Threading and Synchronization

Fahd Albinali
Parallelism

• Parallelism and Pseudo-parallelism

• Why parallelize?

• Finding parallelism
  • Advantages: better load balancing, better scalability
  • Disadvantages: process/thread overhead and communication
Parrallelism – Cont’d

• distribute computation and data
  – Assign which processor does which computation
  – If memory is distributed, decide which processor stores which data (why is this?)
    • Data can be replicated also
  – Goals: minimize communication and balance the computational workload
    • Often conflicting goals
Parallelism – Cont’d

• synchronize and/or communicate
  – If shared-memory machine, synchronize
    • Both mutual exclusion and sequence control
      – Locks, semaphores, condition variables, barriers, reductions
  – If distributed-memory machine, communicate
    • Message passing
    • Usually communication involves implicit
      synchronization
Definition of Thread

• Thread
  – Lightweight process (LWP)
  – Threads of instructions or thread of control
  – Shares address space and other global information with its process
  – Registers, stack, signal masks and other thread-specific data are local to each thread

• Threads may be managed by the operating system or by a user application

• Examples: Win32 threads, C-threads, Pthreads
• Threads have become prominent due to trends in
  – Software design
    • More naturally expresses inherently parallel tasks
  – Performance
    • Scales better to multiprocessor systems
  – Cooperation
    • Shared address space incurs less overhead than IPC
• Thread states
  – Born state
  – Ready state (Runnable state)
  – Running state
  – Dead state
  – Blocked state
  – Waiting state
  – Sleeping state
    • Sleep interval specifies for how long a thread will sleep
Figure 4.3 User-level threads.
• User-level threads perform threading operations in user space
  – Threads are created by runtime libraries that cannot execute
    privileged instructions or access kernel primitives directly

• User-level thread implementation
  – Many-to-one thread mappings
    • Operating system maps all threads in a multithreaded process to single
      execution context
    • Advantages
      – User-level libraries can schedule its threads to optimize performance
      – Synchronization performed outside kernel, avoids context switches
      – More portable
    • Disadvantage
      – Kernel views a multithreaded process as a single thread of control
        » Can lead to suboptimal performance if a thread issues I/O
        » Cannot be scheduled on multiple processors at once
Thread Signal Delivery

• Two types of signals
  – Synchronous:
    • Occur as a direct result of program execution
    • Should be delivered to currently executing thread
  – Asynchronous
    • Occur due to an event typically unrelated to the current instruction
    • Threading library must determine each signal’s recipient so that asynchronous signals are delivered properly

• Each thread is usually associated with a set of pending signals that are delivered when it executes

• Thread can mask all signals except those that it wishes to receive
Thread Termination

• Thread termination (cancellation)
  – Differs between thread implementations
  – Prematurely terminating a thread can cause subtle errors in processes because multiple threads share the same address space
  – Some thread implementations allow a thread to determine when it can be terminated to prevent process from entering inconsistent state
Linux task state-transition diagram.
Example Concurrent Program
(x is shared, initially 0)

• code for Thread 0
  foo( )
  x := x+1

• code for Thread 1
  bar( )
  x := x+2

Assume both threads execute at about the same time.

What’s the output?
Example Concurrent Program (cont.)

• One possible execution order is:
  – Thread 0: R1 := x (R1 == 0)
  – Thread 1: R2 := x (R2 == 0)
  – Thread 1: R2 := R2 + 2 (R2 == 2)
  – Thread 1: x := R2 (x == 2)
  – Thread 0: R1 := R1 + 1 (R1 == 1)
  – Thread 0: x := R1 (x == 1)

• Final value of x is 1 (!!)

• Question: what if Thread 1 also uses R1?
1. head

   Insert: elem->next := head;

2. head

   Delete: t := head;

3. head

   elem

   elem

   t

4. head

   elem

   t

   Insert: head := elem;

5. head

   elem

   t

   Delete: head := head->next;

   Delete: return t;
Some Definitions

• Race condition
  – when output depends on ordering of thread execution
  – more formally:
    • (1) two or more threads access a shared variable with no synchronization, and
    • (2) at least one of the threads writes to the variable
More Definitions

• Atomic Operation
  – an operation that, once started, runs to completion
    • note: more precisely, logically runs to completion
  – indivisible
  – in this class: loads and stores
    • meaning: if thread A stores “1” into variable x and thread B
      stores “2” into variable x about about the same time, result
      is either “1” or “2”
  – <await (B) S>
    • atomically (evaluate B, wait until true, execute S)
Critical Section

• section of code that:
  – must be executed by one thread at a time
  – if more than one thread executes at a time, have a race condition
  – ex: linked list from before
  • Insert/Delete code forms a critical section
  • What about just the Insert or Delete code?
    – is that enough, or do both procedures belong in a single critical section?
Critical Section (CS) Problem

• Provide entry and exit routines:
  – all threads must call entry before executing CS
  – all threads must call exit after executing CS
  – thread must not leave entry routine until it’s safe

• CS solution properties
  – Mutual exclusion: at most one thread is executing CS
  – Absence of deadlock: two or more threads trying to get into CS => at least one succeeds
  – Absence of unnecessary delay: if only one thread trying to get into CS, it succeeds
  – Eventual entry: thread eventually gets into CS
Structure of threads for Critical Section problem

Threads do the following:

while (1) {
    do other stuff (non-critical section)
call enter
execute CS
call exit
do other stuff (non-critical section)
}
Critical Section Assumptions

• Threads must call enter and exit
• Threads must not die or quit inside a critical section
• Threads **can** be context switched inside a critical section
  – this does **not** mean that the newly running thread may enter the critical section
Hardware Support

• Provide instruction that is:
  – atomic
  – fairly easy for hardware designer to implement

• Read/Modify/Write
  – atomically read value from memory, modify it in some way, write it back to memory

• Use to develop simpler critical section solution for any number of threads
Test-and-Set

Many machines have it

function TS(var target: bool) returns bool
  var b: bool := target;  /* return old value */
  target := true;
  return b;

Executes atomically
Basic Idea with Atomic Instructions

• Each thread has a local flag
• One variable shared by all threads
• Use the atomic instruction with flag, shared variable
  – on a change, allow thread to go in
  – other threads will not see this change
• When done with CS, set shared var back to initial state
Problems with busy-waiting CS solution

- Complicated
- Inefficient
  - consumes CPU cycles while spinning
- Priority inversion problem
  - low priority thread in CS, high priority thread spinning can end up causing deadlock
  - example: Mars Pathfinder problem

Want to block when waiting for CS
Locks

• Two operations:
  – Acquire (get it, if can’t go to sleep)
  – Release (give it up, possibly wake up a waiter)
• entry( ) is then just Acquire(lock)
• exit( ) is just Release(lock)

Lock is shared among all threads
Problems with Locks

• Not general
  – only solve simple critical section problem
  – can’t do any more general synchronization
  – often must enforce strict orderings betw. threads

• Condition synchronization
  – need to wait until some condition is true
  – example: bounded buffer (next slide)
  – example: thread join
Semaphores (Dijkstra)

• Semaphore is an object
  – contains a (private) value and 2 operations

• **Semaphore value must be nonnegative**

• P operation (atomic):
  – if value is 0, block; else decrement value by 1

• V operation (atomic):
  – if thread blocked, wake up; else value++

• Semaphores are “resource counters”
Critical Sections with Semaphores

sem mutex := 1
entry( )
   – P(mutex)
exit( )
   – V(mutex)

• Semaphores more powerful than locks
• For mutual exclusion, initialize semaphore to 1
Bounded Buffer
(1 producer, 1 consumer)

char buf[n], int front := 0, rear := 0
sem empty := n, full := 0

Producer()
    do forever...
    produce message m
    P(empty)
    buf[rear] := m;
    rear := rear “+” 1
    V(full)

Consumer()
    do forever...
    P(full)
    m := buf[front]
    front := front “+” 1
    V(empty)
    consume m
Bounded Buffer (multiple producers and consumers)

char buf[n], int front := 0, rear := 0
sem empty := n, full := 0, mutexC := 1, mutexP := 1

Producer()  Consumer()

    do forever...
    produce message m
    P(empty); P(mutexP)
    buf[rear] := m;
    rear := rear “+” 1
    V(mutexP); V(full)

    do forever...
    P(full); P(mutexC)
    m := buf[front]
    front := front “+” 1
    V(mutexC); V(empty)
    consume m
Scratching the surface

- Readers/Writers
- Barriers
- Monitors
- Fairness/Enforcing ordering