csci 3411: Operating Systems

Synchronization

Gabriel Parmer

Slides evolved from Silberschatz and West

TODO for next year.

- Change i and j to me and them
- Add interactive sessions for each of the algorithm where they sit and work it out.

Synchronization Motivation

- Multithreaded applications: threads share
  - ...the same virtual address space
  - ...share the same data-structures
- Concurrently executing threads
  - ...have unknown execution order w.r.t. each other
  - ...can access data-structures in unpredictable order
- How does a system make this work!?

Linked List...of Students

```
struct student_node {
	struct student_node *next = NULL
	char *name
}
struct student_node *list = NULL
```

```
list_push(list, new_sn):
	tmp = list
	new_sn->next = tmp
	list = new_sn
```

```
list_find(list, name):
	while (n = list ; n ; n = n->next):
		if (n->name == val) return n
	return NULL
```

```
list_pop(list):
	tmp = list
	if (tmp):
		list->first = tmp->next
		tmp->next = NULL
	return tmp
```

1) Adding while adding?
2) Adding while finding?
3) Adding while removing?
4) Removing while finding?

Producer/Consumer Problem

Producer:

```
while(1) {
    struct item i = produce_item();

    while (count == BUFFER_SIZE)
        ;

    buffer[in] = i;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

Consumer:

```
while(1) {
    struct item i;

    while (count == 0)
        ;

    i = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    consume_item(i);
}
```

Synchronization Motivation

- count++ is really

```
tmp = count;
tmp = tmp+1;
count = tmp;
```

- count-- is

```
tmp = count;
tmp = tmp – 1;
count = tmp
```

Synchronization Motivation

- count++ is really

  tmp = count;

  tmp = tmp+1;

  count = tmp;

```
mov count_mem_addr, %reg0
add %reg0, $1
mov %reg0, count_mem_addr
```

- count-- is

  tmp = count;

  tmp = tmp – 1;

  count = tmp

```
mov count_mem_addr, %reg0
sub %reg0, $1
mov %reg0, count_mem_addr
```

Synchronization Motivation

- Initially, say count = 1
- If two threads execute “count++” and “count--” concurrently
- What is count?

Synchronization Motivation

```
mov count_mem_addr, %reg0
add %reg0, $1
mov %reg0, count_mem_addr
mov count_mem_addr, %reg0
sub %reg0, $1
mov %reg0, count_mem_addr
```

```
mov count_mem_addr, %reg0
mov count_mem_addr, %reg0
add %reg0, $1
mov %reg0, count_mem_addr
sub %reg0, $1
mov %reg0, count_mem_addr
```

```
mov count_mem_addr, %reg0
mov count_mem_addr, %reg0
sub %reg0, $1
mov %reg0, count_mem_addr
add %reg0, $1
mov %reg0, count_mem_addr
```

? ? ?

What is count in each case?

Principle of Synchronization

- The buffer in the producer/consumer is inconsistent without an accurate “count”
- *Arbitrary interleavings* of the execution of concurrent threads when accessing *shared data* can lead to inconsistency
  - Otherwise known as race conditions
  - We used “count”, could be e.g. pointers in a linked list
- Threads accessing data must cooperate to access data one at a time using some method that enforces this *synchronization*

Synchronization in the Kernel

- Operating system kernels must worry about synchronization
  - Interrupts made kernel code concurrent
    - Normal kernel code:
      count++
    - Interrupt service routine (ISR):
      count--
  - Ouch.
  - Threads...everywhere!

Critical Sections

- Segments of code that access shared data
  - Only one thread of control at a time can execute in a critical section
  - Put another way: Critical sections require *mutually exclusive* access
- Main problem: How can the system provide mutually exclusive access to shared data?
  - In a manner that is easy to program

Critical Section Solution Criteria

1) Mutual exclusion – No two threads can concurrently access in the critical section (CS)
2) Progress – threads wishing to enter an “unoccupied” CS cannot be indefinitely prevented from doing so
3) Arbitrary interleaving – no assumptions regarding relative speeds of thread execution can be made
4) Bounded Waiting – the number of times other threads enter the CS before a specific thread is chosen must be bounded

First Naive Attempt

- “CS_occupied” initialized to false

```
while (1) {
    normal_processing();
    while (CS_occupied) ;
    CS_occupied = true;
    critical_section_code();
    CS_occupied = false;
}
```

Satisfy all critical section properties?

First Naive Attempt

- “CS_occupied” initialized to false

```
while (1) {
    normal_processing();
    while (CS_occupied) ;
    CS_occupied = true;
    critical_section_code();
    CS_occupied = false;
}
```

Satisfy all critical section properties?

You try!!! Mutual exclusion?

First Real Attempt: Two Threads

- Alternation between threads
  - Thread id *me* is “current” thread, *you* is “other” thread
  - “turn” initialized to *me*

```
while(1) {
    normal_processing();
    while (turn != me);
    critical_section_code();
    turn = you;
}
```

Problems?

Second Attempt: Peterson's Alg.

```
// is a thread trying to enter a CS:
boolean flag[2] = {false, false};
int turn = me; // either me or you

while(1) {
    normal_processing();
    flag[me] = true;
    turn = you;
    while ((flag[you] == true) && (turn == you)) ;
    critical_section();
    flag[me] = false;
}
```

Second Attempt: Peterson's Alg.

```
boolean flag[2] = {false, false};
int turn = 0;
me = pthreads_self(); // thread library function
you = other_thread_id(); // our function
if (!turn) turn = me;

while(1) {
    normal_processing();
    flag[me] = true;
    turn = you;
    while ((flag[you] == true) && (turn == you)) ;
    critical_section();
    flag[me] = false;
}
```

Second Attempt: Peterson's Alg.

```
boolean flag[2] = {false, false};
int turn = i;
```

```
// me = red, you = blue
while(1) {
    normal_processing();
    flag[i] = true;
    turn = j;
    while ((flag[j] == true)
            && (turn == j)) ;
    critical_section();
    flag[i] = false;
}
```

```
// j = blue, i = red
while(1) {
    normal_processing();
    flag[j] = true;
    turn = i;
    while ((flag[i] == true)
            && (turn == i)) ;
    critical_section();
    flag[j] = false;
}
```

More than Two Threads: Bakery Alg.

- Bakery algorithm (or the DMV alg.):
  - Get a ticket
  - If you have the lowest ticket, you're served next!
  - But two customers can have the same number...
    - Use ID to break ties
    - Thread 1 proceeds before thread 2 as 1<2
    - Threads must be numerically identified

Bakery Algorithm II

- Shared data structures (for n threads):

```
boolean choosing[n] = {false, ...};
int number[n] = {0, ...};
int me = pthread_self();
```

- Notation:
  - (a,b) < (c,d) if (a<c) || ((a==c) & (b<d))
  - max(a_0, ..., a_{n-1}) = largest value in {a_0, ..., a_{n-1}}

Bakery Algorithm III

```
while(1) {
    choosing[me] = true;
    number[me] = max(number[0], ..., number[n-1]) + 1;
    choosing[me] = false;
    for (them = 0 ; them < n ; them++) {
        while(choosing[them]) ;
        while((number[them] != 0) &&
                (number[them], them) < (number[me], me)) ;
    }
    critical_section();
    number[me] = 0;
    additional_processing();
}
```

...so wait, lets get this straight...

- I have to have two arrays of the size of the *maximum* number of threads for *every* CS???

- Hardware, please come save us!

  1) Disable interrupts while in critical sections
     - Prevents preemption!
     - Should user-level processes be able to do this?
     - Work on multiprocessors?

  2) *atomic* instructions
     - Prevent preemption while executing instruction

Test & Set

- Functionally identical to

```
boolean test_and_set(boolean *memory_location)
{
    boolean b = *memory_location;
    *memory_location = true;

    return b;
}
```

- But all carried out *atomically!*

Mutual Exclusion via Test & Set

```
while(1) {
    while(test_and_set(&lock)) ;
    critical_section();
    lock = false;

    normal_processing();
}
```

- lock shared across threads, initially set to false
- *Problems with this solution??? (4 criteria)*

Compare & Swap (cas)

```
boolean cas(int *mem, int val, int newval) {
     if (*mem != val) return false;
     *mem = newval;
     return true;
}    /* all of this is atomic! */
```

```
     boolean done = false;
     do {
          int val = lock;
          if (!val) done = cas(&lock, val, true);
     } while(!done); /* spin while cs is held, or while our cas fails */
     critical_section();
     lock = false;
     normal_processing();
```

CAS Usage – Expanded

```
boolean done = false;
do {
    int val = lock;
    if (!val) {
        if (lock != val) done = false;
        else {
            lock  = true;
            done = true;
        }
    }
} while(!done); /* spin while cs is held, or while our cas fails */
critical_section();
lock = false;
normal_processing();
```

done atomically

Semaphores

- Higher-level mechanism for synchronization
- Semaphore, **s**, is an integer and a set of operations
- Conceptually, atomic operations are:
  - *wait(s)*: **while(s <= 0) ; s--;**
  - *signal(s)*: **s++;**

- As above implementation requires atomicity, how could it really be implemented?
  - What is the code for this???
  - Other option on uniprocessors?

Semaphores II

- Binary semaphore:
  - *mutex*
  - *s* = 1

```
semaphore_t mutex; // binary sem, s = 1
while(1) {
    normal_processing();
    wait(&mutex);
    critical_section();
    signal(&mutex);
}
```

- Counting semaphore:
  - *s* initialized to any integer value
  - Can initialize *s* to any positive value
  - What do positive values of *s* mean?

Semaphores III

- Higher-level sync primitives built using lower-level ones

atomic instuctions → Semaphores → Monitors

How can we implement semaphore's wait and signal using atomic instructions???

Semaphores IV

- Busy waiting:
  - **while(s <= 0) ;** s--;
  - Is this a good strategy if
    - Critical sections are long?
    - Critical sections are short?
    - “spin locks” are common (ubiquitous)!
      - *Where are they useful?*

Blocking Semaphores

- Blocking Semaphores: wait queue associated w/ semaphore
  - *Block* – place thd invoking *wait* onto semaphore's waiting queue
  - *Wakeup* – remove *one* thd from wait queue, place into runqueue
    - How do we decide *which* thread to remove?
- What do positive and negative values of *s* mean?
  - *Counting semaphore* implementation:

```
wait(s) {
    s--;
    if (s < 0) {
        waitq_enqueue(curr_thd);
        block&schedule();
    }
}
```

```
signal(s) {
    s++;
    if (s <= 0) {
        t = waitq_dequeue();
        wakeup&schedule(t);
    }
}
```

Some Issues with Semaphores

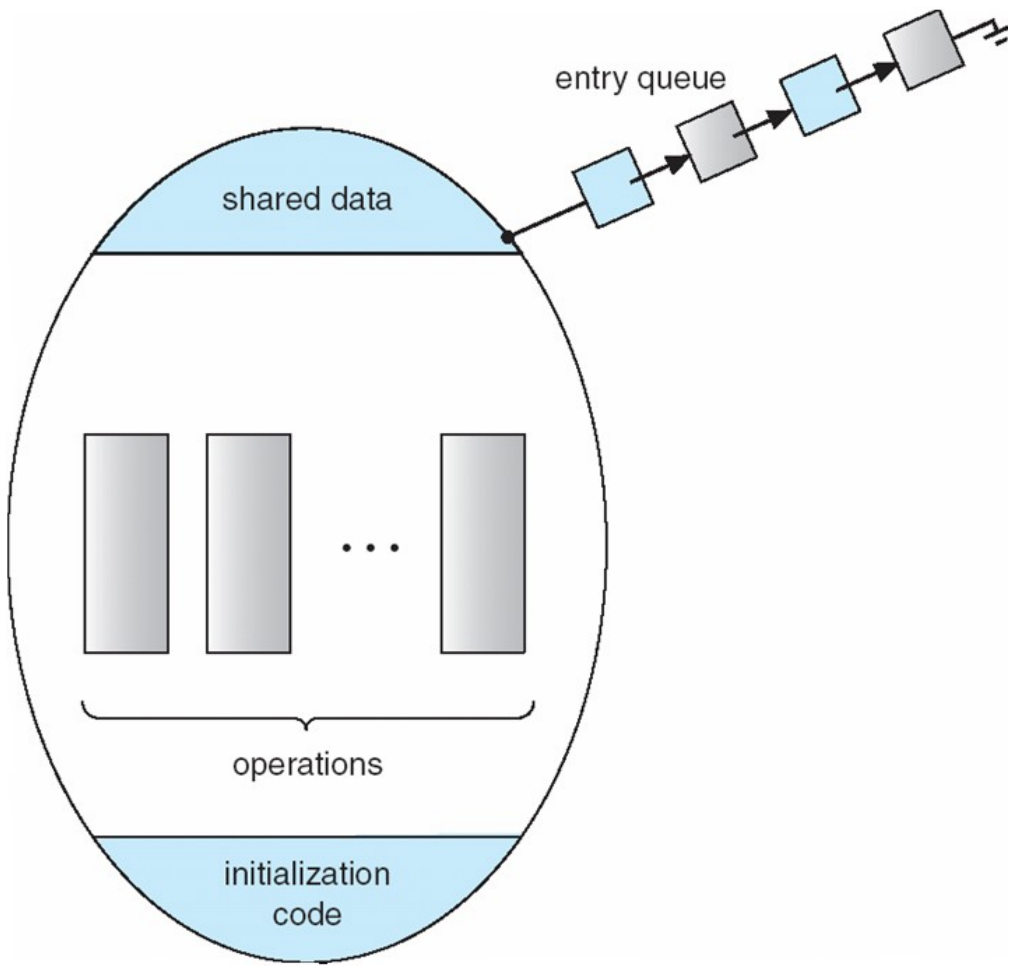
- Starvation
  - LIFO ordered wait-queues
  - What should the “correct” queueing policy be?
- Priority Inversion
  - Example
  - Must consider in real-time systems!
- Deadlocks
  - Example
  - Next lecture!

Monitors

- Higher-level abstraction that eases programming burden of thread synchronization
- Monitor includes set of data-structures *and* associated procedures (fns) to modify structures
- Fns can only access data-structures and arguments

- Mutual exclusion within monitor (via bin. semaphore)
  - functions are atomically executed
  - Results in data-structure mutual exclusion

Monitors II



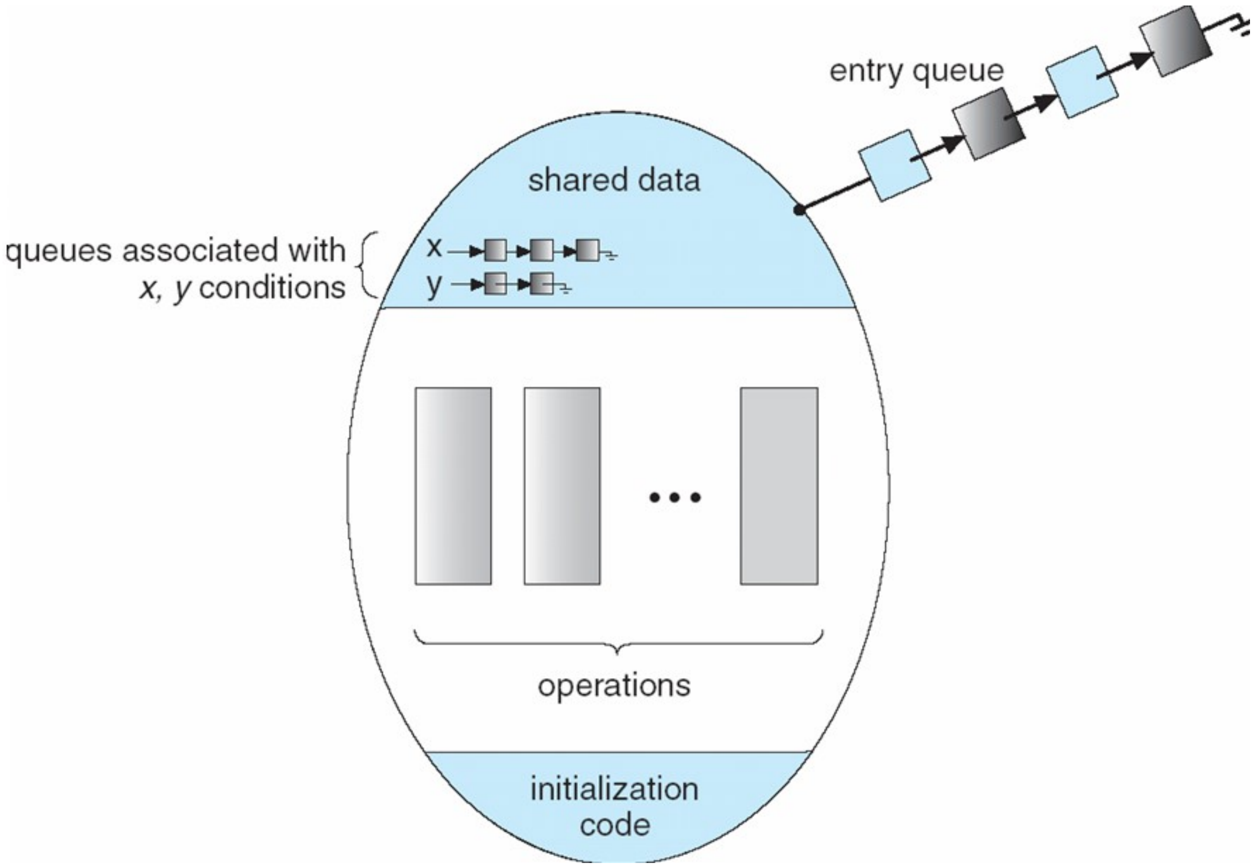
Monitors III

```
monitor name {
    //data structures...
    void fnA(...) {…}
    void fnB(...) {…}
    void initialization_fn(...) {…}
}
```

This look familiar to anyone?

- What if one of the functions wants to wait for some condition to happen...
  - e.g. wait for data to arrive in ring-buffer, user to press key,...
  - Condition variables – associated with specific monitor
    - wait_cv(cv) – block on cv queue, release monitor semaphore
    - signal_cv(cv) – unblock thd on cv queue, place in monitor q

Monitors IV



entry queue
shared data
queues associated with
x, y conditions
x
y
operations
initialization
code

Monitors V

- Example usage
  - Threads making blocking I/O

Problem???

```
bool IO_ready = false;
int nblked = 0;
mutex_t IO_mux;
cv_t IO_blklist;

wait_for_IO(void) {
    wait(IO_mux);
    if (!IO_ready) {
        nblked++;
        wait_cv(IO_blklist, IO_mux);
    }
    signal(IO_mux);
}

signal_IO(void) {
    wait(IO_mux);
    if (nblked) {
        signal_cv(IO_blklist);
        nblked--;
    }
    signal(IO_mux);
}
```

Monitors V

Important exercise:
Implement condition variables using mutexes!

- Example usage
  - Threads making blocking I/O

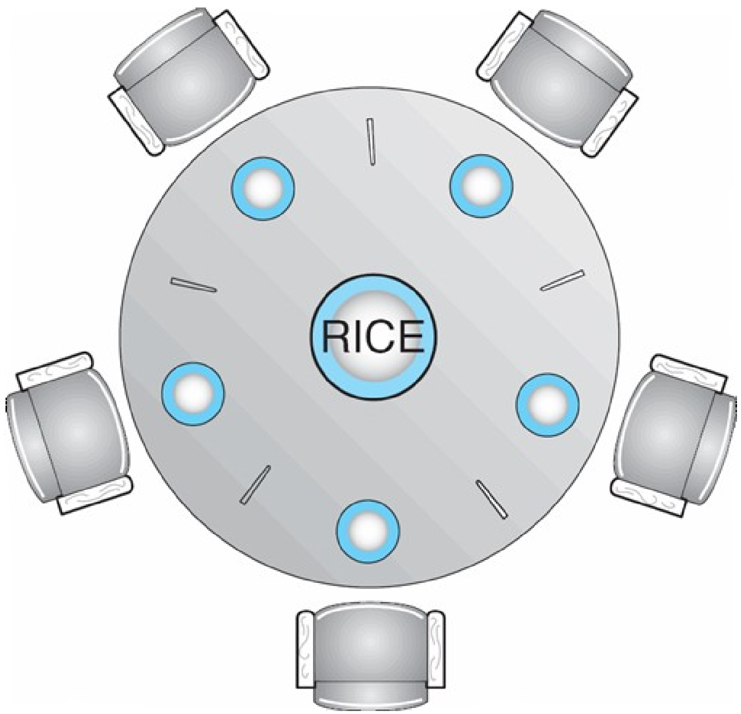
```
bool IO_ready = false;
mutex_t IO_mux;
cv_t IO_blklist;

wait_for_IO(void) {
     wait(&IO_mux);
     if (!IO_ready) {
          ...
     }
     signal(&IO_mux);
}
```

```
bool IO_ready = false;
mutex_t IO_mux;
cv_t IO_blklist;

wait_for_IO(void) {
     wait(IO_mux);
     while (!IO_ready) {
          ...
     }
     signal(IO_mux);
}
```

Dining Philosophers



Dining Philosophers II

- Each philosopher is in one of three states
  - thinking, hungry, or eating
- *hungry*: tries to acquire chopsticks, one at a time
- Only if both chopsticks are not used, can be they both be picked up
  - Transition into *eating* state
  - Later, philosopher places both chopsticks on table, transitions to *thinking* state

Dining Philosophers Solution I

```
mutex chopstick[5];
int right(int i) { return (i+1)%5; }
int left(int i) { return (i+4)%5; }

while (1) {
    wait(chopstick[i]);
    wait(chopstick[right(i)]);
    eat_and_be_jolly();
    signal(chopstick[i]);
    signal(chopstick[right(i)]);
    think_deep_thoughts();
}
```

Problems?

Dining Philosophers Solution II

```
while (1) {
    pickup(i);
    eat_and_be_jolly();
    put_down(i);
    think_deep_thoughts();
}
```

Dining Philosophers Solution III

```
monitor DP {
    enum {THINKING, HUNGRY, EATING} state[5];
    condition_var_t eat_time[5]; //condition → time to eat

    void pickup(int i) {
        state[i] = HUNGRY;
        time_to_eat?(i);
        if(state[i] != EATING)
            wait(eat_time[i]);
    }

    void put_down(int i) {
        state[i] = THINKING;
        time_to_eat?(right(i));
        time_to_eat?(left(i));
    }
```

```
void time_to_eat?(int i) {
    if ((state[right(i)] != EATING) &&
       (state[i] == HUNGRY &&
       (state[left(i)] != EATING)) {
         state[i] = EATING;
         signal(eat_time[i]);
    }
}
```

Remember: mutex held while executing all fns in the monitor!

Amdahl's law

- Parallelism speeds up multi-threaded computation
- ...but critical sections force mutual exclusion → sequential execution.
- Amdahl's law:
  - parallelization speedup limited by sequential code
  - Example:
    - 5% of your code's execution is in a critical section
    - infinite processors: maximum 20x speedup

Readers/Writers

- If a data-structure is *read* often, and *written* infrequently
  - Concurrent reads allowed!
  - Writes wait for *all* reads to complete before reading/writing the data

Readers/Writers II

```
semaphore mutex = 1, write_mut = 1;
int read_num = 0;
```

Reader:

```
wait(mutex);
read_num++;
if (read_num == 1)
    wait(write_mut);
signal(mutex);

read_data_struct();

wait(mutex);
read_num--;
if (read_num == 0)
    signal(write_mut);
signal(mutex);
```

Writer:

```
wait(write_mut);

read_data_struct();
write_data_struct();

signal(write_mut);
```

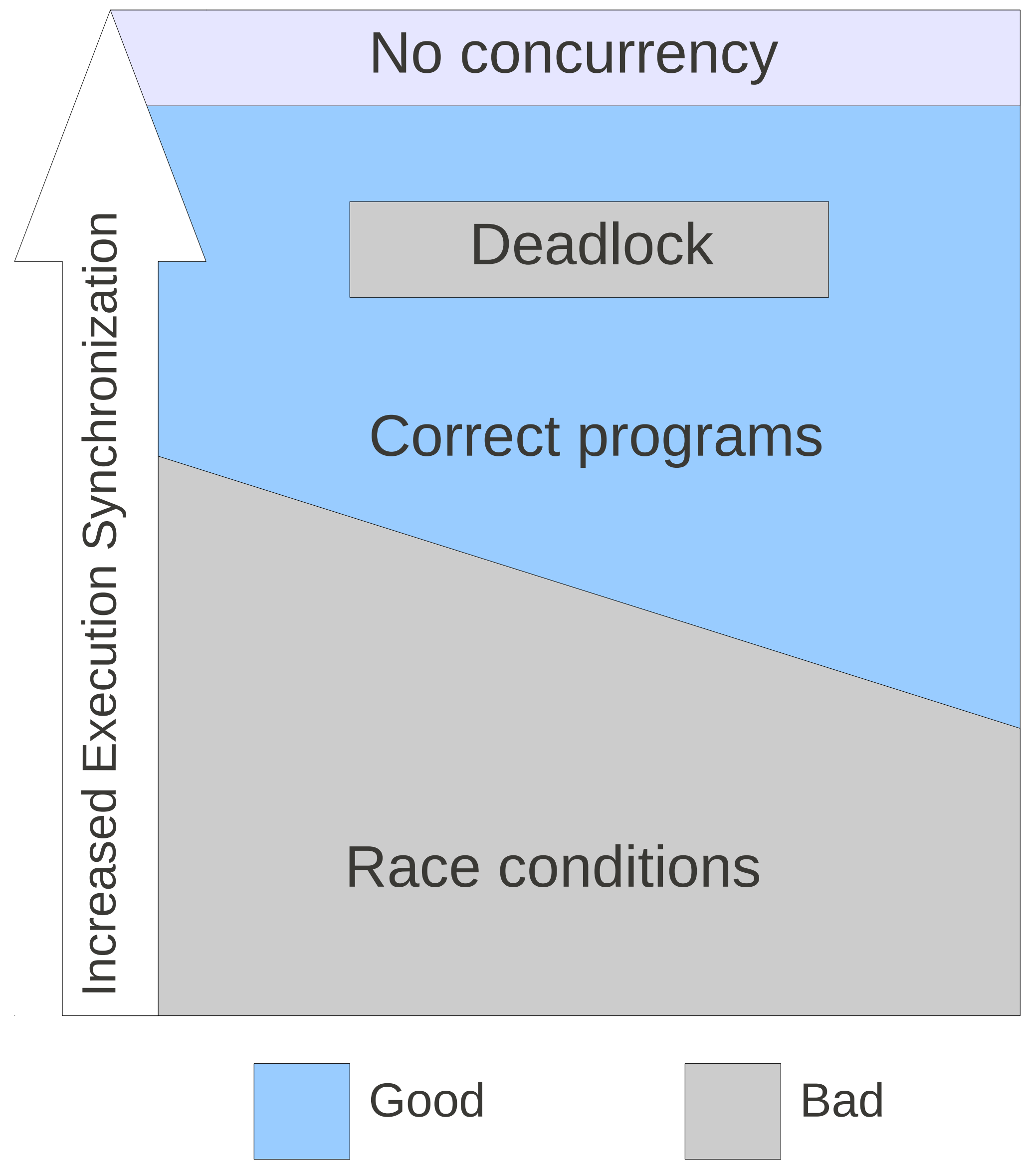
Downsides to this approach?

The View from Up High

- Why not just do this?

```
int main(void) {
     wait(&big_lock);
     compute();
     signal(&big_lock);
}
```

- Necessary evil



My Recent Errors

```
wake_me_later = 1;
thd->state = TASK_STATE_INTERRUPTABLE;
schedule(); //will place into wait queue
```

```
TIMER IRQ:

if (wake_me_later) {
    thd->state = TASK_STATE_RUNNABLE;
    wake_up(thd);
    wake_me_later = 0;
}
```