Frequently asked questions from the previous class survey

- Producer-consumer with bounded buffer
  - Should the production and consumption rates be a perfect match?
  - Can the producer add more than 1 item at a time?

Topics covered in today's lecture

- Synchronization examples
- Atomic transactions

Synchronization in Solaris

- Condition variables
- Semaphores
- Adaptive mutexes
- Reader-writer locks
- Turnstiles

Synchronization in Solaris: Adaptive mutex

- Starts as a standard semaphore implemented as spinlock
- On SMP systems if data is locked and in use?
  - If lock held by thread on another CPU
    - Spin waiting for lock to be available
  - If thread holding the lock is not in the run state
    - Block until awakened by release of the lock
Adaptive mutex:
On a single processor system

- Only one thread can run at a time
- So thread sleeps (instead of spinning) when a lock is encountered

Adaptive mutex is used only for short code segments

- Less than a few hundred instructions
  - Spinlocks inefficient for code segments larger than that
  - Cheaper to put a thread to sleep and awaken it
  - Busy waiting in the spinlock is expensive
- Longer code segments?
  - Condition variables and semaphores used

Reader-writer locks

- Used to protect data accessed frequently
  - Usually accessed in a read-only manner
- Multiple threads can read data concurrently
  - Unlike semaphores that serialize access to the data
- Relatively expensive to implement
  - Used only on long sections of code

Solaris: Turnstiles

- Queue structure containing threads blocked on a lock
- Used to order threads waiting to acquire adaptive mutex or reader-writer lock
- Each kernel thread has its own turnstile
  - As opposed to every synchronized object
  - Thread can be blocked only on one object at a time

Turnstiles are organized according to the priority inheritance protocol

- If the thread is holding a lock on which a higher priority thread is blocked?
  - Will temporarily inherit priority of higher priority thread
  - Revert back to original priority after releasing the lock
Linux: Prior to 2.6, Linux was a nonpreemptive kernel

- Provides spinlocks and semaphores

<table>
<thead>
<tr>
<th>Single processor</th>
<th>Multiple processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable kernel preemption</td>
<td>Acquire spinlock</td>
</tr>
<tr>
<td>Enable kernel preemption</td>
<td>Release spinlock</td>
</tr>
</tbody>
</table>

17 December 2003 - Linux 2.6.0 was released (5,929,913 lines of code)
4 January 2011 - Linux 2.6.37 was released (13,996,612 lines of code)

Kernel is not preemptible if a kernel-mode task is holding a lock

- Each task has a thread-info structure
  - Counter preempt_count indicates number of locks being held by task
  - preempt_count incremented when lock acquired
  - Decremented when lock released
  - If preempt_count > 0, not safe to preempt
  - OK otherwise; if no preempt_disable() calls pending

Linux: Other mechanisms

- Atomic integers atomic_t
  - All math operations using atomic integers are performed without interruption
  - E.g. Set, add, subtract, increment, decrement

- Mutex locks
  - mutex_lock(): Prior to entering critical section
  - mutex_unlock(): After exiting critical section
  - If lock is unavailable, task calling mutex_lock() is put to sleep
  - Awakened when another task calls mutex_unlock()

Atomic transactions

- Mutual exclusion of critical sections ensures their atomic execution
  - As one uninterruptible unit

- Also important to ensure, that critical section forms a single logical unit of work
  - Either work is performed in its entirety or not at all
  - E.g. transfer of funds
    - Credit one account and debit the other

Transaction

- Collection of operations performing a single logical function
- Preservation of atomicity
  - Despite the possibility of failures

ATOMIC TRANSACTIONS

Mutual exclusion of critical sections ensures their atomic execution and prevents race conditions. Atomicity is the guarantee that a transaction will either execute completely or not at all, ensuring the consistency of operations even in the presence of failures.
A transaction is a program unit that accesses/updates data items on disk
- Simply a sequence of read and write operations
  - Terminated by commit or abort
- **Commit**: Successful transaction termination
- **Abort**: Unsuccessful due to
  - Logical error or system failure

Transaction rollbacks
- An aborted transaction may have **modified** data
- State of accessed data must be **restored**
  - To what it was before transaction started executing

Log-based recovery to ensure atomicity: Rely on stable storage
- Record info describing all modifications made by transaction to various accessed data.
- Each log record describes a **single** write
  - Transaction name
  - Data item name
  - Old value
  - New value
- Other log records exist to record significant events
  - Start of transaction, commit, abort etc

Actual update cannot take place prior to the logging
- Prior to write(X) operation
  - Log records for X should be written to stable storage
- Two physical writes for every logical write
  - More storage needed
- Functionality worth the price:
  - Data that is extremely important
  - For fast failure recovery

Populating entries in the log
- Before transaction \( T_1 \) starts execution
  - Record \(<T_1 \text{ starts}>\) written to the log
- Any write by \( T_1 \) is **preceded** by writing to the log
- When \( T_1 \) commits
  - Record \(<T_1 \text{ commits}>\) written to log

Storage system hierarchy based on speed, cost, size and volatility
- Registers
- Cache
- Main Memory
- Electronic Disk
- Magnetic Disk
- Optical Disk
- Magnetic Tapes
- Cost/bit increases
- Access times increase
- Volatile
The system can handle any failure without loss of information: Log

- undo($T_i$)
  - Restores value of all data updated by $T_i$ to old values
- redo($T_i$)
  - Sets value of all data updated by $T_i$ to new values
- undo($T_i$) and redo($T_i$)
  - Are idempotent
  - Multiple executions have the same result as 1 execution

If system failure occurs restore state by consulting the log

- Determine which transactions need to be undone; and which need to be redone
- $T_i$ is undone if log
  - Contains $<T_i$ starts> but no $<T_i$ commits> record
- $T_i$ is redone if log
  - Contains both $<T_i$ starts> and $<T_i$ commits>

Rationale for checkpointing

- When failure occurs we consult the log for undoing or redoing
- But if done naively, we need to search entire log!
  - Time consuming
  - Recovery takes longer
    - Though no harm done by redoing (idempotency)

In addition to write-ahead logging, periodically perform checkpoints

- Output the following to stable storage
  - All log records residing in main memory
  - All modified data residing in main memory
  - A log record $<\text{checkpoint}\>$
- The $<\text{checkpoint}\>$ allows a system to streamline recovery procedure

Implications of the checkpoint record

- $T_i$ committed prior to checkpoint
  - $<T_i$ commits$>$ appears before $<\text{checkpoint}\>$
  - Modifications made by $T_i$ must have been written to stable storage
    - Prior to the checkpoint or
    - As part of the checkpoint
- At recovery no need to redo such a transaction
Refining the recovery algorithm

- Search the log backward for first checkpoint record.
- Find transactions $T_i$ following the last checkpoint
- Redo and undo operations applied only to these transactions

Looking at the log to determine which one to redo and which one to undo

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>starts &gt;</td>
</tr>
<tr>
<td>$T_1$</td>
<td>... write record</td>
</tr>
<tr>
<td>$T_1$</td>
<td>aborts</td>
</tr>
<tr>
<td>$T_2$</td>
<td>starts &gt;</td>
</tr>
<tr>
<td>$T_2$</td>
<td>... write record</td>
</tr>
</tbody>
</table>
| $T_2$       | commits >
|         | <checkpoint> |
| $T_3$       | starts > |
| $T_3$       | ... write record |
| $T_3$       | commits >
| $T_4$       | starts > |
| $T_4$       | ... write record |
| $T_4$       | commits >
| $T_5$       | starts > |
| $T_5$       | ... write record |

T4 will be redone
T5 will be undone

Concurrent atomic transactions

- Since each transaction is atomic
  - Executed serially in some arbitrary order
  - Serializability
  - Maintained by executing each transaction within a critical section
  - Too restrictive
- Allow transactions to overlap while maintaining serializability
- Concurrency control algorithms

Serializability

- Serial schedule: Each transaction executes atomically
  - $n!$ schedules for $n$ transactions

Non-serial schedule: Allow two transactions to overlap

- Does not imply incorrect execution
  - Define the notion of conflicting operations
- $O_1$ and $O_2$ conflict if they access same data item
  - AND at least one of them is a write operation
- If $O_1$ and $O_2$ do not conflict, we can swap their order
  - To create a new schedule
Concurrent serializable schedule

<table>
<thead>
<tr>
<th>Serial Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
</tr>
<tr>
<td>T1</td>
</tr>
</tbody>
</table>

Conflict serializability

- If schedule $S$ can be transformed into a serial schedule $S'$
  - By a series of swaps of non-conflicting operations

The contents of this slide-set are based on the following references