Lecture 22: Semaphores and Conditional Variables

CS 105

Problems with Locks

- Problem 1: Correct Synchronization with Locks is Hard
- Problem 2: Locks are Slow
  - threads that fail to acquire a lock on the first attempt must "spin", which wastes CPU cycles
    - replace no-op with yield()
  - threads get scheduled and de-scheduled while the lock is still locked
    - need a better synchronization primitive

Semaphores

- A semaphore s is a stateful synchronization primitive comprised of:
  - a value n (non-negative integer)
  - a lock
  - a queue

- Interface:
  - **init(sem_t * s, unsigned int val)**
  - **P(sem_t * s):** If s is nonzero, the P decrements s and returns immediately. If s is zero, then adds the thread to queue(s); after restarting, the P operation decrements s and returns.
  - **V(sem_t * s):** Increments s by 1. If there are any threads in queue(s), then V restarts exactly one of these threads, which then completes the P operation.

Semantics of P and V

- P(sem_t * s)
  - block (**suspend thread**) until value n > 0
  - when n > 0, decrement n by one

```
P(sem_t * s){
  while(s->n == 0){
    ;
  }
  s->n -= 1
}
```

- V(sem_t * s)
  - increment value n b 1
  - **resume a thread waiting on s (if any)**

```
V(sem_t * s){
  s->n += 1
}
```

Why P and V?

- Edsger Dijkstra was from the Netherlands
  - P comes from the Dutch word proberen (to test)
  - V comes from the Dutch word verhogen (to increment)

- Better names than the alternatives
  - decrement_or_if_value_is_zero_block_then_decrement_after_waking
  - increment_and_wake_a_waiting_process_if_any

Binary Semaphore (aka mutex)

- A binary semaphore is a semaphore whose value is always 0 or 1
- Used for mutual exclusion---it's a more efficient lock!

```
sem_t s
init(&s, 1)
```

T1

```
P(&s)
CriticalSection()
V(&s)
```

T2

```
P(&s)
CriticalSection()
V(&s)
```

Example: Shared counter

```
volatile long cnt = 0;
```

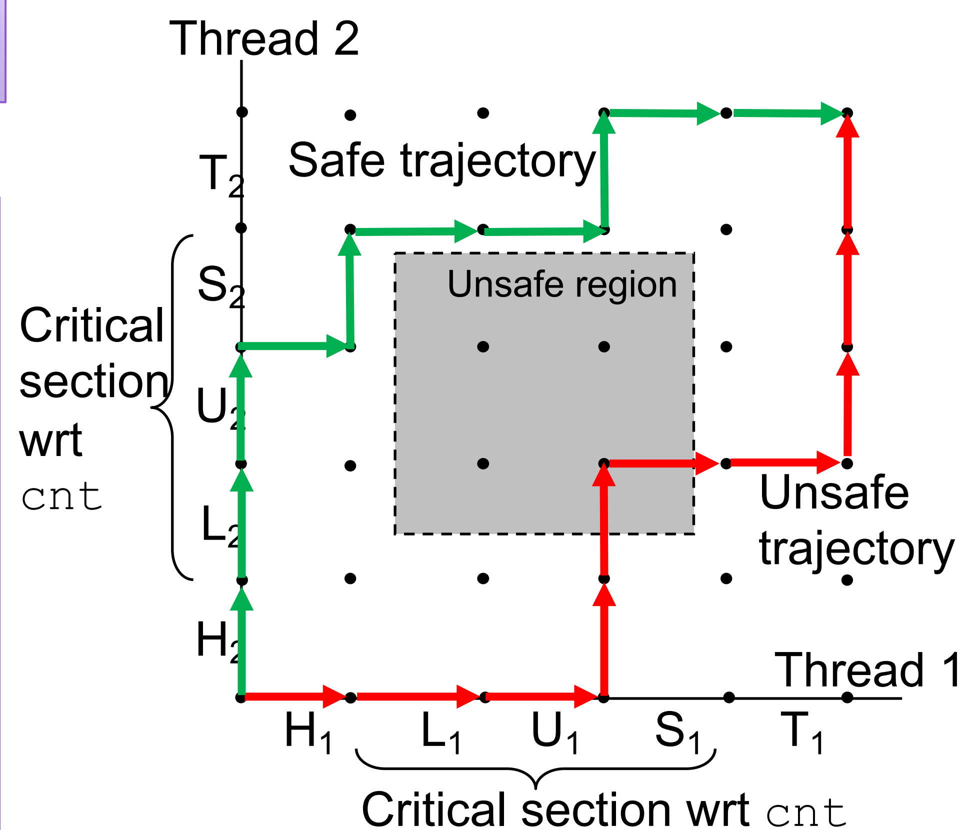
```
/* Thread routine */
void *thread(void *vargp)
{
  long niters = *((long *)vargp);

  long i;
  for (i = 0; i < niters; i++){

    cnt++;

  }

  return NULL;
}
```



Example: Shared counter

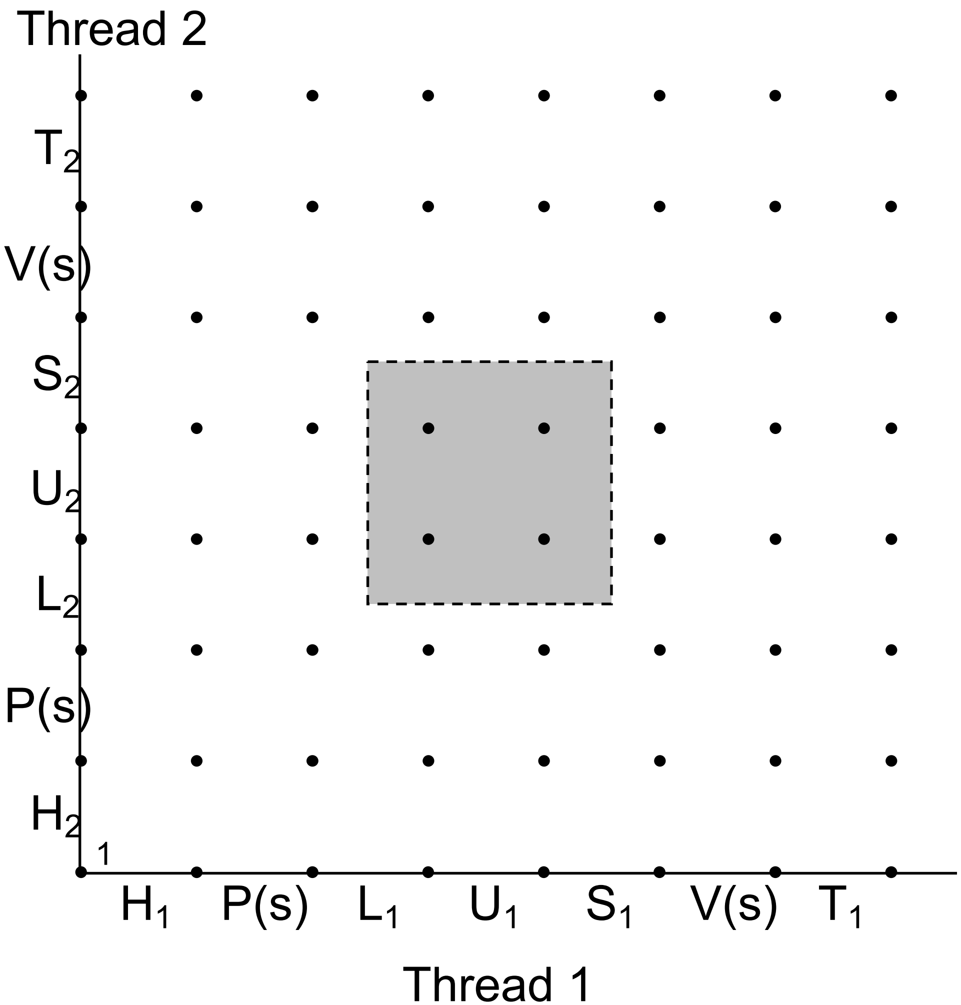
```
volatile long cnt = 0;
sem_t s;
```

```
sem_init(&s, 1);
```

```
/* Thread routine */
void *thread(void *vargp)
{
  long niters = *((long *)vargp);

  long i;
  for (i = 0; i < niters; i++){
    P(&s)
    cnt++;
    V(&s)
  }

  return NULL;
}
```



Exercise 1: Semaphores

- What would be the value in the semaphore at the four bad points?

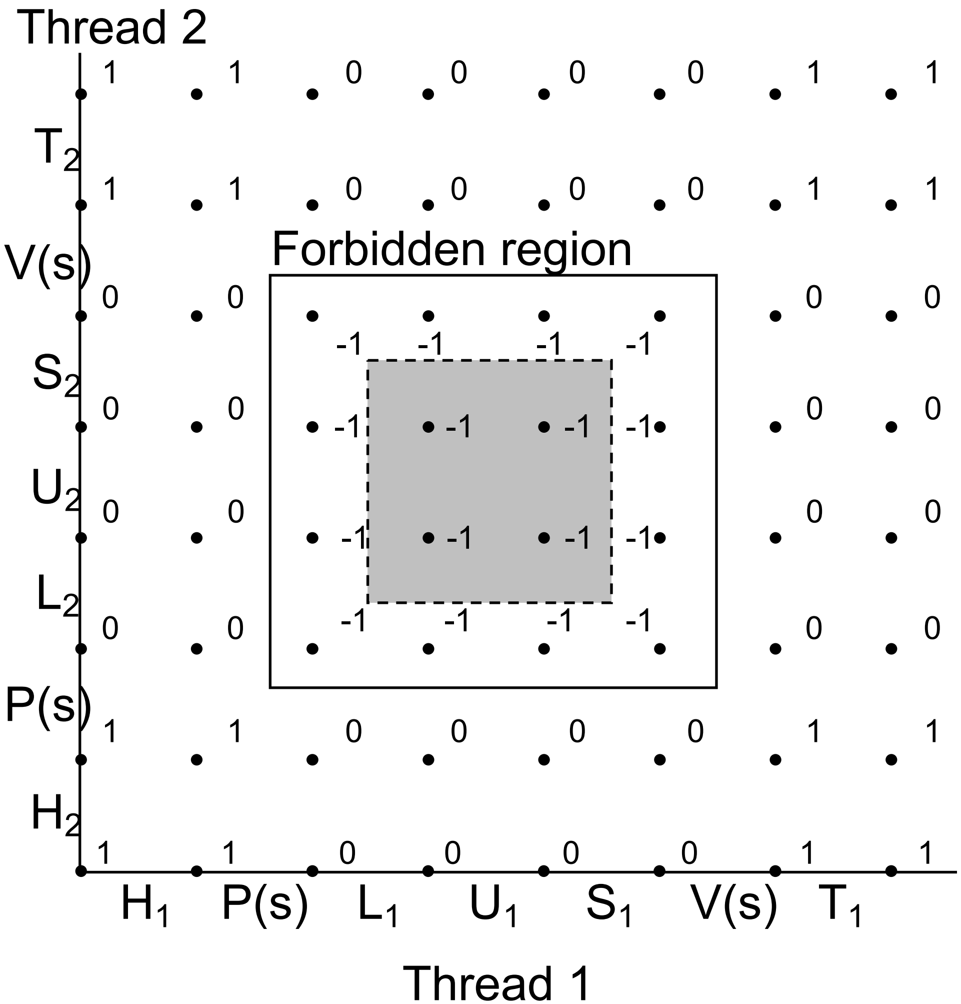
```
volatile long cnt = 0;
sem_t s;
```

```
sem_init(&s, 1);
```

```
/* Thread routine */
void *thread(void *vargp)
{
  long niters = *((long *)vargp);

  long i;
  for (i = 0; i < niters; i++){
    P(&s)
    cnt++;
    V(&s)
  }

  return NULL;
}
```



Example: Synchronization Barrier

- With data parallel programming, a computation proceeds in parallel, with each thread operating on a different section of the data. Once all threads have completed, they can safely use each others results.
  - MapReduce is an example of this!

- To do this safely, we need a way to check whether all n threads have completed.

```
volatile int results = 0;
volatile int done_count = 0;
sem_t count_mutex;
sem_init(&count_mutex, 1)
sem_t barrier;
sem_init(&barrier, 0)
```

```
void *thread(void *args){
    parallel_computation(args);

    P(&count_mutex);
    done_count++;
    V(&count_mutex);

    if(done_count == n){
        V(&barrier);
    }
    P(&barrier);
    V(&barrier);
    use_results();
}
```

Counting Semaphores

- A semaphore with a value that goes above 1 is called a counting semaphore
- Provide a more flexible primitive for mediating access to shared resources

Example: Bounded Buffers

finite capacity (e.g. 20 loaves)
implemented as a queue

Threads A: produce loaves of bread and put them in the queue

Threads B: consume loaves by taking them off the queue

Example: Bounded Buffers

finite capacity (e.g. 20 loaves)
implemented as a queue

Separation of concerns:
1. How do you implement a bounded buffer?
2. How do you synchronize concurrent access to a bounded buffer?

Threads A: produce loaves of bread and put them in the queue

Threads B: consume loaves by taking them off the queue

Example: Bounded Buffers

	0	1	2	3	4	5	(n = 6)
b	3			2	4	1	
		rear		front			

Values wrap around!!

```
typedef struct {
    int *b;        // ptr to buffer containing the queue
    int n;         // length of array (max # slots)
    int front;     // index of first element, 0 <= front < n
    int rear;      // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

```
void init(bbuf_t * ptr, int n){
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->front = 0;
  ptr->rear = 0;
}
```

```
void put(bbuf_t * ptr, int val){
  ptr->b[ptr->rear]= val;
  ptr->rear= ((ptr->rear)+1)%(ptr->n);
}
```

```
int get(bbuf_t * ptr){
  int val= ptr->b[ptr->front];
  ptr->front= ((ptr->front)+1)%(ptr->n);
  return val;
}
```

Exercise 2: What can go wrong?

Example: Bounded Buffers

	0	1	2	3	4	5	(n = 6)
b	3	2			4	1	
			rear		front		

```
typedef struct {
    int *b;
    int n;
    int front;
    int rear;
    sem_t mutex;
    sem_t slots;
    sem_t items;
} bbuf_t
```

```
void put(bbuf_t * ptr, int val){
    P(&(ptr->slots))
    P(&(ptr->mutex))
    ptr->b[ptr->rear]= val;
    ptr->rear= ((ptr->rear)+1)%(ptr->n);
    V(&(ptr->mutex))
    V(&(ptr->items))
}
```

```
void init(bbuf_t * ptr, int n){
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->front = 0;
  ptr->rear = 0;
  sem_init(&mutex, 1);
  sem_init(&slots, n);
  sem_init(&items, 0);
}
```

```
int get(bbuf_t * ptr){
    P(&(ptr->items))
    P(&(ptr->mutex))
    int val= ptr->b[ptr->front];
    ptr->front= ((ptr->front)+1)%(ptr->n);
    V(&(ptr->mutex))
    V(&(ptr->slots))
    return val;
}
```

Exercise 3: Readers/Writers

- Consider a collection of concurrent threads that have access to a shared object
- Some threads are readers, some threads are writers
  - a unlimited number of readers can access the object at same time
  - a writer must have exclusive access to the object

```
// global variables
int num_readers = 0;
sem_t num_lock;
sem_t ojb_lock;
```

```
int reader(void *shared){
    P(&num_lock);
    num_readers++;
    if(num_readers == 1)
        P(&obj_lock);
    V(&num_lock);
    int x =  read(shared);
    P(&num_lock);
    num_readers--;
    if(num_readers == 0)
        V(&obj_lock);
    V(&num_lock);
    return x
}
```

```
void init(){
    sem_init(&num_lock, 1);
    sem_init(&ojb_lock, 1);
}
```

```
void writer(void *shared, int val){
    P(&ojb_lock);
    write(shared, val);
    V(&ojb_lock);
}
```

Programming with Semaphores

C

- Semaphore type:

```
sem_t
```

- Initialization:

```
int sem_init(sem_t* s,
             int pshared,
             unsigned value)
```

- P

```
sem_wait(sem_t * s)
```

- V

```
sem_post(sem_t * s)
```

Python

- Semaphore type:

```
class Semaphore
```

- Initialization:

```
s = Semaphore(value)
```

- P

```
s.acquire()
```

- V

```
s.release()
```

Limitations of Semaphores

- semaphores are a very spartan mechanism
  - they are simple, and have few features
  - more designed for proofs than synchronization
- they lack many practical synchronization features
  - it is easy to deadlock with semaphores
  - one cannot check the lock without blocking
- strange interactions with OS scheduling (priority inheritance)

Condition Variables

- A condition variable cv is a stateless synchronization primitive that is used in combination with locks (mutexes)
  - condition variables allow threads to efficiently wait for a change to the shared state protected by the lock
  - a condition variable is comprised of a waitlist

- Interface:
  - **wait(CV * cv, Lock * lock):** Atomically releases the lock, suspends execution of the calling thread, and places that thread on cv's waitlist; after the thread is awoken, it re-acquires the lock before wait returns
  - **signal(CV * cv):** takes one thread off of cv's waitlist and marks it as eligible to run. (No-op if waitlist is empty.)
  - **broadcast(CV * cv):** takes all threads off cv's waitlist and marks them as eligible to run. (No-op if waitlist is empty.)

Using Condition Variables

1. Add a lock. Each shared value needs a lock to enforce mutually exclusive access to the shared value.
2. Add code to acquire and release the lock. All code access the shared value must hold the objects lock.
3. Identify and add condition variables. A good rule of thumb is to add a condition variable for each situation in a function must wait for.
4. Add loops to wait. Threads might not be scheduled immediately after they are eligible to run. Even if a condition was true when signal/broadcast was called, it might not be true when a thread resumes execution.

Example: Synchronization Barrier

- With data parallel programming, a computation proceeds in parallel, with each thread operating on a different section of the data. Once all threads have completed, they can safely use each others results.
  - MapReduce is an example of this!

- To do this safely, we need a way to check whether all n threads have completed.

```
int done_count = 0;
Lock lock;
CV all_done;
```

```
/* Thread routine */
void *thread(void *args)
{
    parallel_computation(args)
    acquire(&lock);
    done_count++;
    if(done_count < n){
        wait(&all_done, &lock);
    } else {
        broadcast(&all_done);
    }
    release(&lock);
    use_results();

}
```

Exercise 4: Readers/Writers

- Consider a collection of concurrent threads that have access to a shared object
- Some threads are readers, some threads are writers
  - a unlimited number of readers can access the object at same time
  - a writer must have exclusive access to the object

```
int num_readers = 0;
int num_writers = 0;
Lock lock;
CV readable;
CV writeable;
```

```
int reader(void *sha
    acquire(&lock);
    while(num_writers > 0)
        wait(readable, &lock);
    num_readers++;
    release(&lock);
    int x =  read(shared);
    acquire(&lock);
    num_readers--;
    if(num_readers == 0)
        signal(writeable);
    release(&lock);
    return x
}
```

```
void writer(void *shared, int val){
    acquire(&lock);
    while(num_readers > 0)
        wait(writeable, &lock);
    num_writers=1;
    release(&lock);
    write(shared, val);
    acquire(&lock);
    num_writers=0;
    signal(writeable);
    broadcast(readable);
    release(&lock);
}
```

Programming with CVs

C

- Initialization:

```
pthread_mutex_t lock =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv =
    PTHREAD_COND_INITIALIZER;
```

- Lock acquire/release:

```
pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock);
```

- CV operations:

```
pthread_cond_wait(&cv, &lock);
pthread_cond_signal(&cv);
pthread_cond_broadcast(&cv);
```

Python

- Initialization:

```
lock = Lock()
cv = Condition(lock)
```

- Lock acquire/release:

```
lock.acquire()
lock.release()
```

- V

```
cv.wait()
cv.notify()
cv.notify_all()
```

Exercise 5: Feedback

1. Rate how well you think this recorded lecture worked
   1. Better than an in-person class
   2. About as well as an in-person class
   3. Less well than an in-person class, but you still learned something
   4. Total waste of time, you didn't learn anything

2. How much time did you spend on this video (including exercises)?

3. Do you have any particular questions you'd like me to address in this week's problem session?

4. Do you have any other comments or feedback?