CPSC 213

Introduction to Computer Systems

Unit 2b

Threads

Reading

- Text
  - *Concurrent Programming with Threads*
  - 2ed: 12.3
  - 1ed: 13.3

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."*

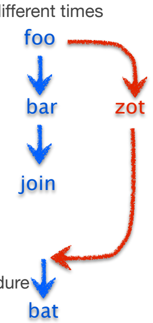
- The Thread (as we now call it)
  - a single thread of synchronous execution of a program
    - the illusion of a single system such as the Simple Machine
  - can be stopped and restarted
    - stopped when waiting for an event (e.g., completion of an I/O operation)
    - restarted with the event fires
  - can co-exist with other processes sharing a single hardware processor
    - a scheduler multiplexes processes over processor
    - synchronization primitives are used to ensure mutual exclusion and for waiting and signalling

Illusion of Synchrony

- Multiple things co-existing on the same physical CPU
  - disk reads as motivation (huge disk/CPU speed mismatch)
  - supporting this illusion is a core purpose of operating system
    - *scheduler* decides what thing to run next
- Threads
  - multiple flows within a single program
  - example use: loading big file while maintaining responsive user interface
- Processes
  - multiple programs running on single CPU
  - example use: email and browser and game and debugger
  - more on how we manage to do this later (with virtual memory)
- Multiprocessor systems
  - multiple CPUs
    - each CPU can have multiple processes, each process can have multiple threads

Thread

- An abstraction for execution
  - looks to programmer like a sequential flow of execution, a private CPU
  - it can be stopped and started, it is sometimes running and sometimes not
  - the physical CPU thus now multiplexes multiple threads at different times
- Creating and starting a thread
  - like an asynchronous procedure call
  - starts a new thread of control to execute a procedure
- Stopping and re-starting a thread
  - stopping a thread is called *blocking*
  - a blocked thread can be re-started (i.e., *unblocked)*
- Joining with a thread
  - blocks the calling thread until a target thread completes
  - returns the return value of the target-thread's starting procedure
  - turns thread create back into a synchronous procedure call



Revisiting the Disk Read

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

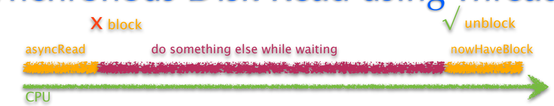
```
read         (buf, siz, blkNo);  // read siz bytes at blkNo into buf
nowHaveBlock (buf, siz);         // now do something with the block
```

  - 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);
waitForInterrupt ();
nowHaveBlock     (buf, siz);
```

```
interruptHandler() {
  signalBlockedThread();
}
```


Threads in Java

- Create a procedure that can be executed by a thread
  - build a class that implements the Runnable interface

```
class ZotRunnable implements Runnable {
  Integer result, arg;
  ZotRunnable (Integer anArg) {
    arg = anArg;
  }
  public void run() {
    result = zot (arg);
  }
}
```

- Create a thread to execute the procedure and start it

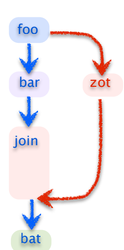
```
ZotRunnable zot = new ZotRunnable (0);
Thread t        = new Thread      (zot);
t.start();
```


- Later join with thread to get zot's return value

```
Integer result;
try {
  t.join();
  result = zot.result;
} catch (InterruptedException ie) {
  result = null;
}
```

- So that the entire calling sequence is

```
foo();
ZotRunnable zot = new ZotRunnable (0);
Thread t        = new Thread (zot);
t.start();
bar();
Integer result = null;
try {
  t.join();
  result = zot.result;
} catch (InterruptedException ie) {
}
bat();
```



UThread: A Simple Thread System for C

- The UThread Interface file (uthread.h)

```
struct uthread_TCB;
typedef struct uthread_TCB uthread_t;

void       uthread_init   (int num_processors);
uthread_t* uthread_create (void* (*star_proc)(void*), void* start_arg);
void       uthread_yield  ();
void*      uthread_join   (uthread_t* thread);
void       uthread_detach (uthread_t* thread);
uthread_t* uthread_self   ();
```

- Explained
  - uthread_t — is the datatype of a thread control block
  - 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
  - uthread_detach — indicate no thread will join specified thread
  - uthread_self — a pointer to the TCB of the current thread

Example Program using UThreads

```
void ping () {
  int i;
  for (i=0; i<100; i++) {
    printf ("ping %d\n",i); fflush (stdout);
    uthread_yield ();
  }
}
```

```
void pong () {
  int i;
  for (i=0; i<100; i++) {
    printf ("pong %d\n",i); fflush (stdout);
    uthread_yield ();
  }
}
```

```
void ping_pong () {
  uthread_init   (1);
  uthread_create (ping, 0);
  uthread_create (pong, 0);
  while (1)
    uthread_yield ();
}
```


Example: Yield vs Join

```
void ping () {
  int i;
  for (i=0; i<100; i++) {
    printf ("ping %d\n",i); fflush (stdout);
    uthread_yield ();
  }
}
```

```
void pong () {
  int i;
  for (i=0; i<100; i++) {
    printf ("pong %d\n",i); fflush (stdout);
    uthread_yield ();
  }
}
```

```
void ping_pong () {
  uthread_init   (1);
  uthread_create (ping, 0);
  uthread_create (pong, 0);
  while (1)
    uthread_yield ();
}
```

```
void ping_pong () {
  uthread_init   (2);
  uthread_create (ping, 0);
  uthread_create (pong, 0);
  uthread_join (ping_thread);
  uthread_join (pong_thread);
}
```

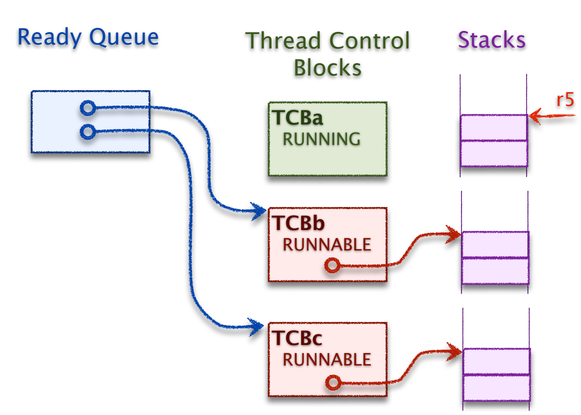

Implement Threads: Some Questions

- 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

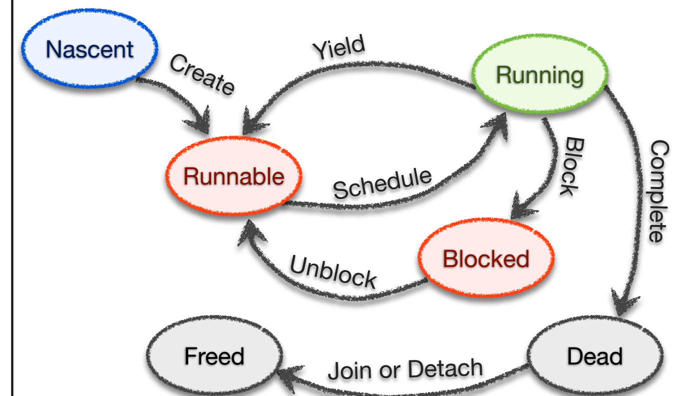
Implementing UThreads: Data Structures

- Thread State
  - running: register file and runtime stack
  - stopped: Thread Control Block and runtime stack
- Thread-Control Block (TCB)
  - thread status: (NASCENT, RUNNING, RUNNABLE, BLOCKED, or DEAD)
  - pointers to thread's stack base and top of its stack
  - scheduling parameters such as priority, quantum, pre-emptability etc.
- Ready/Runnable 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 State Machine



Threads, Queues, and Execution Order

- Queue is confusing name!
  - **scheduler may choose what to run next in any order**
  - safest to think of these as sets
- Using threads: create/join revisited
  - create
    - starts new thread, immediately adds to queue of threads waiting to run
  - join
    - blocks calling thread until target thread completes
- Do not assume order of execution!
  - order of joining is not necessarily order of execution
    - thread joins runnable queue with create call, not with join call
  - order of creating is not necessarily order of execution
    - scheduler may choose what to run next in any order
  - nondeterministic results mean threads often difficult to debug
    - uthreads deterministic if uthread_init(1), nondeterministic with more simulated processors

Question: Execution Order

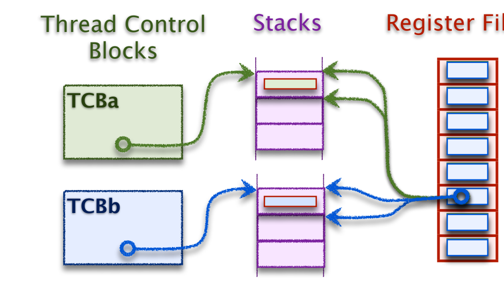
```
void* A (void* x) { printf ("A");fflush(stdout);}
void* B (void* x) { printf ("B");fflush(stdout);}
void* C (void* x) { printf ("C");fflush(stdout);}

int main (int* argc, char** argv) {
   uthread_t *C_thread, *B_thread, *A_thread;
   int i;
   uthread_init (10);
   printf ("D"); fflush(stdout);
   B_thread = uthread_create (B, 0);
   for (i=0; i<10000; i++) {}
   printf ("E"); fflush(stdout);
   C_thread = uthread_create (C, 0);
   for (i=0; i<10000; i++) {}
   printf ("F"); fflush(stdout);
   A_thread = uthread_create (A, 0);
   printf ("G"); fflush(stdout);
   uthread_join(A_thread);
   printf ("H"); fflush(stdout);
   uthread_join(B_thread);
   printf ("I"); fflush(stdout);
   uthread_join(C_thread);
   printf ("J"); fflush(stdout);
}
```

- Which sequences possible?
  - 1. ABCDEFGHIJ
  - 2. DBECFAJHGI
  - 3. DEFCBAGHIJ
  - 4. CBADEFGHIJ
  - 5.
  - 6. DBECFAGHIJ
  - 7. DEFGAHBICJ
  - 8. DEFGHIJABC

Implementing Threads: 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
  - example ...
- 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 top of stack 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



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

Example Code for Thread Switch

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

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

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

- Example for concreteness
  - you are not expected to understand Intel assembly...)

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 () {
  uthread_t* to_thread   = dequeue (&ready_queue);
  uthread_t* from_thread = uthread_cur_thread ();
  if (to_thread) {
    from_thread->state = TS_RUNABLE;
    enqueue (&ready_queue, from_thread);
    uthread_switch (to_thread);
  }
}
```


Question

- The uthread_switch procedure saves the *from* thread's registers to the stack, switches to the *to* thread's stack pointer and restores its registers from the stack, but what does it do with the program counter?
  - (A) It saves the *from* thread's program counter to the stack and restores the *to* thread's program counter from the stack.
  - (B) It saves the *from* thread's program counter to its thread control block.
  - (C) It does not need to change the program counter 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 Switching and the PC

- every thread switches in the same procedure: uthread_switch
  - thus PC of every thread in blocked or ready queue is same
    - instruction right after the one that changes stack pointer in uthread_switch
- every thread calls this procedure from different spot in application code
  - thus PC of caller already saved on stack as part of procedure call setup
  - no need to do any extra work
- enter switch on one stack, leave switch on another

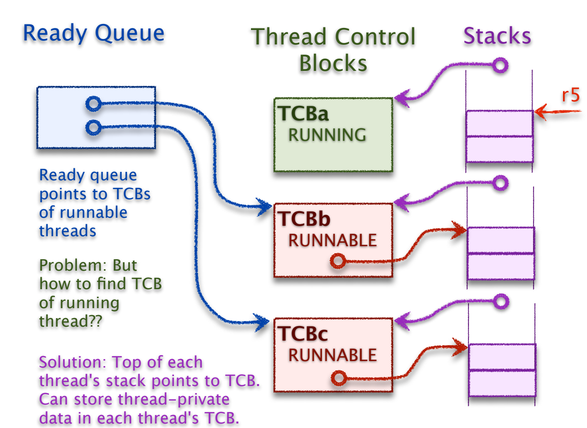
Multiple Processors

- Processors are
  - the physical / hardware resource that runs threads
  - a system can have more than one
- Uni-Processor System
  - a single processor runs all threads
  - no two threads run at the same time
- Multi-Processor System
  - multiple processors run the threads
  - two threads can be running at the same time
- More about this later, but we have a problem now ...
  - how do we compute the value of cur_thread, the current thread's TCB?
  - we need this to yield the thread, for example, to place it on ready queue
  - but, can't use a global variable

Thread Private Data

- Threads introduce need for another type of variable
  - a thread-private variable is a global variable private to a thread
  - like a local variable is private to a procedure activation
    - sometimes called thread-local storage
- For example
  - cur_thread, the address of the current thread's activation frame
  - It's a global variable to thread, but every thread has its own copy
- Implementing Thread Private Data
  - store Thread-private data in TCB
  - store pointer to TCB at top of every stack
  - compute current stack top from stack pointer
    - simple computation if stack starts at aligned location in memory, stack size is power of 2
    - StackTop = r5 & ~(StackSize - 1), where StackSize = 2^k

Thread Private Data



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*?
  - do some operations have *deadlines*?
  - should one thread be able to *pre-empt* another?
  - if threads can be pre-empted, are there times when this shouldn't happen?

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 are 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
- Benefits
- Drawbacks and mitigation

Preemption

- Preemption occurs when
  - a "yield" is forced upon the current running thread
  - current thread is stoped 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 10 ms
- How is quantum-based preemption implemented?

Implementing Quantum Preemption

- 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
- How is running thread preempted

Real-Time Scheduling

- Problem with round-robin, preemptive, priority scheduling
  - some applications require threads to run at a certain time or certain interval
  - but, what does round-robin guarantee and not guarantee?
- Real-time Scheduling
  - hard realtime – e.g., for controlling or monitoring devices
    - thread is guaranteed a regular timeslot and is given a time budget
    - thread can not exceed its time budget
    - thread will not be "admitted" to the run in the first place, unless its schedule can be guaranteed
  - soft realtime – e.g., for media streaming
    - option 1: over-provision and use round-robin
    - option 2: thread expresses its scheduling needs, scheduler tries its best, but no guarantee

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
- 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

Disassembly Strategy: Multiple Passes

- 1. find large-scale control flow
  - **draw arrows** to notice patterns!
    - `if` vs. `if/else` vs. `while`
    - procedures: `getpc/j` to call, `j 0(r6)` to return
- 2. find small-scale patterns: correspondences and symmetries across different spots in code
  - variable usage: address given in `.pos` is read from and written to
  - push/pop function arguments on stack: `inca/deca r5`
  - function returns value: use of `r0` value after function call
  - local variables within function: offsets from r5
- 3. comment ASM line by line
- 4. first pass from comments to verbose C
  - don't worry about arrays vs variables vs structs: can't tell
  - assume all loops are `while`
  - avoid sign error: `if/while (a)` is opposite from `beq`
- 5. second pass to tighten
  - can you eliminate temporary vars, e.g. inside loop?
  - can you turn verbose `while` loop into concise `for`?