Chapter 12 :: Concurrency

Programming Language Pragmatics

Michael L. Scott

ELSEVIER

Background and Motivation

- A PROCESS or THREAD is a potentially-active execution context
- Classic von Neumann (stored program) model of computing has single thread of control
- Parallel programs have more than one
- A process can be thought of as an abstraction of a physical PROCESSOR

Background and Motivation

- Processes/Threads can come from
  - multiple CPUs
  - kernel-level multiplexing of single physical machine
  - language or library level multiplexing of kernel-level abstraction
- They can run
  - in true parallel
  - unpredictably interleaved
  - run-until-block
- Most work focuses on the first two cases, which are equally difficult to deal with

Background and Motivation

- Two main classes of programming notation
  - synchronized access to shared memory
  - message passing between processes that don't share memory
- Both approaches can be implemented on hardware designed for the other, though shared memory on message-passing hardware tends to be slow

Background and Motivation

- Race conditions
  - A race condition occurs when actions in two processes are not synchronized and program behavior depends on the order in which the actions happen
  - Race conditions are not all bad; sometimes any of the possible program outcomes are ok (e.g. workers taking things off a task queue)

Background and Motivation

- Race conditions (we want to avoid race conditions):
  - Suppose processors A and B share memory, and both try to increment variable X at more or less the same time
  - Very few processors support arithmetic operations on memory, so each processor executes
    - LOAD X
    - INC
    - STORE X
  - Suppose X is initialized to 0. If both processors execute these instructions simultaneously, what are the possible outcomes?
    - could go up by one or by two

Background and Motivation

- Synchronization
  - SYNCHRONIZATION is the act of ensuring that events in different processes happen in a desired order
  - Synchronization can be used to eliminate race conditions
  - In our example we need to synchronize the increment operations to enforce MUTUAL EXCLUSION on access to X
  - Most synchronization can be regarded as either
    - Mutual exclusion (making sure that only one process is executing a CRITICAL SECTION [touching a variable, for example] at a time),

      or as
    - CONDITION SYNCHRONIZATION, which means making sure that a given process does not proceed until some condition holds (e.g. that a variable contains a given value)

Background and Motivation

- One might be tempted to think of mutual exclusion as a form of condition synchronization (the condition being that nobody else is in the critical section), but it isn't
  - The distinction is basically *existential v. universal* quantification
    - Mutual exclusion requires multi-process consensus
- We do NOT in general want to over-synchronize
  - That eliminates parallelism, which we generally want to encourage for performance
- Basically, we want to eliminate "bad" race conditions, i.e., the ones that cause the program to give incorrect results

Background and Motivation

- Historical development of shared memory ideas
  - To implement synchronization you have to have something that is ATOMIC
    - that means it happens all at once, as an indivisible action
    - In most machines, reads and writes of individual memory locations are atomic (note that this is not trivial; memory and/or busses must be designed to arbitrate and serialize concurrent accesses)
    - In early machines, reads and writes of individual memory locations were *all* that was atomic
  - To simplify the implementation of mutual exclusion, hardware designers began in the late 60's to build so-called read-modify-write, or fetch-and-phi, instructions into their machines

Concurrent Programming Fundamentals

- SCHEDULERS give us the ability to "put a thread/process to sleep" and run something else on its process/processor
  - start with coroutines
  - make uniprocessor run-until-block threads
  - add preemption
  - add multiple processors

Concurrent Programming Fundamentals

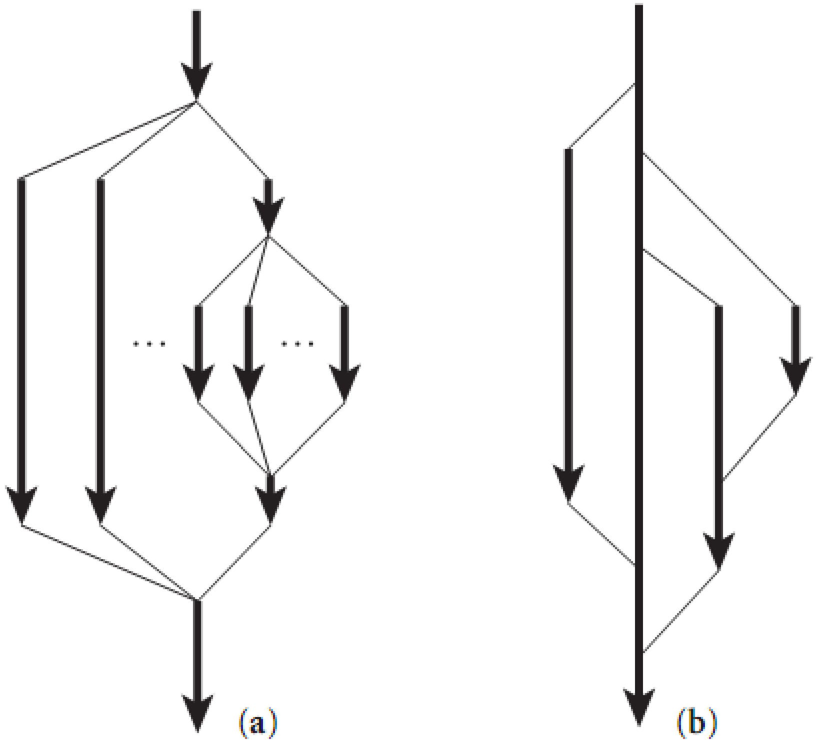

Figure 12.5 **Lifetime of concurrent threads.** With co-begin, parallel loops, or launch-at-elaboration (a), threads are always properly nested. With fork/join (b), more general patterns are possible.

Concurrent Programming Fundamentals

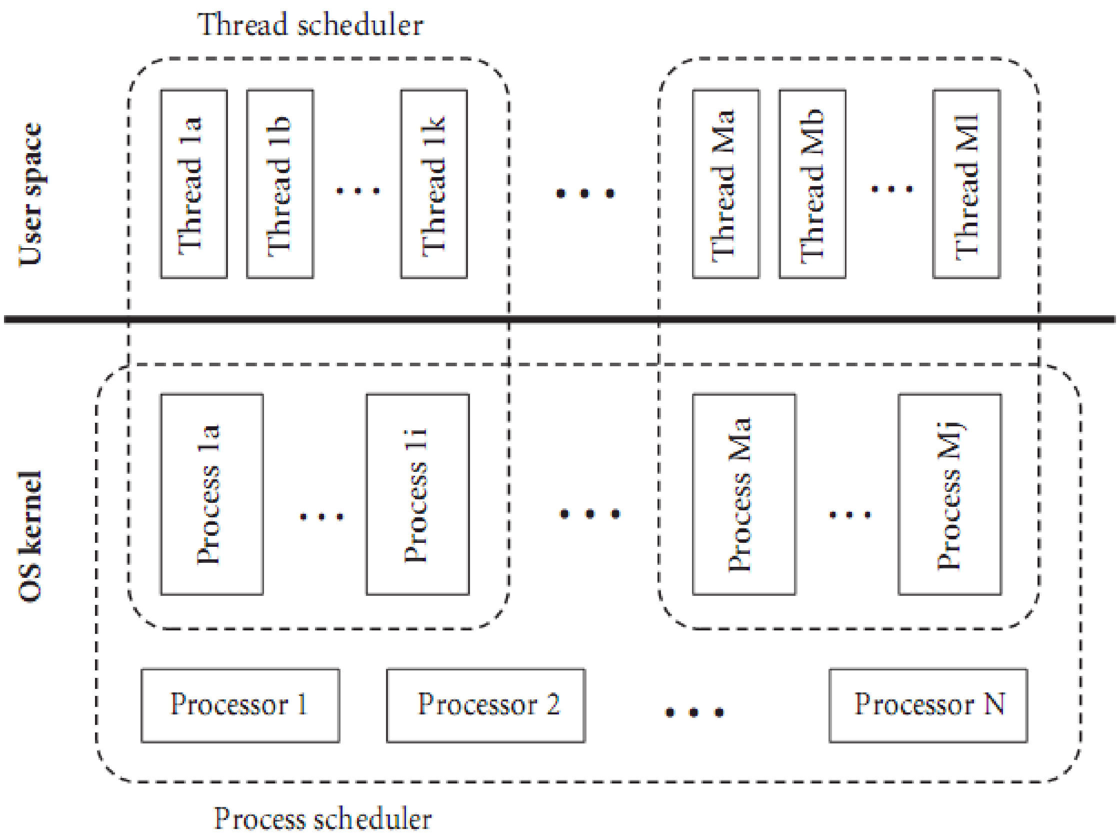

Figure 12.6 Two-level implementation of threads. A thread scheduler, implemented in a library or language run-time package, multiplexes threads on top of one or more kernel-level processes, just as the process scheduler, implemented in the operating system kernel, multiplexes processes on top of one or more physical processors.

Concurrent Programming Fundamentals

- Coroutines
  - Multiple execution contexts, only one of which is active
  - Transfer (other):
    - save all callee-saves registers on stack, including ra and fp
    - *current := sp
    - current := other
    - sp := *current
    - pop all callee-saves registers (including ra, but NOT sp!)
    - return (into different coroutine!)
  - Other and current are pointers to CONTEXT BLOCKs
    - Contains sp; may contain other stuff as well (priority, I/O status, etc.)
  - No need to change PC; always changes at the same place
  - Create new coroutine in a state that looks like it's blocked in transfer. (Or maybe let it execute and then "detach". That's basically early reply)

Concurrent Programming Fundamentals

- Run-until block threads on a single process
  - Need to get rid of explicit argument to transfer
  - Ready list data structure: threads that are runnable but not running

```
procedure reschedule:
    t : cb := dequeue(ready_list)
    transfer(t)
```

  - To do this safely, we need to save 'current' somewhere - two ways to do this:
    - Suppose we're just relinquishing the processor for the sake of fairness (as in MacOS or Windows 3.1):

```
procedure yield:
    enqueue (ready_list, current)
    reschedule
```

    - Now suppose we're implementing synchronization:

```
sleep_on(q)
    enqueue(q, current)
    reschedule
```

  - Some other thread/process will move us to the ready list when we can continue

Concurrent Programming Fundamentals

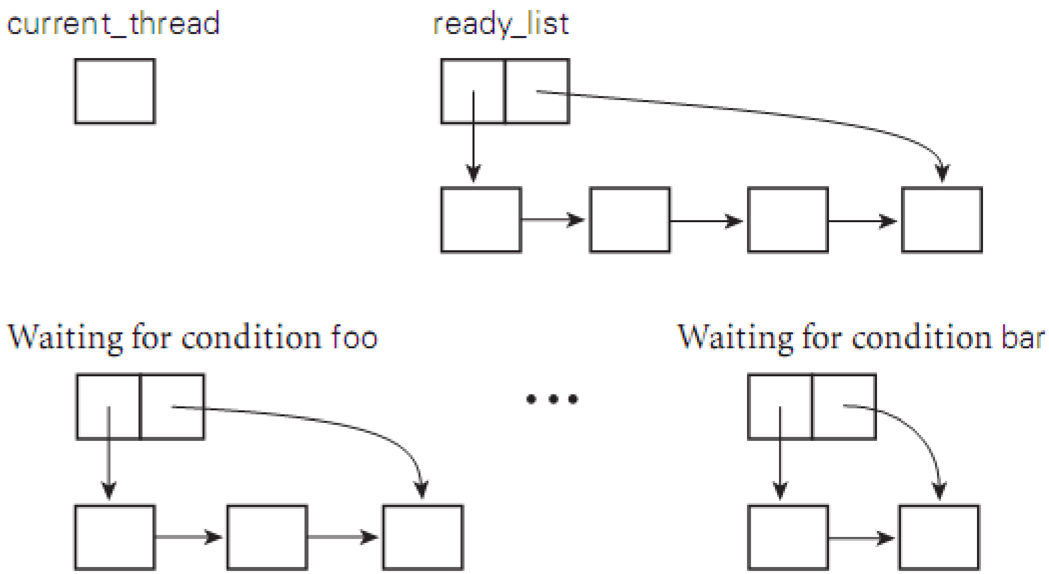

Figure 12.7 **Data structures of a simple scheduler.** A designated current_thread is running. Threads on the ready list are runnable. Other threads are blocked, waiting for various conditions to become true. If threads run on top of more than one OS-level process, each such process will have its own current_thread variable. If a thread makes a call into the operating system, its process may block in the kernel.

Concurrent Programming Fundamentals

- Preemption
  - Use timer interrupts (in OS) or signals (in library package) to trigger involuntary yields
  - Requires that we protect the scheduler data structures:

```
procedure yield:
    disable_signals
    enqueue(ready_list, current)
    Reschedule
    re-enable_signals
```

  - Note that reschedule takes us to a different thread, possibly in code other than yield Invariant: EVERY CALL to reschedule must be made with signals disabled, and must re-enable them upon its return

```
disable_signals
if not <desired condition>
    sleep_on <condition queue>
re-enable signals
```

Concurrent Programming Fundamentals

- Multiprocessors
  - Disabling signals doesn't suffice:

```
procedure yield:
        disable_signals
        acquire(scheduler_lock)        // spin lock
        enqueue(ready_list, current)
        reschedule
        release(scheduler_lock)
        re-enable_signals

disable_signals
acquire(scheduler_lock)        // spin lock
if not <desired condition>
     sleep_on <condition queue>
release(scheduler_lock)
re-enable signals
```

Implementing Synchronization

- Condition synchronization with atomic reads and writes is easy
  - You just cast each condition in the form of "location X contains value Y" and you keep reading X in a loop until you see what you want
- Mutual exclution is harder
  - Much early research was devoted to figuring out how to build it from simple atomic reads and writes
  - Dekker is generally credited with finding the first correct solution for two processes in the early 1960s
  - Dijkstra published a version that works for N processes in 1965
  - Peterson published a much simpler two-process solution in 1981

Implementing Synchronization

- Repeatedly reading a shared location until it reaches a certain value is known as SPINNING or BUSY-WAITING
- A busy-wait mutual exclusion mechanism is known as a SPIN LOCK
  - The problem with spin locks is that they waste processor cycles
  - Sychronization mechanisms are needed that interact with a thread/process scheduler to put a process to sleep and run something else instead of spinning
  - Note, however, that spin locks are still valuable for certain things, and are widely used
    - In particular, it is better to spin than to sleep when the expected spin time is less than the rescheduling overhead

Implementing Synchronization

- SEMAPHORES were the first proposed SCHEDULER-BASED synchronization mechanism, and remain widely used
- CONDITIONAL CRITICAL REGIONS and MONITORS came later
- Monitors have the highest-level semantics, but a few sticky semantic problem - they are also widely used
- Synchronization in Java is sort of a hybrid of monitors and CCRs (Java 3 will have true monitors.)
- Shared-memory synch in Ada 95 is yet another hybrid

Implementing Synchronization

- A semaphore is a special counter
- It has an initial value and two operations, P and V, for changing that value
- A semaphore keeps track of the difference between the number of P and V operations that have occurred
- A P operation is delayed (the process is de-scheduled) until #P-#V <= C, the initial value of the semaphore

Implementing Synchronization

```
shared scheduler_lock : low_level_lock
shared ready_list : queue of thread
per-process private current_thread : thread

procedure reschedule
    -- assume that scheduler_lock is already held
    -- and that timer signals are disabled
    t : thread
    loop
        t := dequeue(ready_list)
        if t ≠ null
            exit
        -- else wait for a thread to become runnable
        release_lock(scheduler_lock)
        -- window allows another thread to access ready_list
        -- (no point in reenabling signals;
        -- we're already trying to switch to a different thread)
        acquire_lock(scheduler_lock)
    transfer(t)
    -- caller must release scheduler_lock
    -- and reenable timer signals after we return

procedure yield
    disable_signals
    acquire_lock(scheduler_lock)
    enqueue(ready_list, current_thread)
    reschedule
    release_lock(scheduler_lock)
    reenable_signals

procedure sleep_on(ref Q : queue of thread)
    -- assume that caller has already disabled timer signals
    -- and acquired scheduler_lock, and will reverse
    -- these actions when we return
    enqueue(Q, current_thread)
    reschedule
```

Note: a possible implementation is shown on the next slide

Figure 12.12 Pseudocode for part of a simple reentrant (parallelism-safe) scheduler. Every process has its own copy of current_thread. There is a single shared scheduler_lock and a single ready_list. If processes have dedicated processors, then the low_level_lock can be an ordinary spin lock; otherwise it can be a "spin-then-yield" lock (Figure 12.13). The loop inside reschedule busy-waits until the ready list is nonempty. The code for sleep_on cannot disable timer signals and acquire the scheduler lock itself, because the caller needs to test a condition and then block as a single atomic operation.

Implementing Synchronization

```
type semaphore = record
      N : integer –– usually initialized to something nonnegative
      Q : queue of threads

procedure P(ref S : semaphore)
      disable_signals
      acquire_lock(scheduler_lock)
      S.N –:= 1
      if S.N < 0
            sleep_on(S.Q)
      release_lock(scheduler_lock)
      reenable_signals

procedure V(ref S : semaphore)
      disable_signals
      acquire_lock(scheduler_lock)
      S.N +:= 1
      if N ≤ 0
            –– at least one thread is waiting
            enqueue(ready_list, dequeue(S.Q))
      release_lock(scheduler_lock)
      reenable_signals
```

Figure 12.14 Semaphore operations, for use with the scheduler code of Figure 12.12.

Implementing Synchronization

```
shared buf : array [1..SIZE] of bdata
shared next_full, next_empty : integer := 1, 1
shared mutex : semaphore := 1
shared empty_slots, full_slots : semaphore := SIZE, 0

procedure insert(d : bdata)
    P(empty_slots)
    P(mutex)
    buf[next_empty] := d
    next_empty := next_empty mod SIZE + 1
    V(mutex)
    V(full_slots)

function remove : bdata
    P(full_slots)
    P(mutex)
    d : bdata := buf[next_full]
    next_full := next_full mod SIZE + 1
    V(mutex)
    V(empty_slots)
    return d
```

Figure 12.15 Semaphore-based code for a bounded buffer. The mutex binary semaphore protects the data structure proper. The full_slots and empty_slots general semaphores ensure that no operation starts until it is safe to do so.

Implementing Synchronization

- It is generally assumed that semaphores are fair, in the sense that processes complete P operations in the same order they start them
- Problems with semaphores
  - They're pretty low-level.
    - When using them for mutual exclusion, for example (the most common usage), it's easy to forget a P or a V, especially when they don't occur in strictly matched pairs (because you do a V inside an if statement, for example, as in the use of the spin lock in the implementation of P)
  - Their use is scattered all over the place.
    - If you want to change how processes synchronize access to a data structure, you have to find all the places in the code where they touch that structure, which is difficult and error-prone

Language-Level Mechanisms

- Monitors were an attempt to address the two weaknesses of semaphores listed above
- They were suggested by Dijkstra, developed more thoroughly by Brinch Hansen, and formalized nicely by Hoare (a real cooperative effort!) in the early 1970s
- Several parallel programming languages have incorporated monitors as their fundamental synchronization mechanism
  - none, incorporates the precise semantics of Hoare's formalization

Language-Level Mechanisms

- A monitor is a shared object with operations, internal state, and a number of condition queues. Only one operation of a given monitor may be active at a given point in time
- A process that calls a busy monitor is delayed until the monitor is free
  - On behalf of its calling process, any operation may suspend itself by waiting on a condition
  - An operation may also signal a condition, in which case one of the waiting processes is resumed, usually the one that waited first

Language-Level Mechanisms

- The precise semantics of mutual exclusion in monitors are the subject of considerable dispute. Hoare's original proposal remains the clearest and most carefully described
  - It specifies two bookkeeping queues for each monitor: an entry queue, and an urgent queue
  - When a process executes a signal operation from within a monitor, it waits in the monitor's urgent queue and the first process on the appropriate condition queue obtains control of the monitor
  - When a process leaves a monitor it unblocks the first process on the urgent queue or, if the urgent queue is empty, it unblocks the first process on the entry queue instead

Language-Level Mechanisms

- Building a correct monitor requires that one think about the "monitor invariant“. The monitor invariant is a predicate that captures the notion "the state of the monitor is consistent."
  - It needs to be true initially, and at monitor exit
  - It also needs to be true at every wait statement
  - In Hoare's formulation, needs to be true at every signal operation as well, since some other process may immediately run
- Hoare's definition of monitors in terms of semaphores makes clear that semaphores can do anything monitors can
- The inverse is also true; it is trivial to build a semaphores from monitors (Exercise)