CS 422/522 Design & Implementation of Operating Systems

Lecture 5: Concurrency and Threads

Zhong Shao
Dept. of Computer Science
Yale University

1

Motivation

- Operating systems (and application programs) often need to be able to handle multiple things happening at the same time
  - Process execution, interrupts, background tasks, system maintenance
- Humans are not very good at keeping track of multiple things happening simultaneously
- Threads are an abstraction to help bridge this gap

2

Why concurrency?

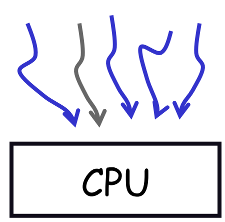
- Servers (expressing logically concurrent tasks)
  - Multiple connections handled simultaneously
- Parallel programs
  - To achieve better performance
- Programs with user interfaces
  - To achieve user responsiveness while doing computation
- Network and disk bound programs
  - To hide network/disk latency

3

The multi-threading illusion

- Each thread has its illusion of own CPU
  - yet on a uni-processor all threads share the same physical CPU!
  - How does this work?
- Two key pieces:
  - TCB --- thread control block, one per thread, holds execution state
  - dispatching loop:

```
while(1)
  interrupt thread
  save state
  get next thread
  load state, jump to it
```



4

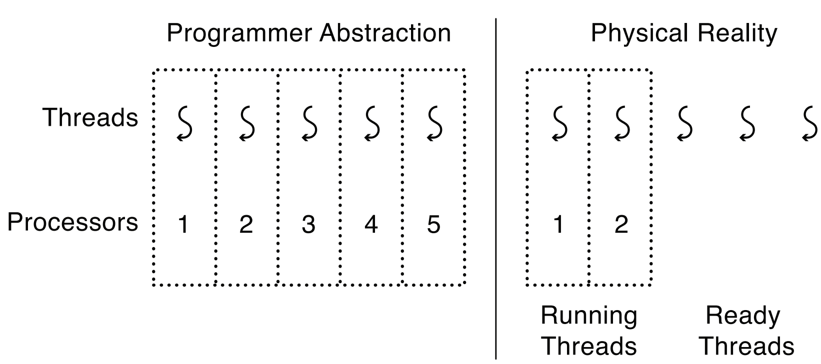
Definitions

- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time
- Protection is an orthogonal concept
  - Can have one or many threads per protection domain
  - Different processes have different privileges (& address spaces); switch OS's idea of who is running
    - switch page table, etc.
  - Problems for processes: How to share data? How to communicate?
  - The PL world does not know how to model "process" yet.

5

Thread abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule



6

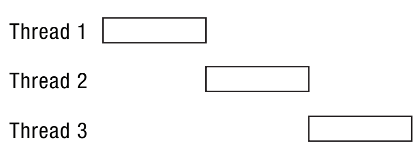
Programmer vs. processor view

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	....	Thread is resumed.
		y = y + x;	....
		z = x + 5y;	z = x + 5y;

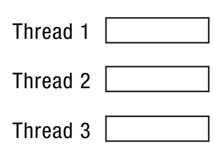
7

Possible executions

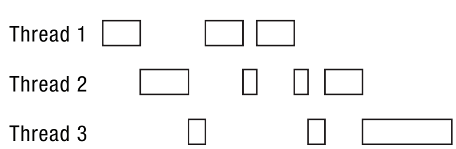
One Execution



Another Execution



Another Execution



8

Thread operations

- thread_create(thread, func, args)
  - Create a new thread to run func(args)
- thread_yield()
  - Relinquish processor voluntarily
- thread_join(thread)
  - In parent, wait for forked thread to exit, then return
- thread_exit
  - Quit thread and clean up, wake up joiner if any

9

Example: threadHello

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
  for (i = 0; i < NTHREADS; i++)
      thread_create(&threads[i], &go, i);
  for (i = 0; i < NTHREADS; i++) {
    exitValue = thread_join(threads[i]);
    printf("Thread %d returned with %ld\n", i, exitValue);
  }
  printf("Main thread done.\n");
}
void go (int n) {
  printf("Hello from thread %d\n", n);
  thread_exit(100 + n);
  // REACHED?
}
```

10

threadHello: example output

- Why must "thread returned" print in order?
- What is maximum # of threads running when thread 5 prints hello?
- Minimum?

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

11

Fork/Join concurrency

- Threads can create children, and wait for their completion
- Data only shared before fork/after join
- Examples:
  - Web server: fork a new thread for every new connection
    - As long as the threads are completely independent
  - Merge sort
  - Parallel memory copy

12

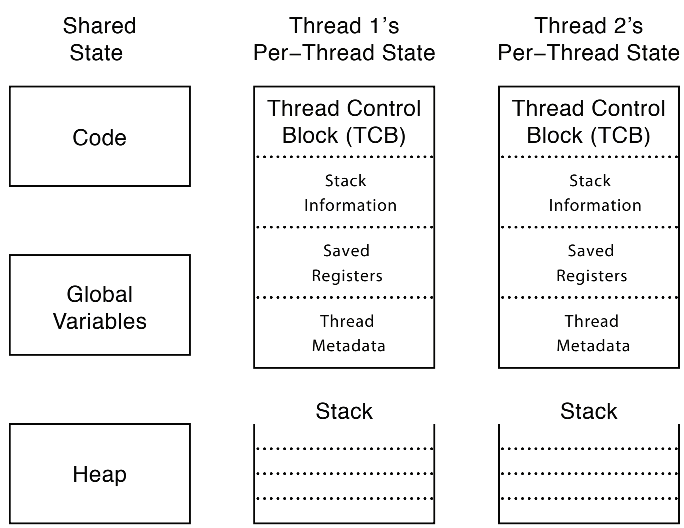
bzero with fork/join concurrency

```
void blockzero (unsigned char *p, int length) {
 int i, j;
 thread_t threads[NTHREADS];
 struct bzeroparams params[NTHREADS];

 // For simplicity, assumes length is divisible by NTHREADS.
 for (i=0, j=0; i<NTHREADS; i++, j+=length/NTHREADS) {
      params[i].buffer = p + i * length/NTHREADS;
      params[i].length = length/NTHREADS;
      thread_create_p(&(threads[i]), &go, &params[i]);
    }
 for (i = 0; i < NTHREADS; i++) {
        thread_join(threads[i]);
    }
}
```

13

Thread data structures



14

Thread context

- Can be classified into two types:
  - Private
  - Shared
- Shared state
  - Contents of memory (global variables, heap)
  - File system
- Private state
  - Program counter
  - Registers
  - Stack

15

Classifying program variables

```
int   x;                          ← global variable

void foo() {
    int   y;                      ← stack variable
    x = 1;
    y = 1;
}

main() {
    int   *p;
    p = (int *)malloc(sizeof(int));
    *p = 1;                       ← heap access
}
```

16

Classifying program variables (cont'd)

Addresses of stack variables defined at "call-time"

```
void foo() {
    int x;
    printf("%x", &x);
}
void bar() {
    int y;
    foo();
}
main() {
    foo();
    bar();
}         // different addresses will get printed
```

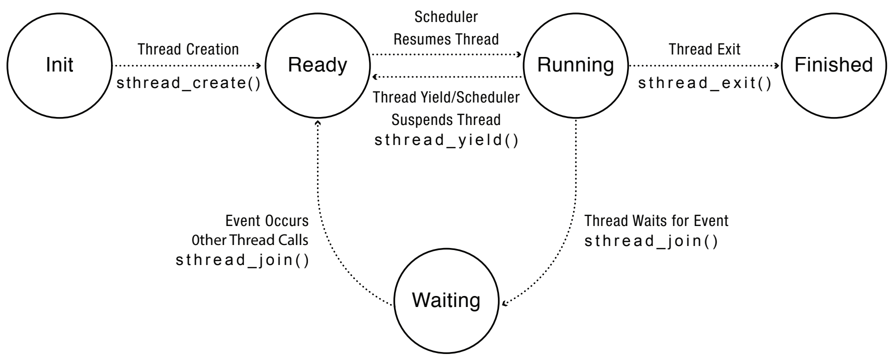
17

Thread control block (TCB)

- Current state
  - Ready: ready to run
  - Running: currently running
  - Waiting: waiting for resources
- Registers
- Status (EFLAGS)
- Program counter (EIP)
- Stack

18

Thread lifecycle



19

Implementing threads

- Thread_create(thread, func, args)
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)
- stub(func, args):
  - Call (*func)(args)
  - If return, call thread_exit()

20

Pseudo code for thread_create

```
// func is a pointer to a procedure;   arg is the argument to be passed to that procedure.
void  thread_create(thread_t *thread, void (*func)(int), int arg) {
   TCB *tcb = new TCB(); // Allocate TCB and stack

   thread->tcb = tcb;
   tcb->stack_size = INITIAL_STACK_SIZE;
   tcb->stack = new Stack(INITIAL_STACK_SIZE);

   // Initialize registers so that when thread is resumed, it will start running at stub.
   tcb->sp = tcb->stack + INITIAL_STACK_SIZE;
   tcb->pc = stub;

   // Create a stack frame by pushing stub's arguments and start address onto the stack: func, arg
   *(tcb->sp) = arg;       tcb->sp--;
   *(tcb->sp) = func;     tcb->sp--;

   // Create another stack frame so that thread_switch works correctly
   thread_dummySwitchFrame(tcb);

   tcb->state = #\readyThreadState#;
   readyList.add(tcb);    // Put tcb on ready list
}

void stub(void (*func)(int), int arg) {
   (*func)(arg);          // Execute the function func()
   thread_exit(0);        // If func() does not call exit,  call it here.
}
```

21

Thread context switch

- Voluntary
  - Thread_yield
  - Thread_join (if child is not done yet)
- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority

22

Voluntary thread context switch

- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return
- Exactly the same with kernel threads or user threads

Pseudo code for thread_switch

```
// We enter as oldThread, but we return as newThread.
// Returns with newThread's registers and stack.

void thread_switch(oldThreadTCB, newThreadTCB) {
   pushad;           // Push general register values onto the old stack.
   oldThreadTCB->sp = %esp; // Save the old thread's stack pointer.
   %esp = newThreadTCB->sp; // Switch to the new stack.
   popad;                 // Pop register values from the new stack.
   return;
}
```

Pseudo code for thread_yield

```
void thread_yield() {
  TCB *chosenTCB, *finishedTCB;

   // Prevent an interrupt from stopping us in the middle of a switch.
   disableInterrupts();

   // Choose another TCB from the ready list.
   chosenTCB = readyList.getNextThread();
   if (chosenTCB == NULL) {
      // Nothing else to run, so go back to running the original thread.
   } else {
      // Move running thread onto the ready list.
      runningThread->state = #\readyThreadState#;
      readyList.add(runningThread);
      thread_switch(runningThread, chosenTCB);     // Switch to the new thread.
      runningThread->state = #\runningThreadState#;
   }

   // Delete any threads on the finished list.
   while ((finishedTCB = finishedList->getNextThread()) != NULL) {
      delete finishedTCB->stack;
      delete finishedTCB;
   }
   enableInterrupts();
}
```

25

A subtlety

- Thread_create puts new thread on ready list
- When it first runs, some thread calls thread_switch
  - Saves old thread state to stack
  - Restores new thread state from stack
- Set up new thread's stack as if it had saved its state in thread_switch
  - "returns" to stub at base of stack to run func

26

Pseudo code for dummySwitchFrame

```
// thread_create must put a dummy frame at the top of its stack:
// the return PC & space for pushad to have stored a copy of the
// registers. This way, when someone switches to a newly created
// thread, the last two lines of thread_switch work correctly.

void thread_dummySwitchFrame(newThread) {

   *(tcb->sp) = stub;      // Return to the beginning of stub.
   tcb->sp--;
   tcb->sp -= SizeOfPopad;

}
```

27

Two threads call Yield

Thread 1's instructions	Thread 2's instructions	Processor's instructions
"return" from thread_switch into stub		"return" from thread_switch into stub
call go		call go
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	"return" from thread_switch into stub	"return" from thread_switch into stub
	call go	call go
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch

28

Involuntary thread switch

- Timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version
  - End of interrupt handler calls switch()
  - When resumed, return from handler resumes kernel thread or user process
  - Thus, processor context is saved/restored twice (once by interrupt handler, once by thread switch)

29

A quick recap

- Thread = pointer to instruction + state
- Process = thread + address space + OS env (open files, etc.)
- Thread encapsulates concurrency; address space encapsulates protection
- Key aspects:
  - per-thread state
  - picking a thread to run
  - switching between threads
- The Future:
  - how to share state among threads?
  - how to pick the right thread/process to run?
  - how to communicate between two processes?

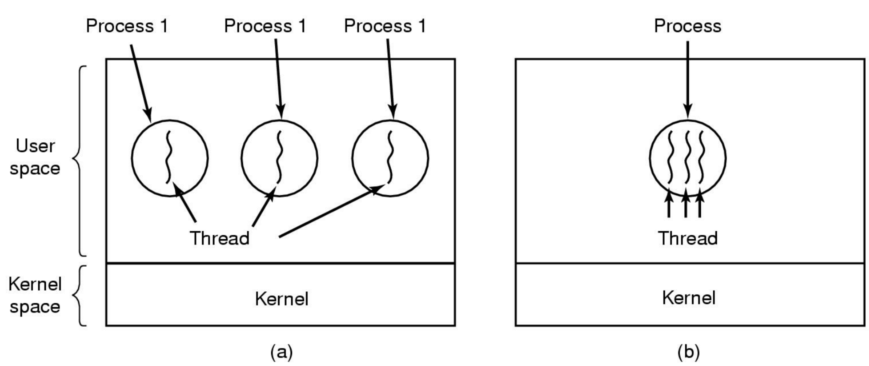
30

Threads in the kernel and at user-level

- Multi-threaded kernel
  - multiple threads, sharing kernel data structures, capable of using privileged instructions
- Multiprocess kernel
  - Multiple single-threaded processes
  - System calls access shared kernel data structures
- Multiple multi-threaded user processes
  - Each with multiple threads, sharing same data structures, isolated from other user processes

31

Threads revisited



(a) Three processes each with one thread
(b) One process with three threads

32

Implementation of processes

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Potential fields of a PCB

33

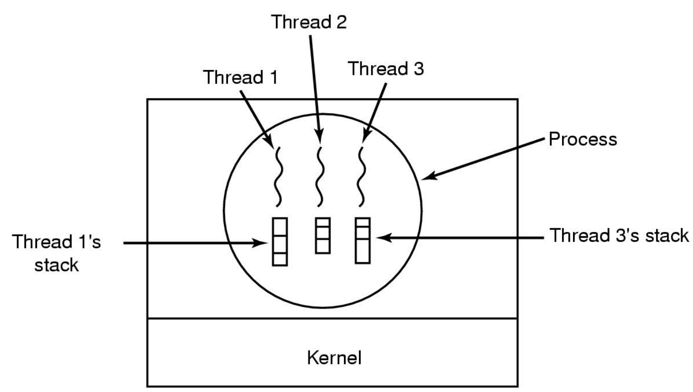
Implementation of processes (cont' d)

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Skeleton of what lowest level of OS does when an interrupt occurs

34

Threads (cont' d)



Each thread has its own stack

35

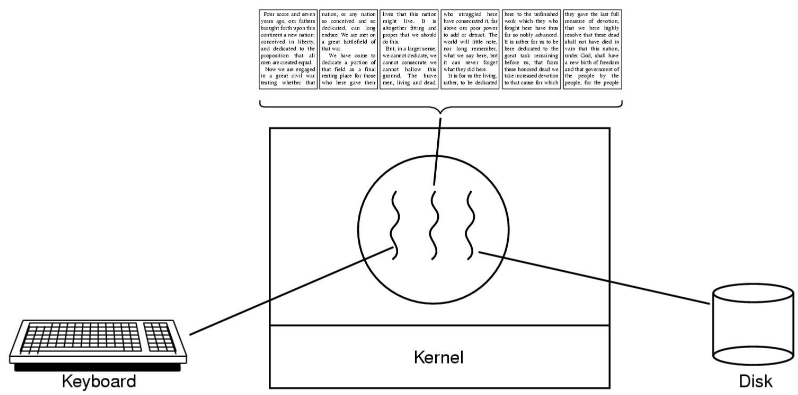
Threads (cont' d)

Per process items	**Per thread items**
Address space Global variables Open files Child processes Pending alarms Signals and signal handlers Accounting information	Program counter Registers Stack State

- Items shared by all threads in a process
- Items private to each thread

36

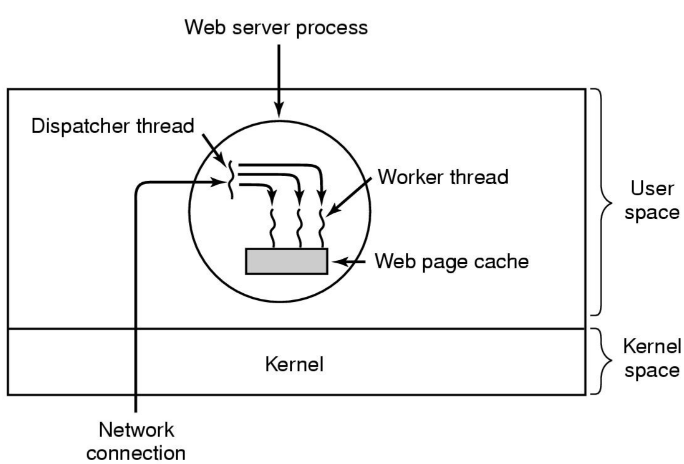
Thread usage



A word processor with three threads

37

Thread usage (cont’d)



A multithreaded Web server

38

Thread usage (cont' d)

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
       read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

- Rough outline of code for previous slide
  (a) Dispatcher thread
  (b) Worker thread

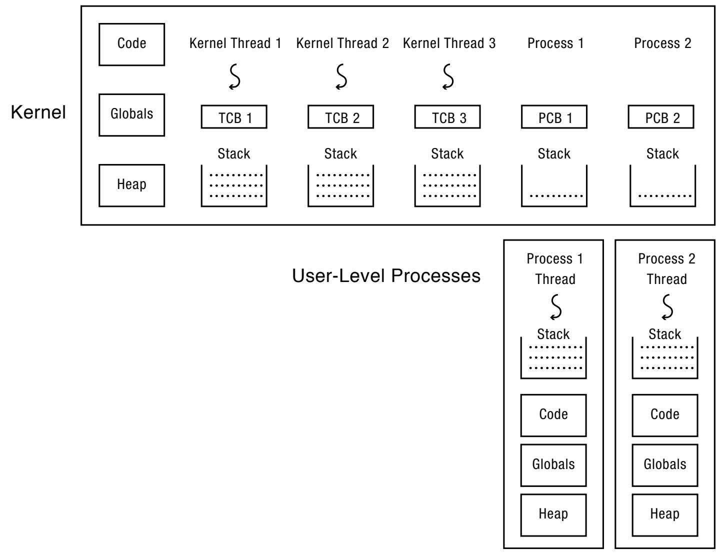
39

Implementing threads: roadmap

- Kernel threads
  - Thread abstraction only available to kernel
  - To the kernel, a kernel thread and a single threaded user process look quite similar
- Multithreaded processes using kernel threads (Linux, MacOS)
  - Kernel thread operations available via syscall
- User-level threads
  - Thread operations without system calls

40

Multithreaded OS Kernel



41

Faster thread/process switch

- What happens on a timer (or other) interrupt?
  - Interrupt handler saves state of interrupted thread
  - Decides to run a new thread
  - Throw away current state of interrupt handler!
  - Instead, set saved stack pointer to trapframe
  - Restore state of new thread
  - On resume, pops trapframe to restore interrupted thread

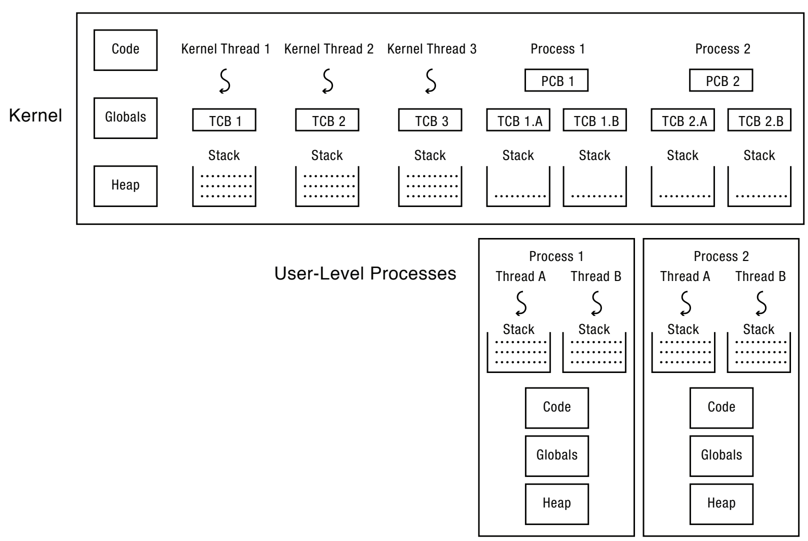
42

Multithreaded user processes (Take 1)

- User thread = kernel thread (Linux, MacOS)
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switch
  - Simple, but a lot of transitions between user and kernel mode

43

Multithreaded user processes (Take 1)



44

Multithreaded user processes (Take 2)

- Green threads (early Java)
  - User-level library, within a single-threaded process
  - Library does thread context switch
  - Preemption via upcall/UNIX signal on timer interrupt
  - Use multiple processes for parallelism
    - Shared memory region mapped into each process

45

Multithreaded user processes (Take 3)

- Scheduler activations (Windows 8)
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - Thread library decides what thread to run next
  - Upcall whenever kernel needs a user-level scheduling decision
    - Process assigned a new processor
    - Processor removed from process
    - System call blocks in kernel

46