CS 61C:
Great Ideas in Computer Architecture
Redundant Arrays of Inexpensive Disks and C Memory Management

Instructor:
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

You Are Here!

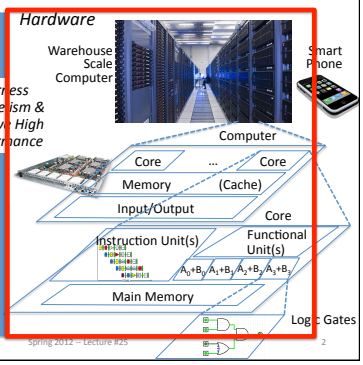
Today's Lecture

Software

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance

Hardware



Review

- Great Idea: Redundancy to Get Dependability
  - Spatial (extra hardware) and Temporal (retry if error)
- Reliability: MTTF & Annualized Failure Rate (AFR)
- Availability: % uptime (MTTF-MTTR/MTTF)
- Memory
  - Hamming distance 2: Parity for Single Error Detect
  - Hamming distance 3: Single Error Correction Code + encode bit position of error
  - Hamming distance 4: SEC/Double Error Detection
- CRC for many bit detection, Reed Solomon per disk sector for many bit error detection/correction

Agenda

- DRAM @ Google
- RAID Intro
- Administrivia
- RAID 5
- RAID at Berkeley
- What happened to RAID?
- C Memory Management
- Common Memory Problems
- Summary

DRAM: Google Early Years

- In 2000 didn't even use parity for memory
- During test of new search index program found it was suggesting random documents!
- Found that stuck-at-zero bit memory faults were corrupting new index
- Added software consistency checks
- Went to ECC as soon as ECC DIMMs became more affordable

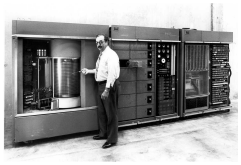
Real DRAM Failures at Google

- 2009 Study: Failures were much higher than published
- Used variation of Hamming code that can detect if all bits in wide DRAM are zero or one (4 bit or 8 bit wide DRAM): "Chip Kill"*
  - Otherwise failure rates 4 to 10 times higher
- Failures affected 8% of DRAM DIMMs
  - Dual In Line Modules have 8 to 16 DRAM chips
- Average DIMM had 4000 correctable errors and 0.2 uncorrectable errors per year
- Average server had 22,000 correctable errors and 1 uncorrectable error per year (average 5 DIMMs/server)
- In 1/3 servers, 1 memory error corrected every 2.5 hours
  - Without Chip Kill, 1 error every 15 to 40 minutes!
- If only parity, had to reboot machine on parity error and reboot is 5 minutes, then 1/3 servers spend 20% of time rebooting!

* *One simple scheme scatters the bits of a Hamming ECC word across multiple memory chips, such that the failure of any one memory chip will affect only one ECC bit per word. Memory DIMM much wider than 64 bits.*

Evolution of the Disk Drive

IBM 3390K, 1986

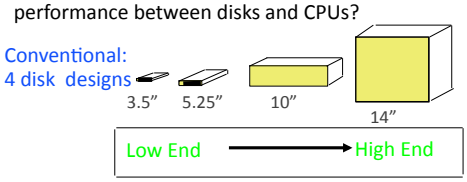


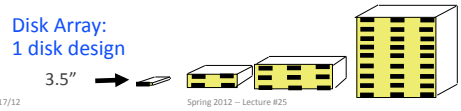
IBM RAMAC 305, 1956

Apple SCSI, 1986

Arrays of Small Disks

Can smaller disks be used to close gap in performance between disks and CPUs?





Replace Small Number of Large Disks with Large Number of Small Disks! (1988 Disks)

	IBM 3390K	IBM 3.5" 0061	x70	
Capacity	20 GBytes	320 MBytes	23 GBytes	
Volume	97 cu. ft.	0.1 cu. ft.	11 cu. ft.	9X
Power	3 KW	11 W	1 KW	3X
Data Rate	15 MB/s	1.5 MB/s	120 MB/s	8X
I/O Rate	600 I/Os/s	55 I/Os/s	3900 IOs/s	6X
MTTF	250 KHrs	50 KHrs	??? Hrs	
Cost	$250K	$2K	$150K	

Disk Arrays have potential for large data and I/O rates, high MB per cu. ft., high MB per KW, but what about reliability?

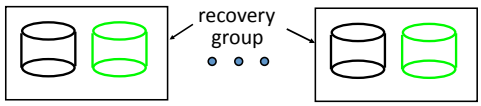
Array Reliability

- Reliability - whether or not a component has failed
  - measured as Mean Time To Failure (MTTF)
- Reliability of N disks
  = Reliability of 1 Disk ÷ N
  (assuming failures independent)
  - 50,000 Hours ÷ 70 disks = 700 hour
- Disk system MTTF:
  Drops from 6 years to 1 month!
- Disk arrays too unreliable to be useful!

RAID: Redundant Arrays of (Inexpensive) Disks

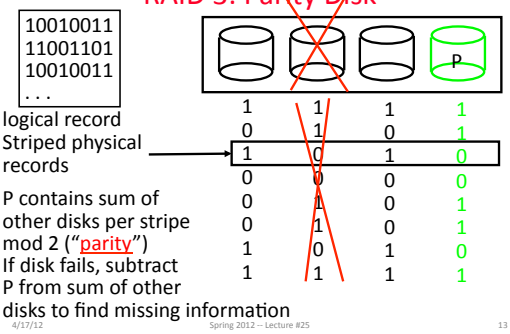
- Files are "striped" across multiple disks
- Redundancy yields high data availability
  - Availability: service still provided to user, even if some components failed
- Disks will still fail
- Contents reconstructed from data redundantly stored in the array
  - ⇒ Capacity penalty to store redundant info
  - ⇒ Bandwidth penalty to update redundant info

Redundant Arrays of Inexpensive Disks RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its "mirror"
  Very high availability can be achieved
- Bandwidth sacrifice on write:
  Logical write = two physical writes
  Reads may be optimized
- Most expensive solution: 100% capacity overhead

Redundant Array of Inexpensive Disks RAID 3: Parity Disk



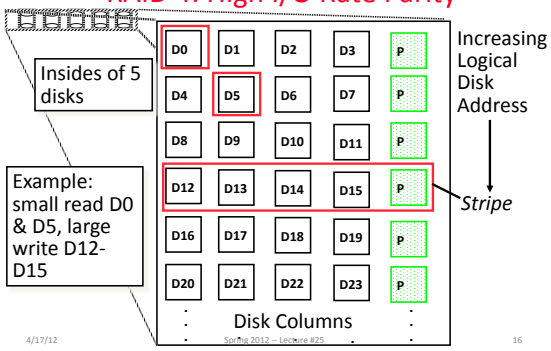
RAID 3

- Sum computed across recovery group to protect against hard disk failures, stored in P disk
- Logically, a single high capacity, high transfer rate disk: good for large transfers
- Wider arrays reduce capacity costs, but decreases availability
- 33% capacity cost for parity if 3 data disks and 1 parity disk

Inspiration for RAID 4

- RAID 3 relies on parity disk to discover errors on Read
- But every sector has an error detection field (Reed Solomon Code)
- To catch errors on read, rely on error detection field vs. the parity disk
- Allows independent reads to different disks simultaneously

Redundant Arrays of Inexpensive Disks RAID 4: High I/O Rate Parity



Administrivia

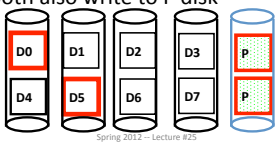
- Lab 13 this week – Malloc/Free in C
- Extra Credit: Fastest Version of Project 3
- Lectures: 2 on Virtual Machines/Memory, End f course Overview/HKN Evaluation
- Will send final survey end of this week
- All grades finalized: 4/27
- Final Review: Sunday April 29, 2-5PM, 2050 VLSB
- Extra office hours: Thu-Fri May 3 and May 4
- Final: Wed May 9 11:30-2:30, 1 PIMENTEL

Agenda

- RAID Intro
- Administrivia
- RAID 5
- RAID at Berkeley
- What happened to RAID?
- C Memory Management
- Common Memory Problems
- Summary

Inspiration for RAID 5

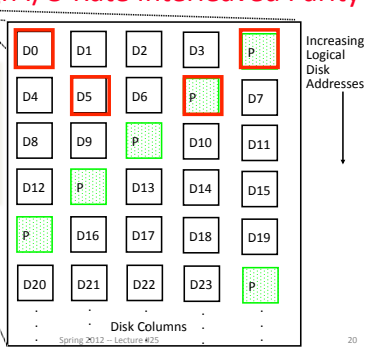
- RAID 4 works well for small reads
- RAID 4 small writes (write to one disk):
  - Option 1: read other data disks, create new sum and write to Parity Disk
  - Option 2: since P has old sum, compare old data to new data, add the difference to P
- Small writes are limited by Parity Disk: Write to D0, D5 both also write to P disk



RAID 5: High I/O Rate Interleaved Parity

Independent writes possible because of interleaved parity

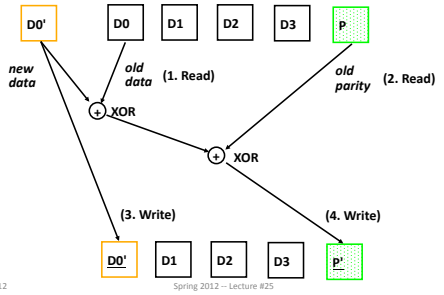
Example: write to D0, D5 uses disks 0, 1, 3, 4



Problems of Disk Arrays: Small Writes

RAID-5: Small Write Algorithm

1 Logical Write = 2 Physical Reads + 2 Physical Writes



Peer Instruction

I. **RAID 1 (mirror) and 5 (rotated parity) both help with performance and availability**

II. **RAID 1 has higher cost than RAID 5**

III. **Small writes on RAID 5 are slower than on RAID 1**

A)(orange) Only I is True

B)(green) Only II is True

C)(pink) I is True and II is True

D)(yellow) All are true

RAID 6: Recovering from 2 failures

- Why > 1 failure recovery?
  - operator accidentally replaces wrong disk during a failure
  - since disk bandwidth is growing more slowly than disk capacity, the MTT Repair a disk in a RAID system is increasing
    ⇒increases the chances of a 2nd failure during repair since takes longer
  - reading much more data during reconstruction meant increasing the chance of an uncorrectable media failure during read, which would result in data loss

RAID 6: Recovering from 2 failures

- Network Appliance's *row-diagonal parity* or *RAID-DP*
- Like the standard RAID schemes, it uses redundant space based on parity calculation per stripe
- Since it is protecting against a double failure, it adds two check blocks per stripe of data.
  - If p+1 disks total, p-1 disks have data; assume p=5
- Row parity disk is just like in RAID 4
  - Even parity across the other 4 data blocks in its stripe
- Each block of the diagonal parity disk contains the even parity of the blocks in the same diagonal

RAID-I

- RAID-I (1989)
  - Consisted of a Sun 4/280 workstation with 128 MB of DRAM, four dual-string SCSI controllers, 28 5.25-inch SCSI disks and specialized disk striping software

RAID II

- 1990-1993
- Early (first?) Network Attached Storage (NAS) System running a Log Structured File System (LFS)

RAID II

Tech Report Read 'Round the World
(December 1987)

Google scholar Case for RAID

Scholar Articles and patents anytime include citations Create ema

[BOOK] A case for redundant arrays of inexpensive disks (RAID)
DA Patterson, G Gibson… - 1988 - dl.acm.org
Abstract Increasing performance of CPUs and memories will be squandered if not matched by a similar performance increase in I/O. While the capacity of Single Large Expensive Disks (SLED) has grown rapidly, the performance improvement of SLED has been modest. ...
Cited by 2713 - Related articles - Library Search - All 228 versions

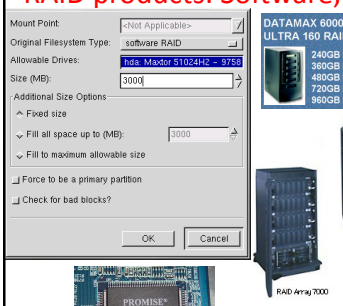
Increasing performance of CPUs and memories will be squandered if not matched by a similar performance increase in I/O. While the capacity of Single Large Expensive Disk (SLED) has grown rapidly, the performance improvement of SLED has been modest. Redundant Arrays of Inexpensive Disks (RAID), based on the magnetic disk technology developed for personal computers, offers an attractive alternative to SLED, promising improvements of an order of magnitude in performance, reliability, power consumption, and scalability.
This paper introduces five levels of RAIDs, giving their relative cost/performance, and compares RAIDs to an IBM 3380 and a Fujitsu Super Eagle.

What Happened to RAID?

- Article in *Byte Magazine** led to RAID becoming popular with PCs
- EMC forced out of making DRAM for IBM Mainframes, read tech report and started making storage arrays for IBM Mainframes
- RAID sold as fast, reliable storage but expensive
  - Industry: OK if we change I in RAID from Inexpensive to Independent?

*M. H. Anderson. "Strength (and safety) in numbers: RAID systems may boost PC performance and reliability." *Byte Magazine*, 15(13):337–339, December 1990

RAID products: Software, Chips, Systems



- RAID was $32 B industry in 2002, 80% nonPC disks sold in RAIDs

Margin of Safety in CS&E?

- Like Civil Engineering, never make dependable systems until add margin of safety ("margin of ignorance") for what we don't (can't) know?
  - Before: design to tolerate expected (HW) faults
- RAID 5 Story
  - Operator removing good disk vs. bad disk
  - Temperature, vibration causing failure before repair
  - In retrospect, suggested RAID 5 for what we anticipated, but should have suggested RAID 6 (double failure OK) for unanticipated/safety margin?
- CS&S Margin of Safety: Tolerate human error in design, in construction, and in use?

What about RAID Paper?

David A. Patterson, Garth Gibson , Randy H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *Proc. 1988 ACM SIGMOD int'l conf. on Management of Data*, 1988

1. Test of Time Award from ACM SIGMOD, 1998
2. Hall of Fame Award from ACM Special Interest Group in Operating System, 2011
3. Jean-Claude Laprie Award in Dependable Computing from IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, 2012

What Happened with RAID?

- Project holds reunions (10th below, 20th in August)

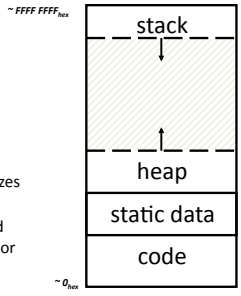
Front Row (Left-Right): Me, Randy Katz (Berkeley), John Ousterhout (Stanford/Electric Cloud founder).
Middle row: Pete Chen (Michigan), Garth Gibson (CMU/Panasas founder), Ann Chervenak (USC ISI), Ed Lee (Data Domain), Mary Baker (HP Labs), Kim Keeton (HP Labs), Ken Shirriff (Sun Labs), Fred Douglis (IBM Research).
Last row: Ken Lutz (Berkeley), Martin Schulze, Rob Pfile, Ethan Miller (UC Santa Cruz), Mendel Rosenblum (Stanford/VMware founder), John Hartman (Arizona), Steve Strange (NetApp).

Agenda

- RAID Intro
- Administrivia
- RAID 5
- RAID at Berkeley
- What happened to RAID?
- C Memory Management
- Common Memory Problems
- Summary

Recap: C Memory Management

- Program's *address space* contains 4 regions:
  - stack: local variables, grows downward
  - heap: space requested for pointers via `malloc()`; resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change



OS prevents accesses between stack and heap (via virtual memory)

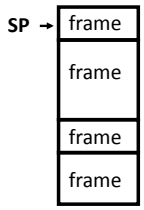
Recap: Where are Variables Allocated?

- If declared outside a procedure, allocated in "static" storage
- If declared inside procedure, allocated on the "stack" and freed when procedure returns
  - main() is treated like a procedure

```
int myGlobal;
main() {
  int myTemp;
}
```

Recap: The Stack

- Stack frame includes:
  - Return "instruction" address
  - Parameters
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer indicates top of stack frame
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames



Observations

- Code, Static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- *Managing the heap is tricky*: memory can be allocated / deallocated at any time

Managing the Heap

- C supports five functions for heap management: **`malloc()`**, `calloc()`, **`free()`**, `cfree()`, `realloc()`
- `malloc(n)`:
  - Allocate a block of uninitialized memory
  - NOTE: Subsequent calls need not yield blocks in continuous sequence
  - `n` is an integer, indicating size of allocated memory block in bytes
  - `sizeof` determines size of given type in bytes, produces more portable code
  - Returns a pointer to that memory location; NULL return indicates no more memory
  - Think of ptr as a *handle* that also describes the allocated block of memory; Additional control information stored in the heap around the allocated block!
- Example:

```
int *ip;
ip = malloc(sizeof(int));

struct treeNode *tp;
tp = malloc(sizeof(struct treeNode));
```


Managing the Heap

- `free(p)`:
  - Releases memory allocated by `malloc()`
  - `p` is pointer containing the address *originally* returned by `malloc()`

```
int *ip;
ip = malloc(sizeof(int));
... .. ..
free(ip);  /* Can you free(ip) after ip++ ? */

struct treeNode *tp;
tp = malloc(sizeof(struct treeNode));
    ... .. ..
free(tp);
```

  - When insufficient free memory, `malloc()` returns `NULL` pointer; **Check for it!**

```
if ((ip = malloc(sizeof(int))) == NULL){
    printf("\nMemory is FULL\n");
    exit(1);
}
```

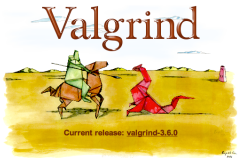
  - When you free memory, you must be sure that you pass the **original address** returned from `malloc()` to `free()`; Otherwise, system exception!

Common Memory Problems

- Using uninitialized values
- Using memory that you don't own
  - Deallocated stack or heap variable
  - Out of bounds reference to stack or heap array
  - Using NULL or garbage data as a pointer
- Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
- Memory leaks (you allocated something you forgot to later free)

Memory Debugging Tools

- Runtime analysis tools for finding memory errors
  - Dynamic analysis tool: collects information on memory management while program runs
  - Contrast with static analysis tool like `lint`, which analyzes source code without compiling or executing it
  - No tool is guaranteed to find ALL memory bugs – this is a very challenging programming language research problem
  - Runs 10X slower



http://valgrind.org

Using Memory You Don't Own

- What is wrong with this code?

```
int *ipr, *ipw;
void ReadMem() {
     *ipr = malloc(4 * sizeof(int));
     int i, j;
     i = *(ipr – 1000); j = *(ipr + 1000);
     free(ipr);
   }

   void WriteMem() {
     *ipw = malloc(5 * sizeof(int));
     *(ipw – 1000) = 0; *(ipw + 1000) = 0;
     free(ipw);
   }
```

Student Roulette

Using Memory You Don't Own

- Using pointers beyond the range that had been malloc'd
  - May look obvious, but what if mem refs had been result of pointer arithmetic that erroneously took them out of the allocated range?

```
int *ipr, *ipw;
void ReadMem() {
   *ipr = malloc(4 * sizeof(int));
   int i, j;
   i = *(ipr – 1000); j = *(ipr + 1000);
   free(ipr);
}

void WriteMem() {
   *ipw = malloc(5 * sizeof(int));
   *(ipw – 1000) = 0; *(ipw + 1000) = 0;
   free(ipw);
}
```


Faulty Heap Management

- What is wrong with this code?

```
int *pi;
void foo() {
  pi = malloc(8*sizeof(int));
  …
  free(pi);
}

void main() {
  pi = malloc(4*sizeof(int));
  foo();
  …
}
```

Student Roulette

Faulty Heap Management

- Memory leak: *more mallocs than frees*

```
int *pi;
void foo() {
  pi = malloc(8*sizeof(int));
  /* Allocate memory for pi */
  /* Oops, leaked the old memory pointed to by pi */
  …
  free(pi); /* foo() is done with pi, so free it */
}

void main() {
  pi = malloc(4*sizeof(int));
  foo(); /* Memory leak: foo leaks it */
  …
}
```


Faulty Heap Management

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
  … … …
  plk++;
}
```

Student Roulette

Faulty Heap Management

- Potential memory leak – handle has been changed, do you still have copy of it that can correctly be used in a later free?

```
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
  … … …
  plk++; /* Potential leak: pointer variable
  incremented past beginning of block! */
}
```

Faulty Heap Management

- What is wrong with this code?

Student Roulette

```
void FreeMemX() {
  int fnh = 0;
  free(&fnh);
}

void FreeMemY() {
  int *fum = malloc(4 * sizeof(int));
  free(fum+1);
  free(fum);
  free(fum);
}
```


Faulty Heap Management

- Can't free non-heap memory; Can't free memory that hasn't been allocated

```
void FreeMemX() {
  int fnh = 0;
  free(&fnh); /* Oops! freeing stack memory */
}

void FreeMemY() {
  int *fum = malloc(4 * sizeof(int));
  free(fum+1);
  /* fum+1 is not a proper handle; points to middle
  of a block */
  free(fum);
  free(fum);
  /* Oops! Attempt to free already freed memory */
}
```


Using Memory You Haven't Allocated

- What is wrong with this code?

Student Roulette

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  printf("%s\n", str);
}
```


Using Memory You Haven't Allocated

- Reference beyond array bounds

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  /* Write Beyond Array Bounds */
  printf("%s\n", str);
  /* Read Beyond Array Bounds */
}
```


Using Memory You Don't Own

- What's wrong with this code?

Student Roulette

```
char *append(const char* s1, const char *s2) {
  const int MAXSIZE = 128;
  char result[128];
  int i=0, j=0;
  for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
   result[i] = s1[j];
  }
  for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
   result[i] = s2[j];
  }
  result[++i] = '\0';
  return result;
}
```


Using Memory You Don't Own

- Beyond stack read/write

```
char *append(const char* s1, const char *s2) {
  const int MAXSIZE = 128;
  char result[128];
  int i=0, j=0;
  for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
   result[i] = s1[j];
  }
  for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
   result[i] = s2[j];
  }
  result[++i] = '\0';
  return result;
}
```

`result` is a local array name – stack memory allocated

Function returns pointer to stack memory – won't be valid after function returns

Using Memory You Don't Own

- What is wrong with this code?

```
typedef struct node {
   struct node* next;
   int val;
} Node;

int findLastNodeValue(Node* head) {
   while (head->next != NULL) {
      head = head->next;
   }
   return head->val;
}
```

Student Roulette

Using Memory You Don't Own

- Following a NULL pointer to mem addr 0!

```
typedef struct node {
   struct node* next;
   int val;
} Node;

int findLastNodeValue(Node* head) {
   while (head->next != NULL) {
   /* What if head happens to be NULL? */
      head = head->next;
   }
   return head->val; /* What if head is NULL? */
}
```


Summary

- RAID
  - Goal was performance with more disk arms per $
  - Reality: availability what people cared about
  - Adds availability option for small number of extra disks
  - Today RAID is multibillion dollar industry, started at UC berkeley
- C has three pools of data memory (+ code memory)
  - Static storage: global variable storage, ~permanent, entire program run
  - The Stack: local variable storage, parameters, return address
  - *The Heap (dynamic storage): `malloc()` gets space from here, `free()` returns it*
- Common (Dynamic) Memory Problems
  - Using uninitialized values
  - Accessing memory beyond your allocated region
  - Improper use of free by changing pointer handle returned by malloc
  - Memory leaks: mismatched malloc/free pairs

Peer Instruction: What's Wrong with this Code?

```
int *ptr () {
 int y;
 y = 3;
 return &y;
};

main () {
 int *stackAddr,content;
 stackAddr = ptr();
 content = *stackAddr;
 printf("%d", content); /* 3 */
 content = *stackAddr;
 printf("%d", content); /*13451514 */
};
```

Green: printf modifies stackAddr
Yellow: printf bug modifies content
Red: printf overwrites stack frame

Managing the Heap

- `calloc(n, size)`:
  - Allocate n elements of same data type; n can be an integer variable, use `calloc()` to allocate a dynamically size array
  - `n` is the # of array elements to be allocated
  - `size` is the number of bytes of each element
  - `calloc()` guarantees that the memory contents are initialized to zero

  E.g.: allocate an array of 10 elements

  ```
  int *ip;
  ip = calloc(10, sizeof(int));
  ```

  `*(ip+1)` refers to the 2nd element, like `ip[1]`
  `*(ip+i)` refers to the i+1th element, like `ip[i]`

  Beware of referencing beyond the allocated block: e.g., `*(ip+10)`
  - `calloc()` returns `NULL` if no further memory is available
- `cfree(p)`:
  - `cfree()` releases the memory allocated by `calloc()`; E.g.: `cfree(ip);`

Using Uninitialized Values

- What is wrong with this code?

```
void foo(int *pi) {
    int j;
    *pi = j;
}

void bar() {
    int i=10;
    foo(&i);
    printf("i = %d\n", i);
}
```

Student Roulette

Using Memory You Don't Own

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
  ptr = realloc(ptr, new_size*sizeof(int));
  memset(ptr, 0, new_size*sizeof(int));
  return ptr;
}

int* fill_fibonacci(int *fib, int size) {
  int i;
  init_array(fib, size);
  /* fib[0] = 0; */ fib[1] = 1;
  for (i=2; i<size; i++)
   fib[i] = fib[i-1] + fib[i-2];
  return fib;
}
```

Student Roulette