CPSC 213

Introduction to Computer Systems

Unit 3

Course Review

Learning Goals 1

- Memory
  - Endianness and memory-address alignment
- Globals
  - Machine model for access to global variables; static and dynamic arrays and structs
- Pointers
  - Pointers in C, & and * operators, and pointer arithmetic
- Instance Variables
  - Instance variables of objects and structs
- Dynamic Storage
  - Dynamic storage allocation and deallocation
- If and Loop
  - If statements and loops
- Procedures
  - Procedures, call, return, stacks, local variables and arguments
- Dynamic Flow Control
  - Dynamic flow control, polymorphism, and switch statements

Learning Goals 2

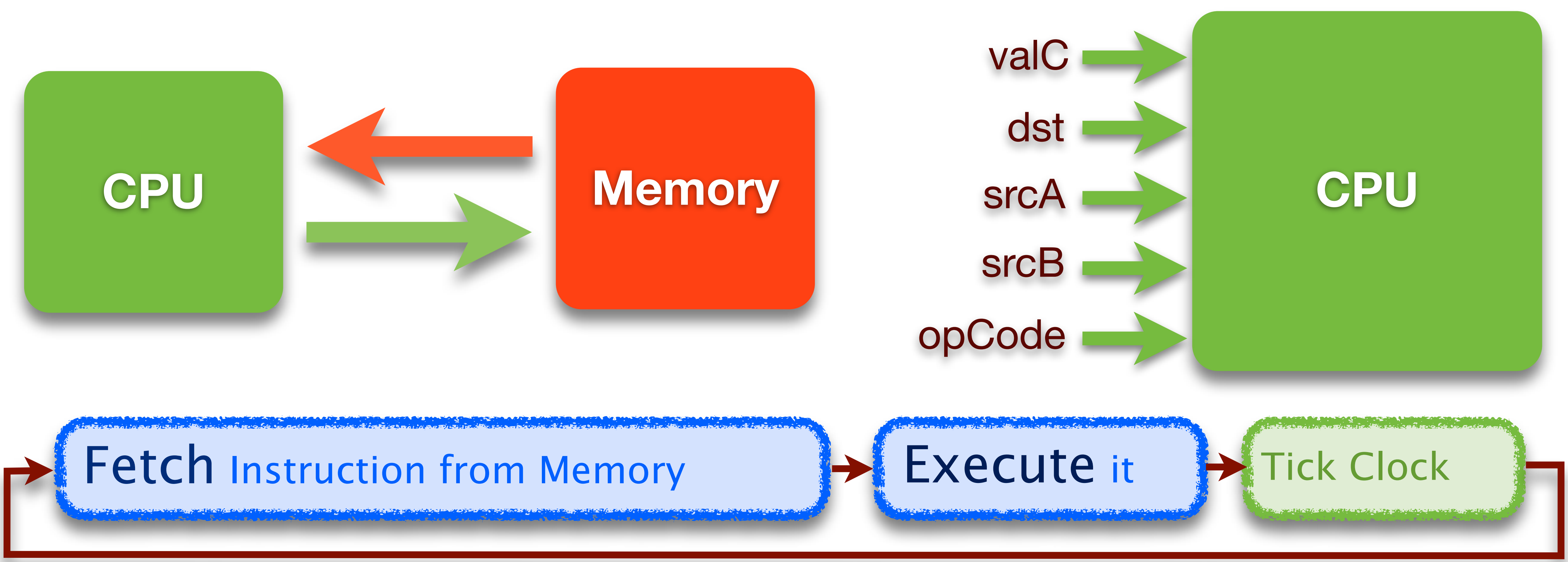
- Read Assembly
  - Read assembly code
- Write Assembly
  - Write assembly code
- ISA-PL Connection
  - Connection between ISA and high-level programming language
- Asynchrony
  - PIO, DMA, interrupts and asynchronous programming
- Threads
  - Using and implementing threads
- Synchronization
  - Using and implementing spinlocks, monitors, condition variables and semaphores
- Virtual Memory
  - Virtual memory translation and implementation tradeoffs

Not Covered on Final

- Details of memory management
  - Java weak references, reference objects, reference queues
    - slides 22-24 of module 1c, details of Lab 3 Java memory leak solution
  - C reference counting
    - slides 17-18 of module 1c
- Details of Hoare blocking signal for condition variables
  - slides 24-26 of module 2c
- OS/Encapsulation
  - module 2e
- Interprocess Communication, Networking, Protocols
  - module 2f

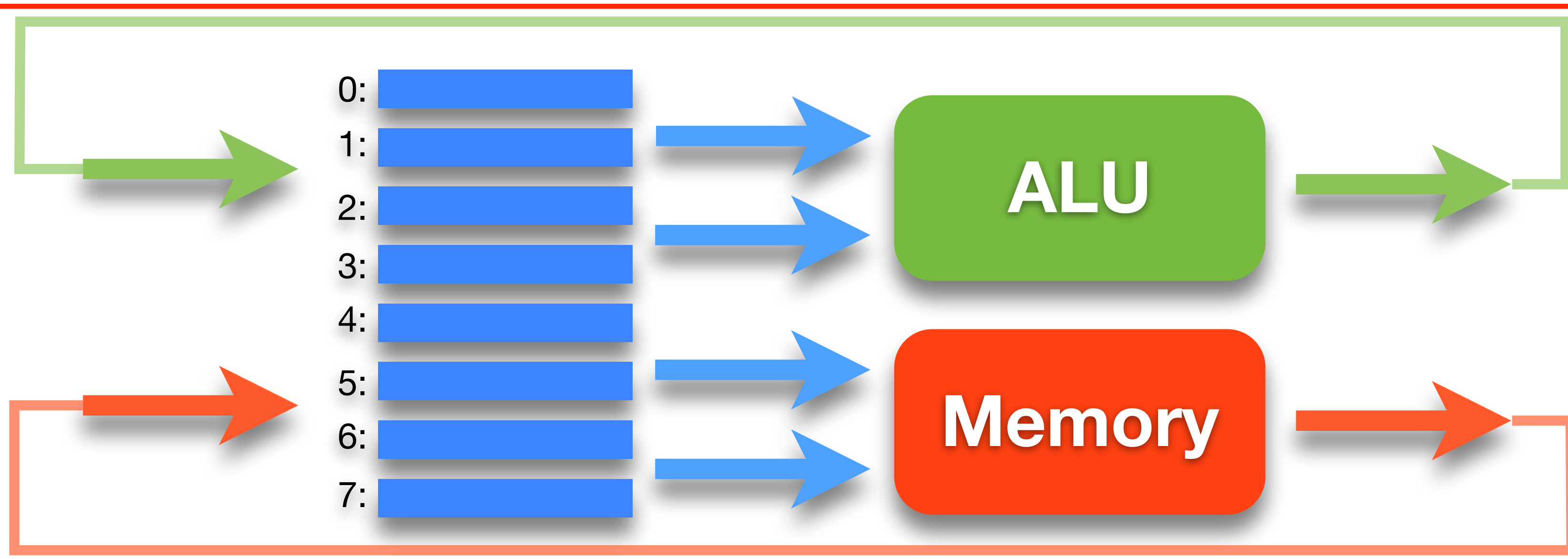
Big Ideas: First Half

- Static and dynamic
  - anything that can be determined **before execution** (by compiler) is called ***static***
  - anything that can only be determined **during execution** (at runtime) is called ***dynamic***
- SM-213 Instruction Set Architecture
  - hardware context is CPU and main memory with fetch/execute loop



Memory Access

- Memory is
  - an array of bytes, indexed by byte ***address***
- Memory access is
  - restricted to a transfer between registers and memory
  - the ALU is thus unchanged, it still takes operands from registers
  - *this is approach taken by Reduced Instruction Set Computers (RISC)*
- Common mistakes
  - wrong: trying to have instruction read from memory and do computation all at once
    - must always load from memory into register as first step, then do ALU computations from registers only
  - wrong: trying to have instruction do computation and store into memory all at once
    - all ALU operations write to a register, then can store into memory on next step



Loading and Storing

- load into register
  - immediate value: 32-bit number directly inside instruction
  - from memory: base in register, direct offset as 4-bit number
    - offset/4 stored in machine language
    - common mistake: forget 0 offset when just want store value from register into memory
  - from memory: base in register, index in register
    - computed offset is 4*index
  - from register
- store into memory
  - base in register, direct offset as 4-bit number
  - base in register, index in register
  - common mistake: cannot directly store immediate value into memory

Name	Semantics	Assembly	Machine
load immediate	r[d] ← v	ld $v, rd	0d-- vvvvvvvv
load base+offset	r[d] ← m[r[s]+(o=p*4)]	ld o(rs), rd	1psd
load indexed	r[d] ← m[r[s]+4*r[i]]	ld (rs,ri,4), rd	2sid
register move	r[d] ← r[s]	mov rs, rd	60sd
store base+offset	m[r[d]+(o=p*4)] ← r[s]	st rs, o(rd)	3spd
store indexed	m[r[d]+4*r[i]] ← r[s]	st rs, (rd,ri,4)	4sdi

Numbers

- Hex vs. decimal vs. binary
  - in SM-213 assembly
    - 0x in front of number means it's in hex
    - otherwise it's decimal
  - converting from hex to decimal
    - convert each hex digit separately to decimal
    - $0x2a3 = 2x16^2 + 10x16^1 + 3x16^0$
  - converting from hex to binary
    - convert each hex digit separately to binary: 4 bits in one hex digit
  - converting from binary to hex
    - convert each 4-bit block to hex digit
  - exam advice
    - reconstruct your own lookup table in the margin if you need to do this

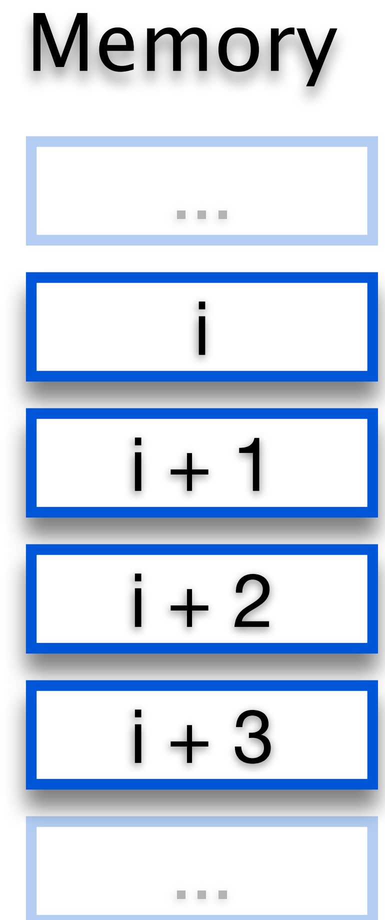
dec	hex	bin
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Numbers

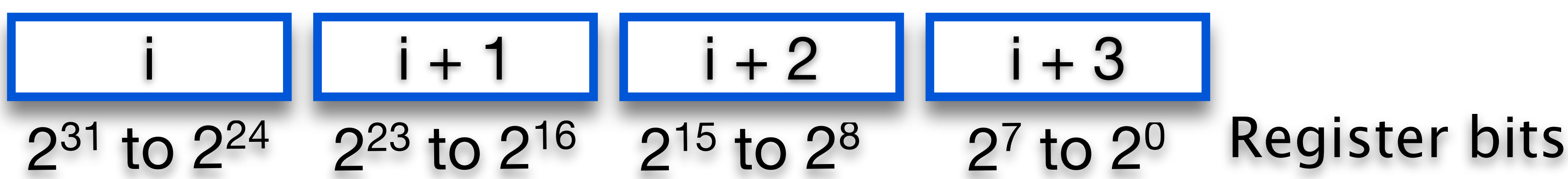
Common mistakes

- treating hex number as decimal: interpret 0x20 as 20, but it's actually decimal 32
- using decimal number instead of hex: writing 0x20 when you meant decimal 20
- wasting your time converting into format you don't particularly need
- wasting your time trying to do computations in unhelpful format
  - think: what do you really need to answer the question?
  - adding small numbers easy in hex: B+2=D
  - for serious computations consider converting to decimal
  - unless multiply/divide by power of 2: then hex or binary is fast with bitshifting!

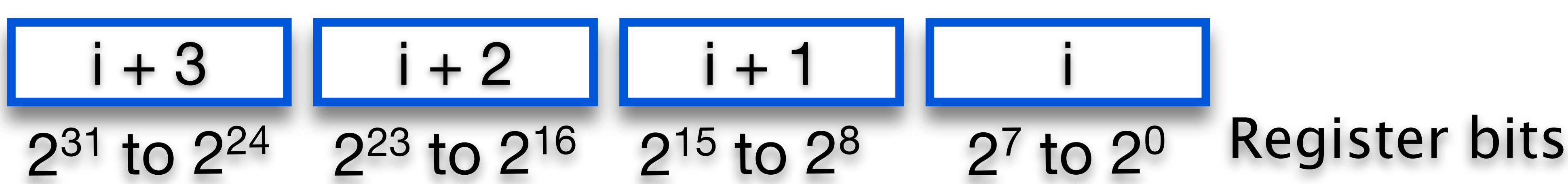
Endianness



- Consider 4-byte memory word and 32-bit register
  - it has memory addresses i, i+1, i+2, and i+3
  - we'll just say its "***at address i and is 4 bytes long***"
  - e.g., the word at address 4 is in bytes 4, 5, 6 and 7.
- Big or Little Endian
  - we could start with the BIG END of the number
    - most computer makers except for Intel, also network protocols



  - or we could start with the LITTLE END
    - Intel



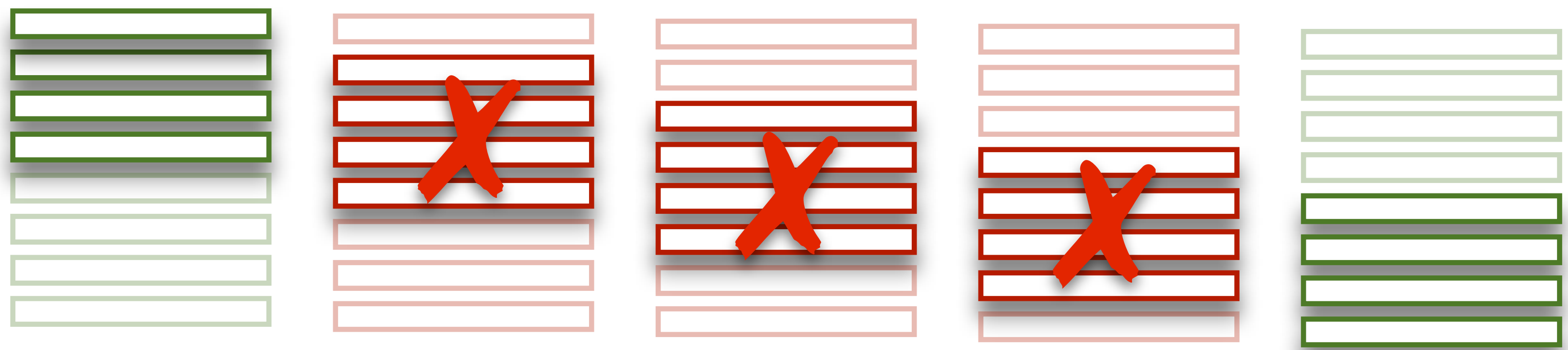
Determining Endianness of a Computer

```c
#include <stdio.h>

int main () {
  char a[4];

  *((int*)a) = 1;

  printf("a[0]=%d a[1]=%d a[2]=%d a[3]=%d\n",a[0],a[1],a[2],a[3]);
}
```

- how does this C code check for endianness?
  - create array of 4 bytes (char data type is 1 byte)
  - cast whole thing to an integer, set it to 1
  - check if the 1 appears in first byte or last byte
- things to understand:
  - concepts of endiananess
  - casting between arrays of bytes and integers
  - masking bits, shifting bits

Alignment

- Power-of-two aligned addresses simplify hardware
  - required on many machines, faster on all machines



  - computing alignment: for what size integers is address X aligned?
    - byte address to integer address is division by power to two, which is just shifting bits

$$j \,/\, 2^k == j >> k \qquad \text{(j shifted k bits to right)}$$

    - convert address to decimal; divide by 2, 4, 8, 16, ...; stop as soon as there's a remainder
    - convert address to binary; sweep from right to left, stop when find a 1

Static Variable Access (static arrays)

```
int a;
int b[10];

void foo () {
  ....
  b[a] = a;
}
```

b[a] = a;

Static Memory Layout

0x1000: value of a
0x2000: value of b[0]
0x2004: value of b[1]
...
0x2020: value of b[9]

- Key observations
  - address of **b[a]** cannot be computed statically by compiler
  - address can be computed dynamically from base and index stored in registers
    - element size can known statically, from array type
- Array access: use load/store indexed instruction

Name	Semantics	Assembly	Machine
load indexed	r[d] ← m[r[s]+4*r[i]]	ld (rs,ri,4), rd	**2sid**
store indexed	m[r[d]+4*r[i]] ← r[s]	st rs, (rd,ri,4)	**4sdi**

Static vs Dynamic Arrays

- Same access, different declaration and allocation
  - for static arrays, the compiler allocates the whole array
  - for dynamic arrays, the compiler allocates a pointer

```
int a;
int b[10];

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

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

```
ld $a_data, r0   # r0 = address of a
ld (r0), r1      # r1 = a
ld $b_data, r2   # r2 = address of b
st r1, (r2,r1,4) # b[a] = a
```

```
int  a;
int* b;

void foo () {
  b = (int*) malloc (10*sizeof(int));
  b[a] = a;
}
```

```
0x2000: value of b
```

```
ld $a_data, r0   # r0 = address of a
ld (r0), r1      # r1 = a
ld $b_data, r2   # r2 = address of b
ld (r2), r3      # r3 = b
st r1, (r3,r1,4) # b[a] = a
```

extra dereference

Dereferencing Registers

- Common mistakes
  - no dereference when you need it
  - extra dereference when you don't need it
  - example

```
ld $a_data, r0   # r0 = address of a
ld (r0), r1      # r1 = a
ld $b_data, r2   # r2 = address of b
ld (r2), r3      # r3 = b
st r1, (r3,r1,4) # b[a] = a
```

    - a dereferenced once
    - b dereferenced twice
      - once with offset load
      - once with indexed store
  - no dereference: value in register
  - one dereference: address in register
  - two dereferences: address of pointer in register

Basic ALU Operations

- 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[d]	not rd	67-d

- Shifting, NOP and Halt

Name	Semantics	Assembly	Machine
shift left	r[d] ← r[d] << S = s	shl rd, s	7dSS
shift right	r[d] ← r[d] << S = -s	shr rd, s	
halt	*halt machine*	halt	f0--
nop	*do nothing*	nop	ff--

Pointers

- Notation
  - & X the address of X
  - * X the value X points to
    - we also call this operation *dereferencing*

```
int  a;
int* b;

void foo () {
  a = 3;
  *b = 4;
}
```

address of a	0x1000:	3	value of a
address of b	0x2000:	0x3000	value of b
address of *b	0x3000:	4	value of *b

  - &a = 0x1000, a = 3, *a = (whatever is at address 0x3...)
  - &b = 0x2000, b = 0x3000, *b = 4
  - common mistakes
    - use address of pointer
    - try to dereference integer storing value

Pointer Arithmetic in C

- Alternative to a[i] notation for dynamic array access
  - a[x] equivalent to *(a+x)
  - &a[x] equivalent to (a+x)
- Pointer arithmetic takes into account size of datatype
  - `int a[4];`

    0x2000: value of a[0]
    0x2004: value of a[1]
    0x2008: value of a[2]
    0x200a: value of a[3]

    - &a[0] = 0x2004; &a[2] = 0x2008
    - (& a[2]) - (& a[1])) == 1 == (a+2) - (a+1)
  - compiler treats pointer-to-int differently than int!
    - even though both can be stored with 32 bits on IA-32 machine
- Common mistake
  - treat pointer arithmetic like direct calculations with addresses
    - off by 4 when doing pointer arithmetic with integers

Pointer Arithmetic Example Program

- Exam studying advice
  - try writing simple test programs, use gdb and print to explore

```
tmm% cat array2.c
#include <stdio.h>
int main (int argc, char** argv) {
 int a[4] = {100, 110, 120, 130};
 int k = &a[4];
 int m = &a[1];
 int n = k-m;
 int o = &a[4]-&a[1];
 printf ("k hex: %x, k dec: %d, m hex: %x, m dec %d, n: %d, o: %d \n",k, k, m, m, n, o);
}

tmm% gcc -g -o array2 array2.c
array2.c: In function 'main':
array2.c:6: warning: initialization makes integer from pointer without a cast
array2.c:7: warning: initialization makes integer from pointer without a cast

tmm% ./array2
k hex: bffff7d0, k dec: -1073743920, m hex: bffff7c4, m dec -1073743932, n: 12, o: 3

tmm% gdb array2
(gdb) p &a[4]
$1 = (int *) 0xbffff510
(gdb) p k
$2 = -1073744624
```

Summary: Static Scalar and Array Variables

- Static variables
  - the compiler knows the address (memory location) of variable
- Static scalars and arrays
  - the compiler knows the address of the scalar value or array
- Dynamic arrays
  - the compiler does not know the address the array
- What C does that Java doesn't
  - static arrays
  - arrays can be accessed using pointer dereferencing operator
  - arithmetic on pointers
- What Java does that C doesn't
  - typesafe dynamic allocation
  - automatic array-bounds checking

Structs

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

```
struct D  d0;
```

address of d0

(also)

address of d0.e	0x1000: value of d0.e
address of d0.f	0x1004: value of d0.f
address of d0.g	0x100c: value of d0.g

Key observation

- offset from base of struct to a specific field is static
  - can always be computed by compiler
- address can be computed dynamically from base stored in register and offset computed by compiler and encoded directly into instruction
  - difference from arrays: fields do not all have to be same size, so cannot necessarily compute offset from index

Struct access: use load/store offset instruction

Name	Semantics	Assembly	Machine
load base+offset	r[d] ← m[r[s]+(o=p*4)]	ld o(rs), rd	1psd
store base+offset	m[r[d]+(o=p*4)] ← r[s]	st rs, o(rd)	3spd

Static vs. Dynamic Structs

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

- Static and dynamic differ by an extra memory access
  - dynamic structs have dynamic address that must be read from memory

```
struct D  d0;
d0.e = d0.f;
```

```
0x1000: value of d0.e
0x1004: value of d0.f
```

```
m[0x1000] ← m[0x1004]
```

```
r[0]     ← 0x1000

r[2]     ← m[r[0]+4]
m[r[0]] ← r[2]
```

```
struct D* d1;
d1->e = d1->f;
```

```
0x1000: 0x2000
```

```
0x2000: value of d1->e
0x2004: value of d1->f
```

```
m[m[0x1000]+0] ← m[m[0x1000]+4]
```

```
r[0]     ← 0x1000
r[1]     ← m[r[0]]
r[2]     ← m[r[1]+4]
m[r[1]] ← r[2]
```

extra dereference

Memory Management in C

- Explicit allocation with malloc and deallocation with free
- Dangling pointer problem
  - pointer to object that has already been freed
  - happens when allocate and free happen in different parts of code
  - various strategies to avoid (reduce likelihood, but not a guaranteed cure)
    - use local variables (allocated on the stack) and pass in address of the local from caller, instead of dynamic allocation in callee
    - coding conventions
    - explicit reference counting (heavyweight solution)
- Memory leak problem
  - allocated memory is not deallocated when no longer needed, so memory usage steadily grows (problem especially for long-running programs)
- Common mistake
  - don’t free any memory to avoid dangling pointer problem (in Lab 3)
    - result is memory leak, leads to later problems even though no immediate crash

Memory Management in Java

- Garbage collection model
  - allocation with new
  - deallocation handled by Java system, not programmer
    - thus some kinds of programmer errors are impossible, including dangling pointers
- Advantages
  - much easier to program
- Disadvantages
  - some performance penalties
    - system knows less than programmer in best case
    - GC pass could occur at bad time (realtime/interactive situation)
  - programmers tempted to ignore memory management completely
    - GC is not perfect, memory leaks can still occur!

Static Control Flow for If/Loop

- conditional branches: do if register is
  - *equal to* zero
  - *greater than* zero
  - often requires ALU calculation to change condition into zero check
    - tradeoff is keep ISA compact, vs. require more instructions to execute desired behavior
    - continue with RISC approach: pick compact
- unconditional
  - PC-relative (branch)
    - 8 bits to encode address with respect to current PC, fits into 2-byte instruction
    - in assembly, target is label specifying location
  - absolute (jump)
    - 32 bits to encode address, requires 6-byte instruction

Name	Semantics	Assembly	Machine
branch	pc ← (a==pc+oo*2)	br a	8-oo
branch if equal	pc ← (a==pc+oo*2) if r[c]==0	beq rc, a	9coo
branch if greater	pc ← (a==pc+oo*2) if r[c]>0	bgt rc, a	acoo
jump	pc ← a	j a	b--- aaaaaaaa

Implementing *for* Loops

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

- Transformation
  - calculate condition into zero check
  - use two branches
    - conditional to end at start
    - unconditional after loop body
  - defer store to memory
    - only after loop end
    - (when posssible)

```
           temp_i=0
           temp_s=0
top_loop:  temp_t=temp_i-10
           goto end_loop if temp_t==0
           temp_s+=a[temp_i]
           temp_i++
           goto top_loop
end_loop:  s=temp_s
           i=temp_i
```

```
          ld   $0x0, r0          # r0 = temp_i = 0
          ld   $a, r1            # r1 = address of a[0]
          ld   $0x0, r2          # r2 = temp_s = 0
          ld   $0xfffffff6, r4   # r4 = -10
loop:     mov  r0, r5            # r5 = temp_i
          add  r4, r5            # r5 = temp_i-10
          beq  r5, end_loop      # if temp_i=10 goto +4
          ld   (r1, r0, 4), r3   # r3 = a[temp_i]
          add  r3, r2            # temp_s += a[temp_i]
          inc  r0                # temp_i++
          br   loop              # goto -7
end_loop: ld   $s, r1            # r1 = address of s
          st   r2, 0x0(r1)       # s = temp_s
          st   r0, 0x4(r1)       # i = temp_i
```

Implementing *if-then-else*

```
if (a>b)
  max = a;
else
  max = b;
```

```
         temp_a=a
         temp_b=b
         temp_c=temp_a-temp_b
         goto then if (temp_c>0)
else:    temp_max=temp_b
         goto end_if
then:    temp_max=temp_a
end_if: max=temp_max
```

- Transformations: same idea
  - calculate condition into zero check
  - two branches for most cases
    - conditional on top
    - unconditional to bottom to skip next case
      - except for last case, do not need
  - defer store to memory when possible

```
        ld   $a, r0            # r0 = &a
        ld   0x0(r0), r0       # r0 = a
        ld   $b, r1            # r1 = &b
        ld   0x0(r1), r1       # r1 = b
        mov  r1, r2            # r2 = b
        not  r2                # temp_c = ! b
        inc  r2                # temp_c = - b
        add  r0, r2            # temp_c = a-b
        bgt  r2, then          # if (a>b) goto +2
else:   mov  r1, r3            # temp_max = b
        br   end_if            # goto +1
then:   mov  r0, r3            # temp_max = a
end_if: ld   $max, r0          # r0 = &max
        st   r3, 0x0(r0)       # max = temp_max
```

- Common mistake (if and for)
  - only using one branch

Static Control Flow: Procedure Calls

- Set up return value
  - read the value of the program counter (PC): convention is to use r6
  - increment to skip next two instructions (incr itself, and jump)
- Do jump to callee
  - jump to a dynamically determined target address stored in register
- Procedure call: use indirect jump (with zero offset)

Name	Semantics	Assembly	Machine
get pc	r[d] ← pc	gpc rd	6f-d
indirect jump	pc ← r[t] + (o==pp*2)	j o(rt)	ctpp

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

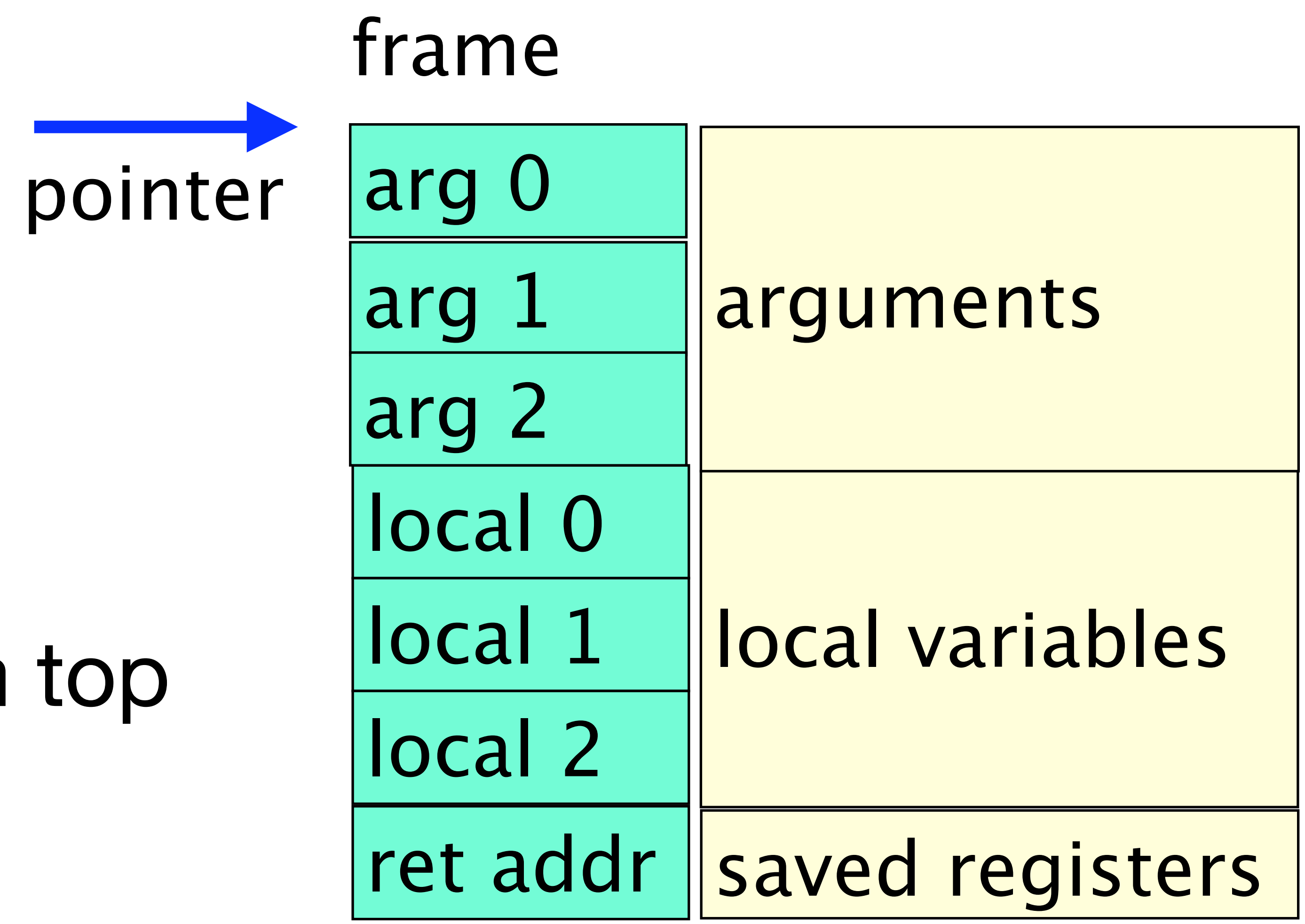
```
foo:   ld   $ping, r0   # r0 = address of ping ()
       gpc  r6          # r6 = pc of next instruction
       inca r6          # r6 = pc + 4
       j    0(r0)       # goto ping ()
```

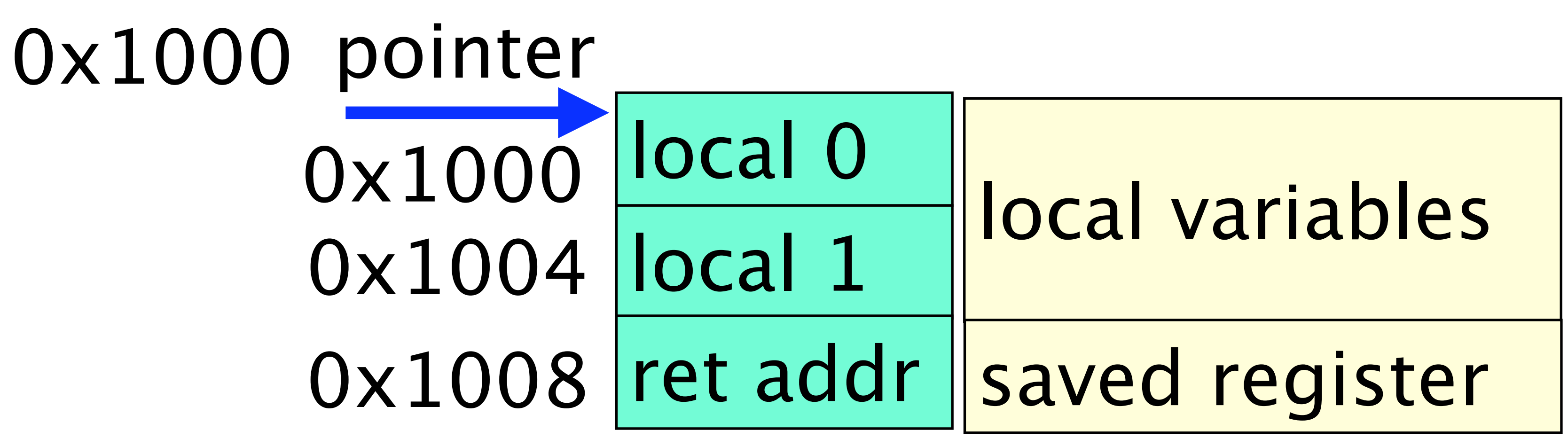
```
void ping () {}
```

```
ping:  j    0(r6)       # return
```

Procedure Storage Needs

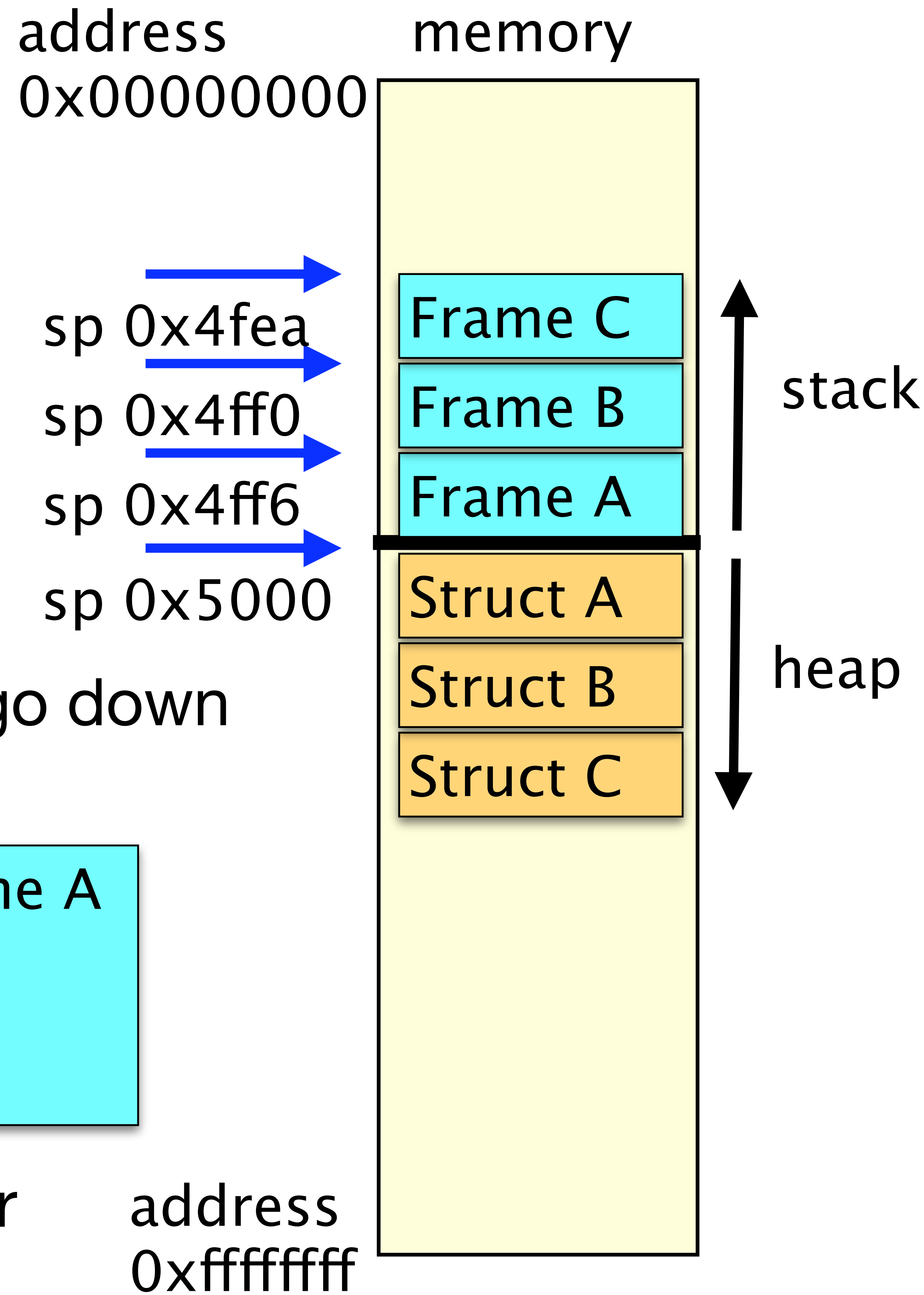
- frame
  - arguments
  - local variables
  - saved registers
    - return address
- access through offsets from top
  - just like **structs** with base
- simple example
  - two local vars
  - saved return address

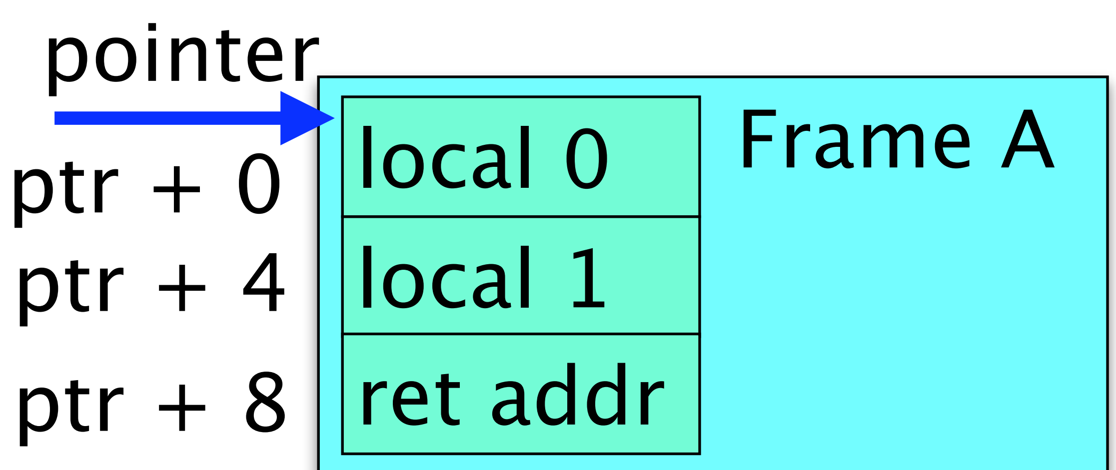




Stack vs. Heap

- split memory into two pieces
  - heap grows down
  - stack grows up
- move stack pointer up to smaller number when add frame
- but within frame, offsets still go down





- convention: r5 is stack pointer

Snippet 8: Caller vs. Callee

```
foo: deca r5              # sp-=4
     st   r6, 0x0(r5)     # save r6 to stack
```

1 allocate bar frame (1)
save r6

```
     ld   $b, r0          # address of b ()
     gpc  r6              # r6 = pc
     inca r6              # r6 = r6 + 4
     j    0x0(r0)         # goto b ()
```

2 call b()

```
     ld   0x0(r5), r6     # restore r6 from stack
     inca r5              # sp+=4
     j    0x0(r6)         # return
```

6 restore r6
dealloc bar frame (1)
return

```
b:   ld   $0xfffffff8, r0 # r0 = -8 (frames size)
     add  r0, r5          # create frame on stack
```

3 allocate bar frame (2)

```
     ld   $0, r0          # r0 = 0
     st   r0, 0x0(r5)     # l0 = 0
     ld   $0x1, r0        # r0 = 1
     st   r0, 0x4(r5)     # l1 = 1
```

4 body

```
     ld   $0x8, r0        # r0 = 8 = (frame size)
     add  r0, r5          # teardown frame
     j    0x0(r6)         # return
```

5 dealloc bar frame (2)
return

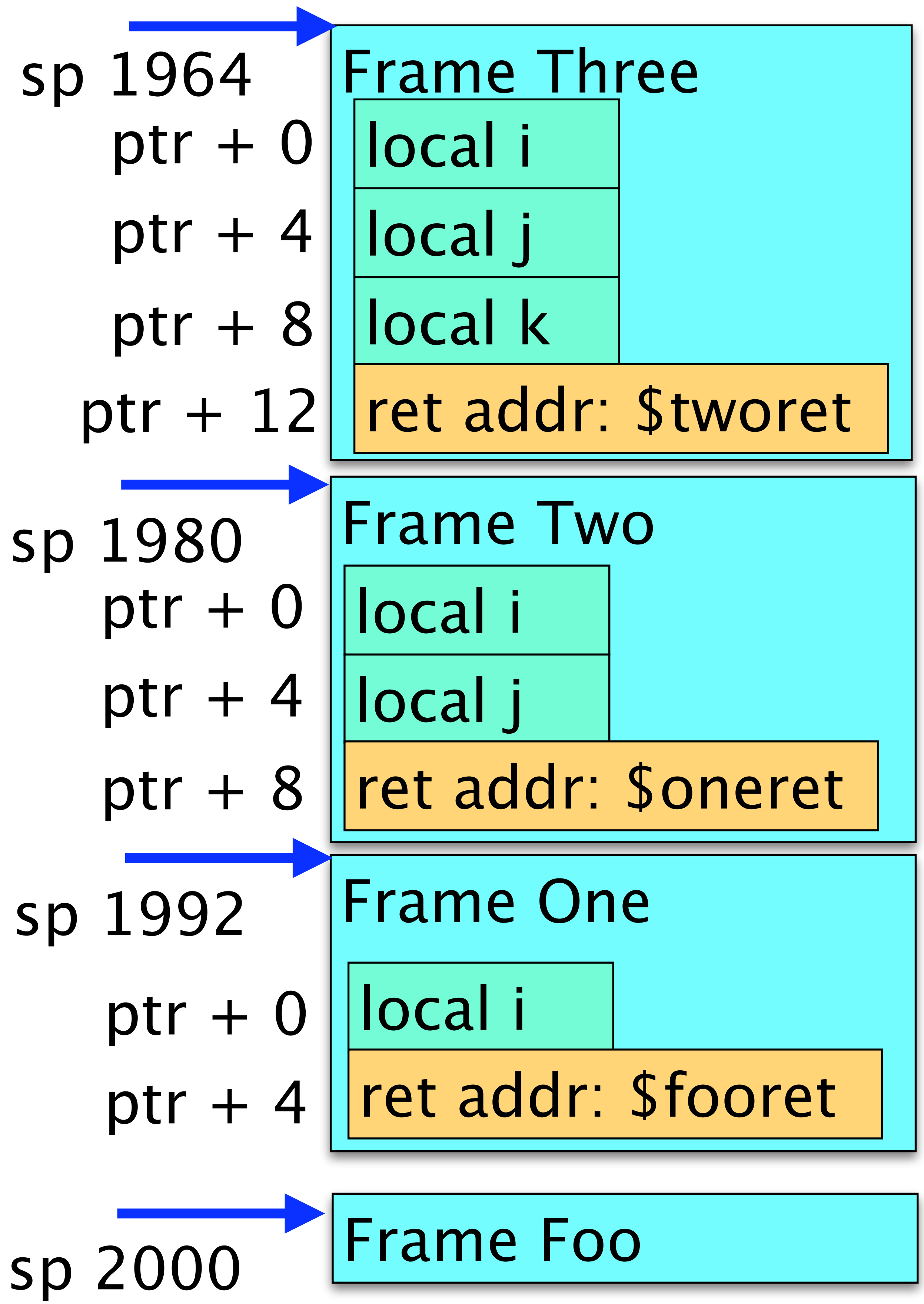
Stack Frame Setup: Caller/Callee Work

```
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 ();
}
```



before jump to three() code: save r6 to stack then set r6 to $threeret

before jump to two() code: save r6 to stack then set r6 to $tworet

before jump to one() code: save r6 to stack then set r6 to $oneret

r6 is$fooret

Arguments and Return Value

- Return value
  - convention: store in r0 register
  - common mistake:
    - push return value on stack instead of using r0
- Arguments
  - in registers or on stack
  - pushing on stack requires more work, but holds unlimited number
  - work must be done by caller
  - common mistake:
    - allocate space and save off arguments to stack in callee

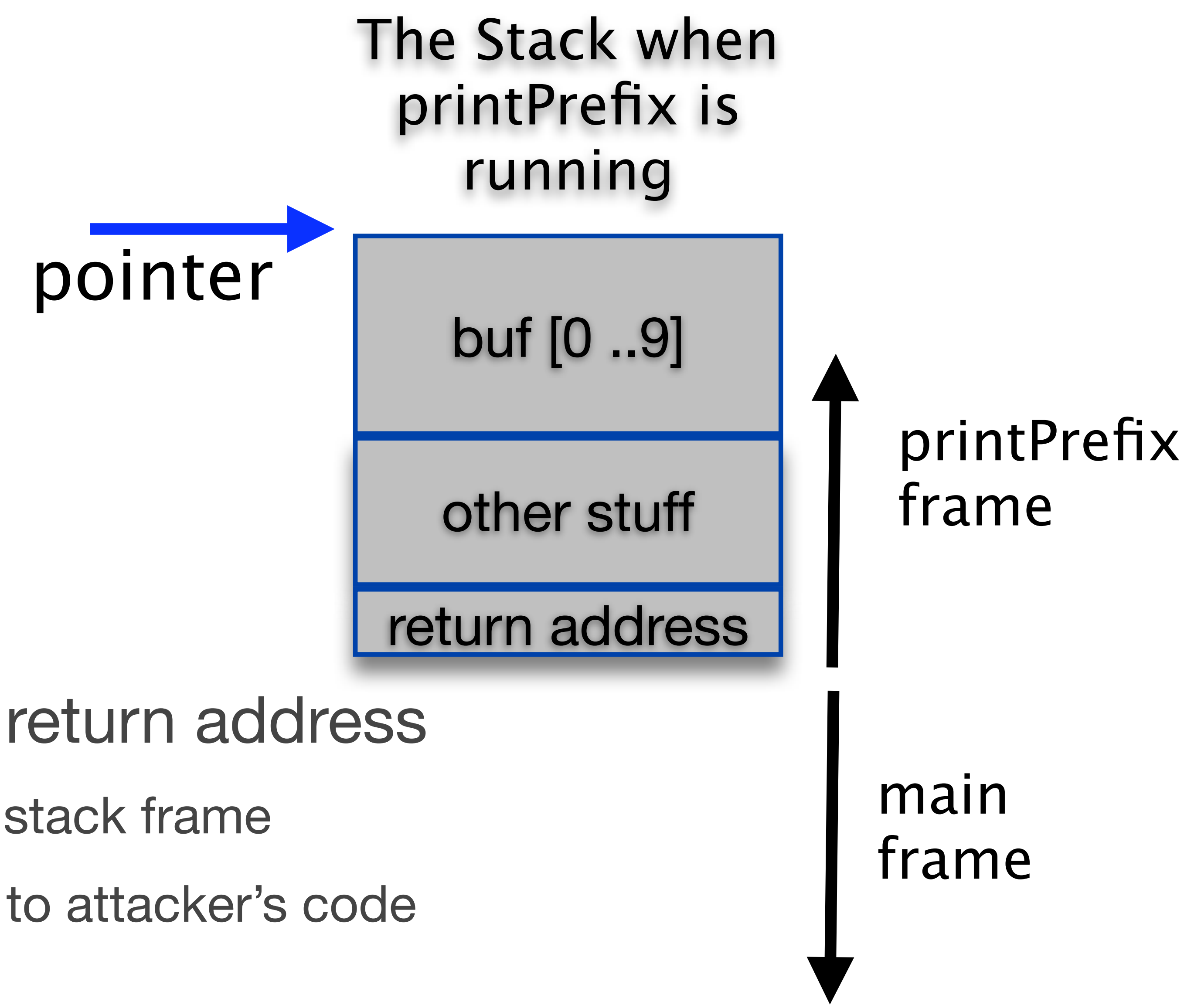
Stack Summary

- stack is managed by code that the compiler generates
  - stack pointer (sp) is current top of stack (stored in r5)
    - grows from bottom up towards 0
    - push (allocate) by decreasing sp value, pop (deallocate) by increasing sp value
- accessing information from stack
  - callee accesses local variables, arguments as static offsets from base of stack pointer (r5)
- stack frame for procedure created by mix of caller and callee work
  - common mistake: confusion about what caller vs callee should do
  - caller setup
    - allocates room for old value of r6 and saves it to stack
    - if arguments passed through stack: allocates room for them and save them to stack
    - sets up new value of r6 return address (to next instruction in this procedure, after the jump)
    - jumps to callee code
  - callee setup
    - allocates space on stack for local variables
  - callee teardown
    - ensure return value in r0
    - deallocates stack frame space for locals
    - jump back to return address (location stored in r6)
  - caller teardown
    - deallocates stack frame space for arguments
    - restores old r6 (and any other saved registers)
    - use return value (if any) in r0

Security Vulnerability: Buffer Overflow

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

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

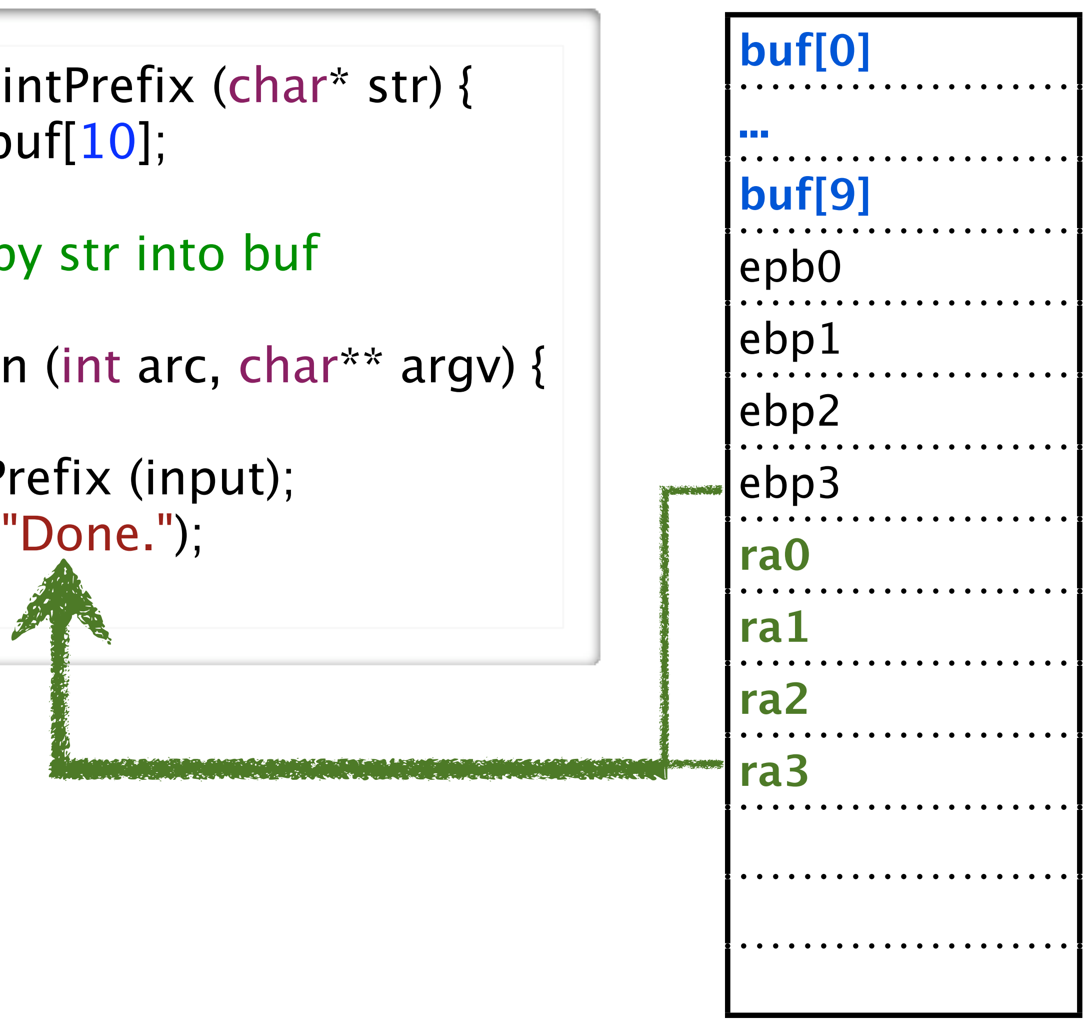


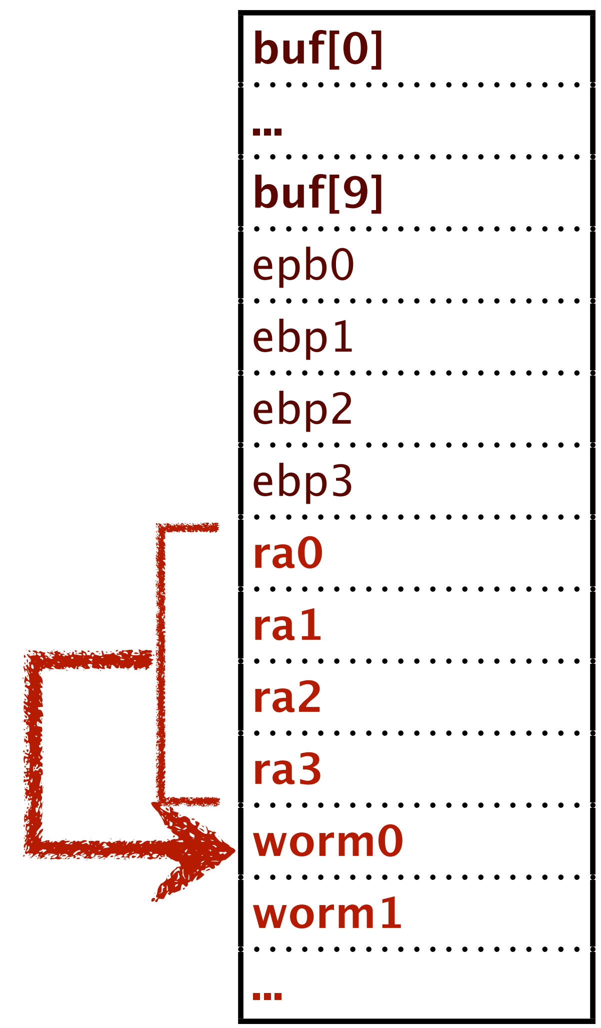
- The vulnerability
  - attacker can change printPrefix's return address
    - buf[XX] can overwrite return address on stack frame
    - instead of return to caller code, "return" to attacker's code
      - execute arbitrary code

Overflow Attack

- The attack input string has three parts
  - a portion that writes memory up to the return address
  - a new value of the return address
  - the worm code itself that is stored at this address
- Sequence
  - worm loaded on stack just below changed return address
  - return address changed so points to that location
  - when r6 called, control flow goes to worm code

```
void printPrefix (char* str) {
  char buf[10];
  ...
  // copy str into buf
}
int main (int arc, char** argv) {
  ...
  printPrefix (input);
  puts ("Done.");
}
```





Variables Summary

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

Polymorphic Dispatch

- Method address is determined dynamically
  - compiler can not hardcode target address in procedure call
  - instead, compiler generates code to lookup procedure address at runtime
  - address is stored in memory in the object's class *jump table*
- Class Jump table
  - every class is represented by class object
  - the class object stores the class's jump table
  - the jump table stores the address of every method implemented by the class
  - objects store a pointer to their class object
- Static and dynamic of method invocation
  - address of jump table is determined dynamically
  - method's offset into jump table is determined statically

Dynamic Jumps in C

- Function pointer
  - a variable that stores a pointer to a procedure
  - declared
    - <return-type> (*<variable-name>)(<formal-argument-list>);
  - used to make dynamic call
    - <variable-name> (<actual-argument-list>);
- Example

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

```
void foo () {
  void (*aFunc) ();

  aFunc = ping;
  aFunc ();
}
```

calls ping

Double-Indirect Jump: Base/Offset

- Key observation
  - base address stored in register (dynamic)
  - for polymorphism jump table, offset can be computed statically by compiler
- Function pointers: use double-indirect base/offset jump instruction

Name	Semantics	Assembly	Machine
dbl-ind jump b+o	pc ← m[r[**t**] + (**o**==**pp***2)]	j ***o**(r**t**)	**dtpp**

Switch Statement

```
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;
  }
}
```

```
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;
}
```

- Semantics the same as simplified nested if statements
  - choosing one computation from a set
  - restricted syntax: static, cardinal values
- Potential benefit: more efficient computation (usually)
  - jump table to select correct case with single operation
  - if statement may have to execute each check
    - number of operations is number of cases (if unlucky)

Switch Statement Strategy

- Choose one of two strategies to implement
  - 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 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 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 Snippet

```
switch (i) {
  case 20:  j=10; break;
  case 21:  j=11; break;
  case 22:  j=12; break;
  case 23:  j=13; break;
  default:  j=14; break;
}
```

```
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, l0           # goto l0 if i>19
         br   default          # goto default if i<20
l0:      ld   $0xffffffe9, 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]
```

```
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
```

```
jmptable: .long 0x00000140     # & (case 20)
          .long 0x00000148     # & (case 21)
          .long 0x00000150     # & (case 22)
          .long 0x00000158     # & (case 23)
```

Double-Indirect Jump: Indexed

- Key observation
  - base address stored in register (dynamic)
  - for switch jump table, have index stored in register
- Switch: use double-indirect jump indexed instruction

Name	Semantics	Assembly	Machine
dbl-ind jump indexed	pc ← m[r[**t**] + r[**i**]*4]	j *(r**t**,r**i**,4)	**eti-**

Static and Dynamic Jumps

Jump instructions

- specify a *target address* and a *jump-taken condition*
- target address can be static or dynamic
- jump-target condition can be static (unconditional) or dynamic (conditional)

Static jumps

- jump target address is static
- compiler hard-codes this address into instruction

Name	Semantics	Assembly	Machine
branch	pc ← (a==pc+oo*2)	br a	8-oo
branch if equal	pc ← (a==pc+oo*2) if r[c]==0	beq a	9coo
branch if greater	pc ← (a==pc+oo*2) if r[c]>0	bgt a	acoo
jump	pc ← a	j a	b--- aaaaaaaa

Dynamic jumps

- jump target address is dynamic

Dynamic Jumps

Indirect jump

- Jump target address stored in a register
- We already introduced this instruction, but used it for ***static*** procedure calls

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

Double indirect jumps

- Jump target address stored in memory
- Base-plus-displacement (function pointers) and indexed (switch) modes for memory access

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

Dynamic Control Flow Summary

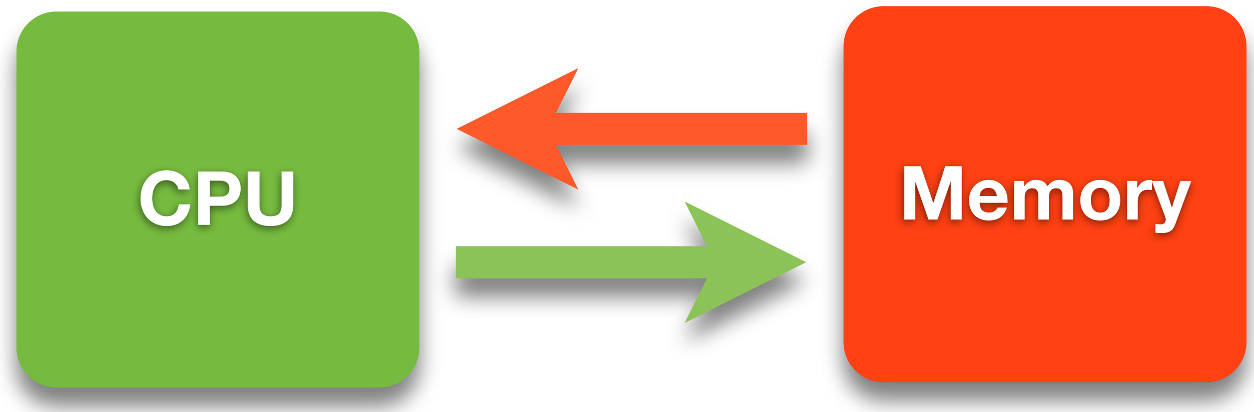
- Static vs dynamic flow control
  - static if jump target is known by compiler
  - dynamic for polymorphic dispatch, function pointers, and switch statements
- Polymorphic dispatch in Java
  - invoking a method on an object in Java
  - method address depends on object's type, which is not known statically
  - object has pointer to class object; class object contains method jump table
  - procedure call is a double-indirect jump – i.e., target address in memory
- Function pointers in C
  - a variable that stores the address of a procedure
  - used to implement dynamic procedure call, similar to polymorphic dispatch
- Switch statements
  - syntax restricted so that they can be implemented with jump table
  - jump-table implementation running time is independent of the number of case labels
  - but, only works if case label values are reasonably dense

Big Ideas: Second Half

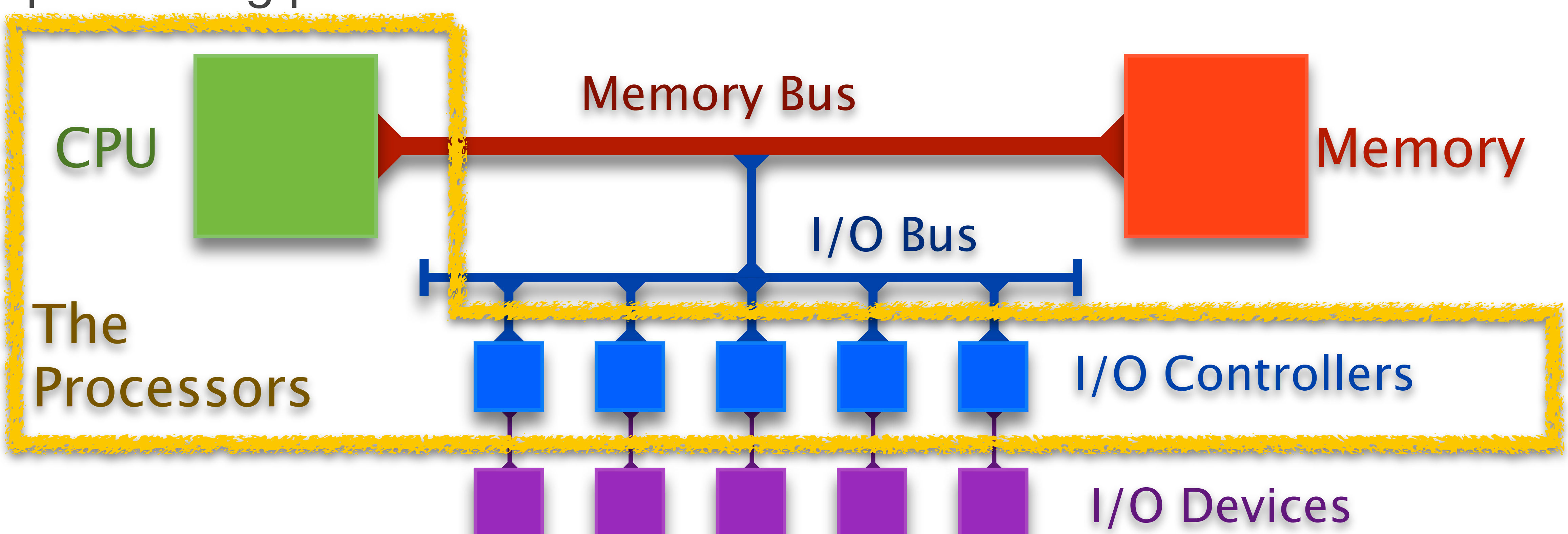
- Memory hierarchy
  - progression from small/fast to large/slow
    - registers (same speed as ALU instruction execution, roughly: 1 ns clock tick)
    - memory (over 100x slower: 100ns)
    - disk (over 1,000,000x slower: 10 millisec)
    - network (even worse: 200+ millisec RT to other side of world just from speed of light in fiber)
  - implications
    - don’t make ALU wait for memory
      - ALU input only from registers, not memory
    - don’t make CPU wait for disk
      - interrupts, threads, asynchrony
- Clean abstraction for programmer
  - ignore asynchronous reality via threads and virtual memory (mostly)
  - explicit synchronization as needed

Adding I/O to Simple Machine

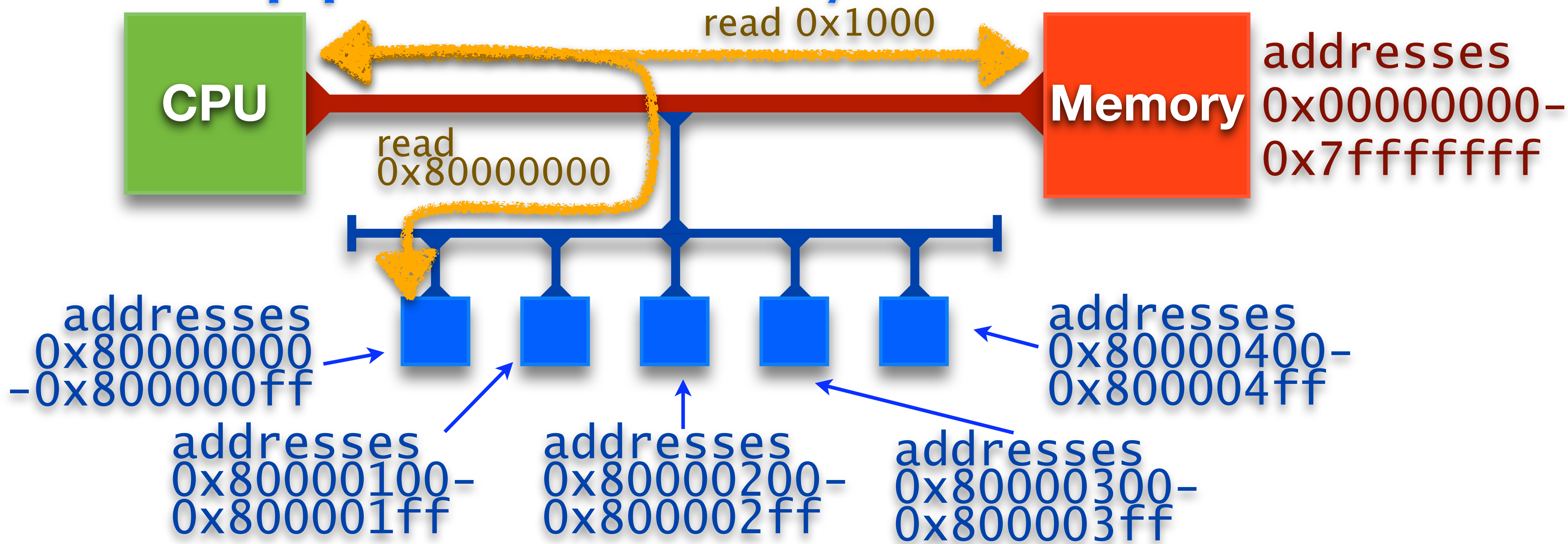
- Beyond CPU/memory
  - CPU: ALU and registers



- I/O devices have small processors: I/O controllers
  - processing power available outside CPU



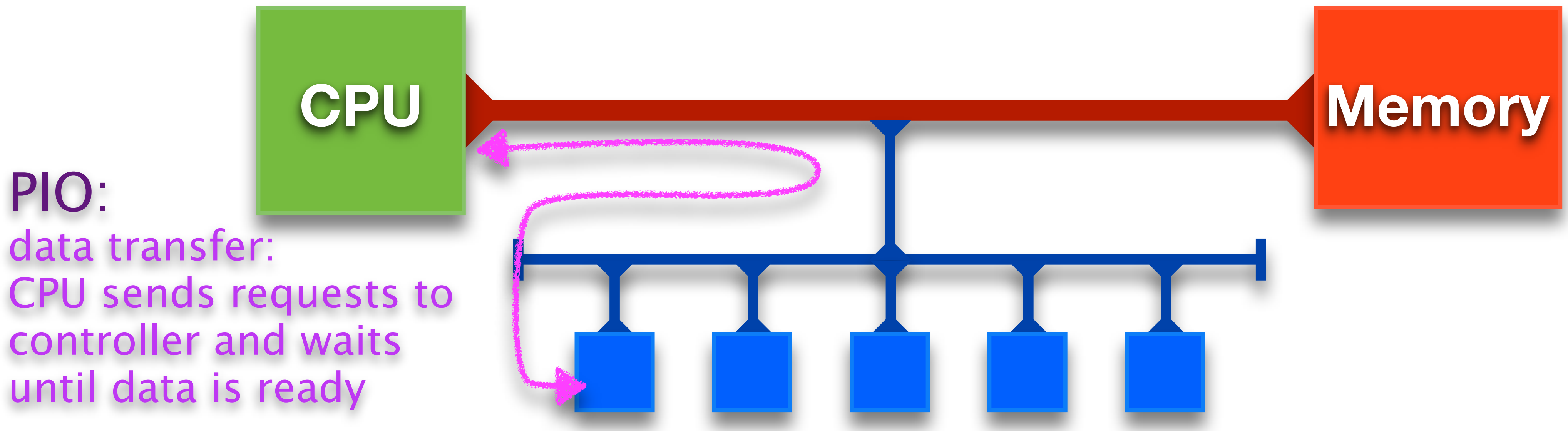
I/O-Mapped Memory



- I/O-Mapped Memory
  - use familiar syntax for load/store for both memory and I/O
  - memory addresses beyond the end of main memory handled by I/O controllers
    - mapping configured at boot time
  - loads and stores are translated into I/O-bus messages to controller
- Example
  - to read/write to controller at address 0x80000000

```
ld  $0x80000000, r0
st  r1 (r0)         # write the value of r1 to the device
ld  (r0), r1        # read a word from device into r1
```

Programmed IO (PIO)



- CPU requests one word at a time and waits for I/O controller
  - CPU must wait until data is available
    - but I/O devices may be **much** slower than CPU (disks millions of times slower)
  - large transfers slow since must be done one word at a time
  - CPU must check back with I/O controller (for instance by polling)
    - poll too often means high overhead
    - poll too seldom means high latency
  - no way for I/O controller to initiate communication
    - for some devices CPU has no idea when to poll (network traffic, mouse click)

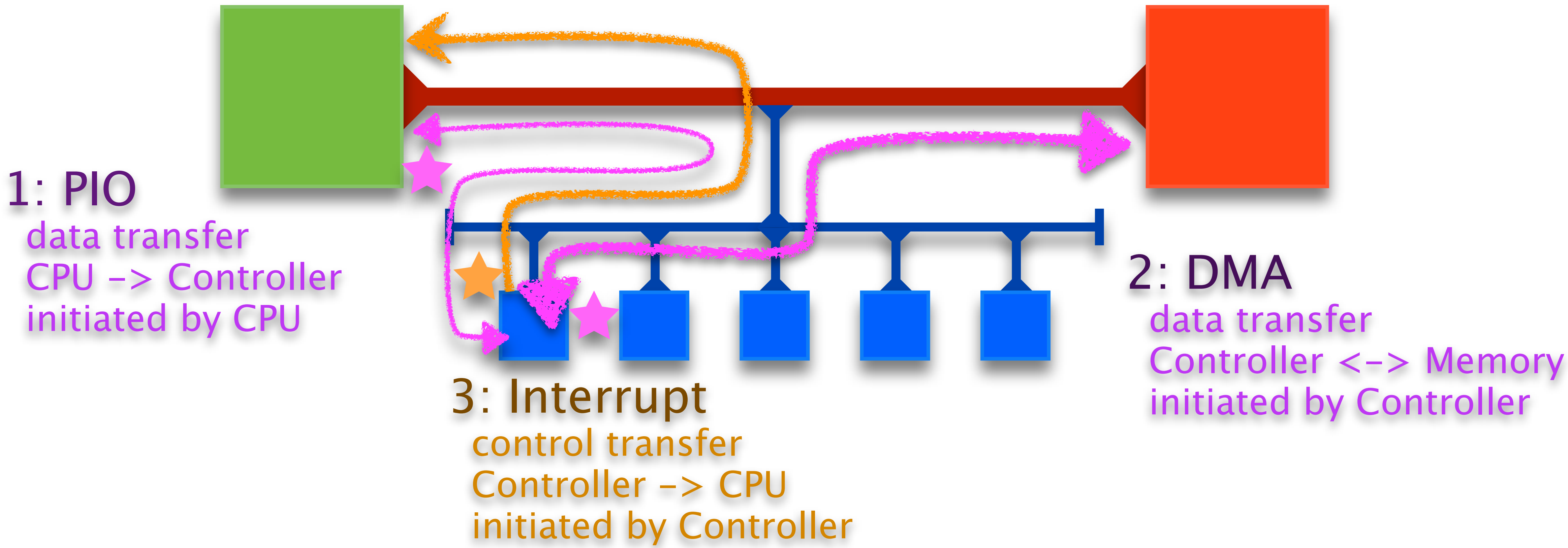
Interrupts

CPU Interrupts

- controller can signal the CPU by setting special-purpose registers
  - **isDeviceInterrupting** set by I/O Controller to signal interrupt
  - **interruptControllerID** set by I/O Controller to identify interrupting device
- CPU checks for interrupts on every fetch-execute cycle
  - polling, but very low overhead of register access: does not slow down computation
- CPU jumps to controller's *Interrupt Service Routine* to service interrupt
  - **interruptVectorBase** interrupt-handler jump table, initialized at boot time

```
while (true) {
  if (isDeviceInterrupting) {
    m[r[5]-4] ← r[6];
    r[5]      ← r[5]-4;
    r[6]      ← pc;
    pc        ← interruptVectorBase [interruptControllerID];
  }
  fetch ();
  execute ();
}
```

Direct Memory Access (DMA)



- I/O controller transfers data to/from main memory independently of CPU
  - process initiated by CPU using PIO
    - send request to controller with addresses and sizes
  - data transferred to memory without CPU involvement
  - controller signals CPU with interrupt when transfer complete
- can transfer large amounts of data with one request
  - not limited to one word at a time

Asynchronous Disk Reading

- Cannot depend on synchronized execution where result is available before next statement executed

```
read        (buf, siz, blkNo);
nowHaveBlock (buf, siz);
```

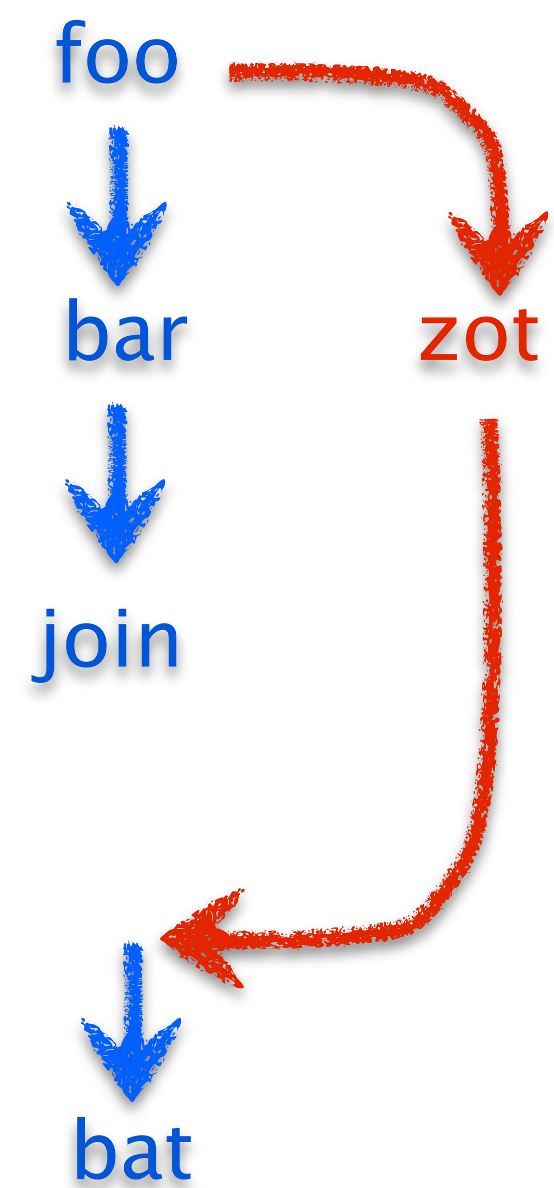
- Handling disk reads asynchronously
  - each request has completion routine that should run after interrupt

```
asyncRead (buf, siz, blkNo, nowHaveBlock);
```

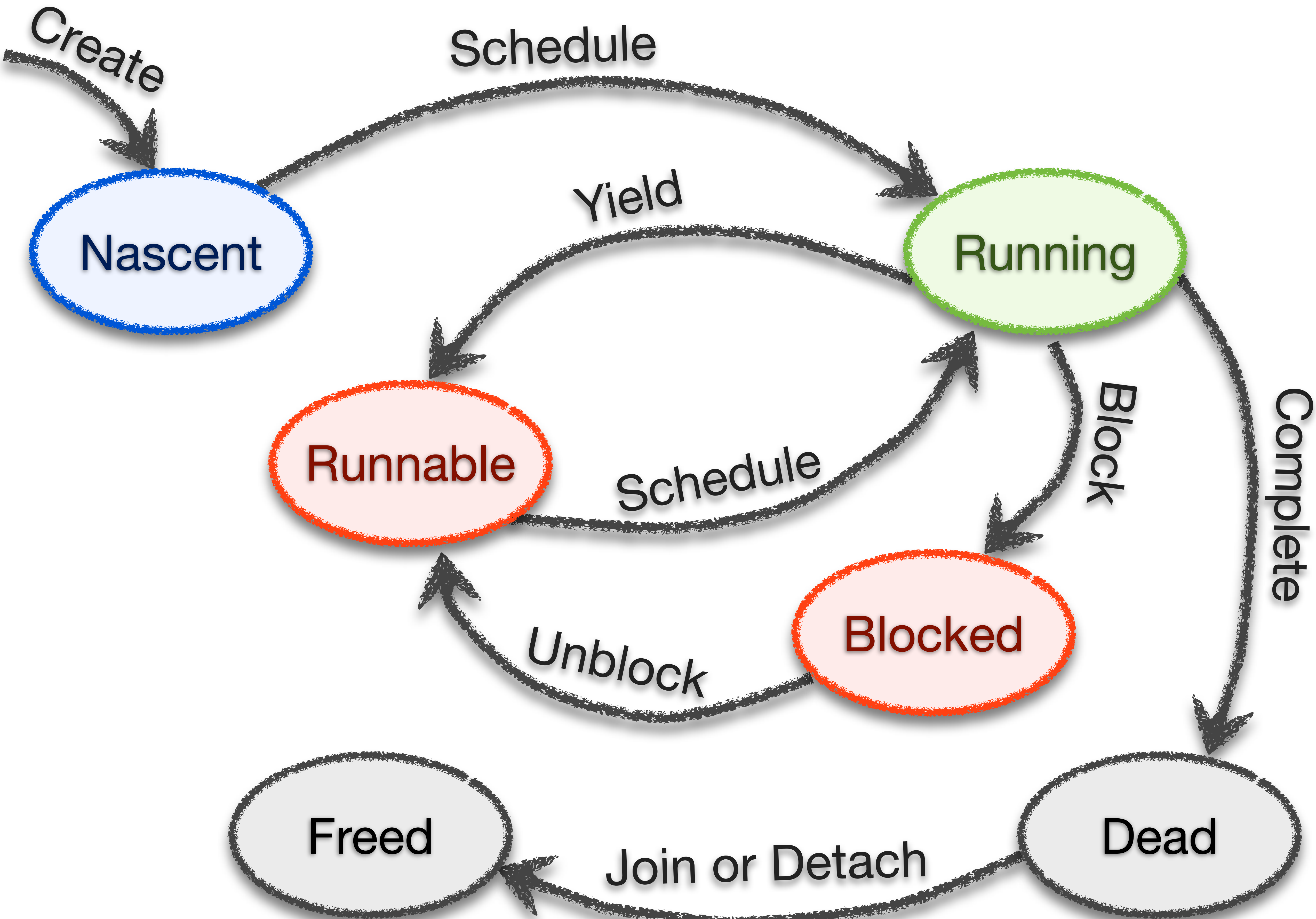
  - need queue so can handle multiple pending requests
- Challenges of asynchrony
  - either programmers must use explicitly asynchronous programming model
    - decoupled event triggering and handling as with event-driven GUI programming
    - imagine if not just on mouse clicks, but for every memory access!
  - or system can provide abstractions to hide asynchrony from programmers
    - threads, processes, virtual memory

Threads

- Abstraction for execution
  - programmer's view
    - statements are executed one after another, appearance of sequential flow
  - system reality
    - threads maybe be blocked (stopped)
    - often thread is not running because CPU is running a different thread
    - blocked threads can be restarted
- Using threads
  - create
    - starts new thread, immediately adds it to queue of threads waiting to run
  - join
    - blocks calling thread until target thread completes
  - common mistakes:
    - assume that order of joining is order of execution
    - assume that order of creating is order of execution
      - thread joins runnable queue with create call, not with join call
      - scheduler may choose what to run next in any order



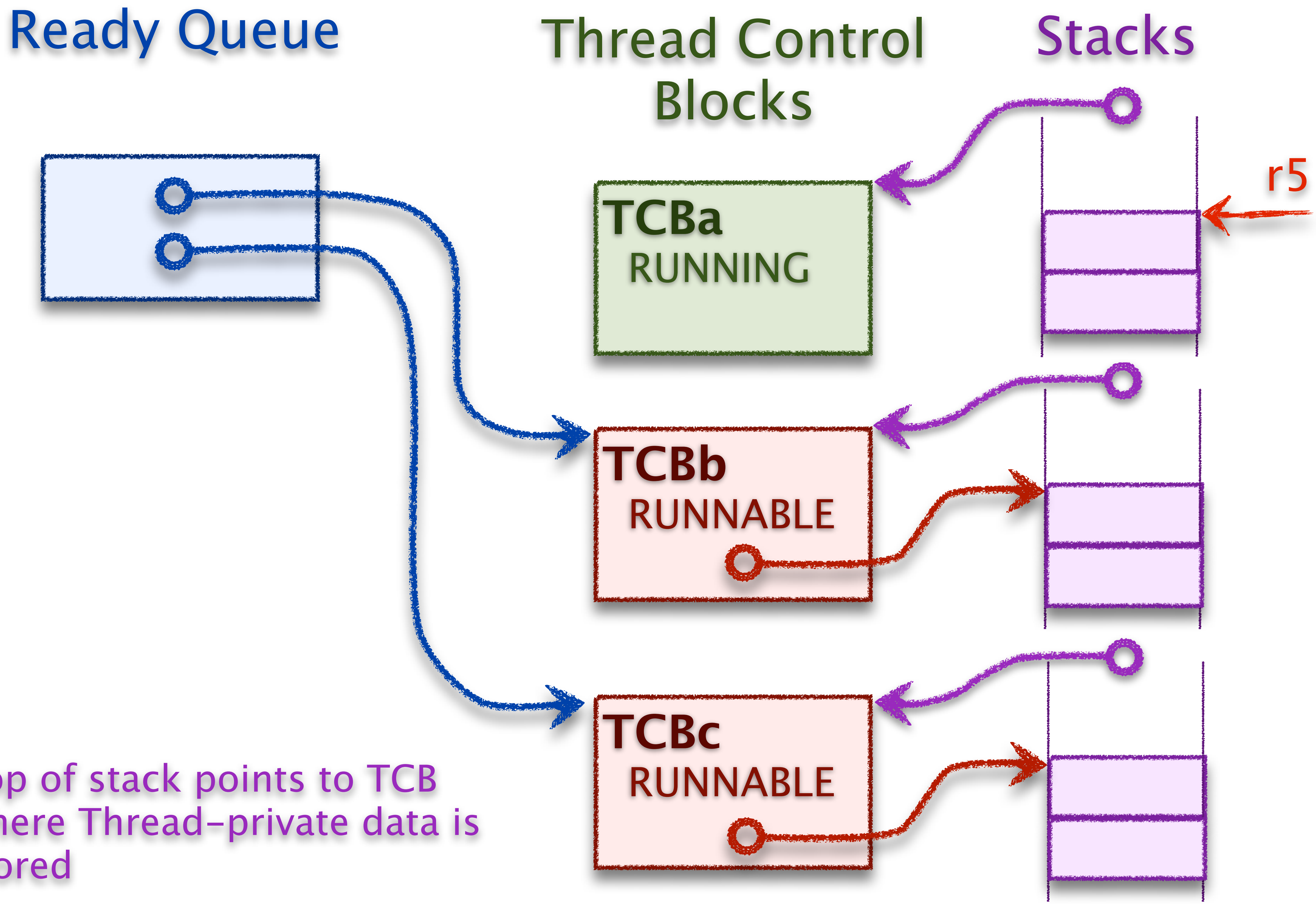
Thread Status DFA



Implementing Threads

- Each thread has own copy of stack
- Thread-Control Block (TCB)
  - thread status: (NASCENT, RUNNING, RUNNABLE, BLOCKED, or DEAD)
  - pointers to base of thread's stack base and top of thread's stack
  - scheduling parameters such as priority, quantum, pre-emptability, etc.
- Queues
  - ready: list of TCB's of all RUNNABLE threads
  - blocked: list of TCB's of BLOCKED threads
- Thread switch (stops Ta and starts Tb)
  - save all registers to stack
  - save stack pointer to T_a's TCB
  - set stack pointer to stack pointer in T_b's TCB
  - restore registers from stack

Thread Private Data



Top of stack points to TCB where Thread-private data is stored

Thread Scheduling Policies

- Priority
  - choose highest priority runnable thread to run
- Round-Robin
  - equal-priority threads get fair share of processor, in round-robin fashion
- Preemptive
  - priority-based
    - lower priority thread preempted as soon as higher priority becomes runnable
  - quantum-based
    - thread preempted when its time quantum expires
    - timer device: I/O controller connected to clock, sends interrupts to CPU at regular intervals
- Can be combined

Mutual Exclusion

- Use mutual exclusion to guard critical sections where data shared between multiple threads is accessed
  - avoid race conditions where conflicting operations on shared data are interleaved arbitrarily leading to nondeterministic behavior
    - example: stack corruption when push and pop interleaved without being guarded
- Mutual exclusion with locks
  - spinlock
    - thread busy-waits until lock acquired
    - use when locks only needed for short time
  - blocking locks
    - thread blocks if lock not available
    - thread returned to runnable state when lock becomes available
    - use when locks may be held for long periods

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 primitives
  - **lock** acquire lock, wait if necessary
  - **unlock** release lock, allowing another thread to acquire if waiting
- using locks for the shared stack

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

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

  return e;
}
```

Spinlocks Require Atomic Read/Write

- Impossible when read and write are separate operations

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

Another thread could run in between read and write

- Need **atomic** read and write that is single indivisible unit
  - with no intervening access to that memory location from any other thread allowed
- Atomic Memory Exchange
  - one type of atomic memory instruction (there are other types)
  - group a load and store together atomically
  - exchanging the value of a register and a memory location
  - much higher overhead than standard load or store

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

Implementing Spinlocks

- Spin first on fast normal read, then try slow atomic 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
loop: ld   (%r1), %r0
      beq  %r0, try
      br   loop
try:  ld   $1, %r0
      xchg (%r1), %r0
      beq  %r0, held
      br   loop
held:
```

- common mistake:
  - assume that atomic exchange always succeeds; could fail!

Blocking Locks

- If a thread may wait a long time
  - it should block so that other threads can run
  - it will then unblock when it becomes runnable (lock available or event notification)
- Blocking locks for mutual exclusion
  - if lock is held, locker puts itself on waiter queue and blocks
  - when lock is unlocked, unlocker restarts one thread on waiter queue
- Blocking locks for event notification (condition variables)
  - waiting thread puts itself on a a waiter queue and blocks
  - notifying thread restarts one thread on waiter queue (or perhaps all)
- Implementing blocking locks using spinlocks
  - lock data structure includes a waiter queue and a few other things
  - data structure is shared by multiple threads; lock operations are critical sections
  - thus we use spinlocks to guard these sections in blocking lock implementation

Implementing a Blocking Lock

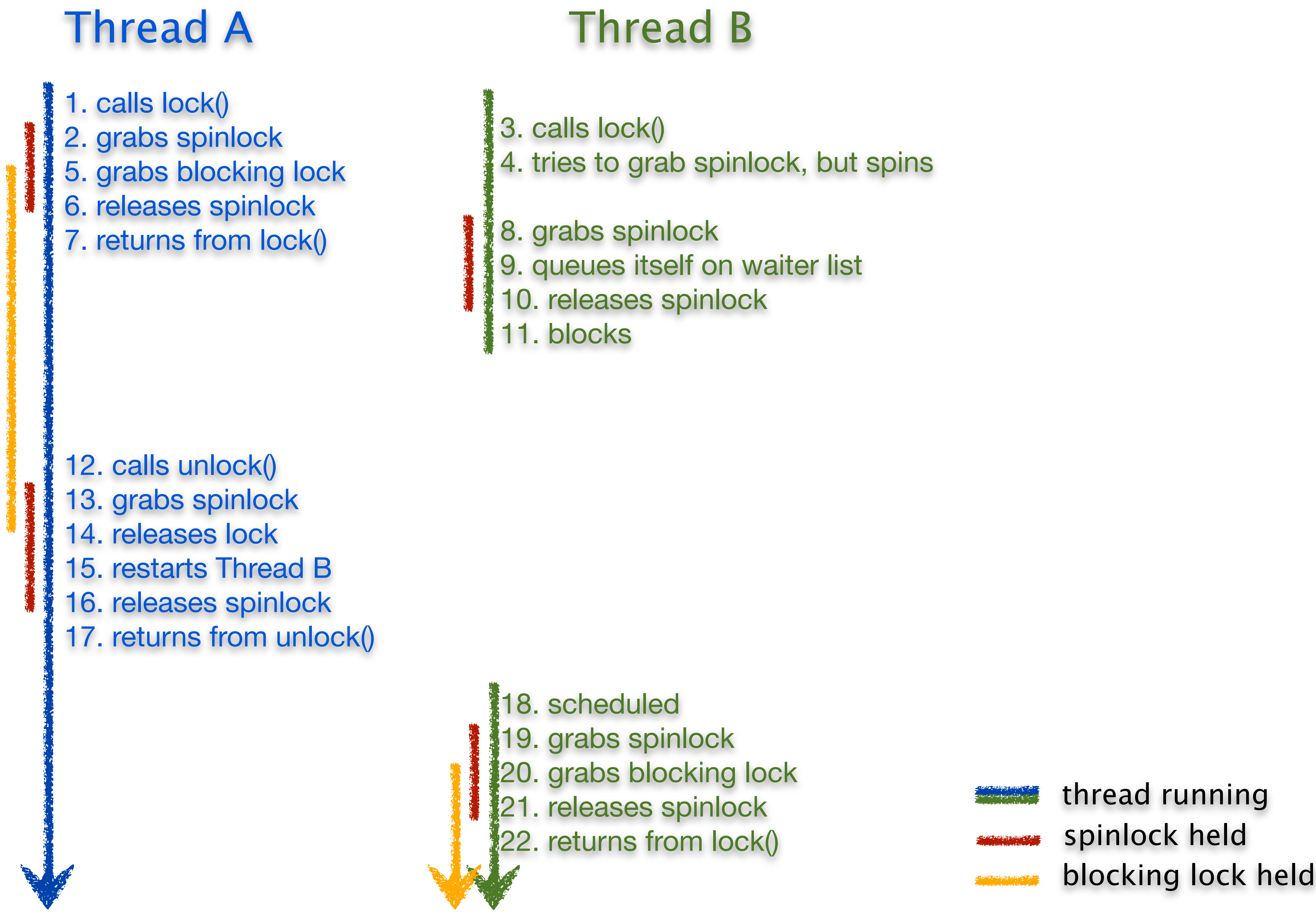
```
void lock (struct blocking_lock l) {
  spinlock_lock (&l->spinlock);
    while (l->held) {
      enqueue         (&waiter_queue, uthread_self ());
      spinlock_unlock (&l->spinlock);
      uthread_switch  (ready_queue_dequeue (), TS_BLOCKED);
      spinlock_lock   (&l->spinlock);
    }
    l->held = 1;
  spinlock_unlock (&l->spinlock);
}
```

```
void unlock (struct blocking_lock l) {
  uthread_t* waiter_thread;

  spinlock_lock (&l->spinlock);
  l->held = 0;
  waiter_thread = dequeue (&l->waiter_queue);
  spinlock_unlock (&->spinlock);
  waiter_thread->state = TS_RUNNABLE;
  ready_queue_enqueue (waiter_thread);
}
```

```
struct blocking_lock {
  spinlock_t        spinlock;
  int               held;
  uthread_queue_t   waiter_queue;
};
```

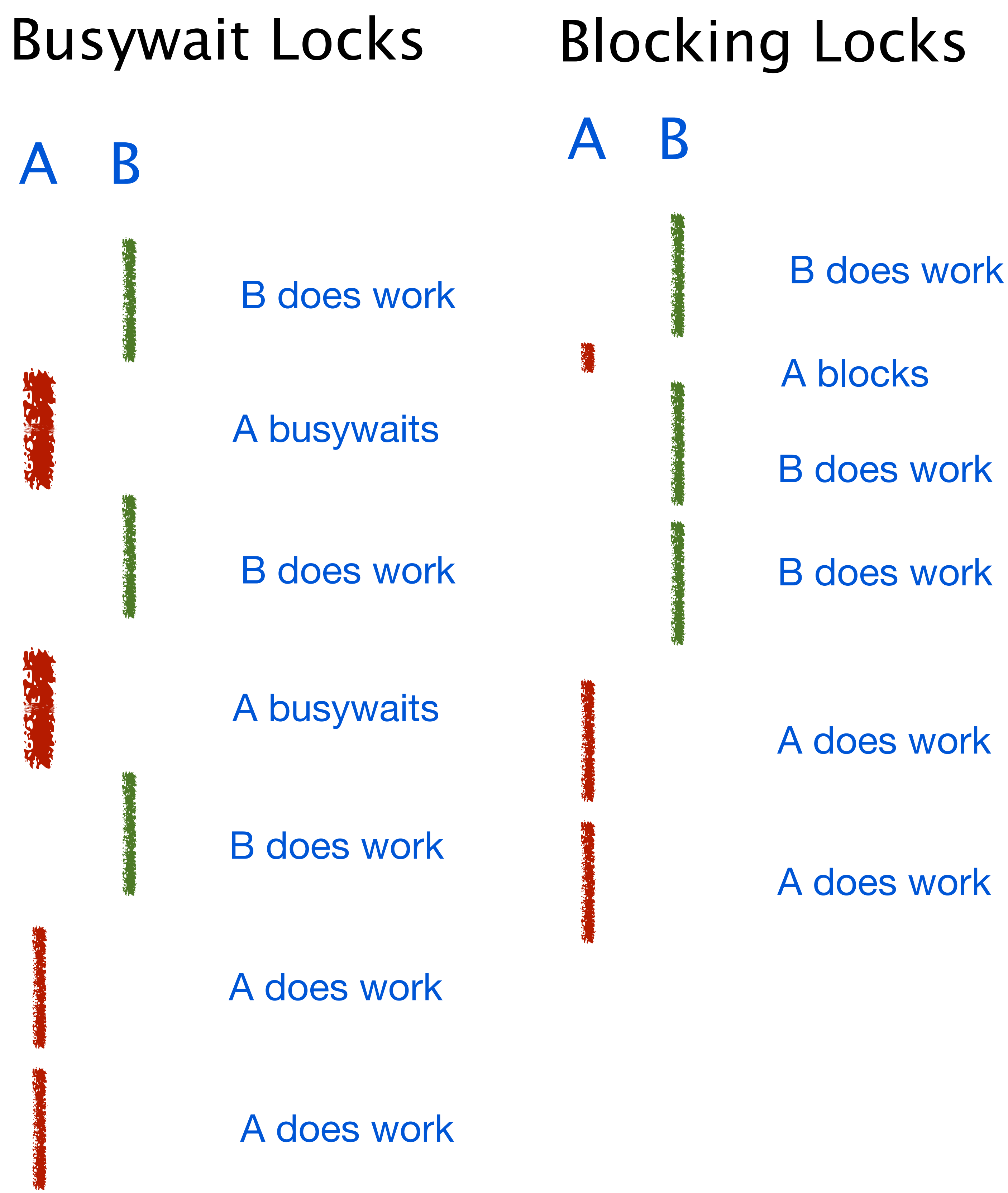
- Spinlock guard
  - on for critical sections
  - off before thread blocks

Blocking Lock Example Scenario



Busywaiting vs Blocking

- Using spinlocks to busywait for long time wastes CPU cycles
  - use for short things
    - including within implementation of blocking locks
- Using blocking locks has high overhead
  - use for long things
- Common mistake
  - assume that CPU is busywaiting during blocking locks
    - thread does not run again until after blocking lock is released



Locks and Loops Common Mistakes

- Confusion about spinlocks inside blocking locks
  - use spinlocks in the implementation of blocking locks
  - two separate levels of lock!
    - holding spinlock guarding variable read/write
    - holding actual blocking lock
- Confusion about when spinlocks needed
  - must turn on to guard access to shared variables
  - must turn off before finishing or blocking
- Confusion about loop function
  - busywait
    - only inside spinlock
  - thread blocked inside loop body, **not** busywaiting
    - yield for blocking lock
      - re-check for desired condition: is lock available?
    - blocking wait for CV, blocking wait for semaphore P implementation
      - re-check for desired condition

Synchronization Abstractions

- Monitors and condition variables
  - monitor guarantees mutual exclusion with blocking locks
  - condition variable provides control transfer among threads with wait/notify
  - abstraction supports explicit locking
- Semaphores
  - blocking atomic counter, stop thread if counter would go negative
  - introduced to coordinate asynchronous resource use
  - abstraction implicitly supports mutex, no need for explicit locking by user
  - use to implement monitors, barriers (and condition variables, sort of)

Monitors

- Provides mutual exclusion with blocking lock
  - **enter** lock
  - **exit** unlock

```
void doSomething (uthread_monitor_t* mon) {
  uthread_monitor_enter (mon);
    touchSharedMemory();
  uthread_monitor_exit (mon);
}
```

- Standard case: assume all threads could overwrite shared memory.
  - mutex: only allows access one at a time
- Special case: distinguish read-only access (readers) from threads that change shared memory values (writers).
  - mutex: allow multiple readers but only one writer

Condition Variables

- Mechanism to transfer control back and forth between threads
  - uses monitors: CV can only be accessed when monitor lock is held
- Primitives
  - **wait** blocks until a subsequent **notify** operation on the variable
  - **notify** unblocks one waiter, continues to hold monitor
  - **notify_all** unblocks all waiters (broadcast), continues to hold monitor
- Each CV associated with a monitor
- Multiple CVs can be associated with same monitor
  - independent conditions, but guarded by same mutex lock

```
uthread_monitor_t* beer = uthread_monitor_create ();
```

```
uthread_cv_t* not_empty = uthread_cv_create (beer);
uthread_cv_t* warm      = uthread_cv_create (beer);
```

Wait and Notify Semantics

- Monitor automatically exited before block on wait
  - before waiter blocks, it exits monitor to allow other threads to enter
- Monitor automatically re-entered before return from wait
  - when trying to return from wait after notify, thread may block again until monitor can be entered (if monitor lock held by another thread)
- Monitor stays locked after notify: does not block
- Implication: cannot assume desired condition holds after return from blocking wait
  - other threads may have been in monitor between wait call and return
    - must explicitly re-check: usually enclose wait in while loop with condition check
    - same idea as blocking lock implementation with spinlocks!

```
void pour () {
  monitor {
    while (glasses==0)
      wait;
    glasses--;
  }}
```

```
void refill (int n) {
  monitor {
    for (int i=0; i<n; i++) {
      glasses++;
      notify;
    }}}
```

Condition Variables

- Final will not cover Hoare blocking signal semantics
  - just nonblocking notify Hansen semantics
- Common mistake:
  - CVs do not have internal storage variables (boolean flags or int counters)
    - CVs *are* variables: named so can tell them apart from each other
    - wait/notify `tired` vs. wait/notify `hungry`

Semaphores

- Atomic counter that can never be less than 0
  - attempting to make counter negative blocks calling thread
- P(s): acquire
  - try to decrement s
  - if s would be negative, atomically blocks until s positive, then decrement s
- V(s): release
  - increment s
  - atomically unblock any threads waiting in P
- Explicit locking not required when using semaphores since atomicity built in

```
uthread_semaphore_t* glasses = uthread_create_semaphore (0);
```

```
void pour () {
  uthread_P (glasses);
}
```

```
void refill (int n) {
  for (int i=0; i<n; i++)
    uthread_V (glasses);
}
```

Semaphores

- Using semaphores: good building block for implementing many other things
  - monitors
  - condition variables (almost)
  - rendezvous: two threads wait for each other before continuing
  - barriers: all threads must arrive at barrier before any can continue

- Implementing semaphores: similar spirit to blocking locks

```
struct uthread_semaphore {
  spinlock_t          spinlock;
  int                 count;
  uthread_queue_t  waiter_queue;
};
```

```
struct blocking_lock {
  spinlock_t          spinlock;
  int                 held;
  uthread_queue_t  waiter_queue;
};
```

(really should be boolean...)

Deadlock and Starvation

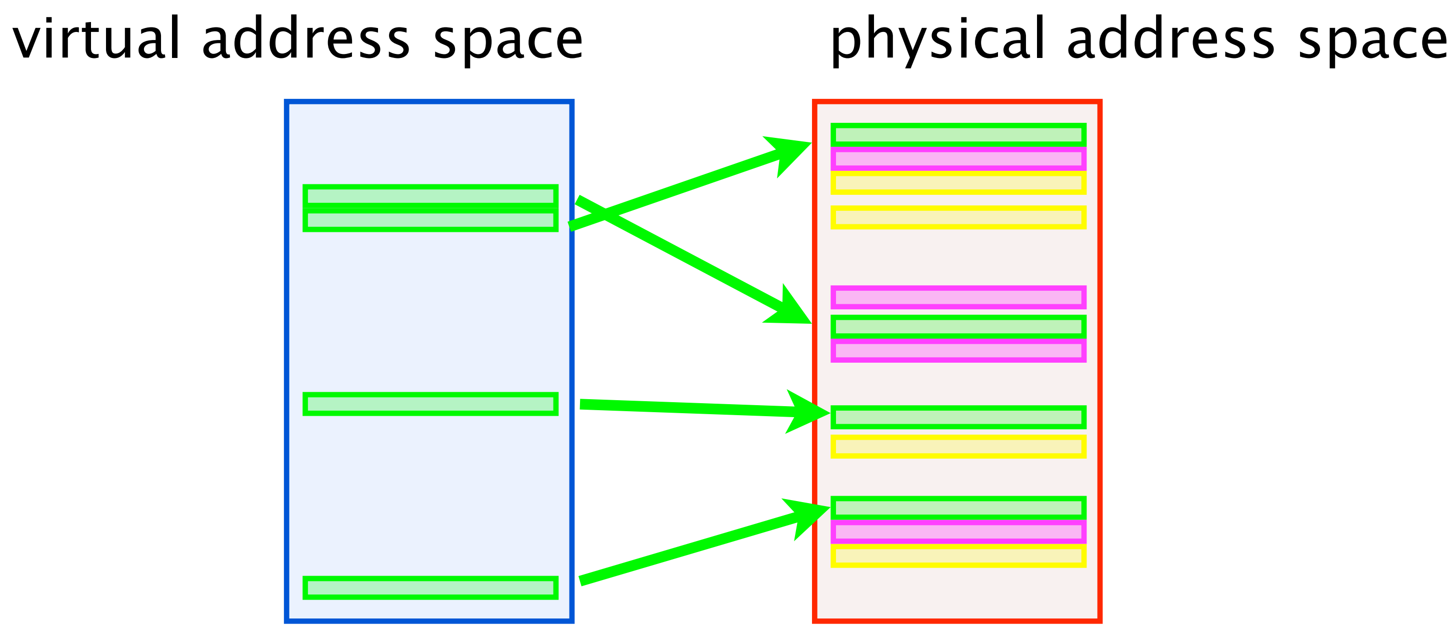
- Solved problem: race conditions
  - solved by synchronization abstractions: locks, monitors, semaphores
- Unsolved problems when using multiple locks
  - deadlock: nothing completes because multiple competing actions wait for each other
  - starvation: some actions never complete
  - no abstraction to simply solve problem, major concern intrinsic to synchronization
  - some ways to handle/avoid:
    - precedence hierarchy of locks
    - detect and destroy: notice deadlock and terminate threads

Virtual Memory

- Virtual Address Space
  - an abstraction of the *physical* address space of main (i.e., *physical*) memory
  - programs access memory using virtual addresses
  - memory management unit translates virtual address to physical memory addresses
    - MMU hardware performs translation on **every** memory access by program
- Process
  - a program execution with a private virtual address space
    - may have one or many threads
  - private address space required for static address allocation and isolation

Paging

- Key idea
  - Virtual address space is divided into set of fixed-size segments called pages
  - number pages in virtual address order
  - virtual page number = virtual address / page size
- Page table
  - indexed by virtual page number (vpn)
  - stores **base physical address** (actually address / page size (pfn) to save space)
  - stores **valid flag**



Address Space Translation Tradeoffs

- Single, variable-size, non-expandable segment
  - internal fragmentation of segment due to sparse address use
- Multiple, variable-size, non-expandable segments
  - internal fragmentation of segments when size isn’t know statically
  - external fragmentation of memory because segments are variable size
  - moving segments would resolve fragmentation, but moving is costly
- Expandable segments
  - expansion must by physically contiguous, but there may not be room
  - external fragmentation of memory requires moving segments to make room
- Multiple, fixed-size, non-expandable segments
  - called pages
  - need to be small to avoid internal fragmentation, so there are many of them
  - since there are many, need indexed lookup instead of search

Translation: Search vs. Lookup Table

- Translate by searching through all segments: too slow!

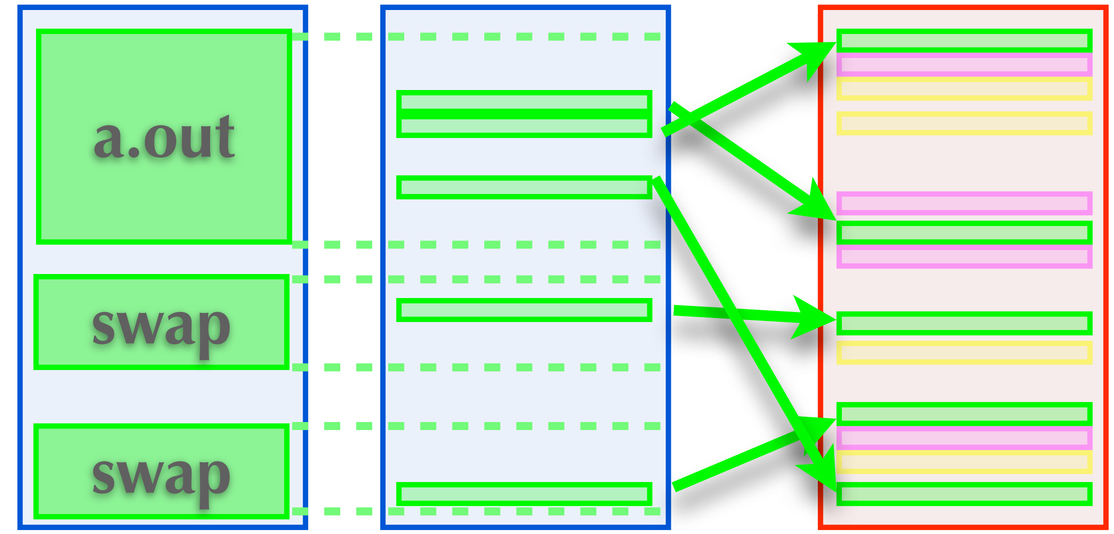
```
for (int i=0; i<segments.length; i++) {
  int offset = va - segment[i].baseVA;
  if (offset > 0 && offset < segment[i].bounds) {
    pa = segment[i].basePA + offset;
    return pa;
  }
}
throw new IllegalAddressException (va);
```

- Translate with indexed lookup: Page Table

```
class AddressSpace {
  PageTableEntry pte[];

  int translate (int va) {
    int vpn    = va / PAGE_SIZE;
    int offset = va % PAGE_SIZE;
    if (pte[vpn].isValid)
      return pte[vpn].pfn * PAGE_SIZE + offset;
    else
      throw new IllegalAddressException (va);
}}
```

```
class PageTableEntry {
  boolean isValid;
  int     pfn;
}
```

Demand Paging



- Key idea
  - some application data is not in memory
  - transfer from disk to memory, only when needed
- Page table
  - only stores entries for pages that are in memory
  - pages that are only on disk are marked invalid
  - access to non-resident page causes a page-fault interrupt
- Memory map
  - a second data structure managed by the OS
  - divides virtual address space into regions, each *mapped* to a file
  - page-fault interrupt handler checks to see if faulted page is mapped
  - if so, gets page from disk, update Page Table and restart faulted instruction
- Page replacement
  - pages can now be removed from memory, transparent to program
  - a replacement algorithm choose which pages should be resident and swaps out others

Context Switch

- Context switch: switching between threads from different processes
  - each process has private virtual address space and thus its own page table
- Context switch operations
  - thread switch (save regs, switch stacks, restore regs)
  - page table switch
    - change PTBR (page table base register) so points to new page table
    - invalidate stale page table cache entries: may require flushing entire cache
      - page table cache: TLB (translation lookaside buffer)
        - fast cache storing recent page table translations
      - new process has no valid TLB entries, so many misses
    - many pages may need reloading from disk because of demand paging
  - thus context switch can be much more expensive than thread switch

Paging Summary

- Paging
  - a way to implement address space translation
  - divide virtual address space into small, fixed sized virtual page frames
  - page table stores base physical address of every virtual page frame
  - page table is indexed by virtual page frame number
  - some virtual page frames have no physical page mapping
  - some of these get data on demand from disk

OS & Hardware Enforced Encapsulation

- Protecting operating system (OS) functions from application-level access
  - VM already protects memory: data in one address space cannot be named by process with another virtual address space
  - add hardware protection for OS function access
- User mode vs. kernel mode
  - all OS code/data included in every application page table and address space
  - split address space into two *protection domains*
    - application/user: check during VM to PM translation disallows access to OS part of space
    - user/kernel: everything accessible, including all system functionality
  - add user/kernel mode bit to each page table entry
  - add kernel mode register to CPU
  - protect switch from user to kernel mode: only through system calls
    - handled like interrupts with jump table in kernel memory
- Module not covered on final exam

Interprocess Communication

- Communication for processes that don't share memory
  - on same processor or different ones connected by network
- Key ideas
  - client/server model, packet-based transport
  - naming endpoints: IP address and port
  - communication protocol layers
    - transport (TCP/UDP), routing (IP), data (Ethernet), physical (radio/cable)
- Sockets: OS abstraction for asynchronous control transfer
  - **send**: initiate sending message payload to receiving process, but do not wait
  - **recv:** receive next available message, either blocking or not if no data waiting
- Module not covered on final exam

Summary: Second Half

- Single System Image
  - hardware implements a set of instructions needed by compilers
  - compilers translate programs into these instructions
  - translation assumes private memory and processor
- Threads
  - an abstraction implemented by software to manage asynchrony and concurrency
  - provides the illusion of single processor to applications
  - differs from processor in that it can be stopped and restarted
- Virtual Memory
  - an abstraction implemented by software and hardware
  - provides the illusion of a single, private memory to application
  - not all data need be in memory, paged in on demand
- Hardware Enforced Encapsulation
  - kernel mode register and VM mapping restriction
  - allows OS to export a public interface and to encapsulate (hide) the implementation