CSci 4061
Introduction to Operating Systems

CVs in Posix

Posix condition variables

```
#include <pthread.h>
```

```
pthread_cond_t cond =
    PTHREAD_COND_INITIALIZER;
```

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

Bounded-Buffer (two CVs)

- There is a finite-sized buffer that producer threads want to add items to ... and consumer threads want to remove items from … repeatedly

- **Two** kinds of synchronization needed:
  - **Me**—to protect integrity of the buffer
  - **Correctness**—producer must block if buffer is full and consumer must block if buffer is empty…

Example

```
pthread_mutex_t ring_access =
    PTHREAD_MUTEX_INITIALIZER;

// consumer: wait for content
pthread_cond_t some_content =
    PTHREAD_COND_INITIALIZER;

// producer: wait for a free slot
pthread_cond_t free_slot =
    PTHREAD_COND_INITIALIZER;
```

Two CVs, ONE lock!

```
pthread_mutex_t ring_access =
    PTHREAD_MUTEX_INITIALIZER;

// consumer: wait for content
pthread_cond_t some_content =
    PTHREAD_COND_INITIALIZER;

// producer: wait for a free slot
pthread_cond_t free_slot =
    PTHREAD_COND_INITIALIZER;
```

Example (cont’d)

```
item_t remove_item (buffer *b){
  item_t st;
  pthread_mutex_lock (&mtx);
  while (b->next_slot_to_retrieve ==
         b->next_slot_to_store)
          pthread_cond_wait(&free_slot,&mtx);

  st = b->items [b->next_slot_to_retrieve];
  b->next slot to retrieve++;
  // adjust next_slot_store if needed
  pthread_cond_signal(&some_content);
  pthread_mutex_unlock(&mtx);
  return st;
}
```

Example (cont’d)

```
void insert_item (buffer *b, item_t st) {
 pthread_mutex_lock (&mtx);
 while (b->next_slot_to_store == MAX)
    pthread_cond_wait (&some_content,&mtx);
                 …
 // insert_item
 pthread_cond_signal(&free_slot);
 pthread_mutex_unlock(&mtx);
}
```

BB in action

```
void insert_item (buffer *b, item_t st) {
 pthread_mutex_lock (&mtx);
 while (b->next_slot_to_store == MAX)
         pthread_cond_wait (&some_content,&mtx);

 // insert_item
 pthread_cond_signal(&free_slot);
 pthread_mutex_unlock(&mtx);
}
```

```
item_t remove_item (buffer *b){
  item_t st;
  pthread_mutex_lock (&mtx);
  while (b->next_slot_to_retrieve ==
…
         b->next_slot_to_store)
    pthread_cond_wait(&free_slot,&mtx);
  st = b->items [b->next_slot_to_retrieve];
  b->next slot to retrieve++;
  // adjust next_slot_store if needed
  pthread_cond_signal(&some_content);
  pthread_mutex_unlock(&mtx);
  return st;
}
```

Efficiency

- Taking turns

```
pthread_mutex_t L = PTHREAD_MUTEX_INITIALIZER;;
pthread_cond_t CV = PTHREAD_COND_INITIALIZER;;
int turn = 0;
void* ring (int my_id) {
       while (1) {
          pthread_mutex_lock (&L);
          if (turn == my_id) {
             printf (“%d,”, my_id);
             turn = (turn + 1) % N;
             pthread_cond_broadcast (&CV);
          }
       else pthread_cond_wait (&CV, &L);
       pthread_mutex_unlock (&L);
       }
}
```

Locking

- Serializes access
- Two optimizations
  - Granularity
  - Reader-writer

Granularity

- Expedia.com
  - lock (all_hotels_airlines_rental_car_DB)
  - lock (all_hotels)
  - lock (hilton_minneapolis)

Reader Writer

- Now suppose 1000 just looking at Hilton minneapolis
    - price
    - look of the rooms

Reader-writer

- Implement using CVs

```
num_readers=0; // 0 free, -1 writer has it
Lock L;
Condition CV;
```

```
acquire_rlock () {
  lock (&L);
  if (num_readers < 0)
   wait (&CV, &L);
  else
   num_readers++;
  unlock (&L);
}
```

```
release() {
  lock (&L);
  if (num_readers == -1)
   num_readers = 0;
  else
   num_readers--;
  if (num_readers == 0)
    broadcast (&CV, &L);
  unlock (&L);
}
```

Write `acquire_wlock` on your own.

```
num_readers=0; // 0 free, -1 writer has it
Lock L;
Condition CV;
```

```
acquire_rlock () {
  lock (&L);
  if (num_readers < 0)
   wait (&CV, &L);
  else
   num_readers++;
  unlock (&L);
}
```

```
release() {
  lock (&L);
  if (num_readers == -1)
   num_readers = 0;
  else
   num_readers--;
  if (num_readers == 0)
    broadcast (&CV, &L);
  unlock (&L);
}
```

Problems with the implementation?

In POSIX

```
#include <pthread.h>
pthread_rwlock_t  L;
pthread_rwlock_rdlock (pthread_rwlock_t *);
pthread_rwlock_wrlock (pthread_rwlock_t *);
pthread_rwlock_unlock (pthread_rwlock_t *);
```

Semaphore

- Synchronization tool does not require busy waiting
- Semaphore operations:

  `create_sem`: creates semaphore

  `init_sem(ivalue)`: set *value* of semaphore to `ivalue`

  `P()`: atomic and indivisible (down/wait)

  `V()` :atomic and indivisible (up/post)

  `P`: if value is 0 block, otherwise decrement value

  `V`: increments *value*, release if anyone blocked

- Internally semaphore structure maintains
  - *Value*//semaphore value, always >= 0
  - Queue // list of threads waiting in P() for the value to be >0

Example

- Counting semaphore example:
  - suppose there are N free resources, n threads (n>N)

```
semaphore S: //create
pthread_t t[MaxT]
init_sem (&S,N);
//create N threads
for(i=1;i<n,i++)
 t[i]=pthread_create(…);
```

```
void *fn(…){
…
P(&S);

//got resource!
//do something with it
V(&S);…}
```

- Like a lock, it has state!
  - N=1=> binary semaphore=>mutual exclusion

Posix Semaphores

```
#include <semaphore.h>
int sem_wait(sem_t* sem);  //like P or down
int sem_post(sem_t* sem); // like V or up


//pshared=0 => only threads of process can access
int sem_init(sem_t* sem, int pshared,
             unsigned value);

sem_t sem;  //this is akin to create
```

BB with semaphores

```
//BB of size N with semaphores
sem_t consumer_slots, producer_slots;

sem_init (&consumer_slots,0,0);
sem_init (&producer_slots,0,N);
```

BB: Posix semaphore (cont’d)

```
void buffer_insert(item_t item){
     sem_wait(&producers_slots);  //this is like a P()
     pthread_mutex_lock(&ring_access);
     …
     pthread_mutex_unlock(&ring_access);
     sem_post(&consumer_slots);  //this is like a V()
}
```

Semaphores using Condition Variables

```
condition CV;
lock mutex;
void P(){
    acquire(&mutex);
    while (value == 0)
        wait(&CV,&mutex);
    value = value - 1;
    release(&mutex);
}
```

Deadlock and Synchronization

Dining philosopher

```
while (1) {
    get_forks();
    eat();
    put_forks();
}
```