)
  - and more we’ll get to soon ...

Linker’s Job

- Piece together components of program
  - arrange within address space
    - » code (and read-only data) goes into text region
    - » initialized data goes into data region
    - » uninitialized data goes into bss region
- Modify address references, as necessary

A Program

```
int nprimes = 100;
int *prime, *prime2;
int main() {
    int i, j, current = 1;
    prime = (int *)malloc(nprimes*sizeof(*prime));
    prime2 = (int *)malloc(nprimes*sizeof(*prime2));
    prime[0] = 2; prime2[0] = 2*2;
    for (i=1; i<nprimes; i++) {
    NewCandidate:
        current += 2;
        for (j=0; prime2[j] <= current; j++) {
            if (current % prime[j] == 0)
                goto NewCandidate;
        }
        prime[i] = current; prime2[i] = current*current;
    }
    return 0;
}
```

data

bss

dynamic

text

... with Output

```
int nprimes = 100;
int *prime, *prime2;
int main() {
   ...
   printcol(5);
   return 0;
}

void printcol(int ncols) {
   int i, j;
   int nrows = (nprimes+ncols-1)/ncols;
   for (i = 0; i<nrows; i++) {
      for (j=0; (j<ncols) && (i+nrows*j < nvals); j++) {
         printf("%6d", prime[i + nrows*j]);
      }
      printf("\n");
   }
}
```


... Compiled Separately

should refer to same thing

```
int nprimes = 100;
int *prime, *prime2;
int main() {
   ...
   printcol(5);
   return 0;
}
```

ditto

primes.c

```
extern int nprimes;
int *prime;
void printcol(int ncols) {
   int i, j;
   int nrows = (nprimes+ncols-1)/ncols;
   for (i = 0; i<nrows; i++) {
      for (j=0; (j<ncols)
            && (i+nrows*j < nvals); j++) {
         printf("%6d", prime[i + nrows*j]);
      }
      printf("\n");
   }
}
```

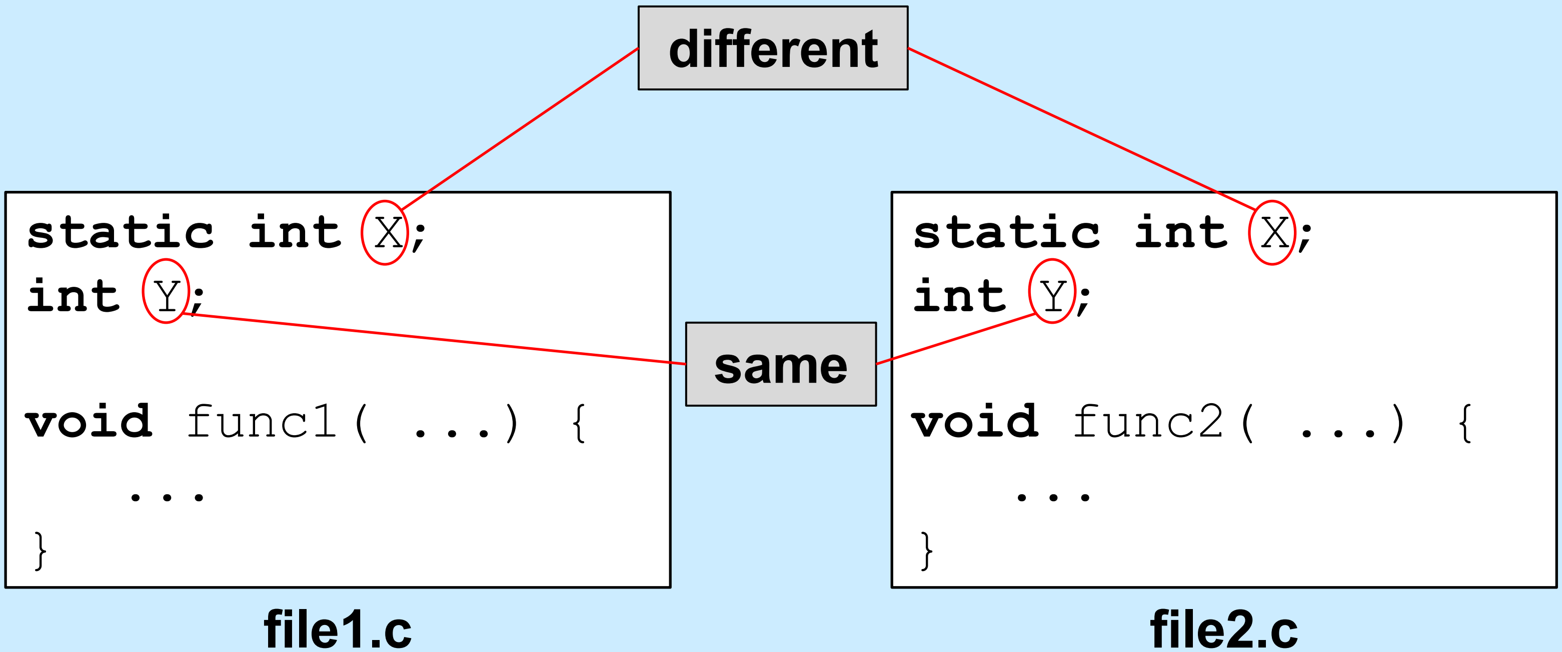
printcol.c

gcc –c primes.c
gcc –c printcol.c
gcc –o primes primes.o printcol.o

Global Variables

- **Initialized vs. uninitialized**
  - **initialized allocated in *data* section**
  - **uninitialized allocated in *bss* section**
    - **implicitly initialized to zero**
- **File scope vs. program scope**
  - ***static* global variables known only within file that declares them**
    - **two of same name in different files are different**
    - **e.g., `static int X;`**
  - **non-static global variables potentially shared across all files**
    - **two of same name in different files are same**
    - **e.g., `int X;`**

Scope


different
static int X;
int Y;
void func1(...) {
...
}
same
static int X;
int Y;
void func2(...) {
...
}
file1.c
file2.c

Static Local Variables

```
int *sub1() {
  int var = 1;
  …
  return &var;
  /* amazingly illegal */
}
```

```
int *sub2() {
  static int var = 1;
  …
  return &var;
  /* (amazingly) legal */
}
```


Reconciling Program Scope (1)

tentative definition

```
int X;

void func1( ...) {
   ...
}
```

file1.c

(complete) definition

```
int X=1;

void func2( ...) {
   ...
}
```

file2.c

Where does X go?
What's its initial value?

- **tentative definitions overridden by compatible (complete) definitions**
- **if not overridden, then initial value is zero**

Reconciling Program Scope (2)

```
int X=2;

void func1( ...) {
    ...
}
```

file1.c

```
int X=1;

void func2( ...) {
    ...
}
```

file2.c

What happens here?

Reconciling Program Scope (3)

```
int X=1;

void func1( ...) {
    ...
}
```

file1.c

```
int X=1;

void func2( ...) {
    ...
}
```

file2.c

Is this ok?

Reconciling Program Scope (4)

```
extern int X;

void func1( ...) {
    ...
}
```

file1.c

```
int X=1;

void func2( ...) {
    ...
}
```

file2.c

What's the purpose of "extern"?

Does Location Matter?

```
int main(int argc, char *[]) {
   return(argc);
}
```

```
main:
  pushq %rbp    ; push frame pointer
  movq  %rsp, %rbp     ; set frame pointer to point to new frame
  movl  %edi, %eax     ; put argc into return register (eax)
  movq  %rbp, %rsp     ; restore stack pointer
  popq  %rbp    ; pop stack into frame pointer
  ret           ; return: pops end of stack into rip
```


Location Matters …

```
int X=6;
int *aX = &X;

int main() {
    void subr(int);
    int y = *aX;
    subr(y);
    return(0);
}

void subr(int i) {
    printf("i = %d\n", i);
}
```

Coping

- **Relocation**
  - **modify internal references according to where module is loaded in memory**
  - **modules needing relocation are said to be *relocatable***
    - » **which means they *require* relocation**
  - **the compiler/assembler provides instructions to the linker on how to do this**

A Revised Version of Our Program

```c
extern int X;
int *aX = &X;
int Y = 1;

int main() {
    void subr(int);
    int y = *aX+Y;
    subr(y);
    return(0);
}
```

main.c

```c
#include <stdio.h>
int X;

void subr(int XX) {
    printf("XX = %d\n", XX);
    printf("X = %d\n", X);
}
```

subr.c

```
gcc -o prog -O1 main.c subr.c
```

main.s (1)

```
            .file   "main.c"
0:          .text
0:          .globl  main
0:          .type   main, @function
0: main:
0: .LFB0:
0:          .cfi_startproc
0:          subq    $8, %rsp
4:          .cfi_def_cfa_offset 16
4:          movq    aX(%rip), %rax
11:         movl    (%rax), %edi
13:         addl    Y(%rip), %edi
19:         call    subr
24:         movl    $0, %eax
29:         addq    $8, %rsp
33:         .cfi_def_cfa_offset 8
33:         ret
34:         .cfi_endproc
34:.LFE0:
34:         .size   main, .-main
```

must be replaced with aX's address, expressed as an offset from the next instruction

must be replaced with Y's address, expressed as an offset from the next instruction

must be replaced with subr's address, expressed as an offset from the next instruction

main.s (2)

```
        .globl  Y
0:      .data
0:      .align 4
0:      .type   Y, @object
0:      .size   Y, 4
0:  Y:
0:      .long   1
4:      .globl  aX
8:      .align 8
8:      .type   aX, @object
8:      .size   aX, 8
8:  aX:
8:      .quad   X
8:      .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
0:      .section        .note.GNU-stack,"",@progbits
```

Y should be made known to others

aX should be made known to others

must be replaced with address of X

subr.s (1)

```
          .file   "subr.c"
0:        .section        .rodata.str1.1,"aMS",@progbits,1
0: .LC0:
0:        .string "XX = %d\n"
9: .LC1:
9:        .string "X = %d\n"
```

subr.s (2)

```
0:          .text
0:          .globl   subr
0:          .type    subr, @function
0: subr:
0: .LFB11:
0:          .cfi_startproc
0:          subq     $8, %rsp
4:          .cfi_def_cfa_offset 16
4:          movl     %edi, %esi
6:          movl     $.LC0, %edi
11:         movl     $0, %eax
16:         call     printf
21:         movl     X(%rip), %esi
27:         movl     $.LC1, %edi
32:         movl     $0, %eax
37:         call     printf
42:         addq     $8, %rsp
46:         .cfi_def_cfa_offset 8
46:         ret
47:         .cfi_endproc
47:.LFE11:
47:         .size    subr, .-subr
```

subr should be made known to others

must be replaced with .LC0's address

must be replaced with .LC1's address

must be replaced with printf's address, expressed as an offset from the next instruction

subr.s (3)

```
0:          .comm   X,4,4
0:          .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
0:          .section        .note.GNU-stack,"",@progbits
```

reserve 4 bytes of 4-byte aligned storage for X

Quiz 3

```
int X;
int func(int arg) {
  static int Y;
  int Z;

  ...

}
```

Which of *X*, *Y*, *Z*, and *arg* would the compiler know the addresses of at compile time?

a) none
b) just *X* and *Y*
c) just *arg* and *Z*
d) all

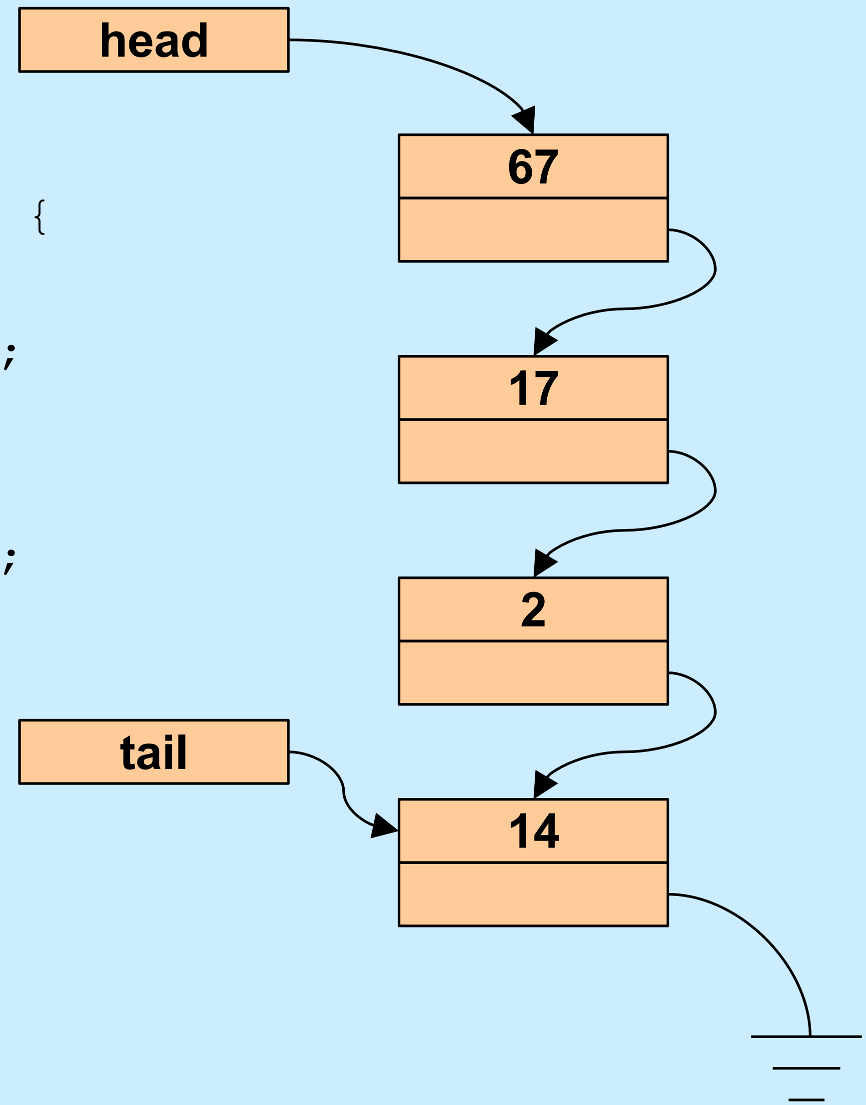
CS 33

Intro to Storage Allocation

A Queue

```
typedef struct list_element {
  int value;
  struct list_element *next;
} list_element_t;

list_element_t *head, *tail;
```



Enqueue

```
int enqueue(int value) {
  list_element_t *newle
      = (list_element_t *)malloc(sizeof(list_element_t));
  if (newle == 0)
    return 0; // can't do it: out of memory
  newle->value = value;
  newle->next = 0;
  if (head == 0) {
    // list was empty
    assert(tail == 0);
    head = newle;
  } else {
    tail->next = newle;
  }
  tail = newle;
  return 1;
}
```

Dequeue

```
int dequeue(int *value) {
  list_element_t *first;
  if (head == 0) {
    // list is empty
    return 0;
  }
  *value = head->value;
  first = head;
  head = head->next;
  if (tail == first) {
    assert(head == 0);
    tail = 0;
  }
  return 1;
}
```

What's wrong with this code???

Storage Leaks

```
int main() {
  while(1)
    if (malloc(sizeof(list_element_t)) == 0)
      break;
  return 1;
}
```

For how long will this program run before terminating?

Dequeue, Fixed

```
int dequeue(int *value) {
  list_element_t *first;
  if (head == 0) {
    // list is empty
    return 0;
  }
  *value = head->value;
  first = head;
  head = head->next;
  if (tail == first)
    assert(head == 0);
    tail = 0;
  }
  free(first);
  return 1;
}
```

Quiz 4

```
int enqueue(int value) {
  list_element_t *newle
      = (list_element_t *)malloc(sizeof(list_element_t));
  if (newle == 0)
    return 0;
  newle->value = value;
  newle->next = 0;
  if (head == 0) {
    // list was empty
    assert(tail == 0);
    head = newle;
  } else {
    tail->next = newle;
  }
  tail = newle;
  free(newle); // saves us the bother of freeing it later
  return 1;
}
```

This version of enqueue makes unnecessary the call to free in dequeue.

a) **It works well.**
b) **It fails occasionally.**
c) **It hardly ever works.**
d) **It never works.**

malloc and free

```
void *malloc(size_t size)
```

– allocate *size* bytes of storage and return a pointer to it

– returns 0 (NULL) if the requested storage isn't available

```
void free(void *ptr)
```

– free the storage pointed to by *ptr*

– *ptr* must have previously been returned by *malloc* (or other storage-allocation functions — *calloc* and *realloc*)

realloc

```
void *realloc(void *ptr, size_t size)
```

- change the size of the storage pointed to by *ptr*
- the contents, up to the minimum of the old size and new size, will not be changed
- *ptr* must have been returned by a previous call to *malloc*, *realloc*, or *calloc*
- it may be necessary to allocate a completely new area and copy from the old to the new
  - thus the return value may be different from *ptr*
  - if copying is done the old area is freed
- returns 0 if the operation cannot be done

Get (contiguous) Input (1)

```
char *getinput() {
  int alloc_size = 4;  // start small
  int read_size = 4;   // max number of bytes to read
  int next_read = 0;   // index in buf of next read
  int bytes_read;      // number of bytes read
  char *buf = (char *)malloc(alloc_size);
  char *newbuf;

  if (buf == 0) {
    // no memory
    return 0;
  }
```

Get (contiguous) Input (2)

```
while (1) {
  if ((bytes_read
        = read(0, buf+next_read, read_size)) == -1) {
    perror("getinput");
    return 0;
  }
  if (bytes_read == 0) {
    // eof
    break;
  }
  if ((buf+next_read)[bytes_read-1] == '\n') {
    // end of line
    break;
  }
```

Get (contiguous) Input (3)

```
    next_read += read_size;
    read_size = alloc_size;
    alloc_size *= 2;
    newbuf = (char *)realloc(buf, alloc_size);
    if (newbuf == 0) {
      // realloc failed: not enough memory.
      // Free the storage allocated previously and report
      // failure.
      free(buf);
      return 0;
    }
    buf = newbuf;
  }
```

Get (contiguous) Input (4)

```
  // reduce buffer size to the minimum necessary
  newbuf = (char *)realloc(buf,
      alloc_size - (read_size - bytes_read));
  if (newbuf == 0) {
    // couldn't allocate smaller buf
    return buf;
  }
  return newbuf;
}
```