Malloc Debugging & Networking

15-213: Introduction to Computer Systems
Recitation 12: Monday, April 7th, 2014

Before We Start …

Some networking fun:

```
telnet towel.blinkenlights.nl
```

Today

- Malloc Debugging
  - FAQs
  - Common Errors
  - Heapchecker
  - GDB
- Networking: Sockets API

Today

- Malloc Debugging
  - **FAQs**
  - Common Errors
  - Heapchecker
  - GDB
- Networking: Sockets API

Malloc FAQs

- Starter codes clarification
  - Alignment: 8-byte instead of 4-byte
  - Counting unit: word (4 bytes) instead of byte
  - Free/allocated flag: MSB instead of LSB
  - Size: payload size instead of block size
  - Block pointer: pointing to header instead of payload

- Sorry… starter codes have some errors

```
static inline uint32_t* block_prev(uint32_t* const block) {
    return block - block_size(block - 1) - 2;
}
static inline uint32_t* block_next(uint32_t* const block) {
    return block + block_size(block) + 2;
}
```

Malloc FAQs

- Macro or inline?
  - Macros are simply expanded by the preprocessor
  - Macro is sometimes evil, hard to debug

```
#define abs(i) ( (i) >= 0 ? (i) : -(i) )
int foo();
Void usercode(int x) {
    int ans;
    ans = abs(x++);      // 'x++' is executed twice!
    ans = abs(foo());    // Danger! Side effects.
}
```

  - More evil macro examples:
    http://www.parashift.com/c++-faq/inline-vs-macros.html

Malloc FAQs

- Macro or inline?
  - Inline functions are parsed by the compiler, providing type checking and input handling
  - An inline function is as fast as a macro
  - Compiler may not inline the inline functions, because certain usages can make it unsuitable for inline substitution, e.g. what would happen if I make a recursion function inline?

Malloc FAQs

- const uint32_t* ptr or uint32_t* const ptr?
  - const uint32_t* ptr: mutable pointer to constant data
  - uint32_t* const ptr: constant pointer to mutable data
  - e.g.

```
static inline unsigned int block_size (const uint32_t* block)
{
    return (block[0] & 0x3FFFFFFF);
}

static inline uint32_t* block_next (uint32_t* const block)
{
    return block + block_size(block) + 2;
}
```

Today

- Malloc Debugging
  - FAQs
  - **Common Errors**
  - Heapchecker
  - GDB
- Networking: Sockets API

Malloc Common Errors

- Garbled bytes
  - If the driver complains about garbled bytes, that means you are overwriting part of an allocated payload.
  - Possible solution: Check your pointer arithmetic, and also be careful casting pointers.

- Pointer casting
  - Nothing actually happens in a pointer cast. It's just an assignment. Remember all pointers have the same size.
  - The magic happens in dereferencing and arithmetic
  - Cannot dereference `void*`, pointers must get assigned (or cast) into the right type

Malloc Common Errors

- Pointer arithmetic
  - Consider:

```
<type> * pointer1 = (<type> *)pointer2 + a;
```

  - Think about it as:

```
leal (pointer2, a, sizeof(type)), pointer1;
```

  - e.g.

```
void *bp = (void *)0x1000;
(char *)bp + 1 = ?
(int *)bp + 1 = ?
(int **)bp + 1 = ?
struct ST {
    int x;
    char y;
};
(struct ST *)bp + 1 = ?
```

Malloc Common Errors

- ‘needle’ fails
  - If you waste too much space, some tests (particularly needle) will fail with out of memory errors.
  - This might happen if your allocator loses track of some blocks, or your free() doesn’t work, or you are not splitting free blocks when allocating.
- mm_init
  - Remember that you need to reinitialize everything when mm_init is called. We will call it between all traces.
  - May require you to update some of your pointers

Malloc Common Errors

- Segmentation fault – our old friend
  - Don't just tell your TA you get segfault like this:

    ヽ(ᐛ _ ᐛ)ノ: Segfault. I'm doomed.
  - printf may not be the best way to debug segfaults
  - Segfaults are usually caused either by pointer arithmetic errors or violation of your invariants (corruption of the heap)
  - Use your heapchecker and GDB

Today

- Malloc Debugging
  - FAQs
  - Common Errors
  - **Heapchecker**
  - GDB
- Networking: Sockets API

Malloc Heapchecker

- General tips
  - Your heap checker should not print things out unless it finds an error. This lets you sprinkle calls to it throughout your code.
  - Once you know what you want your heap structure to look like, write a heap checker for that structure so that you can debug the rest of your malloc implementation.
  - Your heap checker should be detailed enough that the rest of your functions are guaranteed to work on any heap that your heap checker passes.

Invariants (last recitation)

- Block level:
  - Header and footer match
  - Payload area is aligned
- List level:
  - Next/prev pointers in consecutive free blocks are consistent
  - Free list contains no allocated blocks
  - All free blocks are in the free list
  - No contiguous free blocks in memory (unless you defer coalescing)
  - No cycles in the list (unless you use circular lists)
  - Segregated list contains only blocks that belong to the size class
- Heap level:
  - Prologue/Epilogue blocks are at specific locations (e.g. heap boundaries) and have special size/alloc fields
  - All blocks stay in between the heap boundaries
- And your own invariants (e.g. address order)

Today

- Malloc Debugging
  - FAQs
  - Common Errors
  - Heapchecker
  - **GDB**
- Networking: Sockets API

Useful gdb Techniques

- Locate segfaults
  - 'gdb mdriver' and then ‘run’
  - If it’s not in the heapchecker, put your heapchecker before that line and use it
- Backtrace
  - ‘backtrace’ or ‘bt’
  - A summary of how your program got where it is. One line per frame, starting with the currently executing frame
  - e.g.

```
#0 m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8) at
builtin.c:993
#1 0x6e38 in expand_macro (sym=0x2b600) at macro.c:242
#2 0x6840 in expand_token (obs=0x0, t=177664, td=0xf7fffb08)
at macro.c:71
```

Useful gdb Techniques

- Conditional breakpoints
  - Tell the debugger to break only if <variable> is equal to <value>

```
(gdb) b *0xdeadbeef
Breakpoint 1 at 0xdeadbeef
(gdb) condition 1 <variable> == <value>
```

- Watch points
  - When a location in memory is written you are notified and execution is suspended just like for a break point
  - To break when the integer at address 0x12345678 is modified, use

```
watch *((int *) 0x12345678)
```

- A simple gdb tutorial:
  - http://www.cs.cmu.edu/~gilpin/tutorial/

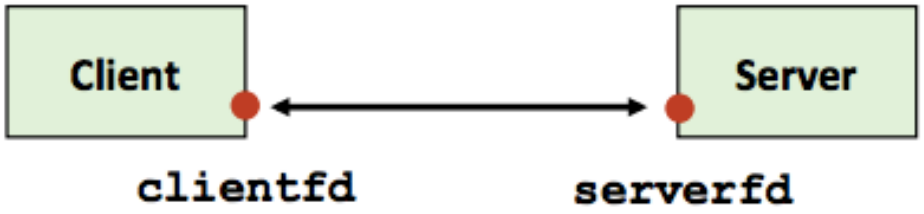
Questions?

Today

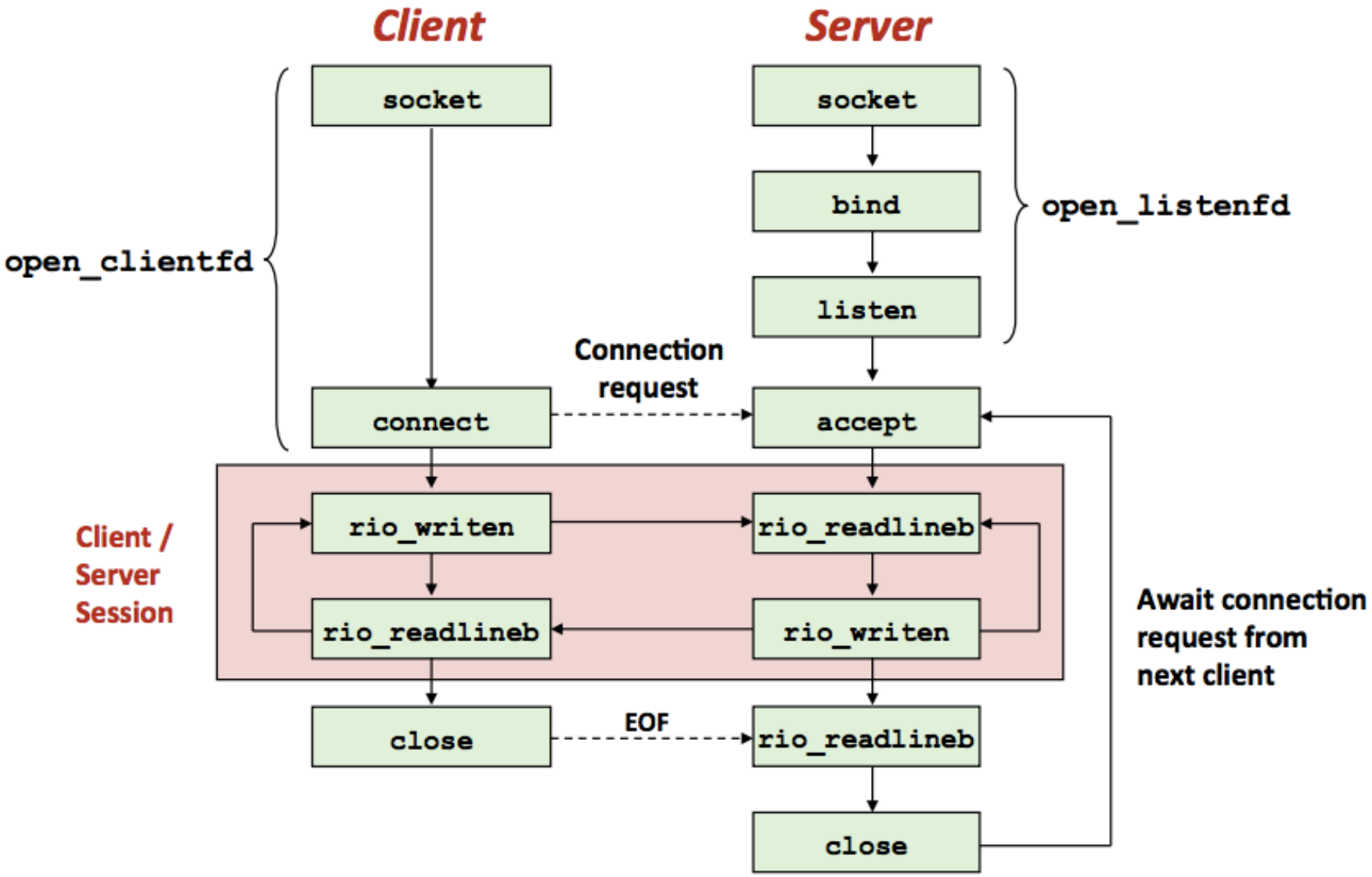
- Malloc Debugging
  - FAQs
  - Common Errors
  - Heapchecker
  - GDB
- **Networking: Sockets API**

Sockets

- To get a struct hostent for a domain name
  - `struct hostent * gethostbyname(const char *name);`
  - not threadsafe, threadsafe version is gethostbyname_r
- What is a socket?
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
  - (all Unix I/O devices, including networks, are modeled as files)
- Clients and servers communicate with each other by reading from and writing to socket descriptors



Overview of the Sockets Interface



Sockets API

- int socket(int domain, int type, int protocol);
- int bind(int socket, const struct sockaddr *address, socklen_t address_len);
- int listen(int socket, int backlog);
- int accept(int socket, struct sockaddr *address, socklen_t *address_len);
- int connect(int socket, struct sockaddr *address, socklen_t address_len);
- int close(int fd);
- ssize_t read(int fd, void *buf, size_t nbyte);
- ssize_t write(int fd, void *buf, size_t nbyte);

Sockets API

- int socket(int domain, int type, int protocol);
  - used by both clients and servers
  - int sock_fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  - Create a file descriptor for network communication
  - One socket can be used for two-way communication

Sockets API

- int bind(int socket, const struct sockaddr *address, socklen_t address_len);
  - used by servers
  - ```
    struct sockaddr_in sockaddr;
    memset(&sockaddr, 0, sizeof(sockaddr);
    sockaddr.sin_family = AF_INET;
    sockaddr.sin_addr.s_addr = INADDR_ANY;
    sockaddr.sin_port = htons(listenPort)
    err = bind(sock_fd, (struct sockaddr *) sockaddr,
    sizeof(sockaddr));
    ```
  - sock_fd: file descriptor of socket
  - my_addr: address to bind to, and information about it, like the port
  - addrlen: size of addr struct
  - Associate a socket with an IP address and port number
    ```

Sockets API

- int listen(int socket, int backlog);
  - used by servers
  - err = listen(sock_fd, MAX_WAITING_CONNECTIONS);
  - socket: socket to listen on
  - backlog: maximum number of waiting connections
- int accept(int socket, struct sockaddr *address, socklen_t *address_len);
  - used by servers
  - ```
    struct sockaddr_in client_addr;
    socklen_t my_addr_len = sizeof(client_addr);
    client_fd = accept(listener_fd, &client_addr, &my_addr_len);
    ```
  - socket: socket to listen on
  - address: pointer to sockaddr struct to hold client information after accept returns
  - return: file descriptor
    ```

Sockets API

- int connect(int socket, struct sockaddr *address, socklen_t address_len);
  - used by clients
  - attempt to connect to the specified IP address and port described in **address**
- int close(int fd);
  - used by both clients and servers
  - (also used for file I/O)
  - fd: socket fd to close

Sockets API

- ssize_t read(int fd, void *buf, size_t nbyte);
  - used by both clients and servers
  - (also used for file I/O)
  - fd: (socket) fd to read from
  - buf: buffer to read into
  - nbytes: buf length
- ssize_t write(int fd, void *buf, size_t nbyte);
  - used by both clients and servers
  - (also used for file I/O)
  - fd: (socket) fd to write to
  - buf: buffer to write
  - nbytes: buf length

Questions?