Multiprocessor Threads

1. Group presentation none!
2. Questions
    a. Are the gains worth it?
    b. Upcalls – what makes them fast or slow?
    c. Coarse grained vs fine-grained parallelism?
    d. Deadlock recovery?
    e. Malicious applications?
    f. How can user threads deadlock?
        i. If one thread holding scheduler lock when preempted, kernel starts another thread that tries to acquire it
        ii. Condition variables: one thread waits for another to signal. Without a kernel thread to run another thread and signal, may never be signaled.
    g. How does user-level threading/context switching works
    h. Complexity of implementation?
    i. Real-time implications?
3. Notes from Reviews:
4. Review: what is a thread?
    a. Stack + registers + scheduling information
        i. Can be scheduled/context switched
        ii. Share address space
5. Multicore Systems – what is needed?
    a. In general what needs to be done in modern systems?
        i. timer coalescing (for power)
        ii. Share multiple cores across multiple applications
        iii. gang scheduling: run all threads of a program at once
    b. How use multiple cores?
        i. old world: mostly I/O bound workloads (database, web server). HPC important but not huge market for OS vendor
        ii. new world: everybody is multicore, many more parallel programs (CPU bound)
    c. Scheduler queue alternatives:
        i. QUESTION: What is the problem?
            1. Lock contention – many cores accessing a shared data structure
            2. Cache coherence – moving cache lines between cores
        ii. One big queue with a single lock
            1. Idle cores search for work: spin checking queue
            2. Simple but contend on everything
        iii. Per-core free list, global ready queue
            1. threads have free list of TCBs, stacks – no locking

                a. Need to balance free lists
                    i. length threshold to release resources to global pool
            2. Result: lock contention on ready list, but allocation/freeing is pretty cheap (for creating threads)
        iv. Central queue of idle processors, central ready queue
            1. New threads dequeue an idle processor
            2. Still have central lock contention for thread creation
            3. No benefit in absence of idle processors
        v. Local Ready Queue, free lists
            1. enqueuing, dequeuing threads is parallel
            2. Problem: balancing ready lists
                a. Solution 1: lock local lists, idle processors scan remote lists
                b. *** THIS IS THE BEST ONE
    d. Spin lock alternatives
        i. Spin on xchng/t&s
            1. lots of writes expensive for memory system – often invalidated, lots of bus traffic
        ii. Spin on read
            1. test-and-test-and-set
                a. spin on local copy, when invalidated try test-and-set (xchng)
            2. Better – but unnecessary xchng invalidating others
        iii. Exponential backoff
            1. wait for an increasing amount of time before trying for lock
            2. Avoids all threads trying xchng at the same time and flooding interconnect
        iv. Idle queue
            1. Waiter adds self to queue in lock
                a. subsequent waiters wait on previous waiter
            2. Spin on thread-local variable
                a. Lock release writes that variable
            3. No contention, no bus traffic
        v. Overall: exponential backoff a bit slower with few processors, short critical sections, but more scalable
6. Number of threads in a program
    a. How do people write/run parallel programs?
        i. Count # of CPUs in the system
        ii. Start up that many threads
        iii. Hope the OS gives them to you
    b. See a problem?
        i. What happens with multiple parallel programs: how many cores does each one get?
            1. The same number?

        2. The one that scales the best?
        3. The interactive or the batch one?
    ii. Apple solution: Grand Central Dispatch
        1. Give work items to a globally coordinated queue
        2. Makes sure that available CPUs are shared across processes
        3. Handles using idle cores and how many threads to create via a programming model
  c. How do programs request CPU resourcces?
    i. Linux/Windows: Threads?
        1. Problem is that they occasionally use the CPU, usage may vary widely
    ii. Virtual machines: virtual CPUs
        1. Scheduler within a guest OS decided what to run

7. **BIG PROBLEM: information flow between OS (what is available) and program (what is needed)**
8. Context:
  a. 2 approaches:
    i. user threads: purely userlevel code for:
        1. context switching
        2. synchronization
        3. scheduling
    ii. kernel threads: same features in the kernel (e.g. mach threads, windows NT threads)
  b. Kernel Rules:
    i. Must completely control scheduling of processors
        1. Can't let user code have control
        2. Can let user code advice
        3. Can't let advice be used for correctness – otherwise commandeering
    ii. Kernel multiplexes processors between processes – can't be done within a process
  c. Thread operations:
    i. spawn a thread
    ii. synchronize (e.g. locking)
    iii. terminate a thread
    iv. schedule a new thread
  d. Issue:
    i. User level threads fast; no need to go to kernel
        1. Expensive to enter kernel
        2. Kernel approach must be general & work for all applications
    ii. But: Kernel events pre-empt user threads
        1. Block high-priority threads on I/o, or preemption for time slicing

            2. Schedule kernel threads on separate mechanism; may run wrong thread or too many threads
                a. e.g. wake up low-priority thread instead of high-priority
            3. Correctness: kernel may schedule wrong threads – ones that are blocked waiting for a user thread blocked in kernel: may not have enough kernel threads to schedule a new user thread and make progress.

9. Goals:
    a. No kernel intervention in common case
    b. No processor idles when any program has threads to schedule
    c. No high-priority thread waits for processor while low-priority thread executes
    d. When a thread blocks in kernel, processor can be rescheduled with a different thread
    e. User-level portion can be customized
10. Key observations:
    a. Problem: kernel threads are not the right abstraction; too coarse
        i. Provide both user-level context for execution and a kernel-level context for blocking & system calls
    b. Kernel needs information/notification from user level about:
        i. When threads runnable
        ii. When processors not needed
    c. User level threads need information from kernel:
        i. When thread pre-empted
        ii. When thread resumed
        iii. When processor removed
        iv. When processor returned
    d. Goal: don't want to spend lots of time communicating information to kernel; is too expensive
        i. Give each side complete **control** over its domain
        ii. Provide other side with **knowledge** of what it is doing
        iii. Separate out the two tasks
11. Solution:
    a. HIGH LEVEL SOLUTION: interface / abstraction
        i. Expose info to across information
    b. Rules:
        i. Kernel controls which kernel threads run, how many run
            1. Kernel notifies user of any **processor** scheduling events
                a. Adding/Taking away a processor
                b. Blocking/unblocking kernel thread
        ii. User controls which user threads run
            1. User notifies kernel when needs fewer or more processors

        iii. QUESTION: how are processors allocated to address spaces?
            1. ANSWER: Any way you want – priority, fair share, etc. Just schedule processors instead of threads
            2. More fair – balances processes with many threads for I/O against those with few threads for compute
    c. User code runs in "scheduler activations", not threads
        i. Scheduler activations run to completion; they are never re-scheduled
        ii. System upcalls into user-level scheduler on all interesting events
            1. Thread blocked
            2. Thread wakes up
            3. Thread pre-empted
            4. Processor available
            5. Processor removed
        iii. **NOTE: removing "sequential processor abstraction" of THE and every other OS (but exokernel and barrelfish)**
            1. Instead, makes interruptions directly visible as upcalls with registers of interrupted thread
        iv. **QUESTION**: how do preemptive multithreading?
            1. Can do context switch when kernel preempts a user thread
            2. Can register for timer signals indicating quantum expires
        v. **Up-call may pre-empt** existing thread, causes notification that two threads are runnable

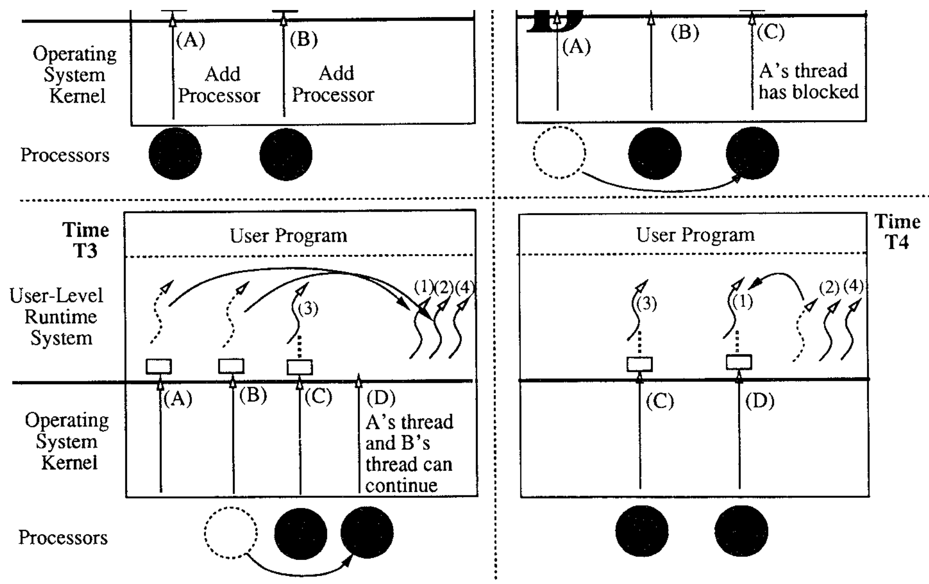

Fig. 1. Example: I/O request/completion.

kernel takes the processor that had been running thread 1 and performs a upcall in the context of a fresh scheduler activation. The user-level threa scheduler can then use the processor to take another thread off the ready li and start running it.

At time T3, the I/O completes. Again, the kernel must notify the user-lev

vi. thread system of the event, but this notification requires a processor. Th

d. Application notifies kernel of available parallelism
   i. Fewer contexts needed
      1. call "idle" – can be taken if needed
   ii. More contexts could be used
      1. call "add more processors"
   iii. NOTE: no need to notify of other events, such as which thread is running now or context switches
   iv. Why? Transition is when kernel takes action; actual level not that important
      1. Security issues?
         a. Process can already create/destroy threads to request CPUs
e. IMPLEMENTATION:
      1. Add code to kernel yield routine to generate scheduler activation
      2. Add code to syscall_ret to return to a common place if the thread had been blocked
   ii. KEY POINT:
      1. No need to get permission to pre-empt; just notify on another thread afterward
      2. Detail: on page fault, may delay of fault again on same point
      3. Detail: may wait until next time a kernel thread is available if no threads currently running

iii.
iv. SHOW EXAMPLE

f. Critical sections
    i. QUESTION: What is problem?
        1. Kernel may preempt SA running thread holding lock
    ii. SOLUTIONS:
        1. **Preemption control** kernel lets user decide which threads should not be preempted
            a. May require pinning memory to avoid page faults
            b. Yields control to user level from kernel
            c. To avoid deadlock, must be a guarantee – not just a wish
            d. **NOTE:** used in Solaris
        2. **Recovery**:
            a. When preempt thread holding a lock, can continue it instead of running scheduler. On release of lock, goes back to upcall and into scheduler
            b. Mechanism:
                i. Goal: zero overhead for common case
                    1. Solution: mark critical sections in assembly
                    2. Copy code to new place
                        a. On preempt, run new copy
                    3. New code returns control to scheduler on releasing a lock instead of continuing
                    4. Problem: locks acquired in one indirect function call, released in another
                        a. Use flags in this case
                ii. Key idea: use knowledge of source code for fast common case behavior – zero overhead.
                iii. Used in Linux for trap handling: certain places are marked as "safe" for traps, stores a fixup routine to recover. E.g. copy from user
            c. **THOUGHTS:**
                i. This shows what could be done to make it faster, but in practice not very realistic
                ii. Only for short, CPU-only critical sections
    iii. Performance:
        1. QUESTION: What do you want to show?
            a. A: no cost for cpu-bound operations

            b. A: Better than both for blocking operations
         2. As fast as fast threads when CPU bound
         3. Fixed amount better than kernel threads when I/O bound – no unnecessary blocking
      iv. QUESTION: where did we see this before?
         1. Spin, Exokernel – allow choice of what thread to run next when CPU given to a process

12. Processor allocation policy
    a. Space sharing: each app is dedicated a set of CPUs
    b. Time sharing: apps share time on a set of CPUs
    c. QUESTION: which is better?
       i. Almost all apps run better on fewer dedicated CPUs than more shared CPUs
13. Key points: Knowledge + control
    a. Kernel notifies User-level of what it is doing
    b. user level can control what runs next when kernel provides a processor
    c. Kernel retains control of proecessor
14. Issues/comments
    a. Preemption control is what survives – the approach they say has too much overhead
    b. User level threads largely died – for server applications, access to I/O, system calls important
       i. May be coming back for parallel applications
15. OVERALL SUMMARY
    a. Scheduler activation is like a thread, but always resumes at a single place (with context as a parameter) instead of returning where it was preempted
    b. Scheduler activations get to keep CPU when current thread blocks in kernel (often)
16. Microsoft Windows 7 User-mode threads
    a. basically scheduler activations