2019 Spring - System Programming Finals

Concurrent Programming

- Hard!
  - Human mind is sequential, misleading notion of time
  - Considering all possibilities of **interleaving** is impossible
  - **Races**: Outcome depends on arbitrary scheduling decisions
  - **Deadlock**: Improper resource allocation preventing progress, stuck waiting for an event that will never happen
  - Livelock, starvation, fairness etc.
- **Concurrent Servers**
  - **Process**-based: Automatic interleaving by kernel, private address space for each flow
  - **Event**-based: Manual interleaving, shared address space, *I/O multiplexing*
  - **Thread**-based: Automatic interleaving by kernel, shared address space (Mixup)
- **Process Based Server**
  - Separate process for each client (`fork`)
  - Must reap all zombie children
  - `accept` process
    1. Server blocks in `accept`, waits for connection request on `listenfd`
    2. Client makes connection request by `connect`
    3. Server returns `connfd` from `accept`, *Forks child to handle client*
    4. Connection between `clientfd` and `connfd` is established
  - *No shared states between clients*
  - Both parent & child have copies of `listenfd` and `connfd`: parent should close `connfd`, child should close `listenfd` (considering `refcnt`)
  - Pros
    * Clean sharing model - file tables (o), descriptors/global var.(x)
    * Simple and straightforward
  - Cons

    * Additional overhead for process control
    * Hard to share data between processes (IPC)

- **Event Based Server**
  - Maintains a set of active connections by an array of `connfds`
  - Repeats:
    * `select` which descriptors have pending inputs
    * If `listenfd` has input, `accept` the connection
    * Add new `connfd` to array
    * Service all `connfds` with pending inputs
  - Pros
    * One logical control flow and shared address space
    * Can single step with debugger
    * No process or thread control overhead
    * Gives programmers more control over the behavior
  - Cons
    * Too complex
    * Hard to provide fine-grained concurrency
    * Cannot take advantage of multi-core (single control)
- **Thread**: Logical flow that runs in the context of a process
  - **Thread context**: Registers, Condition Codes, Stack Pointer, Program Counter, Thread ID, own stack (for local var)
  - Threads share same code, data, and kernel context (VA space)
  - Threads $\approx$ pools of concurrent flow that access the same data[1]
  - Concurrent if flows overlap in time
- **Threads vs. Processes**
  - Similarities
    * Own logical control flow
    * Can run concurrently with others

[1] Processes form a tree hierarchy, where threads do not

    - Context switching
  - **Differences**
    - Threads share all code and data (except local stacks)
    - Threads are less expensive than processes (they have less context)

- **Posix Threads** (`pthreads`) **Interface**
  - Standard interface that manipulate threads from C programs
  - Creating Threads
    - `int pthread_create(pthread_t *tid, NULL, func *f, void *arg);`
    - `tid`: contains id of created thread
    - `f`: thread routine
    - `arg`: arguments for `f`
  - Terminating Threads
    - `void pthread_exit(void *thread_return);`
    - `int pthread_cancel(pthread_t tid);`
    - Terminates the thread with `tid`
  - Reaping Threads
    - `int pthread_join(pthread_t tid, void **thread_return);`
    - Blocks until thread `tid` terminates, and reaps terminated thread
    - Can only wait for a specific thread
  - Detaching Threads
    - Joinable thread: Can be reaped and killed by other threads, memory is not freed until reaped.
    - Detached thread: Cannot be reaped by other threads, memory is freed automatically on termination
    - `int pthread_detach(pthread_t tid);`

- **Thread Based Server**
  - Run only detached threads: reaped automatically
  - Free `vargp`, `close(connfd)` necessary
  - Each client handled by each peer thread
  - Careful to avoid unintended sharing (`malloc`)

- Functions in the thread routine must be thread-safe
- Pros
  - Easy to share data between threads (perhaps too easy)
  - Efficient than processes (cheaper context switch)
- Cons
  - Unintended sharing
  - Difficult to debug

Synchronization

- **Threads Memory Model**
  - Variable shared $\iff$ Multiple threads reference the variable
  - Multiple threads run within the context of a single process
  - Threads have its own thread context (TID, SP, PC, CC, REG)
  - Share the remaining process context
- **Variable Instances in Memory**
  - Global Variables: Exactly one instance
  - Local Variables: Each thread stack has one instance each
  - Local *Static* Variables: Exactly one instance[2]
- **Concurrent Execution & Process Graphs**
  - Interleaving of any order possible; May cause errors
  - **Process Graph** depicts the discrete *execution state space* of concurrent threads
  - Axis: sequential order of instructions in a thread
  - Each point: Possible *execution state*
  - Trajectory: is a sequence of legal state *transitions* of possible concurrent execution (one set of interleaving)
  - **Critical Section** (w.r.t a shared var): load $\sim$ store instruction
  - Instructions in critical section should not be interleaved
  - **Unsafe Region**: Intersection of critical sections
  - Trajectory is *safe* $\iff$ does not pass unsafe region
  - Enforce **mutual exclusion** to **synchronize** the execution of threads so that they can never have an unsafe trajectory
- **Semaphores**: Non-negative global integer synchronization variable
  - Mainpulated by `P`, `V` operations
  - `P(s)` (= Lock)
    * If $s \neq 0$, `s--` and return (happens atomically)

[2] It's similar to global variables, just that its scope is limited to the function

    - If $s = 0$, **suspend** until $s \neq 0$, and the thread is restarted by a `V` operation
    - After restart, `P` decrements `s` and returns control to caller
  - `V(s)` (= Unlock)
    - `s++` and return
    - If any threads blocked in `P` are waiting, restart exactly one of those threads,[3] which enables `P` to decrement `s`.
  - **Semaphore Invariant**: $s \geq 0$

- **Semaphores for Synchronization**
  - Associate a unique semaphore **mutex** (initially 1) with each shared var
  - Surround corresponding critical sections with `P`, `V` operations
  - *Binary Semaphores*: Value is 0 or 1
  - *Mutex*: Binary semaphores for **mut**ual **ex**clusion
  - *Counting Semaphore*: Counter for set of available resources
  - Synchronization makes programs run slower
  - The semaphore invariant surrounds critical sections, which is the *forbidden region*
  - Semaphore is < 0 in the forbidden region, therefore cannot be passed by any trajectory

- **Semaphores to Coordinate Access to Shared Resources**
  - Semaphore operation to notify another thread that some condition has become true
  - Use counting semaphores to keep track of resource state
  - Mutex for protecting access to the resource

- **Producer-Consumer Problem**
  - They share a bounded buffer with n slots
  - Producer produces new items, inserts them to the buffer, notify consumer
  - Consumer consumes items, removes them from the buffer, notify producer
  - `sbuf` (shared buffer) package
  - `slots`: counts available slots in the buffer
  - *items*: counts available items in the buffer

[3] You don't know which will be restarted...

- **Reader-Writers Problem**
  - Reader threads only read object
  - Writer threads modify the object $\rightarrow$ Must have exclusive access
  - Unlimited readers can access the object
  - *First readers-writers problem* (Reader Favoring)
    * No reader should be kept waiting if writer doesn't have access
    * Reader has priority over writers
    * Starvation for writers may happen
  - *Second readers-writers problem* (Writer Favoring)
    * Once a writer is ready to write, performs write ASAP
    * Readers that arrive after a writer must wait, even if the writer is also waiting
    * Starvation for readers may happen
- **Prethreaded Concurrent Server**
  - Creating/reaping thread is expensive! Maintain a set of worker threads!
  - Server consists of main thread and a set of worker threads
  - Main thread repeatedly accepts connection from clients and places `connfd` in a bounded buffer
  - Each worker thread removes `connfd` from the buffer, services client and waits for the next descriptor
- **Thread Safety**
  - Functions called in a thread routine must be **thread safe**
  - Thread Safe $\iff$ Always produces correct results when called repeatedly from multiple concurrent threads
  - Classes of unsafe functions
    1. Functions that do not protect *shared variables*
       * Use `P`, `V` operations to synchronize
    2. Functions that keep *states across multiple invocations*
       * Modify function to be *re-entrant*
    3. Functions that return a *pointer to a static variable*

        * Rewrite function so caller passes address of variable to store result
        * Lock and copy: Lock and copy to a another private memory location to store the result (write a wrapper function)
    4. Functions that call other unsafe functions
        * Just don't call them

- **Reentrancy**
    - Function is **reentrant** $\iff$ Does not access shared variables when called by multiple threads
    - Requires no synchronization process (which is expensive)
- **Race Conditions**
    - Race when the correctness of program depends on on thread reaching point x before another thread reaches y
    - Happens usually when programmer assumes some particular trajectory
    - Avoid unintended sharing to prevent races
- **Deadlocks**
    - Deadlock $\iff$ Waiting for a condition that will never be true
    - `P` operation is a potential problem because it blocks
    - Trajectory entering deadlock region will reach deadlock state
    - Often non-deterministic
    - Fix: Acquire shared resources in the same order

Thread-Level Parallelism

- Multicore/Hyperthreaded CPUs offer another opportunity
  - Spread work over threads executing in parallel
  - Happens automatically, if many independent tasks
  - Write code to make one big task go faster
- Out-of-Order Processor Structure
  - Instruction control dynamically converts program into stream of operations
  - Mapped onto functional units to execute in parallel
- Hyperthreading Implementation
  - Replicate enough instruction control to process K instruction streams
  - K copies of all registers, share functional units
- Summation Example
  - `psum-mutex`: Takes too long! `P`, `V` operations are expensive
  - `psum-array`: Peer thread `i` sums into global array element `psum[i]`. Eliminates need for mutex synchronization
  - `psum-local`: Reduce memory references. Sum into a local variable.
- Characterizing Parallel Performance
  - p processor cores, T_k is the running time using k cores
  - **Speedup**: $S_p = T_1/T_p$
    - *Relative Speedup* if T_1 is run time of parallel ver. of the code on 1 core
    - *Absolute Speedup* if T_1 is run time of sequential ver. of the code on 1 core
  - **Efficiency**: $E_p = S_p/p = \frac{T_1}{pT_p}$
    - Measures the overhead due to parallelization
- **Amdahl's Law**: Captures difficulty of using parallelism to speed things up
  - T: Total sequential time required
  - p: Fraction of total that can be sped up $(0 \leq p \leq 1)$
  - k: Speedup factor

- Resulting performance

$$T_k = p\frac{T}{k} + (1-p)T \implies S_p = \frac{T}{pT/k + (1-p)T} = \frac{1}{1-p+\frac{p}{k}}$$

- Least possible running time: $k = \infty \implies T_\infty = (1-p)T$

- Snoopy Caches
  - Write-back caches, without coordination between them may cause problems...!
  - Tag each cache block with states
    - `I`: Invalid - Cannot use value
    - `S`: Shared - Readable copy
    - `E`: Exclusive - Writeable copy
  - When cache sees request for one of its `E` tagged blocks: Supply value from cache and set tag to `S`

Spin Locks and Contention

- Kinds of Architectures
  - SISD (Single Instruction Single Data - Uniprocessor)
    * Single instruction stream
    * Single data stream
  - SIMD (Single Instruction Multiple Data - Vector)
  - MIMD (Multiple Instruction Multiple Data - Multiprocessors)
- Spin Lock
  - Lock which causes a thread to acquire it to simply wait in a loop while repeatedly checking if the lock is available.
  - Thread is active but does not perform any useful task (busy waiting)
- Test-and-Set
  - Instruction used to write 1 (true) to a memory location and return its old value as a single atomic operation
  - No other process may begin another test-and-set until the first process's test-and-set is finished
  - Can reset by writing 0 (false)
  - Lock is free is value is false
  - Lock is taken if value is true
  - Release lock by writing false
- Test-and-Test-and-Set Locks
  - Lurking stage
    * Wait until lock looks free
    * Spin while read returns true (lock is taken)
  - Pouncing State
    * As soon as lock looks available
    * Read returns false (lock is free)
    * Call TAS to acquire lock
    * If TAS loses, back to lurking

- Mystery
  - Both TAS and TTAS do the same thing (in our model)
  - Except that TTAS performs much better than TAS
  - Neither approaches ideal
- Write-Back Caches
  - Accumulate changes in cache
  - Write back when needed - need the cache for sth else, another processor wants it
  - On first modification - invalidate other entries, requires non-trivial protocol
  - Three States
    - Invalid: Contains raw bits
    - Valid: I can read but I can't write
    - Dirty: Data has been modified - Intercept other load requests and write back to memory before using cache
- Why does `TASLock` perform so poorly?
  - Because all threads must use the bus to communicate with memory, these `getAndSet()` calls delay all threads, even those not waiting for the lock
  - `getAndSet()` call forces other processors to discard their own cached copies of the lock, so every spinning thread encounters a cache miss almost every time, and must use the bus to fetch the new, but unchanged value
  - When the thread holding the lock tries to release it, it may be delayed because the bus is monopolized by the spinners
- `TTASLock` algorithm
  - Lock is held by a thread A. The first time thread B reads the lock it takes a cache miss, forcing B to block while the value is loaded into B's cache
  - As long as A holds the lock, B repeatedly rereads the value, but hits in the cache every time.
  - B thus produces no bus traffic, and does not slow down other threads' memory accesses.
  - Moreover, a thread that releases a lock is not delayed by threads spinning on that lock
  - *When the lock is released*

    - The lock holder releases the lock by writing false to the lock variable, which immediately invalidates the spinners' cached copies
    - The first to succeed invalidates the others, who must then reread the value, causing a storm of bus traffic. Eventually, the threads settle down once again to local spinning
- *local spinning*, where threads repeatedly reread cached values instead of repeatedly using the bus, is an important principle critical to the design of efficient spin locks
- **contention** occurs when multiple threads try to acquire a lock at the same time
- Backoff
    - more effective for the thread to back off for some duration, giving competing threads a chance to finish
    - Easy to implement, Beats TTAS lock
    - Must choose params. carefully, not portable
- Anderson Queue Lock
    - Shorter handover than backoff
    - FIFO fairness
    - First truly scalable lock
    - Simple, easy to implement
    - Space hog, One bit per thread
- CLH Lock
    - FIFO order, No starvation
    - Small, constant size overhead per thread
    - Lock release affects predecessor only
    - Small, constant sized space
    - But doesn't work for uncached NUMA architectures
    - Each thread spin's on predecessor's memory
    - Could be far away...
    - L: number of locks
    - N: number of threads
    - ALock: $O(LN)$, CLH: $O(L+N)$
- NUMA Architectures

    * Non-Uniform Memory Architecture
    * Illusion: flat shared memory
    * Truth: No caches, some memory regions faster than others
- MCS lock
    * FIFO order
    * Spin on local memory only
    * Small, constant size overhead