“Programming with Threads”
Birrell

- Multiprocessors
- Waiting for slow devices
- Human users
- Shared network servers multiplexing among client (each client served by its own thread)
- Maintenance tasks

Hardware Assistance for Synchronization

- Most modern architectures provide some support for building synchronization: atomic \texttt{read-modify-write} instructions.

  \textbf{Example:} \texttt{test-and-set (loc, reg)}

  \[
  \text{sets bit to 1 in the new value of loc;
  returns old value of loc in reg.}
  \]

- Other examples: \texttt{compare-and-swap, fetch-and-op}

Busywaiting with Test-and-Set

- Declare a shared memory location to represent a \texttt{busyflag} on the critical section we are trying to protect.

- \texttt{enter\_region} (or \texttt{acquiring} the “lock”):

  \begin{verbatim}
  waitloop: tsl busyflag, R0  // R0 = busyflag; busyflag = 1
  bnz R0, waitloop // was it already set?
  \end{verbatim}

- \texttt{exit\_region} (or \texttt{releasing} the “lock”):

  \[
  \text{busyflag} = 0
  \]

Pros and Cons of Busywaiting

- \textbf{Key characteristic} - the “waiting” process is actively executing instructions in the CPU and using memory cycles.

- \textbf{Appropriate when}: - High likelihood of finding the critical section unoccupied (don’t take context switch just to find that out) or estimated wait time is very short

- \textbf{Disadvantages}:
  - Wastes resources (CPU, memory, bus bandwidth)
  - Cache-miss heavy
  - Looks busy if system is observing behavior
Better Implementations from Multiprocessor Domain

- Dealing with contention of Test&Set spinlocks:
  - Don’t execute test&set so much
  - Spin without generating bus traffic
- Test&Set with Backoff
  - Insert delay between test&set operations (not too long)
  - Exponential seems good (k * ci)
  - Not fair
- Test-and-Test&Set
  - Spin (test) on local cached copy until it gets invalidated, then issue test&set
  - Intuition: No point in trying to set the location until we know that it’s not set, which we can detect when it gets invalidated...
  - Still contention after invalidate
  - Still not fair
- Analogies for Energy?

Blocking Synchronization

- OS implementation involving changing the state of the “waiting” process from running to blocked.
- Need some synchronization abstraction known to OS - provided by system calls.
  - mutex locks with operations acquire and release
  - semaphores with operations P and V (down, up)
  - condition variables with wait and signal

Template for Implementing Blocking Synchronization

- Associated with the lock is a memory location (busy) and a queue for waiting threads/processes.
- Acquire syscall:
  ```
  while (busy) {enqueue caller on lock's queue} /*upon waking to nonbusy lock*/
  busy = true;
  ```
- Release syscall:
  ```
  busy = false;
  /* wakup */ move any waiting threads to Ready queue
  ```

Pros and Cons of Blocking

- Waiting processes/threads don’t consume resources
- Appropriate: when the cost of a system call is justified by expected waiting time
  - High likelihood of contention for lock
  - Long critical sections
- Disadvantage: OS involvement
  -> overhead
Semaphores

• Well-known synchronization abstraction
• Defined as a non-negative integer with two atomic operations
  \[ P(s) \cdot \text{[wait until } s > 0\text{; } s --\text{]} \]
  \[ V(s) \cdot \text{[} s++\text{]} \]
• The atomicity and the waiting can be implemented by either busywaiting or blocking solutions.

Semaphore Usage

• Binary semaphores can provide mutual exclusion (solution of critical section problem)
• Counting semaphores can represent a resource with multiple instances (e.g. solving producer/consumer problem)
• Signalling events (persistent events that stay relevant even if nobody listening right now)

The Critical Section Problem

```c
while (1)
{
  ...other stuff...
  P(mutex)
  ...critical section...
  V(mutex)
}
```

Semaphore: mutex initially 1

Knowing shared from private...

SRC Thread Primitives

• SRC thread primitives
  – Thread = Fork (procedure, args)
  – result = Join (thread)
  – LOCK mutex DO critical section END
  – Wait (mutex, condition)
  – Signal (condition)
  – Broadcast (condition)
  – Acquire (mutex), Release (mutex) //more dangerous
Monitor Abstraction

- Encapsulates shared data and operations with mutual exclusive use of the object (an associated lock).
- Associated Condition Variables with operations of Wait and Signal.

Condition Variables

- We build the monitor abstraction out of a lock (for the mutual exclusion) and a set of associated condition variables.
- **Wait on condition**: releases lock held by caller, caller goes to sleep on condition’s queue. When awakened, it must reacquire lock.
- **Signal condition**: wakes up one waiting thread.
- **Broadcast**: wakes up all threads waiting on this condition.

**Monitor Abstraction**

```
EnQ(aquire (lock); 
   if (head == null) 
      (head = item; 
      signal (lock, notEmpty); 
   else tail->next = item; 
   tail = item; 
   release(lock;)
)
```

```
deQ(aquire (lock); 
   if (head == null) 
      wait (lock, notEmpty); 
   item = head; 
   if (tail == head) tail = null; 
   head=item->next; 
   release(lock;)
)
```
Monitor Abstraction

EnQ (acquire (lock);
  if (head == null)
    (head = item;
      signal (lock, notEmpty);)
  else tail->next = item;
  tail = item;
  release (lock);
)
deQ (acquire (lock);
  if (head == null)
    wait (lock, notEmpty);
  item = head;
  if (tail == head) tail = null;
  head=item->next;
  release (lock);)

Monitor Abstraction

EnQ (acquire (lock);
  if (head == null)
    (head = item;
      signal (lock, notEmpty);)
  else tail->next = item;
  tail = item;
  release (lock);
)
deQ (acquire (lock);
  if (head == null)
    wait (lock, notEmpty);
  item = head;
  if (tail == head) tail = null;
  head=item->next;
  release (lock);
)
Pitfalls

- Race conditions, failure to implement mutual exclusion within critical sections of code.
- Deadlock
- Starvation
- Priority inversion
- Performance Issues (including energy implications)
  - Difficulty of detecting idleness with busy waiting synchronization
  - Lock granularity issues

5 Dining Philosophers

Template for Philosopher

```c
while (food available) {
  /*pick up forks*/
  eat;
  /*put down forks*/
  think awhile;
}
```

Naive Solution

```c
while (food available) {
  P(fork[left(me)]);
  P(fork[right(me)]);
  /*pick up forks*/
  eat;
  V(fork[left(me)]);
  V(fork[right(me)]);
  /*put down forks*/
  think awhile;
}
```
Philosophy 101
(or why 5DP is interesting)

- How to eat with your Fellows without causing Deadlock.
  - Circular arguments (the circular wait condition)
  - Not giving up on firmly held things (no preemption)
  - Infinite patience with Half-baked schemes (hold some & wait for more)
- Why Starvation exists and what we can do about it.

Circular Wait Condition

```
while (food available)
{
  if (me == 0) {P(fork[left(me)]); P(fork[right(me)]);}
  else {P(fork[right(me)]); P(fork[left(me)]); }
  eat;
  V(fork[left(me)]); V(fork[right(me)]);
  think awhile;
}
```

Ordered resources

Hold and Wait Condition

```
while (food available)
{
  P(mutex);
  while (forks [me] != 2)
      {blocking[me] = true; V(mutex); P(sleepy[me]); P(mutex);}
  forks [leftneighbor(me)] --; forks [rightneighbor(me)]--;
  V(mutex);
  eat;
  P(mutex); forks [leftneighbor(me)] ++; forks [rightneighbor(me)]++;
  if (blocking[leftneighbor(me)]) V(sleepy[leftneighbor(me)]);
  if (blocking(rightneighbor(me)]) V(sleepy[rightneighbor(me)]);
  V(mutex);}
  think awhile;
}
```

Starvation

The difference between deadlock and starvation is subtle:
- Once a set of processes are deadlocked, there is no future execution sequence that can get them out of it.
- In starvation, there does exist some execution sequence that is favorable to the starving process although there is no guarantee it will ever occur.
- Rollback and Retry solutions are prone to starvation.
- Continuous arrival of higher priority processes is another common starvation situation.
**Issues**

- Locking overhead (granularity)
- Broadcast vs. Signal and other causes of spurious wakeups
- Nested lock/condition variable problem
- Priority inversions

**Lock Granularity**

```
while (!required_conditions) wait (m, c);
```

- Why we use “while” not “if” — invariant not guaranteed
- Why use broadcast — using one condition queue for many reasons. Waking threads have to sort it out. Possibly better to separate into multiple conditions (more complexity to code)
Mars Pathfinder Example

- In July 1997 Pathfinder’s computer reset itself several times during data collection and transmission from Mars.
  - One of its processes failed to complete by a deadline, triggering the reset.

- Priority Inversion Problem
  - A low priority process held a mutual exclusion semaphore on a shared data structure but was preempted to let higher priority processes run.
  - The high priority process that failed to complete on time was blocked on this semaphore and priority inheritance was not enabled.
  - Meanwhile a bunch of medium priority processes ran, until finally the deadline ran out. The low priority semaphore-holding process never got the chance to run again in that time to the point of releasing the mutex.

Tricks (mixed syntax)

```plaintext
if (some_condition) // as a hint
{
  LOCK m DO
  if (some_condition) // the truth
    {stuff}
  END
}
```

Cheap to get info but must check for correctness; always a slow way.

More Tricks

General pattern:
```
while (! required_conditions) wait (m, c);
```

Broadcast works because waking up too many is OK (correctness-wise) although a performance impact.

```plaintext
LOCK m DO
  ...
  deferred_signal = true;
END
```

Spurious lock conflicts caused by signals inside critical section and threads waking up to test mutex before it gets released.

Alerts

Thread state contains flag, alert pending.

Exception alerted:
```
TRY
while (empty)
   AlertWait (m, nonempty); return (nextchar());
EXCEPT
   Thread.Alerted:
      return (eof);
```

Alert (thread):
```
   alert-pending to true, wakeup a waiting thread
   AlertWait (mutex, condition)
      if alert-pending set to false and raise exception
         else wait as usual
   Boolean b = TestAlert()
```

tests and clear alert-pending
Using Alerts

```c
sibling = Fork (proc, arg);
while (!done)
{ done = longComp();
  if (done) Alert (sibling);
  else done = TestAlert();
}
```

Wisdom

<table>
<thead>
<tr>
<th>Do s</th>
<th>Don’t s</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Reserve using alerts for when you don’t know what is going on</td>
<td>• Call into a different abstraction level while holding a lock</td>
</tr>
<tr>
<td>• Only use if you forked the thread</td>
<td>• Move the “last” signal beyond scope of Lock</td>
</tr>
<tr>
<td>• Impose an ordering on lock acquisition</td>
<td>• Acquire lock, fork, and let child release lock</td>
</tr>
<tr>
<td>• Write down invariants that should be true when locks aren’t being held</td>
<td>• Expect priority inheritance since few implementations</td>
</tr>
<tr>
<td></td>
<td>• Pack data and expect fine grain locking to work</td>
</tr>
</tbody>
</table>