OS Concurrency

Concurrency Primitives Recap

The operating system manages system resources and makes them available to multiple tasks (threads) that execute concurrently. Uncontrolled scheduling of threads (i.e., *thread interleaving*) can introduce concurrency problems such as

- **Race Conditions**. A race condition occurs when the timing or ordering of events affects the correctness of a program. Usually, external timing or ordering non-determinism is required to produce a race condition. Examples include context switches, OS signals, memory operations on multiprocessors, or hardware interrupts. Thread interleaving leads to **non-deterministic behavior**, where different executions of a program yield different results.

```
status_t transfer(int amount, atomic<int>& account_from, atomic<int>& account_to) {
    if (account_from.load() < amount) return NOPE;
    account_to.fetch_add(amount);
    account_from.fetch_add(-amount);
    return YEP;
}
```

- **Data Races**. A data race occurs when multiple threads access a shared variable without synchronization, and at least one of the accesses is a write.

```
int counter = 0;

void inc() {
    counter++; // non-atomic read-modify-write
}
```

Both race conditions and data races result from a *lack of atomicity* and can be resolved using *mutual exclusion*.

Mutual Exclusion, Locks and Atomic Instructions

Hardware provides **atomic instructions** to support higher-level synchronization primitives:

- `int xchg(int *ptr, int val)`: Atomic exchange;
- `int cas(int *ptr, int expected, int val)`: Atomic compare-and-swap;
- `int faa(int *ptr, int inc)`: Atomic fetch-and-add.

Mutual exclusion can be implemented using **locks**, which ensure that only one thread can hold the lock and enter the critical section (CS). After exiting the CS, the thread releases the lock.

A **spin lock** is implemented by spinning in a loop until a flag is set to `true` (acquiring the lock) and resetting it to `false` upon exiting the CS. Spinning (i.e., *busy waiting*) does not execute useful instructions and thrashes the cache line, wasting CPU cycles. However, spin locks are suitable for small critical sections because they avoid costly context switches. To manage the effects of the spin lock, alternative approaches involve yielding the processor after unsuccessful acquiring of the lock, or using a mutex. A **mutex** will cause a thread to block, and be woken up by another thread that currently holds a mutex.

An **RW lock** allows multiple readers or a single writer to access a shared resource at a time. It uses expensive atomic operations to track the number of readers, incurring additional overhead for read operations.

Bugs in Concurrent Programming

1. **Atomicity Violation Bugs:** Occur when a sequence of operations intended to be atomic is interrupted, leading to inconsistent states. Solution: Use locks to protect shared resources.
2. **Order Violation Bugs:** Occur when the expected sequence of operations is not followed. Solution: Use condition variables to enforce the desired order of execution.
3. **Deadlocks:** Arise when multiple threads wait on each other to release resources, resulting in no progress. Deadlocks occur if these four conditions are satisfied:
   i. **Mutual Exclusion:** Only one thread can hold a resource at a time.
   ii. **Hold and Wait:** Threads hold resources while waiting for additional ones.
   iii. **No Preemption:** Resources cannot be forcibly taken; they must be released voluntarily.
   iv. **Circular Wait:** A circular chain of threads exists, with each waiting for a resource held by the next.

   Deadlock Prevention: At least one of the above conditions must be eliminated:
   i. **Eliminate Mutual Exclusion:** Disallow exclusive control of resources (impractical in most cases).
   ii. **All-or-Nothing Allocation:** Acquire all required resources simultaneously or retry later.
   iii. **Preempt Resources:** Allow resources to be forcibly taken from lower-priority threads.
   iv. **Resource Ordering:** Enforce a strict total ordering of resource acquisition.

RCU: Read-Copy Update

RCU is a synchronization mechanism that minimizes lock usage, enabling concurrent reads and updates to pointer-linked elements. It is effective for data structures that are **read frequently but updated infrequently** (e.g., Linux dentry cache). RCU is implemented and extensively used in the Linux kernel. It supports only a limited set of data structures (e.g., linked lists, trees, hash tables).

Core Synchronization Strategy

- Writers are serialized using a lock.
- Reads are lock-free, minimizing overhead for most operations.
- Updates ensure readers always observe a consistent view of the data structure.

RCU Linked List

- **Adding a Node:** Writers create and initialize a new node before linking it into the list, making it safe to read as soon as it's observed by other threads. Updates to the list pointer are lock-free but may require (acquire & release) memory barriers .
- **Removing a Node:** Writers adjust the list pointer to skip over the node. Deallocation is deferred until all concurrent readers finish accessing the node.
- Readers traversing the list always have a consistent view.
  - Readers concurrent with a writer observe either the old or the new state.
  - Readers that start after the writer ends are guaranteed to observe the new state.
- The node removed from the list is safe to be deallocated only when the concurrent readers are finished traversing the data structure. To ensure a **safe deallocation**:
  - Readers traversing an RCU-protected data structure must not sleep (e.g., due to kernel preemption or I/O waits). Consequently, if a reader is descheduled from its CPU, it is guaranteed to have finished accessing the data structure.
  - After a write operation, when every CPU has **quiesced** (indicated by at least one call to `schedule()`), all concurrent readers must have returned, and the removed node can be safely deallocated.
  - CPUs maintain `schedule()` event counters which can be used to determine if a CPU has quiesced.

RCU API

- `rcu_read_lock()`: Lightweight read lock.
- `rcu_assign_pointer()`: Publishes new data (includes a release memory barrier).
- `rcu_dereference()`: Accesses current data (includes an acquire memory barrier).
- `call_rcu(object, delete_func)`: Defers deletion until all readers finish.

Operating System Transactions

Applications frequently rely on multiple system calls to update a system resource. If these system calls are not executed atomically, a **system-level race condition** can occur, leading to inconsistent updates, such as in the case of a **TOCTOU** vulnerability.

A common approach to addressing this issue is to introduce **more complex system calls** that combine the functionality of several simpler ones. However, this can become unmanageable and would require OS developers to continually add new system calls to meet the evolving needs of users.

OS transactions offer a straightforward and effective way to enforce consistency requirements across system calls. By grouping multiple system calls into atomic and isolated units, OS transactions ensure consistent updates in the presence of concurrent users. These transactions adhere to **ACID** properties (Atomicity, Consistency, Isolation, Durability). Code regions that require consistency are enclosed within `sys_xbegin()` and `sys_xend()`, while an in-progress transaction can be aborted using `sys_xabort()`.

Version Management

TxOS maintains multiple versions of kernel data structures so that a transaction can **isolate the effects** of system calls **until it commits**, and in order to undo its effects if it cannot complete.

Shared kernel data structures are decomposed into the **header component** (which are never replaced by transactions) and **data component**, pointed to from the header. Each transaction maintains a private data component for each object it modifies, so it can isolate the effects of system calls until it commits.

Transactions **commit** changes by replacing the pointer in the header to a modified copy of the data, and **abort** by simply discarding the private data.

Conflict Detection & Resolution

TxOS allows for the use of both transactional and non-transactional code, enabling incremental adoption of transactions and back-portability. As a consequence, serializing transactions and non-transactional system calls is necessary.

The object's header component is augmented with information about transactional readers and writers, allowing for detection of accesses that would violate serializability of both transactions and non-transactional system calls.

To **resolve a conflict** (between two transactions, at least one of which updates the object), a contention manager is used to arbitrate the resolution of the conflict (e.g., by favoring threads with higher scheduling priority).

To **resolve an asymmetric conflict** (between a transaction and a non-transactional system call, at least one of which updates the object), the kernel utilizes preemption - it deschedules the non-transactional thread before executing the system call, allowing the transaction to complete before scheduling the non-transactional thread again.

False Conflicts. To minimize false conflicts and increase concurrency, TxOS supports defining conflicts for each object type separately. An example of a false (write/write) conflict is when two transactions (or a transaction and a system call) create two different files in the same directory without reading the directory's content. .

Transactions for Application State

TxOS transactions can be used to manage system-level state only. User-level applications must use different techniques for managing application-level state (CoW paging, hardware or software transactional memory, etc.).

Performance

- Installing a Linux package inside a transaction incurs up to 70% overhead because the data is first written to a private copy, and then published to the rest of the sytem.
- The use of transactions causes write speedups of up to 20x (HDD) due to better I/O scheduling. The boundaries of a transaction provide I/O scheduling hints to the OS.
- Non-transactional system calls have a 42% overhead (geomean), which can be reduced to 14% with additional optimizations.