Synchronization / Resource allocation

Operating Systems, EDA092 - DIT400

6.15 Servers can be designed to limit the number of open connections. For example, a server may wish to have only *N* socket connections at any point in time. As soon as *N* connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

- Shared variables
  - **var** *connections = MAX* : *semaphore*
- New Connection

wait(connections);

accept connection

|

terminate connection

signal(connections);

The Bounded-Buffer (Producer-Consumer) Problem

- *N* locations, each can hold one item
- **Producer** inserts items; must wait if buffer full
- **Consumer** removes items; must wait if buffer empty

- Solve this synch problem using semaphores

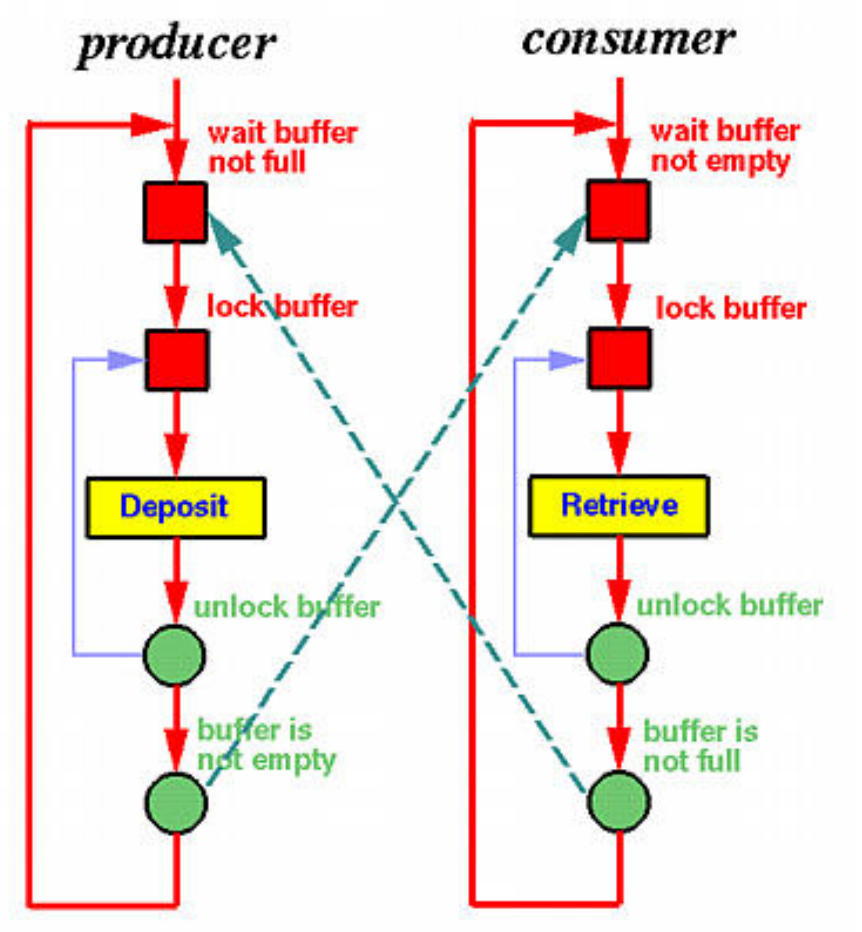
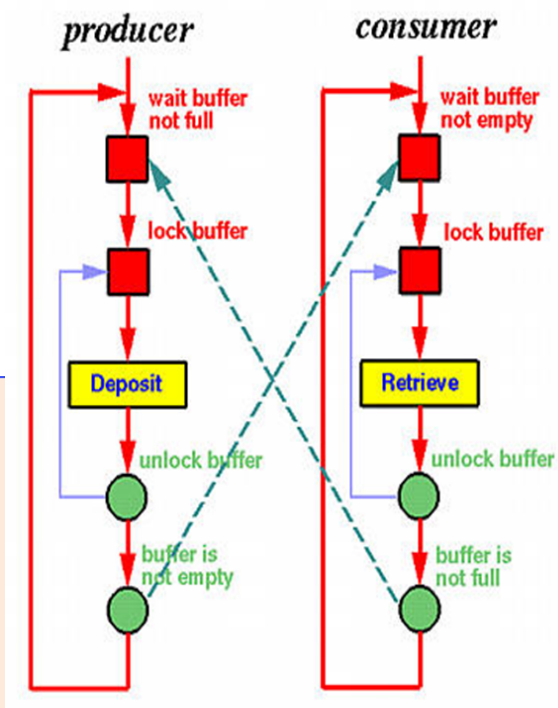


fig C.K. Shene
http://www.cs.mtu.edu/~shene/NSF-3/e-Book/

The Bounded-Buffer (Producer-Consumer) Problem

Synchronization variables:

- Binary semaphore mutex initialized to 1
- General semaphore buffer-has-items initialized to 0
- General semaphore buffer-has-space initialized to N.



producer

```
do {
        // produce item

        wait (buffer-has-space);
        wait (mutex);

        // add item to buffer

        signal (mutex);
        signal (buffer-has-items);
} while (TRUE);
```

consumer

```
do {
        wait (buffer-has-items)
        wait (mutex);

        // remove item from buffer

        signal (mutex);
        signal (buffer-has-space);

        // consume item
} while (TRUE);
```

Readers/Writers Problem

Similar to critical section , but

several readers can execute “critical section” at the same time

- But if a writer is in its critical section, then no other process can be in its critical section.

The readers–writers problem has several variations, all involving priorities.

Readers/Writers Problem

Readers have "priority"…

Shared variables:
w, mutex: boolean semaphore; initially 1
rc: int //readers-counter

Writer::
Wait(*w*); //exclusive access
  CS;
Signal(*w*)

Reader::
Wait(*mutex*); // enforce mutex when changing rc
rc := *rc* + 1;
  if *rc* = 1 then Wait(*w*) fi; // first reader wait if writer writes
Signal(*mutex*);

CS;

Wait(*mutex*); // enforce mutex when changing rc
  rc := *rc* − 1;
  if *rc* = 0 then Signal(*w*) fi; // last reader signals writer
Signal(*mutex*)

The starvation in the readers-writers problem could be avoided by keeping timestamps associated with waiting processes.

- Writer task wakes up the process that has been waiting longest.
- Readers only access the critical section if there are no waiting writers.

CHAPTER 7
(Silberschatz)

7.14 A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if both a northbound and a southbound farmer get on the bridge at the same time (Vermont farmers are stubborn and are unable to back up). Using semaphores, design an algorithm that prevents deadlock. Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, and vice versa).

```
semaphore ok_to_cross = 1;
   void enter_bridge() {
            ok_to_cross.wait();
      }
   void exit_bridge() {
      ok_to_cross.signal();
   }
```

Old Bridge

a) Correctness Constraints
   - I. At most 3 cars are on the bridge at a time
   - II. All cars on the bridge go in the same direction
   - III. Whenever the bridge is empty and a car is waiting, that car should get on the bridge
   - IV. Whenever the bridge is not empty or full and a car is waiting to go the same direction as the cars on the bridge, that car should get on the bridge
   - V. Only one thread accesses shared state at a time

b) Cars will be waiting to get on the bridge, but in two directions. Use an array of two condition variables, `waitingToGo[2]`.

c) It will be necessary to know the number of cars on the bridge (`cars`, initialized to 0), and the direction of these cars if there are any (call it `current-direction`). It will also be useful to know the number of cars waiting to go in each direction; use an array `waiters[2]`.

Old Bridge

```
ArriveBridge(int direction) {
  lock.acquire();

  // while can't get on the bridge, wait
  while ((cars == 3) ||
         (cars > 0 && currentdirection != direction)) {
    waiters[direction]++;
    waitingToGo[direction].wait();
    waiters[direction]--;
  }

  // get on the bridge
  cars++;
  currentdirection = direction;

  lock.release();
}
```

https://inst.eecs.berkeley.edu/~cs162/fa13/hand-outs/synch-problems.html

https://inst.eecs.berkeley.edu/~cs162/fa13/hand-outs/synch-solutions.html

Old Bridge

```
ExitBridge() {
  lock.acquire();

  // get off the bridge
  cars--;

  // if anybody wants to go the same direction, wake them
  if (waiters[currentdirection] > 0)
    waitingToGo[currentdirection].signal();
  // else if empty, try to wake somebody going the other way
  else if (cars == 0)
    waitingToGo[1-currentdirection].broadcast();

  lock.release();
}
```

https://inst.eecs.berkeley.edu/~cs162/fa13/hand-outs/synch-problems.html

https://inst.eecs.berkeley.edu/~cs162/fa13/hand-outs/synch-solutions.html