Discussion 3.5: Condition Variables, Futex, RW Locks

February 20, 2026

Contents

1 Condition Variable

Condition variables are synchronization variables that let a thread efficiently wait for a change to a shared state. They store a queue of threads such that the waiting threads are allowed to sleep inside the critical section which is in contrast to other synchronization variables like semaphores.

Condition variables are used in conjunction with a lock which together form a **monitor**.

Condition variables provide the following operations. For all these methods, the thread calling them must be holding the lock.

Wait
Atomically releases lock and suspends execution of calling thread.

Signal
Wake up the next waiting thread in the queue.

Broadcast
Wake up all waiting threads in the queue.

Infinite Synchronized Buffer

Consider an infinite synchronized buffer problem of vending machines where there's a producer and consumer. "Infinite" refers to the fact that the machine has no limit on how much coke it can hold.

```
Lock bufferLock;
ConditionVar bufferCV;

Producer() {
  bufferLock.acquire();
  put 1 coke in machine
  bufferCV.signal();
  bufferLock.release();
}

Consumer() {
  bufferLock.acquire();
  while (machine is empty)
    bufferCV.wait(bufferLock);
  take 1 coke out
  bufferLock.release();
}
```

By using condition variables, we can avoid busy waiting inside a critical section.

Semantics

Different semantics define signal and wait differently.

When a thread is signaled using **Hoare** semantics, the ownership of the lock is immediately transferred to the waiting thread. Furthermore, the execution of this thread resumes immediately. After this thread releases the lock, ownership of the lock is transferred back to the signaling thread. As a result, signaling in Hoare semantics can be thought of as an atomic operation.

On the other hand, **Mesa** semantics makes no guarantees about the execution order when a thread is signaled.

This leads to a subtle but important difference in the code. Using the same bounded buffer example as before,

Hoare

```
if (machine is empty)
  bufferCV.wait(bufferLock);
take 1 coke out
```

Mesa

```
while (machine is empty)
  bufferCV.wait(bufferLock);
take 1 coke out
```

We can use a if statement for Hoare semantics because we're guaranteed an execution order between the waiting and signaling thread. However, that is not the case for Mesa semantics, meaning between the time the waiting thread is signalled and it actually executes, some other thread could have run in between and rendered the condition false again, so a while loop is necessary.

1.1 Condition Check

1. Will this program compile/run?

```c
pthread_mutex_t lock;
pthread_cond_t cv;
int hello = 0;
void print_hello() {
  hello += 1;
  printf("First line (hello=%d)\n", hello);
  pthread_cond_signal(&cv);
  pthread_exit(0);
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, (void *) &print_hello, NULL);
  while (hello < 1)
    pthread_cond_wait(&cv, &lock);
  printf("Second line (hello=%d)\n", hello);
}
```

cv_hello.c

2. Fill in the blanks such that the program always prints "Yeet Haw". Assume the system behaves with Mesa semantics.

```c
int ben = 0;
_____________________;
_____________________;

void *helper(void *arg) {
  _____________________;
  ben += 1;
  _____________________;
  _____________________;
  pthread_exit(NULL);
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, NULL);
```

```
  pthread_yield();
  ________________________;
  ________________________;
  ________________________;
  if (ben == 1)
    printf("Yeet Haw\n");
  else
    printf("Yee Howdy\n");
  ________________________;
}
```

cv_howdy.c

1.2 Office Hours Queue

Suppose we want to use condition variables to control access to a CS162 (digital) office hours room for three types of people: students, TAs, and professors. A person can attempt to enter the room (or will wait outside until their condition is met), and after entering the room they can then exit the room. The follow are each type's conditions:

- Suppose professors get easily distracted and so they need solitude, with no other students, TAs, or professors in the room, in order to enter the room.
- TAs don't care about students being inside and will wait if there is a professor inside, but there can only be up to 9 TAs inside (any more would clearly be imposters from CS161 or CS186).
- Students don't care about other students or TAs being in the room, but will wait if there is a professor.
- Students and TAs are polite to professors, and will let a waiting professor in first.

To summarize the constraints,

- Professor must wait if anyone else is in the room
- TA must wait if there are already 9 TAs in the room
- TA must wait if there is a professor in the room or waiting outside
- Students must wait if there is a professor in the room or waiting outside

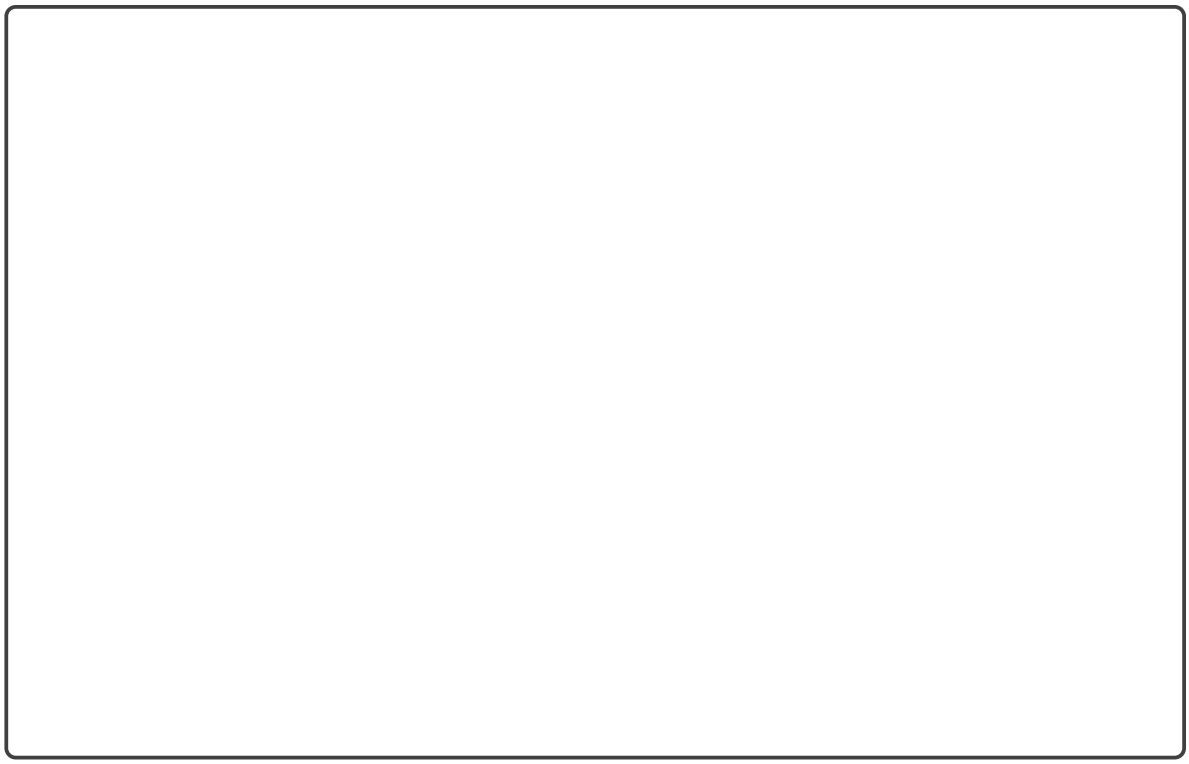
```
typedef struct room {
  pthread_mutex_t lock;
  pthread_cond_t student_cv;
  int waiting_students, active_students;
  pthread_cond_t ta_cv, prof_cv;
  int waiting_tas, active_tas;
  int waiting_profs, active_profs;
} room_t;

enter_room(room_t* room, int mode) {
  pthread_mutex_lock(&room->lock);
  if (mode == 0) {
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    room->active_students++;
  } else if (mode == 1) {
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    room->active_tas++;
  } else {
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    room->active_profs++;
  }
  pthread_mutex_unlock(&room->lock);
}

exit_room(room_t* room, int mode) {
  pthread_mutex_lock(&room->lock);
  if (mode == 0) {
    room->active_students--;
    ________________________;
    ________________________;
    ________________________;
  } else if (mode == 1) {
    room->active_tas--;
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    ________________________;
  } else {
```

```
    room->active_profs--;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
  }
  pthread_mutex_unlock(&room->lock);
}
```

oh.c

1. Fill in `enter_room`.

2. Fill in `exit_room`.



2 Futex

Normally, a user program cannot put a thread to sleep without entering the kernel. However, kernel-based locks like `pthread_mutex_lock` are slow when there is no contention, since they require a syscall every time. A **futex** (fast userspace mutex) solves this by keeping the fast path entirely in user space, and only entering the kernel when a thread actually needs to sleep or wake another thread.

```
futex(int *addr, int operation, int val)
```

addr
Pointer to an integer in user space (the futex word).

FUTEX_WAIT
Atomically checks if `*addr == val`. If so, puts the calling thread to sleep. If not, returns immediately.

FUTEX_WAKE
Wakes up to `val` threads currently sleeping on `addr`.

The general pattern when using a `FUTEX_WAIT` is

```
while (*addr != SOMETHING)
    futex(addr, FUTEX_WAIT, SOMETHING);
```

The key insight is that `FUTEX_WAIT` checks the value *atomically* with putting the thread to sleep. This avoids a race condition where the value changes between the check and the sleep.

Futex Lock

We can build a lock using a futex and atomic operations. The lock is represented by a single integer with three states:

```
typedef enum { UNLOCKED, LOCKED, CONTESTED } Lock;
Lock mylock = UNLOCKED;
```

UNLOCKED
The lock is free.

LOCKED
The lock is held, and no other thread is waiting.

CONTESTED
The lock is held, and at least one other thread is waiting.

The `CONTESTED` state is an optimization: it tells the releasing thread that it needs to call `futex(FUTEX_WAKE)` to wake a waiter. If the lock is merely `LOCKED`, the releasing thread can avoid the syscall entirely.

```
1  acquire(Lock *thelock) {
2      if (compare&swap(thelock, UNLOCKED, LOCKED)) // If unlocked, grab lock!
3          return;
4
5      while (swap(thelock, CONTESTED) != UNLOCKED) // Keep trying to grab lock, sleep in futex
6          futex(thelock, FUTEX_WAIT, CONTESTED);  // Sleep unless someone releases here!
7  }
8
9  release(Lock *thelock) {
10     if (swap(thelock, UNLOCKED) == CONTESTED) // If someone sleeping,
11         futex(thelock, FUTEX_WAKE, 1); // wake someone else up
12 }
```

futex_lock.c

2.1 Barrier

A common synchronization primitive used is a barrier. A barrier will block all threads from passing it until every thread has gotten to the location. Think about a barrier placed before F1 drivers to ensure they all reach the starting line before the race begins.

We want to implement a barrier which does not busy wait and is able to be used multiple times.

A barrier can be described by the following pseudo code:

```
structure Barrier {
  count        // number of threads that must arrive
  waiting      // counter of how many threads have arrived
}

procedure barrier_init(B, num_threads):
  B.count = num_threads
  B.waiting = 0

procedure barrier_wait(B):
  B.waiting += 1
  if B.waiting == B.count:          // last thread arrives
    B.waiting = 0                   // reset for next use
    wake_waiting_threads()          // wake all other threads
  else:
    wait_for_all_threads()          // wait until everyone gets here
```

For reference, here is the C code for compare and swap

```
bool CAS(int *mem, int cmp, int swp) {
  if (*mem == cmp) {
    *mem = swp;
    return true;
  } else {
    return false;
  }
}
```

Here is some C starter code to get you started on implementing barrier.

```
struct barrier {
  int threads;
  int waiting;
  bool direction;
}


void barrier_init(struct barrier *b, int threads) {
  b->threads = threads;
  b->waiting = 0;
  b->direction = 0;
}
```

Implement the method `barrier_wait` on the next page!

```
void barrier_wait(struct barrier *b) {
  int wanted_dir = _______[A]_______;
  int w;

  do {
    w = _______[B]_______;
  } while (_______[C]_______)

  if (_______[D]_______) {
    b->waiting = 0;
    b->direction = wanted_dir;
    futex(_______[E]_______);
  } else {
    while (_______[F]_______)
      futex(_______[G]_______);
  }
}
```

Fill in the blanks in the `barrier_wait` method above to complete the implementation.

[A]

[B]

[C]

[D]

[E]

[F]

[G]

3 Pintoast

Diana is the owner of Pintoast Bakery. There are multiple chefs that bake customers' orders one at a time. A customer will first enter the bakery and place their order. After a chef bakes their order, the customer will pay and leave the bakery, allowing another customer to come in.

1. Chefs should bake orders in a first-come first-serve manner.
2. Chefs should not block other cooks or customers while baking an order.
3. A bakery can only hold up to 100 customers at any given time.

```
typedef struct bakery {
    int capacity;
    struct list orders;
    struct condition customerWait;
    struct condition customerDone;
    struct condition chefDone;
    struct lock orders_lock;
} bakery_t;

typedef struct order {
   bool cooked;
   struct list_elem elem;
   /* Other fields hidden */
} order_t;

/* Assume bake, enter, and leave always execute successfully. */

void bake(order_t* order) { /* Implementation details hidden */ };
void enter(bakery_t* bakery) { /* Implementation details hidden */ };
void leave(bakery_t* bakery) { /* Implementation details hidden */ };
```

Implement the following program to help Diana set up her Pintoast bakery in Pintos.

```
/* Assume that all members of bakery are properly initialized. Assume that a chef will
continuously call this function while the bakery is open. */

void chef(bakery_t* bakery) {
    _______[A]_______;

    while (_______[B]_______)
        cond_wait(&bakery->customerWait, &bakery->orders_lock);

    struct list_elem* e = _______[C]_______;
    order_t* order = list_entry(e, order_t, elem);
    _______[D]________;

    bake(order);

    _______[E]x4________;
}
```

The notation `_____[Y]xN_____` indicates a response that can be at most `N` lines long and should be written in the answer box for Part Y.

3.1 Chef Implementation

[A] (max 1 line)

[B] (max 1 line)

[C] (max 1 line)

[D] (max 1 line)

[E] (max 4 lines)

3.2 Customer Implementation

```
/* Assume that bakery->capacity = 100 before any customer enters the bakery.
If there are already 100 customers in the bakery, a new customer should wait
for someone to leave before entering the bakery. */

void customer(bakery_t* bakery, order_t* order) {
    _______[A]_______
    while (_______[B]_______)
        _______[C]_______

    _______[D]_______
    enter(bakery);

    list_push_back(&bakery->orders, &order->elem);
    _______[E]_______

    while (_______[F]_______)
         _______[G]_______

    leave(bakery);
    _______[H]x3_______
}
```

[A] (max 1 line)

[B] (max 1 line)

[C] (max 1 line)

[D] (max 1 line)

[E] (max 1 line)

[F] (max 1 line)

[G] (max 1 line)

[H] (max 3 lines)