True or False

1. A capacity miss can occur even when the cache isn't full.
   True

2. The main benefit of coalescing neighboring free blocks is that it increases throughput.
   False, the main benefit is reducing external fragmentation.

3. A parent process waits on the termination of grandchild processes.
   False, it only waits on the termination of its child processes.

4. Stack memory is shared across threads.
   False, *heap* memory is shared across threads.

5. Fine-grained locking solutions are always faster than global locking solutions in thread-level parallelism.
   False, consider the case where a resource must be locked whenever it's accessed, and it's accessed in every single operation.

Datalab

1. Which of the following returns 2^k, for all k > 0? Select all that apply.
    a. 1 << k
    b. 2 << k
    c. 2 << (k-1)
    d. 4 << (k-1)

2. Which of the following returns the absolute value of int x? Select all that apply.
    a. ((x >> 31) + x) ^ (x >> 31)
    b. ((x >> 31) ^ x) + ~(x>>31) + 1
    c. ((x >> 63) + x) ^ (x >> 63)
    d. ((x >> 63) ^ x) + ~(x>>63) + 1

Bomblab

Dr. Evil decided to take revenge because you completed bomblab! He says he's going to make it extra hard this time…

You see these lines of assembly in the dump, and grab this info from GDB:

```
4011a1:     48 8d 35 62 0e 00 00    lea    0xe62(%rip),%rsi        # 40200a
...
4011ad:     e8 ae fe ff ff          call   401060 <__isoc99_sscanf@plt>

(gdb) x/s 0x40200a
0x40200a:   "%ud %ud %ud %ud %ud %ud %ud"
```

a. What is the input format to the bomb? 7 unsigned integers

Dr. Evil kindly parses in the input for you, so you can assume the function `bomb` takes in seven of the types you said in part a. He also provides you with the assembly dump below:

```
000000000040116f <bomb>:
  40116f:   f3 0f 1e fa             endbr64
  401173:   48 83 ec 28             sub    $0x28,%rsp
  401177:   83 7c 24 30 05          cmpl   $0x5,0x30(%rsp)
  40117c:   77 25                   ja     4011a3 <bomb+0x34>
  40117e:   89 3c 24                mov    %edi,(%rsp)
  401181:   89 74 24 04             mov    %esi,0x4(%rsp)
  401185:   89 54 24 08             mov    %edx,0x8(%rsp)
  401189:   89 4c 24 0c             mov    %ecx,0xc(%rsp)
  40118d:   44 89 44 24 10          mov    %r8d,0x10(%rsp)
  401192:   44 89 4c 24 14          mov    %r9d,0x14(%rsp)
  401197:   b8 00 00 00 00          mov    $0x0,%eax
  40119c:   ba 00 00 00 00          mov    $0x0,%edx
  4011a1:   eb 25                   jmp    4011c8 <bomb+0x59>
  4011a3:   e8 ae ff ff ff          call   401156 <explode_bomb>
  4011a8:   b8 01 00 00 00          mov    $0x1,%eax
  4011ad:   eb 47                   jmp    4011f6 <bomb+0x87>
  4011af:   48 63 f0                movslq %eax,%rsi
  4011b2:   8b 0c b4                mov    (%rsp,%rsi,4),%ecx
  4011b5:   03 4c 24 14             add    0x14(%rsp),%ecx
  4011b9:   89 0c b4                mov    %ecx,(%rsp,%rsi,4)
  4011bc:   48 63 c8                movslq %eax,%rcx
```

```
4011bf:   03 14 8c              add    (%rsp,%rcx,4),%edx
4011c2:   c1 fa 02              sar    $0x2,%edx
4011c5:   83 c0 01              add    $0x1,%eax
4011c8:   83 f8 05              cmp    $0x5,%eax
4011cb:   7f 18                 jg     4011e5 <bomb+0x76>
4011cd:   85 c0                 test   %eax,%eax
4011cf:   7e de                 jle    4011af <bomb+0x40>
4011d1:   8d 70 ff              lea    -0x1(%rax),%esi
4011d4:   48 63 f6              movslq %esi,%rsi
4011d7:   48 63 f8              movslq %eax,%rdi
4011da:   8b 0c bc              mov    (%rsp,%rdi,4),%ecx
4011dd:   03 0c b4              add    (%rsp,%rsi,4),%ecx
4011e0:   89 0c bc              mov    %ecx,(%rsp,%rdi,4)
4011e3:   eb d7                 jmp    4011bc <bomb+0x4d>
4011e5:   83 c2 0c              add    $0xc,%edx
4011e8:   8b 44 24 30           mov    0x30(%rsp),%eax
4011ec:   39 14 84              cmp    %edx,(%rsp,%rax,4)
4011ef:   75 0a                 jne    4011fb <bomb+0x8c>
4011f1:   b8 00 00 00 00        mov    $0x0,%eax
4011f6:   48 83 c4 28           add    $0x28,%rsp
4011fa:   c3                    ret
4011fb:   e8 56 ff ff ff        call   401156 <explode_bomb>
401200:   b8 01 00 00 00        mov    $0x1,%eax
401205:   eb ef                 jmp    4011f6 <bomb+0x87>
```

b. Fill in the following code skeleton:

```
int bomb(unsigned int one, unsigned int two, unsigned int three, unsigned int
four, unsigned int five, unsigned int six, unsigned int idx) {
    if (idx>=6) {
        explode_bomb();
        return 1;
    }

    unsigned int array[] = {one, two, three, four, five, six};
    int acc = 0;

    for (int i=0; i<6; i++) {
        if (i>0) array[i] += array[i-1];
        else array[i] += array[5];
        acc += array[i];
```

```
        acc >>= 2;
    }
    acc += 12;

    if (array[idx] != acc) {
        explode_bomb();
        return 1;
    }

    return 0;
}
```

c. Given the command line input of the first six elements is “15 21 3 51 3 14”, what is the seventh input to prevent the bomb from exploding? 1

Attacklab

You are auditing a target program that reads one line of user input with **Gets()**, a function that (like **gets()**) reads characters until a newline and performs **no length checking**.

```
The C source of solve is:
void solve(void) { char buf[24]; Gets(buf); }
000000000040064d <win>:
   40064d:  push   %rbp
   40064e:  mov    %rsp,%rbp
   ...             (prints "Access granted!")
   4006a1:  pop    %rbp
   4006a2:  retq

00000000004006b5 <solve>:
   4006b5:  sub    $0x38,%rsp
   4006b9:  lea    0x10(%rsp),%rdi
   4006be:  callq  40073f <Gets>
   4006c3:  add    $0x38,%rsp
   4006c7:  retq
```

The target also contains the following gadget, and win **expects its** argument in **%edi**:

```
   400700:  pop    %rdi
   400701:  retq
```

(a) Draw the stack frame of solve at the moment Gets is called. Label buf and its 24 bytes, any unused padding, and the saved return address, and mark where %rsp points.

After sub $0x38,%rsp, %rsp points to the bottom of a 0x38-byte frame. buf's 24 bytes occupy 0x10(%rsp) through 0x27(%rsp), and Gets writes upward starting at 0x10(%rsp); bytes 0x0-0xf below buf and 0x28-0x37 above it are unused padding. The saved return address sits at 0x38(%rsp), just above the frame, so from the start of buf the return address is exactly 0x28 (40) bytes away.

(b) Briefly explain why this code has a buffer overflow vulnerability.

Gets performs no bounds checking, so an input line longer than the space between buf and the saved return address keeps writing upward and overwrites the saved return address (and anything else above buf) with attacker-controlled bytes.

(c) Give an exploit string that makes the program execute **win** when solve returns.

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
4d 06 40 00 00 00 00 00
```

(d) Now suppose **win** only prints "Access granted!" if its 1st argument equals **0x15213**, and the stack is non-executable so you cannot inject code. Give the full exploit string layout that calls win(0x15213), using the gadget above. Explain what happens on each **ret** instruction.

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 07 40 00 00 00 00 00
13 52 01 00 00 00 00 00
4d 06 40 00 00 00 00 00

The ret at the end of solve jumps to the gadget; pop %rdi loads 0x15213 into %rdi (so %edi is 0x15213); the gadget's ret pops the next value, the address of win, into %rip, entering win with the correct argument.

Cachelab

You have a cache of size 64 bytes, 4 sets, and 2 lines per set, in a 32-bit address space. What is its configuration?

s = 2, b = 2, tag bits = 25

Given a trace with 6 arbitrary operations, which of the following is possible? Assume the cache is empty before the trace starts. Select all that apply.

a. Cold miss
b. Conflict miss
c. Capacity miss
d. None of the above

Given a trace with 9 arbitrary operations, however, all map to set 1. Which of the following is possible? Select all that apply.

a. Cold miss
b. Conflict miss
c. Capacity miss
d. None of the above

What is the miss rate of the code below? Assume the variable int `sum` is stored in a register, the int array `array` contains 64 ints, and array[0] aligns to set 0 of the cache.

None

```
int sum = 0;

for (int j=0; j<8; j++) {
       for (int i=0; i<8; i++) {
              sum += array[i*8 + j];
       }
}
```

Miss rate = 100%

What could be done to improve the hit rate? Flip the order of the i and j loops

Virtual Memory

1. Consider an x86-64 system with 48-bit virtual address space, 4 KB pages, and 8-byte page table entries. Select all true statements.

A. A TLB miss always results in a page fault.
B. The same virtual address in two different processes can map to two different physical pages.
C. Multi-level page tables make address translation faster than single-level page tables, because each level is smaller to search.
D. A single 4 KB page can hold 512 PTEs.
E. The virtual address space contains 2^48 virtual pages.

2. Consider the following partial page table with 14-bit virtual addresses, 12-bit physical addresses, 64-byte pages.

VPN	PPN	Valid
0x68	0x11	1
0x69	0x2B	1
0x6A	0x07	0
0x6B	0x30	1

For virtual address 0x1A7C, give (a) the VPN, (b) the VPO, (c) the physical address, or say “page fault” if it occurs.

VPN: 0x3C
VPO: 0x69
PPN: 0x2B
PA: 0xAFC

Malloclab

Advait is working on malloclab, but his throughput is really slow. Which of the following could help him improve this? Select all that apply.

a. Implementing coalesce_block
b. Implementing segregated lists
c. Changing from best_fit to first_fit
d. Implementing miniblocks

With your help, Advait's code is now meeting the throughput requirements. Now, he's trying to improve his utilization. Which of the following could help him reduce internal fragmentation? Select all that apply,

a. Implementing coalesce_block
b. Implementing footer removal
c. Implementing explicit lists
d. Implementing miniblocks

You are given the following trace. Assume coalesce_block is implemented, there are 8 byte headers and footers on all blocks, payloads are 16-byte aligned, and blocks are allocated via first_fit. Every call to extend the heap makes a request for a multiple of 32 bytes.

```
None
void *a = malloc(8);
void *b = malloc(32);
void *c = malloc(16);
free(b);
void *d = malloc(64);
free(c);
```

What does the heap look like at the end of this trace?

| block a = 32 bytes | free = 96 bytes | block d = 80 bytes | free = 16 bytes |
^ heap_start ^ heap end

What is the utilization of the heap at the end of this trace? 72 / 224 = 0.321

Tshlab

Consider the following code:

```C/C++
void sigchld_handler(int sig) {
    sio_printf("baby shark\n");
}

void sharks(void) {
    if (fork() == 0) {
        sio_printf("mommy shark\n");
        exit(0);
    }
    wait(NULL);

    Signal(SIGCHLD, sigchld_handler);
    if (fork() == 0) {
        sio_printf("daddy shark\n");
        exit(0);
    }
    sio_printf("doo-doo\n");
    exit(0);
}
```

a. Draw the process graph. Label all fork, wait, print, and exit events

b. Which of the following prints are possible: (where commas represent line breaks, select all correct options)

A. mommy shark, doo-doo, baby shark, daddy shark
B. mommy shark, daddy shark, baby shark, doo-doo
C. daddy shark, mommy shark, doo-doo
D. mommy shark, daddy shark, doo-doo, baby shark
E. mommy shark, baby shark, daddy shark, doo-doo
F. mommy shark, doo-doo, daddy shark

c. For each pair, write Always, Sometimes, or Never: does the first line print before the second?

- “mommy shark” before “doo-doo”: Always
- “daddy shark” before “doo-doo”: Sometimes
- “baby shark” before “daddy shark”: Never
- “baby shark” before “doo-doo”: Sometimes

File I/O

Q: The file input.txt contains ABCDE, with no newline. Consider the following code:

```
char buf[2];
int a = open("input.txt", O_RDONLY);
int b = dup(a);
read(a, buf, 1);
close(a);
a = open("input.txt", O_RDONLY);    /* X */
read(b, &buf[0], 1);
read(a, &buf[1], 1);
write(STDOUT_FILENO, buf, 2);
```

Assume that initially only descriptors 0, 1, and 2 are open, none refers to input.txt, and all system calls succeed and transfer the requested number of bytes. The file does not change.

(a) Draw the file descriptor table and open file table immediately after X. Include descriptor numbers and file offsets; omit descriptors 0–2.

a = 3 and b = 4
Descriptor 3 points to a new open-file-table entry for input.txt at offset 0. Descriptor 4 points to the original entry for input.txt at offset 1. Both entries are read-only and each has one descriptor reference. Reopening reuses the lowest available descriptor number, but creates an independent open-file-table entry.

(b) What exactly is printed?

BA
The read through b gets B from the original entry, whose offset is 1. The read through a gets A from the newly opened entry, whose offset is 0.

(c) If the line at X is replaced with a = dup(b);, what is printed instead? Briefly explain.

BC
Both descriptors now share the original open-file-table entry and its offset. After the earlier read of A, reading through b gets B and advances the shared offset; reading through a then gets C.

Proxylab

Ray is working on proxylab and has the following code:

```
None
parser_retrieve(parser, METHOD, &method);
parser_retrieve(parser, URI, &uri);
host_header = parser_lookup_header(parser, "Host");

buflen = snprintf(buf, MAXLINE,
              "%s %s HTTP/1.0\r\nUser-Agent: %s\r\n"
              "Connection: close\r\nProxy-Connection: close\r\n",
              method, uri, header_user_agent);
rio_writen(fd, buf, buflen);

while ((curr_header = parser_retrieve_next_header(parser)) != NULL) {
    if ((strcmp("User-agent", curr_header->name) != 0) &&
        strcmp("Host", curr_header->name) != 0 &&
        strcmp("Connection", curr_header->name) != 0 &&
        strcmp("Proxy-connection", curr_header->name) != 0) {
        buflen = snprintf(buf, MAXLINE, "%s: %s\r\n", curr_header->name,
                          curr_header->value);
        rio_writen(fd, buf, buflen);
    } else if (strcmp("Host", curr_header->name) == 0 &&
               strcmp("", curr_header->value) != 0) {
        host_header->value = curr_header->value;
    }
}

buflen = snprintf(buf, MAXLINE, "Host: %s\r\n", host_header->value);
rio_writen(fd, buf, buflen);

return;
```

She notices that it’s timing out when sending client headers to the server. What’s a possible reason why? You may assume it is *not* due to buffer overflow or `rio` call failure.

We’re missing the final \r\n return carriage to indicate the end of the message.

Yay! With your help, Ray managed to get her proxy working. She now wants to make it multithreaded, but is having more issues with concurrent message race conditions.

```
None
listenfd = open_listenfd(argv[1]);
connfp = Malloc(sizeof(int));

   while (1) {
       pthread_t tid;
       clientlen = sizeof(clientaddr);
       connfd = accept(listenfd, (struct sockaddr *restrict)&clientaddr,
                       &clientlen);
       getnameinfo((struct sockaddr *restrict)&clientaddr, clientlen, hostname,
                   MAXLINE, port, MAXLINE, 0);
       *connfp = connfd;
       pthread_create(&tid, NULL, thread, (void *)connfp);
   }
```

Please identify the problematic line in the code above and describe a change to fix it.

Threads share heap space, so we cannot send the same connfp across all threads. If we move the malloc call inside the while loop, this will fix the issue as we will reallocate with every loop iteration.

SFSlab

Sean decided to take a break from 213 to go to the park! But since he's in Pittsburgh, it's raining… so he decides to code virtual monkey bars to swing on instead.

```
None
struct bar {
	int num;
	char *person;
	bar *next;
};

struct monkey_bars {
	bar *start;
};

// indicates a person started swinging on the monkey bars
void start_monkey_bars(monkey_bars *bars, char *person) {
	bars.start.person = person;
}

// moves a person forward one monkey bar
bar *next_bar(monkey_bars *bars, bar *curr, char *person) {
	curr.person = "";
	curr = curr.next
	curr.person = person;
	return curr;
}

// moves a person through all monkey bars from current bar
void finish_bars(monkey_bars *bars, bar *curr, char *person){
	while (curr.next != NULL) {
		next_bar(bars, curr, person);
		curr = curr.next;
	}
}
```

Sophia was also planning on going to the park, and decides to join Sean at his virtual monkey bars. If they both call start_monkey_bars at the same time, which of the following outcomes are possible? Select all that apply.

a. Sophia is shown on the first monkey bar
b. Sean is shown on the first monkey bar
c. Both are shown on the first monkey bar
d. Neither is shown on the first monkey bar

Sophia says that this isn't safe, and that only one person should be able to go on the monkey bars at a time. Help Sophia make the monkey bars accessible to only one person at a time:

None

```
struct bar {
	int num;
	char *person;
	bar *next;
};
struct monkey_bars {
	bar *start;
	pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
};

void start_monkey_bars(monkey_bars *bars, char *person) {
	pthread_mutex_lock(&bars.lock);
	bars.start.person = person;
	pthread_mutex_unlock(&bars.lock);
}

bar *next_bar(monkey_bars *bars, bar *curr, char *person) {
	pthread_mutex_lock(&bars.lock);
	curr.person = "";
	curr = curr.next
	curr.person = person;
	pthread_mutex_unlock(&bars.lock);
	return curr;
}

void finish_bars_unsafe(monkey_bars *bars, bar *curr, char *person){
	while (curr.next != NULL) {
		next_bar(bars, curr, person);
		curr = curr.next;
	}
}

// Extra space here, which you can use to write helpers!
void finish_bars(monkey_bars *bars, bar *curr, char *person){
	pthread_mutex_lock(&bars.lock);
	finish_bars(bars, curr, person);
	pthread_mutex_unlock(&bars.lock);
}
```

Thanks to you, it's now safe for Sophia and Sean to use the monkey bars. However, Sean says it's no fun having to wait for Sophia to finish the monkey bars to wait his turn.

Sean says that as long as he isn't on the same monkey bar as Sophia, it's safe for them to both use the monkey bars. How could you change the code from part b to accomplish this?

We could put a lock on each bar, and lock it when accessing a bar. This would change the &bars.lock to &bar.lock (or &curr.lock) on every function except start_monkey_bars. In start_monkey_bars, we'd lock &bars.start.lock. We'd unlock the same way for all functions.

Sophia and Sean don't agree on which solution is better. What do you think? Why?

Sean's solution is better, because it prevents the unsafe behavior of two people being on the same bar at once, and allows more people to access the monkey bars at once. Basically, it allows for greater parallelism.