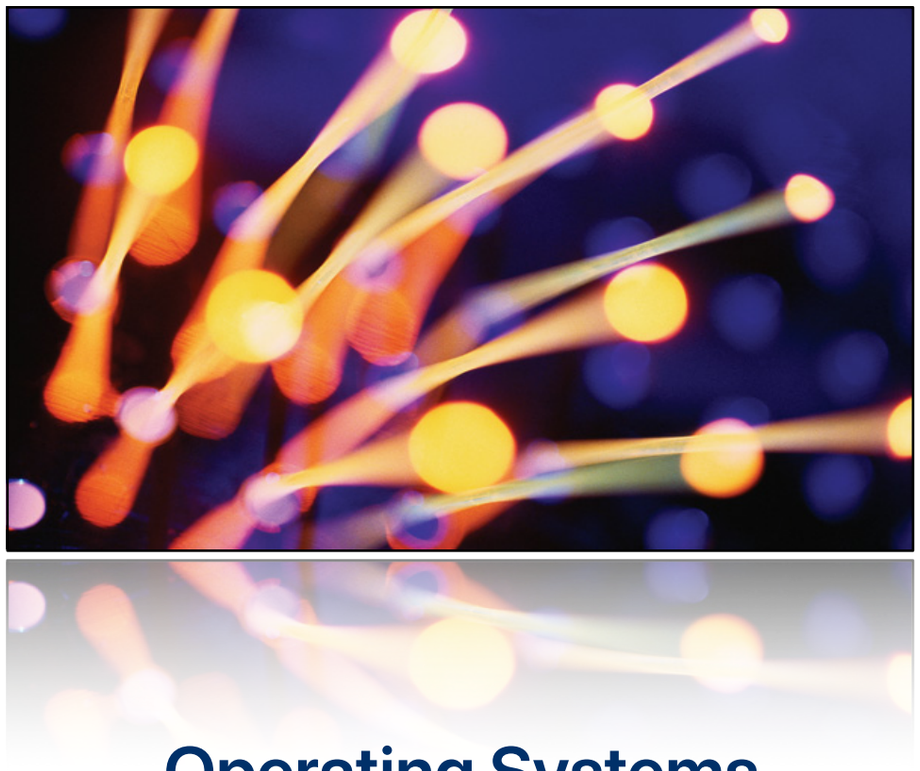
Threads and Context Switching in BLITZ



Operating Systems

Baochun Li
University of Toronto

Recall: The BLITZ Architecture

BLITZ has only one CPU, which simplifies the situation

It has two groups of 16 general-purpose integer registers, r0 to r15 — one for user mode and one for kernel mode

so that there is no need to save registers when switching to kernel mode

r0 will never need to be saved, as it is always zero

r15 is the stack pointer, used when calling (call instruction) and returning from (ret instruction) a function

The BLITZ thread scheduler: an overview

The thread manager is called “thread scheduler” in BLITZ

BLITZ uses kernel threads: the thread scheduler runs in the kernel

The thread scheduler maintains single linked lists of thread objects

- A ready list is used to select threads to run in a round-robin fashion
- A thread is executed till the next timer interrupt, at which time it is placed at the tail of the ready list
- A list of threads to be destroyed is also maintained
- A list of unused Thread objects is maintained
- It is way simpler than the Linux kernel
  - which uses doubly-linked lists and a thread object can be on multiple lists!

The Thread data structure (in KPL)

```
class Thread
   superclass Listable
   fields
     regs: array [13] of int    // space for r2 to r14
     stackTop: ptr to void      // space for r15
                                // (top of system stack)
     name: ptr to array of char
     status: int                // JUST_CREATED, READY,
                                // RUNNING, BLOCKED, UNUSED
   initialFunction: ptr to function (int)
                                // starting function
   initialArgument: int         // arguments to function
   systemStack: array [SYSTEM_STACK_SIZE] of int
                                // SYSTEM_STACK_SIZE = 1000
```

On a timer interrupt (Runtime.s)

```
TimerInterrupt:
   jmp TimerInterruptHandler

TimerInterruptHandler:
   save all int registers on the interrupted thread's
   system stack (r1 to r12)
   call _P_Thread_TimerInterruptHandler // KPL routine
   restore all int registers
   reti  // restores Status Register and PC
```

The KPL TimerInterruptHandler routine

```
TimerInterruptHandler()
  // interrupts are disabled by the processor
  // as part of the interrupt processing sequence
  currentInterruptStatus = DISABLED
  currentThread.Yield()
  currentInterruptStatus = ENABLED
```

BLITZ Implementation of Yield()

```
Yield()
  disable interrupts
  nextThread = readyList.remove()
  if nextThread
    status = READY
    readyList.AddToEnd(self)
    Run(nextThread)
  endIf
  restore interrupts
```

BLITZ Implementation of Run()

```
Run(nextThread: ptr to Thread)
  prevThread = currentThread
  currentThread = nextThread
  nextThread.status = RUNNING
  Switch(prevThread, nextThread)
  while !threadsToBeDestroyed.IsEmpty()
    th = threadsToBeDestroyed.Remove()
    th.status = UNUSED
  endWhile
```

BLITZ Implementation of Switch() (Switch.s)

```
Switch:
  save r2 to r14 in prevThread.regs
  save r15 in prevThread.stackTop
  restore r2 to r14 from nextThread.regs
  restore r15 from nextThread.stackTop
  ret
```

BLITZ Implementation of Switch()

`Switch()` changes the stack pointer (r15) to the one in `nextThread`

When it returns, it returns to a different invocation of `Run()`, in the next thread

- **`Switch()`** is only called within **`Run()`** in BLITZ

`Run()` returns to `Yield()`

`Yield()` restores interrupts

Creating a new thread in BLITZ: Fork()

```
Fork(func: ptr to function(int), arg: int)
  disable interrupts
  initialFunction = func
  initialArgument = arg
  stackTop = stackTop - 4
  *(stackTop asPtrTo int) = ThreadStartUp asInteger
  status = READY
  readyList.AddToEnd(self)
  restore interrupts
ThreadStartUp:
  call _P_Thread_ThreadStartMain
ThreadStartMain()
  enable interrupts
  mainFunc = currentThread.initialFunction
  mainFunc(currentThread.initialArgument)
  ThreadFinish()
```

Terminating a Thread in BLITZ

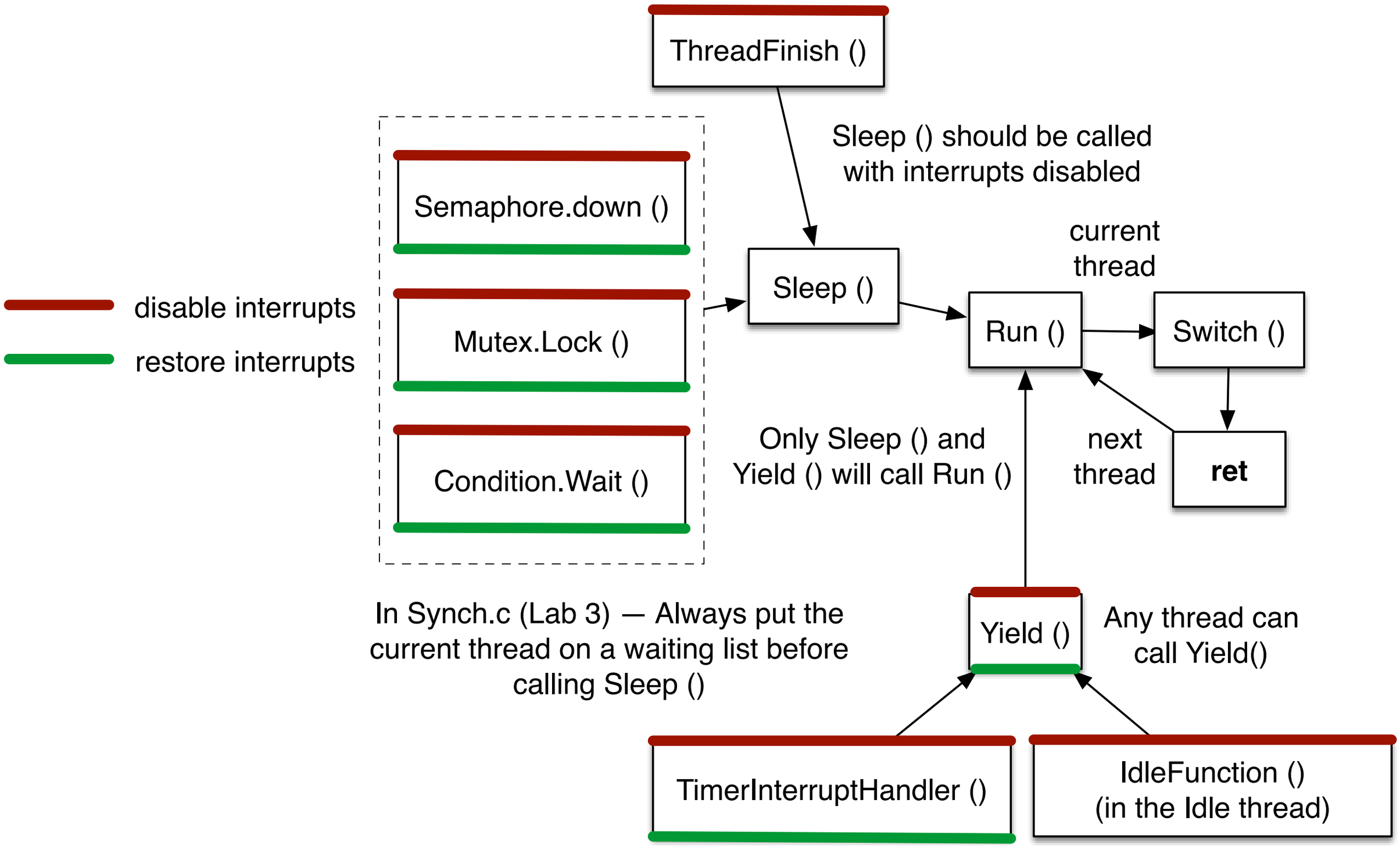
```
ThreadFinish()
  disable interrupts
  threadsToBeDestroyed.AddToEnd(currentThread)
  currentThread.Sleep()

Sleep()
  status = BLOCKED
  nextThread = readyList.Remove()
  Run(nextThread)
```

What if the readyList is empty?

The Idle Thread

```
IdleFunction()
  while true
    disable interrupts
    if readyList.empty()
      // reenable and wait for interrupts
      wait
    else
      currentThread.Yield()
    endIf
endWhile
```

The flow of calls related to context switching



What we've covered so far

BLITZ Documentation: "The Thread Scheduler and Concurrency Control Primitives," pages 1-31 on the Thread Scheduler

Lab 2 source code:

```
Synch.[c, h]
Thread.[c, h]
Runtime.s
Switch.s
```