CSCI315 – Operating Systems Design

Department of Computer Science
Bucknell University

Synchronization Tools: semaphores

Ch 6.6

This set of notes is based on notes from the textbook authors, as well as L. Felipe Perrone, Joshua Stough, and other instructors.
Xiannong Meng, Fall 2021.

Issue With the Lock Solution

- While **locks** (and other hardware-based solutions we discussed in last segment) do well to ensure the exclusive access to shared data, the solution is simplistic.
  - It may result in "busy waiting," not a good use of resources.
  - It is possible that the waiting time is not bounded as we cannot control the order with locks.

Semaphores

- **Semaphore** – an abstract data type consisting of two parts, a counter and a queue, working together to provide atomic operations

- **Counting semaphore** – the counter value is unlimited

- **Binary semaphore** – the counter can only be 0 or 1; it can be simpler to implement (also known as **mutex** locks).

- Provides **mutual exclusion**:

```
semaphore S(1);  // initialized to 1
wait(S);         // or acquire(S) or P(S)
criticalSection();
signal(S);       // or release(S) or V(P)
```

Semaphore Implementation

```
typedef struct {
  int value;
  struct process_t *list;
} semaphore;
```

It looks like a normal C variable, except that operations on semaphores are **atomic**, just like what we saw in **test_and_set()** and **compare_and_swap()** to ensure the integrity of the value.

Semaphore Implementation

```
wait(semaphore *S) { // try to enter
    S->value--;
    if (S->value < 0) { // others in CR
            add the process to S->list;
            sleep();    // or wait()
     }
}
```

These two operations have to be **atomic**!

```
signal(semaphore *S) { // leave
    S->value++;
    if (S->value <= 0) { // others waiting
            remove a process P from S->list;
            wakeup(P);  // or signal()
    }
}
```

Here **wait()** is also known as the **P** operation, and **signal()** as **V**. These are Dutch words were given by Dijkstra, a world-renowned Dutch-native computer scientist, who invented the notion.

https://en.wikipedia.org/wiki/Semaphore_(programming)#Operation_names

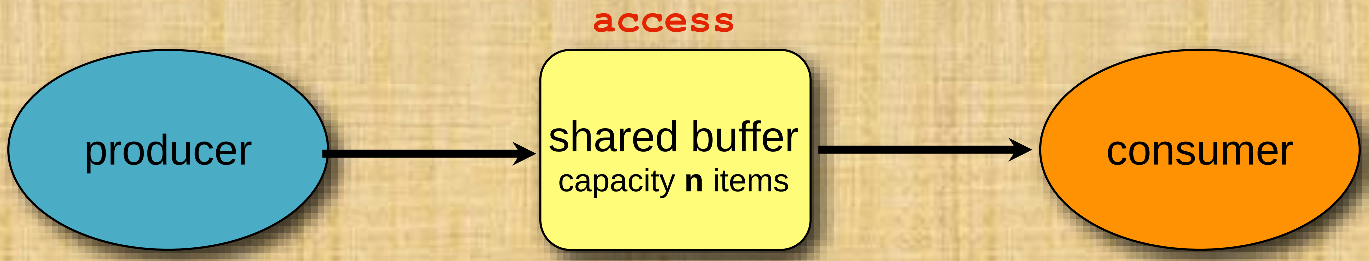
The Bounded-Buffer Problem

Binary semaphore

Counting semaphore

```
int n;
mutex access;
semaphore empty;
semaphore full;
```

```
init(&access,1);
init(&empty,n);// wait for space
init(&full,0); // wait for item
```

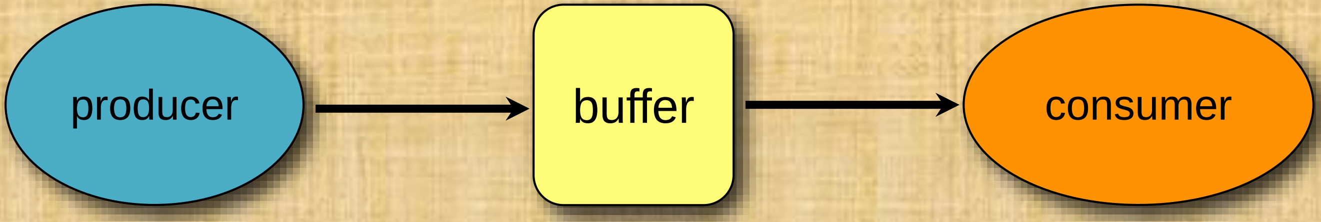


Why do we initialize **access** to be 1?
Why **empty** be n?
Why **full** be zero?

The Bounded-Buffer Problem

```
do {// produce item and save
    wait(&empty);
    wait(&access);
    // add item and save
    signal(&access);
    signal(&full);
} while (true);
```

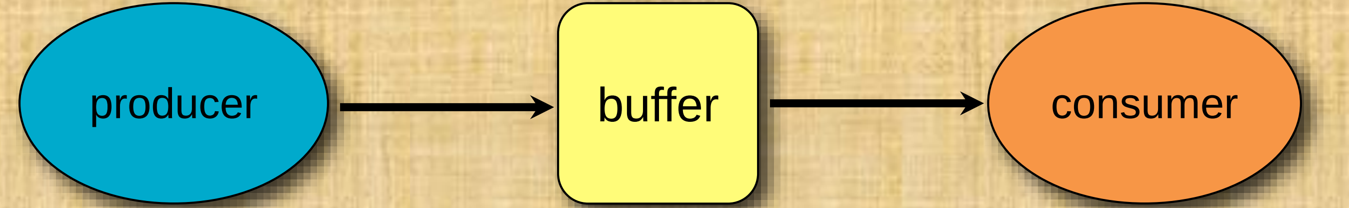
Producer



The Bounded-Buffer Problem

```
do {// produce item and save
     wait(&empty);
     wait(&access);
     // add item and save
     signal(&access);
     signal(&full);
} while (true);
```

critical section



producer

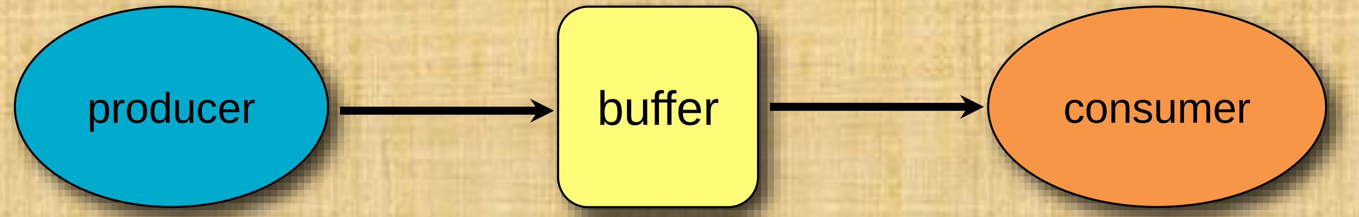
buffer

consumer

The Bounded-Buffer Problem

Consumer

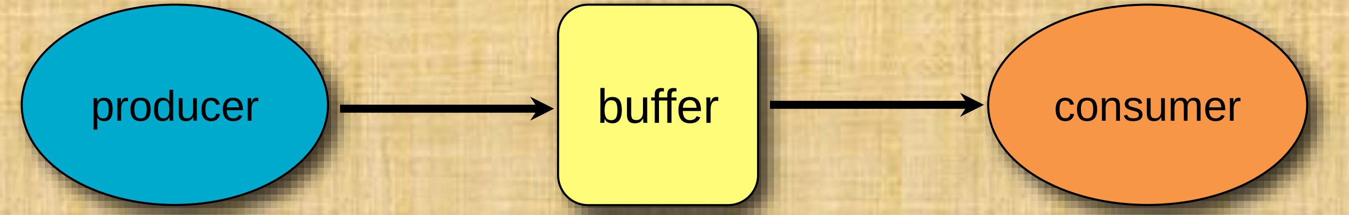
```
do {
        wait(&full);
        wait(&access);
        // remove item and save
        signal(&access);
        signal(&empty);
        // consume save item
} while (true);
```



The Bounded-Buffer Problem

Critical Section

```
do {
       wait(&full);
       wait(&access);
       // remove item and save
       signal(&access);
       signal(&empty);
       // consume save item
} while (true);
```



Monitor

- Semaphores are low-level synchronization resources.
- A programmer's honest mistake can compromise the entire system (well, that is almost always true). We should want a solution that reduces the risk.
- The monitor is one such data type:

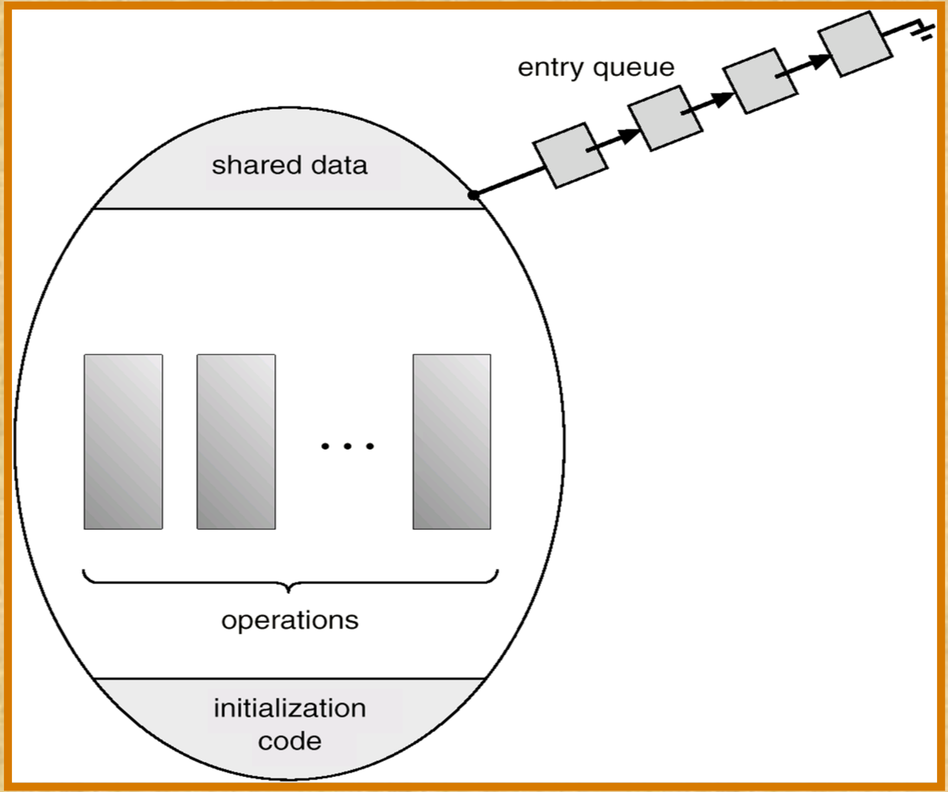
```
monitor mName {
   // declare shared variables
   procedure P1 (…) {
       …
   }
   procedure Pn (…) {
       …
   }
   init code (…) {
      ….
   }
}
```

A *procedure* can access only local variables defined within the monitor.

There cannot be concurrent access to procedures within the monitor (only one process/thread can be *active* in the monitor at any given time).

Condition variables: queues are associated with variables. Primitives for synchronization are wait and signal.

Monitor


entry queue
shared data
. . .
operations
initialization
code