CPSC 213

January 8th - Unit 0: Introduction

- Staff
    - Instructor: Geoffrey Tien
        - gctien@cs.ubc.ca
    - Course Coordinator: Gale Chen
        - cpsc213-admin@cs.ubc.ca
        - Administrative queries
- Course Logistics
    - Canvas
    - PrarieLearn
    - iClicker Cloud
    - Piazza
- Grading
    - Participation
        - 1% - lab participation (80% of 12 labs)
        - 1% - lecture iClicker questions for 80% lectures
    - Assignments
        - 25% for 10 assignments
        - best of 8 of A1-A9 + A10
    - Quizzes
        - 33% for 3 quizzes
        - in-person in CBTF (self-schedule 1-hour slot during quiz weeks)
    - Final exams
        - 40% final exam
    - MUST
        - Achieve at least 50% on quiz/final average
        - Submit at least 80% of homework
- What is a computer?
    - Computer
        - Stores information and can perform a simple set of operations

        - Uses existing information plus **work** to produce a new information
        - Performs a sequence of operations determined by stored instructions
            - stored instructions are part of the machine's stored information
        - Can be modeled by a Turing machine
    - Computer System
        - Abstraction of a computing machine
        - Used in various combinations by programmers
        - Abstraction examples
            - Hardware instruction-set architectures
            - Operating systems
            - Compilers and high-level languages
            - Middleware and libraries
        - Programming
            - Building new abstractions in software
            - Use existing abstractions
    - Why care?
        - Mental model of computation is rooted in the machine
        - Understanding abstraction will
            - Understand and respect limits
            - Be more flexible and powerful
            - Holistic view of computational stack

```
public class A {
  static int s = 0;
  int i;
  public A() {
    s = s + 1;
    i = s;
  }
  public static void main(String[] args) {
    A a0 = new A();
    A a1 = new A();
    a0 = a1;
    a1.i = a1.i + 1;
    System.out.printf("%i, %i", a0.i, a1.i);
  }
}
```

In the code on the left, what prints when `main()` executes?

A. `0, 2`
B. `1, 3`
C. `2, 3`
D. `3, 3`
E. Something else

Draw a picture!

- - Correct Answer: D
    - Explanation: Notice that s is a singleton (static variable), and a0, a1 both refer to the same object (a0 = a1).
- Inside the machine

- All states are stored in a single, **main memory**
- Each *byte* has an unique numeric *address*
- A *variable* is a name for a memory location that has an **address, size, value**
    - Statistically allocated variables: address is *specified at compilation time*
    - Dynamically allocated variables: address is *determined at runtime*
- A *pointer* is a variable whose value is a memory address

```
public class B {
  int val;
  B(int val) {
    this.val = val;
  }
  int add(B x, B y) {
    this.val = this.val + x.val;
    this.val = this.val + y.val;
    return this.val;
  }
  public static void main(String[] args) {
    B a = new B(4);
    B b = a;
    B c = a;
    System.out.println( a.add(b,c) );
  }
}
```

In the code on the left, what is printed when `main()` executes?

A. 4
B. 8
C. 12
D. 16
E. 20

Draw a picture!

- - Correct Answer: D
    - Explanation: All a, b, and c are referring to the same object. Thus when add() method was called, this.val will be updated for all three references. this.val = 4 → 8 → 16; a, b, and c are pointers
- Inside the machine (Continued)
    - A new object is created only when the "new" keyword is called
    - Object is identified by its *memory address*, a number
    - **Pointer variables** are stored in memory too
        - Their value is the **memory address** of some object

January 10th - Unit 1a: Numbers and Memory

- Simple Computing Machine
    - Memory
        - Stores data encoded as bits
        - Program instructions and state
    - CPU
        - Reads instruction and data from memory

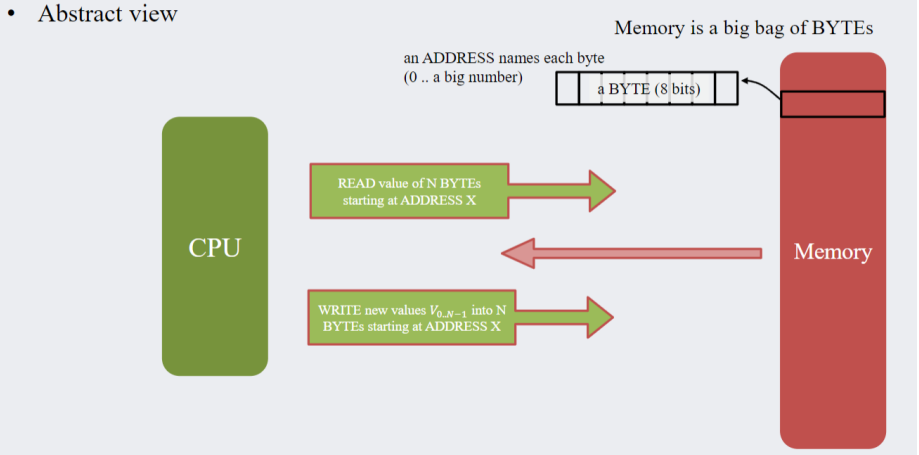
        - Performs specified computation and writes result back to memory
    - Example
        - $C = A + B$
        - Memory: stores add instruction, variables A, B, C
        - CPU: reads instruction and values of A, B; adds values; writes result to C
- Memory (Endianness will be talked about later)
    -

ADDRESS	BIT 0	1	2	3	4	5	6	7
0000	*	*	*	*	*	*	*	*
0001	*	*	*	*	*	*	*	*
0010	*	*	*	*	*	*	*	*
0011	*	*	*	*	*	*	*	*

    - CPU: READ value of N BYTEs starting at ADDRESS X;

      Memory: Gives the requested data to CPU

      CPU: WRITE new values of N BYTEs starting at ADDRESS X
    - • Abstract view



    - Naming
        - Unit of addressing is **byte** (8 bits)
        - Every byte of memory has an **unique address**
        - Some machines (this class: SM213) have 32-bit memory addresses, some have 64
            - 2^{32} bytes in memory is about 4GB.
    - Access
        - Many things are too big to fit in a single byte
        - CPU accesses memory in **contiguous, power-of-two-sized** chunks of bytes
        - Address of a chunk is address of its **first byte**
    -

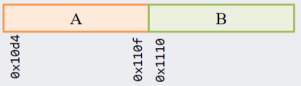
# BYTES	# BITS	C	JAVA	ASM
1	8	char	byte	b - byte

# BYTES	# BITS	C	JAVA	ASM
2	16	short	short	w - word
4	32	int	int	l - long
8	64	long	long	q - quad

        - We will only use 32-bit integers (4 bytes)
- Numbers and Representation
    - Integer value of a chunk of bytes
        - Binary to Decimal
    - The bits themselves
        - Use power-of-two sized way to identify addresses
        - Binary: `0b`
        - Octal: `0`
        - Hexadecimal: `0x`
            - 4 bits = hex "digit"
            - 32-bit address = 8-hexit address
            - 1 byte = 2 hexits

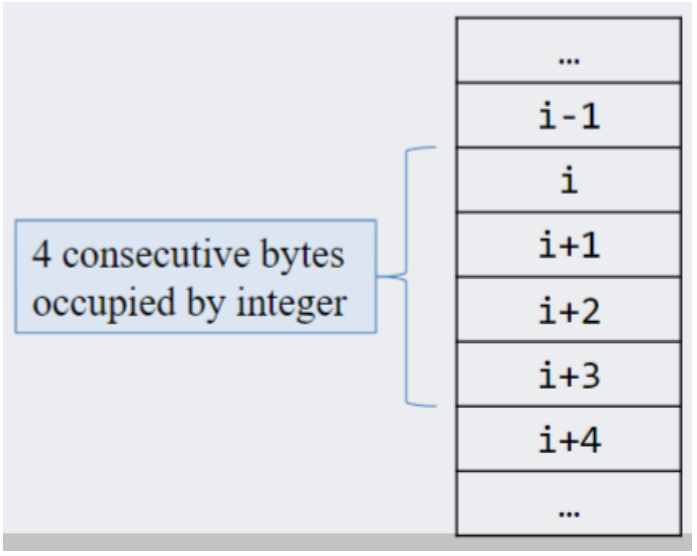
> Which of the following statements is true?
>
> A. The Java constants `16` and `0x10` are exactly the same integer
> B. The Java constants `16` and `0x10` are different integers
> C. Neither of the statements above is always true
> D. I don't know
> E. I'm just choosing E for participation credit

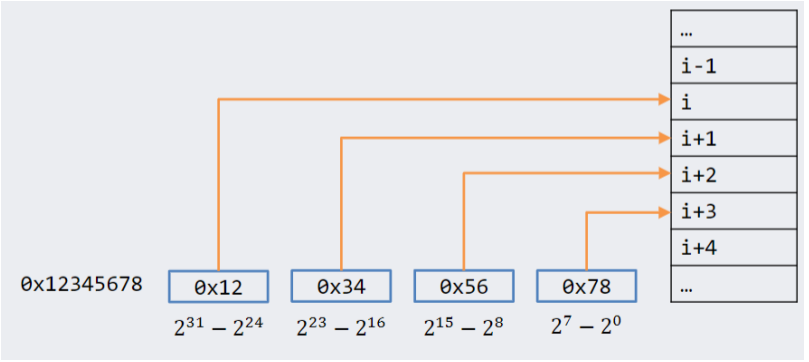
            - Correct Answer: A
    - Hexadecimal operations
        - For memory addresses
        - Addition / Subtraction
            - Convert to decimal
            - Calculate in hex directly
            - Subtraction: 2's complement
            - Carry for addition, Borrow for subtraction

> - Object A is at address `0x10d4`, and object B is at `0x1110`. They are stored contiguously in memory (i.e. they are adjacent to each other without any gap). How big is A?
>
> 
>
>
> A. 16 bytes
> B. 48 bytes
> C. 60 bytes
> D. 80 bytes
> E. Not enough information to tell

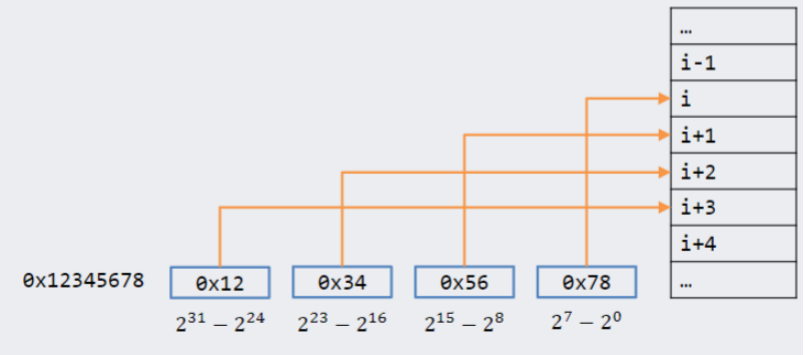
            - Correct Answer: C
            - Explanation: 0x1110 - 0x10d4 = 0x003c = 60_{10}

January 12th - Unit 1a: Cont.d

- Making Integers from Bytes
    - Each memory address hold 1 byte
    - Architectural decision
        - Integers = multiple bytes of memory
        - 32-bit integer (**int** in Java / C)
            - 4 bytes in memory
            - If memory address start at i, we also need addresses $i+1, i+2, i+3$
            - 

- Endianness
    - Big-endian
        - First byte stored is the most significant byte (bit 31 - 24)
        - Used in old IBM servers, network connections
        - 

        - If integer is signed, first byte can tell us +/-.
    - Little-endian
        - First byte is the least significant byte (bit 8-0)
        - Used in most Intel-based system

- 

    - First byte can tell us even/odd; In C, we can use 0/1 represent F/T, then the least significant byte is useful, as we only need the least significant byte.

- What is the Little-endian 4-byte integer value at address `0x4`?

A. `0xc1406b37`
B. `0x1c04b673`
C. `0x73b6041c`
D. `0x376b40c1`
E. `0x739a8732`

Address	Value
0x0:	0xfe
0x1:	0x32
0x2:	0x87
0x3:	0x9a
0x4:	0x73
0x5:	0xb6
0x6:	0x04
0x7:	0x1c

    - Correct Answer: B
- Address alignment
    - 4-byte integer can be placed at almost any address
    - Forcing address alignment is better for hardware
        - Consider an 8-byte memory, if we place the integer on addresses 1, 2, 3, then we cannot place another 4-byte integer, this leads to memory waste (decrease in efficiency)
        - Aligned address: addresses whose numeric value is a **multiple** of the object size (arrays and structs can be more complicated)
        - Smaller data types can fit inside larger things
            - 2× shorts = 1× int
            - Significant for arrays, as we can move in multiples of the object size.
        - Some CPUs do not support misaligned address
- Power-of-2 alignment
    - Address computation in arrays is achieved by shifting **bits**.
        - If we want to access element i, where each element occupies 2^j bits, then:

```
&a[i] == &a[0] + i * (size == 2^j) == &a[0] +
i << j
```

        - Memory implementation detail
            - Memory is organized internally into larger chunks called **blocks**
            - Suppose a block is 16 bytes
            - Every memory access requires accessing one of these blocks
                - This relates to memory caches (caching - provide these blocks to CPU when requested)
- Alignment and Memory access
    - CPU
        - Read/write N bytes starting at address A
    - Memory converts above request into
        - Read/write N bytes starting at the O^{th} byte of block B
            - O - block offset; B - block number
            - Blocks are numbered so that Block 0 contains address $0 \dots 15$
            - If generally, Block B contains address $X \dots X + blocksize - 1$
            - Then, $X \leq A \leq X + blocksize - 1$
        - So to access this address given A, we need to find the Block number and Block offset, then consider a tuple that satisfy some function of A: $(B, O) = f(A)$
    - The process is simplified if
        - Block size = power of 2
        - Object size (N) = power of 2
        - Absolute address (A) = aligned to N
        - e.g Assume Block size = 8; $N = 4$. We want 4 bytes starting at address $A = 84_{10}$
            - $(B, O) = f(84) \rightarrow (B, O) = (10, 4)$, want 4 bytes from 10^{th} block offset by 4 bytes
                - $B = A/8, O = A\%8$
                - Since block size is power of 2, calculating B is just a right shift
                - Calculating O is obtained by **masking** the least significant bits (in this case, the last 3 bits)

Jan. 15th - Unit 1a: Cont.d

- Address alignment
  - Which of the following statement(s) is/are true?

    A. The address 6 (110_2) is aligned for addressing a `short`
    B. The address 6 (110_2) is aligned for addressing an `int` (i.e. 4 bytes)
    C. The address 20 (10100_2) is aligned for addressing a `long` (i.e. 8 bytes)
    D. Two or more of the above
    E. None of the above
  - Correct answer: A
  - Explanation: **short** are block sizes of 2 bytes, and 6 % 2 = 0; the other data types have remainders.
  - Which of the following statement(s) is/are true?

    A. The address `0x14` is aligned for addressing a `short`, but not an `int`
    B. The address `0x14` is aligned for addressing a `short` and `int`, but not a `long`
    C. The address `0x14` is aligned for addressing `short`, `int`, and `long`
    D. None of the above
    E. P-A-R-T-I-C-I-P-A-T-I-O-N!
  - Correct answer: B
  - Explanation: 0x14 = 20; only 2 bytes and 4 bytes divide 20.
- Extending integers (Small type to large type)
  - Extending
    - Increasing the number of bits (bytes) used to store an integer
    - ```
      byte b = -6;
      int i = b;
      out.printf("b: 0x%x, %d; i: 0x%x, %d", b, b,
      i, i);
      ```
      - Output: "b: 0xfa, -6; i: 0xfffffffa, -6"
  - Signed extension
    - Used with signed numbers (e.g. in Java)
    - Copies signed bit into upper empty bits of the extended number (use 2's complement to ensure the same number)
  - Zero extension
    - Used with unsigned number (e.g. in C)
    - Sets upper empty bits to 0
    - Can be forced by **masking** using bitwise AND operator
    - ```
      int u = b & 0xff;
      printf("u: 0x%x, %d\n", u, u);
      ```

- Output: "u: 0xfa, 250"

- Truncating integers (Large type to small type)
  - We can use fewer bits to represent a value
    - ```
      int i = -6;
      byte b = i;
      out.printf("b: 0x%x, %d; i: 0x%x, %d");
      ```
      - This piece of code segment will run with error "possible loss of precision"
      - Cast explicitly to hide such a warning:
        - ```
          byte b = (byte) i
          ```
          - This would run without big issues
    - If we use $i = 256 = 0x100$, b would only be $0x00 = 0$; If we use $i = 128$, b would be treated as 0x80, with leading 0s, which is $b = -128$.

- Bit operations
  - a << b: shift all bits in a to the left b times, fill remaining right bits with zero
  - a >> b: shift all bits in a to the right b times
    - C: if a is unsigned, zero extension, otherwise sign extension
    - Java: >> sign extension; >>> zero extension
  - a & b: bitwise AND
  - a | b: bitwise OR
  - a ^ b: bitwise XOR
  - ¬a: bitwise NOT
  - Bit shifts
    - Shifting to the left: multiplication by 2^b
    - Shifting to the right: division by 2^b
    - Negative numbers? (Signed shift ✓)

- What is the value of `i` after this Java statement executes?

```
int i = 0xff8b0000 & 0x00ff0000;
```

A. `0xffff0000`
B. `0x0000008b`
C. `0x008b0000`
D. `0xff8b0000`
E. None of the above

- Correct answer: C

- What is the value of `i` after this Java statement executes?

```
int i = 0x8b << 16;
```

A. `0x8b`
B. `0x0000008b`
C. `0x008b0000`
D. `0xff8b0000`
E. None of these

- Correct answer: C (Assumed leading 0s)

- What is the value of `i` after this Java statement executes?

```
int i = 0x0000008b << 16;
```

A. `0x008b`
B. `0x0000008b`
C. `0x008b0000`
D. `0xff8b0000`
E. None of these

- Correct answer: C

- What is the value of `i` after this Java statement executes?

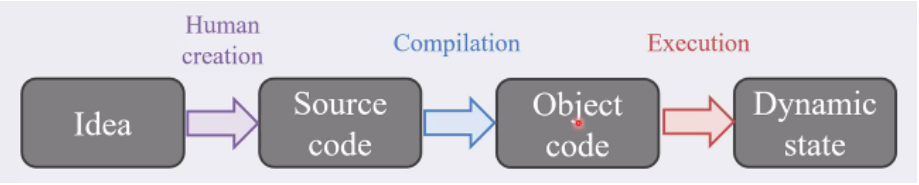
```
int i = ((byte) 0x8b) << 16;
```

A. `0x8b`
B. `0x0000008b`
C. `0x008b0000`
D. `0xff8b0000`
E. None of these
F. I don't know

- Correct answer: D
- Explanation: When 0x8b is casted into byte, it is interpreted as a negative value as there is a leading 1. Thus, when left shift 16 bits, we then would yield an integer 0xff8b0000 to preserve sign.

Jan. 17th - Unit 1b: Static Scalars and Arrays

- Develop a model of computation, Analyze the processor, Explore the language
- The CPU
  - CPU executes instructions
  - Execution
    - Fetch: Load the next instruction from memory
    - Decode: Figure out what the instruction is
    - Execute: Do what the instruction specifies
  - These executing steps are looped over and over again
  - Instructions
    - Read/load a value from memory
    - Write/store a value to memory
    - Add/subtract/AND/OR two numbers
    - Shift a number
    - Control flow
    - Some operations are carried out in Arithmetic & Logic Unit (ALU), used in execution stage
- Phases of computation
  - Human creation: design program in high-level language
  - Compilation: Convert high-level language to machine-executable text
  - Execution: a physical machine executes the code
    - Parameterized by input values **unknown** at compilation
    - Producing output values that are **unknowable** at compilation



    - Object code is binary machine code
  - Anything that **the compiler can compute** is called ***static***

    Anything that can **only be discovered during execution** is called ***dynamic***

- Consider the code below:

```
int a;
a = 5;
```

The value assigned to variable a is:

A. static
B. dynamic
C. neither static nor dynamic
D. it depends

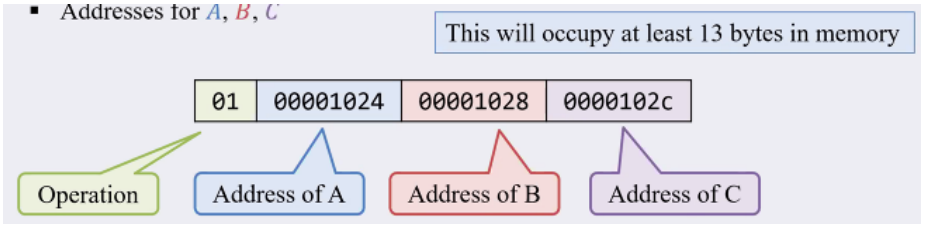
    - Correct answer: A
    - Explanation: Compiler gives space to a, then assigned value to a, so it is static, as it is in the source code.

- Assume that a variable of type `int` is assigned a value that is read from the user through an input dialog.

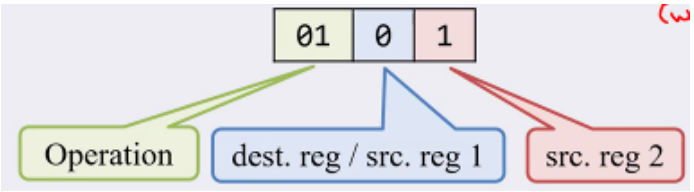
This value is:
A. static
B. dynamic
C. neither static nor dynamic
D. it depends

    - Correct answer: B
    - Explanation: Compiler still makes space for the variable, but cannot assign a value to the variable until runtime

- The processor (CPU)
    - Implements a set of instructions
    - Each instruction is implemented using **logic gates**
        - Built from **transistors**
    - Instruction design philosophies
        - **RISC**: fewer and simpler instructions makes processor design simpler
        - CISC: having more types of instructions and more complex instructions allows for shorter/simpler program code and simpler compilers
- First proposed instruction: +
    - We want to do: $A \leftarrow B + C$
    - Instruction parameters
        - Addresses of A, B, C: 32-bits
    - Instruction is encoded as a sequence of bits
        - Operation name
        - Addresses for variables

- 

- Accessing memory is SLOW
    - ~100 CPU cycles for every memory access
    - goal: fast programs avoid accessing memory when possible
- Big instructions are costly
    - Memory addresses are big (instruction is then big)
    - Big instructions → big programs
    - Reading memory is slow
    - Large instructions use more CPU resources (transfer, storage)
- General purpose registers
    - Register file
        - Small, fast memory in CPU itself
        - roughly **single-cycle** access
    - Register
        - Named by a number (0-15)
        - Size: architecture's common integer
- Improve using registers
    - Memory instructions handle only memory
        - Load data from memory to register
        - Store data from register to memory
    - Accessing data in registers
        - Small and fast
    - Then it becomes:
      

        - Takes only about 3 bytes
    - To further improve instruction size; we can share register for one source and destination
        - 

        - Takes only 2 bytes

- $B \leftarrow B + C$
- Minimize interaction with memory will increase processing speed
    - Suppose we are adding more than 2 variables
    - Direct memory access will require the above non-register-involved process to run at least twice
    - However, in the register-involved method, only memory accesses are loading each variable's address and store the final result to a designated memory
    - This will reduce the times to access memory, speeding up the program.
- Special purpose registers
    - e.g. Program Counter (PC): contains address of next instruction to execute

      Instruction Register (IR): contains the machine instruction that has just been fetched from memory
        - IR needs to be large enough to contain the longest instruction

Jan. 19th - Unit 1b: Cont.d

- Instruction Set Architecture (ISA)
    - Formal interface to a processor implementation
        - Defines the instructions the processor implements
        - Defines the format of each instruction (in binary/hexidecimal)
    - Types of instructions
        - Math and logic
        - Memory access
        - Control transfer: "goto" and conditional "goto"
    - Design
        - Instruction format
            - Sequence of bits: **opcode** and **operand** values
            - All represented as numbers → defines machine code
        - Assembly language
            - Symbolic representation of machine code
- Representing Instruction Semantics
    - RTL: pseudo language to describe semantics
        - Right-to-left format, describes machine steps
    - Syntax
        - Line form: $LHS \leftarrow RHS$

- LHS is a memory or register that receives a value
- RHS is a constant, memory, register, or expression on 2 registers
- $m[a]$ is a memory in address a
- $r[i]$ is a register with number i (0 to 7, for general purpose registers)

- Example 1
  - $r[0] \leftarrow$ 0x2000

    Register 0 receives 0x2000 (as an address)

    $r[1] \leftarrow m[r[0]]$

    Register 1 receives memory whose address is $r[0]$

    $r[2] \leftarrow r[2] + r[1]$

    Register 2 increases by the value of Register 1
  - ```
    // Assume at 0x2000 address, we have a value of 7
    int n = 7;
    int r_2 = 10;

    // A pointer that represents Register 0, that stores the address of n
    int *r_0 = &n;
    // r_0 now stores 0x2000

    // Register 1 stores the value at the address stored in Register 0, which in this case, the value of variable n
    int r_1 = *r_0;
    // r_1 now stores 7;

    r_2 += r_1;
    // r_2 would now store the value of 17.
    ```

- Example 2
  - Assume at address 0x1234, value 22 is stored, consider the following instructions
  - $r[0] \leftarrow 10$

    $r[1] \leftarrow$ 0x1234

    $r[2] \leftarrow m[r[1]]$

    $r[2] \leftarrow r[0] + r[2]$

    $m[r[1]] \leftarrow r[2]$
  - Step by step:
    ```

- Register 0 stores the value of 10
- Register 1 stores the address of 0x1234
- Register 2 stores the value at the address stored in Register 1 (0x1234), which is 22
- Register 2 increases the value by the value stored in Register 0, which is 10, resulting in 32
- Value at memory stored in Register 1 (0x1234) now stores the value in Register 2, which is 32

- Variables
  - Named storage locations for values
  - Features
    - Name - static
    - Type/Size - static
    - Scope (where in the code we can access the variables) - static
    - Lifetime (availability of the variable at a certain part of the program) - dynamic
    - Memory location (address) - dynamic/static
    - Value - dynamic/static
  - Static variables (built-in types)
    - In Java
      - Static data members are allocated to a class, not an object
      - Store built-in scalar types or references to arrays or objects
    - In C
      - Global variables and any other variable declared static
      - Can be static scalars, arrays or structs or pointers
      - ```
        int a;
        int b[10];

        void foo() {
            a = 0;
            b[a] = a;
        }
        ```
    - Allocation
      - Assigning a memory location to a variable
    - Static vs. Dynamic computation
      - Global/static variables can exist before program starts and live until after it finishes
        ```

- Compiler allocates variables, giving them a constant address
- No dynamic computation is required to allocate, they just exist
- Compiler tracks free space during compilation - it is in control of program's memory

- Assume a is a global variable in C. When is space for a allocated? In other words, when is its address determined?

A. The compiler assigns the address when it compiles the program
B. The compiler calls the memory to allocate a when it compiles the program
C. The compiler generates code to allocate a before the program starts running
D. The program locates available space for a before it starts running
E. The program locates available space as soon as the variable is used for the first time

- Correct answer: A
- Explanation: B. Memory itself is not involved; C. Compilation happens even before program starts running; D. The program is already running; E. Dynamic allocation

Jan. 22nd - Unit 1b: Cont.d

- What is a proper RTL instruction for
  `a = 0;`

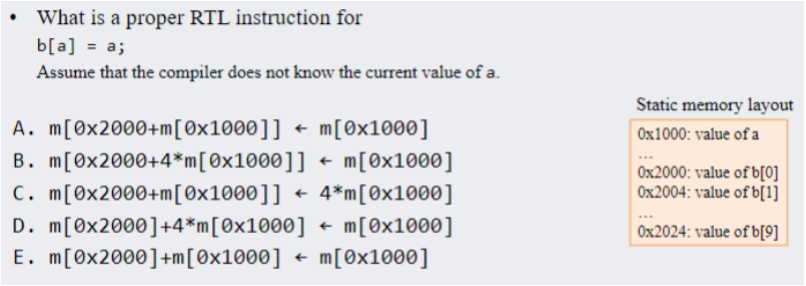
```
A. r[0x1000] ← 0
B. m[0x1000] ← 0
C. 0x1000 ← r[0]
D. m[r[0x1000]] ← 0
E. 0x1000 ← m[0]
```

Static memory layout

0x1000: value of a
...
0x2000: value of b[0]
0x2004: value of b[1]
...
0x2024: value of b[9]

- Correct Answer: B
- Variables - Cont.d
  - Static variable access
    - Observations
      - Value of a is generally unknown at compilation time
      - Even though addresses of $b[0\ldots]$ are known **statically**, which one being used is **dynamic**
      - In general, given $b[a] = a$, the compiler does not know the address of $b[a]$
    - Array access is computed from a **base address** and **index**
      - Address = base + offset
      - Offset = index × element size

        - The base address and element size are static, but the index is dynamic



        - Correct answer: B
- Alternative for instructions sets
    - For $a = 0$
        - Option 1:
            - Static address and value: m[0x1000] ← 0x0
            - 9 bytes (instruction code + two integers)
            - If the instruction is $a = b$, b can come from user input, so we may require dynamic value
        - Option 2:
            - Static address, dynamic value:

              r[0] ← 0x0

              m[0x1000] ← r[0]
            - 5 bytes for immediate (constant) value instruction (instruction + register + one integer)
            - 5 bytes for memory instruction (instruction + register + one integer)
            - op codes are always half a byte, register is always half a byte
        - Option 3:
            - Dynamic address, dynamic value:

              r[0] ← 0x0

              r[1] ← 0x1000

              m[r[1]] ← r[0]
            - 2 bytes for memory instruction: instruction code + two registers
    - For $b[a] = a$
        - Option 1:
            - Static base address, index, and value:

              m[0x2000 + 4*m[0x1000]] ← m[0x1000]
            - 13 bytes (instruction + 3 integers)

- Option 2
    - Calculate address explicitly

      r[0] ← 0x1000

      r[1] ← m[r[0]]

      r[3] ← r[1] << 2

      r[2] ← 0x2000

      r[2] ← r[2] + r[3]

      m[r[2]] ← r[1]
    - Array access are very common: calculating address every time is costly
- Option 3
    - Dynamic base address, index, and value:

      r[0] ← 0x1000

      r[1] ← m[r[0]]

      r[2] ← 0x2000

      m[r[2] + r[1] << 2] ← r[1]
    - The indexed memory instruction require 2 bytes: instruction + 3 registers

Jan. 24th - Unit 1b: Cont.d

- ISA design goals (RISC paradigm)
    - a = 0; m[0x1000] ← 0x0
        - r[0] ← 0x0
        - r[1] ← 0x1000
        - m[r[1]] ← r[0] (store)
    - b[a] = a; m[0x2000 + 4 * m[0x1000]] ← m[0x1000]
        - r[0] ← 0x2000
        - r[1] ← m[r[0]] (load)
        - r[2] ← 0x2000
        - m[r[2] + 4 * r[1]] ← r[1]
    - Minimizes the number of memory instructions in ISA (at most 1 memory access per instruction)
    - Minimize the total number of instructions in ISA
    - Minimize the size of each instruction
- Instructions so far
    - Load a constant value into a register: r[0] ← v
    - Store a value in a register into memory at some address: m[r[y]] ← r[x]

- Load a value in memory into a register: r[x] ← m[r[y]]
- Store value in a register into memory at address in register * 4 + base address
    - m[r[z] + 4 * r[x]] ← r[y]
- Load value in memory at address in register * 4 + base address into register
    - r[y] ← m[r[z] + 4 * r[x]]
- **SM213 ISA**

  The first five instructions so far

	Name	Semantics	Assembly	Machine
1	load immediate	r[d] ← v	ld $v, rd	0d-- vvvvvvvv
3	load base + offset	r[d] ← m[r[s]]	ld (rs), rd	1?sd
5	load indexed	r[d] ← m[r[s]+4*r[i]]	ld (rs, ri, 4), rd	2sid
2	store base + offset	m[r[d]] ← r[s]	st rs, (rd)	3s?d
4	store indexed	m[r[d]+4*r[i]] ← r[s]	st rs, (rd, ri, 4)	4sdi

- s: source; i: index; d: destination; ?: 4 bits to encode offset

- Code translation
    - High-level code → RTL → Assembly (1 : 1 correspondence) → Machine code (1 : 1 correspondence)
    - .pos: position directive, next item places into memory at the specified address.

```
int i;
int a[10];

a[2] = a[i];
```

Name	Semantics	Assembly
load immediate	r[d] ← v	ld $v, rd
load base + offset	r[d] ← m[r[s]]	ld (rs), rd
load indexed	r[d] ← m[r[s]+4*r[i]]	ld (rs, ri, 4), rd
store base + offset	m[r[d]] ← r[s]	st rs, (rd)
store indexed	m[r[d]+4*r[i]] ← r[s]	st rs, (rd, ri, 4)

- Which correctly implements `a[2] = a[i];`?

A.
```
st ($a, $i, 4), ($a, 2, 4)
```

B.
```
ld ($a, $i, 4), r0
st r0, ($a, 2, 4)
```

C.
```
ld $a, r0
ld $i, r1
ld (r1), r1
ld (r0, r1, 4), r2
st r2, 2(r0)
```

D.
```
ld $a, r0
ld $i, r1
ld (r1), r1
ld (r0, r1, 4), r2
ld $2, r1
st r2, (r0, r1, 4)
```

E. None of these / I don't know

- - Correct Answer: D
    - Explanation: C. offset must be multiple of 4; D.

```
r0 <- &a;               # r0: address
of a
r1 <- &i;               # r1: address
of i
r1 <- *r1;              # r1: value of
i
r2 <- m[r0 + 4 * r1];   # r2: value of
a[i]
r1 <- 2;                # r1: 2
m[r0 + 4 * r1] <- r2    # a[2]: value
of r2
```

- SM213 ISA
  - Architecture
    - Register file: eight 32-bit general purpose registers (0-7)
    - CPU: one cycle per instruction (fetch + execute)
    - Main Memory: byte addressed, big endian integers
  - Instruction format
    - 2 or 6 byte instructions (each character is a hexit)
      - x-01, xxsd, x0vv, or x-sd vvvvvvvv
    - x is op-code
    - -: means unused
    - s and d are operand register numbers
    - vv, vvvvvvvv are immediate/constant values

Jan. 26th - Unit 1b: Cont.d

- Machine code
  - `[addr:] x-01 [vvvvvvvv]`
    - `addr:` sets starting address for subsequent instructions
    - `x-01` hex value of an instruction with opcode x and operands 0 and 1
    - `vvvvvvvv` hex value of optional extended value
- Assembly code
  - `[label:] [instruction | directive] [# comment]`
    - `directive :: (.pos number) | (.long number)`
    - `instruction :: opcode operand+`
    - `operand :: $literal | reg | offset(reg) | (reg, reg, 4)`
    - `reg :: r0..7`
    - `literal :: number`
    - `offset :: number`
    - `number :: decimal | 0xhex`

Name	Semantics	Assembly	Machine
load immediate	`r[d] ← v`	`ld $v, rd`	`0d-- vvvvvvvv`
load base + offset	`r[d] ← m[r[s]+(o=4*p)]`	`ld o(rs), rd`	`1psd`
load indexed	`r[d] ← m[r[s]+4*r[i]]`	`ld (rs, ri, 4), rd`	`2sid`
store base + offset	`m[r[d]+(o=4*p)] ← r[s]`	`st rs, o(rd)`	`3spd`
store indexed	`m[r[d]+4*r[i]] ← r[s]`	`st rs, (rd, ri, 4)`	`4sdi`

- 4 addressing modes
  - **immediate**: constant value stored as part of instruction
  - **register**: operand is register number; register stores value
  - **base+offset**: operand is register number; register stores memory address of value (+ offset = p * 4), p encodes 0...15, so offset ranges from 0 to 60.
  - **indexed**: two register-number operands; store base memory address and value of index

- Arithmetic

Name	Semantics	Assembly	Machine
register move	`r[d] ← r[s]`	`mov rs, rd`	`60sd`
add	`r[d] ← r[d] + r[s]`	`add rs, rd`	`61sd`
and	`r[d] ← r[d] & r[s]`	`and rs, rd`	`62sd`
inc	`r[d] ← r[d] + 1`	`inc rd`	`63-d`
inc address	`r[d] ← r[d] + 4`	`inca rd`	`64-d`
dec	`r[d] ← r[d] - 1`	`dec rd`	`65-d`
dec address	`r[d] ← r[d] - 4`	`deca rd`	`66-d`
not	`r[d] ← ~r[s]`	`not rd`	`67-d`

- Logical (shifting), NOP, and halt

Name	Semantics	Assembly	Machine
shift left	`r[d] ← r[d] << v=s`	`shl $v, rd`	`7dss (ss>0)`
shift right	`r[d] ← r[d] >> v=s`	`shr $v, rd`	`7dss (ss<0)`
halt	*halt machine*	`halt`	`F0--`
nop	*do nothing*	`nop`	`FF--`

- 4 incrementing/decrementing is useful when calling functions
- Shifting: distinguished by the sign of the last byte (2's complement)

- What instruction is represented by the bytes: `0x46 0x24`?

```
A. ld (r4, r4, 4), r6
B. st r4, (r6, r2, 4)
C. st r6, (r2, r4, 4)
D. st r6, 2(r4)
E. st r6, 8(r4)
```


Name	Semantics	Assembly	Machine
load immediate	`r[d] ← v`	`ld $v, rd`	`0d-- vvvvvvvv`
load base + offset	`r[d] ← m[r[s]+(o=4*p)]`	`ld o(rs), rd`	`1psd`
load indexed	`r[d] ← m[r[s]+4*r[i]]`	`ld (rs, ri, 4), rd`	`2sid`
store base + offset	`m[r[d]+(o=4*p)] ← r[s]`	`st rs, o(rd)`	`3spd`
store indexed	`m[r[d]+4*r[i]] ← r[s]`	`st rs, (rd, ri, 4)`	`4sdi`

- Correct answer: C
- Explanation: 0x4624 → m[r[d] + 4 * r[i]] ← r[s] → st r6, (r2, r4, 4)

- What instruction is represented by the bytes: `0x75 0xF8`?

```
A. shl $5, r8
B. shl $8, r5
C. shr $5, r8
D. shr $8, r5
E. shl $0xf8, r5
```

Name	Semantics	Assembly	Machine	
shift left	r[d] ← r[d] << v=s	shl $v, rd	7dss	(ss>0)
shift right	r[d] ← r[d] >> v=s	shr $v, rd	7dss	(ss<0)

- - Correct answer: D
  - Explanation: 0x75f8, f8 is negative → r[d] ← r[d] >>/<< v=s → r[5] ← r[5] >> f8 → shr $8, r5
- CPU implementation
  - Internal state
    - PC (program counter): address of **next** instruction
    - Instruction: the value of current instruction
  - Cycle stage
    - Fetch
    - Execute
  - Internal Registers
    - insOp0, insOp1, insOp2, insOpImm, insOpExt, pc
    - Read using get()
    - Change using set(value)
  - General purpose registers
    - reg.set
    - reg.get
  - Main Memory
    - mem.readInteger
    - mem.writeInteger
- Global dynamic arrays
  - Java
    - array variable stores reference to array allocated dynamically with **new** statement
  - C
    - array variables can store static arrays, or
    - pointers to arrays allocated dynamically with call to **malloc** library procedure
    - ```
      // dynamic array
      ```
      ```

```
int a;
int * b; // global variables still statically
allocated
         // pointer to integer (array)

void foo() {
    b = malloc(10 * sizeof(int)); // malloc
happens when program runs
    b[a] = a;
}

// static array
int a;
int b_data[10];
int * b = &b_data[0];
```

- C pointers
    - Use the term *pointer* instead of *reference* (in Java)
    - Declaration
        - the type is a pointer to the type of its elements, indicated with a *
        - store internally as an address, regardless of element type
    - Type safety
        - any pointer can be typecast to any other pointer type
    - Bounds checking
        - C performs no array bounds checking
        - out-of-bounds access manipulates memory that is not part of the array
        - Performance is better, but a major source of vulnerability
    - Dynamic allocation in C
        - done by **malloc**
        - Returns the **address of a contiguous block of bytes**, with **no associated type or initialization**
            - Element type and dereferencing determines how the binary data are interpreted
        - Dynamic allocation will be discussed in more details in future units
    - Static vs. Dynamic allocation
        - For static arrays, the compiler allocates the array
        - For dynamic arrays, the compiler allocates a pointer

- The dynamic array is allocated by **malloc**, e.g. at address 0x3000
- Variable **b** is set to the memory address of this array

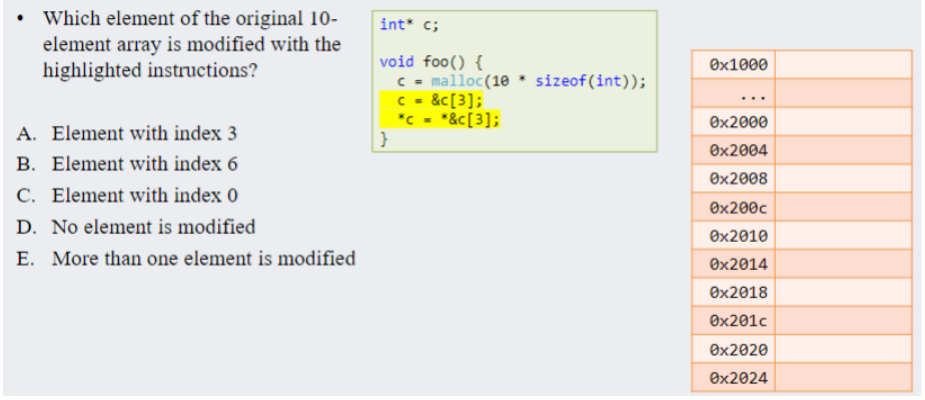
Jan. 29th - Unit 1b: Cont.d and Unit 1c: Instance Variables and Dynamic Allocation

- Summary: Arrays in Java and C
    - Similarities
        - Array: list of items of the same type
        - Array elements are named by non-negative integers starting at 0
        - syntax for accessing element **i** of array **b** is **b[i]**
    - In Java
        - variable stores a reference to the array
    - In C
        - variable can store a pointer to the array, or the array itself
            - distinguished by variable type (e.g. pointer variable vs. array variable)
        - array element access via pointer can be done by
            - **b[i]**
            - *(b + i)
- Declaring C pointers
    - Declaration
        - ```
          TYPE * ptr;
          ```
            - TYPE: means ptr is a pointer to type TYPE
        - ```
          int * x, * y;
          ```
    - Pointers can be assigned the address of a variable of type TYPE
        - ```
          int a = 5;
          int * ptr_a = &a;
          ```
    - Access the value being pointed by **dereferencing** with *
        - ```
          int a = 5;
          int * ptr_a = &a;
          *ptr_a = 37;
          ```

- Pointer Arithmetic
    - Purpose: an alternative way to access array elements compared to a[i]
    - Address are numbers
        - Can add / subtract an integer index to a pointer (address)
            - Results in the address of something else
            - Value of the pointer is offset by **index** × **size** of the pointer's referent (+ 3, is equivalent to a pointer value 12 bytes larger than the original)
        - Subtracting two pointers of the same type
            - Gives us the number of elements between two addresses
            - &a[7] - &a[5] == 2 == (a + 7) - (a + 5)
    -

```
int * a;
a[7] = 5;

int * a;
*(a + 7) = 5;
```

    - 

    -
        - Correct answer: A (c's base address has been changed, 6th element of the original c is now c[3])
        - Explanation:

```
int * c;

void foo() {
    c = malloc(10 * sizeof(int)); //
Dynamically allocate 10-element array
    // c == 0x2000

    c = &c[3];  // c = 0x200c (0x200c becomes
the new base address)
    *c = *&c[3]; // *c == c[3]
                 // *&c[3] == c[6]
}
```

- Summary
    - C yes, Java no
        - Static arrays
        - Arrays can be accessed using pointer dereference operator
        - pointer arithmetic
    - C no, Java yes
        - Type-safe dynamic allocation
        - Automatic array bounds checking
- Unit 1c - Instance Variables
    - Variables that are an instance of a class or struct
        - Created dynamically (usually)
        - Many instances of the same class/struct can coexist
    - Java vs C
        - Java: objects are instances of non-static variables of a class
        - C: structs are named variable groups, instance also called a struct
    - Accessing an instance variable
        - requires a reference/pointer to a particular object/struct
        - then variable name chooses a variable in that object/struct
- Structs in C
    - A struct is a collection of variables of arbitrary type
        - allocated and accessed together
        - only variables, no functions
    - Declaration
        - similar to declaring a Java class without methods
        - name is "struct" plus name provided by programmer
    - The definition of a struct only describes layout of member variables in memory when an instance is created
    - ```
      struct D {
          int e;
          int f;
      }

      struct D d0;    // Static declaration
      struct D * d1;  // Dynamic declaration

      d0.e = d0.f;    // Static access
      d1 -> e = d1 -> f;  // Dynamic access
      ```
      ```

- Static struct are allocated by the compiler
    - ```
      .pos 0x1000
      d0.e:                .long 0xffffffff
      .pos 0x1004
      d0.f:                .long 0xffffffff
      ```
    - Address of the first member is the same as address of the struct itself
- Dynamic struct are allocated runtime
    - the variable that stores the struct pointer may be static or dynamic
    - the struct itself is allocated when the program calls **malloc**
    - ```
      .pos 0x1000
      d1:                  .long 0x00001000

      .pos 0x2000 (malloc)
      d1->e:               .long 0xffffffff
      ```
    - ```
      struct D * d1;

      void foo() {
          d1 = malloc(sizeof(struct D));
      }
      ```
- Struct Access
    - Struct members can be accessed using **offset** from base address
        - Offset to each member from base of struct is also **static**
    - As before, static and dynamic differ by an extra memory access

Jan. 31st - Unit 1c: Cont.d

- Memory addressing modes
    - Scalar
        - ```
          i = a;
          ```
        - Address in register
        - Access memory at address in register: ld (r1), r0
    - Arrays
        - ```
          i = a[j]
          ```
          ```

- **base address** in register
- dynamic **index** in register
- access memory at base + index × element size: ld (r1, r2, 4), r0

- Struct members

```
i = a.j;
i = b -> k;
```

- Base address in register
- Static **offset**
- Access memory at base plus offset
- Equivalent to array access with static index (index can be used but requires extra register, index must be manually computed)

Given:

```
struct X {
  int i;
  int j;
}
```

```
struct X s;  // s is a global variable

ls $s, r0  # r0: address of s
```

Which of the following loads `s.j` into `r1`?

A.
```
ld $4, r1
```

B.
```
ld 4(r0), r1
```

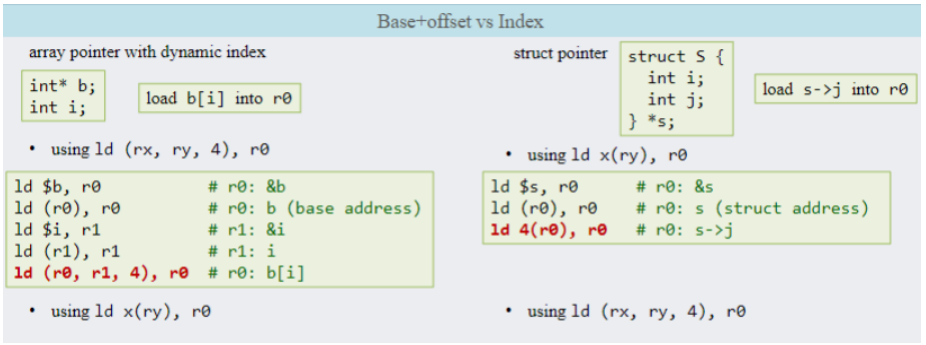
C.
```
ld (r0), r1
add 4, r1
```

D.
```
ld (r0), r0
add 4, r0
ld (r0), r1
```

E. None of these

- Correct answer: B

- Offset approaches



- Array pointer with dynamic index: **cannot** use "ld x(ry), r0", because "x" here is static, but we need it to be dynamic
- Struct pointer: **can** use "ld (rx, ry, 4), r0", because the offset required is static, we can use a dynamic register to offset it. "ry" would contain the "index" of "j", which is 1.

From previous slide:

array pointer with dynamic index

```
int* b;
int i;
```

load b[i] into r0

- using ld (rx, ry, 4), r0

```
ld $b, r0           # r0: &b
ld (r0), r0         # r0: b (base address)
ld $i, r1           # r1: &i
ld (r1), r1         # r1: i
ld (r0, r1, 4), r0  # r0: b[i]
```

Suppose we have run the first 4 assembly statements. How else can we load b[i] into r0?

A.
```
ld i(r0), r0
```

B.
```
ld r1(r0), r0
```

C.
```
shl $2, r1
ld r1(r0), r0
```

D.
```
shl $2, r1
add r0, r1
ld (r1), r0
```

E.
```
add r0, r1
ld (r1), r0
```

- - Correct answer: D
  - Explanation: r1 becomes 4 * i; r1 becomes r0 + 4 * i; r1 is now the address of b[i]

- Variations in struct declaration
  - Struct variables can be declared inside other structs
    -
```
struct D {
    int e;
    int f;
}

struct X {
    int i;
    struct D d;
    int j;
}

struct Y {
    int i;
    struct D * d;
    int j;
}

struct Z {
    int i;
    struct Z * z;
    int j;
}
```

      - struct Z needs to be terminated somehow
  - struct members can be arrays or pointers

```
struct W {
    int i;
    int a[10];
    int j;
}

struct U {
    int i;
    int * a;
    int j;
}
```

- What is the offset (in bytes) of member `j` in `struct X` below?

```
struct D {
  int e;
  int f;
};
```

```
struct X {
  int i;
  struct D d;
  int j;
};
```

A. 0
B. 4
C. 8
D. 12
E. something else

- Correct answer: D

- Struct size and alignment
  - Alignment rules apply inside of a struct
  - Each instance variables will be aligned within the struct according to its type size
  - structs are usually aligned according to their largest instance variable type
  - structs can mix data types (**padding** may be needed to align the members)

```
// needs 8 bytes
struct X {
    char a; // e.g. 0x1000
    char b; //      0x1001
    int i;  //      0x1004
}

// needs 12 bytes
struct Y {
    char a; // e.g. 0x1000
    int i;  //      0x1004
    char b; //      0x1008
}
```

- Multiple of the largest element

- What is the offset (in bytes) of member `j` in `struct X` below? *Assume 8-byte addresses.*

```
struct D {
  int e;
  int f;
};
```

```
struct X {
  int i;
  struct D* d;
  int j;
};
```

A. 0
B. 8
C. 12
D. 16
E. something else

- Correct answer: D

Feb. 2nd - Unit 1c: Cont.d

- Arrays and structs
  - Array in struct

```
struct S {
    int i;
    int j[10];
}
```

  - Array of structs

```
struct S a[10];
```

  - Array of pointers to structs

```
struct S * b[10];
```

- What is the offset (in bytes) of member `j` in `struct X` below?

```
struct D {
  int e;
  int f;
};
```

```
struct X {
  int i;
  struct D d[10];
  int j;
};
```

- Correct Answer: 84
- Explanation: Assume 4-byte integer, and 4-byte address; d[10] can be shifted to eliminate padding, because each data in D is still 4-byte. Thus, it is $4 \times 1 + 8 \times 10 = 84$ bytes.

- What is the offset (in bytes) of member `j` in `struct X` below?

```
struct Y {
  char c;
  short s;
};
```

```
struct X {
  int i;
  struct Y y[10];
  int j;
};
```

    - Correct answer: 44
    - Explanation: char (1 byte), short (2 bytes), int (4 bytes); each Y will occupy 4 bytes with padding after short. Thus, the offset is $4 \times 1 + 10 \times (1 + 1 + 2) = 44$

- Given

```
struct A {
  char a;
  int* b;
  int* p;
};
```

```
struct X {
  char c;
  struct A a;
  int j;
};
```

```
struct X s; // a global variable
```

- How many memory reads to load `s.a.b[2]` into `r1`?

A. 1
B. 2
C. 3
D. 4
E. Not enough information

    - Correct answer: B
    - Explanation:

```
ld $s, r0          # r0 <- s
ld 8(r0), r2       # r2 <- m[r0 + 8]
ld 8(r2), r1       # r1 <- m[r2 + 8]
```

- Dynamic allocation in C and Java
    - Programs can allocate memory dynamically
        - Reserves a range of memory
    - In Java, instances of classes are allocated by **new** keyword
    - In C, byte ranges are allocated by call to **malloc** procedure
        - Bytes can be used for *any type* that can fit in them
    - C
        - Memory allocation

```
void * malloc(unsigned long n)
```

            - **n** is the number of bytes to allocate
            - return type is void *: just an address
                - A pointer to anything (no specific type)

        - Can be cast to/from any other pointer type
        - Cannot be dereferenced directly
    - **sizeof** to determine the number of bytes to allocate
        - **sizeof(x)** statically computes the number of bytes in a type or variable
        - **sizeof(ptr)** gives the size of a *pointer*, not what it *points* to
        - ```
          struct Foo * f = malloc(sizeof(struct Foo));
          struct Foo * f = malloc(sizeof(*f))
          ```
    - The system marks the chunk of memory as allocated: the allocation belongs to the program for the lifetime; or until explicitly deallocated
- Deallocation
    - In Java
        - Automatic garbage collection: requires keeping track of every reference to an object
    - In C
        - Dynamic memory must be deallocated explicitly by calling **free** (call on an address which was returned by malloc previously)
        - **free** deallocates memory immediately; does not check to see if it is still in use
    - **Memory Leak**
        - Dynamically allocated data is not deallocated
- Memory heap
    - The **heap** is a large section of memory from which **malloc** allocates objects
    - In Java, all objects are stored on the heap
- Issues with explicit deallocation
    - **free(x)** does not do
        - change the value of **x**, the pointer still points to the freed object
        - other variables may still point there too
        - the binary data stored at address **x** is not erased
          ```

```
struct MBuf * create() {
    struct MBuf * mBuf =
malloc(sizeof(*mBuf));
    //
    return mBuf;
}

void destroy(struct MBuf * buf) {
    //
    free(buf);
}
```

- What if access buf after destroying?

  Dangling pointer
- What if call destroy twice on `buf`?

  Double-free
- What if forget to call destroy?

  Memory leak

Feb. 5th - Unit 1c: Cont.d

- `malloc` and `free`
  - Organization of memory
    - statically allocated: code, static data
    - the rest is unused or dynamically allocated
  - `malloc/free`
    - implemented in the library `<stdlib.h>`
    - manage a large chunk of memory called the heap
    - keep track of what's allocated or free
  - `malloc`
    - find a free chunk of memory of requested size
    - mark it as allocated
    - return a pointer to it (keeping track of the size of allocation) - address of the first byte
  - `free`
    - use pointer to determine allocated size
    - mark referenced memory as free
    - can only be called on an address associated with an allocation
- Dangling pointers

- A pointer to an object that has been freed / could point to unallocated memory / to another object (a later `malloc` re-used the address)
    - Often caused by use-after-free
    - Can happen with incorrect pointer manipulation
    - ```
      struct D {
          int e;
          int f;
      };

      struct D * d;       // global

      void foo() {
          d = malloc(sizeof(struct D));
      }

      // execution
      void someFunction() {
          foo();
          d -> e = 5;
          d -> f = 12;
          // some other stuff
          free(d);
          // some other stuff
          short * s = malloc(8 * sizeof(short));
      // s happen to be

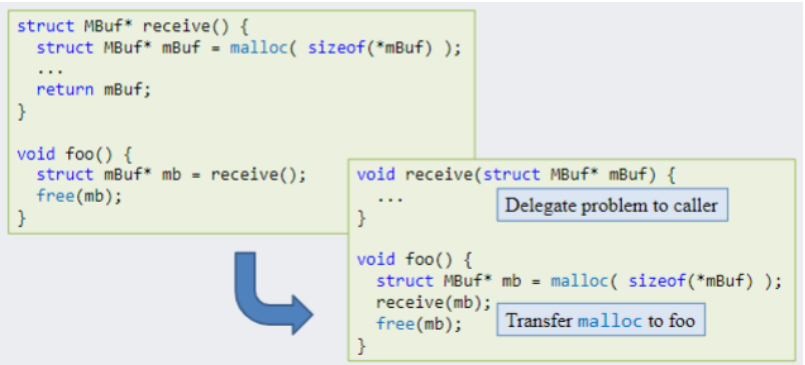
      // address in d
          s[0] = -13;
          s[1] = 2;
          s[2] = 5;

          printf("%d\n", d -> e); // grabs s[0] and
      s[1] together in binary
                                  // as e
      }
      ```
    - Why a problem?
        - program thinks it is writing to object of type X, but it isn't actually, it may be writing to an object of type Y
- Memory leak
    - A dynamic memory object with no pointers pointing to it
        - Usually happens if a program doesn't free object properly
        - May also happen if a pointer to a valid object is changed to another value
      ```

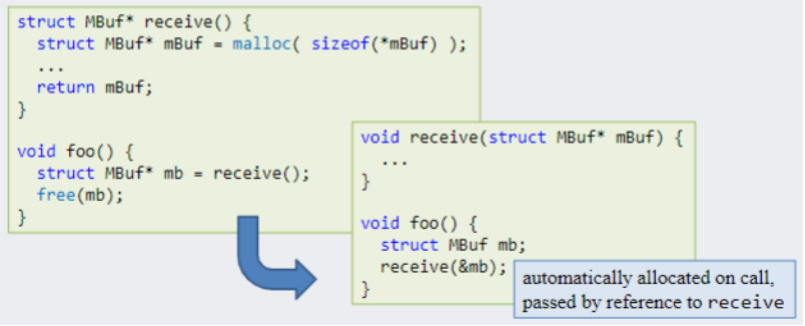
```
//
d = malloc(sizeof(struct D));
// ...
d = malloc(sizeof(struct D));
```

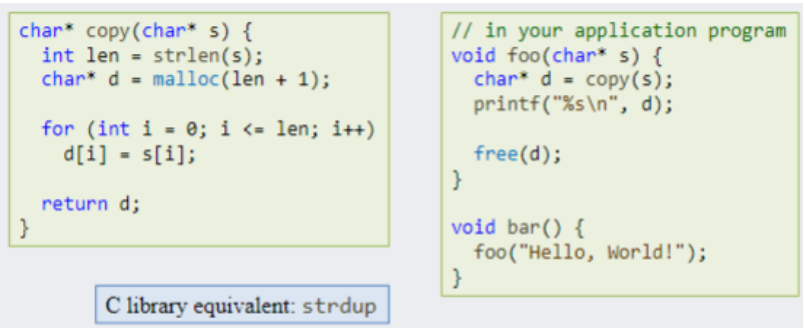
- Why problem?
  - If contains useful information, it can no longer be accessed
  - If object is large, or many memory leaks happen at the same time, the program uses up too much memory

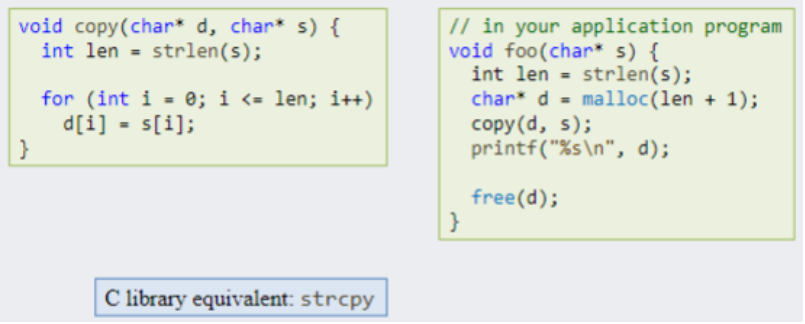
- Avoiding memory problems in C
  - Avoid the problem cases, **if possible**
    - restrict dynamic allocation / free to single procedure
    - don't write procedures that return pointers
    - use local variables instead - automatically allocated on call and freed on return
  - Engineer for memory management
    - Define rules for which procedure is responsible for deallocation
    - implement explicit **reference counting** if multiple potential deallocators
    - define rules for which pointers can be stored in data structures
    - use coding conventions and documentation to ensure rules are followed
  - Co-locate allocation and deallocation
    - allocate the object in its caller, and pass pointer to it - to the procedure (callee)
    - good if caller itself does both `malloc`/`free` itself



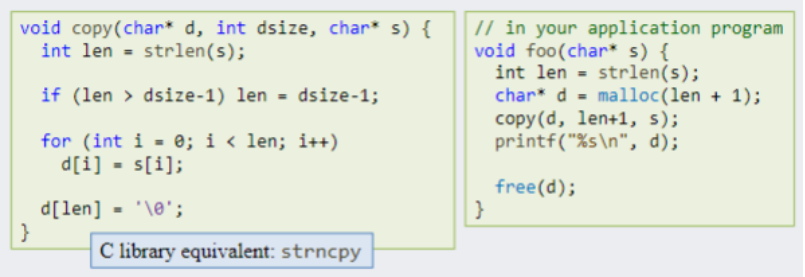
    - as long as `receive()` does not pass along the reference
  - Use local variables instead
    - If a procedure does both `malloc`/`free`
      - Use a "static" local variable instead of `malloc`, and don't give another procedure a pointer to the local variable
    - Local variables are allocated on call and deallocated on return

- 

- C strings
  - A string `"like this"` in C
    - It is an array of `char`s
    - Has its end indicated by the first null `'\0'` in the array
    - Every string has a maximum length (capacity of array -1), and a printable length (determined by the position of the first null character)
    - Each character resides at some address (`char *`)
  - Standard C library `<string.h>` has operations on strings
    - `strlen(s)` returns the printable length of a string, starting at address `s`
  - How can we create a new string that copies the existing string?
    - 

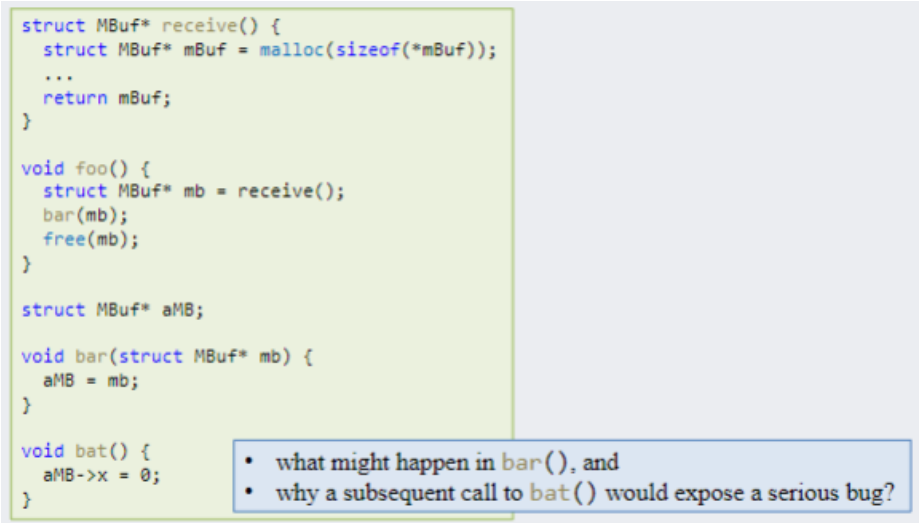
    - 

    -
      - Pro: `malloc` and `free` are in the same procedure.
      - Con: Need to make sure we `malloc` sufficient space, which can potentially lead to out-of-bound `d` is too short



- `len` will be set to minimum between two strings
- `strncpy` copies minimum of $\{n, \texttt{len(source\_string)}\}$

- Detecting memory problems
  - Valgrind
    - performs dynamic analysis of the runtime of a program
    - Monitors dynamic allocation and deallocation
    - Memory leaks, use after free (dangling pointers) etc
  - "... if possible..."
    - A reference may needs to be passed among multiple modules
    - Correctly `free` an object
      - `free` should only happen if the object is still in use
      - need to keep track of all its users
        - start and stop



Feb. 7th - Unit 1c: Cont.d

- Detecting memory problems

```
void foo() {
    struct MBuf * mb = receive();
    bar(mb);
    free(mb);       // How do we know we can do
this?
                    // Need information about
whether the pointer is used
}

void bar(struct MBuf * mb) {
    if (smth_complicated_private) {
        aMB = mb;       // may or may not occur
    }
}
```

- Reference counting
  - Initialize the reference count to 1 (the caller has a reference)
  - any procedure that stores a reference (starts using the object) increments the count
  - any procedure that discards a reference (stops using the object) decrements the count
  - never call **`free`** directly, only **`free`**d when the reference count goes to 0

```
struct MBuf* receive() {
  struct MBuf* mBuf = rc_malloc(sizeof(*mBuf));
  ...
  return mBuf;
}

struct MBuf* aMB = 0;

void foo() {
  struct MBuf* mb = receive();
  bar(mb);
  rc_free_ref(mb);
}

void bar(struct MBuf* mb) {
  // free existing reference in aMB (why?)
  if (aMB != 0)
    rc_free_ref(aMB);
  rc_keep_ref(mb);
  aMB = mb;
}
```

  - replace **`malloc`** with a reference-count **`rc_malloc`**; replace **`free`** with a reference-count **`rc_free_ref`**
  - Reference counter can be part of a **`struct`**
    - updates to reference counter are done through special methods
    - no longer explicitly call **`malloc`**/**`free`** directly when creating the struct instances

```
struct buffer* rc_malloc() {
  struct buffer* buf = malloc(sizeof *buf);
  buf->ref_count = 1;
  return buf;
}

void rc_keep_ref(struct buffer* buf) {
  buf->ref_count++;
}

void rc_free_ref(struct buffer* buf) {
  buf->ref_count--;
  if (buf->ref_count == 0)
    free(buf);
}

struct buffer {
  ...
  // the actual data attributes
  ...
  int ref_count;
}
```

        - Problem: every data type requires their own struct definition, and their reference counting procedures.
    - Implementing reference counting
        - `refcount.h`

```
void * rc_malloc(int nbytes);   //
alternative malloc with room for
                                // ref_count
void rc_keep_ref(void * p);     // call when
reference added
void rc_free_ref(void * p);     // call when
reference removed
```

        - `refcount.c`

```
void * rc_malloc(int nbytes) {
    // Assume the largest atomic data type is
8 bytes
    int * ref_count = malloc(nbytes + 8);
    *ref_count = 1;
    return ((void *) ref_count) + 8;
}

void rc_keep_ref(void * p) {
    int * ref_count = p - 8;
    (*ref_count) += 1;
}

void rc_free_ref(void * p) {
    int * ref_count = p - 8;
    (*ref_count) -= 1;
    if (*ref_count == 0) {
        free(ref_count);
    }
}
```

- Reference counting and procedure calls
    - When reference is a parameter
        - the caller has a reference and maintains it for the duration of the call
        - the callee need not perform a `keep_ref` unless
            - callee saves the pointer someplace that will outlast the call (global variable)
            - callee returns a pointer to the caller
    - When reference is a return value
        - the callee must have a reference to the value

- it passes that reference to the caller
    - the callee implicitly gives up its reference as part of the return
    - the reference is transferred to the caller
- the caller must call `free_ref` when it no longer stores the reference

- When reference is stored in a local variable
    - that variable goes away implicitly when the procedure returns
    - the procedure must call `free_ref` before it returns

Feb. 9th - Unit 1c: Cont.d + Unit 1d: Static Control Flow

- What if there is more to the quiz structure?

```
struct quiz {
  char* name;
  int* grades;
  int num_grades;
};
```

```
struct quiz* quiz_create(char* name) {
  struct quiz* quiz = rc_malloc(sizeof(*quiz));
  int name_size = strlen(name) + 1;
  quiz->name = rc_malloc(name_size);
  strlcpy(quiz->name, name, name_size);
  quiz->grades = NULL;
  return quiz;
}

void addGrades(struct quiz* quiz, int* grades, int numGrades) {
  quiz->grades = rc_malloc(sizeof(*quiz->grades) * numGrades);
  for (int i = 0; i < numGrades; i++)
    quiz->grades[i] = grades[i];
}

void quiz_delete(struct quiz* quiz) {
  rc_free_ref(quiz->name);
  rc_free_ref(quiz);
}
```

How should we fix quiz_delete?

-
    -
```
void quiz_delete(struct quiz * quiz) {
    rc_free_ref(quiz -> name);
    if (quiz -> grades != NULL) {
        rc_free_ref(quiz -> grades);
    }
    rc_free_ref(quiz);
}
```

- Garbage Collection (Java)
    - In Java, objects are deallocated implicitly
        - Program never calls `free`
        - the runtime system tracks every object reference
        - garbage collector runs periodically to deallocate unreachable objects
    - Advantage compared to explicit `free` in C
        - No dangling pointers

        - (almost) No memory leaks
        - Reference cycles are not a problem
            - Object a has pointer to b; Object b has pointer to a
            - Cause memory leak when using reference counting alone
- Discussion
    - Advantages of explicit `free`
        - fast, efficient, forces us to be careful about memory management
    - Advantages of reference counting
        - Reduce coupling between modules
    - Advantages of garbage collection
        - convenient, automatic, works well for most cases
    - Should we ignore deallocation in Java
        - No, HashMap
- Memory management in Java
    - Memory "leaks" will occur
        - Garbage collector fails to reclaim **unneeded** objects
        - Still a significant problem for long-running programs where garbage accumulates
    - Why would an object not be reclaimed
        - Garbage collector only reclaims **unreachable** objects
        - Unneeded / unused objects that keep references are not reclaimed
        - Collections and maps may maintain old unneeded objects.

Static Control Flow

- Control flow
    - Definition: sequence of instruction executions performed by a program
    - Every program execution can be described by such a linear sequence
- Loops

- ```
  public class Foo {
      static int s = 0;
      static int i;
      static int a[] = new int[] {2, 4, 6, 8, 10, 12,
  14, 16, 18, 20};

      static void foo() {
          for (int i = 0; i < 10; i += 1) {
              s += a[i];
          }
      }
  }
  ```

- ```
  int s = 0;
  int i;
  int a[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

  void foo() {
      for (i = 0; i < sizeof(a); i++) {
          s += a[i];
      }
  }
  ```

- ```
  int s = 0;
  int * p;
  int a[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}

  void foo() {
      p = a;
      while (p < a + 10) {
          s += *p++
      }
  }
  ```

    - `s += *a++` is not valid because `a` is a static array (base address), cannot change
    - even if we could, the loop condition needs to be updated

- Implementing loops in SM213
  ```

- ```
  int s = 0;
  int i;
  int a[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

  void foo() {
      for (i = 0; i < sizeof(a); i++) {
          s += a[i];
      }
  }
  ```

    - *This* loop can be unrolled.
- Dissecting loops (using `goto`)
    - ```
      for (i = 0; i < n; i += 1) {
          s += a[i];
      }
      ```
    - ```
          i = 0
      loop:       goto end_loop if not (i < n)
                  s += a[i]
                  i++
                  goto loop
      end_loop:   // whatever code comes after
      ```
- Control flow in the machine
    - Program counter: contains address of **next instruction** to execute
    - For sequential instructions, PC is updated in the fetch stage
        - **incremented** by 2 or 6 bytes depending on instruction size
    - To break sequential execution, we need to change PC from what it would contain (`goto`)
- Control flow: ISA extensions
    - New instructions: jumps and conditional jumps
        - `pc <- <address>`
        - `pc <- <address> if <condition>`
    - Options for evaluating
        - **Unconditional**
        - Conditional: common in RISC
            - `goto <address> if <register> <condition> 0`
      ```

                - Conditional: based on result of last executed ALU instruction
            - `j address`: jump to `address`
            - `br address`: branch to `address`
            - `beq r0, address`: branch to `address` if register equal to 0
            - `bgt r0, address`: branch to `address` if register strictly greater than 0
        - Jump instruction size
            - Memory addresses are big (32 bits in SM 213)
            - And control flow instructions are common
            - Observations
                - jumps *usually* move a short distance (forward or backward)
        - PC-relative addressing
            - instead of specifying the destination address completely, specify the **offset** from the current location in code
            - use the current value of PC as **base address**
            - Offset must be a signed number
            - Assembly still specifies the actual address/label
            - jumps that use PC-relative addressing are called **branches**
            - Example
                - Suppose we want to do: `0x1000: goto 1008`
                - Option 1: absolute addressing (`X--- 00001008`), 6 bytes
                - Option 2: PC-relative addressing (branch), PC updated to `0x1002` already
                    - `Y-06`, the offset is 6 from 1002 to 1008
                    - The address usually offsets by 2 bytes or 6 bytes (even), we can expand our range by encoding offsets /2. The `Y-06` is in fact `Y-03`

Feb. 12th - Unit 1d: Cont.d

- ISA for static control flow
    - At least one absolute jump
    - At least one PC-relative jump
        - specify relative distance using real distance / 2
        - PC-relative offset is signed
    - Some conditional jumps

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`

  - Convert the `bgt` instruction to machine code:

    ```
    .pos 0x10
        bgt r0, L1
    .pos 0x20
    L1: halt
    ```

    A. `0xa0 0x10`
    B. `0xa0 0x08`
    C. `0xa0 0x0e`
    D. `0xa0 0x07`
    E. `0xa0 0x20`

Name	Semantics	Assembly	Machine
branch if greater	`pc ← a (or pc + p*2) if r[c] > 0`	`bgt rc, a`	`acpp`

- - Correct answer: D
  - Explanation: `bgt r0, L1` is a 2-byte instruction, so after fetching, PC has value 0x12. 0x20 is 0x0e away from 0x12, so half the real distance is 0x07. This means `pp` in machine code corresponds to `0x07`

- - Convert the `br` instruction to machine code.
    - note: each instruction here is 2 bytes

    ```
    loop: mov r0, r5
          add r4, r5
          beq r5, end_loop
          inc r0
          br  loop
    ```

    A. `0x80 0xf5`
    B. `0x80 0xf8`
    C. `0x80 0xfc`
    D. `0x80 0xfb`
    E. `0x80 0xf6`

Name	Semantics	Assembly	Machine
branch	`pc ← a (or pc + p*2)`	`br a`	`8-pp`

- - Correct answer: D
  - Explanation: After we fetch `br loop`, PC holds the address right after the instruction, which is 10 bytes away from `loop`. Half the distance is 5, but since we are moving backwards, it should be -5, which corresponds to `0xfb`.

- - What does the following instruction do?

    ```
    0xa0 0x00
    ```

    A. infinite loop
    B. sets PC to zero
    C. sets PC to beginning of program
    D. nothing
    E. something else

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`

- - Correct answer: D

- Explanation: PC value is updated after fetch, but the offset is `0x00`, so the machine just continues running to the next instruction. This is equivalent to no operation.

- What is the value of r0 after this code executes?

```
      ld $1, r0
      ld $4, r1
loop: beq r1, end
      shl $1, r0
      dec r1
      br loop
end:  halt
```

A. It never terminates

B. 4

C. 8

D. 16

E. 32

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

- Correct answer: D

```
int a = 1;
int i = 4;
while (i != 0) {
    a *= 2;
    i -= 1;
}
```

- Implementing `for` loops

```
for (i = 0; i < 10; i += 1) {
    s += a[i];
}
```

- General form:

```
for (<init>; <continue-condition>; <step>) {
    <statement-block>
}
```

- Each of init, continue, and step can be compound expression

- pseudo-code template

```
          <init>
loop:     goto end_loop if not <continue-condition>
          <statement-block>
          <step>
          goto loop
end_loop:
```

- By the template:

```
for (i=0; i<10; i++)
  s += a[i];
```

```
          i = 0
loop:     goto end_loop if not (i < 10)
          s += a[i]
          i++
          goto loop
end_loop:
```

??

- ISA does not support comparison to 10, but we can compare to 0, and there is no need to store `i` (or `s`) in memory for each loop iteration

- ' == 0

Assembly: assume that all variables are global

```
          ld  $a, r1        # r1 = a = &a[0]
          ld  $s, r2        # r2 = &s
          ld  (r2), r2      # r2 = s = s'
          ld  $0x0, r0      # r0 = i' = 0          init
          ld  $-10, r4      # r4 = -10
loop:     mov r0, r5        # r5 = t' = i'
          add r4, r5        # r5 = i' - 10         cont/exit
          beq r5, end_loop  # if i' = 10, goto +8
          ld  (r1, r0, 4), r3 # r3 = a'[i']
          add r3, r2        # s' += a'[i']
          inc r0            # i'++                 step
          br loop           # goto -14
end_loop: ld  $s, r1        # r1 = &s
          st  r2, (r1)      # s = s'
          st  r0, 4(r1)     # i = i'
```

- Implementing conditionals
  - General form:

```
if <condition> {
    <then-statements>
} else {
    <else-statements>
}
```

-

```
        c' = not <condition>
        goto then if (c' == 0)
else:   <else-statements>
        goto end_if
then:   <then-statements>
end_if:
```

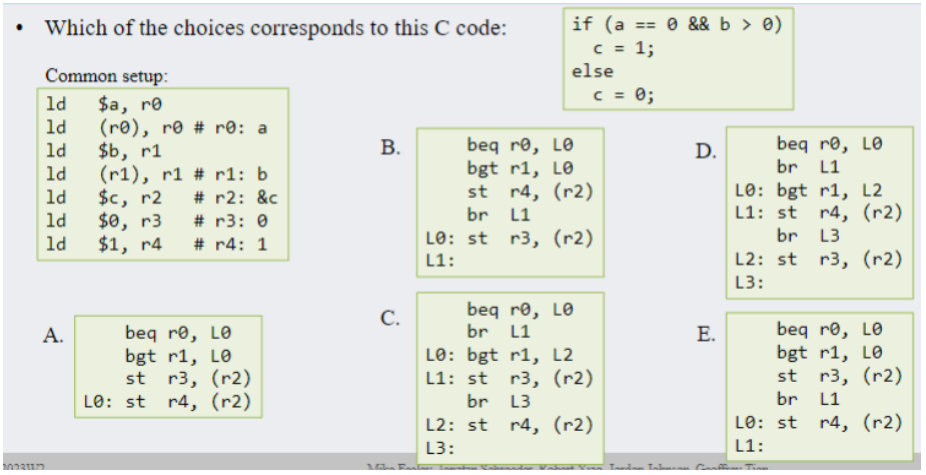
or

```
        c' = <condition>
        goto then if (c' > 0)
else:   <else-statements>
        goto end_if
then:   <then-statements>
end_if:
```

-

Assembly:

```
          ld  $a, r0        # r0 = &a
          ld  (r0), r0      # r0 = a
          ld  $b, r1        # r1 = &b
          ld  (r1), r1      # r1 = b
          mov r1, r2        # r2 = b
          not r2            # c' = !b
          inc r2            # c' = -b
          add r0, r2        # c' = a - b
          bgt r2, then      # if (a>b) goto then
else:     mov r1, r3        # max' = b
          br end_if         # goto end_if
then:     mov r0, r3        # max' = a
end_if:   ld  $max, r0      # r0 = &max
          st  r3, (r0)      # max = max'
```



- Correct answer: C

Feb. 14th - Unit 1d: Cont.d, Unit 1e: Procedure class and the Stack

- What does this code do?

```
.pos 0x1000
    ld  $0, r0
    ld  $0, r1
    ld  $1, r2
    ld  $j, r3
    ld  (r3), r3
    ld  $a, r4
L0: beq r3, L9
    ld  (r4, r0, 4), r5
    and r2, r5
    beq r5, L1
    inc r1
L1: inc r0
    dec r3
    br  L0
L9: ld  $o, r0
    st  r1, (r0)
    halt
```

```
.pos 0x2000
j:  .long 2
a:  .long 1
    .long 2
o:  .long 0
```

- Step 1: comment each line
- Step 2: Refine the comments to look more like C

```
int i = 0;
int j = 2;
int a[2] = {1, 2};
int o;
```

- Step 3: Look for basic blocks by examining branches
- Step 4: Associate control structure with C

```
for (j' = j, i' = 0; j' != 0; j' -= 1, i' += 1) {
    if (a'[i'] & 1) {
        o' += 1;
    }
}
```

- Step 5: Deal with what's left

```
for (i = 0; i != j; i += 1) {
    if (a[i] & 1) {
        o += 1;
    }
}
```

- Static procedure calls
  - Java: method; C: procedure: subroutine with a name, arguments, and local scope
  - Control flow during procedure calls

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

void ping() {
    // smth
}
```

    - Caller ↔ Callee
    - Caller
      - goto ping: `j ping`
      - continue executing from callee's return
    - Callee
      - from caller: then do whatever `ping()` does
      - goto `foo()` just after call to `ping()`, but return to where?
    - The jump from `ping()` is dynamic
- Implementing procedure return
  - **Return address**
    - 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 (a procedure may be called from different locations within a program)
  - Questions

        - How does the procedure know the return address?
        - How does it jump to a dynamic address?
        - How are recursive calls handled?
    - Only the caller can provide the address
        - the caller must save it **before** it makes the call
        - By SM213 convention, caller will save the return address in `r[6]`
        - We need a new instruction to read the PC contents: `gpc`
    - Jumping back to the return address
        - Callee assumes caller saved address in `r[6]`
        - We need a new instruction to jump to dynamic addresses stored in a register
- ISA instructions for static control flow
    - New requirements
        - Read value of the PC
        - jump to dynamically determined target address
    - New 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`
        - do `gpc` before we do the jump

```
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)
```

        - `j ping` is 6 bytes, so the instruction after `ping()` must be offset by 6 for the PC

- What is wrong with this code?

```
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

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

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

- - Since `r6` is updated in `ping`, after running `pong`, the program will just continue to execute `ping: j (r6)`

Feb. 16th - Unit 1e: Cont.d

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

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, one of these can be allocated statically at all

- - Correct answer: C
- Examples (Local variables and recursion)
  - ```
    void b(int a0) {
        int l0 = a0;
        if (a0 > 0) {
            b(a0 - 1);
        }
        printf("%d\n", l0);
    }
    ```
    ```

        - The expected output is: 0, 1, 2
        - There are 3 different `l0`s
        - The actual output with static allocation is going to be: 0, 0, 0
    - ```
      void b(int a0) {
          int l0 = a0;
          c(a0);
      }
      ```
        - What if there is no apparent recursion, in this case, what if `c()` calls `b()`
    - Conclusion: we should allocate arguments and local variables **dynamically**.
- Life of a local (argument/variable)
    - Scope
        - accessible **ONLY** within declaring procedure
        - each execution of a procedure has its own private copy
    - Lifetime
        - allocated when procedure starts
        - de-allocated when procedure returns
    - 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
    - Should we allocate activation frames from the heap?
        - call `malloc()` to create frame on procedure call and call `free()` on procedure return?
        - we do not allocate the frames in the heap
            - 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
                - start by reserving a **big** chunk of memory for all the frames
                - assuming one knows the address of this chunk
      ```

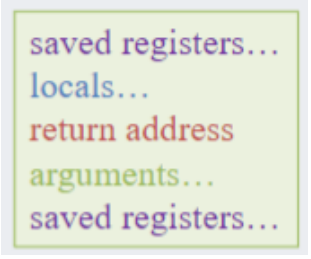
        - how to allocate and free frames? (add/substract the pointer by 1)
        - explicit allocation in heap requires more complicated and thus more costly allocation and deallocation to avoid fragmentation
    - Data-structure alike: Last In First Out → Stack
    - Restriction we place on lifetime of local variables and arguments: within the activation of their own procedure call.
- The runtime stack
    - Stack of activation frames
        - stored in memory
        - grows **UPWARDS** from bottom
    - Stack pointer (SP)
        - general purpose register
        - Use `r5`, stores the base address of current frame
    - Top and bottom
        - current frame is 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
    - Allocation and de-allocation
        - Allocation is done by decrementing SP
        - De-allocation is done by incrementing SP
- Allocating activation frames
    -
```
void foo(int a) {
  int l;
  r5 = malloc(foo_frame_size); // but not really a malloc

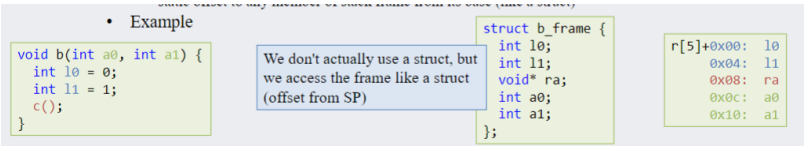

  r5->saved_return_address = r6; // somehow
  ...
  l = a;  // r5->l = r5->a;
  ...

  free(r5); // but not really a free

}
```

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

- Activation frame details
  - **Local variables** and **arguments**
  - **Return address** (ra)
    - previously we put this in `r6`, but does not work with consecutive/recursive function calls
    - We then 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
    - anything that is being overwritten needs to be restored
  - Stack frame layout
    - 

    - compiler decides
    - based on order convenient for stack creation
    - static offset to any member of stack frame from its base (like a `struct`)
    - 

  - Accessing a local variable or argument
    - access like a `struct`
    - base address is in `r5` (stack pointer)
    - offset is known statically
  ```

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

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

  - this becomes

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

- What would be the SM213 assembly code for the following?

```
int l1 = a1;
```

```
r[5]+0x00:  l0
      0x04:  l1
      0x08:  ra
      0x0c:  a0
      0x10:  a1
```

A.
```
ld 16(r5), r0
st r0, 4(r5)
```

B.
```
ld 10(r5), r0
st r0, 0(r5)
```

C.
```
ld 16(r5), r0
st r0, 0(r5)
```

D.
```
ld 10(r5), r0
st r0, 4(r5)
```

E. None of the above

-
  - Correct answer: A
  - Explanation: `a1` is offset 16, `l1` is offset 4

What is the value of `l` (in `foo` when it is active)? Assume no default initialization.

```
int g;

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

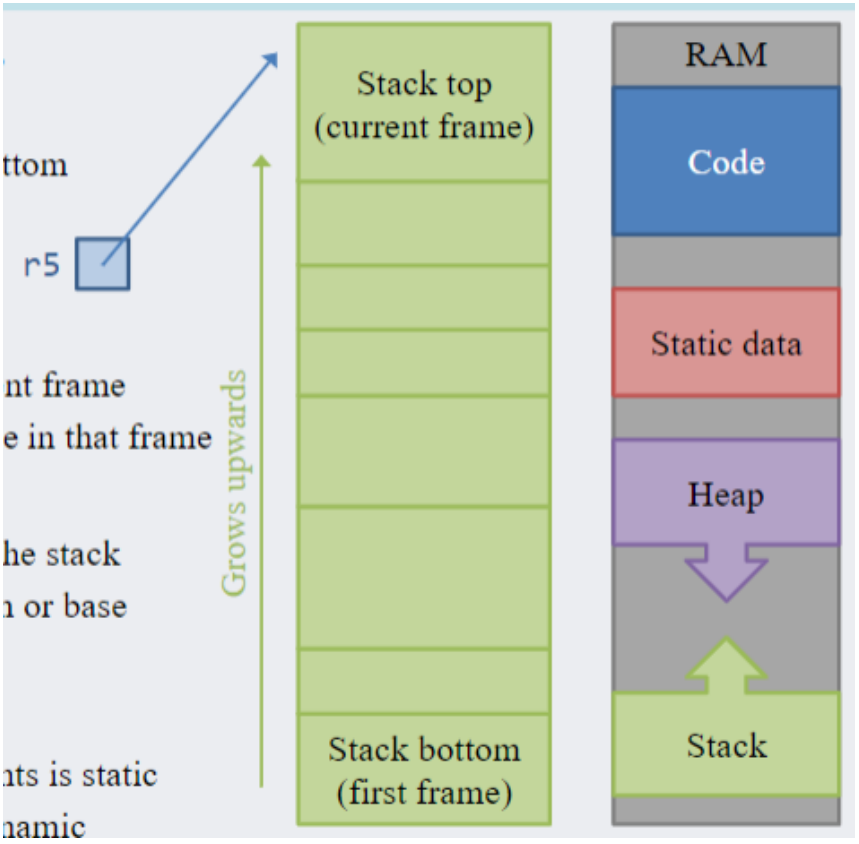
```
void goo() {
  int l = 3;
}
```

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

A. 0
B. undefined
C. it has no value
D. 3
E. I don't know

-
  - Correct answer: D
  - The stack pointer increments when `goo` returns, so the binary data was left there with no allocation; then when `foo` is just called, and `l` is declared, then the stack pointer decrements, so `l` will take the leftover value from `goo()`

Feb. 26th - Unit 1e: Cont.d

- The runtime stack
    - Stack of activation frames
        - stored in memory
        - grows upwards from bottom
    - Stack pointer (SP)
        - general purpose register
        - we use `r5`
        - stores base address of current frame
    - Top and Bottom
        - current frame is the top of stack
        - first activation is bottom or base
    - Static and Dynamic
        - size of frame is static(ish)
        - offset to locals and arguments are static
        - value of stack pointer is dynamic
    - 

- What is wrong with these functions?
    - ```
      int * foo() {
          int l;
          return &l;
      }
      ```
      ```

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

void ping() {
    //
    int * p = foo();
    // when foo returns, p will refer to a de-
allocated space
    // this leads to a dangling pointer
}

void pong() {
    //
    bar();
    // after bar returns, we have an allocation on
heap where no
    // variables in the stack can access, this leads
to a
    // memory leak
}
```

- "Dynamic" arrays in the stack
    - Latest version of C allows local arrays to have dynamic size

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

    - Activation frame layouts/offsets are statically determined, so the question is what is the offset for `b`, when this dynamically-sized array can occur?
    - What code does the compiler generate for the last statement?
        - Compiler chooses a size for the array, large enough to handle most "reasonable" values at `n`.

          This can waste a lot of space if actual `n` is small, but can also possibly overflow when the actual `n` is even larger
        - Compiler convert local array code into array pointer with `malloc` dynamic allocation, inject `free` at end of procedure
- Allocating and deallocating stack frames
    - Allocation of stack is relatively simple
        - Decrement the stack pointer to allocate

        - Increment the stack pointer to free
    - Whose responsibility is it to allocate in the stack
        - Caller doesn't (need to) know the size of callee's stack
        - However, caller needs to prepare (copy) the arguments
    - Division of labour
        - Caller allocates space for arguments
        - Callee allocates space for the rest
    - 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 just before return
            - part in caller just after return
        - possibly restores some saved register values
        - deallocates activation frame and restore stack pointer
            - add frame size to `r5`
- Stack management
    -
```
void foo() {
    b(0, 1);
}

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

    - 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 return
  - deallocate stack space of arguments
  - ```
    r[sp] += 8
    ```

-
```
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
                                                    1. caller prologue
     gpc  $6, r6        # set return address
     j    b             # b (0, 1)
                                                    2. call
     ld   $8, r0        # r0 = 8 = size of caller part of b's frame
     add  r0, r5        # deallocate caller part of b's frame
                                                    6. caller epilogue
     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
                                                    3. callee prologue
     ld   12(r5), r0    # r0 = a0
     st   r0, 0(r5)     # l0 = a0

     ld   16(r5), r0    # r0 = a1
     st   r0, 4(r5)     # l1 = a1
                                                    4. callee body
     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
                                                    5. callee epilogue
```

- Creating the stack
  - Every thread starts with a hidden procedure
    - its name is `start` (or smth like `crt0`)
  - The start procedure
```

- allocates memory for stack
- initializes the stack pointer
- calls `main()`

- For code from previous slide
  - the main procedure is `foo()`
  - we will 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
    ```

- What is the value of `r5` in `three()`?
  - (numbers in decimal to simplify math)
  - Write the letter corresponding to your choice

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

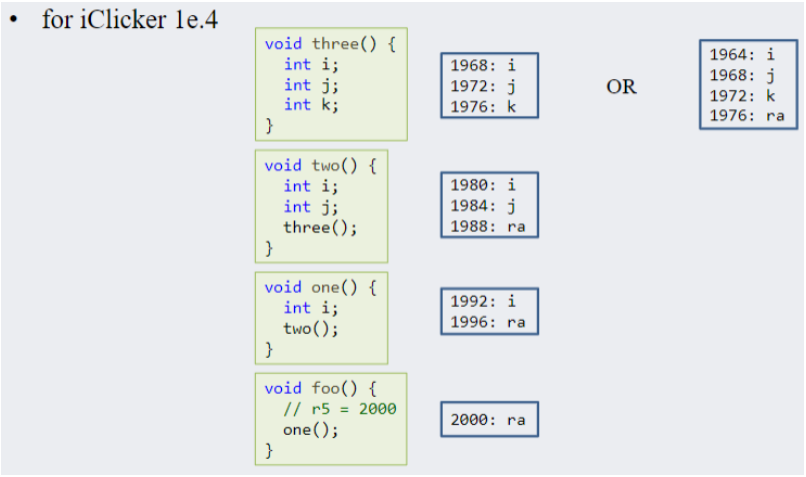
```
void three() {
  int i;
  int j;
  int k;
}

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

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

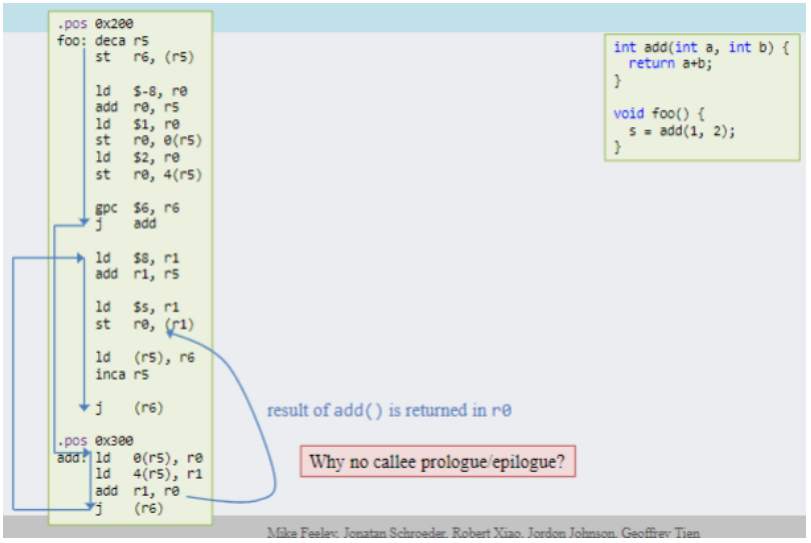
void foo() {
  // r5 = 2000
  one();
}
```

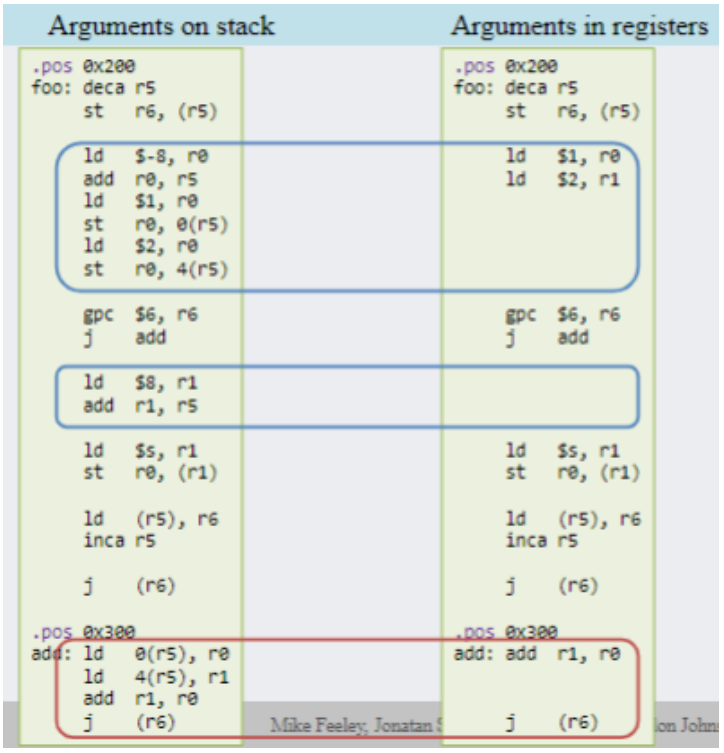
- Correct Answer: A or G
- Explanation:
  - for iClicker 1e.4


```

Feb. 28th - Unit 1e: Cont.d

- Return value, arguments, optimizations
    - `r0`: return value (or, allocate space in activation frame for return value, especially for those types that cannot fit in a register. done in caller prologue); `r5`: stack pointer; `r6`: return address
    - 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 are 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? Procedure does not need many local resources
            - WHEN is this a good idea?
            - This reduces memory access: fewer fetching instructions for allocation, placing arguments in activation frame, retrieving arguments in activation frame
    - Other optimizations
        - return address `r6` does not always need to be saved to the stack
            - procedure does not need to call functions
        - local variables are sometimes not needed or used
            - If computation can be entirely done by registers

- 

- 

- Recursion in SM 213
  - Consider

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

  - First, remove names and types for arguments (and local variables)

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

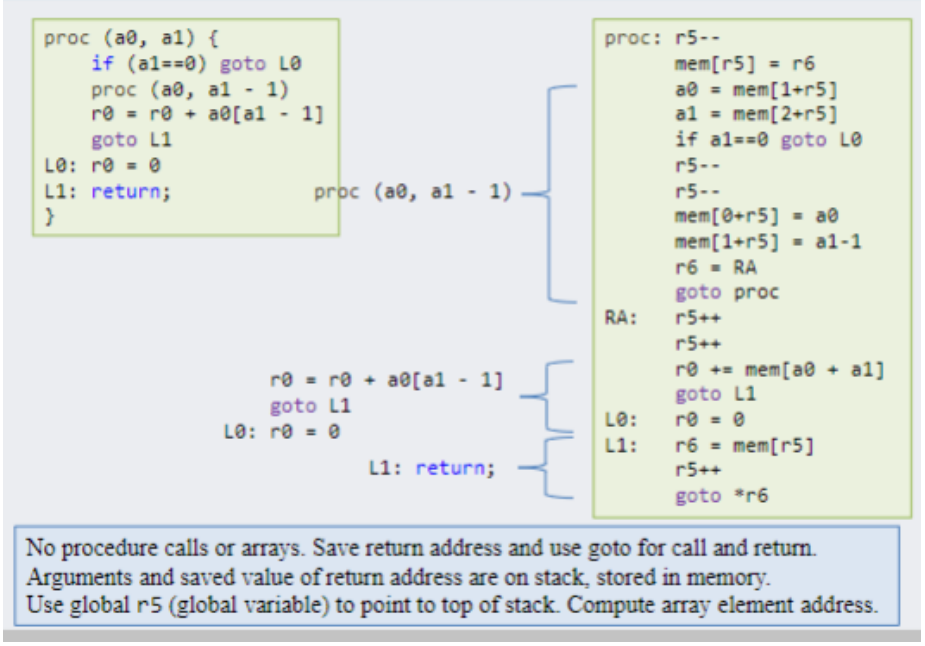
- Second, replace C-style conditional. Use comparison to 0; use `goto`; swap then and else ordering

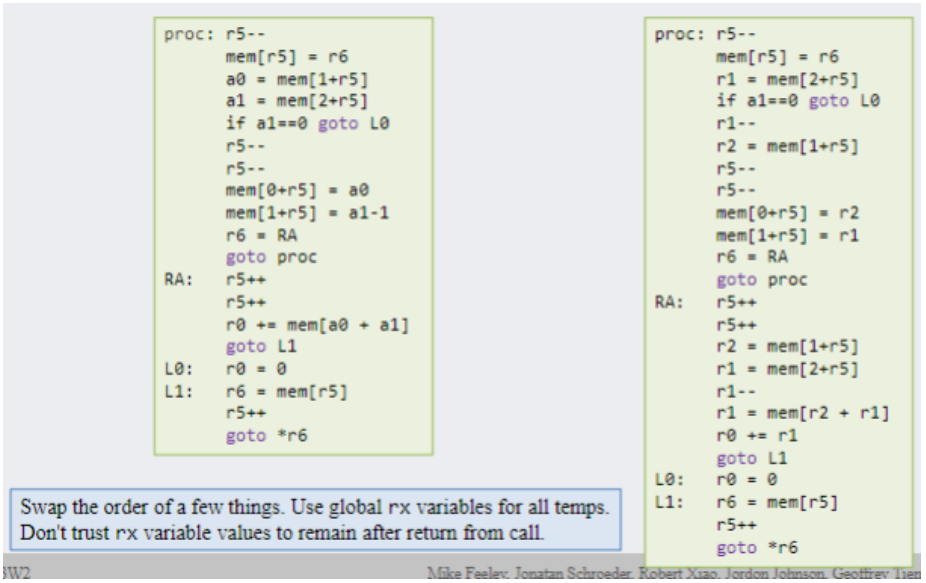
```
int proc(a0, a1) {
    if (a1 == 0)
        goto L0;
    return proc(a0, a1 - 1) + a0[a1 - 1]
    goto L1;
L0: return 0;
L1:
}
```

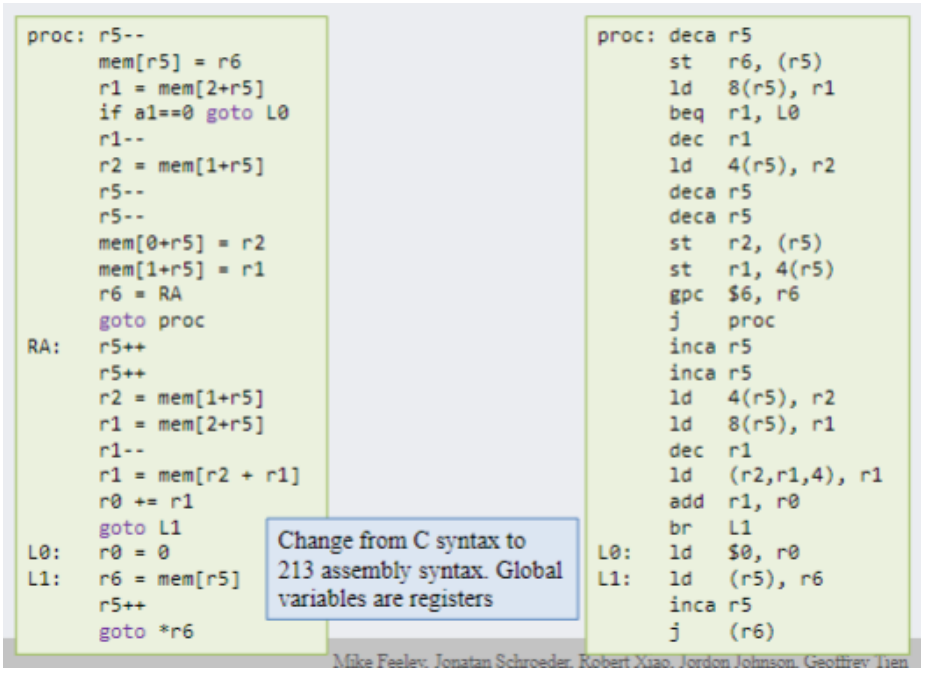
- Third, procedure return value is in `r0`

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

- Fourth,

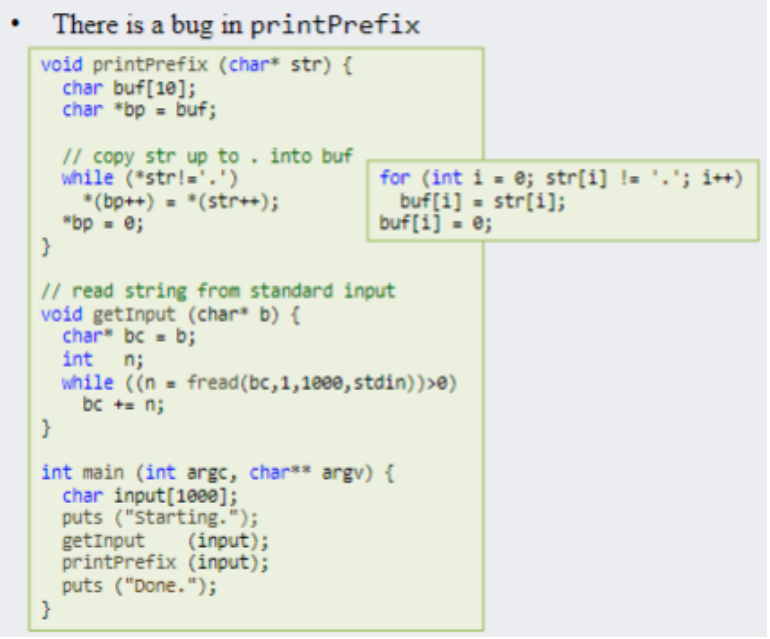


- 

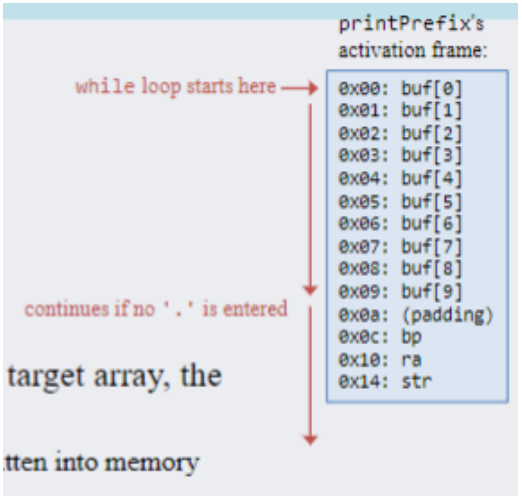
- 

- Summary: arguments and local variables
  - Stack is managed by code that the compiler generates
    - grows from bottom up
    - push by subtracting
    - caller prologue
    - callee prologue
    - callee epilogue
    - caller epilogue
  - accessing local variables and arguments
  - global variables
    - address known statically
  - reference variables
    - variable stores address of value
  - arrays
    - elements, named by index

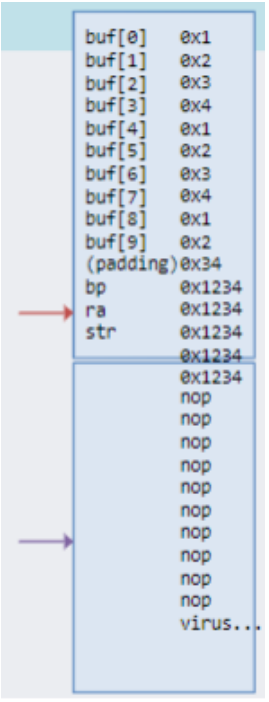
        - address of element is base + index * size of element
    - instance variables
        - offset to variable from start of object/struct known statically
        - address usually dynamic
    - locals and arguments
        - offset to variables from start of activation frame known statically
        - address of stack frame is dynamic
- Security vulnerabilities with activation frames
    - Buffer overflow



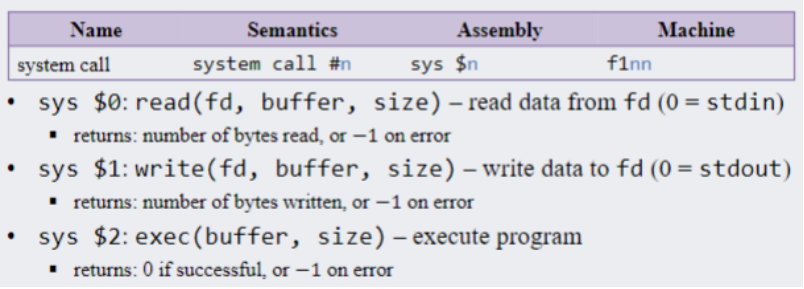
        - What happens if the string never ends with a '.'?
    - Buffer overflow principles
        - Program code is just binary values in memory
        - Program data are also just binary values in memory
        - This means we can include data that **can be interpreted as code**
            - Injected code can do **anything**
            - Most attackers want an interactive shell - inject "shellcode"
        - If the input (e.g. from the previous example) is longer than the target array, the loop will write into memory **beyond the end of the `buf` array**
            - If an attacker can provide the input, it can determine what values are written into memory beyond the end of **`buf`**

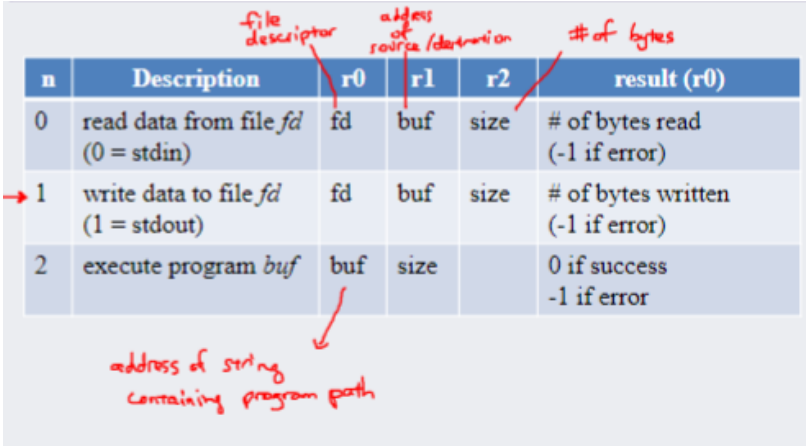
- 

Mar. 1st - Unit 1e: Cont.d & Unit 1f: Dynamic Control Flow

- Security Vulnerabilities
    - Buffer overflow
        - The attacker's goal: introduce code into the program from outside; trick program into running it (change the return address to change control flow)
        - Changing the return address: allows attacker to change control flow; if it points to data, then that data becomes the program.
    - Stack-smash (how to make the attack string)
        - 

        - Hard parts
            - Determining the location of return address in attack string
            - Determining address to change return address to
        - Making it easier

        - Approximate return address value (run the program with a 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/sled/ramp)
        - Finally include the code for the virus
    - Works if
        - return address guess is anywhere in the `nop` slide (in the graph example, `0x1234` needs to be in between `nop` and `virus`)
- System calls
    - CPU instruction that signals OS to do smth
        - typically smth that regular processes don't have permission to do
        - examples: read/write terminal, read/write file, execute programs
    - Similar to function calls
        - arguments are passed in registers `r0, r1, r2`
        - values are prepared ahead of time before calling
    - New ISA
        - Requirements:
            - Instruction encodes which system call to use
            - remaining arguments are passed in registers `r0, r1, r2`
        - 

        - 

```
.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
```

- - Write 12 bytes starting from address `str` to `stdoutput`
  - If we call `sys \$3`, the `buf` is the address of `str` to program path

- Protecting against buffer overflow attacks
  - What if the stack grew downwards?
    - Active frame at highest address
  - Modern protections
    - Non-executable stack
    - Canaries
    - Randomized stack addresses

Unit 1f: Dynamic Control Flow

- Review: static procedure calls
  - Consider

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

void bat() {
    bar();
}

void bar() {
    // does smth
}
```

- `bar()`'s return will be dynamic

- Suppose we have a Java class A with a procedure foo.
  - Suppose this class has subclasses B and C
    - B has its own version of foo, but C does not.
- If procedure bar gets called, which version of foo is called?

```
void bar (A a) {
  a.foo();
}
```

A. A's version
B. B's version
C. C's version
D. All of them, and we choose between the result when they return
E. It cannot be statically determined in general

- Correct Answer: E
- Explanation: consider

```
A obj1 = new A();
B obj2 = new B();

void bar(A a) {
    a.foo();
}

bar(obj1);      // works, executes A's
version
bar(obj2);      // works, executes B's
version
```

- Dynamic Control Flow
  - Calling different functions - a different approach via indirect jumps
    - Because **functions have addresses**
      - We can make **pointers to functions**
    - Useful for
      - Polymorphism
      - Generic operations
      - Jump tables
    -

```
    beq r0, L2
L1: ld $func1, r1 # run func1 if r0 != 0
    br L3
L2: ld $func2, r1 # run func2 if r0 == 0
L3: gpc $2, r6
    j (r1)
    ...
```

- Dynamic jumps in C
  - **Function pointer**
    - a variable that stores a pointer to a procedure
    - declared as
      `<return-type> (*<variable-name>) (<formal-argument-list>);`
    - used to make dynamic call
      `<variable-name>(<actual-argument-list>);`

- Example
    - ```
      void ping() {
          // smth
      }

      void foo() {
          void (* ptr_func) ();
          ptr_func = ping;
          ptr_func();
      }
      ```
        - assign using procedure name without `()`, call using variable name with `()`

```
int (*x) (char *);
```
What does the variable x above represent?

A. A function that receives a char pointer as argument and returns an int pointer

B. A function that receives a function pointer as argument and returns an int pointer

C. A pointer to a function that receives a char pointer as argument and returns an int

D. A pointer to a function that receives a function pointer as argument and returns an int

E. What you get when you fall asleep on your keyboard

- - Correct Answer: C

Mar. 4th - Unit 1f: Cont.d

```
int (*x) (char *);
```
Which of the following is a correct way to call the procedure to which x points?
Assume we have a char* c that has been set to a properly allocated chunk of memory.

```
A. int i = *(*x)(c);

B. int i = *x(*c);

C. int i = x(c);

D. int i = x(*c);

E. int i = *((int*) c);
```

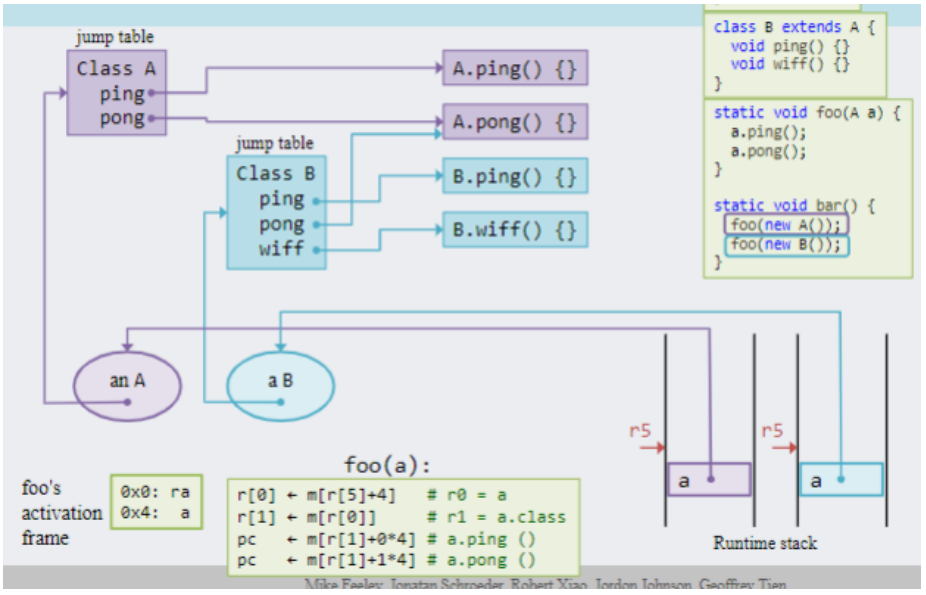
- - Correct Answer: C
- Polymorphism
    - Invoking a method on an object in Java
        - variable that stores the object has a static type (apparent type)
        - the object reference is dynamic and so is its type
```

        - object's actual type must be a subtype of the apparent type of the referring variable
        - but object's actual type may override methods of the apparent type
    - Polymorphic Dispatch
        - target method address depends on the type of the reference object
        - one call site can invoke different methods at different times
        - ```
          class A {
              void ping() {}
              void pong() {}
          }

          class B extends A {
              void ping() {}
              void wiff() {}
          }
          ```
        - ```
          static void foo(A a) {
              a.ping();
              a.pong();
          }

          static void bar() {
              foo(new A());
              foo(new B());
          }
          ```
- Polymorphic dispatch
    - Method address is determined dynamically
        - compiler cannot hard code target address in procedure call
        - instead, compiler generates code to lookup procedure address at runtime
        - address is stored in memory in the object class' jump table
    - Class **jump table**
        - every class is represented by a class object
        - objects store a pointer to their class object
        - the class object stores the class' jump table
        - the jump table stores the address of methods implemented by the class
    - Static and Dynamic of method invocation
        - address of jump table is determined dynamically

        - objects of different actual types will have different jump tables
    - method's offset into jump table is determined statically



- - Same procedure should be expected to be at the same offset for subclasses as their parent class

- Polymorphism in C
    - Use `struct` to store jump table
        - Declaration of class

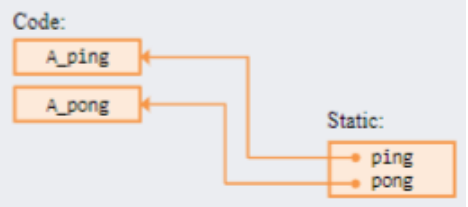
```
struct A_class {
    void (* ping) (void *);
    void (* pong) (void *);
};
```

        - Declaration of instance methods

```
void A_ping(void * thisv) {
    printf("A_ping\n");
}
void A_pong(void * thisv) {
    printf("A_pong\n");
}
// ???
void A_ping(void * thisv) {
    struct A * this = thisv;
    printf("A_ping %d\n", this -> i);
}
```

        - Static allocation and initialization of class object (i.e. jump table)

```
struct A_class A_class_table {
    A_ping, A_pong;
};
```



- Object (instance of class)
    - Object template

```
struct A {
    struct A_class * class; // pointer to
class object
    int i;                  // instance's
attribute
};
```

    - Constructor method

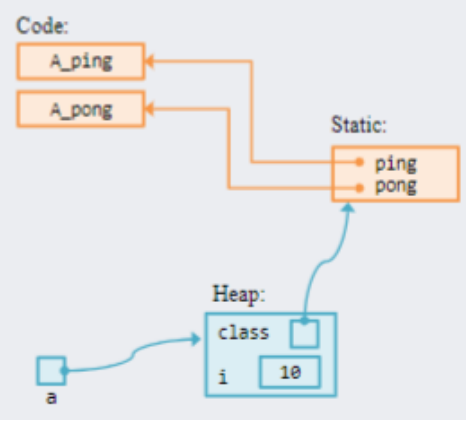
```
struct A * new_A(int i) {
    struct A * obj = malloc(sizeof(struct
A));
    obj -> class = &A_class_table;
    obj -> i = i;
};
```

    - Allocating an instance

```
struct A * a = new_A(10);
```

    - Calling instance methods

```
a -> class -> ping(a);
a -> class -> pong(a);
```

- 

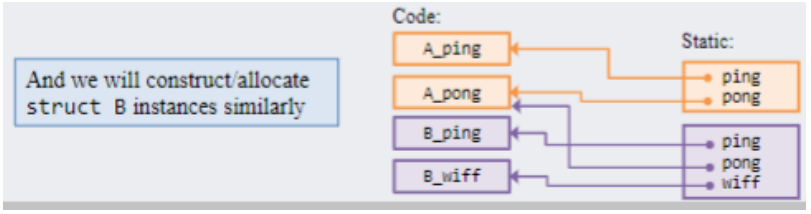
- class `B extends A`
  - `B`'s class struct (jump table) is a super-set of `A`'s

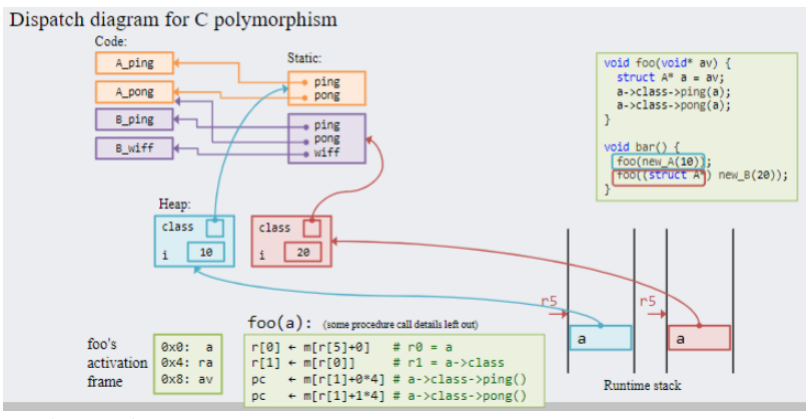
```
struct B_class {
    void (* ping) (void *);
    void (* pong) (void *);
    void (* wiff) (void *);
};
```

  - `B`'s method declarations and class object (static) allocation

```
void B_ping(void * thisv) {
    printf("B_ping\n");
}
void B_wiff(void * thisv) {
    printf("B_wiff\n");
}

struct B_class B_class_table = {
    B_ping, A_pong, B_wiff;
};
```

  - 

  - 

- ISA for polymorphic dispatch.

- How to compile `a -> class -> pong()`?
- Pseudocode: `pc <- m[r[1] + 0 * 4]`
- new ISA
Name	Semantics	Assembly	Machine
jump	pc ← a	j a	b--- aaaaaaaa
indirect jump	pc ← r[t] + o (or r[t] + pp*2)	j o(rt)	ctpp
dbl-ind jump b+o	pc ← m[r[t] + o] (or m[r[t] + pp*2])	j *o(rt)	dtpp

- What is the difference between these two C snippets? Assume `bat` is called.

(1)
```
void foo() { printf("foo\n"); }

void go(void(*proc)()) {
  proc();
}

void bat() {
  go(foo);
}
```

(2)
```
void foo() { printf("foo\n"); }

void go() {
  foo();
}

void bat() {
  go();
}
```

A. (2) calls `foo`, but (1) does not
B. (1) is not valid C
C. (1) jumps to `foo` using a dynamic address and (2) a static address
D. They both call `foo` using dynamic addresses
E. They both call `foo` using static addresses

- Correct Answer: C

Mar. 6th - Unit 1f: Cont.d

- Generic Functions
    - Consider the linked list implementation and print procedure below

```
struct node {
    int key;
    int value;
    struct node * next;
};

void print_list(struct node * list) {
    while (list) {
        printf("%d: %d\n", list -> key, list -> value);
        list = list -> next;
    }
}
```

    - Issues:
        - The `print_list` procedure is very specific
            - What if we need to print in a different format?

```
void print_list_full(struct node *
list) {
    while(list) {
        printf("key: %d, value: %d\n",
list -> key, list -> value);
        list = list -> next;
    }
}
```

- What if we want to print to a file?
- What if we want to perform another task (besides printing) with each key-value pair?

- Solution: function parameter
  - Extra parameter to `print_list`: what to do with each key-value pair
  - ```
    void follow_list(struct node * list,
    void (* fn)(int, int)) {
        while(list) {
            fn(list -> key, list ->
    value);
            list = list -> next;
        }
    }
    ```
  - Function pointer `void (* fn)(int, int)` decides what to do with each key-value pair
- What to do with a key-value pair?
  - ```
    void print_element(int key, int value)
    {
        printf("%10d: %10d\n", key,
    value);
    }

    void print_element_full(int key, int
    value) {
        printf("key: %d, value: %d\n",
    key, value);
    }

    follow_list(list, print_element);
    follow_list(list, print_element_full);
    ```

```
follow_list:
  deca r5
  st r6, (r5)  # save own r.a.
  ld 4(r5), r0 # r0 = list
loop:
  beq r0, L0   # while (list)
  ld 8(r5), r1 # r1 = fn
  ld 0(r0), r2 # r2 = list->key
  ld 4(r0), r3 # r3 = list->value
  deca r5
  deca r5      # allocate arg space for fn
  st r2, 0(r5) # arg0 = key
  st r3, 4(r5) # arg1 = value
  gpc $2, r6   # prep r.a. for fn
  j (r1)       # call fn
  inca r5
  inca r5      # recover arg space for fn
  ld 4(r5), r0 # r0 = list
  ld 8(r0), r0 # list = list->next
  br loop      # repeat from loop
L0:
  ld (r5), r6  # restore r.a.
  inca r5      # recover own r.a. space
  j (r6)       # return
```

- Other uses of function pointers
  - e.g. Quicksort, for sorting integers
  -
```
int partition (int* array, int left, int right, int pivotIndex) {
  int pivotValue, t;
  int storeIndex, i;

  pivotValue         = array [pivotIndex];
  array [pivotIndex] = array [right];
  array [right]      = pivotValue;
  storeIndex = left;
  for (i=left; i<right; i++)
    if (array [i] <= pivotValue) {
      t                  = array [i];
      array [i]          = array [storeIndex];
      array [storeIndex] = t;
      storeIndex += 1;
    }
  t                  = array [storeIndex];
  array [storeIndex] = array [right];
  array [right]      = t;
  return storeIndex;
}
void quicksort (int* array, int left, int right) {
  int pivotIndex;

  if (left < right) {
    pivotIndex = partition (array, left, right, left + (right-left)/2);
    quicksort (array, left,           pivotIndex - 1);
    quicksort (array, pivotIndex + 1, right         );
  }
}
void sort (int* array, int n) {
  quicksort (array, 0, n-1);
}
```
  - The logic/code for **`quicksort/partition`** is **mostly** type-independent
    - change parameter to sort anything
      - actually, like Java, array parameter will be a pointer to anything (or anything the same size as a pointer)

```
int partition (  void** array, int left, int right, int pivotIndex) {
  void* pivotValue, * t;
  int   storeIndex,   i;

  pivotValue         = array [pivotIndex];
  array [pivotIndex] = array [right];
  array [right]      = pivotValue;
  storeIndex = left;
  for (i=left; i<right; i++)
    if (array [i] <= pivotValue) {
      t                  = array [i];
      array [i]          = array [storeIndex];
      array [storeIndex] = t;
      storeIndex += 1;
    }
  t                  = array [storeIndex];
  array [storeIndex] = array [right];
  array [right]      = t;
  return storeIndex;
}
```

Actually, only 3 parts of the code are type-dependent

- array parameter – easy to deal with
- pivotValue type – easy to deal with
- comparison – ???

      -
        - **`void *`** means something
        - `<=` operator may not be meaningful for all types

- In Java,

```
int partition (Comparable<T> array[], int left, int right, int pivotIndex) {
  Comparable<T> pivotValue, t;
  int storeIndex, i;

  pivotValue         = array [pivotIndex];
  array [pivotIndex] = array [right];
  array [right]      = pivotValue;
  storeIndex = left;
  for (i=left; i<right; i++)
    if (array [i] .compareTo (pivotValue)) <= 0) {
      ...
}
```

```
class ComparableInteger<Integer> extends Integer {
  @Override
  int compareTo(Integer i) {
    return intValue() < i.intValue()? -1: intValue == i.intValue()? 0: 1;
  }
}
```

- In C, use a comparator function pointer

```
int partition(void* * array, int left,
int right, int pivotIndex, int (* cmp)
(void*, void*)) {
    // cmp is the specific function
that determines how to compare the
elements.
}
```

-

```
int partition (void** array, ... , int (*cmp) (void*, void*)) {
  void* pivotValue, *t;
  int  storeIndex, i;

  pivotValue         = array [pivotIndex];
  array [pivotIndex] = array [right];
  array [right]      = pivotValue;
  storeIndex = left;
  for (i=left; i<right; i++)
    if (cmp (array [i], pivotValue) <= 0) {
      t                  = array [i];
      array [i]          = array [storeIndex];
      array [storeIndex] = t;
      storeIndex += 1;
    }
  t                  = array [storeIndex];
  array [storeIndex] = array [right];
  array [right]      = t;
  return storeIndex;
}
```

formerly,
array[i] <= pivotValue

- Side note: avoiding `void *` confusion
  - It is a pointer type that cannot be dereferenced (directly)
    - **opaque** pointer
  - in C we used it to represent a pointer to **anything**
  - before using, must be casted to another pointer type
    - There is no type checking on this type case
    - casting can be done implicitly; explicit cast is not needed
  - code with `void*` can be confusing, especially when encountering `verb* *`
  - C's `typedef` statement
    - creates a new type name for a type expression
    - leads to more readable code

- common convention: end type names with a "_t"
- Using void*:

```
int partition ( void** array, ..., int (*cmp) (void*, void*)) {
  void* pivotValue, *t;
  ...
}
```

Using new type defined by typedef:

```
typedef void* element_t;

int partition ( element_t* array, ..., int (*cmp) (element_t, element_t)) {
  element_t pivotValue, t;
  ...
}
```

- Using the parametrized Quicksort
  - ```
    int cmpIntegers(void* av, void* bv) {
        int * a = av;
        int * b = bv;
        return *a < *b ? -1 : (*a == *b ? 0 : 1);
    }

    int a[] = {3, 8, 1};
    int * pa[] = {a, a + 1, a + 2};
    sort((void* *) pa, 3, cmpIntegers);
    // Note: pa's contents will be rearranged so its dereferenced elements will be in increasing order; *pa[0] < *pa[1] < *pa[2], the array a is unchanged
    ```
  - To sort strings (with typedef):

```
int cmpStrings (element_t av, element_t bv) {
  char* a = av;
  char* b = bv;
  return *a < *b ? -1: *a == *b ? 0: 1;
}
```

or simply:

```
return strcmp(a, b);
```

```
char* array[] = {"Psyduck", "Bulbasaur", "Meowth", ..., "Eevee"};
sort ((element_t*) array, sizeof (array) / sizeof (array[0]), cmpStrings);
```

Mar. 8th - Unit 1f: Cont.d

- Function pointers in Racket
  - `map` receives a function and two lists, applies function to pairs of elements
    - `(map + (list 1 4 3) (list 7 2 5)) => (list 8 6 8)`
    - `(map (lambda (a b) (+ a b)) (list 1 4 3) (list 7 2 5)) => (list 8`
    - `(map max (list 1 4 3) (list 7 2 5)) => (list 7 4 5)`
  - Other iterators
    - `(foldl + 0 (list 1 2 3))`
```

- `(filter (lambda (a) (> a 3)) (list 1 2 3 4 5))`

- Implementing `map` in C
  - `map` definition (using function pointer as argument)

```
void map(int (* fn) (int, int), int n, int *
s0, int * s1, int * d) {
    int i;
    for (i = 0; i < n; i += 1) {
        d[i] = fn(s0[i], s1[i]);
    }
}
```

  - Example function to be applied to two elements

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

  - Calling map with add

```
map(add, 3, a, b, c);
```

    - To be even more generic, we can use arrays of `void*` instead of `int`

- Implementing `foldl`

```
int foldl(int (* fn)(int, int), int v, int * a, int
n) {
    int i;
    for (i = n - 1; i >= 0; i -= 1) {
        v = fn(v, a[i]);
    }
    return v;
}
```

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

```
printf("%d\n", foldl(add, 0, a, 3));
```

  - Could we use this to concatenate strings? YES (as long as there is enough space to hold the strings)

        - However, we need to be careful with dynamically allocated memory to avoid leaks
        - TODO: look up the C library function `realloc`
- Reference counting with function pointers
    - Recall A6: `value` needs the same refcount as the element (because `value` can't be automatically be freed otherwise)
    -
```c
struct element {
  int num;
  char* value;
};

struct element* element_new(int num, char* value) {
  struct element* e = rc_malloc(sizeof(*e));
  e->value = rc_malloc(strlen(value) + 1);
  ...
}

void element_keep_ref(struct element* e) {
  rc_keep_ref(e->value);
  rc_keep_ref(e);
}

void element_free_ref(struct element* e) {
  rc_free_ref(e->value);
  rc_free_ref(e);
}
```
    - What if our program has multiple structure definitions with nested allocations?
    - Finalizers:

```c
struct element {
  int num;
  char* value;
};

void element_finalizer(void* ev) {
  struct element* e = ev;
  free(e->value);
}

struct element* element_new(int num, char* value) {
  struct element* e = rc_malloc(sizeof(*e));
  e->value = malloc(strlen(value) + 1);
  ...
}

void element_keep_ref(struct element* e) {
  rc_keep_ref(e);
}

void element_free_ref(struct element* e) {
  rc_free_ref(e);
}
```

But how/when does this finalizer get called? How do we "automate" it?

- Fixing the refcount library:

```c
struct rc_metadata {
  int ref_count;
  void (*finalizer) (void* ev); // finalizer associated with each allocation
};

void* rc_malloc(int size, void (*ef) (void* ev)) {
  struct rc_metadata* md = malloc(size + sizeof(struct rc_metadata));
  md->ref_count = 1;
  md->finalizer = ef;

  return md + 1; // address of payload
}

void rc_free_ref(void* p) { // p is a payload address
  struct rc_metadata* md = p;
  md -= 1; // set md to start of metadata
  (md->refcount) -= 1;
  if (md->refcount == 0) {
    md->finalizer(p); // call the allocation-specified finalizer
    free (ref_count); // free the entire allocation
  }
}
```

We should actually check that the provided finalizer is not NULL before calling it

  - All `rc_malloc` now receives two parameters, the size of data, and function pointer to deallocate
  - should be `free(md)`
- For `struct` without nested allocations, we can use `NULL` for finalizer argument
- Usage of the modified `rc_malloc`, etc.

```c
struct element {
  int num;
  char* value;
};

void element_finalizer(void* ev) {
  struct element* e = ev;
  free(e->value);
}

struct element* element_new(int num, char* value) {
  struct element* e = rc_malloc(sizeof(*e), element_finalizer);
  e->value = malloc(strlen(value) + 1);
  ...
}

void element_keep_ref(struct element* e) {
  rc_keep_ref(e);
}

void element_free_ref(struct element* e) {
  rc_free_ref(e);
}
```

Just use `rc_keep_ref(e)` and `rc_free_ref(e)`, for all allocations

- `switch` statement
-

```c
int i;
int j;

void foo() {
  switch (i) {
    case 0:  j = 10; break;
    case 1:  j = 11; break;
    case 2:  j = 12; break;
    case 3:  j = 13; break;
    default: j = 14; break;
  }
}
```

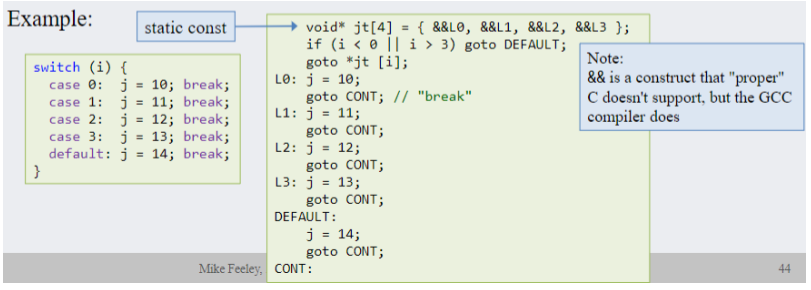
```c
void bar() {
  if (i == 0)
    j = 10;
  else if (i == 1)
    j = 11;
  else if (i == 2)
    j = 12;
  else if (i == 3)
    j = 13;
  else
    j = 14;
}
```

  - Semantically, the same as simplified nested/cascading if statements
    - where condition of each `if` tests the same variable
    - unless you leave out the `break` at the end of a case block
  - Why bother?
    - for humans, ease of writing, and reading of code?
    - for compilers, permitting a more efficient implementation?

- Implementing switch statements
    - anything more to consider than just conditionals?
- Human vs compiler
    - Benefits for human: the syntax models a common idiom, choosing one computation from a set
    - Restrictions
        - Case labels must be **`static, cardinal`** values
            - Cardinal value is a number that specifies a position relative to the beginning of an ordered set
            - integers are cardinal, strings are not
        - Case labels must be compared for equality to a single dynamic expression
            - some languages permit the expression to be an inequality
    -

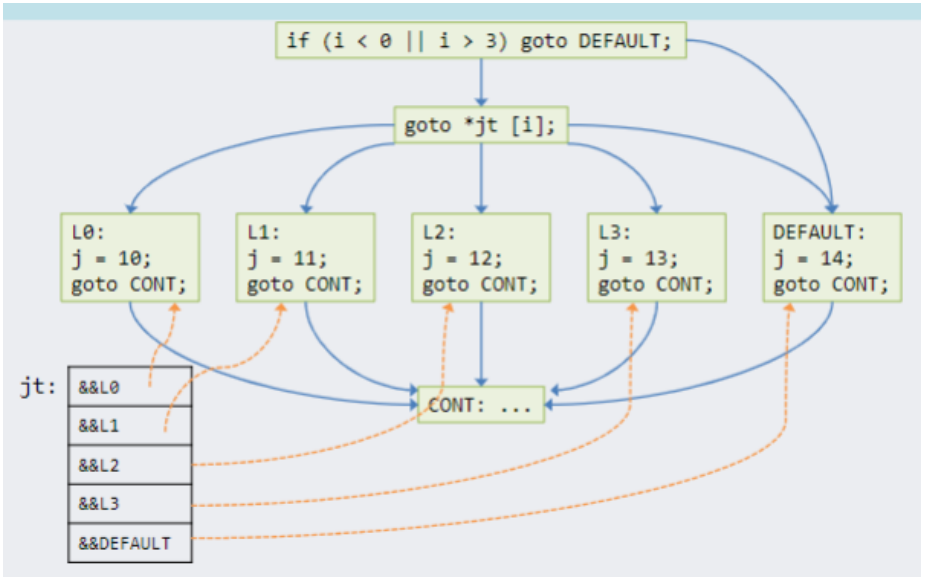
```
switch (pokemonType) {
  case "water": ...
  case "grass": ...
  case "psychic": ...
}
```

```
switch (i, j) {
  case i > 0: ...
  case i == 0 && j > a: ...
  case i < 0 && j == a: ...
}
```

- Why compiles like **`switch`** statements
    - Switch condition evaluates to a number
    - Each case arm has a distinct number
    - So, build a table with the address of every case arm, indexed by case value
    - Switch by indexing into this table and jumping to matching case arm (a **jump table**)
    - 

Mar. 11th - Unit 1f: Cont.d, Unit 2a: IO & Asynchrony

- **`switch`** statement control flow with jump table

- 

- Happy compilers mean happy people
    - Computation can be much more efficient (compare the running time to `if`-based alternative)
        - save time on condition testing
    - Could this all go horribly wrong?
        - construct a switch statement where this implementation technique is a really bad idea
- Basic implementation strategy
    - General form of a **`switch`** statement

```
switch(<cond>) {
    case <label_i>: <code_i>
    default:        <code_default>
}
```

    - Naive strategy

```
goto address of code_default if cond >
max_label_value
goto address of in jumptable[label_i]

statically: jumptable[label_i] = address of code_i
forall label_i
```

    - 2 additional considerations
        - Case labels are not always continuous
            - `case 0: ...; case 1: ...; case 2: ...; case 5: ...;`
        - the lowest case label is not always 0
            - `case 20: ...; case 21: ...; case 22: ...;`
    - Non-contiguous case labels

```
switch(i) {
    case 0:     j = 10; break;
    case 3:     j = 13; break;
    default:    j = 14; break;
}
```

- Problem: indexing based on case number, jump table will have some gaps
- Solution: fill gaps in jump table using default branch
OFFSET	
0	&&L0
1	&&DEFAULT
2	&&DEFAULT
3	&&L3
4	&&DEFAULT

- Case labels not starting at 0

```
switch(i) {
    case 1000:      j = 10; break;
    case 1001:      j = 11; break;
    case 1002:      j = 12; break;
    case 1003:      j = 13; break;
    default:        j = 14; break;
}
```

- Problem: invalid indices (negative numbers), or many unused (default) entries in the table
- Solution: Normalize the case labels relative to 0.

- Implementing `switch` statements
  - Choose strategy
    - use jump table unless case labels are sparse or there are very few of them
    - use nested `if`-statements otherwise
  - Jump table strategy
    - Statically: build jump table for all label values between lowest and highest
    - Generate code to:
      - `goto` default if condition is less than minimum case label or greater than maximum
      - normalize condition value to lowest case label

- use jump table to go directly to code-selected case arm

- ```
  goto address of code_default if cond < min_label_value
  goto address of code_default if cond > max_label_value
  goto address in jumptable [cond - min_label_value]

  statically: jumptable [i - min_label_value] = address of code_i
       forall i: min_label_value <= i <= max_label_value
  ```

```
switch (i) {
  case -20: ... break;
  case 0: ... break;
  case 5: ... break;
  case 42: ... break;
  default: ... break;
}
```

Which of the following implementations is the most appropriate in this case?

A. A jump table

B. A sequence of "if" statements

C. Either option is equally appropriate

D. Neither – we're not even going to bother with implementing this

- Correct Answer: B
- Explanation: the case label range is 62, large gaps between case labels, base + offset access can only support maximum of 254 bytes of offset (maximum size should be around 63 entries)

```
switch (i) {
  case -207: ... break;
  case -208: ... break;
  case -209: ... break;
  case -210: ... break;
  case -211: ... break;
  case -212: ... break;
  case -214: ... break;
  case -215: ... break;
  default: ... break;
}
```

Which of the following implementations is the most appropriate in this case?

A. A jump table

B. A sequence of "if" statements

C. Either option is equally appropriate

D. Neither – we're not even going to bother with implementing this either

- Correct Answer: A
- Explanation: Normalize the case labels relative to 0

- ```
  switch (i) {
    case 20:  j = 10; break;
    case 21:  j = 11; break;
    case 23:  j = 13; break;
    default:  j = 14; break;
  }
  ```

```
     static const void* jt[4] = { &&L20, &&L21, &&DEFAULT, &&L23 };
     if (i < 20 || i > 23) goto DEFAULT;
     goto *jt [i-20];
L20: j = 10;
     goto CONT; // "break"
L21: j = 11;
     goto CONT;
L23: j = 13;
     goto CONT;
DEFAULT:
     j = 14;
     goto CONT;
CONT:
```

-
```
foo:    ld   $i, r0          # r0 = &i
        ld   0x0(r0), r0     # r0 = i
        ld   $0xffffffed, r1 # r1 = -19
        add  r0, r1          # r0 = i-19
        bgt  r1, 10          # goto 10 if i>19
        br   default         # goto default if i<20
10:     ld   $0xfffffff9, r1 # r1 = -23
        add  r0, r1          # r1 = i-23
        bgt  r1, default     # goto default if i>23
        ld   $0xffffffec, r1 # r1 = -20
        add  r1, r0          # r0 = i-20
        ld   $jmptable, r1   # r1 = &jmptable
        j    *(r1, r0, 4)    # goto jmptable[i-20]
```

```
jmptable: .long case20         # & (case 20)
          .long case21         # & (case 21)
          .long default        # & (case 22)
          .long case23         # & (case 23)
```

```
case20:  ld   $0xa, r1        # r1 = 10
         br   done            # goto done
...
default: ld   $0xe, r1        # r1 = 14
         br   done            # goto done
done:    ld   $j, r0          # r0 = &j
         st   r1, 0x0(r0)     # j = r1
         br   cont            # goto cont
```

- ISA
  -

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

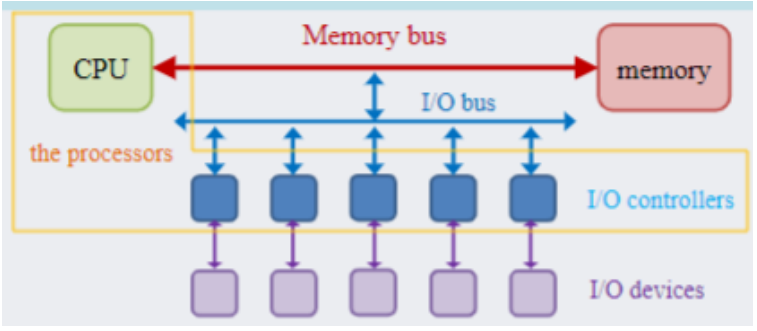
  -

Name	Semantics	Assembly	Machine
indirect jump	pc ← r[t] + o (or r[t] + p*2)	j o(rt)	ctpp

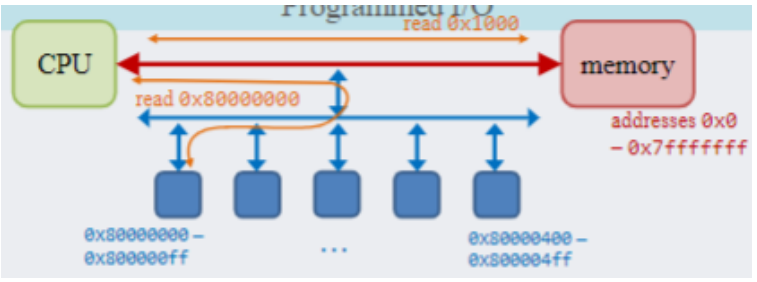
  -

Name	Semantics	Assembly	Machine
dbl-ind jump b+o	pc ← m[r[t] + o] (or m[r[t] + pp*2])	j *o(rt)	dtpp
dbl-ind jump indexed	pc ← m[r[t] + r[i]*4]	j *(rt, ri, 4)	eti- new!

Unit 2a: IO & Asynchrony

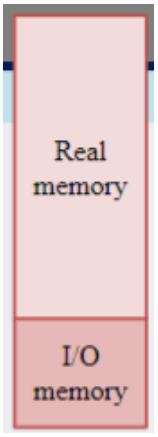


-
  - Memory bus
    - data/control path connecting CPU, main memory, and I/O bus
    - also called the **front-side** bus
  - I/O bus
    - data/control path connecting memory bus and I/O controllers
    - e.g. PCI
  - I/O controller
    - a processor running software (firmware)
    - connects I/O device to I/O bus

        - e.g. SCSI, SATA, Ethernet
    - I/O device
        - I/O mechanism that generates or consumes data
        - e.g. disk, radio, keyboard, mouse
- Talking to an I/O controller
    - Programmed I/O (PIO)
        - CPU transfers one **word** at a time between CPU and IO controller
        - typically use standard load/store instructions but to IO-mapped memory address
    - IO-mapped memory
        - memory addresses beyond the end of main memory
        - used to name IO controllers (usually configured at boot time)
        - loads and stores are translated into IO bus messages to controller
    - Example
        - 

        - read/write to controller at address `0x80000000`

```
ld $0x80000000, r0
st r1, (r0)
ld (r0), r1
```

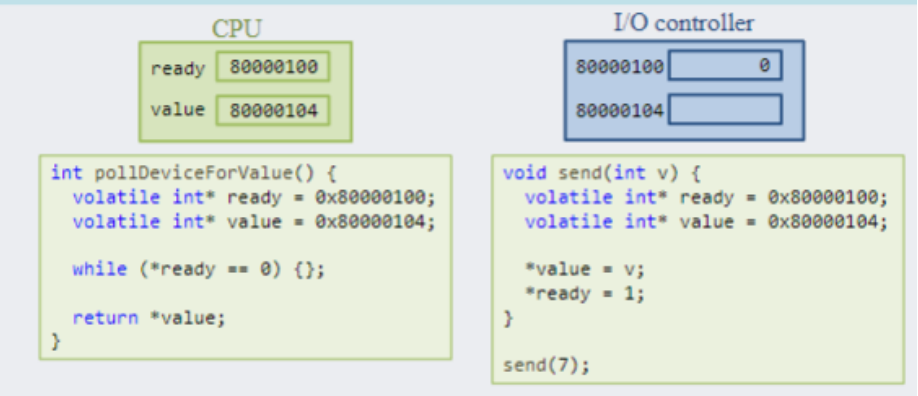
        - 

    - Limitations of PIO
        - Reading or writing large amount of data slows CPU
            - CPU must transfer one word at a time
            - controller/device is much slower than CPU

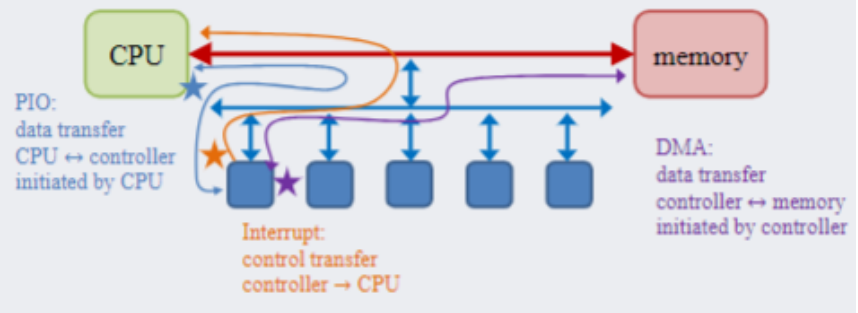
        - so, CPU runs at controller/device speed, mostly waiting for controller
    - IO controller cannot initiate communication
        - sometimes the CPU asks for data
        - but, sometimes the controller devices receives data for the CPU, without CPU asking (mouse click or network packet reception)
        - How does controller notify CPU that it has data the CPU should want
    - One not-so-good idea
        - CPU periodically asks what controllers offer that CPU needs
        - Drawbacks: asking for data takes time, might still not have any data
        - When is it OK? If there is a high chance of having data ready

Mar. 13th - Unit 2a: Cont.d

- Polling and IO memory

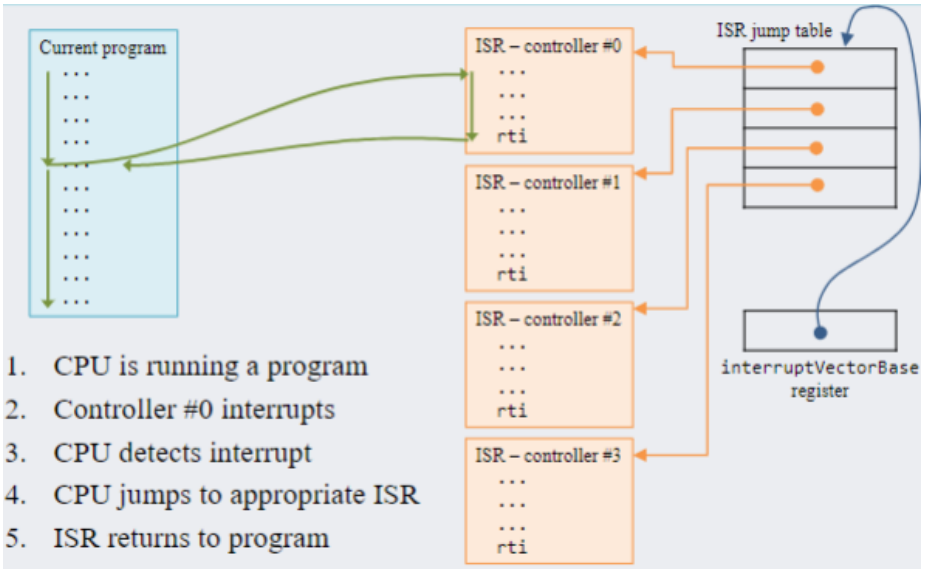


    - `volatile`: data is not guaranteed to be steady, can be changed by external factors
    - Reading (or writing) IO memory is slow
        - CPU → IO device: give me value
        - IO device → CPU: here is the value
    - Polling is OK if poll has low overhead, or has high probability of "hit"
- Key observation
    - CPU and IO controllers have independent processors
        - they should be allowed to work in parallel
        - either should be able to initiate data transfer to/from memory
        - either should be able to signal the other to get the other's attention

- Autonomous controller operation
    - **Direct Memory Access** (DMA)
        - controller can send/read data from/to any main memory address
        - the CPU is oblivious to these transfers
        - DMA addresses and sizes are **programmed** by CPU using PIO
    - CPU interrupts
        - controller can signal the CPU
        - CPU checks for interrupts on every cycle (fast, clock-speed poll)
        - CPU jumps to controller's **Interrupt Service Routine** if it is interrupting
    - 

- Adding interrupts to SM213
    - New special-purpose CPU registers
        - `isDeviceInterrupting`: set by IO controller to signal interrupt
        - `interruptControllerID`: set by IO controller to identify interrupting device
        - `interruptVectorBase`: **interrupt-handler** jump table, initialized at boot time
    - Modified fetch-execute cycle
        - ```
          while (true) {
            if (isDeviceInterrupting) {
              r[5] ← r[5] - 4;  m[r[5]] ← r[6];
              r[6] ← pc;
              pc ← interruptVectorBase[interruptControllerID];
            }
            fetch();
            execute();
          }
          ```
          "polls" on CPU register instead of I/O memory

          RTI: Return from Interrupt Instruction
          ```
          t ← r[6];
          r[6] ← m[r[5]]; r[5] ← r[5] + 4;
          isDeviceInterrupting ← 0;
          pc ← t;
          ```
        - The `if`-block is same as a procedure call
        - `interruptVectorBase`: contains the jump table addresses; `interruptControllerID`: controller number indexing
        - After interruption, the program needs to return from interruption
          ```

- 

- Architectural view of interrupts
  - 

- For which of these devices does PIO polling present a better use-case than interrupts?

A. Devices that process user input (e.g. keyboard, mouse)
B. Programmable timers (e.g., execute a process one second from now)
C. Clock (e.g. get current time of day)
D. Secondary storage (e.g. hard disks)
E. Imaging devices (e.g. cameras, scanners)

- - Correct Answer: C
  - Explanation: Polling - OK for frequent, regular requests, with high probability of new data (small amounts of data); Interrupts - Irregular, infrequent data, large amounts of data, triggered by outside/real-world events.

- A touchscreen monitor needs to notify the CPU that the user touched on a particular position on the screen. Which option is better suited for this operation?

A. CPU periodically requests for the last touched position
B. Monitor controller uses PIO to notify CPU
C. Monitor controller stores location in memory, CPU periodically checks memory for new touch
D. Monitor controller interrupts CPU, which runs a pre-determined interrupt handler
E. Monitor controller changes a pre-determined register with the position of the last touch

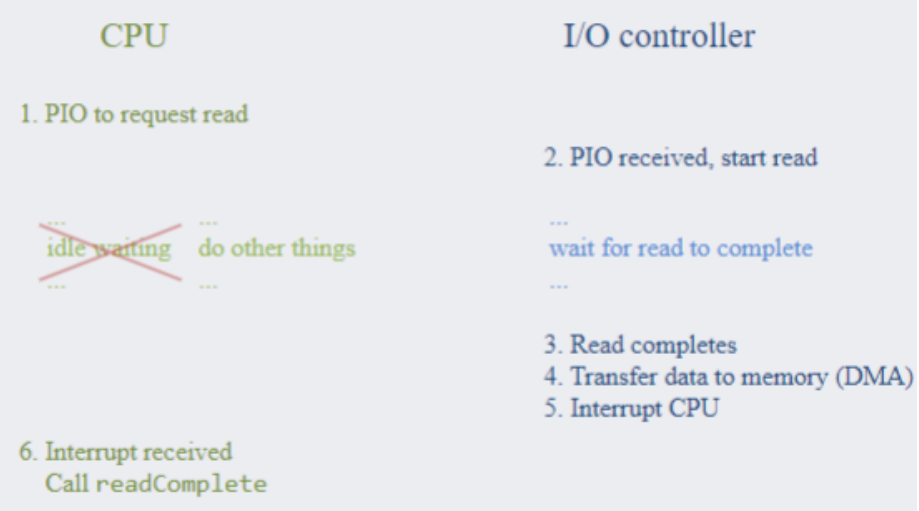
- - Correct Answer: D
  - Explanation: A - polling, inefficient; B - PIO initiated by CPU, not by IO controller; C - DMA, polling memory, inefficient

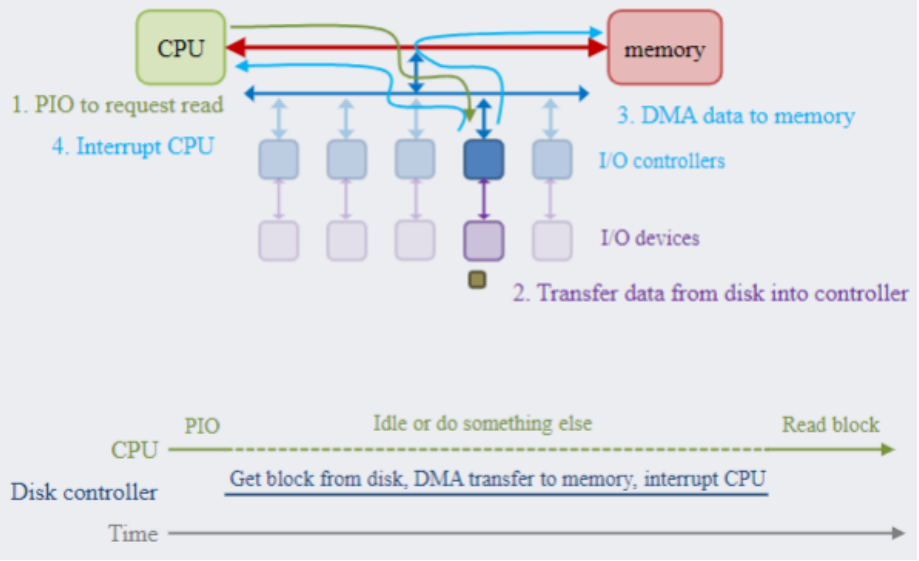
- The CPU decides to turn off the NumLock indicator on the keyboard. Which option is better suited for this operation?

A. CPU sends a PIO request to the keyboard device
B. CPU changes a specific register associated with the indicator
C. CPU changes a specific memory location in main memory associated with the indicator
D. CPU interrupts the keyboard controller requesting a change in indicator values
E. CPU halts until the user hits the Num Lock key

- - Correct Answer: A
  - Explanation: A - uses PIO, but not repeatedly using PIO, only a single request; B - waste register resources; C - too indirect, inefficient; D - CPU cannot interrupt; E - impractical

Programming with I/O

- Disk read timeline
  - A disk stores blocks of data. Each block has a number. CPU can request read or write to a block.
  - 

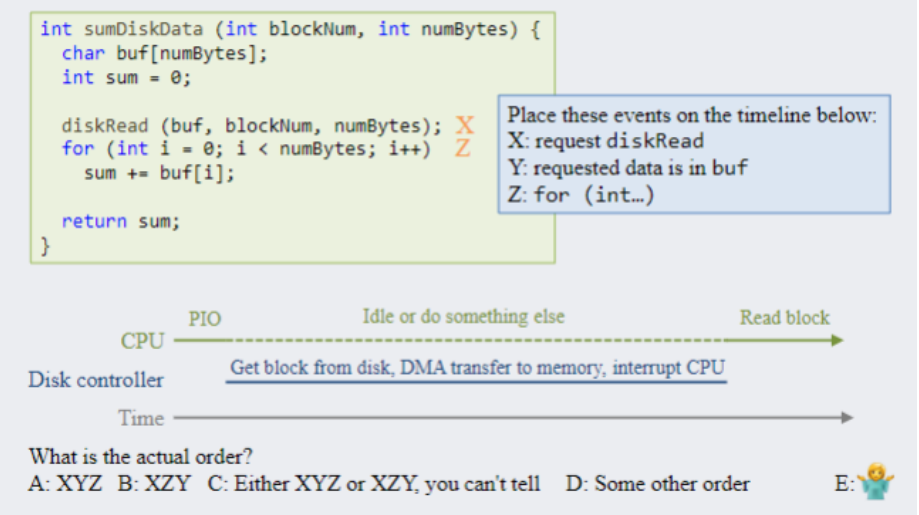
  - Instead of CPU idling, CPU should do other things
  - 

- Sequential execution
  - Consider a program that reads data from a disk
    - You can think of the read as having 3 parts
      - figure out what data you want
      - get the data
      - do smth with the data you got
    - these steps must happen in this order

- we think of these steps as executing in sequence
- ```
  int sumDiskData(int blockNum, int numBytes) {
      char buf[numBytes];
      int sum = 0;

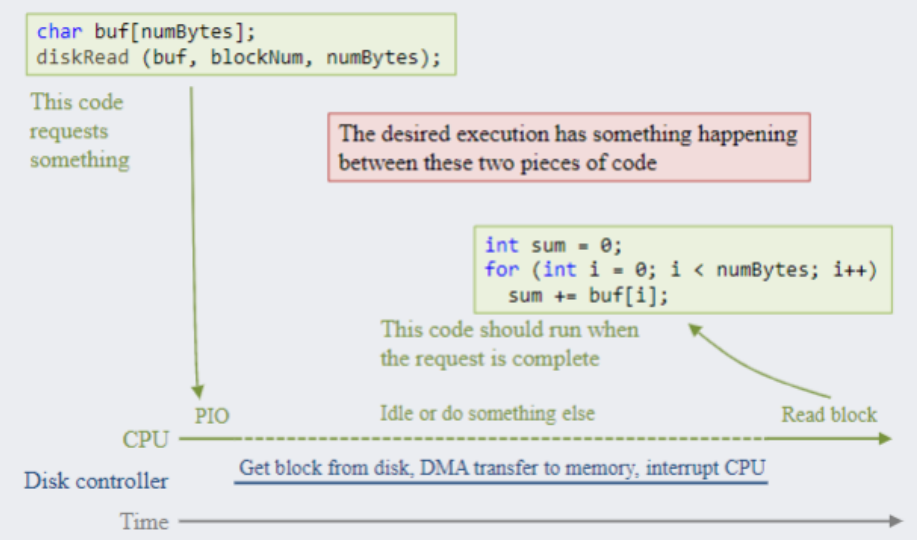
      diskRead(buf, blockNum, numBytes);
      for (int i = 0; i < numBytes; i += 1) {
          sum += buf[i];
      }

      return sum;
  }
  ```



- - Correct Answer: C
  - Explanation: because `diskRead` is slow, therefore, the program could execute the `for` loop before completing `diskRead`

Mar. 15th - Unit 2a: Cont.d

- Making code asynchronous
- 

- Writing asynchronous code in C
  ```

- Events and event handlers
    - things that cause asynchronous code to run are called **events**
    - the code that runs when an event occurs is called an **event handler** (or **callback**)
    - handlers are ***registered*** to a specific event
    - events are ***fired*** to trigger the execution of the handler
- In code
    - request and registration of event handler are listed together in code
    - this code continues, does not wait for event
    - handler runs asynchronously when event occurs
    - ```
      // this function is an event handler
      void computeSum() {
          // do smth after disk read
      }

      int sumDiskBlock(blockNum, numBytes) {
          char buf[numBytes];

          diskRead(buf, blockNum, numBytes,
      computeSum);
      }
      ```
- Disk read PIO
    - Expanded disk read with PIO
        - the message sent to disk has several fields
        - each field has different device-memory address
        - access it like a struct
            - writes are to **device memory**
            - they are messages across bus to device controller; they are not writes to main memory
    - ```
      void * diskAddress = (void *) 0x80001000;

      #DEFINE DISK_OP_READ 1
      #DEFINE DISK_OP_WRITE 2

      struct disk_ctrl {
          int op;
          char * buf;
          int blockNum;
          int numBytes;
      }
      ```

```
void diskRead(char * buf, int blockNum, int
numBytes, ...) {
    struct disk_ctrl * dc = diskAddress;

    // enqueue_handler (whenComplete, buf, blockNum,
numBytes)
    // the following access use base + offset, each
populated in IO ctrller
    // using one st instruction for each
    dc -> op = DISK_OP_READ;
    dc -> buf = buf;
    dc -> blockNum = blockNum;
    dc -> numBytes = numBytes;
}
```

- Implementing disk read (simplified)
    - Interrupt vector, device ID, and PIO address
        - initialize before all this starts
        - by operating system when device connects
        -
```
#define MAX_DEVICES
void (*interruptVector [MAX_DEVICES])();

int  diskID        = 4;
int* diskAddress = (int*) 0x80001000;

interruptVector [diskID] = diskISR;
```
    - Disk Read
        - register event handler
        - request block (PIO)
        -
```
void diskRead (char* buf, int blockNum, int numBytes
               void (*whenComplete) (char*, int, int)) {
  enqueue_handler (whenComplete, buf, blockNum, numBytes);
  // perform PIO ... more about this later
}
```
    - Interrupt handler
        - find event handler and fire event
        - firing means calling the handler procedure
        -
```
void diskISR() {
  struct handler_dsc {
    void (*handler) (char*, int, int);
    char* buf;
    int   blockNum;
    int   numBytes;
  };
  struct handler_dsc hd;
  dequeue_handler (&hd);
  hd.handler(hd.buf, hd.blockNum, hd.numBytes);
}
```

```
int computeSum (char* buf, int blockNum, int numBytes) {
  int sum = 0;
  for (int i = 0; i < numBytes; i++)
    sum += buf[i];
  return sum;
}

void sumDiskBlock (blockNum, numBytes) {
  char buf [numBytes];

  diskRead (buf, blockNum, numBytes, computeSum);
}
```

What procedure calls computeSum?

A. sumDiskBlock
B. the procedure that calls sumDiskBlock
C. another procedure in this program
D. the disk interrupt service routine
E. something else / I don't know

- Correct Answer: D
- Explanation: `computeSum` (callback parameter) is called when `diskRead` is complete, which is signaled through a CPU interrupt, launching `computeSum`

- Connecting asynchrony to program
  - How to use the value computed from the disk block
  - Suppose we want

```
void something() {
    ...
    int s = sumDiskBlock(blk, n);
    printf("%d\n", s);
}
```

  - Ordering in asynchronous programs
    - If something has to happen after an event
      - it must be triggered by the event
      - often this means it must be part of (or called by) event's handler
    - To print the sum

```
void computeSumAndPrint (char* buf, int blockNum, int numBytes) {
  int sum = 0;
  for (int i = 0; i < numBytes; i++)
    sum += buf [i];
  printf("%d\n", sum);
  free (buf);
}

void sumDiskBlock (blockNum, numBytes, whenComplete) {
  char* buf = malloc (numBytes);
  diskRead (buf, blockNum, numBytes, whenComplete);
}

void something() {
  sumDiskBlock (1234, 4096, computeSumAndPrint);
}
```

Huge problem:
- often there is a ton of stuff that depend on the returned data
- these must happen after a particular event

- Improving the code

```
void computeSumAndCallback (char* buf, int blockNum, int numBytes,
                            void (*callback) (int)) {
  int sum = 0;
  for (int i = 0; i < numBytes; i++)
    sum += buf[i];
  callback(sum);
  free(buf);
}
```

```
void computeSumAndCallback (..., void (*sumCallback) (int)) {
  int sum = 0;
  for (int i = 0; i < numBytes; i++)
    sum += buf[i];
  sumCallback(sum);
  free(buf);
}

void sumDiskBlock (blockNum, numBytes, whenComplete, sumCallback) {
  char* buf = malloc (numBytes);

  diskRead (buf, blockNum, numBytes, whenComplete, sumCallback);
}
```

```
void printInt (int i) {
  printf ("%d\n", i);
}

void something() {
  sumDiskBlock(1234, 4096, computeSumAndCallback, printInt);
}
```

- Top-level function calls need to pass all the proper sequence
- "callback hell"

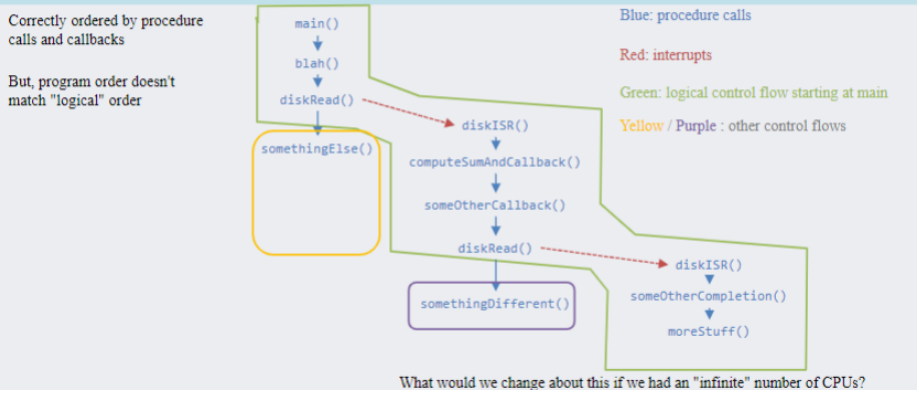
- Happy system, sad programmer
  - - Humans like synchrony
      - we expect each step of a program to complete before the next one starts
      - we use the result of previous steps as input to subsequent steps
      - with disks, for example,
        - we read from a file in one step and then usually use the data we've read in the next step
    - Computer systems are asynchronous
      - the disk controller takes 10-20 milliseconds (10^{-3}s) to read a block
        - CPU can execute 60 million instructions while waiting for the disk to complete one read
        - we must allow the CPU to do other work while waiting for I/O completion
      - many devices send unsolicited data at unpredictable times
        - e.g., incoming network packets, mouse clicks, keyboard-key presses
        - we must allow programs to be interrupted many, many times a second to handle these things
    - Asynchrony makes programmers sad
      - it makes programs more difficult to write and much more difficult to debug
- Possible solutions

- Accept the inevitable
  - use an event-driven programming model
    - event triggering and handling are de-coupled
  - a common idiom in many Java programs
    - GUI programming follows this model
  - *CSP* is a language boosts this idea to first-class status
    - no procedures or procedure calls
    - program code is decomposed into a set of sequential/synchronous processes
    - processes can fire events, which can cause other processes to run in parallel
    - each process has a guard predicate that lists events that will cause it to run
  - Javascript in web browsers and Node.js embrace asynchrony, albeit awkwardly
- Invent a new abstraction
  - an abstraction that provides programs the illusion of synchrony
  - but, what happens when
    - a program does something asynchronous, like disk read?
    - an unanticipated device event occurs?
- What's the right solution?
  - we still don't know — this is one of the most pressing questions we currently face

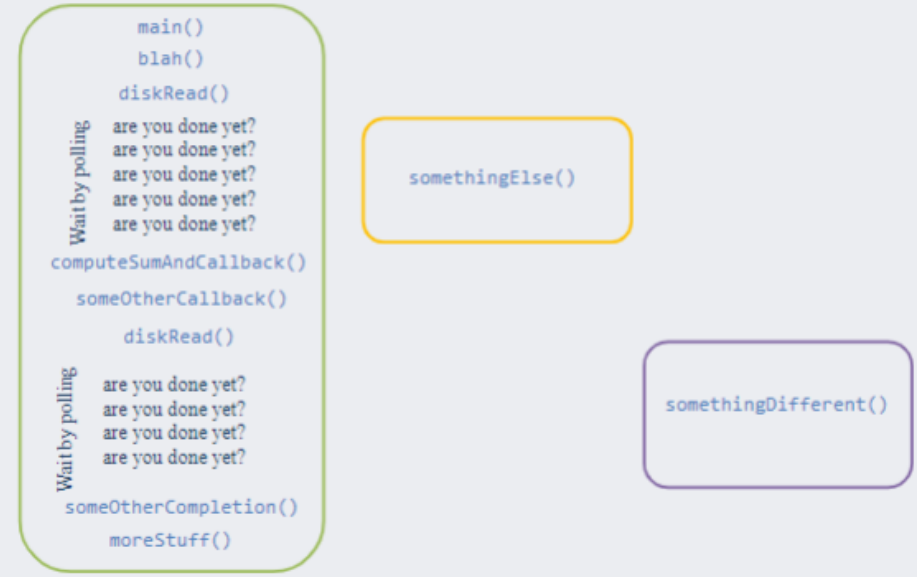
Mar. 18th - Unit 2b: Virtual Processors

- Issues introduced by IO devices
  - Ordering of program events with IO completion
    - `diskRead` triggers IO controller to start read process
    - program has things that can only run after that process completes
  - Do other things while waiting for IO event
    - need multiple independent streams of execution in program
    - one does read and continues after it completes
    - the other does something else in the meantime
  - Asynchronous Programming
    - Order
      - callback function that is called by completion interrupt
    - Multiple Streams
      - one stream continues with return from `diskRead` without the requested block
      - the other starts with when the interrupt calls the completion callback function
  - If something needs to happen after an event, then it must be triggered by the event
    - It must be part of (or called by) the event's handler
  - Problem
    - Usually we request data from IO and need it to proceed
    - Most of executable code depends on returned data

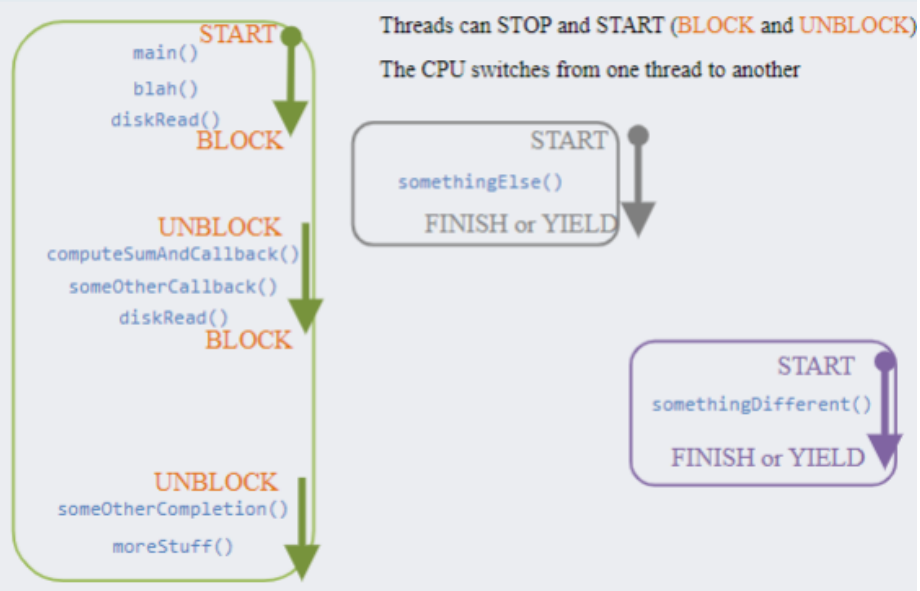
        - Sometimes a sequence of requests must be made that depend on each other
- Streams of control in an asynchronous program



    - We can keep logical flow in its own stream, if we have multiple CPUs, OK to wait at the old stream
    - Poll the devices: infinite CPUs



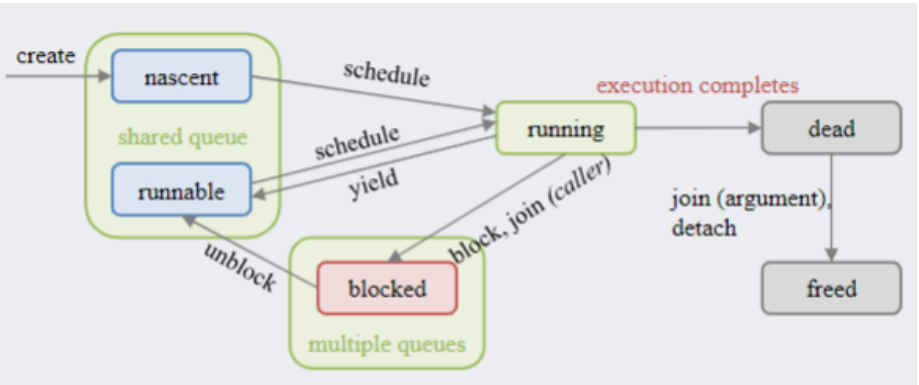
- The Virtual Processor
    - Thread
        - A single stream of synchronous execution of a program
            - the illusion of a single system
        - can be **stopped** and **restarted**
            - stopped when waiting for an event
            - restarted with the event fires
        - can co-exist with other threads sharing a single CPU
            - a scheduler multiplexes processes over processor
            - synchronization primitives are used to ensure mutual exclusion and for waiting and signaling
- Connecting program, and logical-order with threads
    - Threads can STOP and START (or BLOCK and UNBLOCK)
    - The CPU switches from one thread to another

- 

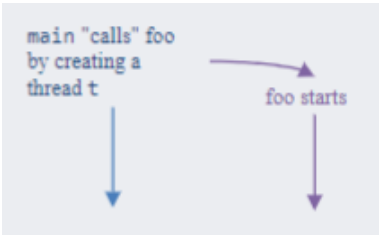
- `UThread`: a simple thread system for C
  - The UThread interface (`uthread.h`):

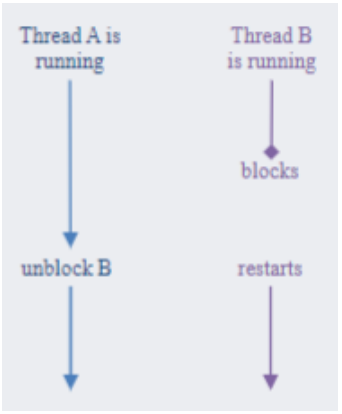
```
void  uthread_init (int num_processors);
uthread_t uthread_create (void* (*proc)(void*), void* arg);
void  uthread_detach (uthread_t t);
int   uthread_join (uthread_t t, void** vp);
uthread_t uthread_self ();
void  uthread_yield ();
void  uthread_block ();
void  uthread_unblock (uthread_t thread);
```

    - Explained:
      - `uthread_t` — thread id data type
      - `uthread_init` — called once to initialize the thread system
      - `uthread_create` — create and start a thread to run specified procedure
      - `uthread_yield` — temporarily stop current thread if other threads are waiting
      - `uthread_join` — join calling thread with specified other thread and get return value
      - `uthread_detach` — indicate no thread will join specified thread
      - `uthread_self` — a pointer to the Thread Control Block (TCB) of the current thread
      - `uthread_block` — block current thread
      - `uthread_unblock` — unblock specified thread and make it runnable
  - Thread status / operations DFA

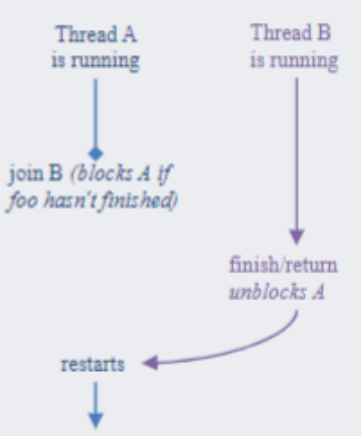


    - runnable: waiting and will run when a CPU becomes available
    - blocked: multiple queues, because threads can be blocked for multiple reasons, but will not become runnable until explicitly unblocked
    - running - join: "I need another thread's produced result, and wait for the other thread to complete"
    - dead - join: "Somebody has asked me to join them to give them my produced result"

- Thread operations - Create
  - 

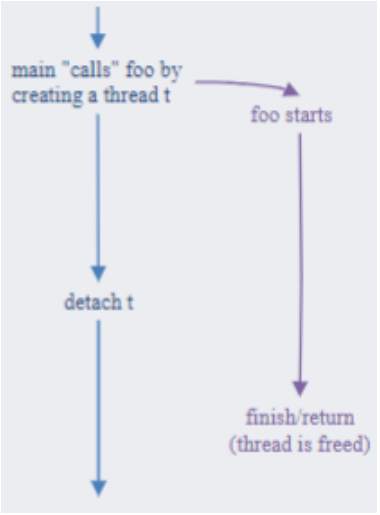
  - Creating and starting a thread
    - "Forks" the control stream
    - ```
      uthread_t t = uthread_create(foo, NULL);
      ```
    - Starts a new thread to execute a procedure
  - A created thread may not start right away
    - If there aren't enough CPUs available to run all threads, it will start when another thread yields (or when pre-empted)
    - created (nascent) threads are considered **`runnable`**
- Thread operations - Block and unblock
  - 

  - Blocking a thread
    - Save current thread's state, stop it, switch to a different thread
      ```
      uthread_block();
      ```
    - Thread cannot be rescheduled until explicitly unblocked (usually some criterion must be satisfied before the thread can be unblocked)
  - Unblocking a thread
    - Move the thread from its queue of blocked threads into the runnable queue
      ```
      unthread_unblock(t);
      ```
- Thread operations - join
      ```

- 

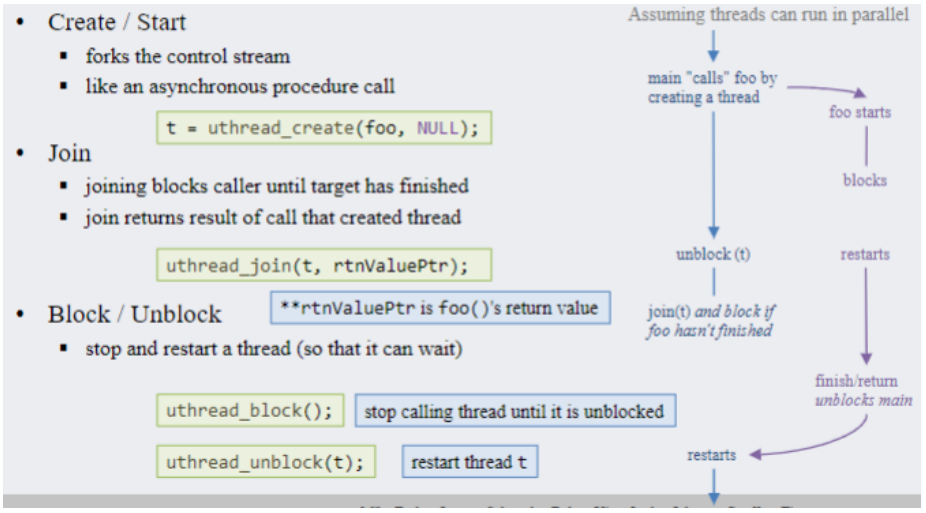
    - What happens when a thread finishes
        - Some other thread may want its output
        - We need to keep its information
    - Joining with a thread
        - Block current thread until a target thread completes

```
uthread_join(t, void* * rtnValuePtr);
```

        - Can obtain the return value of the target thread
        - If target thread has already finished, block is unnecessary
    - In effect, join may turn CPU back into a synchronous procedure call
        - But there may be other threads still running besides the joining thread and the target thread
- Thread operations - Detach
    - 

    - What if no thread is planning to join with it?
        - Detach the thread, notify it that it is not going to be joined
        - If the thread has already completed, free its resources
        - If it is still running, flag it to be freed as soon as it completes
- Thread operation - Yield
    - Yielding a thread
        - Save thread's state and switch to a different runnable thread

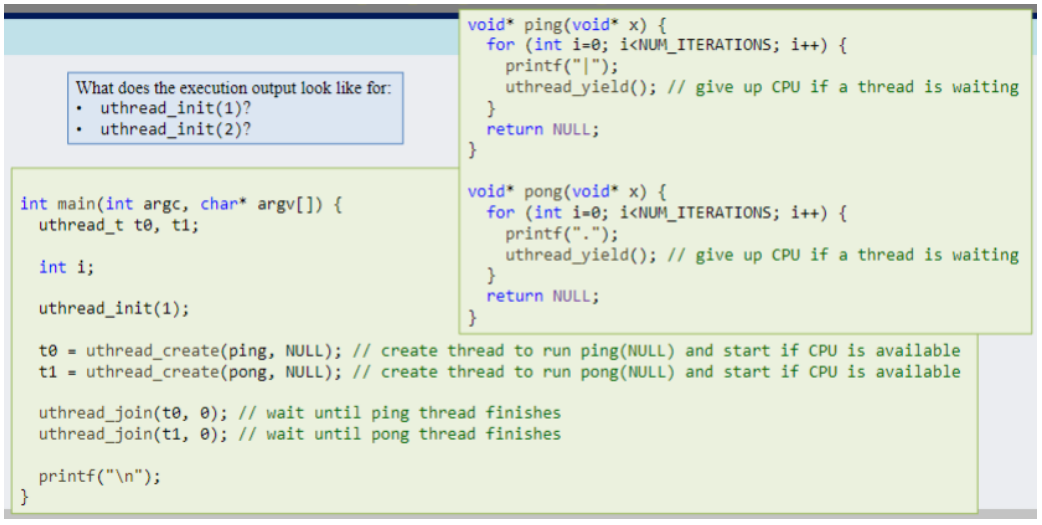
        - Thread can resume as soon as CPU becomes available (another thread yields or blocks)
        - Threads that yield remain runnable
    - Rescheduling a thread
        - Restore a thread's state and resume running it
    - Threads running on a CPU use resources
        - Registers, memory (activation frame), runtime stack
        - when a thread yields, these resources are backed up
        - when a thread resumes, these resources are restored
- Thread operation - overview



    - `rtnValuePtr`: reference to generic type

Mar. 20th - Unit 2b: Cont.d

- Example program



    - If we run with 1 virtual processor
        - `main`: creates thread `t0`, creates thread `t1`, (`t0, t1` in runnable queue); call `join(t0, 0)`, `main` is blocked, CPU becomes available

- `t0` starts to execute: goes into loop, print |, then yields (moves itself into runnable queue, `t1, t0`)
- `t1` starts to execute: goes into loop, print ., then yields (moves itself into runnable queue, `t0, t1`)
- `t0` will restart executing ..., then it prints alternating pattern |.|.|.
- `t0` completes executing, joins `main`; `t1` has one last iteration; `t1` is dead first, and then `main` calls join on `t1`

- With 2 (or more) virtual processors
  - Things can get complicated...

The `main` function creates two threads: `foo` and `bar`. Then, while both are running, suppose `foo` calls `join` on `bar`. What happens?

A. `bar` is blocked until `foo` completes
B. `foo` is blocked until `bar` completes
C. `main` is blocked until `bar` completes
D. main is blocked until both `foo` and `bar` complete
E. `bar` is forced to stop immediately

- Correct Answer: B

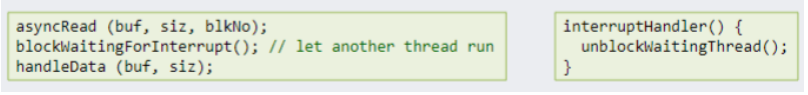
- Revisiting the disk read
  - A program that reads a block from disk
    - Want the disk read to be synchronous

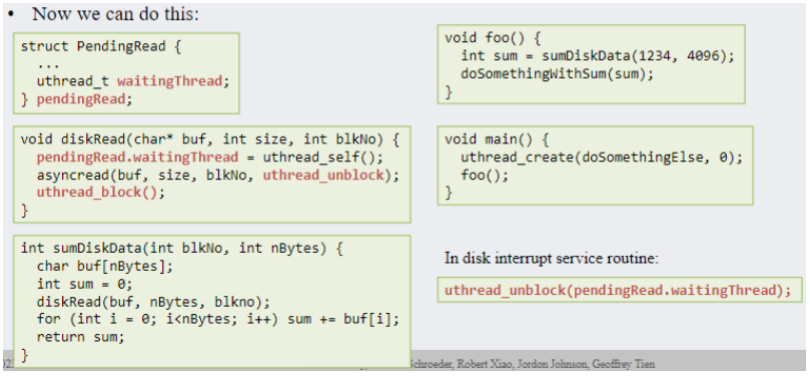
```
read(buf, siz, blkNo);      // read siz bytes
at blkNo into buf
handleData(buf, siz);       // now do
something with the block
```

    - but it is actually asynchronous, so we had

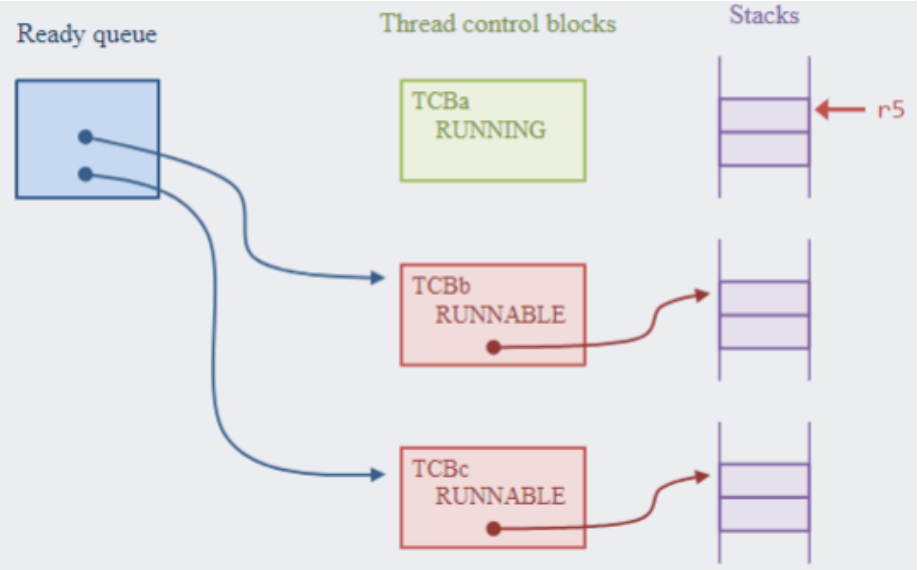
```
asyncRead(buf, siz, blkNo, handleData);
doSomethingElse();
```

  - As a timeline
    - two processors, two separate computations
  - "Synchronous" disk read using threads
    - Create two threads that CPU runs, one at a time
      - One for disk read
      - one for doSomethingElse
    - Illusion of synchronony
      - disk read blocks while waiting for disk to complete
      - CPU runs other threads while first thread is blocked
      - disk interrupt restarts the blocked thread

- 

  - Hiding asynchrony with thread
    - 

- Implementing threads
  - Key concept in implementation: **blocking** and **unblocking**
    - Stop a thread: no longer executing on a CPU
    - What happens to the thread?
    - What happens to the physical processor? CPU has hardware resources for executing processes (registers)
  - Thread management
    - Data structures and operations?
  - Thread state
    - when running: register file (CPU) and runtime stack (in physical memory, activation frames for sequence of called functions)
    - when stopped: thread-control block (TCB) object and runtime stack
  - Thread-control block
    - thread status
    - saved value of thread's stack pointer if it is not running
    - scheduling parameters (priority, quantum, preemptibility)
  - Ready queue
    - list of TCBs of all RUNNABLE threads
  - Blocked queues
    - list of TCBs of BLOCKED threads
    - May be more than one queue, depending on reason for being blocked

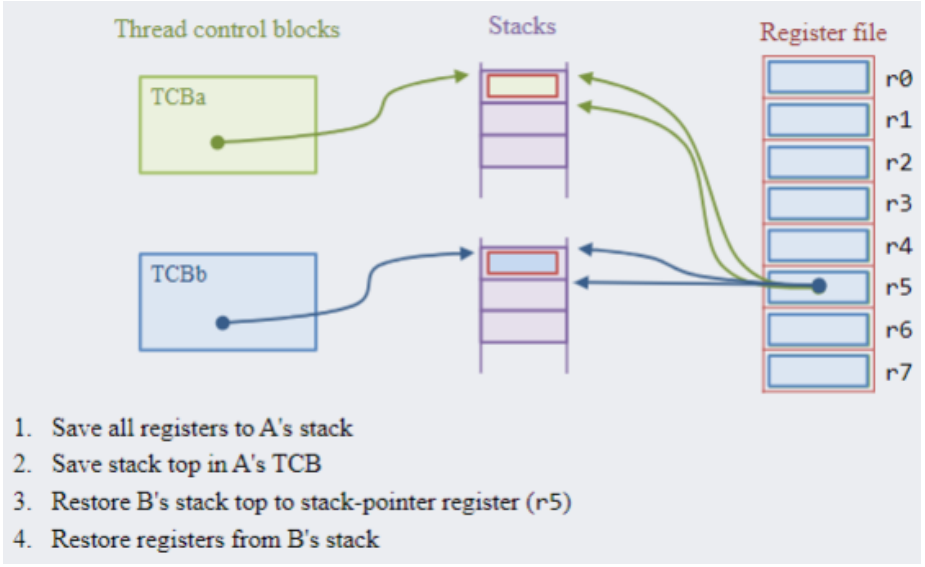
- 

Mar. 22nd - Unit 2b: Cont.d

- Implementing thread yield
    - Thread yield
        - puts current thread on ready queue
        - gets next runnable thread from ready queue
        - switches to next thread
    - ```
      void uthread_yield() {
          ready_queue_enqueue(uthread_self());
          uthread_t to_thread = ready_queue_dequeue();
          assert(to_thread);
          uthread_switch(to_thread, TS_RUNNABLE);
      }
      ```

- Implementing thread switch
    - Goal
        - Implement a procedure `switch`(T_a, T_b) that stops T_a and starts T_b
        - T_a calls `switch`, but it returns to T_b
    - Requires
        - saving T_a's processor state and setting processor state to T_b's saved state
        - state is just registers, which can be saved/restored to/from stack
        - thread-control block has pointer to stack pointer for each thread
    - Implementation
        - save all registers to stack
      ```

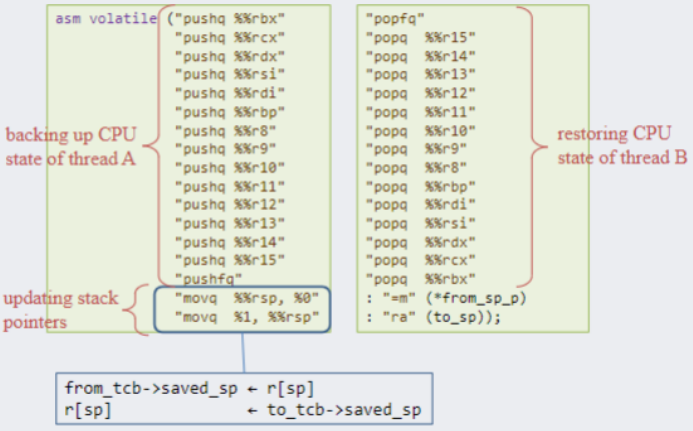
- save stack pointer to T_a's TCB
- set stack pointer to stack pointer T_b's TCB
- restore registers from stack

- 

1. Save all registers to A's stack
2. Save stack top in A's TCB
3. Restore B's stack top to stack-pointer register (r5)
4. Restore registers from B's stack

- Example code

- C / x86 assembly



- The `uthread_switch` procedure saves the ***from*** thread's registers to the stack, switches to the ***to*** thread's stack and restores its registers from the stack. But, what does it do with the program counter (PC)?

A. It saves the *from* thread's PC to the stack and restores the *to* thread's PC from the stack.
B. It saves the *from* thread's PC to its thread control block.
C. Nothing. It does not need to change the PC because the *from* and *to* threads' PCs are already saved on the stack before `switch` is called.
D. It jumps to the *to* thread's PC value.
E. ???

-
  - Correct Answer: C
  - Explanation:

```
// thread A
void ping() {
    // ...
    uthread_yield();    // needs to call to
other addresses
                        // return address
thus stored on activation
                        // frame
}

// thread B
void pong() {
    // ...
    uthread_yield();
}
```

- `yield`'s callee prologue save return address to `ping` on its activation frame. Then `yield` calls `switch`, A's stack is switched to B's stack; thus similarly, when B yields, the return address for `pong` is stored on its activation frame

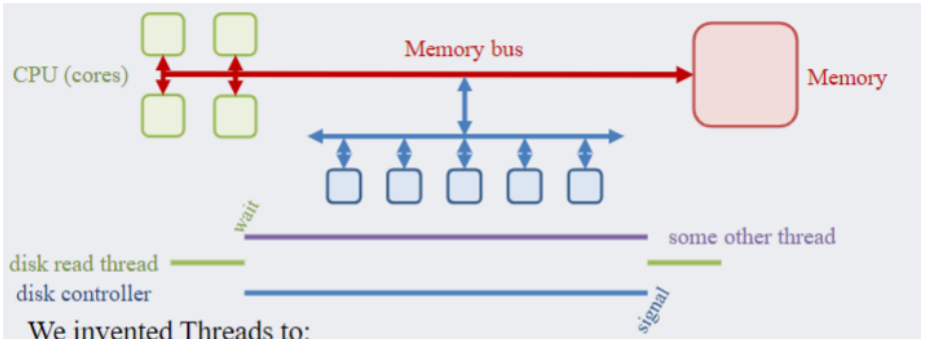
- Thread scheduling
    - Thread scheduling is
        - the process of deciding when threads should run
            - when there are more runnable threads than processors
            - involves a **policy** and a **mechanism**
    - Thread scheduling policy
        - the set of **rules** that determines which threads should be running
    - Things to consider
        - Do some threads have higher **priority** than others?
        - Should threads get **fair access** to the processor?
        - Should threads be guaranteed to **make progress**?
        - Should one thread be able to **pre-empt** another?
- Priority round-robin scheduling policy
    - Priority: number assigned to each thread
        - thread with highest priority goes first
    - When choosing the next thread to run
        - run the highest priority runnable thread
        - when the threads have the same priority, run thread that has waited the longest

- Implementation (mechanism)
    - organize Ready (runnable) Queue as a priority queue
        - highest priority first
        - FIFO among threads of equal **maximum** priority

- Pre-emption
    - This occurs when
        - a "yield" is forced upon the current running thread
        - current thread is stopped to allow another thread to run
    - Priority-based pre-emption
        - when a thread is made runnable (e.g. created or unblocked)
        - if it is higher priority than current-running thread, it pre-empts that thread
    - Quantum-based pre-emption (time-based)
        - each thread is assigned a runtime "quantum" (guaranteed minimum running time on CPU)
        - thread is pre-empted at the end of its quantum
        - How long should quantum be?
            - Disadvantage of too short? Thread may not make much progress
            - Disadvantage of too long? Other threads may result waiting for too long
            - Depends on OS configuration (typically ~100ms)

- Implementing quantum pre-emption
    - The problem
        - when application thread(s) are running, nothing is watching over them
        - for the system scheduler to control things, it needs a CPU to run on
        - if the application thread is running, the system is not
    - Solution: interrupts (timer device)
        - an IO controller connected to a clock/timer device
        - interrupts processor at regular intervals
    - Timer interrupt handler
        - compares the running time of the current thread to its quantum
        - pre-empts it if quantum has expired

Unit 2c: Synchronization

- Synchronization
    - 

    - We invented threads to: **express parallelism** - do things at the same time on different processors

      manage synchrony - do something else while waiting for IO controller
    - Two problems
        - Coordinating access to memory (variables) shared among multiple threads
        - control flow transfers among threads (wait until notified by another thread)
    - **Synchronization** is the mechanism threads use to
        - ensure **mutual exclusion** of critical sections
        - wait for and signal of the occurrence of events

Mar. 25th - Unit 2c: Cont.d

- Communicating through shared data
    - There will be problems
        - threads **share a data structure**
        - operations involve **multiple memory accesses**
        - these accesses can be **arbitrarily interleaved**
    - Example: a stack implemented as an array
        - ```
          int n;
          int array[SIZE];

          void push(i) {
              if (n < SIZE) {
                  array[n] = i;
                  n += 1;
              }
          }

          int pop() {
          ```
          ```

```
    if (n > 0) {
        n -= 1;
        return array[n];
    } else {
        return -1;
    }
}
```

- Suppose two threads accessing concurrently
- What happens if both push? both pop? 1 push, 1 pop?

- A linked-list stack
  -
```
struct SE {
    struct SE * next;
};
struct SE * top = NULL;

void push_st (struct SE * e) {
    e -> next = top;
    top = e;
}

void SE * pop_st() {
    struct SE * e = top;
    top = (top) ? top -> next : NULL;
    return e;
}
```

  - Sequential/synchronous execution of repeated push/pop is OK (not running with any threads)

```
void push_driver(long int n) {
    struct SE * e;
    while (n -= 1) {
        push(malloc(...));
    }
}

void pop_driver(long int n) {
    struct SE * e;
    while (n -= 1) {
        do {
            e = pop();
        } while (!e)
    }
    free(e);
}
```

```
push_driver(n);
pop_driver(n);
assert(top == NULL);
```

- but concurrent execution does not always work

```
int main(void) {
    et = uthread_create(push_driver, num);
    dt = uthread_create(pop_driver, num);

    uthread_join(et, 0);
    uthread_join(dt, 0);

    assert(top == NULL);
}
```

- works for `uthread_init(1)` **without** pre-emption, but not guaranteed to work for `uthread_init(2)`

- The problem
  - Same situation as the array implementation
    - push and pop are critical sections on the shared stack
    - parallel execution leads to potential arbitrary interleaving of operations
    - sometimes, the interleaving corrupts the data structure

- The importance of mutual exclusion
  - Shared data
    - data structure that could be accessed by multiple threads
    - typically, concurrent access to shared data is a bug
  - Critical sections
    - sections of code that access shared data
  - Race condition
    - simultaneous access to critical section by multiple threads (at least one updating)
    - conflicting operations on shared data structure are arbitrarily interleaved
    - unpredictable (non-deterministic) program behaviour - usually a (serious) bug
  - Mutual exclusion
    - a mechanism implemented in software (with some special hardware support)

        - to ensure critical sections of a shared data item are executed by only one thread at a time
        - reading and writing should be handled differently
- Mutual exclusion using locks
    - lock semantics
        - a lock is either **held** by a thread, or available
        - at most one thread can hold a lock at a time
        - a thread attempting to acquire a lock that is already held is forced to wait
    - lock operations (primitives)
        - `lock`: acquire lock, wait if necessary
        - `unlock`: release lock, allowing another thread to acquire if waiting
    - using locks on the shared stack

```
void push_cs(struct SE * e) {
    lock(&aLock);
        push_et(e);
    unlock(&aLock);
}

struct SE * pop_cs() {
    struct SE * e;
    lock(&aLock);
        e = pop_st();
    unlock(&aLock);
    return e;
}
```

    - Questions with locks
        - Thread A calls `push_cs` while Thread B is executing in `push_cs`
            - B holds the lock, A must wait until B releases the lock
        - Thread A calls `push_cs` while Thread B is executing in `pop_cs`
            - B holds the lock, A must wait until B releases the lock
        - What if `push_cs` and `pop_cs` use different locks
            - Suppose A acquires "red" lock, B acquires "blue" lock and may end up with interleaving
        - What if `push_cs` or `pop_cs` never call unlock
            - Waiting threads can never resume
- Implementing simple locks

- An initial attempt
    - use a shared global variable for synchronization
    - `lock`: loops until the variable is 0 and then sets it to 1
    - `unlock`: sets the variable to 0
- ```
  void lock(int * lock) {
      while (*lock == 1) {    }
      *lock = 1;
  }

  void unlock(int * lock) {
      *lock = 0;
  }
  ```

- Problem: There is a race in the lock code
    - Suppose two threads both request a lock at (nearly) the same time
    - ```
      // Thread A
      void lock(int * lock) {
          while (*lock == 1) {    }
          *lock = 1;
      }

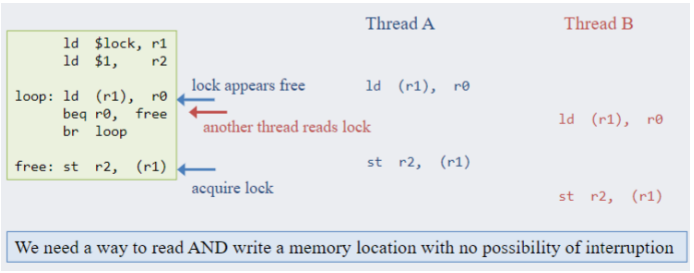
      // Thread B
      void lock(int * lock) {
          while (*lock == 1) {    }
          *lock = 1;
      }
      ```

    - Suppose the execution occurs as follows:

      ```
      1. read *lock == 0, exit loop
                                          2. read *lock == 0, exit loop

      3. *lock = 1
      4. return with lock held
                                          5. *lock = 1
                                          6. return with lock held
      ```

      Both threads think they hold the lock
    - The race exists even at the machine code level
        - two instructions are needed to acquire the lock
            - first, read to check that the lock is available
            - second, set the lock to held
        - but a read by another thread can check the lock before the first thread sets the hold

- 

- Atomic memory exchange
    - Atomicity
        - is a general property in systems
        - where a group of operations are performed as a single, indivisible unit
    - Atomic memory exchange
        - one type of atomic memory instruction
        - group a load and store together atomically
            - no intervening access from other threads
        - exchanging the value of a register and a memory location
    -

Name	Semantics	Assembly
atomic exchange	`r[v] ← m[r[a]]` `m[r[a]] ← r[v]`	`xchg (ra), rv`

    - Implementation
        - Cannot be implemented by CPU alone
            - must synchronize across multiple CPUs
                - accessing the same memory location at the same time
        - Implemented by memory bus
            - memory bus synchronizes every CPU's access to memory
            - the two parts of the exchange (read + write) are coupled on bus
            - bus ensures that no other memory transaction can intervene
            - Higher overhead, slower than normal load and store instructions

Mar. 27th - Unit 2c: Cont.d

- Spinlock
    - A lock where the waiter **spins**, looping on memory reads until lock is acquired, also called a **busy-waiting** lock
    - Implementation using atomic exchange

- Spin on atomic memory operation
    - That attempts to acquire lock while atomically reading its old value
    - ```
      ld $lock, r1        # get address of lock variable
      ld $1, r0           # r0 = 1
      loop:
          xchg (r1), r0
          beq r0, held    # if (r0 = 0), lock is held
          br loop         # else lock is held by others, spin to wait
      held:
          ...
      ```
- Speeding up spinlocks
    - Spin first on normal read
        - normal reads are fast and efficient compared to exchange
        - use normal read in loop until lock appears free
        - when lock appears free, use exchange to try to grab it
        - if exchange fails, then go back to normal read
    - ```
      ld $lock, r1            # r1 = &lock
      loop:   ld (r1), r0     # r0 = lock
              beq r0, try     # if (r0 == 0), has a chance
              br loop         # goto try
      try:    ld $1, r0       # r0 = 1
              xchg (r1), r0   # atomically swap r0 and lock
              beq r0, held    # goto held, if (r0 == 0), actually obtained
              br loop         # try again if another thread holds lock
      held:   ...
      ```
    - Busy-waiting pros and cons
        - Spinlocks are necessary and OK if spinner only waits a short time
        - But, using spinlock to wait for a long time wastes CPU cycles (if running on a single CPU, other threads may not be able to progress)
- Blocking locks

        - If a thread might wait for a long time, it should block so that other threads can run
        - It will then **unblock** when it becomes runnable, when either the lock is unlocked, or an event is signaled
        - Blocking locks for **mutual exclusion**
            - attempting to acquire a held lock blocks calling thread
                - blocked thread's TCB is stored on lock's waiting queue
            - when releasing lock, unblock a waiting thread if there is one
                - remove blocked thread from lock's waiting queue and place it on ready queue
        - Blocking for **event notification**
            - wait by blocking, placing TCB on a waiting queue for event
            - signal a specific waiting queue by moving a thread to ready queue
- Spinlocks vs. Blocking locks
    - Spinlocks
        - uncontended lock has low overhead, thread remains on CPU, execute loop a few times
        - waiting for lock has high overhead and wastes resources, thread still remains on CPU
        - used for short expected waits with minimal contention
        - used in implementation of blocking locks
    - Blocking locks
        - lock has fixed overhead, like thread switch, there is backup & restore CPU registers
        - if uncontended, overhead is higher than spinlocks
        - if contended, less waste of CPU resources
        - used when lock may be held for long time, or with high contention
- Monitors (Mutexes) and condition variables
    - MUTual EXclusion plus inter-thread synchronization
        - basic for synchronization primitives in Unix, Java
    - Monitor / Mutex
        - blocking lock to guarantee mutual exclusion
        - Basic operations: `lock` and `unlock` (sometimes called `enter` and `exit`)
    - Condition variable
        - allows threads to synchronize with each other

- `wait`: blocks until another thread signals on the variable
- `signal`: unblocks a waiting thread, if one exists
- `broadcast`: unblocks all threads currently waiting
- can only be accessed from inside of a monitor

```
struct uthread_mutex;
typedef struct uthread_mutex* uthread_mutex_t;
struct uthread_cond;
typedef struct uthread_cond*  uthread_cond_t;

uthread_mutex_t uthread_mutex_create        ();
void            uthread_mutex_lock          (uthread_mutex_t);
void            uthread_mutex_lock_readonly (uthread_mutex_t);
void            uthread_mutex_unlock        (uthread_mutex_t);
void            uthread_mutex_destroy       (uthread_mutex_t);

uthread_cond_t  uthread_cond_create         (uthread_mutex_t);
void            uthread_cond_wait           (uthread_cond_t);
void            uthread_cond_signal         (uthread_cond_t);
void            uthread_cond_broadcast      (uthread_cond_t);
void            uthread_cond_destroy        (uthread_cond_t);
```

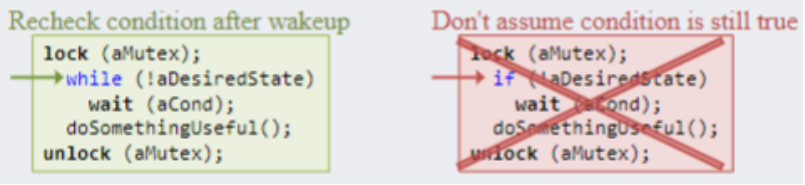
- Using conditions
  - Basic formulation
    - one thread acquires mutex and may wait for a condition to be established

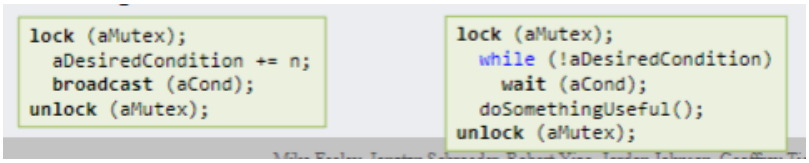
```
uthread_mutex_lock(aMutex);
    while (!aDesiredState) {
        uthread_cond_wait(aCond);
    }
    doSomethingUseful();
uthread_mutex_unlock(aMutex);
```

    - another thread acquires mutex, establishes condition and signals waiter, if there is one

```
uthread_mutex_lock(aMutex);
    aDesiredState = 1;
    uthread_cond_signal(aCond);
uthread_mutex_unblock(aMutex);
```

    - `wait` releases the mutex and blocks thread
      - before waiter blocks, it automatically releases mutex to allow other threads to acquire it
      - when wait unblocks, it re-acquires mutex, waiting/blocking to enter if necessary
      - note: other threads may have acquired mutex between wait call and return
  - `signal` awakens at most one thread

        - waiter does not run until signaler releases the mutex explicitly
        - a third thread could intervene and acquire mutex before waiter
        - waiter must then **re-check** wait condition
        - if no threads are waiting, then calling signal has no effect
        - 

    - `broadcast` awakens all waiting threads
        - may wake up too many
        - that's OK (but wasteful) since threads re-check wait condition and re-wait if necessary
        - 

Apr 3rd - Unit 2c: Cont.d

- Video Streaming with Threads
    - General problem
        - Video playback has 2 parts: fetch/decode, and playback
        - fetch has variable latency, and so we need a **buffer**
            - sometimes you can fetch faster than playback rate
            - but, other times there are long delays
            - buffer hides the delays by fetching ahead of playback position when possible
    - **Bounded buffers** and two independent threads
        - finite buffer of the next few video frames to play
        - maximum size is N frames
        - goal: keep buffer at least 50% full (or some other threshold of our choosing)
    - Producer thread
        - fetch frame from network and put them in buffer
    - Consumer thread
        - fetch frame from buffer, decode them and send them to video driver
    - How are Producer and Consumer connected
        - advantage of this approach is that they are largely decoupled; each has a separate job

    - but, it's the consumer that decides when the producer should run
- Step 1 - Request frames

```
struct video_frame;
#define N 100
struct video_frame buf[N];
int buf_length = 0;
int buf_pcur   = 0;
int buf_ccur   = 0;
```

```
uthread_mutex_t mx;
uthread_cond_t  need_frames;
```

```
void producer() {
  while (1) {
    uthread_lock (mx);
      while (buf_length < N) {
        buf [pcur] =  get_next_frame();
        buf_pcur   =  (pcur + 1) % N;
        buf_length += 1;
      }
      uthread_cond_wait (need_frames);
    uthread_unlock (mx);
  }
}
```

```
void consumer() {
  while (1) {
    uthread_lock (mx);
      assert (buf_length > 0);
      show_frame (buf [buf_ccur]);
      buf_ccur   =  (buf_ccur + 1) % N;
      buf_length -= 1;
      if (buf_length == N/2)
        uthread_cond_signal (need_frames);
    uthread_unlock (mx);
  }
}
```

    - Producer update buffer until full
    - If buffer drops below threshold, Consumer asks Producer to update more frames
- Step 2 - Delivering frames

```
void producer() {
  uthread_lock (mx);
    while (1) {
      while (buf_length < N) {
        buf [pcur] =  get_next_frame();
        buf_pcur   =  (pcur + 1) % N;
        buf_length += 1;
        uthread_cond_signal (have_frame);
      }
      uthread_cond_wait (need_frames);
    }
  uthread_unlock (mx);
}
```

```
void consumer() {
  uthread_lock (mx);
    while (1) {
      while (buf_length == 0)
        uthread_cond_wait (have_frame);
      show_frame (buf [buf_ccur]);
      buf_ccur   =  (buf_ccur + 1) % N;
      buf_length -= 1;
      if (buf_length < N/2)
        uthread_cond_signal (need_frames);
    }
  uthread_unlock (mx);
}
```

    - `uthread_cond_signal(have_frame)`: have at least one frame in the buffer
    - `while (buf_length == 0)`: while buffer is empty, stop and wait for signal
- Full version

```
void producer() {
  uthread_lock (mx);
    while (1) {
      while (buf_length < N) {
        buf [pcur] =  get_next_frame();
        buf_pcur   =  (pcur + 1) % N;
        buf_length += 1;
        uthread_cond_signal (have_frame);
      }
      uthread_cond_wait (need_frames);
    }
  uthread_unlock (mx);
}
```

```
void consumer() {
  uthread_lock (mx);
    while (1) {
      uthread_cond_wait (show_next_frame);
      while (buf_length == 0);
        uthread_cond_wait (have_frame);
      show_frame (buf [buf_ccur]);
      buf_ccur   =  (buf_ccur + 1) % N;
      buf_length -= 1;
      if (buf_length < N/2)
        uthread_cond_signal (need_frames);
    }
  uthread_unlock (mx);
}
```

    - `show_next_frame`: will be signaled every time a new frame is required for the video driver; every $\frac{1}{30}$s
    -

```
uthread_mutex_t mx;
uthread_cond_t  need_frames;
uthread_cond_t  have_frame;
uthread_cond_t  show_next_frame;
```

- Producer and Consumer Threads

- General Template

```
int canGoFlag = 1;
uthread_mutex_t mx;
uthread_cond_t canGoCond;
```

Producer (P)

```
uthread_mutex_lock(mx);
    uthread_cond_signal(canGoCond);
    canGoFlag = 1;
uthread_mutex_unlock(mx);
```

Consumer (C)

```
uthread_mutex_lock(mx);
    while (canGoFlag == 0)
        uthread_cond_wait(canGoCond);
    canGoFlag = 0;
uthread_mutex_unlock(mx);
```

- Key invariant: C does not complete until it is able to change `canGoFlag` from 0 to 1
    - if `canGoFlag` is 0, it waits for P to set it to 1
    - This waiting behavior is achieved by C waiting on (and P signaling) the same condition

```
int canGoFlag = 1;
uthread_mutex_t mx;
uthread_cond_t canGoCond;
```

```
void uthread_cond_wait (uthread_cond_t cond) {
  assert (cond->mutex->holder == uthread_self ());
  uthread_enqueue (&cond->waiter_queue, uthread_self());
  uthread_mutex_unlock (cond->mutex);
  uthread_block();
  uthread_mutex_lock (cond->mutex);
}
```

Producer (P)

```
uthread_mutex_lock(mx);
    uthread_cond_signal(canGoCond);
    canGoFlag = 1;
uthread_mutex_unlock(mx);
```

Consumer (C)

```
uthread_mutex_lock(mx);
    while (canGoFlag == 0)
        uthread_cond_wait(canGoCond);
    canGoFlag = 0;
uthread_mutex_unlock(mx);
```

What would happen if `cond_wait` didn't unlock and lock as it does?

A. nothing; it's fine either way
B. C might return when it shouldn't
C. P might not be able to signal it C correctly in some cases
D. P will never be able to successfully signal C under any circumstances
E. The instructor sees you playing games during the lecture

-
    - Correct Answer: D
    - Explanation: C → acquires mutex, checks condition (succeeds, execute `while` loop) → wait, block (while holding mutex); P → attempts to acquire mutex, fails (C holds) → unable to run, end up waiting and blocked

```
int canGoFlag = 1;
uthread_mutex_t mx;
uthread_cond_t canGoCond;
```

Producer (P)

```
uthread_mutex_lock(mx);
    uthread_cond_signal(canGoCond);
    canGoFlag = 1;
uthread_mutex_unlock(mx);
```

Consumer (C)

```
uthread_mutex_lock(mx);
    while (canGoFlag == 0)
        uthread_cond_wait(canGoCond);
    canGoFlag = 0;
uthread_mutex_unlock(mx);
```

Why not this? (CX)

```
uthread_mutex_lock(mx);
    if (canGoFlag == 0)
        uthread_cond_wait(canGoCond);
    canGoFlag = 0;
uthread_mutex_unlock(mx);
```

Suppose P and CX are the only threads accessing the mutex. Which statement is correct?

A. CX always works
B. CX never works
C. CX works if and only if there is a single P thread
D. CX works if and only if there is a single CX thread
E. Churros are tasty

-
    - Correct Answer: D
    - Explanation:

      CX → acquires mutex, check condition, executes `while` loop → wait, release mutex

      P1 → acquire mutex, set condition, signal → release mutex

      CX → attempt to acquire mutex, fails, wait, go back to sleep

      P2 → acquires mutex, set condition, signal → release mutex

CX → wake up, proceed as if condition passes → set condition to 0 → release mutex

- In D, shared data may become corrupted if there are multiple (CX) consumer threads

  CX1 → acquire mutex, check condition, wait on condition → release mutex

  P → acquire mutex, set condition, signal condition → release mutex

  CX1 and CX2 may have a race to acquire the mutex; assume CX2 obtain mutex first

  CX1 → attempt to acquire mutex (failed), block, on mutex queue

  CX2 → acquires mutex, check condition, set `canGoFlag` to 0 → release mutex

  CX1 → acquires mutex, complete `wait`, proceeds **as if** the `if` condition is true, set `canGoFlag` = 0 → release mutex

Apr. 5th - Unit 2c: Cont.d

- Beer for everyone
  - Beer pitcher is a shared data structure with these operations
    - `pour()`: pours one glass from pitcher (reduces beer left by 1)
    - `refill()`: adds more beer to pitcher (increases beer left by N)
  - Implementation goal
    - synchronize access to pitcher
    - pouring from empty pitcher requires waiting for it to be filled
    - filling pitcher releases waiting threads
  - We will use
    - `glasses`: amount of beer left in the pitcher, in glasses (`int`)
    - `mx`: mutex
    - `hasBeer`: condition indicating there is at least one beer left
  - Setup

```
struct BeerPitcher {
    int glasses;
    uthread_mutex_t mx;
    uthread_cond_t hasBeer;
};

struct BeerPitcher* createAndInitializePitcher() {
    struct BeerPitcher* p = malloc(sizeof(struct
BeerPitcher));
```

```
    p -> glasses = 0;
    p -> mx = uthread_mutex_create();
    p -> hasBeer = uthread_cond_create(p -> mx);
    return p;
}

// Pouring a glass
void pour(struct BeerPitcher* p) {
    uthread_mutex_lock(p -> mx);
        while (p -> glasses == 0) {
            // similar to streaming: block playback
when buffer is empty
            uthread_cond_wait(p -> hasBeer);
        }
        p -> glasses = 1;
    uthread_mutex_unlock(p -> mx);
}

// Refilling the pitcher (pitcher has unlimited
capacity)
void refill(struct BeerPitcher* p, int n) {
    uthread_mutex_lock(p -> mx);
        p -> glasses += n;
        for(int i = 0; i < n; i += 1) {
            uthread_cond_signal(p -> hasBeer);
            // or uthread_cond_broadcast, if most
waiters will be woken up
        }
    uthread_mutex_unlock(p -> mx);
}
```

- Signal and Mutex Race

  Thread A: pour; Thread B: refill; Thread C: pour

  Start with `glasses == 0`

  A → acquires mutex, checks condition → waits, release mutex

  B → acquires mutex, sets `glasses += 1`, signals condition → releases mutex

  A and C race to get mutex, A fails

  C → acquires mutex, sets `glasses -= 1` → releases mutex

  A → acquires mutex, checks condition → waits, releases mutex

- What if we want to refill automatically?
  - a pitcher has capacity `maxGlasses` and current volume `glasses`
  - pouring removes one glass if there is enough beer, and waits otherwise

- refilling loops forever, waiting for pitcher to be empty, and when it is, it refills the pitcher to its full capacity and awakens any pourer
- ```
  // add uthread_cond_t isEmpty =
  uthread_cond_create(p -> mx);
  ```
- ```
  void pour(...) {
      ...
      if (p -> glasses == 0) {
          // similar to streaming: buffer is
  empty, make Producer get more frames
          uthread_cond_signal(p -> isEmpty);
      }
  }

  void refill(...) {
      ...
      while(1) {
          while(p -> glasses > 0) {
              uthread_cond_wait(p -> isEmpty);
          }

          // similar to streaming: fill buffer,
  signal Consumers that frames available
          p -> glasses += p -> maxGlasses;
          for (int i = 0; i < maxGlasses; i +=
  1) {
              uthread_cond_signal(p ->
  hasBeer);
          }
      }
      ...
  }
  ```

- Back to disk reads
  - Suppose we write an async disk read, and we want to block/wait with conditions

```
void read(char * buf, int bytes, int blockno) {
    // some thread can intervene here, ISR may occur
before read thread resumes
    scheduleRead(buf, nbytes, blockno);

    uthread_mutex_lock(mx);
        uthread_cond_wait(disk_op_complete);
    uthread_mutex_unlock(mx);
}
```

```
void readCompleteCalledByISR() {
    uthread_mutex_lock(mx);
        uthread_signal(disk_op_complete);
    uthread_mutex_unlock(mx);
}
```

- Wait-signal problem
    - wait condition check / trigger and wait are not atomic
    - signal could occur before wait, thus waiter could miss signal and never wake up
- Solution
    - Ensure that condition check / trigger and wait are atomic
    - So that wait is ordered before signal

- Naked notify
    - Naked signal / notify
        - A signal called outside of a monitor
        - Should be avoided (can cause wait-signal race)
            - signal usually needed due to modifying shared data, if outside of mutex, shared data can also be modified outside
    - Sometimes, it is necessary
        - when blocking is not allowed
        - e.g. in some asynchronous event handlers (IO operations are slow)
            - if an IO controller holds a mutex, then CPU must wait a very long time to acquire mutex

- Reader-writer monitors
    - Critical sections could be classified as
        - `reader`: if only reads the shared data
        - `writer`: if it updates the shared data
        - we can weaken the mutual exclusion constraint
            - writer requires an **exclusive access** to the monitor
            - but a group of readers can **access monitor concurrently**
    - Read-write monitors
        - monitor states: **free**, **held for reading**, **held for writing**
        - If held for reading, multiple readers can access simultaneously
            - we will need to know how many readers are accessing, and when they are done
    - Operations

        - `mutex_lock()`: lock for writing
            - only acquires lock if it is free
            - sets state to **held for writing**
        - `mutex_lock_read_only()`:
            - if lock is free, set its state to **held for reading**
            - if lock already held for reading, allow a new reader
            - increments a reader count
        - `mutex_unlock()`:
            - if **held for writing**, set state to **free**
            - if **held for reading**, decrement reader count until zero, then set to **free**
    - Fair access to reader-writer block
        - Policy question
            - if monitor state is held for reading and
                - A calls `mutex_lock()`, and blocks while waiting for monitor to be free
                - B calls `mutex_lock_read_only()`
        - Option 1 - Disallow new readers while a writer is waiting
            - affects a thread that could be running but now must wait
            - provides fair access to monitor (writer may have been waiting loger)
        - Option 2 - Allow new readers while a writer is waiting
            - increases concurrency, allows more threads to run
            - writer may need to wait for a long time to get access (starvation)
        - Solution
            - tradeoffs, depend on situation and application

Apr. 8th - Unit 2c: Cont.d

- Semaphores - Edsger Dijkstra (THE system)
    - A non-negative atomic counter
        - any attempt to make counter negative will block the calling thread
        - no operation to read value; only to change it
    - `P(s)`: wait, dec, sub
        - From Dutch: *prober te verlagen* - "try lowering"
        - atomically blocks until $s > 0$, then decrements s

        - will be automatically woken up for resumption when $s > 0$
    - `V(s)`: signal, inc, add
        - From Dutch: *verhogen* - "to increase"
        - atomically increase s, and unblock threads waiting in P appropriate
    - UThread semaphores

```
struct uthread_sem;
typedef struct uthread_sem * uthread_sem_t;

uthread_sem_t   uthread_sem_create(int
initial_value);
void            uthread_sem_destroy(uthread_sem_t);
void            uthread_sem_wait(uthread_sem_t);
void            uthread_sem_signal(uthread_sem_t);
```

- Drinking beer with semaphores
    - Use semaphore to store number of glasses held by pitcher
        - set initial value to empty (zero) when creating it

```
uthread_sem_t glasses =
uthread_sem_create(0);
```

    - Pour and Refill no longer require a monitor
        - since the semaphore atomically changes the counter already

```
void pour(uthread_sem_t glasses) {
    uthread_sem_wait(glasses);
}

void refill(uthread_sem_t glasses, int n) {
    for (int i = 0; i < n; i += 1) {
        uthread_sem_signal(glasses);
    }
}
```

- Implementing monitors using semaphores
    - Implementing a mutex using semaphores
        - create semaphore with initial value 1 (free, representing an available mutex)
            - range of semaphore should be $[0, 1]$
        - lock is `P(), wait()`
        - unlock is `V(), signal()`
    - Implementing condition variables

        - Difficult
            - In condition variables, signal without wait will be ignored / no effect
            - In semaphores, signal can unlock a future wait
    - Example: disk read

```
// Asynchronous read request
void read(char* buf, int nbytes, int blockno) {
    scheduleRead(buf, nbytes, blockno);
    // what if thread gets pre-empted here?
    // even if signal is signaled before, the
following wait() can still execute as intended

    // semaphore is readComplete, initialized to 0
    // thread for read becomes blocked
    uthread_sem_wait(readComplete);
    // handle data
}

// Read completion (called by disk ISR)
void onReadComplete() {
    // increase readComplete and wake up a waiter
    uthread_sem_signal(readComplete);
}
```

        - No critical section, no wait-signal race problem
- Blocking queue using semaphores
    - Semaphores:
        - `mx` - starts at 1 (like a mutex, range $[0, 1]$)
        - `length` - starts at 0
    - `lock` $\iff$ `wait`

      `unlock` $\iff$ `signal`
    -

```
void enqueue (queue_t* queue, item_t item) {
  uthread_sem_wait (queue->mx);
  item->next = NULL;
  if (queue->tail)
    queue->tail->next = item;
  queue->tail = item;
  if (queue->head == NULL)
    queue->head = queue->tail;
  uthread_sem_signal (queue->mx);
  uthread_sem_signal (queue->length);
}
```

-
```
item_t dequeue (queue_t* queue) {
  // locks if queue is empty
  item_t item = NULL;
  uthread_sem_wait (queue->length);
  uthread_sem_wait (queue->mx);
  item = queue->head;
  queue->head = queue->head->next;
  if (queue->head) == NULL)
    queue->tail = NULL;
  item->next = 0;
  uthread_sem_signal (queue->mx);
  return item;
}
```
    - Why are there no loops?
        - "State variable" is a semaphore, read/write occurs atomically, no chance of state being changed since waking up no need to re-check state
    - Why wait on length outside critical section?
        - avoid deadlock
    - Why signal on length outside critical section?
        - It's fine, dequeue go back to sleep briefly on `mx`

Apr. 10th - Unit 2c: Cont.d

- Ordering threads using semaphores
    - If thread B must wait for thread A to finish
        - Initialize semaphore `b` to 0, i.e. `uthread_sem_t b = uthread_sem_create(0)`
        - Thread A: `uthread_sem_signal(b)`
        - Thread B: `uthread_sem_wait(b)`
    - If both threads need to wait for each other (rendezvous)
        - Initialize two semaphores `a, b` with 0
        -
```
// Thread A
uthread_sem_signal(a);
uthread_sem_wait(b);

// Thread B
uthread_sem_signal(b);
uthread_sem_wait(a);
```
        - If the order of wait and signal are reversed for either (or both) threads
            - If reversed for A, both threads run fine
            - If reverse both of them, then deadlock
- Barriers using semaphores

- Local, non-reusable barrier
    - Like thread join, but for any thread rather than a specific one
    - Create semaphore with initial value 0
    - Create N threads, each thread calls signal on the semaphore
    - Main thread can wait for any created thread to finish
    - To wait for all of them, just wait N times
    - Can also be used to wait for some milestone
- Analogy: job application with reference contact information, potential employer receives application and contacts references directly, but will not process application until hearing back from 2 references

- Problems with concurrency
    - Race condition
        - Competing, unsynchronized access to shared variable
            - from multiple threads
            - at least one of the threads is attempting to update the variable
        - Solved with synchronization
            - guaranteeing mutual exclusion for competing access
            - but the language does not help you see what data might be shared - can be very hard
    - Deadlock
        - multiple competing actions wait for each other
        - all actions are prevented from completing
    - Livelock
        - Threads respond to actions by other threads, but other threads also respond
        - Threads are unable to make progress
        - ```
          while (true) {
              lock1.lock();
              if (!lock2.tryLock()) {
                  lock1.unlock();
                  lock2.lock();
                  if (!lock1.tryLock()) {
                      lock2.unlock();
                      continue;
                  }
              }
              break;
          }
          ```
          ```

        - Starvation
            - A thread is unable to gain regular access to a shared resource
            - "Greedy" threads may lock resource for a long time
            - Threads with higher priority may skip ahead
            - Reader threads may acquire reader-writer lock in front of writer
- Systems with multiple mutexes
    - Consider a system with 2 mutexes: `a, b`
    - ```
      void foo() {
          uthread_mutex_lock(a);
          uthread_mutex_unlock(a);
      }

      void bar() {
          uthread_mutex_lock(b);
          uthread_mutex_unlock(b);
      }

      void x() {
          uthread_mutex_lock(a);
              bar();
          uthread_mutex_unlock(a);
      }

      void y() {
          uthread_mutex_lock(b);
              foo();
          uthread_mutex_lock(b);
      }
      ```
        - If `x()` calls `foo()`: thread becomes blocked by itself
        - With `y()`: threads for `x, y`, if each acquire a/b before calling bar/foo → deadlock
- Concurrency problems with recursion
    - If a recursive call attempts to acquire a lock
        - Lock is already being held, so thread blocks
        - Thread is waiting for lock to be released
        - But that same thread is already holding the lock
    - If we release lock before calling recursively
        - Could break critical section's problem
    - Solution: Re-entrant mutex
      ```

        - Allows a lock to be acquired more than once (only if it is acquired again by the same thread)
        - Unlock only releases the lock if called as many times as lock was called
        - Each lock operation (from the same thread) increments counter
        - Unlock decrements it, and releases lock when counter is 0
- Identifying deadlocks with waiter graphs
    - Waiter graph
        - Vertices for threads and locks
        - edge from lock to thread, if lock is **HELD** by thread
        - edge from thread to lock, if lock is **WAITING** for lock
        - a cycle indicates deadlock
- The dining philosophers problem
    - 5 computers competed for access to 5 shared tape drivers
    - Description
        - 5 philosophers sit at a round table with one fork placed in between each
        - Each philosopher is either eating, or thinking
            - not doing two things simultaneously
            - they never speak to each other
        - A large bowl of spaghetti in the middle of the table requires 2 forks to serve
    - Deadlock
        - Assume that philosophers always start with the left fork
        - Also assume that all philosophers decide to start eating at the same time
            - Everyone is able to get left fork
            - But everyone waits for right fork
    - Livelock
        - Assume that, if philosophers can't get second fork, they release the first fork, then wait on second
        - If the process is repeated, and all philosophers are synchronized
            - Philosophers will repeatedly get one fork at a time
            - All are busy, but unable to eat
- Avoiding deadlock
    - If not necessary, don't use multiple threads
        - will have many idle CPU cores and write asynchronous code
    - If not necessary, don't use shared variables

        - Use local variables and parameters whenever possible
        - if threads don't access shared data, no need for synchronization
- Avoid unnecessary locks
        - When possible, use atomic data structures and lock-free synchronization
        - If possible, use only one lock at a time
- Evaluate the scope of the lock
        - can possibly avoid deadlock by reducing portion of code that needs lock?
- Organize locks into precedence hierarchy
        - each lock is assigned a unique precedence number
        - before thread X acquires a lock i, it must hold all higher precedence locks
        - ensures that any thread holding lock i can not be waiting for X
            - so that locks are always acquired in the same order
- Limit wait time on locks
        - Provide an alternate action if lock cannot be acquired
- Detect and destroy
        - If cannot avoid deadlock, detect when it has occurred
        - break deadlock by interrupting/terminating threads