# Threads Implementation

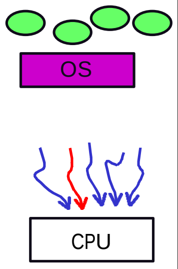
Arvind Krishnamurthy
Spring 2004

## Threads & Processes Recap

- Thread:
  - Stream of instructions executing (in an execution context)
- Process:
  - Thread(s) + Address space (execution context or data)
- Programs:
  - image from which processes are created
- Issues of state:
  - what is the state associated with a process?
    - data section, heap, stack, registers
  - what is the state associated with a thread?
    - private: stack, registers
    - shared: data section (global variables), heap

## 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?
- Question: how do we go about implementing threads?
  - What are the issues?
  - What are the implementation options?



## Choosing a thread to run

- Dispatcher keeps a list of ready threads
- Need to choose amongst them
- Easy cases:
  - Zero ready threads: loop
  - One ready thread
  - More than one ready thread:
    - LIFO (last in, first out): put ready threads at the front, dispatcher picks from the front --> results in starvation
    - FIFO (first in, first out): put threads at the back
    - Priority queue

## Scheduling Policies

- Scheduling issues
  - fairness: don't starve
  - prioritize: more important first
  - deadlines: must do by time 'x' (let's say you have a thread that is working on streaming video or a thread monitoring a manufacturing process)
  - optimization: locality issues
- No universal policy:
  - many variables, can't maximize them all
  - conflicting goals
    - more important jobs vs. starving others
- Given some policy, how to get control?

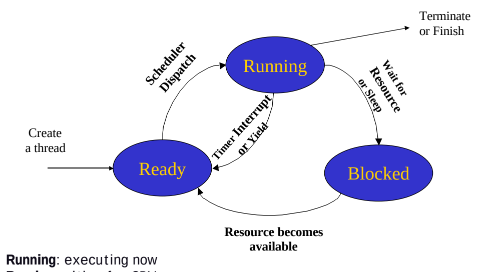
## OS Structure

- Processes block when performing I/O
  - They request OS to perform a service on their behalf
  - Voluntarily yield control to the OS
  - Referred to as a "system call"
- Blocked processes wait for interesting events
  - An interesting event could be a keyboard-press or a disk operation completing
  - When interesting events happen, an "interrupt" is raised
  - Interrupt stops currently executing process and gives control back to OS

## Dispatcher regaining control

- Internal events:
  - Thread blocks on I/O
  - Thread blocks waiting for some other thread to do something
  - Yield explicitly
- What if thread never did I/O, never required anything from other threads, never yielded control?
- Need external events:
  - Interrupts: type character, device (such as a disk) finishes request
  - Timer: set a periodic alarm
- Called “preemption”
  - Threads can also be scheduled in a non-preemptive fashion: means that threads are cooperating to yield control

## State transition



**Running**: executing now
**Ready**: waiting for CPU
**Blocked**: waiting for I/O or lock

## Outline

- Threads implementation requires:
  - Dispatcher loop

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

  - Execution context switching & thread control blocks
    - What needs to be saved/restored?
    - Where is the space allocated for it?
    - How is the save/restore performed?

## Switching between Procedures

- Procedure call:
  - Simple approach: caller saves registers that callee could trash
  - Complex approach: Register set is “divided” into “caller registers” and “callee registers”

```
save callee registers currently being used by caller (on stack)
save program counter
call procedure
procedure:
     saves caller registers that procedure needs on stack
     executes
     restores caller registers & pc
     jumps back to pc
restore active callee registers
```

- How is state saved?
  - Variations come from whether state is saved proactively or lazily?
  - managed by the compiler

## Threads vs. Procedures

- Threads may resume out of order:
  - cannot use LIFO stack to save state
  - general solution: duplicate stack
- Threads switch less often
- Threads involuntarily interrupted:
  - asynchronous: thread switch code saves all registers
- Context switch is always clever code. Consider:
  - Two threads loop, each calling **Yield**
  - **Yield** calls **Switch** to switch to the next thread, but once you start running the next thread, you are on a different execution stack
  - **Switch** is called in one thread’s context, but returns in the other's!

## Announcements

- Some problems with the submit script: should be sorted out soon
- Assignment 0 due on Monday at 11:59 pm
- If you have formed a group for remaining assignments, send an email to the TA

## Case Study: Nachos

- Nachos threads have 4 states: JUST_CREATED, RUNNING, READY, BLOCKED
- Thread creation: "Thread" object is the TCB

```
Thread *t = new Thread ("foo");       // JUST_CREATED
t->Fork (func, arg);                  // READY


Thread::Thread(char* threadName)
{
   name = threadName;
   stackTop = NULL;
   stack = NULL;
   status = JUST_CREATED;
   for (int i = 0; i < MachineStateSize; i++)
      machineState[i] = NULL;
}
```

## Thread Creation in Nachos

- Thread creation:
  - **Thread:Fork** takes two arguments
    - a pointer to the application routine to execute and an argument
    - allocate a stack (of StackSize) and let the stack pointer points to it

```
stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
*(--stackTop) = (int) ThreadRoot;   // a little bit of trickery here
*stack = STACK_FENCEPOST;           // to check for overflow
```

    - the thread is then put on the ready list
  - ThreadRoot(func, arg)
    - (Input registers passed along to ThreadRoot from Fork thru' TCB)
    - call func(arg)
    - (when func returns, if ever) call **ThreadFinish()**

## Nachos "Switch"

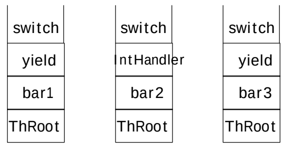
- Code in switch.s

```
void SWITCH (Thread *oldT, Thread *newT);

/ * calling from oldT * /
/* save registers of running thread to TCB of oldT */
TCB[oldT].regs.r7 = CPU.r7;
..............................
TCB[oldT].regs.r0 = CPU.r0
TCB[oldT].regs.sp = CPU.sp
TCB[oldT].regs.retpc = CPU.retpc  /* return address – sometimes on stack */

/* load state of new thread into CPU */
CPU.r7 = TCB[newT].regs.r7;
..............................
CPU.r0 = TCB[newT].regs.r0;
CPU.sp = TCB[newT].regs.sp;
CPU.retpc = TCB[newT].regs.retpc;
/* return to newT */
```

## Thread Stacks



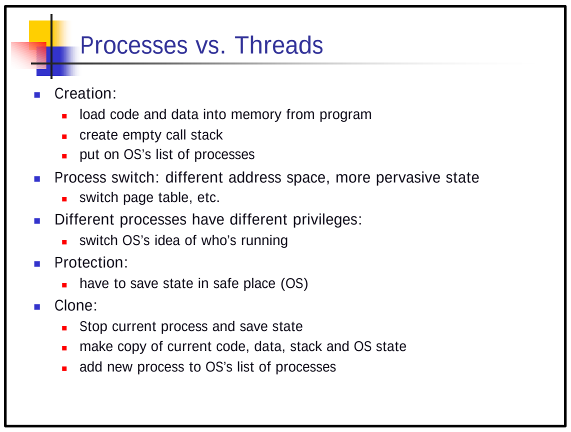
At any given point in time, the top stack frame for all the threads are executing "switch"

## Notes on context switching

- Very machine dependent. Must save:
  - general-purpose & floating point registers, any co-processor state, shadow registers (Alpha, sparc)
- Tricky:
  - OS code must save state without changing any state
  - How to run without touching any registers?
    - **Some CISC machines have single instruction to save all registers on stack**
    - **RISC: reserve registers for kernel or have a way to carefully save one and then continue**
- How expensive? Direct cost of saving; indirect cost of flushing useful state (cache, TLB, etc.)

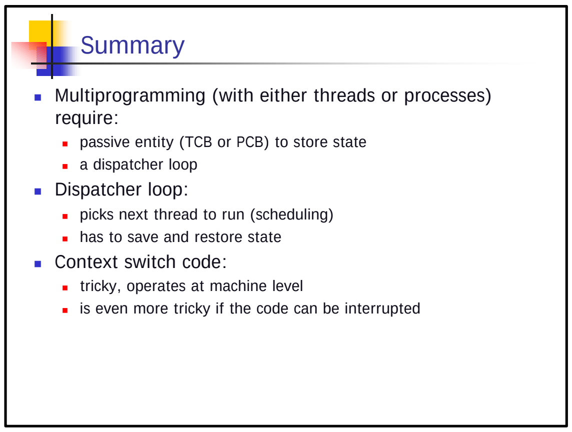
## Thread Context Switching Summary

- Thread constructor creates TCB
- Fork:
  - Makes up a stack
  - Fills it with a return address to "ThreadRoot"
  - TCB is filled with register values that will be used by ThreadRoot
  - Puts thread on ready queue
- Currently running thread invokes "switch"
  - Either through "Yield"
  - Or because it is interrupted and dispatcher performs a switch
- Switch saves current thread state
  - Loads thread state of the new thread or old thread
  - If new thread, the first thing that happens is that "ThreadRoot" gets invoked



## Processes vs. Threads

- Creation:
  - load code and data into memory from program
  - create empty call stack
  - put on OS's list of processes
- Process switch: different address space, more pervasive state
  - switch page table, etc.
- Different processes have different privileges:
  - switch OS's idea of who's running
- Protection:
  - have to save state in safe place (OS)
- Clone:
  - Stop current process and save state
  - make copy of current code, data, stack and OS state
  - add new process to OS's list of processes



## Summary

- Multiprogramming (with either threads or processes) require:
  - passive entity (TCB or PCB) to store state
  - a dispatcher loop
- Dispatcher loop:
  - picks next thread to run (scheduling)
  - has to save and restore state
- Context switch code:
  - tricky, operates at machine level
  - is even more tricky if the code can be interrupted