Synchronization in an OS

1. Questions from reviews:
    a. What should be evaluated? How evaluate a programming model?
2. Context – Pilot development
    a. Authors were writing Pilot in Mesa – needed a way to handle multithreading
        i. For a uniprocessor, with lots of "processes" == threads
    b. Written by a bunch of "do the right thing" kind of people
    c. Recent work by Per Brinch Hanson and C. A. R. Hoare (Tony) developed Monitors
3. What are the problems they are solving?
    a. What are the right language-level constructs for synchronization?
    b. How do you use language-level constructs for synchronization within an OS?
4. Why is this an OS and not a PL problem?
    a. For 30 years, only OS people had concurrency – between processes in the kernel. Programs were almost all single threaded.
5. Synchronization Needs
    a. Need atomicity
        i. Concurrent updates without ordering required:
        ii. Example: Credit(), debit()
    b. Need ordering:
        i. Make sure some operations happen after others
        ii. Example: bounded buffer: consumer has to run after producer
        iii. Initialization: start threads, they all wait for initialization to complete before proceeding
6. What makes this an interesting problem?
    a. Granularity
        i. Fine-grained locking needed for scalable performance on a multiprocessor (see thread alternatives)
        ii. Fine-grained locking needed for responsiveness on a uniprocessor
        iii. Fine-grained locking is hard to get right
            1. Why?
                a. Must acquire locks in canonical order to avoid deadlock
                b. Example: back balance transfer:
                c. Transfer (queue x, queue y, Obj elem) {
```
x.lock();
y.lock();
x.dequeue(obj);
y.enqueue(obj);
y.unlock();
x.unlock();
```
                d. Can cause deadlock if called with transfer(x,y,z) and (y,x,w)
        iv. Coarse-grained locking scales poorly

            1. Can have many unrelated objects protected by one lock; e.g. a lock on all open files
            2. Could have a lock per file
b. Expressing "conditional synchronization":
    i. Want to sometimes wait for something specific to happen
    ii. Example:
        1. Wait for a buffer to be full or empty
        2. Wait for a bunch of workers to complete
        3. Wait for readers to finish before writing
    iii. Need to express what the "condition" is being waited for, need to detect when the condition becomes true
c. OS needs more than critical sections/mutual exclusion; needs ability to wait for things and wakeup
    i. How do you make sure you get notified when to wake up?

```
if (queue_empty)
  wait_for_data();
process_data();
```

d. Programmers want simple ways to do asynchronous tasks.
    i. Synchronous version:
        1. Buffer = readline(terminal)
    ii. In mesa:
        1. p = FORK readline(terminal)
        2. Buffer = join p
    iii. Complex semantics:
        1. What if terminal (input parameter) changes after fork?
            a. Does new thread make a copy, or have reference to the original one that changed?
            b. In C – pointers to local variables on the stack may get overwritten when procedure returns.
    iv. Detaching a thread
        1. Detach p → nobody will wait for p
        2. Race conditions around data used in p and in other thread:
            a. X = malloc()
            b. P = fork f(x);
            c. Detach p
            d. **When is it safe to free x?**
                i. **Answer: f(x) has to free x (or GC)**
    v.
e. Composability
    i. What if you have code like this:

```
f() {
  lock(x);
  a();
  unlock(x);
```

```
}
g() {
  lock(x);
  b();
 unlock(x);
}

a() {

  if (no_data)   wait();
}
b() {
 no_data = FALSE;
}
```

   ii. Critical to (a) allow blocking in a critical section for composability, but conditional synchronization puts limits on it
      1. an never set no_data to false

f. Correctness
   i. Easy to forget to lock things
      1. E.g. failure to lock when updating shared state
   ii. Easy to forget to unlock things:
      1. ```
         Lock(x);
          if (do_something(x) == EFAIL) {
           return(EFAIL);
         …
         unlock(x)
         ```
         a.

g. Priorities
   i. May have different priorities; need to ensure liveness
   ii. E.g. priority inversion:
      1. Low_priority: lock(x) → success
         high_priority:lock(x) → block
         medium_priority: execute something

      2. Result: high priority code is blocked by low priority code, which is blocked by medium priority

h. Interacting with hardware
   i. Want to execute code in response to hardware events (interrupts)
   ii. How does this interact with running code in a critical section?
      1. Pre-empt code and run new code? May be unsafe; should disable interrupts
   iii. Schedule some code to run later?

7. Earlier solutions
   a. Semaphores: too naked
         ```

i. Easy to get wrong, forget to signal or wait, etc.
ii. Example:

```
semaphore fillCount = 0; // items produced
semaphore emptyCount = BUFFER_SIZE; // remaining
space

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
        putItemIntoBuffer(item);
        up(fillCount);
          }
      }

      procedure consumer() {
          while (true) {
              down(fillCount);
              item = removeItemFromBuffer();
              up(emptyCount);
              consumeItem(item);
          }
      }
```

b. Conditional critical region – early 70's approach
   i. Attach "regions" to code/data (a lock)
   ii. Basic critical regions for locking:
      1. with R do {
         a. code
      2. }
      3. Like a java synchronized statement
   iii. Conditional critical regions: waiting for things to happen
      1. with R when (!buffer_empty) do {
      2. do_work();
      3. }

```
int Count = 0; // items produced
int BUFFER_SIZE; // remaining space

procedure producer() {
```

```
    while (true) {
        item = produceItem();
        with R when (count < BUFFER_SIZE0) {
               putItemIntoBuffer(item);
               Count++;
        }
    }

    procedure consumer() {
        while (true) {
            with R when (count > 0) {
                item = removeItemFromBuffer();
                count--;
            }
            consumeItem(item);
        }
    }
```

- iv. Implementation:
  1. Re-evaluate predicate after anyone leaves the region, decide who to take
- v. Issues:
  1. a bit complex to re-evaluate after every region entry, could be slow
  2. cannot do any work before waiting
- c. Windows events
  - i. Usage:
    1. setEvent()
    2. WaitForSingleObjects() to wait for it
    3. Manual reset – have to be rest
       - a. Good to wait for something to start, that happens just once
    4. Automatic reset – resets when someone wake up on it
       - a. Can be used for bounded buffer or to wake up a single thread to respond to something
       - b. No queue (unlike a semaphore); does not remember history
    5. No atomicity for modifying something then signaling
       - a. Need to guarantee when signaling waker will see change

8. Monitor solution
   - a. Tie locking to language, so it gets used in the right places

        i. Monitor == class
        ii. Entry procedure
            1. Public member function
            2. acquire lock on entry
        iii. Internal procedure == private member function
        iv. Public procedure == no acquire lock on entry
    b. Useful sync. Operations – powerful
        i. Wait = release lock, wait for a condition variable to be "notified"
        ii. Notify = a hint, that a logical condition may have become true
            1. QUESTION: Why relax semantics over Hoare, where it was guaranteed?
            2. A: efficiency, not need to do scheduling
            3. A: simpler implementation
            4. A: more general; can do broadcast
            5. A: more general: can have one condvar for all conditions (see Java) as long as you broadcast…
    c. Invariants
        i. Monitors have a data consistency rule that can must be true when unlocked
            1. Example: doubly-linked list is well formed
            2. Sum of accounts in a bank must equal total money
        ii. Rule: monitor invariant true whenever lock released
            1. When leaving
            2. When waiting (more later)
            3. Relies on programmer to enforce monitor invariant
    d. Why use monitors?
        i. Provides both mutual exclusion and signaling
        ii. Provides abstraction & correctness at programming level
    e. **Question: what is correctness criteria for waiting?**
        i. Mesa: If a thread is waiting, it will get woken up
        ii. Compare to locks: will wake exactly 1 thread, no spurious wakeups
        iii. NOTE: correct implementation of wait() = sleep();
9. Hoare Monitor Comparison
    a. Rule: waiters run **immediately** when signal() is called
        i. Must establish monitor invariant
        ii. Must ensure "condition" is true
    b. Example:
        i. Monitor bounded buffer
            1. Int buf[100]
            2. Int size=0;
            3. Cond_var not_full, not_empty;
            4. Entry put_in_buff(data)
                a. If (size == 100) wait (not_full)
                b. Add_to_buf(data)

c. Signal(not_empty);

5. Entry get_from_buff(data)
    a. If (size == 0) wait (not_empty)
    b. Pull_from_buf(data)
    c. Signal(not_full)

ii. How implement? Sempahores
    1. Monitor semaphore: used on normal entry/exit (no signals)
    2. Urgent semaphore: used when signaling thread
    3. Condvar semaphore: used when waiting on a condition variable
    4. Wait:
        a. Waiters++;
        b. If (urgents)
            i. Signal(urgent_sem)
        c. Else signal(monitor_sem)
        d. Wait(condvar_sem)
        e. Waiter--;
    5. Signal:
        a. If (waiters !0)
            i. Urgents++;
            ii. Signal(condvar_sem);
            iii. Wait(urgent_sem);
            iv. Urgents--;
    6. Exit:
        a. If urgents != 0
            i. Signal(urgent_sem);
        b. Else signal(monitor_sem);

iii. How good is this?
    1. Key problem: signaling requires extra context switches (signaler has to wait to exit monitor until signaled runs and returns)
    2. Key problem: require a 1-1 mapping of condition variables to real “conditions” (things in if-clause before signal)
        a. Waiters don’t check to see if true, so must be guaranteed to be true
        b. Cannot “broadcast” and wake up many waiters and have them figure out which ones can proceed

10. Mesa Monitors
    a. Wait happens in a loop
        i. Always check for condition to become true
        ii. Solves preceding two problems
    b. Is a hint
        i. Means: correct implementation is:
            1. unlock
            2. lock
        ii. Can just release lock to let someone else run and return immediately

            1. Called a "spurious wakeup"
        iii. Simplifies implementation
            1. If might have been woken, always save to return immediately just in case
        iv. Can wake someone waiting on the lock rather than the condition variable
        v. Implementation: on signal, move waiting thread from queue for condvar to queue for lock
    c. Can broadcast
        i. Wake up many threads, let them decide which should execute
            1. E.g. thread waiting for enough memory – it can check if there is enough.
        ii. **WHY IMPOSSIBLE WITH HOARE MONITORS?**
            1. **Hard to guar**antee condition is true when every thread awakes
        iii. Can use "covering condition" – something more relaxed
            1. E.g. x > 0 rather than x > 2

11. Why monitors help?
    a. What does it make easier?
        i. Tend to use locking in the right places; can't access private data without lock
        ii. Tend to release locks appropriately (automatic when exit monitor)
    b. Monitors enforce abstraction, but not a protocol
        i. E.g. can call functions out of order
        ii. Motivates need for Singularity contracts
    c. Can use with Groups of objects (as compared to a single instance of a class)
        i. Monitored Records (can skip)
            1. Basically allow explicitly saying what object you are synchronizing on (e.g. java synchronized(object)
            2. Compiler emits code to acquire/release lock, monitored record says what object to lock on.
            3. Imagine a table of locks, one for each object/address
    d.

12. Extensions:
    a. Time out
        i. can wake up after a period
        ii. Works because calling thread has to check condition; can check timeouts also
    b. Abort
        i. Can wake up a sleeping thread and tell it to abort
        ii. Not delivered to running threads; next time it waits it gets abort exception
        iii. Safe to wake thread – not in the middle of executing
    c. Exceptions
        i. What happens if exception happens in or below monitor?
            1. Cannot automatically return: would not restore monitor invariant

    ii. Choices:
        1. abort thread
        2. Return but leave lock held
        3. Make monitor handle
            a. Consequence: monitors cannot pass along exceptions from below
        4. Mesa choice:
            a. Handler runs with monitor lock held; acts like a call out
            b. Return_with_error() exits the monitor first then throws exception

13. Where not help?
    d. Modifying groups of objects at once
        i. May have problems if you have to manipulate more than one at a time:
            a. Transfer(obj a, obj b, int x)
                i. A.transfer(b, x)
                    1. A.debit(x);
                    2. B.credit(x)
            b. If Transfer() is an entry procedure, Will deadlock if called on (a,b,x) and (b,a,x) simultaneously
            c. If transfer() is not, does not guarantee atomicity – another thread could see a balance of zero for A and B

14. Central problem in conditional synchronization: modularity
    a. Consider a world of objects/modules
    b. Would like a function to be able to safely call into any other function while inside a monitor/entry procedure
        i. Not want to know about implementation
        ii. Not want to know about internal synchronization
    e. What is the problem?
        i. What if it holds a lock?
            1. Only if it calls back into caller (callback) – leads to deadlock
            2. Fortunately, fairly rare in general purpose code, as leads to cyclic dependencies
            3. Can it happen in an OS?
                a. VM and FS both call into each other during memory mapping files
        ii. What if callee module blocks on a condition
            1. Release callee's lock only, not callers. No new threads into calling module
            2.
    f. What is the right thing to do?
        i. Release lock on call out?
            1. No: programmer must know of caller will block
        ii. Prevent calls out?
            1. Too restrictive

            iii. Hold lock?
                1. O.k., but must make sure callee doesn't wait for something blocked by caller
                2. E.g.: all entrees to callee go through caller
                3. E.g. callee calls back to caller monitor
        g. Modern solution (more on Thursday): transactions
            i. Abort caller, rollback any changes, retry when necessary condition holds (see "conditional critical regions" above)
2. Extending monitors to the hardware
    a. Cool feature: no interrupts; instead hardware raises a condition
        i. Move interrupt handler to ready Q

```
WHILE (buffered_packets == 0)
   WAIT (packet_cond);
process_next_packet ();
```

    b. Cool feature: on every cycle, hardware checks if there is a higher-priority process to run, and switches if so (~70 cycles)
        i. Makes sure that high priority interrupt handlers run right away
    c. Naked notify:
        i. Call notify while not holding lock
        ii. Problem with naked notify
            1. Thread can test condition, do a wait() but signal comes in between the test and the wait, so it is never received.
            2. Normally, monitor lock prevents this
        iii. Problem: don't want hardware to take locks, so may signal without acquiring lock
        iv. Solution: wakeup-waiting switch
            1. Provides some history to a condition variable, so it stays signaled
            2. Single bit per process. 0 means WAIT acts as usual, 1 means WAIT turns bit back to 0 but never goes to sleep
            3. Device must set wakeup-waiting bit, then NOTIFY driver
            4. Ensures that notify is sticky; a subsequent wait() will not stop
        v. SHOW EXAMPLE
    d. Comparison to locks + condition variables
        i. QUESTION: What does language integration buy you?
            1. Consider Java notify/notify all
            2. Less likely to forget to hold a lock
            3. Locks are visible to compiler, so they can make optimizations about code while lock is held
            4. Loss of flexibility – may want explicit locks, but locks are tied to procedures
3. Transactional Memory
    a. What do locks give you?
        i. Atomicity: entire critical section is executed as a chunk from perspective of other threads

    ii. Isolation: don't see intermediate states of a thread in a critical section
b. Problems:
    i. Deadlock: acquire locks out of order
    ii. Wrong lock: acquiring correct lock for data (see eraser)
    iii. Lock granularity:
        1. Fine grain – lots of time spent locking/unlocking, likely deadlock
        2. Coarse grain – easy, correct, but low concurrency with many processors
c. Transactional memory: allow programmer to declare regions "atomic"
    i. No associating locks with code/data
        1. Just annotate code that should be executed atomically
    ii. Provides atomicity: executes either all the way to the end or not at all
        1. Either acquire all locks first, so can execute to end without waiting, or speculate and abort if got it wrong
    iii. Provides isolation: internal state not visible
        1. Detect concurrent memory accesses from transactions in other threads
        2. Stall/abort/wait on lock if someone tries to access same data
    iv. Automatically detects conflicts
        1. Value written by one transaction is read/written by another transaction
        2. Prevents serializability: execution as if a global lock held for duration of transaction
        3. Solution is to abort one of the two transactions
    v. How works?
        1. Eager system: tm writes to memory, stores old value somewhere else. On coherence requests from other processors, checks whether access is to something accessed by the local transaction
        2. Lazy system: memory is unchanged, new values buffered elsewhere. Subsequent reads must check elsewhere for data. At commit, broadcast set of locations read/written, all conflict transactions abort.
    vi. Tradeoff:
        1. Memory for time; buffers state in memory for atomicity to solve deadlocks.
    vii. Compared to locks:
        1. Only detects conflicts when two threads access the **same memory locations**
            a. Like a perfectly fine-grained lock; only protects memory actually accessed
        2. No need to select the lock to protect data; always detects concurrent access to same memory locations
    viii. Example:
        1. Transfer(queue x, queue y, obj z) {

```
begin_tx
  x.remove(z);
  y.add(z);
end_tx;
```

2. What happens if called on (x,y) and (y,x)?
    a. System detects a conflict, aborts one of them
3. What if called on (x,y) and (a,b)?
    a. Can execute in parallel (fine grained locking)

ix. Contention: what happens when applications conflict?
1. Contention manager (in hw?) applies a policy to decide which transaction gets to keep executing.
2. Common policies:
    a. Oldest wins: ensures liveness
    b. Committer wins: only detect at commit, long tx gets starved
    c. SizeMatters: tx that has read/written more data wins

d. What does it make easier?
  i. No longer remember which lock protects which data
    1. Only use transactions
  ii. No longer have to create lots of locks
    1. Write coarse grained locks, get benefit of fine-grained locks
    2. Just transactions
  iii. Avoid the cost of acquiring/releasing a lock
    1. Atomic instructions are expensive
  iv. No deadlock between pure transactions
    1. Detected by TM system, resolved automatically by abort
    2. If call from tx 1 into tx2, which calls back into code accessing data from tx1, what happens?
        a. F() {

```
 begin_tx;
   x = 1;
   A();
 end_tx;
}
A() {
 begin_tx;
 G();
 end_tx;
}
G() {
 x = 2;
}
```

        b. In a monitor, this will deadock when recursively acquiring monitor lock

                c. With a transaction, this is just fine
        v. What happens instead of deadlock?
            1. Aborts
        vi. What happens where you might have lock contention?
            1. Repeated aborts; even worse than lock contention

4. TM Implementation
    a. Hardware:
        i. Save registers
        ii. Buffer state accessed by a transaction in cache
        iii. Detect coherence request from another core as a conflict, abort transactions in either thread
        iv. Note: faster than locks (no atomic instructions)
    b. Software
        i. Instrument code to note begin/end of transaction
            1. Save registers
        ii. Note all memory accesses and record
        iii. Compare accesses against concurrent transactions from other threads
            1. On conflict, abort one transaction
        iv. Note: 3-10x slower than normal code
    c. What gets harder?
        i. High contention: rather than queuing, tx all try, get aborted, restart
            1. May have mutual death
            2. May have backoff (Ethernet style) to make progress, causing longer delays
        ii. Dealing with non-transactional code
            1. System calls
            2. I/O
                a.
        iii. Synchronization
            1. How do you deal with waiting, signaling?
            2. A: no answer – doesn't help
        iv. Modularity/correctness
            1. Not much better than locks
            2. Can enforce in language to be lexically scoped, to ensure you end transaction
            3. Take away points
    d. Synchronization is hard
    e. Important issues are:
        i. Granularity
        ii. Priority
        iii. Composition
        iv. Synchronization
    f. There is no free bullet