Lecture Topics

• semaphores and their uses
• dependence and condition variables
• performance reasoning for shared memory threads (Lab 2)
• detecting errors in synchronization

Administrivia

• Lab 2 out … due in two weeks
• see POSIX interface material in notes
Semaphores and Their Uses

- a semaphore is a counter that supports a fixed number of participants
  - up/post/increment and down/wait/decrement operations
  - (for the original Dutch symbols, see the 391 notes)

- consider two groups of threads
  - workers: perform tasks (servers)
  - supervisors: assign tasks to workers (clients?)

- three experiments to demonstrate
  - allowing supervisors to sleep (i.e., not waste CPU cycles)
  - fairness issues between supervisors

- Experiment #1: several workers, one supervisor, no semaphore
  - workers repeat: wait for task in your queue, execute task
  - unique supervisor repeats
    - wait for empty worker position (repeat check in round robin order)
    - place a task sheet in the empty position
  - How much time does the supervisor spend idling?
    (This time corresponds to wasted CPU cycles.)

- Experiment #2: several workers, fast and slow supervisors, no semaphore
  - workers as before
  - supervisors repeat
    - look for an empty position (check once in round robin order)
    - if none is available, delay for time commensurate with your speed, then start looking again
    - place a task sheet in the empty position
      (atomically w.r.t. other supervisor!)
  - Do slow and fast supervisors assign a roughly equal number of tasks?
• Experiment #3: N workers, fast and slow supervisors, with a semaphore!
  – semaphore: starts at N
  – workers repeat
    • wait for task in your queue, execute task
    • NEW: up the semaphore
  – supervisors repeat
    • NEW: down the semaphore
    • look for an empty position (always available)
    • place a task sheet in the empty position (atomically w.r.t. other supervisor!)

• same questions from first two experiments…
  – How much time do supervisors spend idling?
    • None.
    • all workers busy means
      – semaphore is at 0
      – supervisors go to sleep immediately
  – Do slow and fast supervisors assign a roughly equal number of tasks?
    • Yes.
    • when a worker finishes, one supervisor woken up
    • semaphore maintains a queue, so they take turns being woken
    • If workers finish tasks faster than slow supervisor can assign,
      – fast supervisor will still have advantage
      – But who cares? Slow supervisor assigning as fast as it can!

• Note: semaphore with initial value 1 is a mutex lock.
Posix Semaphores [NOTES ONLY]

- not part of pthreads package (part of the earlier real-time extension)
- attributes not invented yet
- couldn’t change later (hard to abandon existing code)

- sem_t
  - dynamic initialization
    
    ```c
    int sem_init (sem_t* sem, int pshared,
                  unsigned int value);
    ```

- typical use
  - sem_wait
  - sem_post

- warnings
  - sem_wait can “fail” when interrupted by a signal (returns EINTR)
  - [mutex allows signal, but returns to waiting after it’s handled]
  - semaphore interface does not provide priority inheritance, etc.
  - although some implementations may support it
Posix R/W Locks [NOTES ONLY]

- pthread_rwlock_t
  - static init: = PTHREAD_RWLOCK_INITIALIZER;
  - dynamic init

```c
int pthread_rwlock_init (pthread_rwlock_t* rwlock,
                        const pthread_rwlockattr_t* attr);
```

- typical use
  - pthread_rwlock_rdlock
  - pthread_rwlock_wrlock
  - pthread_rwlock_unlock (both types)

- semantics
  - many readers, only one writer (no difference)
  - return value EAGAIN implies too many readers currently
  - normally puts process to sleep if can’t get lock (try lock calls do exist)
  - writer starvation is left to implementation
    - writer may wait indefinitely
    - or may be prioritized over later read lock requests
**Dependence and Condition Variables**

- problems with unprotected conditions
  - consider a producer-consumer relationship
  - consumer goes to sleep if no work is available
  - producer wakes sleeping consumer after depositing a new task

- What can happen if we write such code without additional synchronization?

- Do “arbitrary” interleavings work correctly?

<table>
<thead>
<tr>
<th>Parasina</th>
<th>Azo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deposit new task for Azo.</td>
<td>Check for a new task.</td>
</tr>
<tr>
<td>If Azo is asleep, wake him up.</td>
<td>If no task is available, sleep.</td>
</tr>
</tbody>
</table>

![Time diagram showing the interaction between Parasina and Azo](https://via.placeholder.com/150)
• How about this interleaving?

Parasina

Deposit new task for Azo.
If Azo is asleep, wake him up.

Azo

Check for a new task.

If no task is available, sleep.

• One solution: critical sections; now are all interleavings ok?

Parasina

Deposit new task for Azo.
If Azo is asleep, wake him up.

Azo

Check for a new task.
If no task is available, sleep.
• producer lock
   – need to have it at some point between deposit and wake-up
   – may need it for larger critical section than wake-up alone
   – wake-up call does not explicitly mention lock; be careful!

• waiting on a condition variable
  int pthread_cond_wait (pthread_cond_t* cond,
                       pthread_mutex_t* mutex);

• signaling (wake one waiter) / broadcasting (wake all waiters)
  int pthread_cond_signal (pthread_cond_t* cond);
  int pthread_cond_broadcast (pthread_cond_t* cond);

• execution order (waiting on condition variable)
  – grab a lock
  – check condition; determine need to go to sleep
    • Why grab lock before checking?
      • for performance, you may want to check, lock, check again
  – call pthread_cond_wait
  – on your thread’s behalf, kernel
    • puts your thread to sleep
    • releases the lock
  – compare order with Azo’s behavior in previous slide
  – lock is reacquired before call returns
  – release the lock

• execution order (signaling a condition variable)
  – don’t forget to do it if you change a condition
  – obtain the correct lock (match the pthread_cond_wait argument)
  – call pthread_cond_signal
  – release the lock
Posix Barriers [NOTES ONLY]

- optional; not supported everywhere

- a barrier is a synchronization point between a set of threads
  - each thread enters the barrier by calling a function
  - threads block (sleep) in barrier function until all threads have called
  - after last thread calls function, all threads made runnable again

- pthread_barrier_t
  - static init: = PTHREAD_RWLOCK_INITIALIZER;
  - dynamic init

    int pthread_barrier_init (pthread_barrier_t* bar,
                            const pthread_barrierattr_t* attr,
                            unsigned count);

- typical use
  - pthread_barrier_wait

- note: again, no priority inheritance support
Thread-Specific Data [NOTES ONLY]

- each thread maintains a private data area

- stores key-value pairs
  - keys are pthread_key_t’s
  - values are pointers to arbitrary data (void*)

- create a new key
  
  ```c
  int pthread_key_create
  (pthread_key_t* new_key,
   void (*destructor) (void* arg));
  ```
  
  - sets associated value to NULL in all threads in process
  - destructor invoked when thread with non-NULL value terminates
  - argument is the value associated with key in terminating thread
  - destructor should set value to NULL
    (or it will be called repeatedly)

- set a value
  
  ```c
  int pthread_set_specific (pthread_key_t key,
                           const void* value);
  ```

- read a value
  
  ```c
  void* pthread_get_specific (pthread_key_t key);
  ```

- destroy a key
  
  ```c
  int pthread_key_delete (pthread_key_t key);
  ```
  
  - safe to call from destructors
  - does NOT invoke destructors!
Execute Exactly Once Semantics [NOTES ONLY]

• motivating example
  – some module used dynamically by several threads
  – must initialize module exactly once
  – must make sure that module is initialized before use
  – if module initialization is expensive
    • want to avoid if never used
    • try to execute lazily (on first use)
    • which thread is first? race condition..

• solution: pthread_once
  – static variable holds state
    
    ```c
    pthread_once_t state = PTHREAD_ONCE_INIT;
    ```
  
  – to call, execute
    
    ```c
    int pthread_once (pthread_once_t* state_ptr,
                     void (*func) (void));
    ```

  – state variable used to serialize
    • only one call is made by some thread
    • no thread passes call until single execution completes
Shared Memory Thread Performance

• some things you should know, but people often forget
  – parallelizing slow code results in slow parallel code
    • Andrea Arpaci-Dusseau studied parallel sorts in her MS
    • biggest performance impact?
    • performance of threads on individual processors!
    • if 50% of each thread’s time is spent in STL…
      – guess what fraction of total time is in STL
      – guess what happens if you parallelize an important
        STL container with one big lock
    – locality is important, particularly with shared hardware resources
      • shared memory threads compete for resources
      • poor locality can lead to thrashing (performance cliffs)
    – premature optimization is bad…
      • more complexity
      • the fast part is faster…now what?
  – …usually
    • parallel hardware more likely to vary/change
    • some planning for future hardware may not be a bad idea
• some “free” gains with shared memory threads
  – good management
    • any thread can run on any processor
    • don’t let a processor idle unless there are < P tasks left
    • note: load imbalance is not impossible!
  – bad management
    • don’t let tasks finish
      – interleave them instead
      – more competitors for cache resources
      – more switching overhead
    • migrate regularly
      – bonus communication
      – move data back and forth between caches
  – tension here
    • optimal solution depends on shared resource use
      – data sharing relations between threads
      – synchronization variables
      – microarch. with core multithreading
    • often not easy to understand, let alone express, locality
    • similarly difficult to model cost of migration
    • heuristics try to capture in some models

• synchronization bottlenecks
  – simplicity and relatively low cost of synchronization can be alluring
  – but most synchronization mechanisms designed for low contention
  – performance falls off quickly with higher contention
  – consider
    • how long should a thread wait on a variable before yielding?
    • all of that time is idle time…
    • as is the switch overhead
  – again, not clear that there is a “correct” decision
• extra complexity in shared memory systems
  
  – false sharing
    • hardware defines size of shared data
      – e.g., cache line, page
      – exact size is not well-defined
    • hardware protocol thrashes for finer-grained sharing
      – e.g., two processors write disjoint words in a cache line
      – line ping-pongs between caches
      – no actual sharing involved
      – but a huge performance penalty

  – performance tuning: sharing issues are not local in code
    • how do you reason about data sharing issues in parallel code?
    • sharing depends on data layout and parallel operations
    • parallel operations are regions of code
    • data layout is defined by data structures
    • not easy to optimize one operation independently of others
    • not easy to optimize entire body of code simultaneously