CSci 4061
Introduction to Operating Systems

Synchronization: Condition Variables

Chapter 13, 14, 16 R&R

Need Richer Synchronization: ~ conditional synchronization

- Want producer (and consumer) to conditionally block if buffer full/empty

// should block if empty

```
item = remove_item (&b);
```

// should block if full

```
insert_item (&b, item);
```

Posix mutex: Bounded Buffer

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
item_t remove_item (buffer *b){
 item t st;
 pthread_mutex_lock (&mtx);
 if (b->next_slot_to_retrieve ==
     b->next_slot_to_store) return ERROR;
 st = b->items [b->next_slot_to_retrieve];
 b->next slot to retrieve++;
 // adjust next_slot_store if needed

 pthread_mutex_lock (&mtx);
 return st;
}
```

What is lacking?

- Cannot suspend/spin while holding a lock
- OK, let’s try conditional synchronization

- `grab lock`
- `=>` ~~`grab lock`~~
- `if <cond> block or spin;`
- `if <cond> unblock or stop spin;`

Need two things

- Conditional synchronization w/o races!
  - grab lock
  - if <cond> block
- Release prior lock atomically

Conditional Variables

- Condition variable are a synchronization construct with simple operations:
  - **wait**: means that the process invoking this operation is suspended until another process/thread invokes **signal**
  - **signal:** operation resumes exactly one suspended process/thread. If no process/thread is suspended, then the signal operation has no effect
  - **broadcast:** wakes up all suspended/processes/threads

Conditional Variables

- Sounds like a lock!

- Almost …

Conditional Variables (cont’d)

```
wait (CV*,Lock*)
```

called with lock held: sleep, atomically releasing lock. Atomically reacquire lock before returning.

some impl don’t need locks here

```
signal (CV*,Lock*)
```

wake up one waiter, if any

```
broadcast (CV*,Lock*)
```

wake up all waiters, if any.

Inside `wait`

atomic

if lock held => {*release lock; sleep*}

else error

=> acquire lock   wakeup and acquire, are **not** atomic

return

Conditional Variables

- *Condition variables* allow *explicit* event notifications

```
acquire/lock (&lock);
while
(<cond>) wait (CV, &lock);
release/unlock (&lock);
```

```
acquire/lock (&lock);
while
(!<cond>)signal(&CV, &lock);
release/unlock (&lock);
```

- Associated with a `mutex` to prevent races on event conditions
- Atomic sleep to prevent deadlock

Example: hello world

Condition CV;

Lock L;

int turn = 1; // hello

T1:

```
lock (&L);
if (turn == 1)
  print (“hello”);
turn = 2;
signal (&CV, &L);
unlock (&L);
```

T2:

```
lock (&L);
if (turn != 2)
   wait (&CV, &L);
print (“world”);
unlock (&L);
```

Wash then Dry; forever using CVs

```
enum sink_t {wash, dry} sink = wash;
Condition CV;
Lock L;
T1 (washer):                  T2 (dryer):
while (1) {                   while (1) {
```

Example #1: License Management

- There are `MAX_L` software licenses
- Must call:
  - `grab_one` to get a license (block if none free)
  - `release` when finished

```
grab_one ();
…
release ();
```

Example #2: Barrier

- Barrier: synchronization construct

```
    init: how_many_threads
    checkin
```

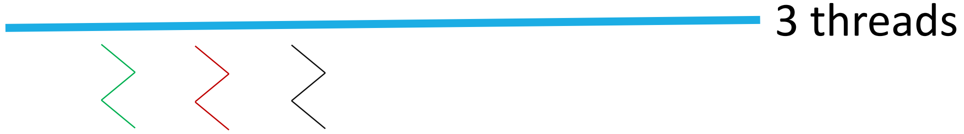
- called by all threads
- blocks all threads until last one checks in

Example #2: Barrier

- Barrier: synchronization construct

```
init: how_many_threads
checkin
```

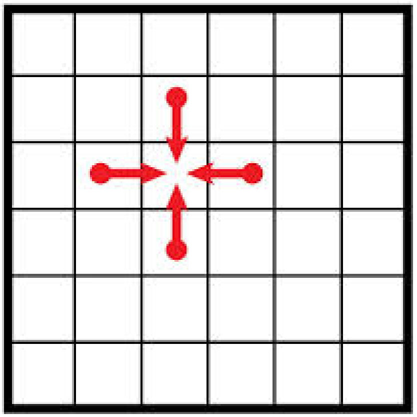
- called by all threads
- blocks all threads until last one checks in



Barrier

- Common in parallel threaded programs

```
for i …
        threads work in parallel on ith iteration
        barrier
```



$$\overset{\downarrow}{x_i^{(k+1)}} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, i\neq j}^{n} a_{ij}\overset{\downarrow}{x_j^{(k)}}\right)$$

Barrier

```
typedef struct {
     int n;
     int num_ci;
     lock L;
     condition CV;
} Barrier;

void init (Barrier *B,
          int num) {
     B->n = num;
     B->num_ci = 0;
}
```

```
//USAGE
Barrier B;

void *thread_fn (…) {
   …
   checkin (&B);
   …
}

void main (…) {
     …
     init (&B, n);
     // launch threads
     …
}
```

```
void checkin (Barrier *B);
```