Chapter 5: Synchronization

CSCI 3753 Operating Systems

Instructor: Chris Womack

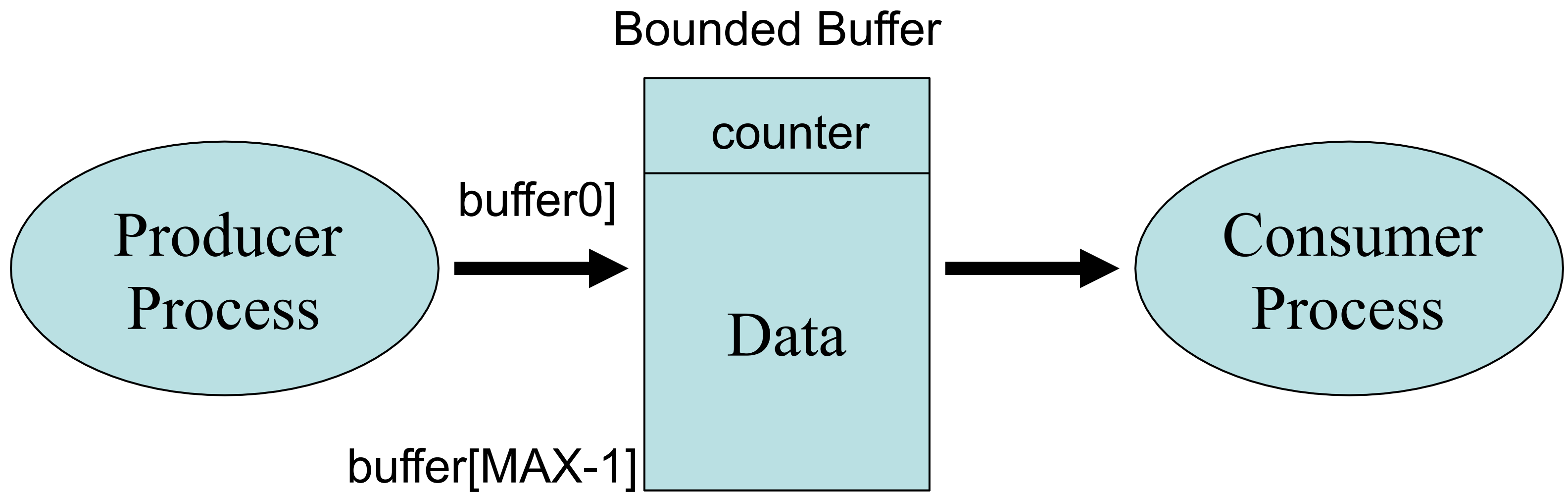
University of Colorado at Boulder

All material by Dr. Rick Han

Synchronization

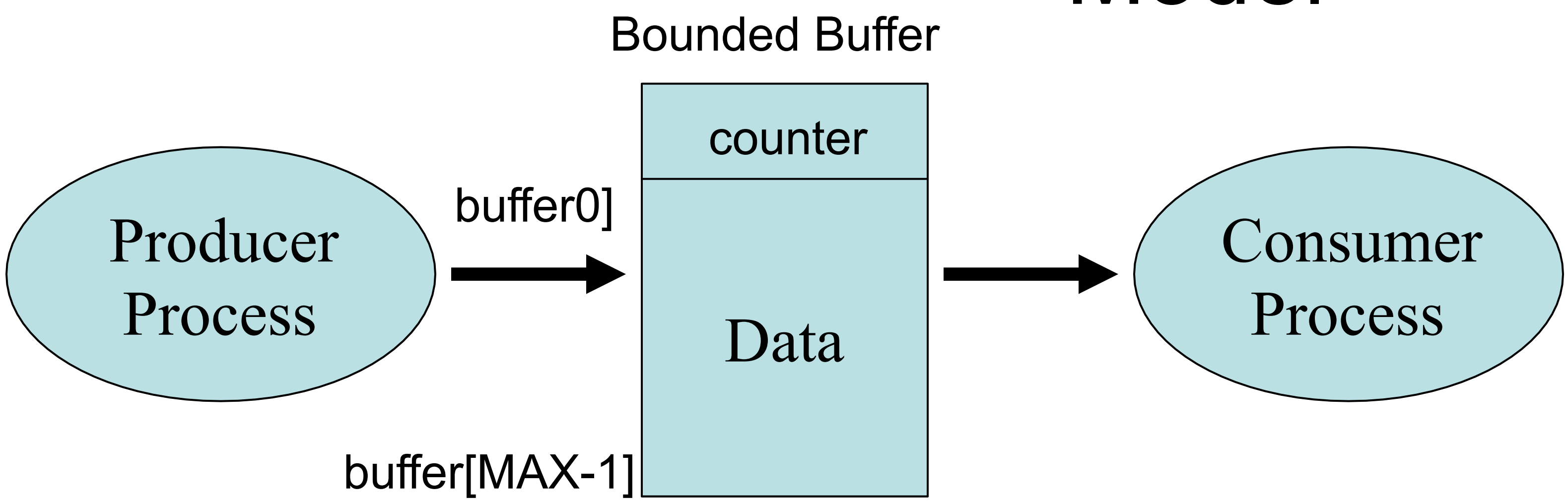
- Protect access to shared common resources, e.g. buffers, variables, files, devices, etc., by using some type of synchronization
- Saw the need for synchronization earlier:
  - 2 processes P1 and P2 use IPC shared memory to modify the same shared memory variable
  - 2 threads T1 and T2 modify the same global or heap variables in same address space, so need thread-safe code
  - Normal and exceptional control flow both try to modify the same global variable
- Producer-Consumer model

Producer-Consumer Model



- a Producer process P1 and a Consumer process C1 share some memory, say a bounded buffer
- Producer writes data into shared memory, while Consumer reads data

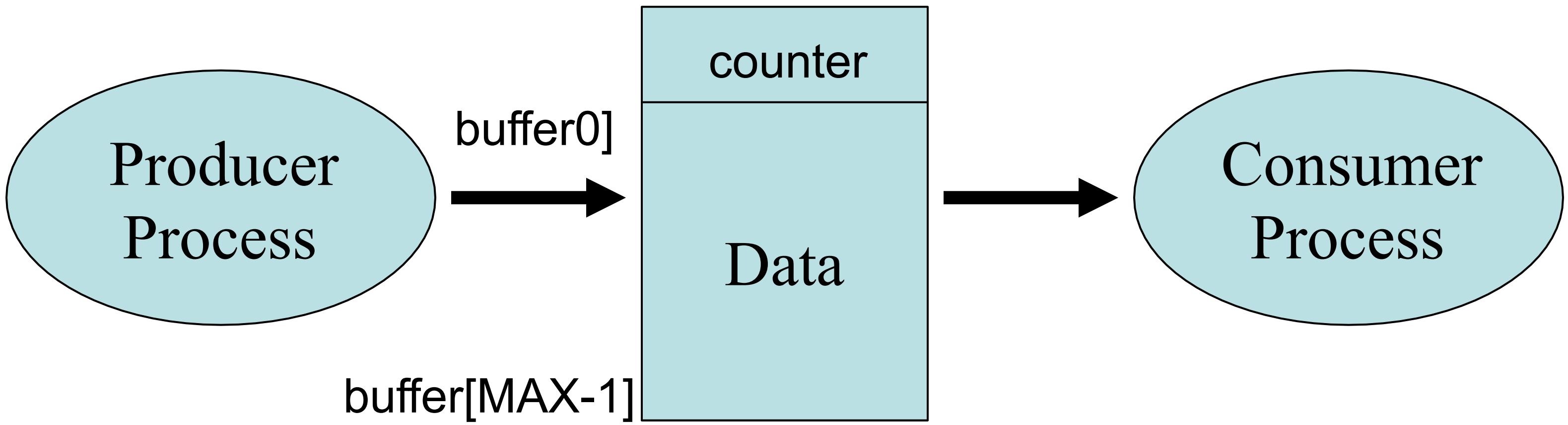
Producer-Consumer Model



- Track buffer level with a variable *counter*
  - Increment counter when new data is produced/written
  - Decrement counter when data is consumed/read
  - keeps track of how much new data is in the buffer that has not yet been read
  - if counter==0, then consumer can't read from the buffer
  - if counter==MAX_BUFF_SIZE, then producer can't write to the buffer

Synchronization

Bounded Buffer




```
while(1) {!
    while(counter==MAX);!
    buffer[in] = nextdata;!
    in = (in+1) % MAX;!
    counter++;!
}!
```

```
while(1) {!
    while(counter==0);!
    getdata = buffer[out];!
    out = (out+1) % MAX;!
    counter--;!
}!
```

Producer writes new data into buffer and increments counter

counter updates can conflict!

Consumer reads new data from buffer and decrements counter

Synchronization

counter++; can compile into several machine language instructions, e.g.

```
reg1 = counter;!
reg1 = reg1 + 1;!
counter = reg1;!
```

counter--; can compile into several machine language instructions, e.g.

```
reg2 = counter;!
reg2 = reg2 - 1;!
counter = reg2;!
```

If these low-level instructions are *interleaved*, e.g. due to context-switching, then the results of counter's value can be unpredictable

A Race Condition Example

- Let brackets [value] denote local value of counter in either the producer or consumer's process. counter=5 initially.

// counter++

```
(1) reg1 = counter;     [5]
(3) reg1 = reg1 + 1;    [6]
(5) counter = reg1;     [6]
```

// counter--;

```
(2) reg2 = counter;     [5]
(4) reg2 = reg2 - 1;    [4]
(6) counter = reg2;     [4]
```

- Counter should be 5 with 1 producer and 1 consumer, but counter = 4! Reversing steps (5) and (6) sets counter=6
- Undesirable and unpredictable *race condition*
- Basic Problem: unprotected access to a shared variable (counter)

Critical Section

- Some kernel data structures could be subject to race conditions, e.g. access to list of open files
- Kernel developer must ensure that no such race conditions occur
- User or kernel developer identifies *critical sections* in code where each process accesses shared variables
  - access to critical sections is controlled by special *entry* and *exit* code

while(1) {

entry section

critical section (manipulate common var's)

exit section

remainder section code

}

Critical Section

- Critical section access should satisfy multiple properties

  mutual exclusion

  - if process P_i is executing in its critical section, then no other processes can be executing in their critical sections

  progress

  - if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next
  - this selection cannot be postponed indefinitely (OS must make a decision eventually, hence "progress")

  bounded waiting

  - there exists a bound, or limit, on the number of times other processes can enter their critical sections after a process X has made a request to enter its critical section and before that request is granted (no starvation)

- For the rest of this chapter, we will primarily be focused on how to achieve mutual exclusion