Lecture C1: Thread abstraction and implementation

****

Review -- 1 min

****

- User/kernel
  - 3 reasons (interrupt, exception, trap)
  - event,
    {switch mode, switch PC, switch stack; save mode, PC, stack}
    save remaining state
    run handler
    restore remaining state
    retstore mode, PC, stack
  - QUESTION: why switch {mode, PC, stack} atomically? (Why not first switch mode, then PC and stack or vice versa?}
  -

  REVIEW QUESTION: How would you implement a hypervisor? [picture]. What happens on an interrupt? What happens on an exception/trap?

So, at this point I've hopefully convinced you that it is "obvious/natural" to have interrupts/trap/execption and for the "natural" semantics to be "save state + switch PC/stack/mode

That you need this just to have dual mode operation/system call/interrupts

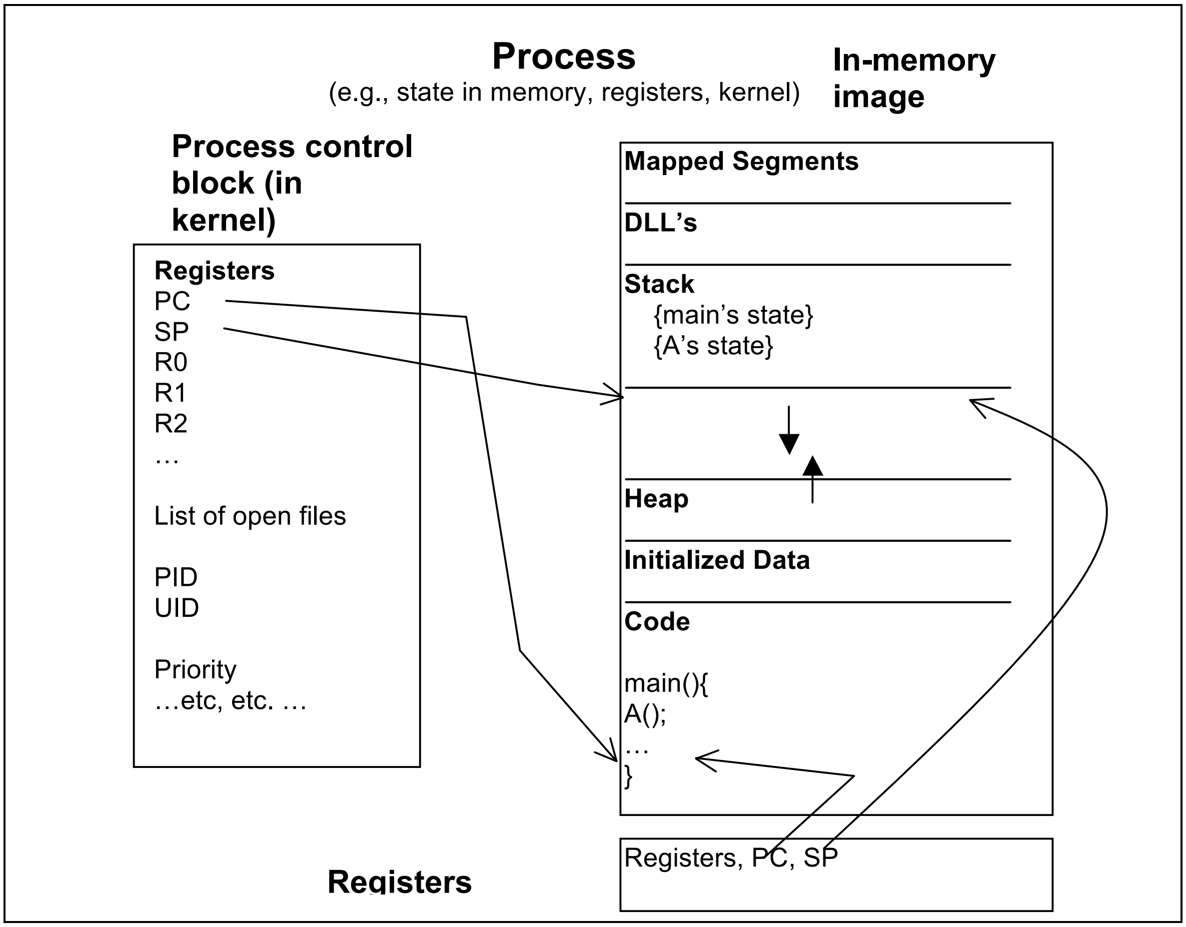
Turns out, I've snuck in almost all of "how to implement threads" -- this simple set of building blocks also gives you threads!

- ~~Virtual memory~~
  - ~~Protection~~
  - ~~Relocation~~
  - ~~Sharing~~
  - ~~Capacity~~
- ~~Process~~
  - ~~Thread – concurrency~~
  - ~~Address space – protection~~

  ~~Traditional Process – 1 thread + 1 address space~~
  ~~Multithreaded process – N threads in 1 address~~

Process state:

Process control block – in-kernel data structure containing per-process state (registers, stack pointer, program counter, priority, open files, …)



****

Outline - 1 min

****

Process: Definition

[from lec2.pdf]

Implementing threads

- Multithreaded processes
- Thread dispatch

****

Preview - 1 min

****

Upcoming: thread abstraction dilemma

- Want an abstraction that makes it seem like my thread is only one running (my own machine, sequential execution, etc.)
- Want threads to be able to cooperate to do some task.

If threads are cooperating, that breaks the independence abstraction – a thread can see that other threads have accomplished work.

Example – chess program where coordinator spawns worker threads to evaluate positions. Coordinator later looks and sees: “whoa! Someone figured out the answer for me.”

→ task is to come up with a new abstraction that allows independence when we need independence, and that allows cooperation when that’s what we need.

****

Lecture - 35 min

****

1. Multithreaded Processes

real world -- things happen concurrently (musicians in a band, server handling requests, air molecules streaming past wing of airplane)

computers simulate and interact with real world --> must model concurrency

example: google earth application

example: operating system uses concurrency so many things can run on machine "at once"

--> multi-threaded process

3 weeks ago: process = 1 thread + 1 address space
(Original definition – pre-1980…)
multithreaded process = N threads + 1 address space
(modern definition)

1.1 Goals of concurrency

(1) Program structure
    google earth example...
(2) Performance -- high-latency IO devices
    google earth example...
(3) Performance -- exploit parallel HW
    google earth example...

1.2 Threads

Address space – all the state needed to run a program
literally – all the addresses that can be touched by the program
provides illusion that program is running on its own machine
protection

Thread – a *separately schedulable sequential execution stream* within a process; *concurrency*

sequential execution stream -- normal programming abstraction

separately schedulable -- virtualize processor; illusion of infinite # CPUs running at diferent speed (control when threads run)

separately schedulable -- virtualize CPU
Main point: Each thread has illusion of own CPU, yet on a uniprocessor, all threads share the same physical CPU. How does this work?

abstraction: each thread runs on virtual processor at *unpredictable speed*

seems strange to assume completely unpredictable speed
-- simplifies programming model
-- physical reality -- lots of factors affect scheduling

Thread life cycle/state machine
[DRAW]

~~1.2.1 why separate these concepts?~~

1) ~~Discuss the "thread" part of a process separate from the "address space" part of a process~~
2) ~~many situations where you want multiple threads per address space~~

~~Why would you want to do this?~~
~~Examples of multi-threaded processes:~~

1) ~~Embedded systems: elevators, planes, medical systems, wristwatches, etc. Single program, concurrent operation~~
2) ~~Most modern OS kernels – internally concurrent because have to deal with concurrent requests made by multiple users. Note no protection needed within kernel.~~
3) ~~Network servers – user applications that get multiple requests concurrently off the network. Again, single program, multiple concurrent operations (e.g. file server, web server, airline reservation system)~~
4) ~~Parallel programming – split program into multiple threads to make it run faster. This is called **multiprocessing**~~
   ~~multiprogramming – multiple jobs or processes~~
   ~~multiprocessing – multiple CPUs~~

~~QUESTION: Is it possible for a multiprocessor (machine with multiple processors) to be uniprogrammed?~~
~~*Recall from last time – uniprogrammed means runs 1 process at a time.*~~
~~ANSWER: Some multiprocessors are in fact uniprogrammed – multiple threads in one address space, but only run one program at a time (e.g. CM5, Cray T3E)~~

~~1.3 Classification~~

~~Real OS's have~~

- ~~one or many address spaces~~
- ~~one or many threads per address space~~

~~# address spaces:~~	~~one~~	~~many~~
~~# threads per address space~~		
~~one~~	~~MS/Dos~~ ~~PalmOS~~	~~traditional Unix~~
~~many~~	~~embedded systems, Pilot (the OS on first personal computer ever built – idea was no need for protection if single user)~~	~~VMS, Mach, NT, Solaris, HP-UX, …~~

~~1.4 Thread programming abstraction~~

~~thread_create(procedure, argument) --~~
~~Thread fork is much like asynchronous procedure call – it means, go do this work, where the calling thread does not wait for the callee to complete.~~

~~What if the calling thread needs to wait?~~
~~Thread **Join()** – wait for a forked thread to finish~~

~~Thus, a traditional procedure call is logically equivalent to doing a fork and then immediately doing a join.~~

~~This is a normal procedure call:~~

```
~~        A(){~~
~~                B();~~
~~        }~~
```

~~The procedure A can also be implemented as:~~

```
~~        A(){~~
~~                Thread t = new Thread(B);~~
```

~~t->Join();~~

~~}~~

~~thread_exit(int status);~~

****

Admin - 3 min

****

So far so good (?)
HW2 due next wednesday
Project 1 due friday
Project 2 available

****

Lecture - 33 min

****

2. Implementing threads

4 variations:
In-kernel threads -- threads within kernel -- TCP and thread management all happens within same address space

Single-threaded process -- extends in-kernel threads -- use same abstraction to run programs in separate address space

Multi-threaded process with kernel supported threads – multiple threads within process; kernel system calls manage thread
kernel stores multiple TCBs per process and involved in dispatch/switch between threads (even between threads in same process)

→ thread management calls are handled with system calls
→ thread control blocks/ready list are kernel-level data structures

User level threads – library in user-level program manages threads within a process
TCB for multiple threads stored at user level; switch between threads in same process done as user-level library w/o kernel involvement
→ all thread management calls (thread_create(), thread_yield(), switch(), thread_destroy(), …) are handled with procedure calls
→ thread control blocks/ready list are user-level data structures

QUESTION: what are advantages/disadvantages of each
- Performance/overhead
- Scheduling policy
- I/O, page faults

(Actually also have hybrid systems – M user level threads on N kernel threads. Don't worry about that for now, but think about it later…)

Start with kernel-supported threads. Same basic ideas apply to others...

How do we implement threads?
Four key concepts
1) per-thread v. shared state
2) thread control block
3) dispatch/switch
4) thread create/delete

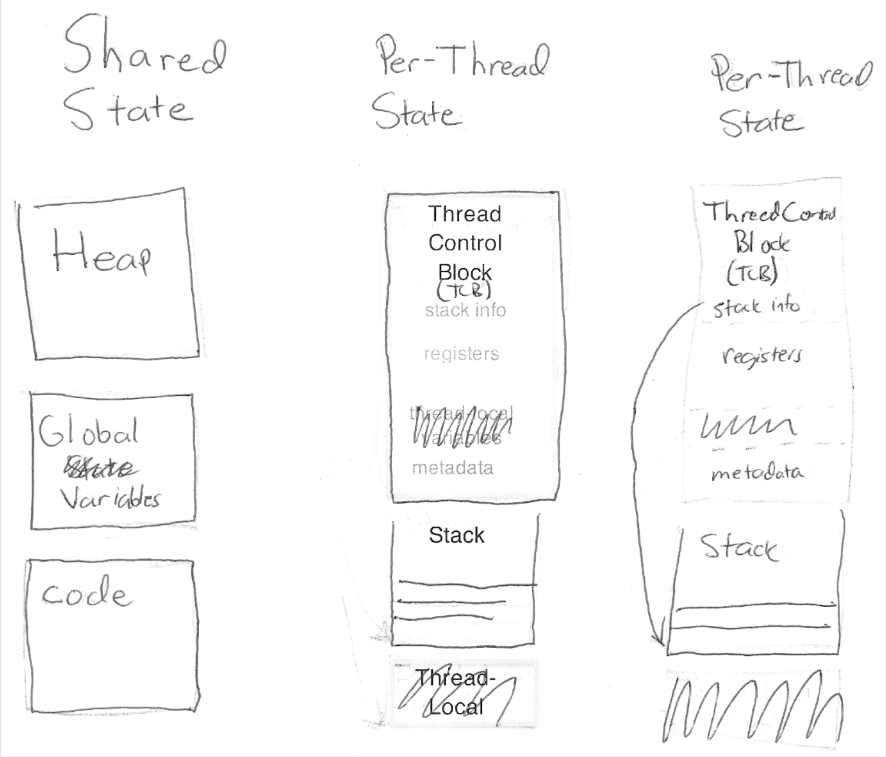
2.1 Thread State

To realize *sequential execution stream* need separate per-thread state to represent current state of computation

What state does a thread have?
1) Some state **shared** by all threads in a process/address space
e.g. global variables, heap, file system
2) Some state "**private**" to each thread – each thread has its own copy
e.g. Program counter, registers, execution stack
represents "sequential execution" state

Registers -- PC, stack pointer, frame pointer, GPRs, ...

Execution stack – where parameters, temporary variables, return PC are kept while called procedures are executing



Note: “Private” data is still in shared address space. Private by convention only. Nothing *enforces* protection. (Similar to “local” variables of a procedure)

[bottom line: programming model is “heap and global variables can be shared, stack variables cannot be shared”; the longer answer is “yes, stack variables in principle could be shared, but it is really a bad idea to do so and you should basically never do so.]

Notice -- of our 4 key concepts -- if we just had "per-thread state" and infinite CPUs, we would be done. Remaining 3 concepts are mechanism -- how do we multiplex multiple threads on finite number of CPUs?

2.1.1 Address space state

Threads encapsulate concurrency
Address spaces encapsulate protection – keep a buggy program from trashing everything else on the system

Address space state
- contents of main memory
- registers

In-kernel data structures: Unix files, user-ID, priority

2.2 Thread Control Block (TCB) Per thread data structure

look at previous picture – if I only want to run 1 of the threads, I load its state onto processor (PC, registers, stack pointer) and say "go" and the right thing happens

If multiple threads, running thread's state on CPU; need to store state of non-running threads somewhere.

Need to keep this per-thread state somewhere: **TCB**

Thread control block
- one per thread
- execution state: registers, program counter, pointer to stack
- scheduling information
- etc. (add stuff as you find a need)

So: for each process, kernel has list of TCBs – one per thread. Kernel can switch running thread by (1) taking state of running thread and moving to TCB and then (2) taking state from TCB and putting it on processor.

2.3 Dispatch/switch

how to switch which thread is running

Looks just like interrupt handler we talked about last time, but instead of restoring state of interrupted thread, restore some **other** thread's state

```
Thread is running
Switch to kernel
Save thread state (to TCB)
Choose new thread to run
Load its state (from TCB)

Thread is running
…
```

2.3.1 Switch to kernel

When thread is running, how does dispatcher get control back?

Let's focus on timer-interrupt case for kernel threads
-- basically the same as kernel-supported threads' system call and interrupt cases

Internal events:

1) system call, e.g.,
   - Thread blocks for I/O (examples: for disk I/O, or emacs waits for you to type at keyboard
   - Thread blocks waiting for some other thread to do something **synchronization**
   - Yield – give up CPU to someone else waiting
2) exception

What if thread never does any I/O, never waits, never yields, and never has an exception? Dispatcher has to gain control back somehow.

External events:

3) interrupt e.g.,
   - I/O (type character, disk request finishes) → wakes dispatcher so it can choose another thread to run
   - Timer – like an alarm clock

Pre-emptive v. non-pre-emptive threads

2.3.2 Save thread state

What state do you need to save/restore when the dispatcher switches to a new thread?

Anything the next thread may trash: PC, registers, change execution stack

Why: Want to treat each thread in isolation

(Note: generally need a bit of HW support to help you save the state of the running thread. In x86, exception or trap

(1) changes stack pointer to the exception stack (in a hardware register)

(2) push the old stack pointer, old stack segment, old execution flags (interrupt mask, etc), old code segment, old PC onto exception stack

(3) change PC to exception handler

software then has to save the rest of the interrupted program's state.

In project 3, you will build interrupt/trap handler that assembles a trapframe on the stack when an interrupt/trap occurs:

```
struct Trapframe {
  u_int tf_edi;
  u_int tf_esi;
  u_int tf_ebp;
  u_int tf_oesp;              /* Useless */
  u_int tf_ebx;
  u_int tf_edx;
  u_int tf_ecx;
  u_int tf_eax;
  u_short tf_es;
  u_int : 0;      /* 0-length unnamed bit field forces next field to
                   * be aligned on an int boundary */
  u_short tf_ds;
  u_int : 0;
  u_int tf_trapno;
```

```
 /* below here defined by x86 hardware */
 u_int tf_err;
 u_int tf_eip;
 u_short tf_cs;
 u_int : 0;
 u_int tf_eflags;
 /* below only when crossing rings (e.g. user to kernel) */
 u_int tf_esp;
 u_short tf_ss;
 u_int : 0;
};
```

2.3.3 Choosing a thread to run

Dispatcher keeps a list of ready threads – how does it choose among them?

- One ready thread – easy
- More than one ready thread: scheduling policies (discuss in a few weeks)

(Avoid "zero ready threads" case by having low-priority "idle process" that spins in an infinite loop.)

2.3.4 Running a thread

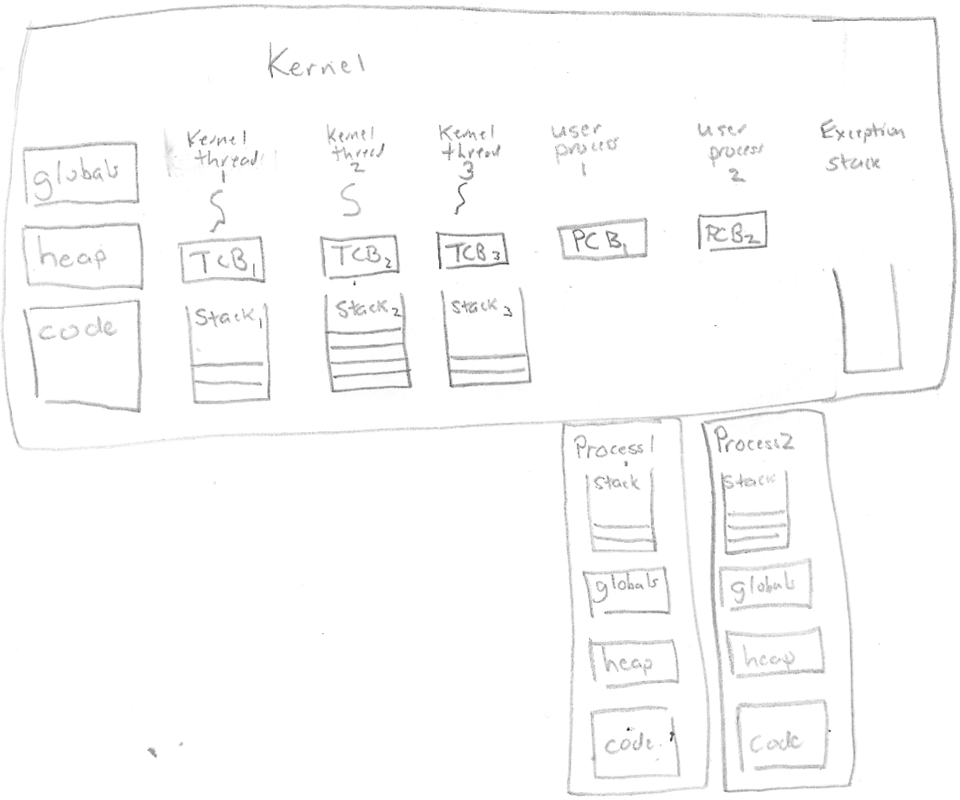
How do I run a thread? Load its state (registers, PC, stack pointer) from TCB into the CPU (e.g., reti)

(Note: HW support to restore state. Reverse what was done above.)

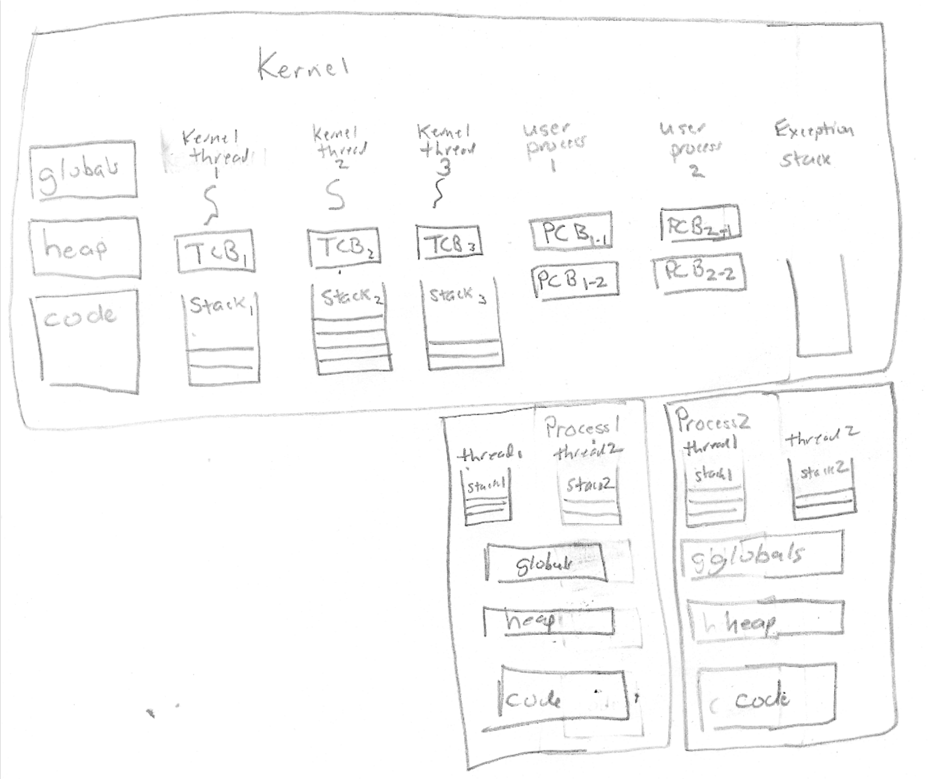
2.4 Extending abstraction to processes

Same concurrency abstraction + mechanisms
Memory -- some "shared" state is now "per-process"

Multi-threaded kernel with single-threaded processes:



Multi-threaded kernel with multi-threaded processes:



~~2.5 Observations~~

~~2.5.1 What does this look like from kernel's point of view?~~

~~User level interrupt → "lightning strikes" and a trap-frame appears on the stack and handler is running~~

~~When kernel gets done handling trap or interrupt, it executes "reti"~~

~~Notice: this "reti" is not a regular return!~~

~~QUESTION: Suppose kernel is several procedure calls deep when "reti" is called, what happens to kernel stack? What does kernel stack look like on next trap?~~
~~——→ it disappears; next trap handler starts on empty stack!~~

~~*The system stack contains a record of all system calls/interrupts that have been made but not yet returned. In a normal procedure call/return stack, each call is eventually returned from, and the stack holds what needs to be remembered while a call is pending. When OS does reti, it has completed processing of the system call/interrupt (and updated any kernel data structures appropriately). It has nothing more to remember. So, it is OK to just throw away the current contents of the system stack!*~~

~~OS creates process environment for user-level processes, but it does not run as a process itself.~~
~~~When user process is running, there exists no kernel thread.~~
~~~

~~Variation: Multi-threaded kernel – several kernel threads exist, have their own stacks, and own thread control blocks. These stacks continue to exist when user thread running (with state of kernel threads in TCBs.) Trap handler still runs on interrupt stack and can pass data to/from kernel threads via multi-threaded programming techniques we will discuss in coming weeks.~~

2.5.2 What does this look like from user-level process's point of view?

System call looks just like a procedure call (it is unaware that in the mean time, the stack pointer has pointed somewhere else and maybe even another process has run)

Timer (or other) interrupt looks like nothing. Process is unaware that it was ever taken off CPU

2.5.3 What does this look like from processor's point of view?

Thread 1	Thread 2	System stack	CPU
while(1)			while(1)
call Yield			call Yield
trap			trap
		save state 1	save state 1
		choose thread 2	choose thread 2
		load state 2	load state 2
		reti	reti
	while(1)		*while(1)*
	call Yield		*call Yield*
	trap		*trap*
		save state 2	save state 2
		choose thread 1	choose thread 1
		load state 1	load state 1
		reti	reti
return Yield			return Yield
call Yield			call Yield
trap			trap
		save state 1	save state 1
		choose thread 2	choose thread 2

…		load state 2	load state 2
	…	reti	reti
		…	...

2.5.4 How would you change this to go to "low power mode" when idle?

Idle process calls "syscall(low power mode)"
Need to restore power mode on trap handler

OR
Have idle loop in kernel (how would this work?)

~~3. User-level threads~~

~~Threads are useful programming abstraction – rather then implement with system calls in kernel, can implement with procedure calls as user-level library~~

~~(Notice: many user-level threads thus share 1 kernel thread)~~

~~--> TCB/ready list stored in user memory not kernel memory~~
~~--> switch happens at user level (thread create, lock, unlock, yield, etc. all are procedure calls not system calls)~~

~~3.1 User-level dispatcher:~~

~~Thread is running~~
~~Switch to kernel Call dispatcher~~
~~Save thread state (to TCB)~~
~~Choose new thread to run~~
~~Load its state (from TCB)~~

~~Thread is running~~
~~...~~

~~Just about the same as kernel threads…~~

~~3.1.1 Call dispatcher~~

~~When thread is running, how does dispatcher get control back?~~
~~**Internal events**:~~

~~1) thread library call, e.g.,~~

- ~~Thread blocks waiting for some other thread to do something~~
- ~~Yield give up CPU to someone else waiting~~

~~**External events:**~~

~~2) signal e.g.,~~

- ~~signal handler provides a way for user-level code to be called when (a) timer goes off (b) some other process sends a signal to this one~~

~~Again, very similar to kernel threads~~
~~Notice: main disadvantage of user-level threads if one user level thread blocks for I/O (system call), they all are blocked kernel scheduler has no way to know that there are other runnable user-level threads~~

~~**user-level signal handler ~= kernel interrupt handler**~~

~~handler code in process~~	~~handler code in kernel~~
~~tell kernel address of handler~~	~~tell hardware address of handler~~
~~tell kernel addr of handler stack~~	~~tell hardware addr of handler stack~~

~~What happens on timer interrupt~~
~~(1) HW switches from running process to kernel handler~~
~~(safe PC, SP, mode; change PC, SP, mode)~~
~~(2) Kernel handler saves remaining state, does stuff~~
~~in this case "does stuff"~~
~~save state of interrupted thread to process handler stack~~
~~reti to proces handler and handler stack~~

~~From process's point of view~~
~~It is running "normal code"~~
~~Then "virtual hardware" switches PC, SP and saves old PC, SP on exception stack~~

~~3.1.2 Thread context switch: Saving/restoring state~~

~~As with kernel need to store/restore registers to/from TCB~~

~~Tricky code~~
~~switch()~~

~~Replace current running thread with a different one from ready list~~

Thread 1	Thread 2	CPU
while(1)		while(1)
call Yield		call Yield
call Switch		call Switch
Save state 1		Save state 1
choose thread 2		choose thread 2
load state 2		load state 2
	while(1)	*while(1)*
	call Yield	*call Yield*
	call Switch	*call Switch*
	save state 2	*save state 2*
	choose thread 1	*choose thread 1*
	load state 1	*load state 1*
If? → my turn!		If? → my turn!
return Switch		return Switch
return Yield		Return Yield
call Yield		call Yield
call Switch		call Switch
save state 1		save state 1
choose thread 2		choose thread 2
load state 2		load state 2
	If? → my turn!	*If? → my turn!*
	return Switch	*return Switch*
…	…	…

QUESTION: what if switch due to timer interrupt?

Pre-emptive v. non-preemptive

- suppose we want to implement round robin scheduler
- set up a timer that interrupts (signals) this process every few milliseconds
- set up *signal handler*
  signal(SIG_INT, void (*handleInterrupt(int));

Option 1: handleInterrupt() gets called on same stack

Option 2: handleInterrupt(pointer to saved context) gets called

QUESTION: how will this be implemented for kernel threads?

4. Implementation details

4.1 Thread create

Thread_create() – create a new thread

```
thread_create(pointer_to_procedure, arg0, …){

    // Q: What per-thread state do I need
    // to create?
    //Allocate a new thread control block and
    //execution call stack
    TCB tcb = new TCB();
    Stack stack = new Stack();

    // Q: How should that state be
    // initialized?
    //Initialize TCB and stack with initial
    //register values and the address of the
    //first instruction to run
    tcb.PC = stub;
    tcb.stack = stack;
    tcb.arg0Reg = procedure;
    tcb.arg1Reg = arg0;
    tcb.arg2Reg = arg1;
    ..

    // Q: What else?
    //Tell dispatcher that it can run the
    //thread (put thread on the ready list)
    readyQ.add(tcb);
}
```

```
Why PC = stub? What is stub?

stub(proc, arg0, arg1, …){
        (*proc)(arg0, arg1, …);
        deleteCurrentThread();
}
```

4.2 Thread exit

deleteCurrentThread() puts TCB on a list of threads to be cleaned up, and then invokes scheduler to pick a thread to run from the ready list

QUESTION: Why not delete/free TCB and stack itself?

```
Thread_exit(int status){
      save status so that join() can get it;
      deleteCurrentThread();
}
```

QUESTION: How to save status?

4.3 Thread switch

user->kernel -- straightforward (interrupt/trap/exception save state...)

kernel->kernel (or user-level threads)

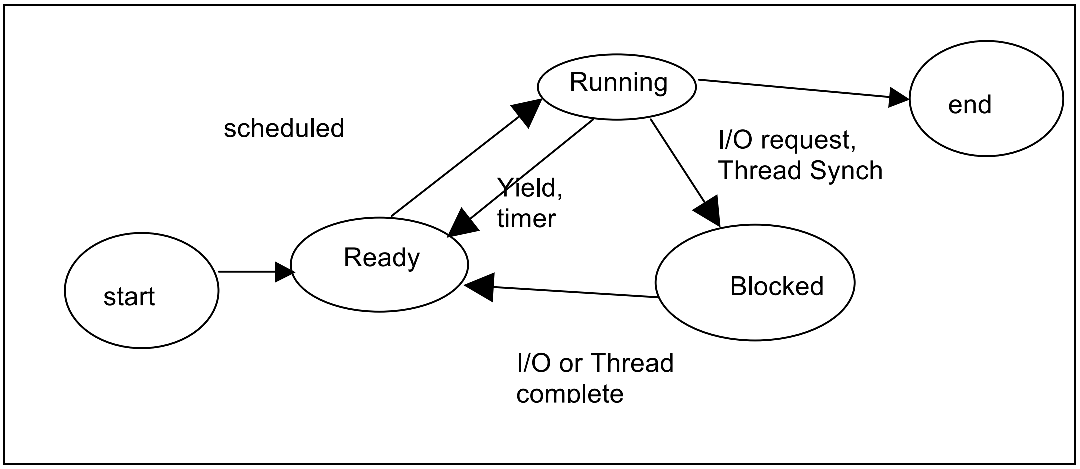
Example: user-level thread yield()

```
thread_yield(){
    switch();
}
//
// First look at bold face – basic pseudo-code
// Then look at red QUESTION – how do we get\
// out?
// Then, how do we fix?
//

// Need to tell if first or second
// time through. "Volatile" tells compiler
// not to optimize out the "if" below
// because "someone else might change it"
volatile int doneThat;

switch(){
    doneThat = 0;

    // save current thread's context
    getcontext(&(runningThread->TCB));

    Question: When/how often do we get here?

    if(!doneThat){

       doneThat = 1;

       // Select new thread to run
       nextTID = schedulePolicy();
       runningThread = thread[nextTid];


       // copy new thread's state to processor
       setContext(&thread[nextTid])
       QUESTION: when do we get here?
    }
    QUESTION: When do we get here?
}
```

4.4 Thread states

Each thread can be in one of 5 states:

0) Start – allocate resources
1) Running – has the CPU
2) Blocked – waiting for I/O or synchronization with another thread
3) Ready to run – on the ready list, waiting for CPU
4) Done – deallocate resources



Where is thread state?

- Running: thread state on CPU
- Ready: thread state in TCB on ready queue
- Blocked: thread state in TCB on waiting queue (mutex lock, condition variable, or semaphore)

~~PROJECT ULT~~
~~Overview/goals~~

~~Suggestions/Hints~~
~~(1) Start with ULT_Yield()~~
~~-- save my state to TCB and move to ready list~~
~~-- Pick a thread from ready list (me!)~~
~~-- Restore state of chosen thread (me!)~~
~~--> Can test, debug yield, switch w/o implementing thread create/destroy~~

~~(2) Then add create/delete; then interrupts; then lock/unlock~~
~~-- my time breakdown was about 25% yield, 20% create/delete, 25% interrupts, 30% lock/unlock [last one would have been less except for stupid bug]~~
~~-- so, part 1/3 is about half of project~~
~~-- (re)factor for code-reuse (as you go or up front)(core of yield gets used in several places)~~

~~(3) "tricky" bits~~
~~-- makecontext returns twice (see above)~~
~~-- interrupt discipline/invariant~~
~~-- understanding code well enough to refactor it~~
~~-- important b/c it will be hard to diagnose/fix bugs if you save/restore context slightly differently in 3 different places in your code...~~

****

Summary - 1 min

****