CS 170 Week 9

Winter 2026

https://sites.cs.ucsb.edu/~rich/class/cs170/

Announcement + Agenda

Announcement:

Final review on Monday

Agenda:

Any and all course work related questions/help

- Lab 3
- etc

Threading and Race Conditions

https://sites.cs.ucsb.edu/~rich/class/cs170/notes/RaceConditions/index.html

Does it have a race condition? If so, where?

If so, how would we fix it?

If not, why no race condition?

Semaphores, PThreads, and other fun

Pthread_t thread1;

pthread_create(&thread1, NULL, void* (*func), void* arg);

pthread_join(pthread_t thread, void** retval);

pthread_exit(void* retval);

Semaphores, PThreads, and other fun

Pthread_cond_t my_condition;

pthread_cond_init(&my_condition, NULL);

pthread_cond_signal(&my_condition);

pthread_cond_wait(&my_condition, pthread_mutex_t* lock);

Semaphores, PThreads, and other fun

pthread_mutex_t lock;

pthread_mutex_init(pthread_mutex_t* lock);

pthread_mutex_lock(pthread_mutex_t* lock);

pthread_mutex_unlock(pthread_mutex_t* lock);

Semaphores, PThreads, and other fun

Semaphores.

P() : Decrements counter. If counter becomes negative, block.

V(): Increment counter. If counter is <= 0, unblock one.

Practice Problem:

You have just been hired by Mother Nature to help her out with the chemical reaction to form water, which she doesn't seem to be able to get right due to synchronization problems. The trick is to get two *H* atoms and one *O* atom react together at the same time. For our problem, each atom is represented by a thread. Each *H* atom thread invokes a procedure called *hReady()* when it is ready to react, and each *O* atom thread invokes a procedure called *oReady()* when it is ready.

For this problem, your task it to write the code for the *hReady()* and *oReady()* procedures. The procedures must delay (block) the calling threads until there are at least two *H* atoms and one *O* atom present. When this is the case, then one (and only one) of the three threads must call the procedure *makeWater()* (which just prints out a debug message that water was made). After the *makeWater()* call, two instances of *hReady*() and one instance of *oReady*() should return.

Your solution must avoid starvation and busy-waiting! You can use any synchronization routine that we discussed in class. When using semaphores, you may assume that the semaphore implementation enforces FIFO order for wakeups – that is, the thread waiting longest in P() is the one woken up after a V() operation by another process.

Problem:

Problem Solving Steps:

1. Define the problem
2. What should the solution look like on high level?
3. What are the tools / algorithms available?
4. Implement
5. Check if your implementation matches #2… expected output

Practice Problem:

You have just been hired by Mother Nature to help her out with the chemical reaction to form water, which she doesn't seem to be able to get right due to synchronization problems. The trick is to get two *H* atoms and one *O* atom react together at the same time. For our problem, each atom is represented by a thread. Each *H* atom thread invokes a procedure called *hReady()* when it is ready to react, and each *O* atom thread invokes a procedure called *oReady()* when it is ready.

For this problem, your task it to write the code for the *hReady()* and *oReady()* procedures. The procedures must delay (block) the calling threads until there are at least two *H* atoms and one *O* atom present. When this is the case, then one (and only one) of the three threads must call the procedure *makeWater()* (which just prints out a debug message that water was made). After the *makeWater()* call, two instances of *hReady*() and one instance of *oReady*() should return.

Your solution must avoid starvation and busy-waiting! You can use any synchronization routine that we discussed in class. When using semaphores, you may assume that the semaphore implementation enforces FIFO order for wakeups – that is, the thread waiting longest in P() is the one woken up after a V() operation by another process.

Problem

Problem Solving Steps:

1. **Define the problem**

   Each atom is a thread. The function `makeWater()` must be called once every time 2 hydrogen and 1 oxygen atoms are available. This is a form of a producer / consumer problem. The consumer (caller of makeWater() waits for the input from the producer.

Problem

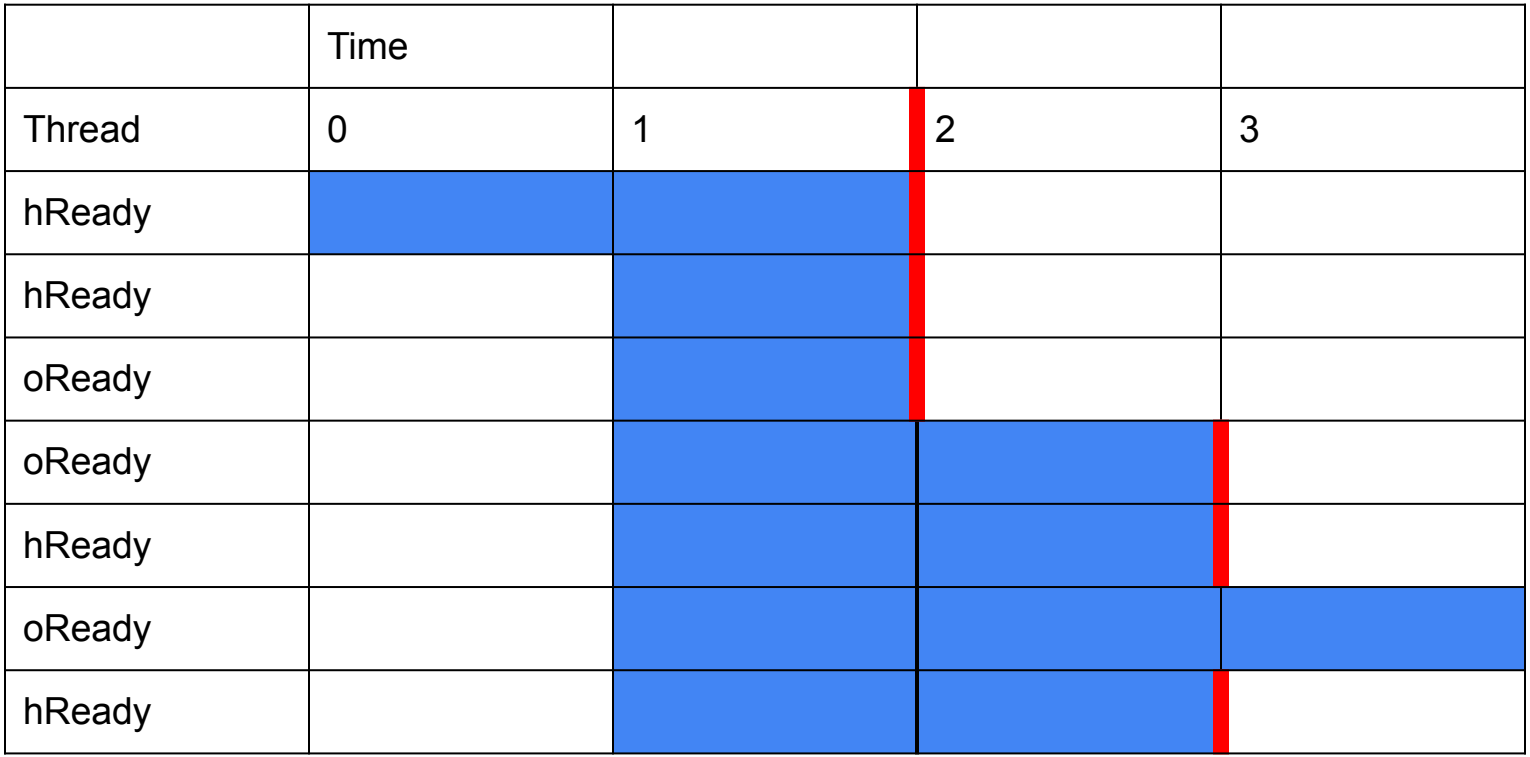
Problem Solving Steps:

1. Define the problem
2. **What should the solution look like on high level?**

   `makeWater()` should be called for every two hReady()'s and one oReady(). Each run in a different thread. *hReady() and oReady() must return only after being used to makeWater()*.

Problem

	Time			
Thread	0	1	2	3
hReady				
hReady				
oReady				
oReady				
hReady				
oReady				
hReady				



Problem

Problem Solving Steps:

1. Define the problem
2. What should the solution look like on high level?
3. **What are the tools and algorithms available?**
    a. Semaphores
    b. Locks
    c. Others??

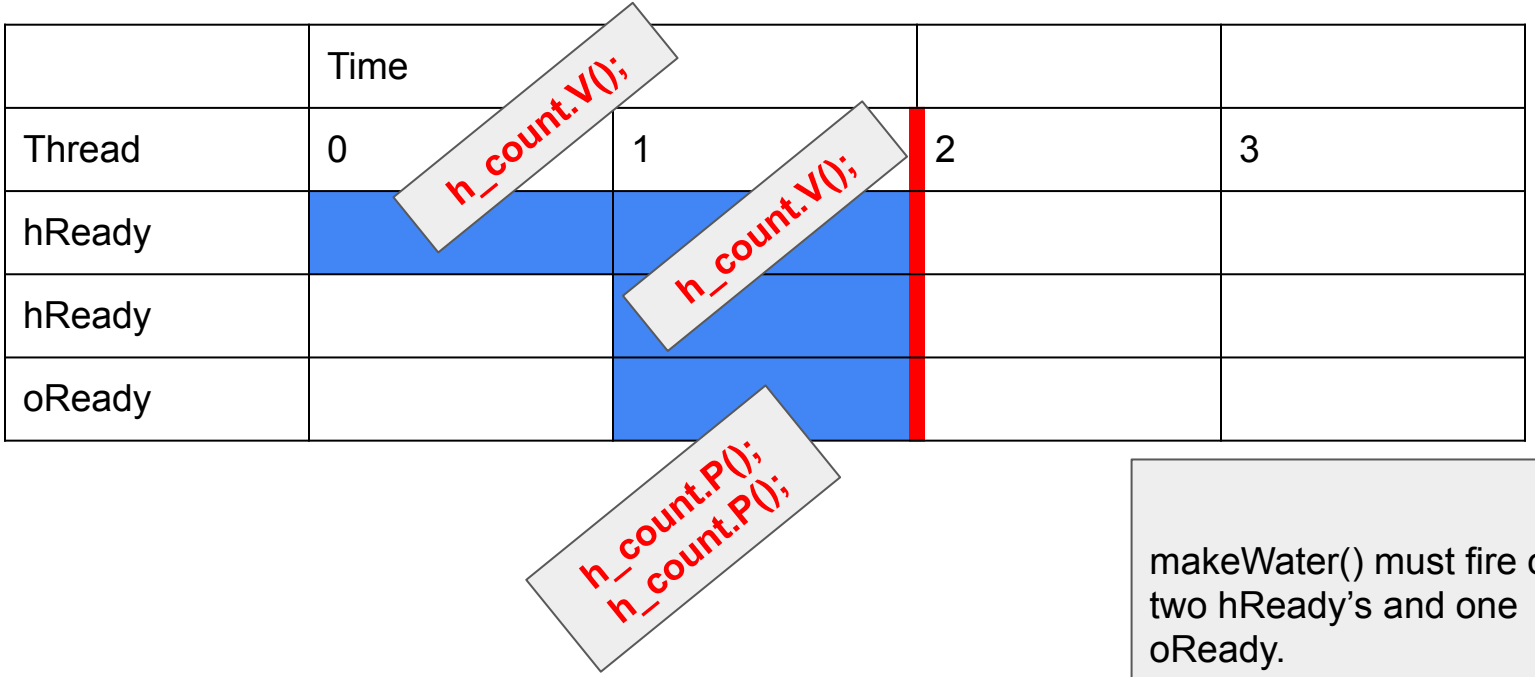
Problem

	Time			
Thread	0	1	2	3
hReady				
hReady				
oReady				

makeWater() must fire on two hReady's and one oReady.

So, let's pick oReady to call makeWater. It should wait (P) for two hydrogen vals.

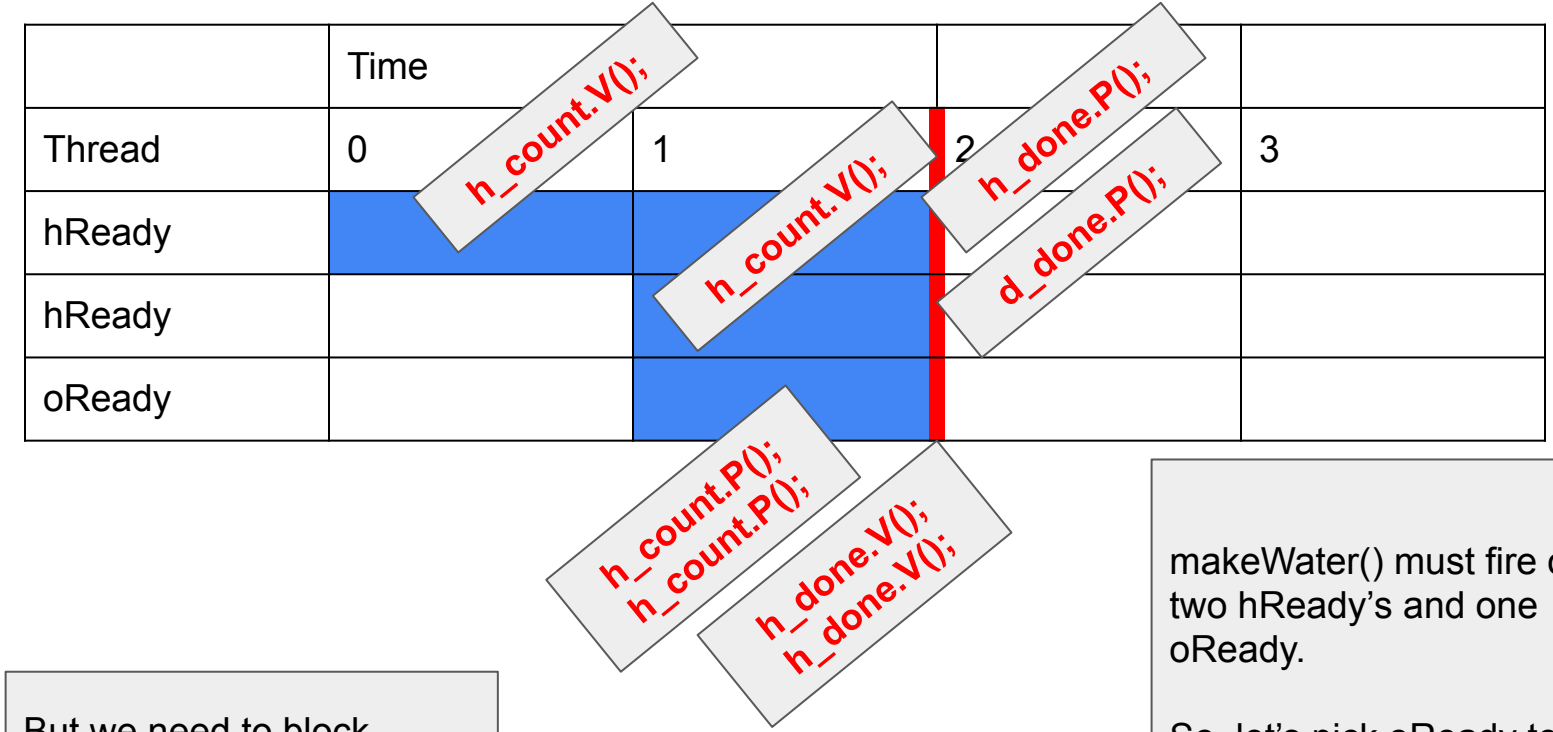
Problem



makeWater() must fire on two hReady's and one oReady.

So, let's pick oReady to call makeWater. It should wait (P) for two hydrogen vals.

Problem



But we need to block hReady until makeWater is fired as well.

makeWater() must fire on two hReady's and one oReady.

So, let's pick oReady to call makeWater. It should wait (P) for two hydrogen vals.

Problem

Problem Solving Steps:

1. Define the problem
2. What should the solution look like on high level?
3. What are the tools / algorithms available?
4. **Implement**
5. **Check if your implementation matches #2… expected output**

Problem

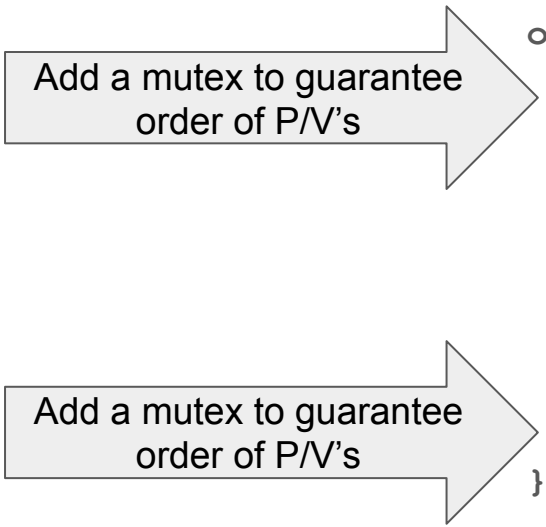
```
Semaphore mutex = 1;
Semaphore h_count = 0;
Semaphore h_done = 0;

h_ready() {
  V(h_count);
  P(h_done);
}

o_ready() {
  P(mutex);

  P(h_count);
  P(h_count);
  makeWater();
  V(h_done);
  V(h_done);

  V(mutex);
}
```



Memory Practice

KOS:

1. 1 MB of total memory
2. 512 bytes per page.

What would be the breakdown of bits for a memory address?

____*frame bits*_____ | _____*byte offset bits*_____

Questions