a place of mind
THE UNIVERSITY OF BRITISH COLUMBIA

CPSC 213
Introduction to Computer Systems

Unit 2b
Virtual Processors

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

Announcements

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

Overview

- Reading
  - 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

Review: issues introduced by I/O devices

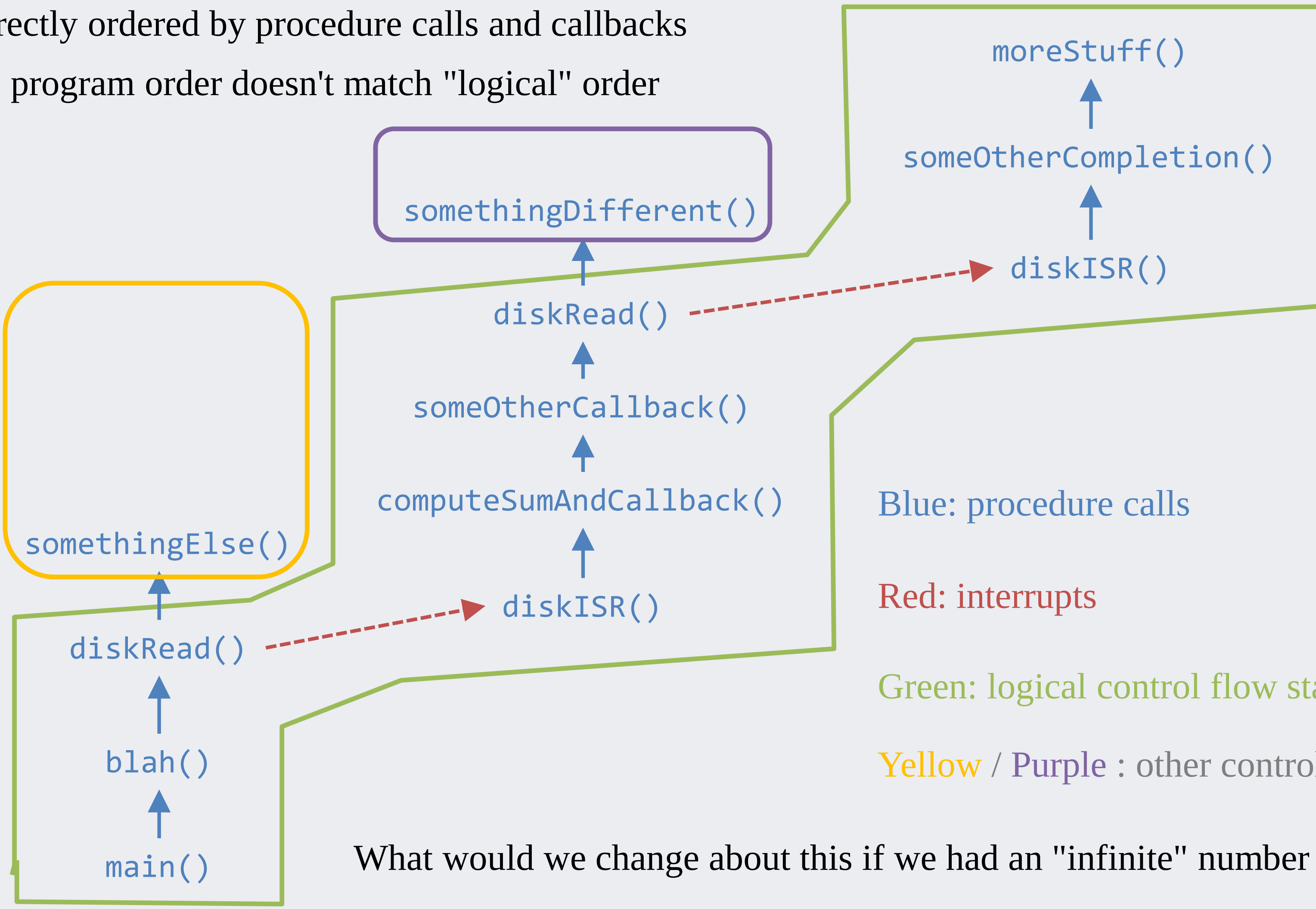
- Ordering of program events with I/O completion
  - `diskRead` triggers I/O controller to start read process
  - program has things that can only run after that process completes
- Do other things while waiting for I/O event
  - need multiple independent streams of execution in program
  - one does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

Correctly ordered by procedure calls and callbacks

But, program order doesn't match "logical" order



Blue: procedure calls

Red: interrupts

Green: logical control flow starting at main

Yellow / Purple : other control flows

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

Infinite CPUs: we can poll the device

```
moreStuff()
someOtherCompletion()
```

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?

```
diskRead()
blah()
main()
```

```
somethingDifferent()
```

```
somethingElse()
```

The Virtual Processor

- Originated with Edsger Dijkstra in "THE Operating 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.

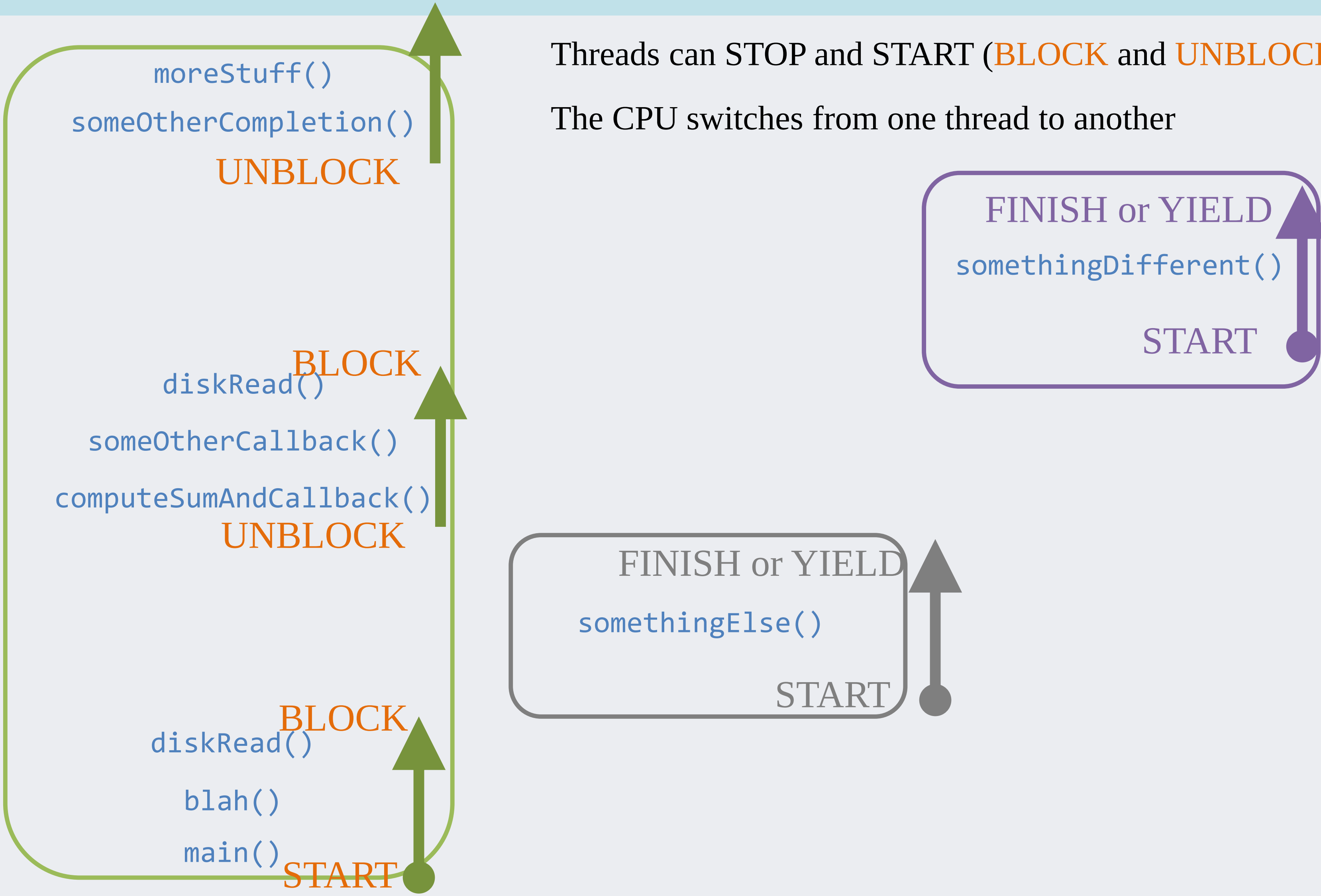
  This decision, inspired by fear, is at the bottom of what I regard as the group's main contribution to the art of system design."

- 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

Connecting program- and logical-order with threads

Threads can STOP and START (BLOCK and UNBLOCK)

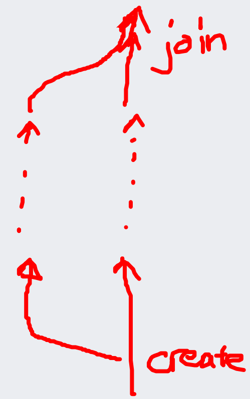
The CPU switches from one thread to another



UThread: a simple thread system for C

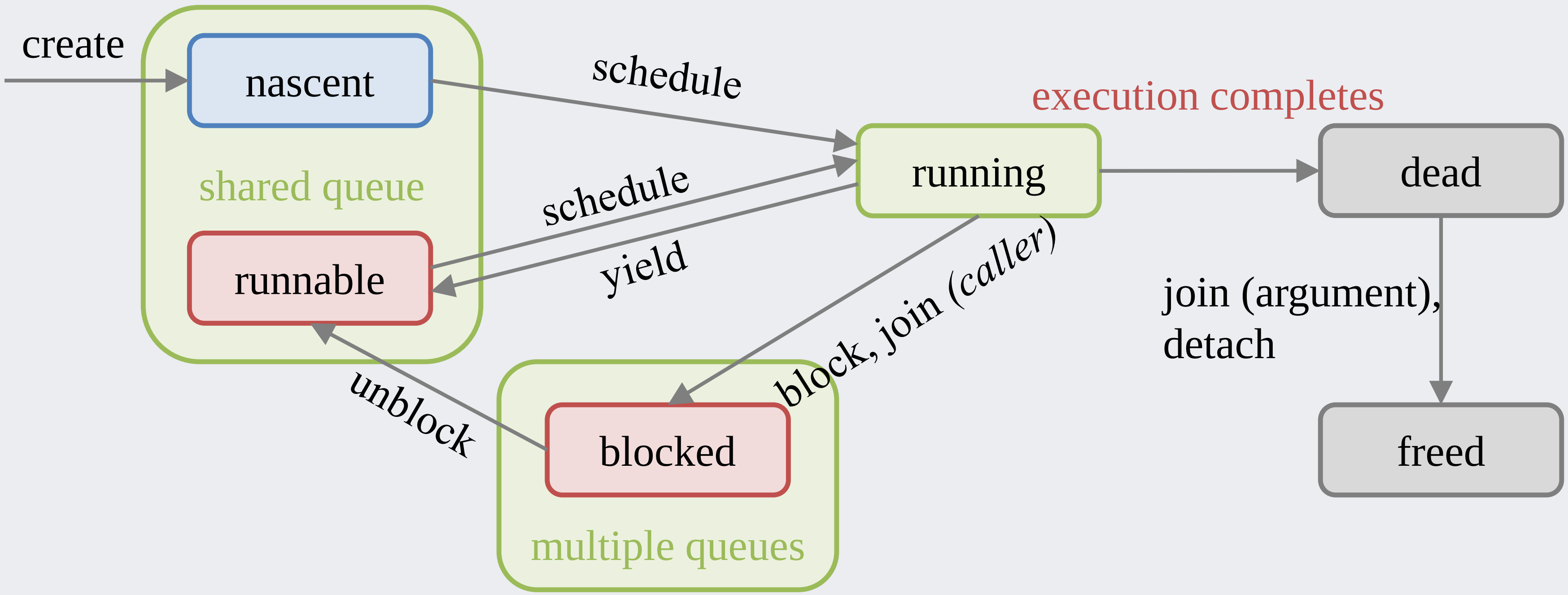
- The UThread interface (uthread.h):

```
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



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, rtnValuePtr);
```

`**rtnValuePtr` is `foo()`'s return value

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

```
uthread_block();
```

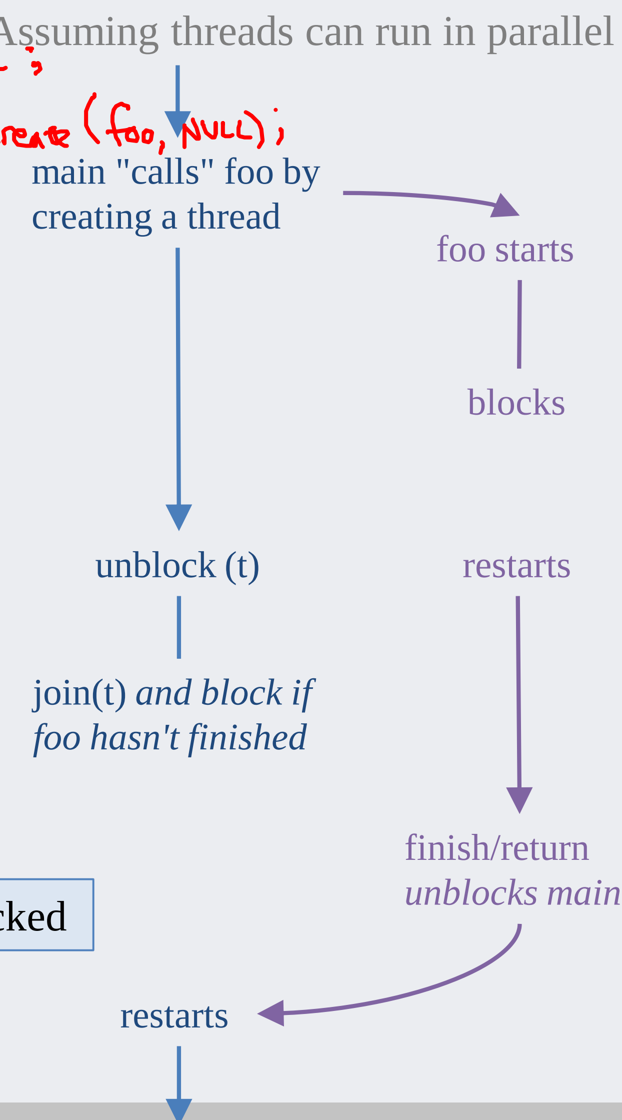
stop calling thread until it is unblocked

```
uthread_unblock(t);
```

restart thread `t`

uthread_t t;

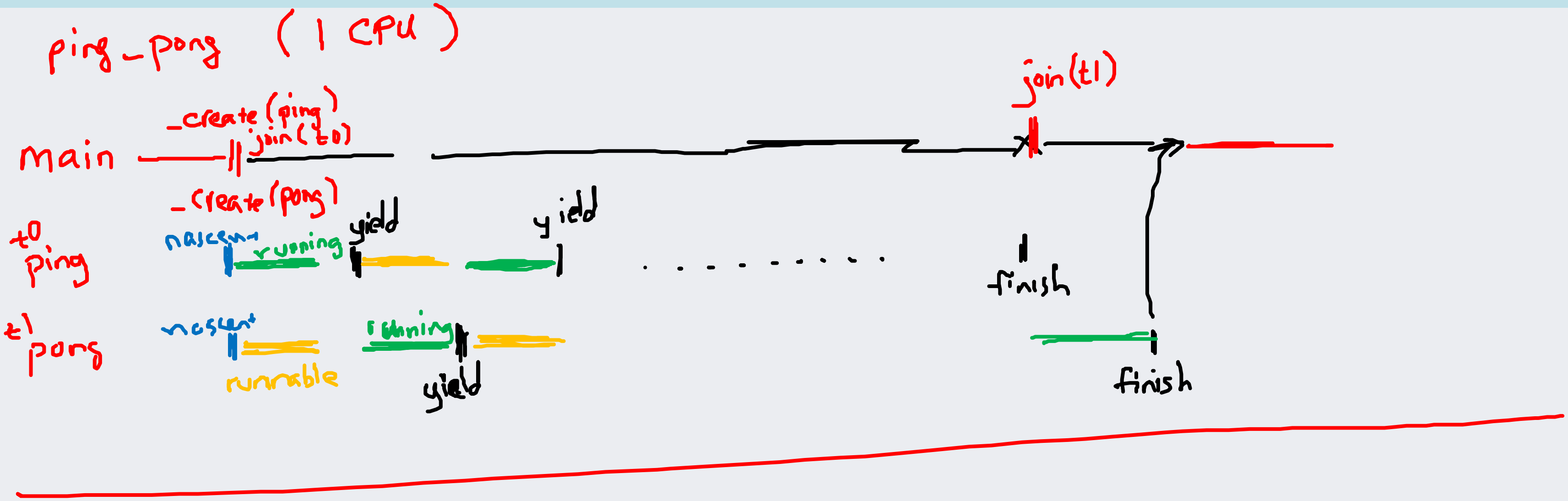
t=uthread_create(foo, NULL);



Example program using `UThreads`

```
void* ping(void* x) {
  for (int i=0; i<NUM_ITERATIONS; i++) {
    printf("|");
    uthread_yield(); // give up CPU if a thread is waiting
  }
  return NULL;
}

void* pong(void* x) {
  for (int i=0; i<NUM_ITERATIONS; i++) {
    printf(".");
    uthread_yield(); // give up CPU if a thread is waiting
  }
  return NULL;
}

int main(int argc, char* argv[]) {
  uthread_t t0, t1;

  int i;

  uthread_init(2);

  t0 = uthread_create(ping, NULL); // create thread to run ping(NULL) and start if CPU is available
  t1 = uthread_create(pong, NULL); // create thread to run pong(NULL) and start if CPU is available

  uthread_join(t0, 0); // wait until ping thread finishes
  uthread_join(t1, 0); // wait until pong thread finishes

  printf("\n");
}
```

also demo `prime.c`


ping_pong (1 CPU)
main
_create(ping)
join(t0)
_create(pong)
join(t1)
t0
ping
nascent
running
yield
yield
finish
t1
pong
nascent
runnable
running
yield
finish

Revisiting the disk read

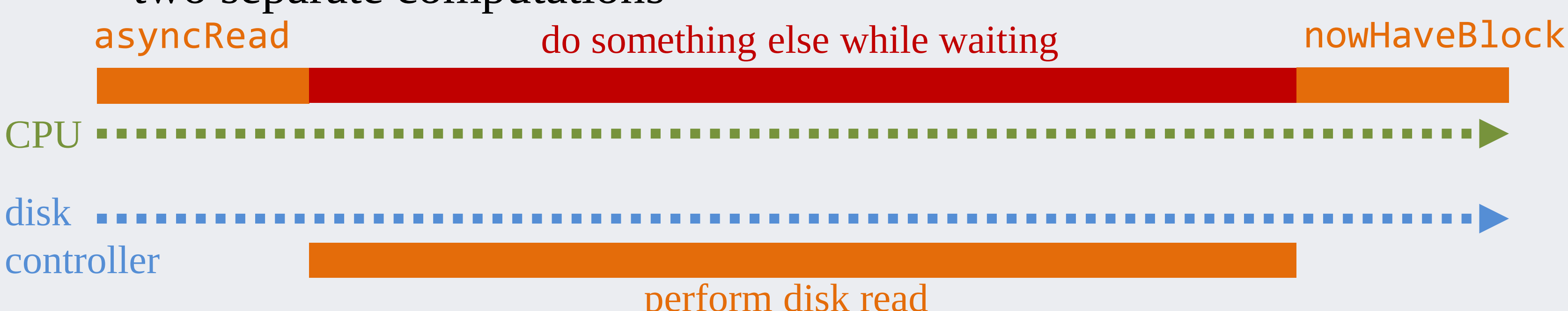
- A program that reads a block from disk
  - want the disk read to be synchronous

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

  - but it is actually asynchronous, so we had this instead:

```
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 (and its subsequent tasks)
  - 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 thread

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

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

Implementing threads

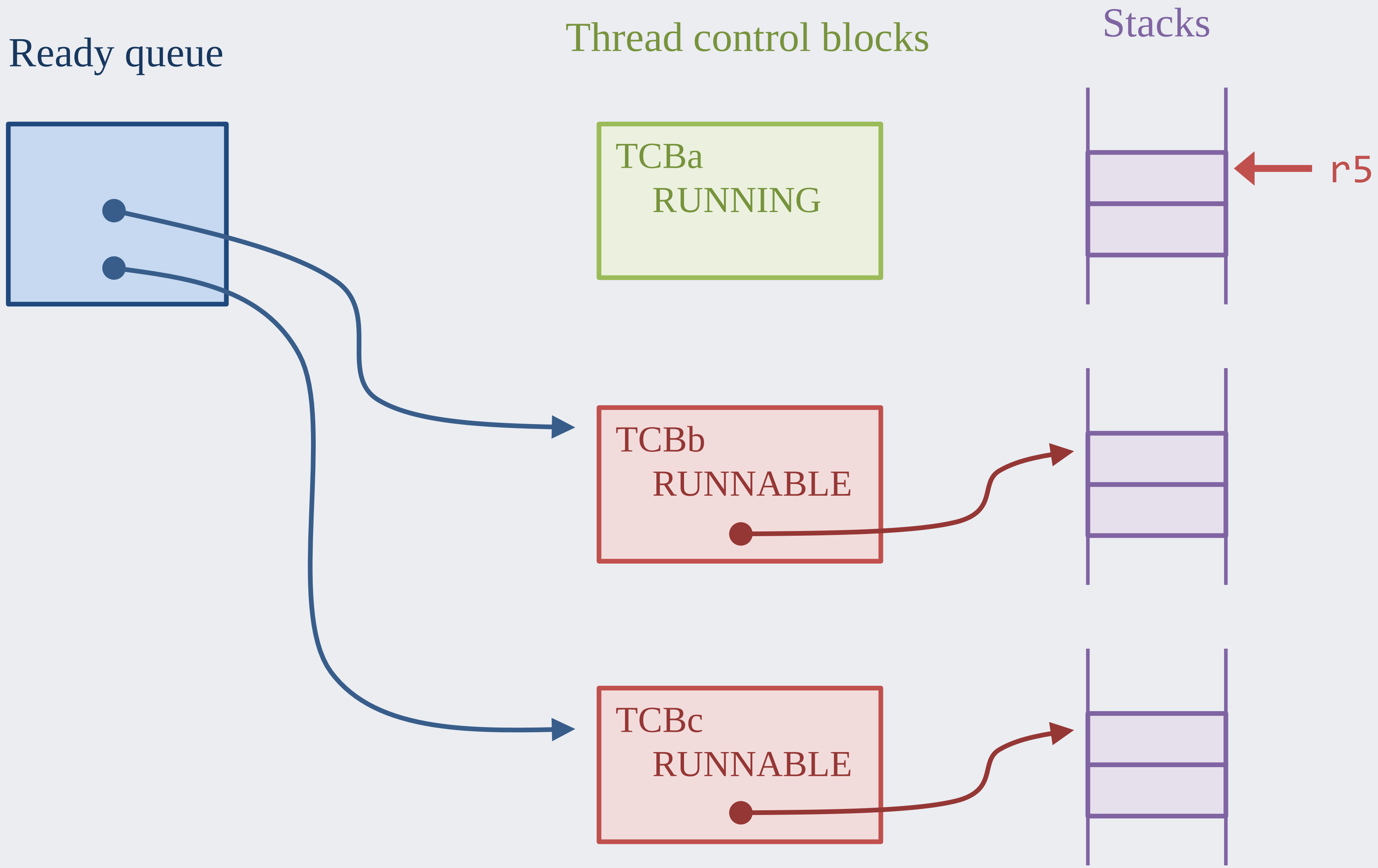
- Key concept in implementation: blocking and unblocking
  - What does it mean to stop a thread?
  - What happens to the thread?
  - What happens to the physical processor?

- Thread management
  - What data structures to 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 (TCB) object and runtime stack
- Thread-control block (TCB)
  - thread status: (NASCENT, RUNNING, RUNNABLE, BLOCKED, DEAD)
  - saved value of thread's stack pointer if it is not running
  - scheduling parameters, e.g. priority, quantum, preemptibility, etc.
- Ready queue
  - list of TCBs of all RUNNABLE threads
- One or more Blocked queues
  - list of TCBs of BLOCKED threads

Thread data structure diagram



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

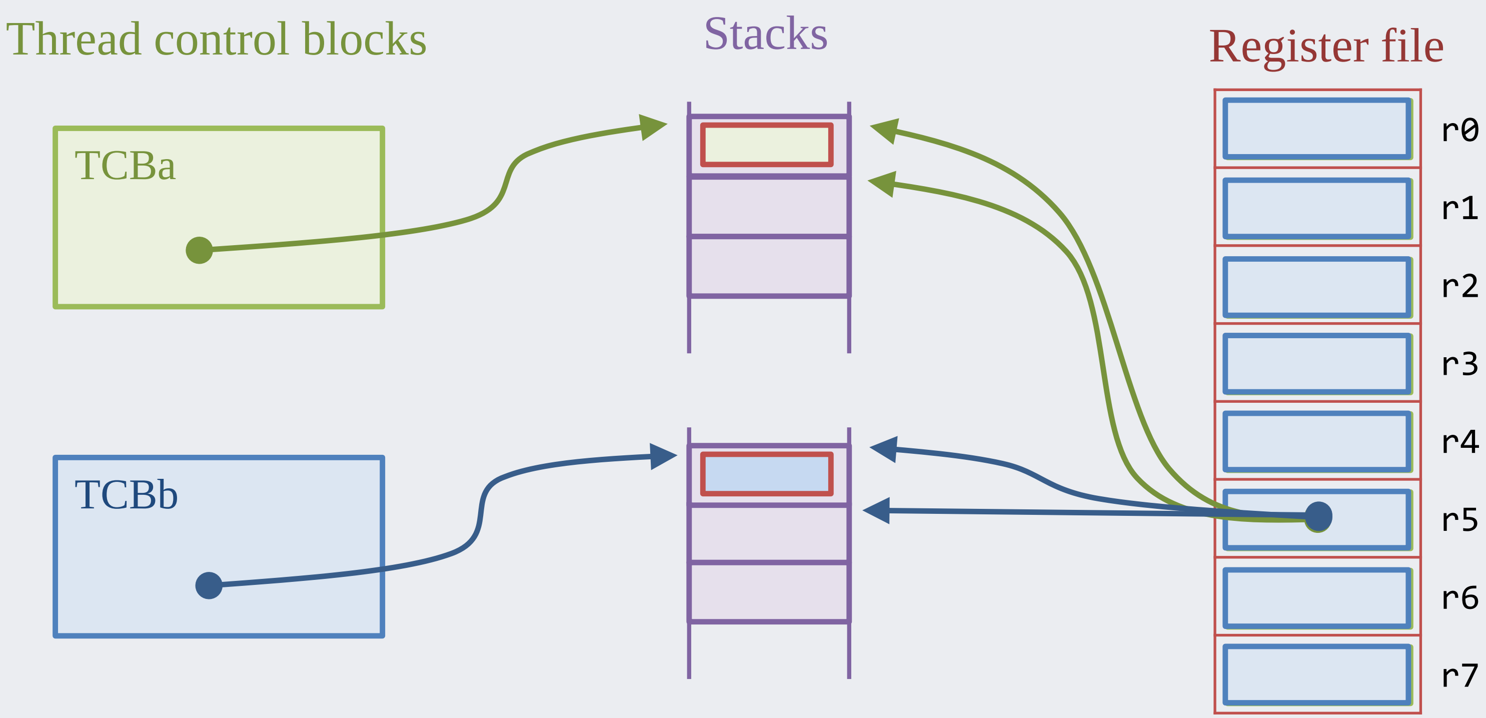
```
void uthread_yield () {
  ready_queue_enqueue (uthread_self());
  uthread_t to_thread = ready_queue_dequeue();
  assert (to_thread);
  uthread_switch (to_thread, TS_RUNNABLE);
}
```

Implementing thread switch

- Goal
  - implement a procedure `switch(`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 in T_b's TCB
  - restore registers from stack

Thread switch

T_a switching to T_b



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 for thread switch

C / x86 assembly

```
asm volatile ("pushq %%rbx"
              "pushq %%rcx"
              "pushq %%rdx"
              "pushq %%rsi"
              "pushq %%rdi"
              "pushq %%rbp"
              "pushq %%r8"
              "pushq %%r9"
              "pushq %%r10"
              "pushq %%r11"
              "pushq %%r12"
              "pushq %%r13"
              "pushq %%r14"
              "pushq %%r15"
              "pushfq"
              "movq  %%rsp, %0"
              "movq  %1, %%rsp"
```

backing up CPU state of thread A

saving stack pointer to TCB

```
"popfq"
"popq  %%r15"
"popq  %%r14"
"popq  %%r13"
"popq  %%r12"
"popq  %%r11"
"popq  %%r10"
"popq  %%r9"
"popq  %%r8"
"popq  %%rbp"
"popq  %%rdi"
"popq  %%rsi"
"popq  %%rdx"
"popq  %%rcx"
"popq  %%rbx"
: "=m" (*from_sp_p)
: "ra" (to_sp));
```

restoring saved CPU state of thread B

restore thread B's stack pointer

```
from_tcb->saved_sp ← r[sp]
r[sp]              ← to_tcb->saved_sp
```

iClicker 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. 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. ???

all have activation frames

→ main() → uthread_init() → uthread_create(foo) → foo() → uthread_yield() → uthread_switch()

Thread scheduling

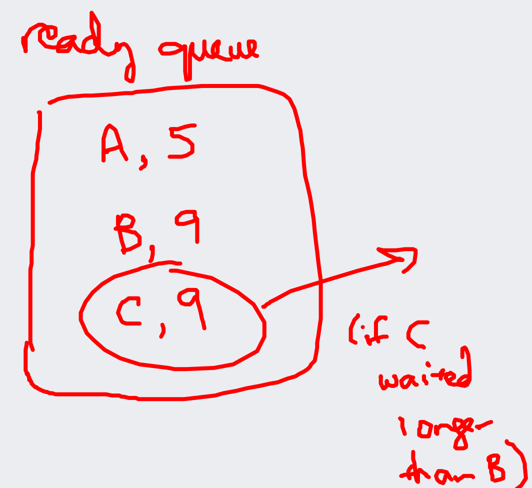
- Thread scheduling is:
  - the process of deciding when threads should run
    - when there are more runnable threads than processors
    - involves a policy and a mechanism
- Thread scheduling policy
  - is the set of rules that determines which threads should be running
- Some things to consider when 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: 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

- Implementation (mechanism)
  - organize Ready Queue as a priority queue
    - highest priority first
    - FIFO (first-in-first-out) among threads of equal priority



Pre-emption

- Pre-emption 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
  - each thread is assigned a runtime "quantum"
  - thread is pre-empted at the end of its quantum
- How long should quantum be?
  - disadvantage of too short? thread makes little progress
  - disadvantage of too long? high priority threads wait 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 isn't

- Solution: timer device
  - an I/O 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

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, pre-emptive, priority thread scheduling
  - lower priority thread pre-empted by higher priority
  - thread pre-empted when its quantum expires
  - equal-priority threads get fair share of processor, in round-robin fashion