

Lecture Topics

- queue examples (algorithms)
- the concept of consistency models
- dependence and condition variables

Administrivia

- [none]

- single-producer, single-consumer FIFO queue (review?)

```
#define Q_LEN 256 // a power of 2—why?
void* queue[Q_LEN];
volatile int32_t q_head = 0;
volatile int32_t q_tail = 0;

bool enqueue (void* data)
{
    if (q_tail == q_head + Q_LEN) { return false; }
    queue[(q_tail % Q_LEN)] = data;
    q_tail++; // volatile disallows load to move upward
              // which in turn prevents store from moving...
    return true;
}
void block_enqueue (void* data)
{
    // now volatility of head is even more important!
    while (!enqueue (data));
}
void* dequeue ()
{
    void* ret_val;
    if (q_head == q_tail) { return NULL; }
    ret_val = queue[(q_head % Q_LEN)]; // read after increment?
    q_head++; // again, volatility is important here
    return ret_val;
}
void* block_dequeue ()
{
    void* ret_val;
    while (NULL == (ret_val = dequeue ()));
    return ret_val;
}
```

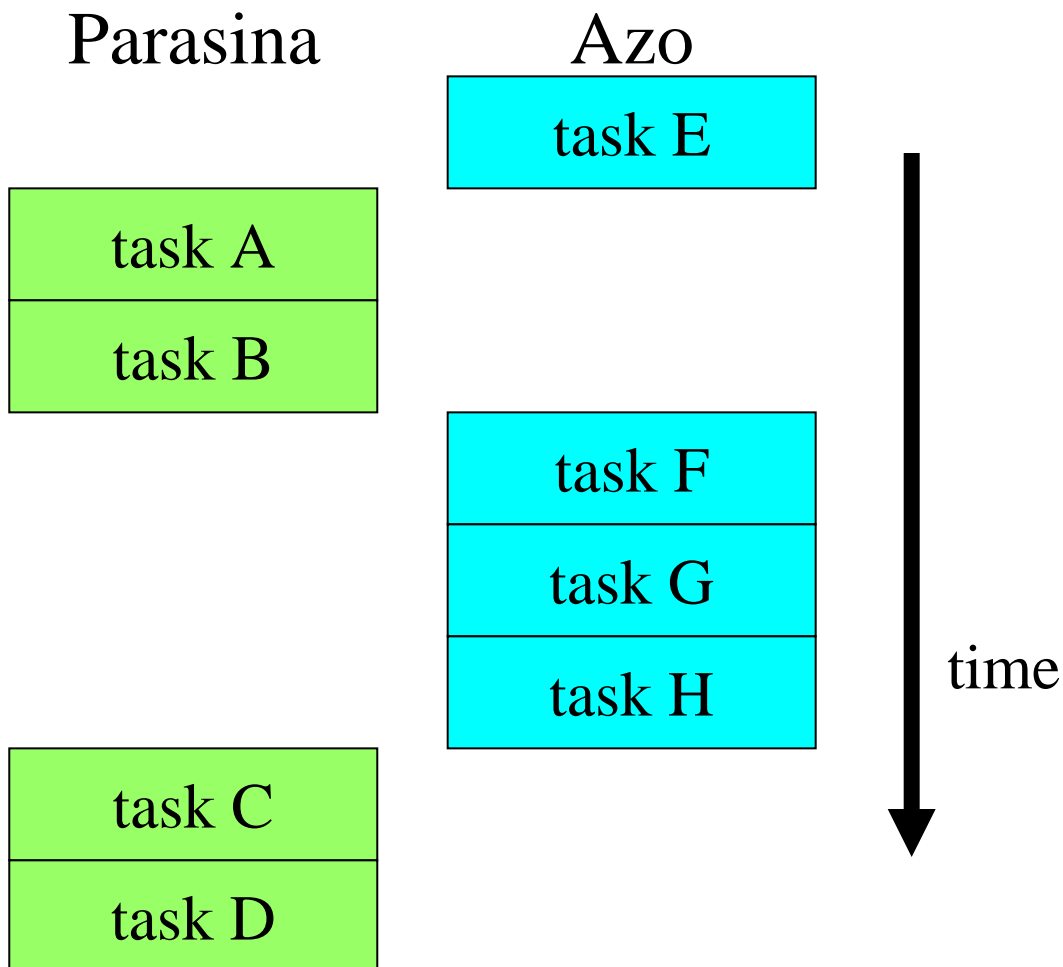
- multi-producer, single consumer queue (see Ch. 3 of my thesis on web page)
 - data structure
 - queue head & tail (separate cache lines)
 - array of one-cache-line “packets” with state + contents
 - code uses Fetch & Increment and Compare & Swap
 - enqueue sequence
 - packet assignment
 - packet claim (atomic state change: FREE to CLAIMED)
 - fill packet
 - packet ready for delivery (state change: CLAIMED TO READY)
 - packet read
 - packet ready for reuse (READY TO FREE; can be claimed again)

```
// returns index of claimed packet in array
int32_t claim_packet (queue_t* q)
{
    int32_t index = (fetch_and_increment (q->tail) % Q_LEN);
    while (1) {
        if (CAS (&q->packet[index].state, FREE, CLAIMED)) {
            return index;
        }
        // back off exponentially (& poll incoming queue...)
    }
}
```

- receiver simply checks state of packet at head of queue for READY
- assigned packets can exceed Q_LEN
 - threads compete for some packet slots
 - back off exponentially to avoid extra memory traffic
- this queue is neither FIFO nor linearizable!

The Concept of Consistency Models

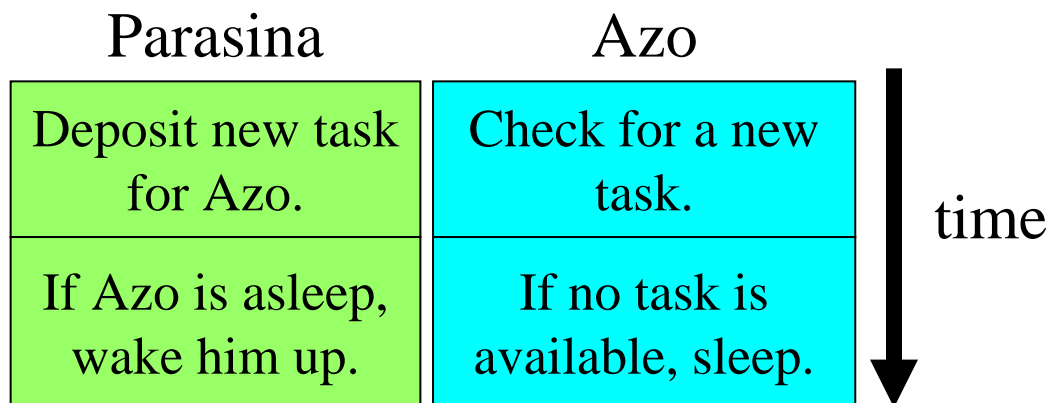
- a simple thread interaction model
 - starring Byron's Parasina and Azo, threads P and A
 - Parasina and Azo work concurrently, so their tasks can be logically interleaved...
- sequential consistency
 - instructions from different threads appear to execute in some interleaving
 - all instructions from any given thread are in execution order
 - all processors perceive the same total order of instructions
 - nice model, but not really how many modern machines work



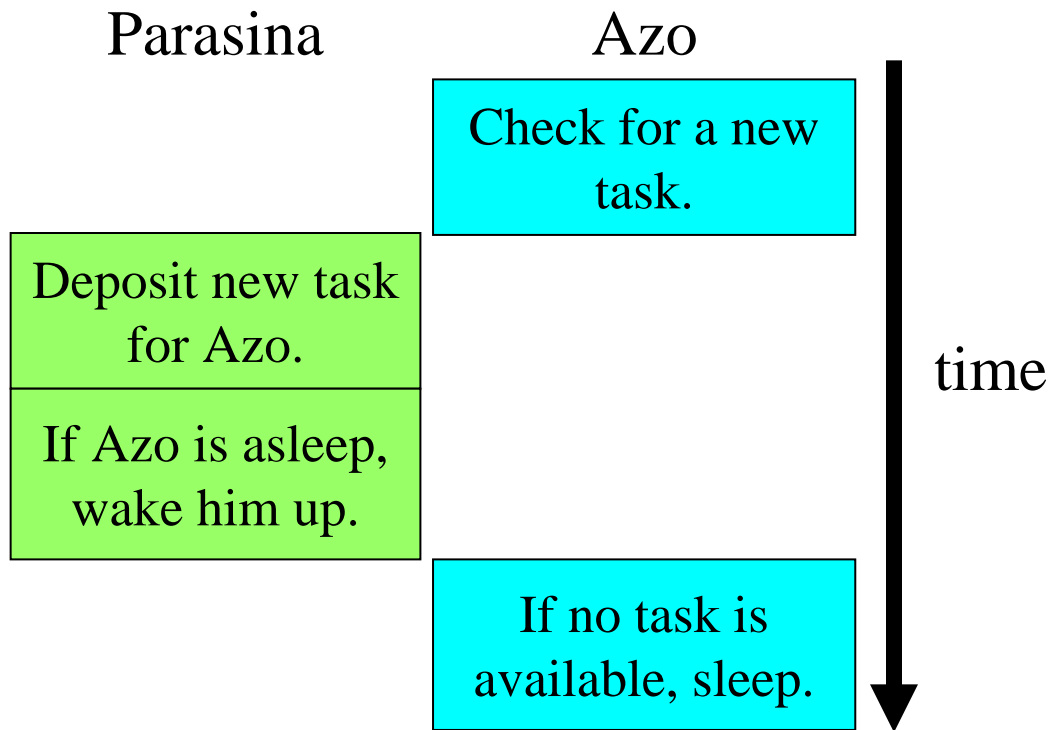
- some common variations and examples
 - operations from a single processor may be reordered
 - execute load before earlier store if addresses known not to conflict
 - What if load is inside a critical section, and store is the lock acquisition for the critical section?
 - perception of execution may be different on different processors
 - two stores go to different memory banks
 - Why wait for the first to finish before sending the second?
 - first is data, second is synchronization flag
 - Now can we change the perceived order of stores?
 - What if another processor loads data after flag changes but before data is stored?

Dependence and Condition Variables

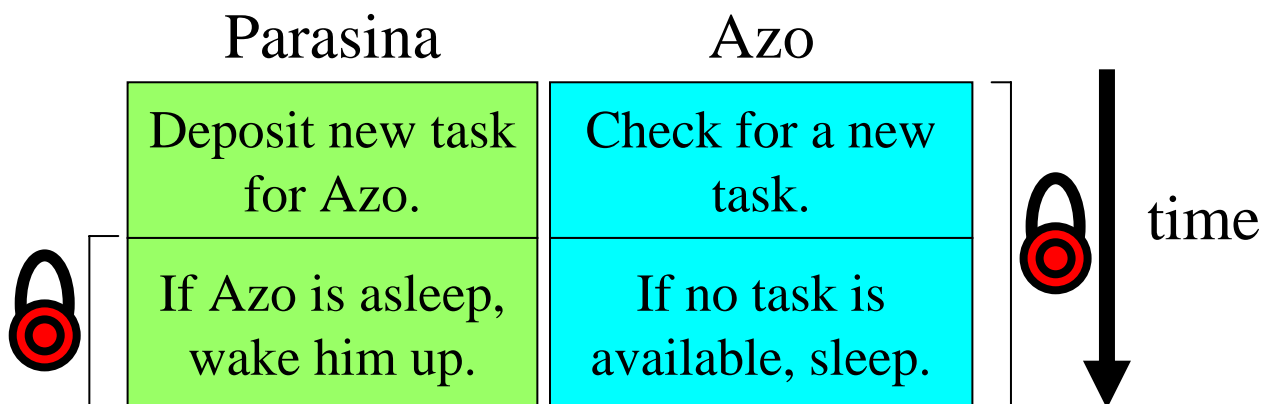
- problems with unprotected conditions
 - consider a producer-consumer relationship
 - consumer goes to sleep if no work is available
 - producer wakes sleeping consumer after depositing a new task
- What can happen if we write such code without additional synchronization?
- Do “arbitrary” interleavings work correctly?



- How about this interleaving?



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



- with Posix threads
 - you get a choice of whether to lock or not...
 - (from man pages...) “however, if predictable scheduling behavior is required, then that mutex shall be locked”
- 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?
 - you might want to have T&T&S-like behavior sometimes:
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