EECS 262a
Advanced Topics in Computer Systems
Lecture 5

Mesa/Transactions
February 3rd, 2016

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

Today's Papers

- Experience with Processes and Monitors in Mesa Butler Lampson and David Redell, Appears in *Communications of the ACM* 23, 2 (Feb. 1980), pp 105-117.
- Principles of Transaction-Oriented Database Recovery, Theo Haerder and Andreas Reuter. Appears in *Journal of the ACM Computing Surveys* (CSUR), Vol 15, No 4 (Dec. 1983), pp287-317

- Thoughts?

Mesa Motivation

- Putting theory to practice – building Pilot OS

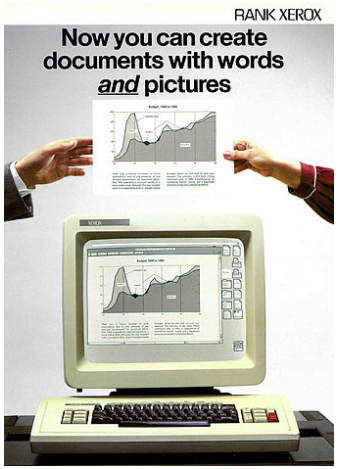
- Focus of this paper: lightweight processes (threads in today's terminology) and how they synchronize with each other

Mesa History

- 2nd system Xerox Star – followed the Alto

- Planned to build a large system using many programmers
  - Some thoughts about commercializing

- Advent of things like server machines and networking introduced applications that are heavy users of concurrency



Mesa History (cont'd)

- Chose to build a single address space system:
  - Single user system, so protection not an issue
  - Safety was to come from the language
  - Wanted global resource sharing
- Large system, many programmers, many applications:
  - Module-based programming with information hiding
- Clean sheet design:
  - Can integrate the hardware, the runtime software, and the language with each other
- Java language considers Mesa to be a predecessor

Programming Models for IPC

- Two Inter-Process Communication models:
  - Shared memory (monitors) vs.
  - Message passing
- Needham & Lauer claimed the two models are duals of each other
- Mesa developers chose shared memory model because they thought they could more naturally fit it into Mesa as a language construct

How to Synchronize Processes?

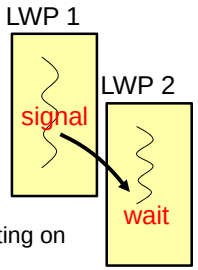
- Non-preemptive scheduler: results in very delicate systems – Why?
  - Have to know whether or not a yield might be called for *every* procedure you call – this violates information hiding
  - Prohibits multiprocessor systems
  - Need a separate preemptive mechanism for I/O anyway
  - Can't do multiprogramming across page faults
- Simple locking (*e.g.*, semaphores):
  - Too little structuring discipline, *e.g.*, no guarantee that locks will be released on every code path
  - Wanted something that could be integrated into a Mesa language construct
- Chose preemptive scheduling of lightweight processes and monitors

Lightweight Processes (LWPs)

- Easy forking and synchronization
- Shared address space
- Fast performance for creation, switching, and synchronization; low storage overhead
- Today we call LWPs, "threads"

Recap: Synchronization Goals

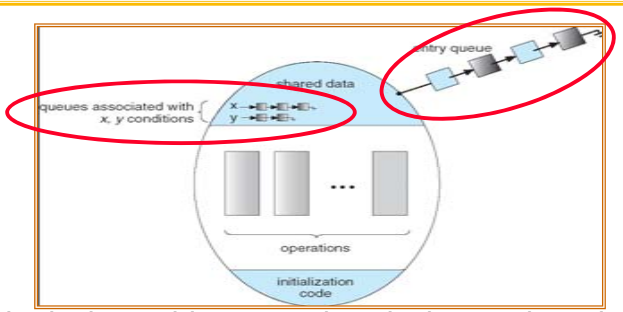
- Mutual exclusion:
  - Arbitrate access to critical section (e.g., shared data)
  - Only a single LWP in critical section at a given time
    - » If one LWP in critical section → all other LWPs that want to enter the critical section need to wait
- Scheduling constraint:
  - A LWP waiting for an event to happen in another thread
- Wait instruction:
  - Don't want busy-waiting, so sleep()
  - Waiting LWPs are woken up when the condition they are waiting on becomes FALSE



Recap: Synchronization Primitives

- Locks: Implement mutual exclusion
  - `Lock.Acquire()`: acquire lock before entering critical section; wait if lock not free
  - `Lock.Release()`: release lock after leaving critical section; wake up threads waiting for lock
- Semaphores: Like integers with restricted interface
  - `P()`: Wait if zero; decrement when becomes non-zero
  - `V()`: Increment and wake a sleeping task (if exists)
  - Use a semaphore for each scheduling constraint and mutex
- Monitors: A lock plus one or more condition variables
  - Condition variable: a queue of LWPs waiting inside critical section for an event to happen
  - Use condition variables to implement sched. constraints
  - Three Operations: `Wait()`, `Signal()`, and `Broadcast()`

Recap: Monitor with Condition Variables



- Lock: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- Condition Variable: a queue of LWPs waiting for something *inside* a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep

Recap: Monitors

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting LWPs can proceed
- Basic structure of monitor-based program:

```
lock.Acquire()
while (need to wait) {
   condvar.wait(&lock);
}
lock.Release()

do something so no need to wait

lock.Acquire()

condvar.signal();

lock.Release()
```

Check and/or update state variables

Wait if necessary

(release lock when waiting)

Check and/or update state variables

Mesa Monitors

- Monitor lock (for synchronization)
- Tied to module structure of the language – makes it clear what's being monitored
- Language automatically acquires and releases the lock
- Tied to a particular invariant, which helps users think about the program
  - Invariant holds on entry and must be maintained before exit or wait
- Condition variable (for scheduling) – when to wait
- Dangling references problem similar to pointers
  - There are also language-based solutions that would prohibit these kinds of errors, such as do-across, which is just a parallel control structure
  - Do-across eliminates dangling processes because the syntax defines the point of the fork and the join

Design Choices and Implementation Issues

- 3 types of procedures in a monitor module:
  - Entry (acquires and releases lock)
  - Internal (no locking done): can't be called from outside the module.
  - External (no locking done): externally callable. Why is this useful?
    - » Allows grouping of related things into a module
    - » Allows doing some of the work outside the monitor lock
    - » Allows controlled release and reacquisition of monitor lock
- Choices for notify semantics:
  - (Hoare monitors) Immediately cede CPU and lock to waking process
    - » Causes many context switches but why would this approach be desirable?
      (Waiting process knows the condition it was waiting on is guaranteed to hold)
    - » Also, doesn't work in the presence of priorities
  - (Mesa monitors) Notifier keeps lock, wakes process with no guarantees => waking process must recheck its condition

Mesa Monitor: Why "while()"?

- **Why do we use "while()" instead of "if() with Mesa monitors?**
  - **Example illustrating what happens if we use "if()", e.g.,**

```
if (queue.isEmpty()) {
  dataready.wait(&lock); // If nothing, sleep
}
```

- **Synchronized (infinite) queue example**

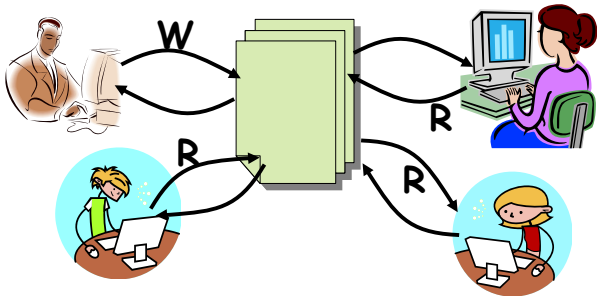
```
AddToQueue(item) {
  lock.Acquire();
  queue.enqueue(item);
  dataready.signal();
  lock.Release();
}
```

```
RemoveFromQueue() {
  lock.Acquire();
  if (queue.isEmpty()) {
    dataready.wait(&lock);
  }
  item = queue.dequeue();
  lock.Release();
  return(item);
}
```

Hoare

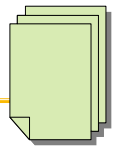
Mesa: Replace "while" with "if"

Readers/Writers Problem



- **Motivation: Consider a shared database**
  - **Two classes of users:**
    - » **Readers – never modify database**
    - » **Writers – read and modify database**
  - **Is using a single lock on the whole database sufficient?**
    - » **Like to have many readers at the same time**
    - » **Only one writer at a time**

Basic Readers/Writers Solution



- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - Reader()
    ```
    Wait until no writers
    Access data base
    Check out – wake up a waiting writer
    ```
  - Writer()
    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
    ```
  - State variables (Protected by a lock called "lock"):
    - » int AR: Number of active readers; initially = 0
    - » int WR: Number of waiting readers; initially = 0
    - » int AW: Number of active writers; initially = 0
    - » int WW: Number of waiting writers; initially = 0
    - » Condition okToRead = NIL
    - » Conditioin okToWrite = NIL

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) {  // Is it safe to read?
    WR++;                  // No. Writers exist
    okToRead.wait(&lock);  // Sleep on cond var
    WR--;                  // No longer waiting
  }
  AR++;                    // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                    // No longer active
  if (AR == 0 && WW > 0)   // No other active readers
    okToWrite.signal();    // Wake up one writer
  lock.Release();
}
```


Code for a Writer

```
Writer() {
  // First check self into system
  lock.Acquire();
  while ((AW + AR) > 0) {  // Is it safe to write?
    WW++;                  // No. Active users exist
    okToWrite.wait(&lock);  // Sleep on cond var
    WW--;                  // No longer waiting
  }
  AW++;                    // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;                    // No longer active
  if (WW > 0){             // Give priority to writers
    okToWrite.signal();    // Wake up one writer
  } else if (WR > 0) {     // Otherwise, wake reader
    okToRead.broadcast();  // Wake all readers
  }
  lock.Release();
}
```


Other Kinds of Notifications

- Timeouts
- Broadcasts
- Aborts
- Deadlocks:
  - Wait only releases the lock of the current monitor, not any nested calling monitors
  - General problem with modular systems and synchronization:
    - » Synchronization requires *global* knowledge about locks, which violates information hiding paradigm of modular programming

Four Requirements for Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - » T_1 is waiting for a resource that is held by T_2
    - » T_2 is waiting for a resource that is held by T_3
    - » …
    - » T_n is waiting for a resource that is held by T_1

Deadlock

- Why is monitor deadlock less onerous than the yield problem for non-preemptive schedulers?
  - Want to generally insert as many yields as possible to provide increased concurrency; only use locks when you want to synchronize
  - Yield bugs are difficult to find (symptoms may appear far after the bogus yield)
- Basic deadlock rule: no recursion, direct or mutual
  - Alternatives? Impose ordering on acquisition
  - "It is unreasonable to blame the tool when poorly chosen constraints lead to deadlock"
- Lock granularity for concurrent access to objects
  - Introduced monitored records so that the same monitor code could handle multiple instances of something in parallel

Interrupts

- Interrupt handler can't afford to wait to acquire a monitor lock
- Introduced naked notifies: notifies done without holding the monitor lock
- Had to worry about a timing race:
  - The notify could occur between a monitor's condition check and its call on Wait
  - Added a wakeup-waiting flag to condition variables
- What happens with active messages that need to acquire a lock? (move handler to its own thread)

Priority Inversion

- High-priority processes may block on lower-priority processes
- A solution:
  - Temporarily increase the priority of the holder of the monitor to that of the highest priority blocked process
  - Somewhat tricky – what happens when that high-priority process finishes with the monitor?
    - » You have to know the priority of the next highest one – keep them sorted or scan the list on exit

BREAK

The Mars Pathfinder Mission

- Widely proclaimed as “flawless” in the early days after its July 4th, 1997 landing on the Martian surface
- Successes included:
  - Its unconventional “landing” – bouncing onto the Martian surface surrounded by airbags
  - Deploying the Sojourner rover
  - Gathering and transmitting voluminous data back to Earth, including the panoramic pictures that were such a hit on the Web
  - 20Mhz PowerPC processor and 128MB of DRAM
- A few days later, just after Pathfinder started gathering meteorological data…
  - The spacecraft began experiencing total system resets, each resulting in losses of data
  - The press reported these failures in terms such as “software glitches” and “the computer was trying to do too many things at once.”

Three Mars Pathfinder Tasks

- Bus management task
  - Ran frequently with high priority to move certain kinds of data in and out of the VME information bus
  - Access to bus synchronized with mutual exclusion locks (mutexes)
- Meteorological data gathering task (ASI/MET)
  - Infrequent, low priority thread that used info bus to publish its data
  - When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex
  - If an interrupt caused bus thread to be scheduled while this mutex was held, and if the bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteo. thread released the mutex before it could continue
- Communications task that ran with medium priority
- Most of the time this combination worked fine…

Priority Inversion

- Very infrequently an interrupt would occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread
  - In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running
- After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset
- This scenario is a classic case of priority inversion

Really Remote Debugging!

- Pathfinder used VxWorks
  - VxWorks can be run in a tracing mode
  - Records a total trace of all interesting system events, including context switches, uses of synchronization objects, and interrupts
- Reproducing bug locally
  - After the failure, JPL engineers spent hours and hours running the system on the exact spacecraft replica in their lab with tracing turned on, attempting to replicate the precise conditions under which they believed that the reset occurred
  - Early in the morning, after all but one engineer had gone home, the engineer finally reproduced a system reset on the replica
- Analysis of the trace revealed the priority inversion

Remote Bug Fixing

- When created, a VxWorks mutex object accepts a boolean parameter that indicates whether priority inheritance should be performed by the mutex
  - The mutex in question had been initialized with the parameter off; had it been on, the low-priority meteorological thread would have inherited the priority of the high-priority data bus thread blocked on it while it held the mutex, causing it be scheduled with higher priority than the medium-priority communications task, thus preventing the priority inversion
  - Once diagnosed, it was clear to the JPL engineers that using priority inheritance would prevent the resets they were seeing
  - By coding convention, the initialization parameter for the mutex in question (and those for two others which could have caused the same problem) were stored in global variables
- How to fix remotely?
  - VxWorks debugging mode!

VxWorks Debugging Mode

- VxWorks debugging mode
  - Has a C language interpreter intended to allow developers to type in C expressions and functions to be executed on the fly during system debugging
- JPL engineers fortuitously decided to launch the spacecraft with this feature still enabled
  - Addresses of initialization parameters for mutexes stored in symbol tables included in the launch software, and available to the C interpreter
- A short C program was uploaded to the spacecraft, which when interpreted, changed the values of these variables from FALSE to TRUE
- No more system resets occurred!
  - One month planned mission lasted for three months instead!

Temporal Logic Assertions for the Detection of Priority Inversion

- Temporal logic – system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time
  - Can express requirements – e.g., ***whenever* a request is made, access to a resource is *eventually* granted, but it is *never* granted to two requestors simultaneously**
- TLA assertions were written as comments in the Pathfinder code
- *Temporal-Rover* software generated code that announces success and/or failure of any assertion during testing
  - **T-R compared the actual program's behavior with the formal specification**
  - T-R captured **all possible "golden" executions of the program**
- Interestingly enough, the JPL engineers actually created a priority inversion situation during testing
  - 1-2 system resets during months of pre-flight testing
  - Not reproducible or explainable, so "was probably caused by a hardware glitch"

TLA (cont'd)

- TLA captures time and order, so let:
  - `HIGHPriorityTaskBlocked()` represent a situation where the info bus thread is blocked by the low priority meteorological data gathering task
  - `HIGHPriorityTaskInMutex()` represent a situation where the information bus thread is in Mutex
  - `LOWPriorityTaskInMutex()` represent a situation where the meteorological thread is in Mutex
  - `MEDPriorityTaskRunning()` represent a situation where the communications task is running
- Assertions:
  - Not Eventually ({`HIGHPriorityTaskBlocked()`} AND {`MEDPriorityTaskRunning()`})
  - This formula specifies that never should it be that `HIGHPriorityTaskBlocked()` And `MEDPriorityTaskRunning()`.
  - Always({`LOWPriorityTaskInMutex()`} Implies Not {`MEDPriorityTaskRunning()`} Until {`HIGHPriorityTaskInMutex()`})
  - This formula specifies that always, if `LOWPriorityTaskInMutex()` then `MEDPriorityTaskRunning()` does not occur until a later time when `HIGHPriorityTaskInMutex()`.

TLA (cont'd)

- Engineers did not manage to analyze their recorded data well enough to conclude that priority inversion is indeed a bug in their system
  - In other words, their test runs were sufficient, but their analysis tools were not
- Part of it was the engineers' focus – extremely focused on ensuring the quality and flawless operation of the landing software
  - Should it have failed, the mission would have been lost
  - Also, first "low-cost" NASA mission
- Entirely understandable for the engineers to discount occasional glitches in the less-critical land-mission SW
  - A spacecraft reset was a viable recovery strategy at that phase of the mission

Exceptions

- Must restore monitor invariant as you unwind the stack
  - But, requires explicit UNWIND handlers (RETURN WITH ERROR[args]), otherwise lock is not released
- Failure to handle exceptions results in debugger invocation
  - "not much comfort, however, when a system is in operational use"
- What does Java do?
  - Release lock, no UNWIND primitive

Hints vs. Guarantees

- Notify is only a hint
  - Don't have to wake up the right process
  - Don't have to change the notifier if we slightly change the wait condition (the two are decoupled)
  - Easier to implement, because it's always OK to wake up too many processes. If we get lost, we could even wake up everybody (broadcast)
    - » Can we use broadcast everywhere there is a notify? Yes
    - » Can we use notify everywhere there is a broadcast? No, might not have satisfied OK to proceed for A, have satisfied it for B
- Enables timeouts and aborts
- General Principle: use hints for performance that have little or better yet no effect on the correctness
  - Many commercial systems use hints for fault tolerance: if the hint is wrong, things timeout and use a backup strategy
    - » Performance hit for incorrect hint, but no errors

Performance

- Assumes simple machine architecture
  - Single execution, non-pipelined – what about multi-processors?
- Context switch is very fast: 2 procedure calls (60 ticks)
- Ended up not mattering much for performance:
  - Ran only on uniprocessor systems
  - Concurrency mostly used for clean structuring purposes
- Procedure calls are slow: 30 instructions (RISC proc. calls are 10x faster); Why?
  - Due to heap allocated procedure frames. Why did they do this?
    - » Didn't want to worry about colliding process stacks
  - Mental model was "any procedure call might be a fork": transfer was basic control transfer primitive
- Process creation: ~ 1100 instructions
  - Good enough most of the time
  - Fast-fork package implemented later that keeps around a pool or "available" processes

3 Key Features about the Paper

- Describes the experiences designers had with designing, building and using a large system that aggressively relies on lightweight processes and monitor facilities for all its software concurrency needs
- Describes various subtle issues of implementing a threads-with-monitors design in real life for a large system
- Discusses the performance and overheads of various primitives and presents three representative applications, but doesn't give a big picture of how important various decisions and features turned out to be

Some Flaws

- Gloss over how hard it is to program with locks and exceptions sometimes – not clear if there are better ways
- Performance discussion doesn't give the big picture
  - Tries to be machine-independent (ticks), but assumes particular model
- A takeaway lesson: The lightweight threads-with-monitors programming paradigm can be used to successfully build large systems, but there are subtle points that have to be correct in the design and implementation in order to do so

Is this a good paper?

- What were the authors' goals?
- What about the evaluation / metrics?
- Did they convince you that this was a good system /approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Transactions (Brief intro before ARES)

- Second paper really a summary paper
  - Ideas behind transactions were stabilizing at this time (1983)
  - Gray, Berstein, Goodman, Codd, …
- Point of reading this paper: Getting transaction concept into your brain
  - Transactions are *Atomic* actions that are all-or-nothing
  - They either complete entirely or they do not even start

```
Begin_Transaction()
Do a bunch of things – read and write data
End_Transaction()
```

  - Until End_Transaction() completes, transaction may be aborted and effects will be nullified
  - After End_Transaction() completes, transaction will be *durable* and results should survive even system crashes
  - Transaction said to "Commit" after End_Transaction()

ACID Semantics

- Full Transactional support includes:
  - Atomicity: It must be an all-or-nothing commitment
  - Consistency: After commit, a transaction will preserve the consistency of the data base
    - » Example: for banking app, net amount of money constant, even after moving money from account to account
  - Isolation: Events within a transaction hidden from other transactions
    - » Writes from uncommitted transactions invisible to other transactions
  - Durability: Once a transaction has been completed and committed its results, results will survive even system crashes
- Much of focus is around techniques for achieving these properties
  - Use of Log to permit transactions to abort/be durable

Recovery actions: The Log

- Log entries:
  - REDO: (forward) enough information to perform update to the data
  - UNDO: (backward) enough information to move backward\
  - BEGIN: Start of transaction
  - COMMIT: End of transaction
  - ABORT: Transaction was aborted
- Once transaction committed in log, transaction durable
  - Updates may not be reflected in permanent state
  - If crash happens before COMMIT is placed into log, should be as if nothing ever happened: may need to undo state
- Checkpoints:
  - Flush log out to well defined point/discard old log entries
  - Good point to recover from
- Ways of updating permanent state
  - Update in place: use log to undo state if necessary
  - Shadow pages: keep old state around and update to new pages
  - Other ideas: keep different views of DB, "Old and new" copies, etc

Other things in paper

- Lots of discussion of implementation techniques
  - ATOMIC, STEAL, FORCE, …
- ARIES paper for next time will show "BEST IN CLASS" combination of features

Is this a good paper?

- What were the authors' goals?
- What about the evaluation / metrics?
- Did they convince you that this was a good system /approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?