CSCE 455/855
Distributed Operating Systems

Threads

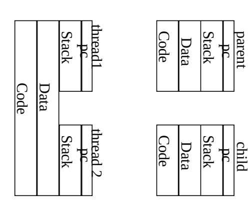
Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE855

1

Two Types of Processes

- "Heavyweight" process
  » normal process that does not share memory with other processes
- "Lightweight" process: Threads
  » memory is shared with other processes
  » maintain own state, program counter, stack



2

Threads

- Characteristics
  - » thread shares address space with other threads
    - ❖ therefore has access to same global variables
    - ❖ can also wipe out other threads' work (overwrite variables, etc)
  - » also shares open files, signals, semaphores, ports, etc.
  - » threads have ready, blocked. running states
    - ❖ scheduled just like processes (1 per CPU)
- Threads simply replicate a program
  - » runs the same code
  - » local variables on private stack, just like any other program
  - » but now we have >1 instances of the program
    - ❖ which may be at different places in the program

3

Example Use of Threads

- File servers must block while waiting for disk
- May want to "multiprogram" the file server
  - » while blocked, process next request
- Threads a convenient way to do this
  - » program one file server
  - » spawn threads as needed

4

Advantages of Threads

- Sharing data is easy
  - reduction in interprocess communication overhead
  - just synchronize critical sections in shared data
- Less overhead for context switching
  - address space (paging tables for ex.) is the same
- Ease of programming
  - only one address space to worry about

5

User vs. Kernel Threads

- User threads
  - kernel schedules processes/tasks
  - task schedules individual threads
  - allows for customized scheduling algorithms
    - but must be done within the task scheduled by kernel
  - problem: how other threads get scheduled
    - only if current thread releases CPU (gets blocked)
  - some systems don't have kernel threads
    - threads packages for UNIX exist
- Kernel threads
  - kernel schedules and manages threads
  - when a thread blocks, choose another thread
    - kernel can also preempt threads
  - either from same or different process/task
  - beware of context switch
    - between task switching is expensive
    - will want to schedule threads from same task together

6

User Threads

- Thread package runs on top of kernel
  - » referred to as ‘user-space runtime system’
  - » kernel maintains processes
  - » user program maintains threads
    - ❖ package provides procedures for managing threads - the runtime system
- Advantages
  - » can run threads on process-oriented systems (like Unix)
  - » each process can have customized scheduling algorithm
  - » context switch is minimized

7

User Threads (cont.)

- Blocking system calls
  - » if a blocking system call goes to the kernel...
    - ❖ the kernel suspends the <u>process</u>
  - » want to call another thread when one becomes blocked
    - ❖ use non-blocking calls (some OS’s don’t support these)
    - ❖ Unix *select()* - use instead of read - if read() would block, call thread manager instead
- Page faults
  - » kernel would block entire process on page fault
    - ❖ even though another thread may not be blocked
  - » no real solution to this...

8

User Threads (cont.)

- Preemption of processes
  - » no good way to perform preemptive scheduling
    - ❖ clock interrupts are at process level
  - » thread must voluntarily give up CPU
  - » can lead to deadlock
    - ❖ for ex: waiting for a semaphore with a blocking system call

9

Kernel Threads

- Kernel schedules threads, not processes
  - » when a thread blocks, kernel can schedule:
    - ❖ another thread from the same process
    - ❖ a thread from a different process
  - » context switch is more expensive
    - ❖ kernel has to maintain process tables with thread entries
- User-level problems with page faults, deadlock don't occur

10

Threads in Distributed Environments

- Multiprocessors
  - » task environment located in shared memory
  - » threads can run in parallel on different CPUs
  - » just need to synchronize shared memory access
- Networked environments
  - » Can a task on machine X have a thread on machine Y?
    - ❖ not without the task environment
  - » Why are threads an issue for distributed systems?
    - ❖ because server model needs multiple copies running the same code

11

Threads in Distributed Environments

- Threads for the client/server model
  - » each client makes a request to the server
  - » server executes identical procedure for all calls
    - ❖ context switching may dominate server processing!
  - » threads reduce overhead
  - » make server programming easier

12

Programming with Threads

- Thread management
  - » static threads: number of threads is fixed
  - » dynamic threads: threads can be created and destroyed at run time
    - ❖ can re-use threads
- No protection between threads
  - » assumption is that threads are designed to cooperate
    - ❖ but global data is shared
    - ❖ what about race conditions?
  - » dependent on thread scheduling strategy
    - ❖ non-preemptive: thread must explicitly yield CPU
    - ❖ preemptive: race conditions can occur

13

Programming with Threads
Critical regions v.s. Resource sharing

- Mutex
  - » essentially a binary semaphore
  - » in C Threads package:
    - ❖ *mutex_lock(mutexId)*
    - ❖ *mutex_unlock(mutexId)*
  - » non-blocking mutex: "trylock"
  - » deadlock can occur with semaphore locks
- Condition variables
  - » use for long term waiting
  - » using signals on condition variables reduces probability of deadlock
  - » *condition_wait(conditionId, mutexId)*
    - ❖ queue request on conditionId
    - ❖ then mutex is released
  - » *condition_signal(conditionId)*
    - ❖ if a thread is queued in conditionId, unblock it

14

Programming with Threads
Handling global data

- Problem: global system variables
  - » but the globals are shared among threads
  - » *errno* variable in Unix
  - » C uses single buffer for each stream
    - multiple threads may write buffer or update pointer inconsistently
- Solution: each thread given private global variable
  - » *create_global("data1")*
    - create local space for the global
    - other threads calling *create_global("data1")* get a separate address space
  - » *set_global("data1")*
    - write to the local data store
  - » *x = read_global("data1")*
    - read the data from local store

15

User vs. Kernel Threads
The Trade-off

- User: good performance
  - » no costly transitions from user to kernel space
- Kernel: ease of programming
  - » no non-blocking reads, etc.

16

Scheduler Activations

- kernel allocates "virtual processors" to process
  - » process can allocate them to threads
- The upcall
  - » kernel communication with thread run-time system
  - » when thread blocks, kernel invoked via blocking system call
  - » kernel activates run-time system and passes info
  - » run-time system can now decide which thread to call
- Interrupts
  - » if process is interested in interrupt, kernel calls interrupt handler in run-time system
  - » otherwise interrupted thread is re-started after kernel handles interrupt

17

RPCs and Threads
Many are satisfied on same machine

- Use shared memory to pass parameters, results
  - » when started, kernel given interfaces for both client and server
    - ❖ kernel creates an argument stack shared by both
  - » when client calls server...
    - ❖ client puts info into stack, traps to kernel
    - ❖ server reads directly from it address space
    - ❖ results passed in same manner
- Pop-up threads
  - » server threads serve no purpose when idle
  - » therefore discard the thread
  - » when server invoked, create thread on-the-fly
    - ❖ map message memory to new thread
    - ❖ note it is less expensive to create a new thread than to restore an existing one...
    - ❖ Why?

18