Lecture 17: Semaphores and Conditional Variables

CS 105

November 5, 2019

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, int process_shared, 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 initialized with value 1.
    - the 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;
```

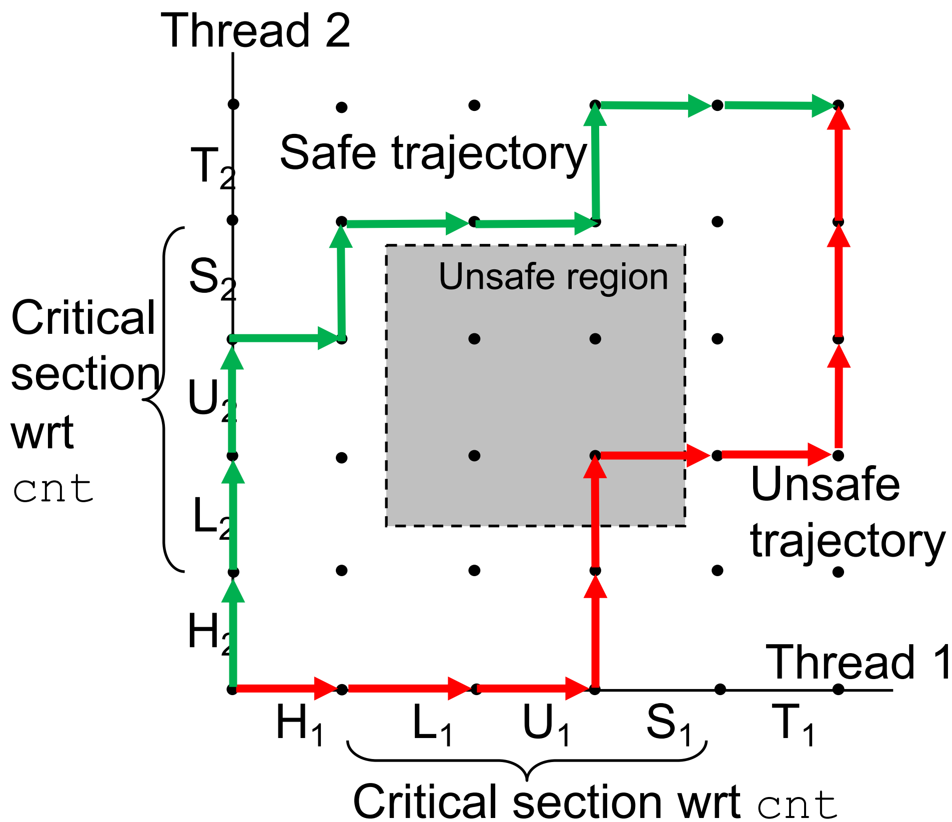
```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
                 *((long *)vargp);

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

        cnt++;

    }

    return NULL;
}
```



Example: Shared counter

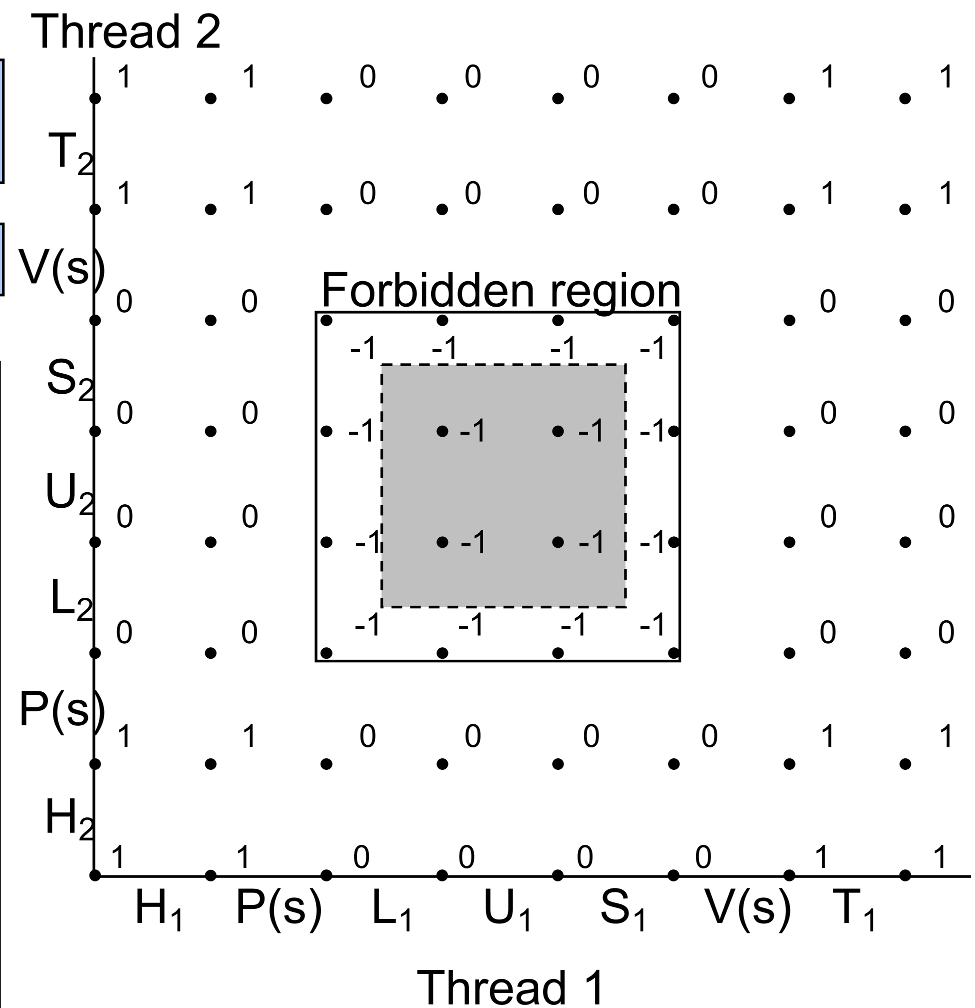
```
volatile long cnt = 0;
sem_t s;
```

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

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

    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;
```

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




    use_results();
}
```

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, FALSE, 1)
sem_t barrier;
sem_init(&barrier, FALSE, 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 that is initialize with a value greater than 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

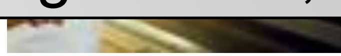
Threads A: produce loaves of bread and put them in the queue

Threads B: consume loaves by taking them off the queue

Example: Bounded Buffers

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

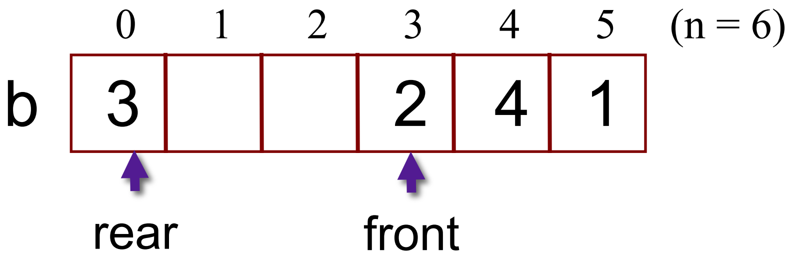
Separation of concerns:
1. How do you implement a queue in an array?
2. How do you implement a bounded buffer, which allows producers to add to it and consumers to take things from it, all in parallel?



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



Values wrap around!!

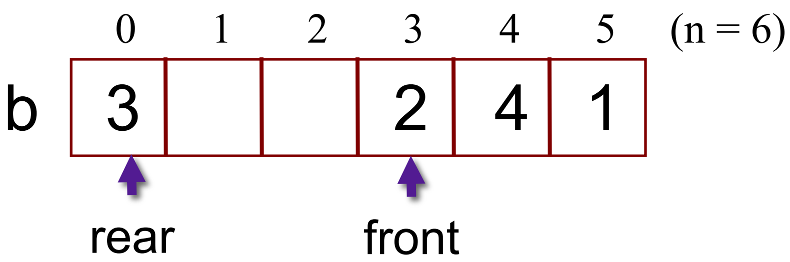
```
typedef struct {
    int *b;         // ptr to buffer containing the queue
    int n;          // length of array (max # slots)
    int front;      // index of first element, 0 <= front < n
    int rear;       // index of last elem, 0 <= rear < n, front==rear if empty
} bbuf_t
```

```
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->rear= ((ptr->rear)+1)%(ptr->n);
  ptr->b[ptr->rear]= val;
}
```

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

Example: Bounded Buffers



Values wrap around!!

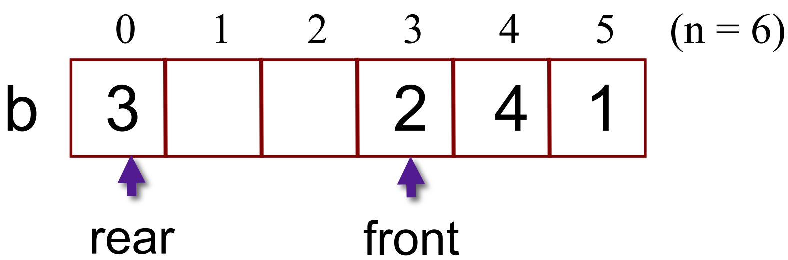
```
typedef struct {
    int *b;         // ptr to buffer containing the queue
    int n;          // length of array (max # slots)
    int front;      // index of first element, 0 <= front < n
    int rear;       // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

```
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;
}
```

Example: Bounded Buffers



```
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, FALSE, 1)
  sem_init(&slots, FALSE, n)
  sem_init(&items, FALSE, 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: 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 the same time
  - a writer must have exclusive access to the object

```
int reader(void *shared){
    int x =  read(shared);
    return x
}
```

```
void writer(void *shared, int val){
    write(shared, val);
}
```

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) a value (non-negative integer)
  - 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 a Condition Variable

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 which a function must wait.
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.

```
volatile int results = 0;
```

```
/* Thread routine */
void *thread(void *args)
{
    parallel_computation(args)




    use_results()

}
```

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;
pthread_mutex_t lock;
pthread_cond_t all_there;
```

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

Exercise: 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 the same time
  - a writer must have exclusive access to the object

```
int reader(void *shared){
    int x =  read(shared);
    return x
}
```

```
void writer(void *shared, int val){
    write(shared, val);
}
```

Condition Variables in C

- *Pthreads:* Standard interface for ~60 functions that manipulate threads from C programs
  - Creating and reaping threads
    - **`pthread_create()`**
    - **`pthread_join()`**
  - Determining your thread ID
    - **`pthread_self()`**
  - Terminating threads
    - **`pthread_cancel()`**
    - **`pthread_exit()`**
    - **`exit()`** [terminates all threads] , **`RET`** [terminates current thread]
  - Synchronizing access to shared variables
    - **`pthread_mutex_init`**
    - **`pthread_mutex_[un]lock`**
    - **`pthread_cond_wait`**
    - **`pthread_cond_signal`**
    - **`pthread_cond_broadcast`**

Condition Variables in C

```
// global declarations
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t has_value = PTHREAD_COND_INITIALIZER;
pthread_cond_t has_space = PTHREAD_COND_INITIALIZER;
```

```
// inside enqueue function
pthread_mutex_lock(&lock);
while ("no space")
    pthread_cond_wait(&has_space, &lock);

  critical section:   … do useful work …

pthread_mutex_unlock(&lock);
pthread_cond_signal(&has_value);
```