Threads

Sanzheng Qiao

Department of Computing and Software

December, 2012

What is a thread?

A sequential execution stream (thread) within a process (also called “lightweight” process).

A process has at least one thread of control.

A processe has two parts: threads (concurrency) and address spaces (protection).
Some systems (e.g., new version of UNIX, Solaris, Windows NT) allow multiple threads per address space.

Thread states

- States shared by all threads in the same process/address space:
  - global variables
  - file system
- States “private” to each thread:
  - PC, registers
  - execution stack contains parameters, temporary variables, return addresses.

Why threads?

Multithreading:

A single program made up of a number of different concurrent execution streams (threads). Also called SPMD (Single Program Multiple Data). It is fast to create threads and context switch threads belonging to the same process.

Why threads?

Multithreading:

A single program made up of a number of different concurrent execution streams (threads). Also called SPMD (Single Program Multiple Data). It is fast to create threads and context switch threads belonging to the same process.

Examples of multithreaded programs:

Window system: single program but one thread per window.

Multiprocessing (multiprocessor systems):

Split program into multiple threads to make it run faster by running on multiple processors. This is called parallel programming.

Creating a thread

Construct a thread class (thread `fork` in Nachos).

Creating a thread

Construct a thread class (thread `fork` in Nachos).

What happens when a thread class is constructed?

Thread `fork` is very much like an asynchronous procedure call. The caller does not wait for the callee to complete (return). A traditional procedure call is like:

```
A() {
    Thread* t = new Thread;
    t->fork(B);
    this->join();
}
```

Context switch

Running a thread

Load its state (registers, PC, stack pointer) into CPU and do a jump.

Context switch

Running a thread

Load its state (registers, PC, stack pointer) into CPU and do a jump.

Switching threads

What do you need to save/restore when thread *T* switches to thread *S*?
Anything thread *S* may trash: PC, registers, execution stack.

Nachos: Yield calls `Switch` to switch to the next thread. `Switch` is called in one threads context, but returns in the other's! There is a real implementation of `Switch` in Nachos in `switch.s`. It's magical!

switch.s

```
SWITCH:
    sw  sp, SP(a0)      # save new stack pointer
    sw  s0, S0(a0)      # save callee-save reg's
    sw  s1, S1(a0)
    sw  s2, S2(a0)
    sw  s3, S3(a0)
    sw  s4, S4(a0)
    sw  s5, S5(a0)
    sw  s6, S6(a0)
    sw  s7, S7(a0)
    sw  fp, FP(a0)      # save frame pointer
    sw  ra, PC(a0)      # save return address
```

switch.s

```
    lw   sp, SP(a1)     # load new stack pointer
    lw   s0, S0(a1)     # load callee-save reg's
    lw   s1, S1(a1)
    lw   s2, S2(a1)
    lw   s3, S3(a1)
    lw   s4, S4(a1)
    lw   s5, S5(a1)
    lw   s6, S6(a1)
    lw   s7, S7(a1)
    lw   fp, FP(a1)
    lw   ra, PC(a1)     # load the return address

    j        ra
         .end SWITCH
```

Thread programming

Write programs with multiple simultaneous points of execution, synchronizing through shared memory.

Concurrent programming has techniques and pitfalls that do not occur in sequential programming.

Global variables are shared among all the threads of the same process. Threads can read and write the same memory locations. The programmer is responsible for using the synchronization mechanisms of the thread facility to ensure that the shared memory is accessed in a manner that will give the correct answer.