CS 153
Design of Operating Systems
Ch. 4 Multithreading

View a program structure as a set of cooperating concurrent activities.

- allows multiple CPUs to be devoted to a single executing process
- allows getting data from many different input devices
- allows writing data to many different output devices

thread – stream of activity within a process
other definitions

- manifested by the existence of a thread descriptor
- "animated spirit" of a procedure
- "locus of control" of a procedure in execution
- entity to which processors are assigned
- "dispatchable" unit
- anything that can wait
- sequence of procedure invocations
    - first is nonblocking
    - rest are blocking by its most recent still-active predecessor

Thread creation
a nonblocking function invocation (caller does not wait for callee to return)

AR for a nonblocking invocation must be pushed onto a fresh stack so each thread has exclusive access to its own unique stack of the ARs of all its blocking invocations

threads can wait – a thread is waiting from the time it is created or suspended until it gets dispatched

a thread is *running* from the time it is dispatched until it is suspended or destroyed

a processor is a thread's *host*; a thread is the processor's *guest*

descriptor for threads contains information about

- dispatch time
- values of key processor register needed on a dispatch

threads passing its processor to another thread

- dispatch the other thread and suspends itself

virtual processor – a thread performing the activities of another thread; can move from stack to stack

Threads vs. processes
different processes have different address spaces
different threads have different stacks, but may share the same process

threads: fundamental units of concurrent activity
processes: data segments that hold global variables for threads

monothreaded – process with one thread
multithreaded – process with multiple threads

only kernel threads directly get access CPUs

other threads run on virtual-processor threads from the kernel

tasks – kernel-known threads in UNIX

pseudo-concurrency can be achieved by multiple user threads passing a task amongst themselves

to achieve true concurrency, we need a system call where a user thread can acquire a task on which to host one of its process mates

Clone()
a system call; one kernel-known thread creates another and specifies which parts of a parent to copy or share with its child

a superset of fork
flags:
COPYVM – child pages are copy-on-write images of the parent pages if set; share the same pages if not set
COPYFD – child's file descriptors are copies of parent's if set; shared if not set

if neither is set, the child is a kernel-known thread serving as a virtual processor for a user thread in its parent's process.

netnews suggestion: a clone flag that shared all of the VM except the stack

response

- impossible to do an efficient process-switch on any current hardware
- impossible to do a sane implementation of good sharing

- pointer passing problems to stack; cannot do so when the thing to be passed is communicating with another thread

4.1 Concurrency and Servers

a server can have 1+ queues of pending service requests from client threads

categories of services

- passive – handler is directly invoked by client thread
- active – handler is invoked by another thread
    - nonpreemptive – client posts request
        - client-blocking (client waits)
        - client-nonblocking (client does not wait)
    - preemptive – request seizes a processor (interrupt)
        - diverting – processor's current thread invokes handler
        - nondiverting – processor switches to another thread, which invokes the handler…can choose whether to leave the prior thread runnable

Active vs. passive services

two ways

- passive – clients queue up to use the server to invoke and run the server's handler
- active – client requests are queued up and agent threads within the server invoke the handlers on behalf of the clients

passive services

client-blocking – the client requesting cannot do other computations until the request has been serviced

remoteness – request for a passive service must be handled in an address space accessible to the client

active services

client-nonblocking – client can do other computations before the request is serviced
no need for remoteness
requests are handled on server-local stacks by agent threads within the server
overhead: need to copy parameters and return values from one stack to another

Preemptive vs. nonpreemptive active services

polling

poll – a thread watches for requests and services them by invoking the handler or dispatching a thread that does so

advantage: occurs at predictable points; easy for architecture and compiler to deal with
disadvantage: instructions that poll must be inserted to occur often enough to make sure requests are serviced; possible overhead in long-running tight loops

preemptive

divert – processor is designed to divert its current guest to the invocation of that service's handler or to temporarily abandon the preemptee and host some other thread to invoke the handler

Preemptee diverting vs. preemptee nondiverting

diverting – the handler of a preemptive service is invoked by or on behalf of the preemptee; blocking

a service is diverting if the handler's activation record is allocated on the preemptee's stack

less overhead than nondiverting since there is less context to switch

Kernels and diversion

a kernel is a server; kernel invocations are service requests

diversion is more common in embedded systems, where hardware lacks protection and all software is trusted

nondiverting kernel invocations are common for multiprogrammed systems.

system calls – passive…a bit like procedure invocations
interrupts – preemptive; nondiverting; treated as thread dispatch

Signals

preemptive service requests to user processes

nondiverting in that they are routed via some protocol to an appropriate thread for handling

occur when the kernel propagates an interrupt or trap to a user thread