Course Operating Systems

Complementary Notes: Locks, condition variables and use in synchronization problems

EDA093, DIT 401

Study Period 1

Ack: pseudocode sections are adaptations
of examples in Leture Notes by John Ousterhout, Stanford Un:
https://web.stanford.edu/~ouster/cgi-bin/cs140-spring14/lecture.php?topic=locks

Locks

- *Lock*: an object that can only be "owned" by a single thread at any given time. Basic operations:
  - `acquire`: mark the lock as owned by the current thread; if some other thread already owns the lock then first wait (spinning or @queue) until the lock is free.
  - `release`: mark the lock as free (it must currently be owned by the calling thread).

Bounded producer-consumer buffer: solve it using locks

Remember the requirements:

- **Producer** inserts items; must wait if buffer full
- **Consumer** removes items; must wait if buffer empty
- Accessing the buffer is a **critical section**

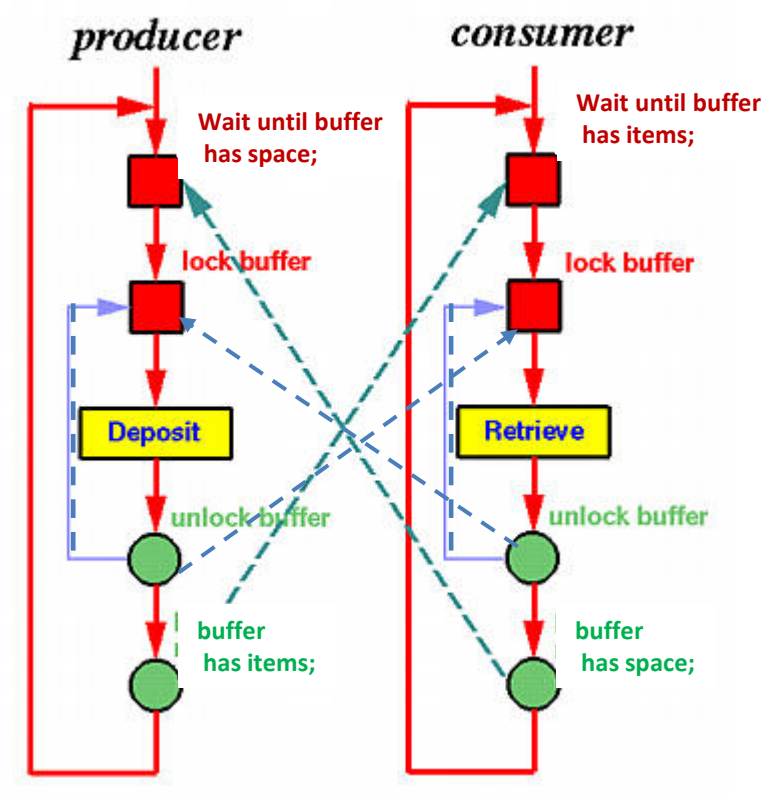


fig C.K. Shene
http://www.cs.mtu.edu/~shene/NSF-3/e-Book/

Bounded buffer using locks
(PintOS-like API)

```
<some_type> buffer[SIZE];
int count = 0;
struct lock l;   // synchronization object
lock_init(&l);
```

producer

```
do {

  // produce item

  lock_acquire(&l);
  while (count == SIZE) {
    lock_release(&l);
    lock_acquire(&l);
  }

  // add item to buffer

  count++;
  lock_release(&l);

} while (True);
```

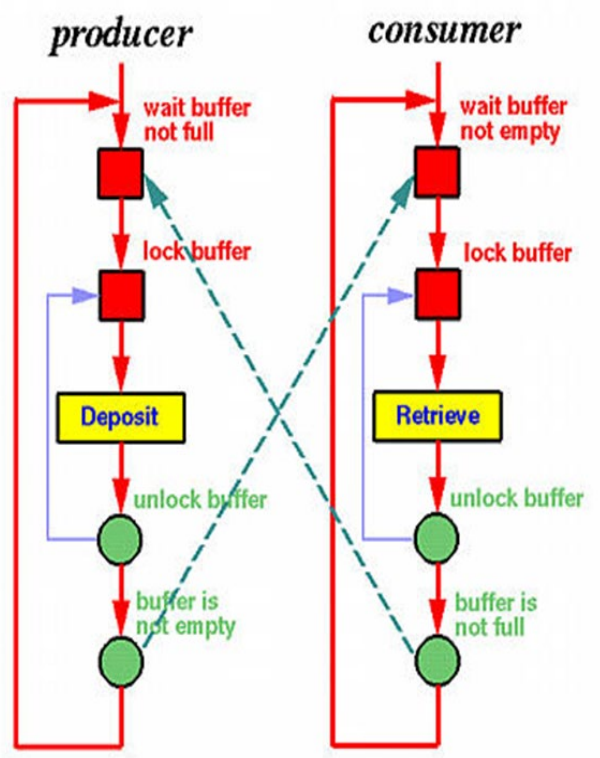
consumer

```
do {
  lock_acquire(&l);
  while (count == 0) {
    lock_release(&l);
    lock_acquire(&l);
  }

  // remove item from buffer

  count--;
  lock_release(&l);

  // use the item
} while (True);
```



Homework: compare it with the semaphore-based solution studied in class

Condition Variables

Condition variable: an object to wait for a particular condition to become true. Associated with a *lock*

API: (PintOS-like (*)):

- `cond_wait(condition, lock)`: release lock, put thread to sleep until condition is signaled; when thread wakes up again, re-acquire lock before returning.
- `cond_signal(condition, lock)`: if any threads are waiting on condition, wake up one of them. Caller must hold lock, which must be the same as the lock used in the wait call.
- `cond_broadcast(condition, lock)`: same as signal, except wake up all waiting threads.

(*) Note: after signal, signaling thread keeps lock, waking thread goes on the queue waiting for the lock.

Check also `Ptreads` API eg here https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf for perspective

Bounded buffer using condition variables (PintOS-like API)

```
<some_type> buffer[SIZE];
int count = 0;
struct lock l;   // synchronization object
struct condition dataAvailable; // synchronization object
struct condition spaceAvailable; // synchronization object

lock_init(&l);
cond_init(&dataAvailable);
cond_init(&spaceAvailable);
```

producer

```
do {

  // produce item

  lock_acquire(&l);
  while (count == SIZE) {
    cond_wait(&spaceAvailable, &l);

  }

 // add item to buffer

count++;
cond_signal(&dataAvailable, &l);
lock_release(&l);
} while (True);
```

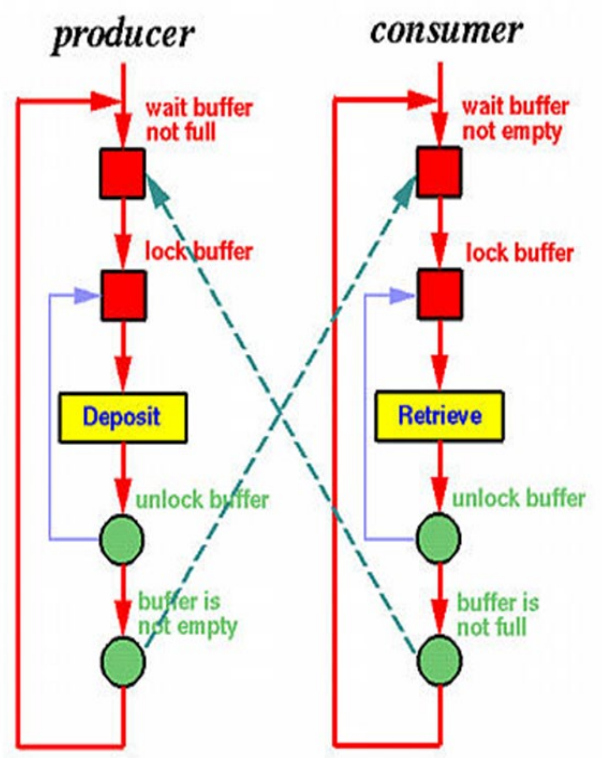
consumer

```
do {
  lock_acquire(&l);
  while (count == 0) {
    cond_wait(&dataAvailable, &l);
}

// remove item from buffer


count--;
cond_signal(&spaceAvailable, &l);
lock_release(&l);

// use the item
} while (True);
```



Homework: compare it with the lock-based and the semaphore-based solutions we studied

Observe

- Condition Variables: higher-level synchronization mechanism that provide
  - Mutual exclusion: easy to create critical sections
  - Scheduling-related functionality: block thread until some desired event occurs
- When locks and condition variables are used together like this, the result is called a *monitor* :
  - A collection of procedures manipulating a shared data structure.

See also:

- Operating Systems: Three Easy Pieces Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau (University of Wisconsin-Madison) https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf