CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

Unit 2b – Jul 30, Aug 1

Threads

Overview

Reading

- Text: 12.3

Learning Goals

- Write C programs using threads
- Explain the execution of a multi-threaded program given the interleaved output of the threads
- Convert an event-driven C program into a procedure-driven using threads
- Identify the *state* of a thread
- Explain what happens when a thread is stopped and started by explaining what happens to the thread and what happens on the CPU that was executing it while the thread is stopped
- Describe the process of switching from one thread to another at the instruction level
- Explain the values for thread status and how threads transition from one status to another
- Identify the execution order of a set of threads of different priority using *priority-based, round-robin scheduling*
- List the benefits and drawbacks of priority-based round-robin scheduling without preemption
- Explain what preemption is, how it is incorporated into round-robin scheduling, and how it is implemented
- Compare real-time scheduling to round-robin scheduling to identify cases where each is useful

Issues Introduced by IO Devices

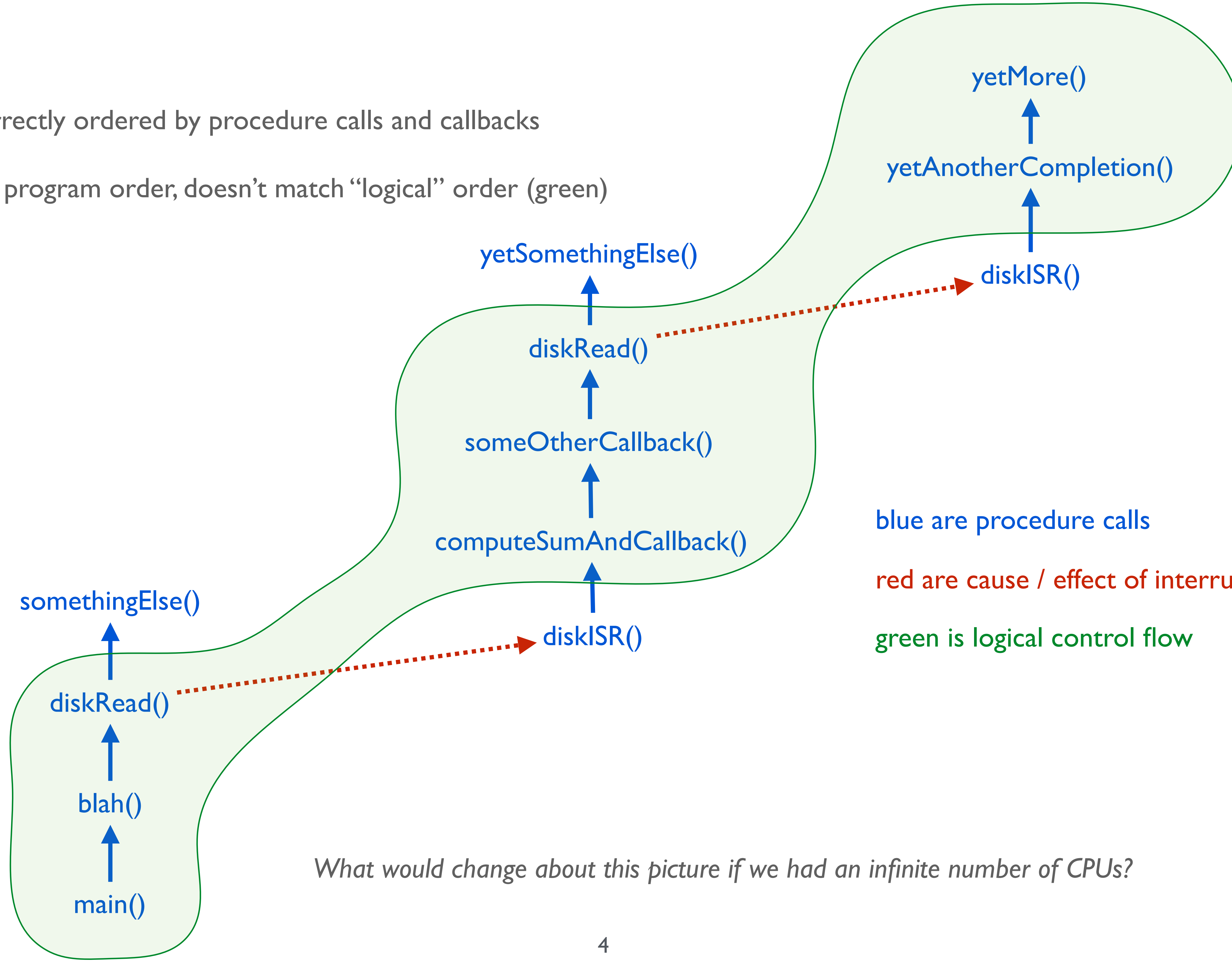
- Ordering of program events with IO completion
  - diskRead triggers IO controller to start read process
  - program has things that can only run after that process completes
- Do other things while waiting for IO event
  - need multiple independent streams of execution in program
  - one does read 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

Streams of Control in Asynchronous Program

Correctly ordered by procedure calls and callbacks

But, program order, doesn't match "logical" order (green)



blue are procedure calls

red are cause / effect of interrupts

green is logical control flow

What would change about this picture if we had an infinite number of CPUs?

Infinite CPUs: We can *Poll* the IO Device

yetMore()

yetAnotherCompletion()

WAIT by POLLING

are you done yet?
are you done yet?
are you done yet?
are you done yet?
are you done yet?

diskRead()

someOtherCallback()

computeSumAndCallback()

WAIT by POLLING

are you done yet?
are you done yet?
are you done yet?
are you done yet?
are you done yet?

diskRead()

blah()

main()

yetSomethingElse()

somethingElse()

The Virtual Processor

- Originated with Edsger Dijkstra in the THE Operating System
  - in *The Structure of the "THE" Multiprogramming System, 1968*

    *"I had had extensive experience (dating back to 1958) in making basic software dealing with real-time **interrupts**, and I knew by bitter experience that as a result of the **irreproducibility** of the interrupt moments a program error could present itself misleadingly like an occasional machine malfunctioning. As a result **I was terribly afraid**. Having fears regarding the possibility of debugging, we decided to be as careful as possible and, prevention being better than cure, to try to prevent nasty bugs from entering the construction.*

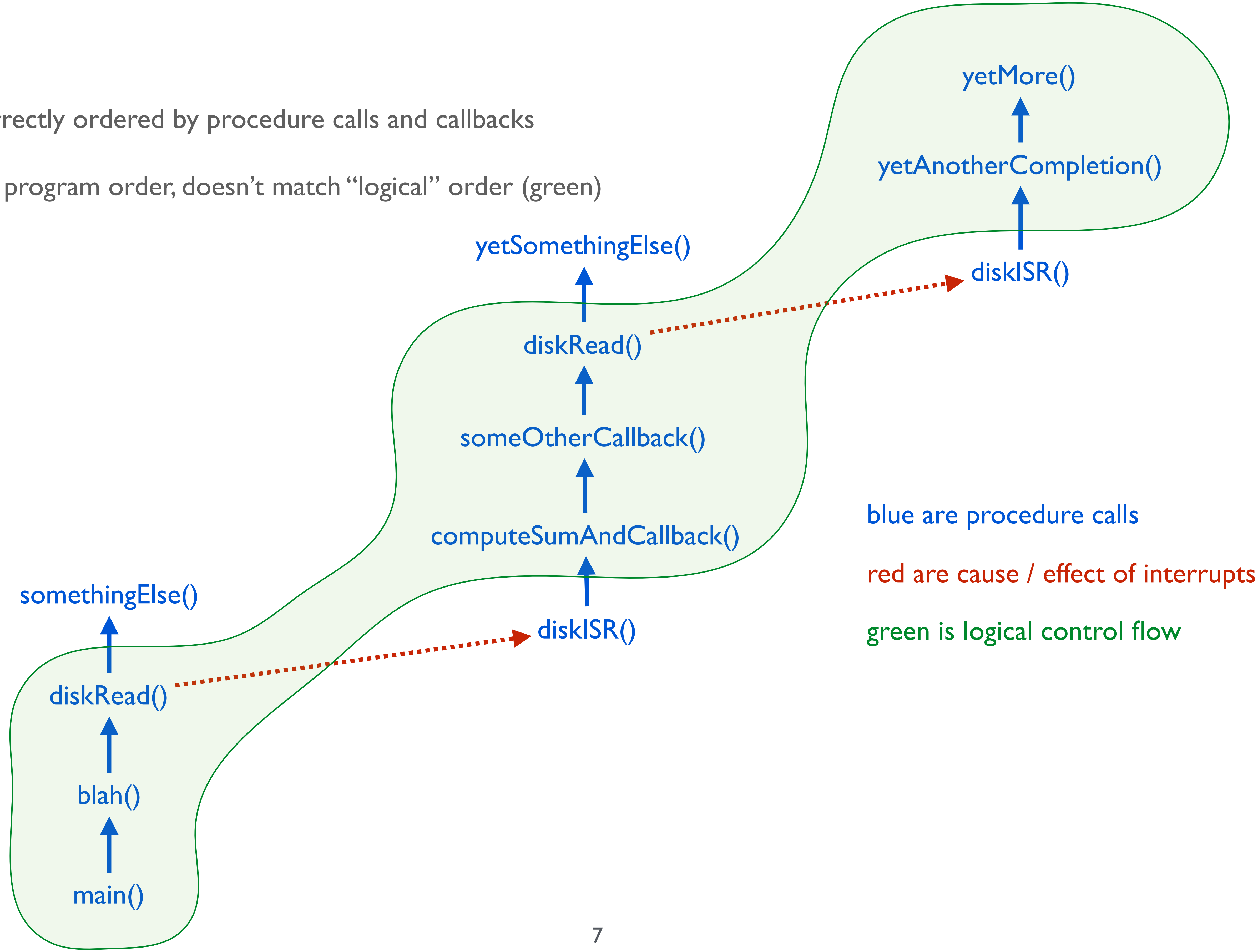
    This decision, inspired by fear, is at the bottom of what I regard as the group's main contribution to the art of system design."

- Thread (Dijkstra called it a Process)
  - a single stream of synchronous execution of a program
    - the illusion of a single system (as we assumed for the first part of the course)
  - can be stopped and restarted
    - stopped when waiting for an event (e.g., completion of an I/O operation)
    - restarted with the event fires
  - can co-exist with other threads sharing a single CPU *(or multiple CPUs)*
    - a scheduler multiplexes processes over processor
    - synchronization primitives are used to ensure mutual exclusion and for waiting and signalling

Streams of Control in Asynchronous Program

Correctly ordered by procedure calls and callbacks

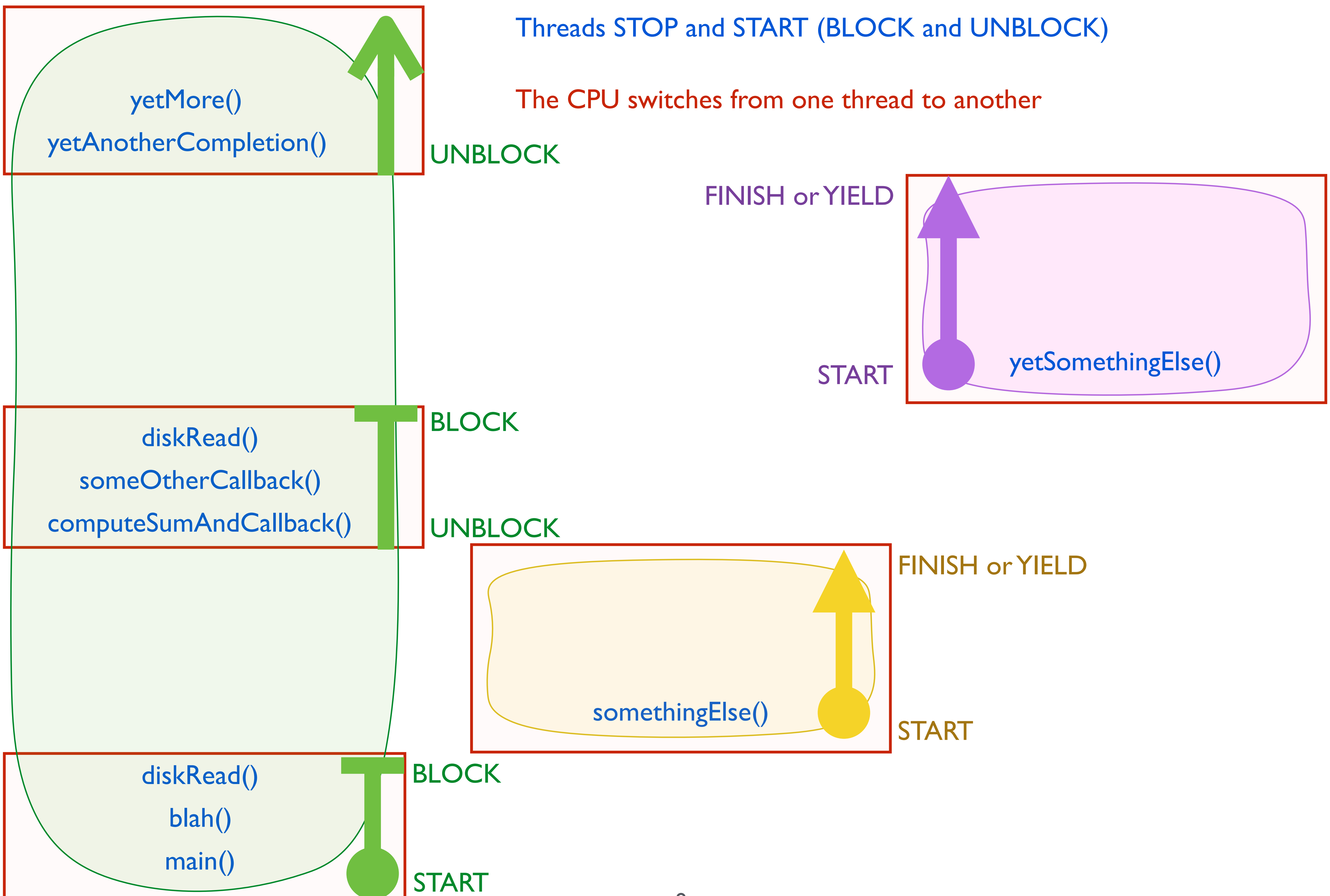
But, program order, doesn’t match “logical” order (green)



Connecting Program- and Logical-Order with Threads

Threads STOP and START (BLOCK and UNBLOCK)

The CPU switches from one thread to another



UThread: A Simple Thread System for C

- The UThread Interface file (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`** is 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 waiting
  - **`uthread_join`** join calling thread with specified other thread and get return value
  - **`uthread_detach`** indicate thread no thread will join specified thread
  - **`uthread_self`** a pointer to the TCB of the current thread
  - **`uthread_block`** block current thread
  - **`uthread_unblock`** unblock specified thread and make it runnable

Start, Stop and Join

- Create / Start
  - forks the control stream
  - like an asynchronous procedure call

```
t = uthread_create(foo, NULL);
```

- Join
  - joining blocks caller until target has finished
  - join returns result of call that created thread

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

**rtnValuePtr is return value of foo()

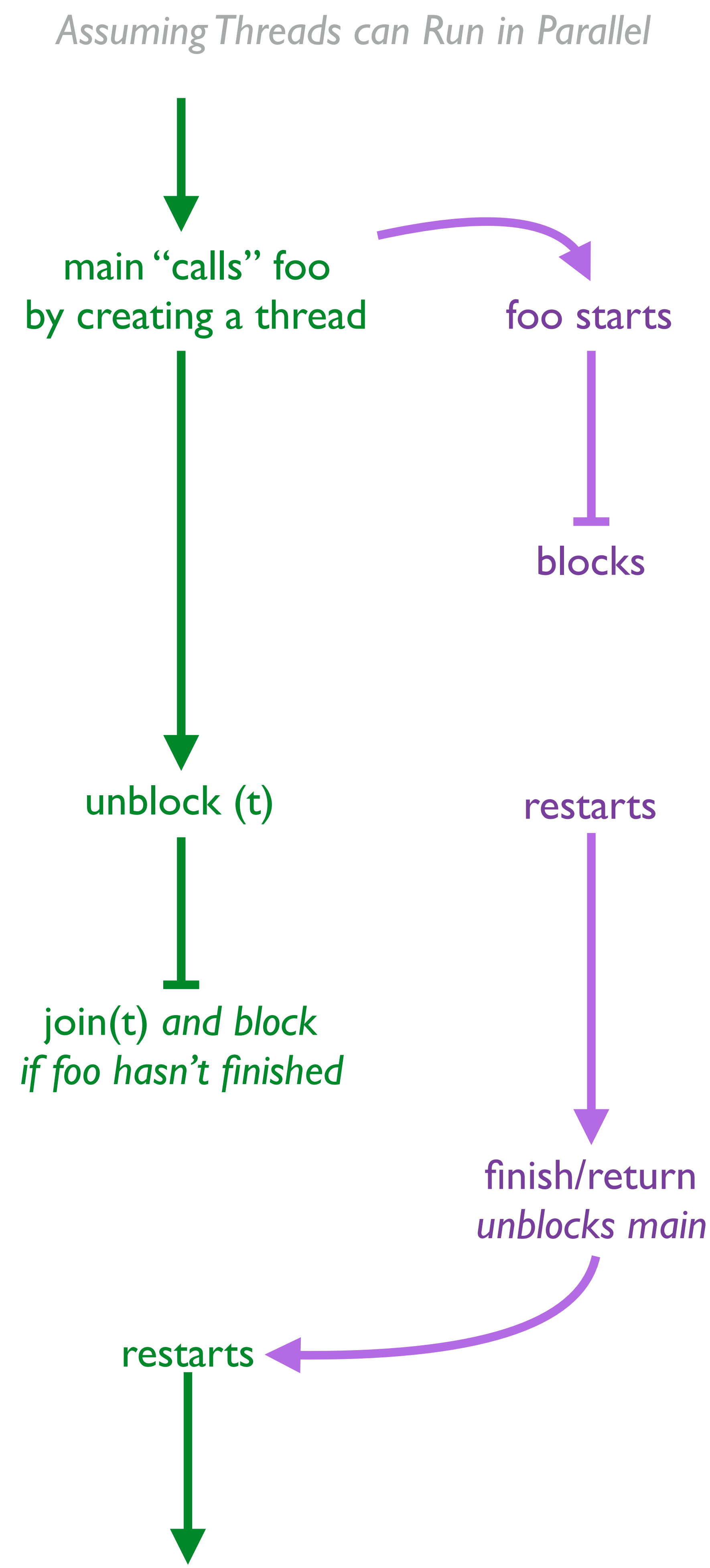
- Block / Unblock
  - stop and restart a thread (so that it can *wait*)

```
uthread_block();
```

stop calling thread until it is unblocked

```
uthread_unblock(t);
```

restart thread *t*



CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

Unit 2b – Jul 30, Aug 1

Threads

Example Program using UThreads

```
void* ping (void* v) {
  int i;
  for (i=0; i<100; i++) {
    printf ("I"); fflush(stdout);
    uthread_yield();
  }
  return 0;
}
```

give up CPU if there's another thread that can run (i.e., pong())

```
void* pong (void* v) {
  int i;
  for (i=0; i<100; i++) {
    printf ("O"); fflush(stdout);
    uthread_yield();
  }
  return 0;
}
```

give up CPU if there's another thread that can run (i.e., ping())

```
void ping_pong () {
  uthread_t t0, t1;
  uthread_init (1);
  t0 = uthread_create (ping, 0);
  t1 = uthread_create (pong, 0);
  uthread_join (t0, 0);
  uthread_join (t1, 0);
}
```

give up CPU to wait for ping() to return and thus its thread to finish

Diagram the execution on 1 CPU and on 2 …

Revisiting the Disk Read

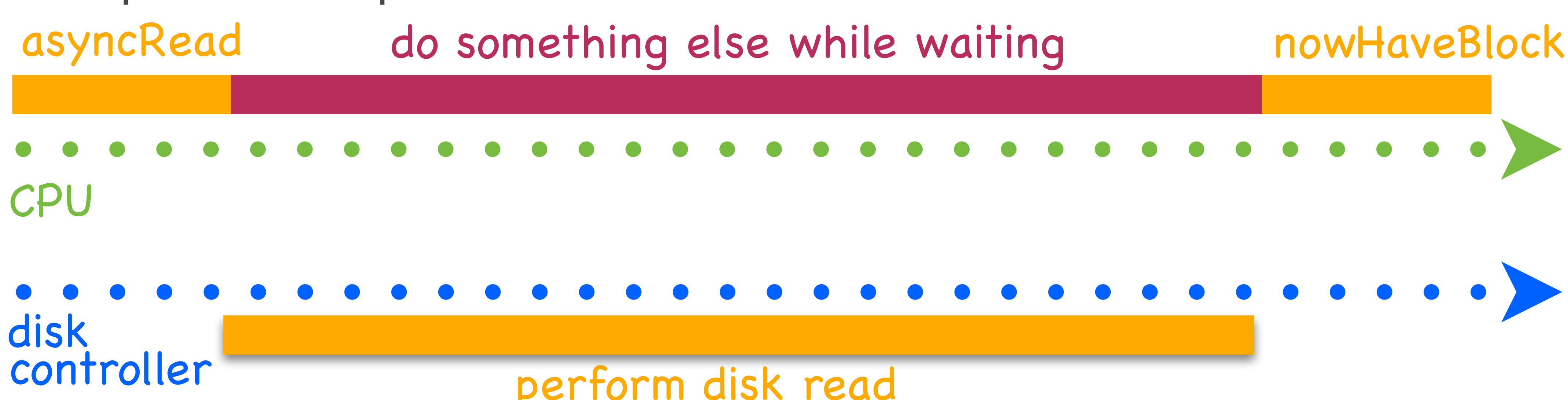
- A program that reads a block from disk
  - want the disk read to be synchronous

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

  - but, it is asynchronous so we have this

```
asyncRead        (buf, siz, blkNo, nowHaveBlock);
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 synchrony
  - disk read blocks while waiting for disk to complete
  - CPU runs other thread(s) while first thread is blocked
  - disk interrupt restarts the blocked read

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

```
interruptHandler() {
    unblockWaitingThread();
}
```

Recall Asynchronous Disk read

We wanted to do this, but couldn't because disk reads are asynchronous…

```
void foo () {
  printf ("%d\n", sumDiskBlock (1234));
}
```

```
int sumDiskBlock (int aBlkNo) {
  char buf [4096];
  int  sum;

  diskRead (buf, 4096, aBlkNo);
  for (int i=0; i<4096; i++)
    sum += buf [i];

  return sum;
}
```

We implemented it this way in event-driven programming style…

```
void foo () {
  readDiskBlock (1234, printSum);
}
```

```
void printSum (char* buf, int n) {
  printf ("%d\n", calcSum (buf, n));
}

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

Hiding Asynchrony with Threads

Now we can do this…

```
void foo () {
  printf (“%d\n”, sumDiskBlock (1234));
}
```

```
void main () {
  uthread_create (doSomethingElse, 0);
  foo();
}
```

```
int sumDiskBlock (int aBlkNo) {
  char buf [4096];
  int  sum;

  diskRead (buf, 4096, aBlkNo);
  for (int i=0; i<4096; i++)
    sum += buf [i];

  return sum;
}
```

We implemented it this way using threads to hide asynchrony…

```
struct PendingRead {
  …
  uthread_t waitingThread;
} pendingReadTh;
```

```
void diskRead (char* aBuf, int aSiz, int aBlkNo) {
  pendingRead.waitingThread = uthread_self();
  asyncRead (aBuf, aSiz, aBlkNo, unblock);
  uthread_block();
}
```

```
uthread_unblock (pendingRead.waitingThread);
```

Called from disk-completion interrupt

Draw thread timeline …

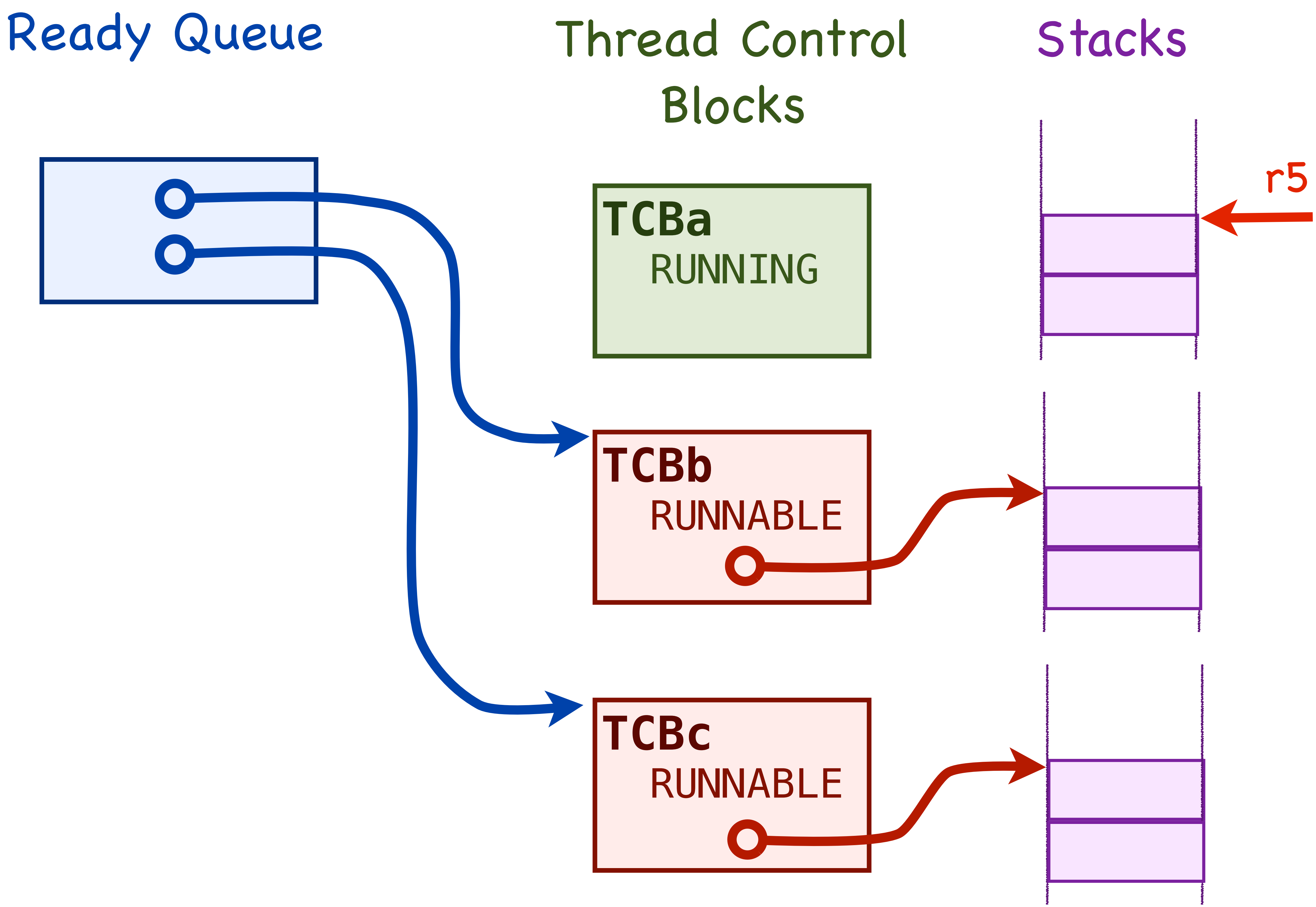
Implementing Threads: Some Questions

- The key new thing is blocking and unblocking
  - what does it mean to stop a thread?
  - what happens to the thread?
  - what happens to the physical processor?
- What data structures do we need
- What basic operations are required

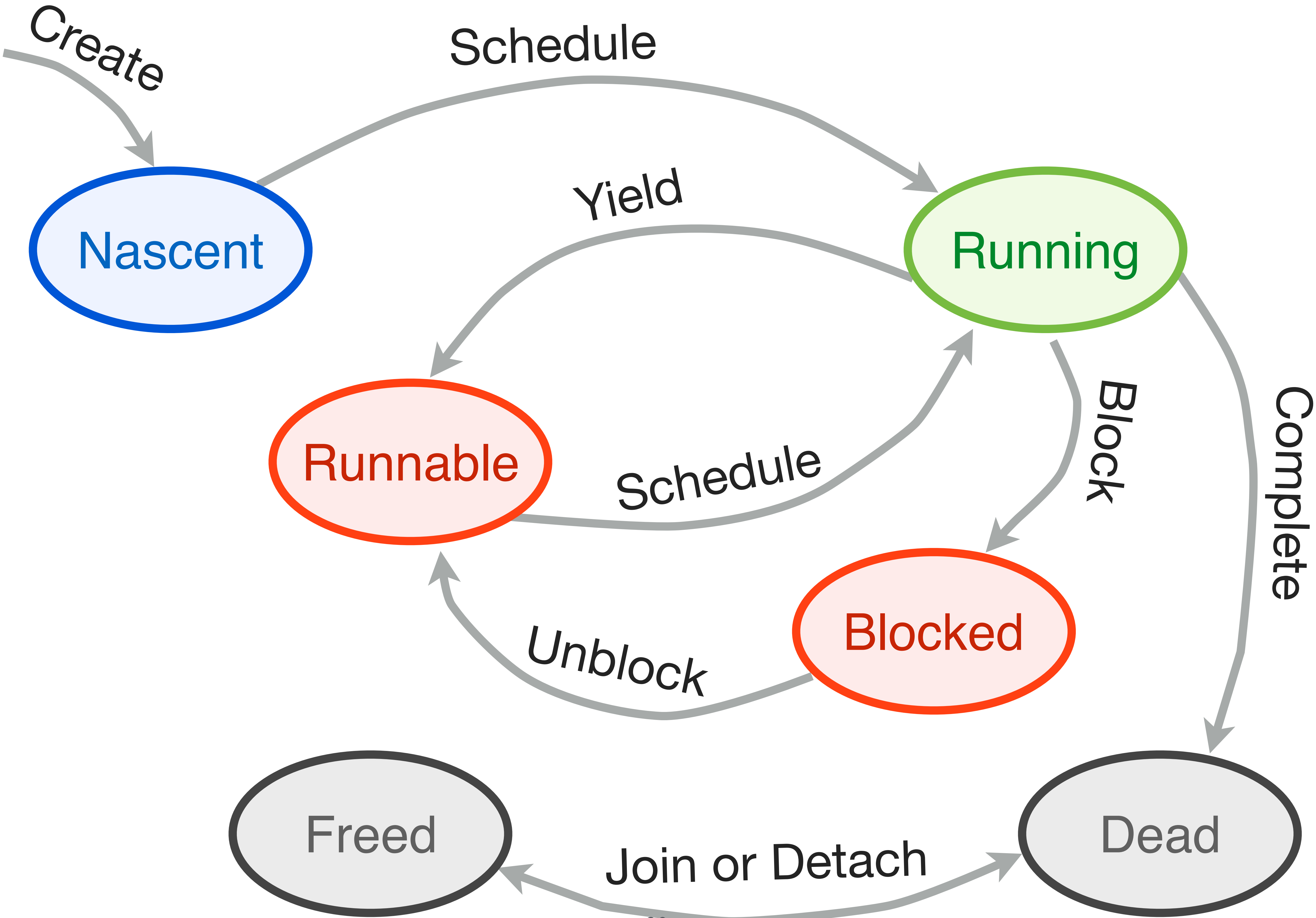
Implementing UThreads: Data Structures

- Thread State
  - *when running*: register file and runtime stack
  - *when stopped*: Thread-Control-Block object and runtime stack
- Thread-Control Block (TCB)
  - thread status: (NASCENT, RUNNING, RUNNABLE, BLOCKED, or DEAD)
  - saved value of thread's stack pointer if its not running
  - scheduling parameters such as priority, quantum, preemptablity etc.
- Ready Queue
  - list of TCB's of all RUNNABLE threads
- One or more Blocked Queues
  - list of TCB's of BLOCKED threads

Thread Data Structure Diagram



Thread Status DFA



Implementing Thread Yield

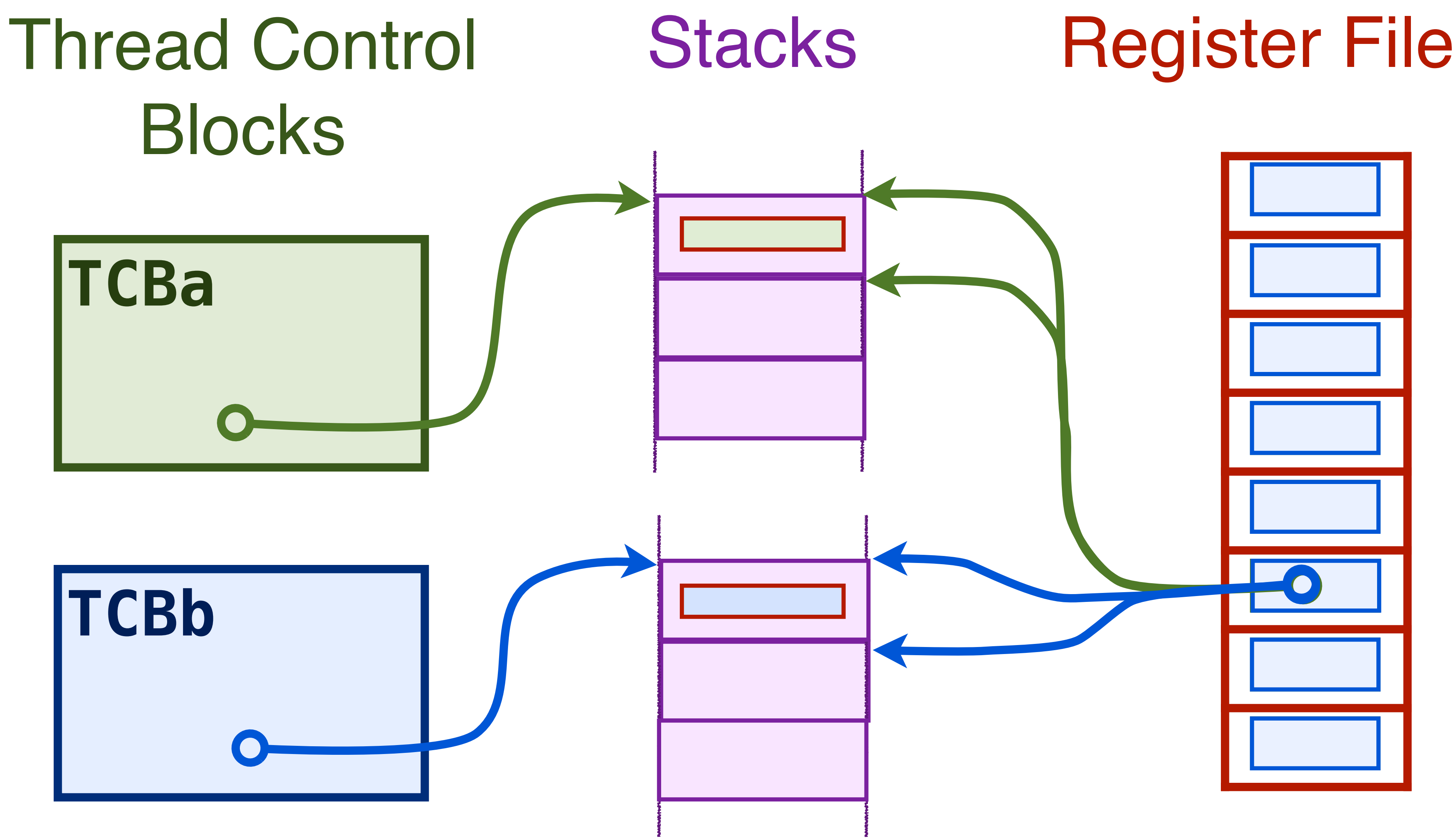
- Thread Yield
  - gets next runnable thread from ready queue (if any)
  - puts current thread on ready queue
  - switches to next thread
- Example Code

```
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 (Ta, Tb)` 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 and registers can be saved and restored to/from stack
  - thread-control block has pointer to stack pointer each thread
- Implementation
  - 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 Switch



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
4. Restore registers from B's stack

Question 2b.1

- 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) 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) I am not sure.

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
  - is the set of rules that determines which threads should be running
- Things you might consider when setting scheduling policy
  - 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
  - is a 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 threads have the same priority, run thread that has waited the longest
- Implementing Thread Mechanism
  - organize Ready Queue as a priority queue
    - highest priority first
    - FIFO (first-in-first-out) among threads of equal priority
  - priority queue: first-in-first out among equal-priority threads

Preemption

- Preemption occurs when
  - a “yield” is forced upon the current running thread
  - current thread is stopped to allow another thread to run
- Priority-based preemption
  - when a thread is made runnable (e.g., created or unblocked)
  - if it is higher priority than current-running thread, it preempts that thread
- Quantum-based preemption
  - each thread is assigned a runtime “quantum”
  - thread is preempted at the end of its quantum
- How long should quantum be?
  - disadvantage of too short?
  - disadvantage of too long?
  - typical value is around 100 ms

Implementing Quantum Preemption

- 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
  - as long as the application threads are running, the system isn't
- Timer Device
  - an I/O controller connected to a clock
  - interrupts processor at regular intervals
- Timer Interrupt Handler
  - compares the running time of current thread to its quantum
  - preempts it if quantum has expired

Summary

- Thread
  - synchronous “thread” of control in a program
  - virtual processor that can be stopped and started
  - threads are executed by real processor one at a time (per processor)
- Threads hide asynchrony
  - by stopping to wait for interrupt/event, but freeing CPU to do other things
- Thread state
  - when running: stack and machine registers (register file etc.)
  - when stopped: Thread Control Block stores stack pointer, stack stores state
- Round-robin, preemptive, priority thread scheduling
  - lower priority thread preempted by higher
  - thread preempted when its quantum expires
  - equal-priority threads get fair share of processor, in round-robin fashion