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 binary 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

---

Solaris: Turnstiles

- Turnstile for the first thread to block on synchronized object
  - Becomes turnstile for the object itself
  - Subsequent threads blocking on lock are added to this turnstile
- When this first thread releases its lock?
  - It **gains a new turnstile** from the list of free turnstiles maintained by kernel
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)
- Version: 4.10.1 [stable version] (~18,000,000 lines of code)
- Version 6.1 Feb 2023 (~35,550,000 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

Storage system hierarchy based on speed, cost, size and volatility

- **Volatile**
  - Registers
  - Cache
  - Main Memory
  - Electronic Disk
  - Magnetic Disk
  - Optical Disk
  - Magnetic Tapes

- **Cost/bit increases**
- **Access times increase**
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 $T_i$ starts execution
  - Record $<T_i \text{ starts}>$ written to the log

- Any write by $T_i$ is **preceded** by writing to the log

- When $T_i$ commits
  - Record $<T_i \text{ commits}>$ written to log

The system can handle any failure without loss of information: Log

- $\text{undo}(T_i)$
  - **Restores** value of all data updated by $T_i$ to old values

- $\text{redo}(T_i)$
  - Sets value of all data updated by $T_i$ to new values

- $\text{undo}(T_i)$ and $\text{redo}(T_i)$
  - 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 \( <T_i \text{ starts}> \) but no \( <T_i \text{ commits}> \) record.

- \( T_i \) is redone if log contains both \( <T_i \text{ starts}> \) and \( <T_i \text{ commits}> \).
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 `<checkpoint>`

- The `<checkpoint>` allows a system to streamline recovery procedure
Implications of the checkpoint record

- $T_i$ committed prior to checkpoint
  - $<T_i\text{ commits}>$ 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

- <T1 starts>
- <T1 ... write record>
- <T1 aborts>
- <T2 starts>
- <T2 ... write record>
- <T2 commits>
- <checkpoint>
- <T3 starts>
- <T3 ... write record>
- ....
- <checkpoint>
- <T4 starts>
- <T4 ... write record>
- <T4 commits>
- <T5 starts>
- <T5 ... write record>

T4 will be redone
T5 will be undone

Transactions?
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

\[
\begin{align*}
T_0 & \quad \text{read}(A) \\
     & \quad \text{write}(A) \\
     & \quad \text{read}(B) \\
     & \quad \text{write}(B) \\
T_1 & \quad \text{read}(A) \\
     & \quad \text{write}(A) \\
     & \quad \text{read}(B) \\
     & \quad \text{write}(B)
\end{align*}
\]
Non-serial schedule:
Allow two transactions to overlap

- Does not imply incorrect execution
  - Define the notion of conflicting operations

- $O_i$ and $O_j$ conflict if they access same data item
  - AND at least one of them is a write operation

- If $O_i$ and $O_j$ do not conflict; we can swap their order
  - To create a new schedule

Concurrent serializable schedule

Serial Schedule

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</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
Locking protocol governs how locks are acquired and released

- There are different modes in which data can be locked
  - A transaction acquires a lock on a data item in different modes
- Shared mode locks
  - $T_i$ can read, but not write, data item Q
- Exclusive mode locks
  - $T_i$ can read and write data item Q

Transactions must request locks on data items in the right mode

- To access data item Q; $T_i$ must first lock it
  - Wait if Q is locked in the exclusive mode
  - If $T_i$ requests a shared-lock on Q
    - Obtain lock if Q is not locked in the exclusive mode
- $T_i$ must hold lock on data item as long as it accesses it
Two-phase locking protocol: Locks and unlocks take place in two phases

- Transaction's **growing** phase:
  - Obtain locks
  - **Cannot release** any lock

- Transaction's **shrinking** phase
  - Can release locks
  - **Cannot obtain** any new locks

Two-phase locking protocol: Conflict serializability

- Conflicts occur when 2 transactions access same data item; and 1 of them is a write

- A transaction acquires locks serially; **without** releasing them during the acquire phase
  - Other transactions **must wait** for first transaction to start releasing locks

- Deadlocks may occur
Order of conflicting transactions

- Two-phase locking
  - Determined at execution time

- How about selecting this order in advance?
  - Timestamp based protocols

Timestamp based protocols

- For each $T_i$ there is a fixed timestamp
  - Denoted $TS(T_i)$
  - Assigned before $T_i$ starts execution

- For a later $T_j$; $TS(T_i) < TS(T_j)$

- Schedule must be equivalent to schedule in which $T_i$ appears before $T_j$. 
Timestamp based locking

- Protocol ensures there will be no deadlock
  - No transaction ever waits!

- Conflict serializability
  - Conflicting operations are processed in timestamp order

Each data item Q has two values

- W-timestamp(Q)
  - Largest timestamp of any transaction that successfully executed write()

- R-timestamp(Q)
  - Largest timestamp of any transaction that successfully executed read()
Transaction issues a \texttt{read}(Q)

- If $TS(T_i) < W$-timestamp(Q)
  - Needs value that was already \textit{overwritten}
  - The \texttt{read} is rejected and $T_i$ is rolled back

- $TS(T_i) \geq W$-timestamp(Q)
  - Operation is executed
  - $R$-timestamp(Q) = $\max(TS(T_i), R$-timestamp(Q))

The key idea here is that when a transaction executes none of the data items must be from the future.

Transaction issues a \texttt{write}(Q)

- If $TS(T_i) < R$-timestamp(Q)
  - Value of Q produced by $T_i$ needed \textit{previously}
    - $T_i$ assumed that this value would never be produced
  - The \texttt{write} is rejected and $T_i$ is rolled back

- If $TS(T_i) < W$-timestamp(Q)
  - Trying to write an \texttt{obsolete} value of Q
  - The \texttt{write} is rejected and $T_i$ is rolled back
What happens when a transaction is rolled back?

- Transactions $T_1$ is assigned a new timestamp
- Restart

Schedule using the timestamp protocol:

$T_2$
read(B)
read(A)

$T_3$
read(B)
write(B)
read(A)
write(A)

Timestamps are assigned to transactions before the start of the first instruction $TS(T_2) < TS(T_3)$
The contents of this slide-set are based on the following references