COMP-SCI-431
Intro Operating Systems

Lecture 4 – Concurrency

Adu Baffour, PhD

University of Missouri-Kansas City
Division of Computing, Analytics, and Mathematics
School of Science and Engineering
aabnxq@umkc.edu

Lecture Objectives

- To understand the concept of process interactions and their significance in operating systems.
- To explore the role and functionality of semaphores as synchronization mechanisms in concurrent programming.
- To examine the implementation of semaphores and their practical application in controlling access to shared resources.
- To investigate the concept of monitors and their role in providing higher-level synchronization abstractions.
- To analyze classic synchronization problems and their solutions within the context of concurrent programming and operating systems.

Outline

Outline

4.1 Process interactions

4.2 Semaphores

4.3 Implementation of semaphores

4.4 Monitors

4.5 Classic synchronization problems

4.1 Process interactions

Process competition

- ***Concurrency*** is the act of multiple processes (or threads) executing simultaneously.
- When multiple physical CPUs are available, the processes may execute in parallel.
- On a single CPU, concurrency may be achieved by time-sharing.
- When concurrent processes access a shared data area, the data must be protected from simultaneous change by two or more processes.
- Otherwise, the updated area may be left in an inconsistent state.
- A ***critical section*** is a segment of code that a process cannot enter while another process executes a corresponding code segment.

4.1 Process interactions

Process competition

- General structure of a process

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

4.1 Process interactions

Any solution to the critical section (CS) problem must satisfy the following requirements:

1. Guarantee ***mutual exclusion***: Only one process may be executed within the CS.
2. Prevent ***lockout***: A process not attempting to enter the CS must not prevent other processes from entering the CS.
3. Prevent ***starvation***: A process (or a group of processes) must not be able to repeatedly enter the CS while other processes are waiting to enter.
4. Prevent ***deadlock***: Multiple processes trying to enter the CS simultaneously must not block each other indefinitely.

4.1 Process interactions

A software solution to the CS problem.

Peterson’s Algorithm

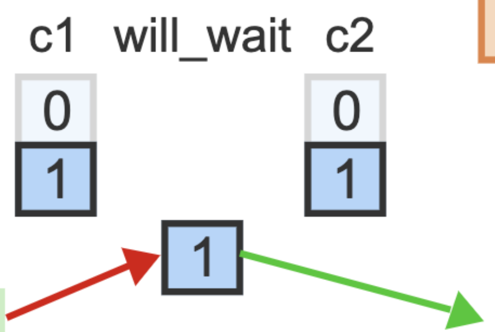
```
int c1 = 0, c2 = 0, WillWait;
cobegin
  p1: while (1) {
        c1 = 1;
        willWait = 1;
        while (c2 && (WillWait==1)); /*wait*/
        CS1; c1 = 0; program1;
      }
  p2: while (1) {
        c2 = 1;
        willWait = 2;
        while (c1 && (WillWait==2)); /*wait*/
        CS1; c2 = 0; program2;
      }
```

4.1 Process interactions

<u>Peterson's Algorithm</u>

Process p1

```
while (1) {
    c1 = 1
    will_wait = 1
    while (c2 && (will_wait==1)) /*wait*/
    CS
    c1 = 0
}
```



Process p2

```
while (1) {
    c2 = 1
    will_wait = 2
    while (c1 && (will_wait==2)) /*wait*/
    CS
    c2 = 0
}
```

4.1 Process interactions

Disadvantages

1. The solutions work only for 2 processes. When 3 or more processes need to share a CS, a different solution must be developed.
2. The solution is inefficient. While one process is in the CS, the other process must wait by repeatedly testing and setting the synchronization variables.
3. The solution addresses only competition among processes.

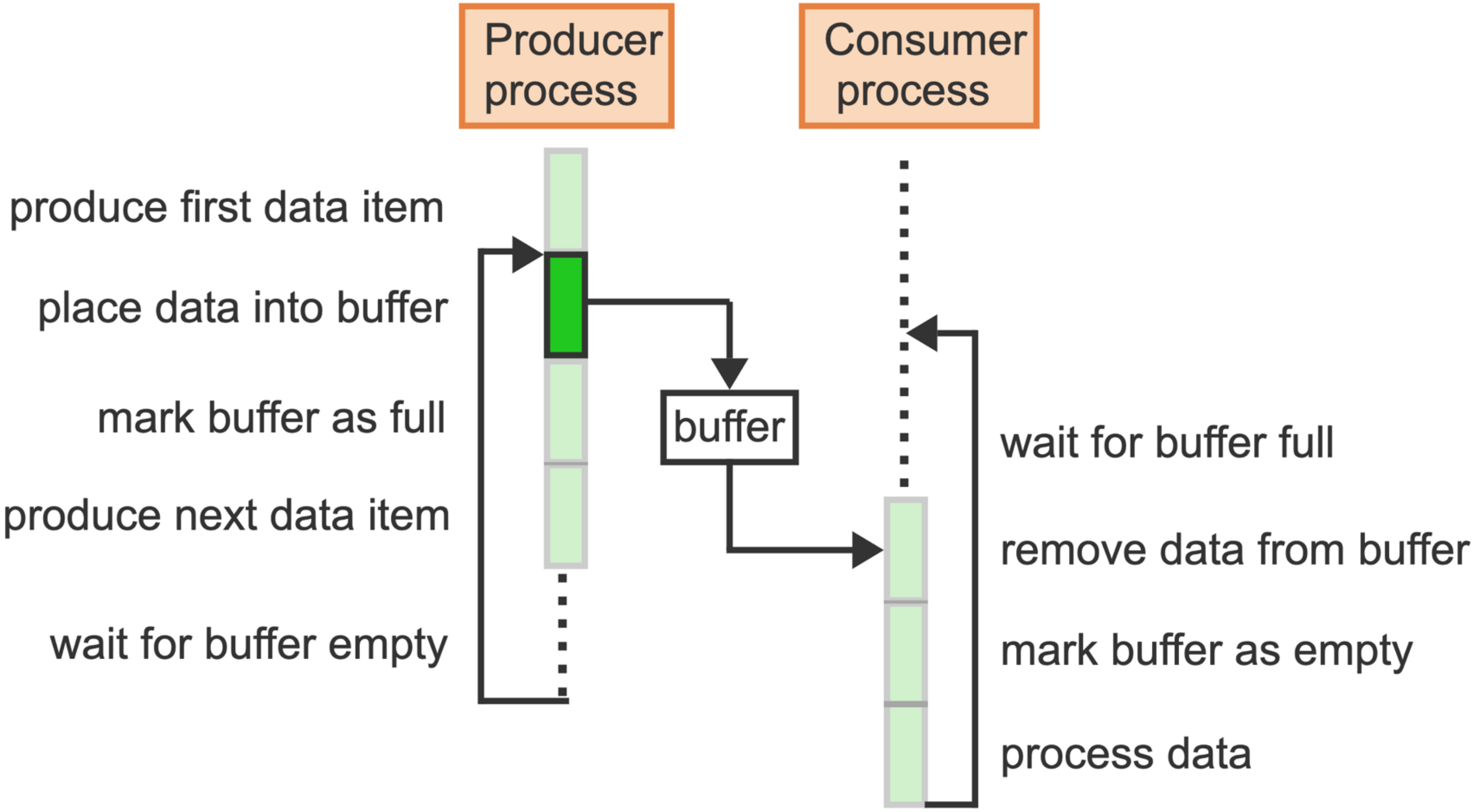
4.1 Process interactions

Process cooperation

- In addition to the CS problem, many applications require the cooperation of processes to solve a common goal.
- Ex: one process produces data needed by another process.
- The producer process must be able to inform the waiting process whenever a new data item is available.
- In turn, the producer must wait for an acknowledgment from the consumer.

4.1 Process interactions

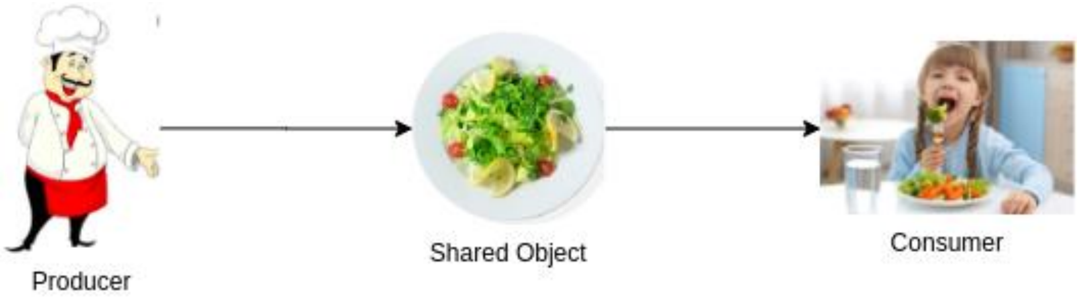
Producer-consumer synchronization



4.2 Semaphores

The bounded-buffer problem

- A producer process shares a fixed-sized buffer with a consumer process.
- The producer fills empty slots with data in increasing order.
- The consumers follow the producer by removing the data in the same order.
- The solution must guarantee the following:
  - Consumers do not overtake producers and access empty slots.
  - Producers do not overtake consumers and overwrite full slots.



Outline

4.1 Process interactions

4.2 Semaphores

4.3 Implementation of semaphores

4.4 Monitors

4.5 Classic synchronization problems

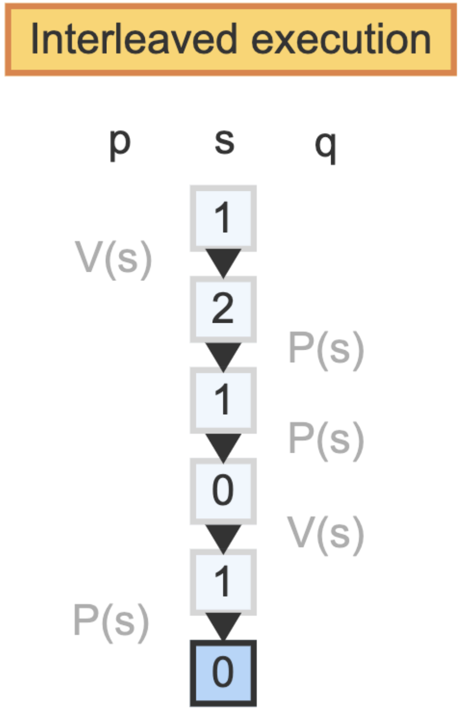
4.2 Semaphores

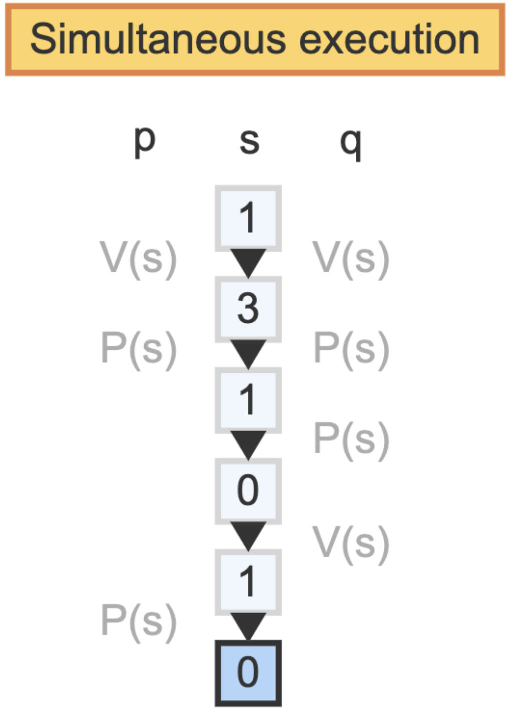
Basic principles of semaphores

- A ***semaphore*** s is a non-negative integer variable that can be accessed using only two special operations, P and V.
  - V(s): increment s by 1
  - P(s): if s > 0, decrement s by 1, otherwise wait until s > 0
- Implementing P and V must guarantee that if several processes simultaneously invoke P(s) or V(s), the operations will occur sequentially in some arbitrary order.
- If more than one process is waiting inside P(s) for s to become > 0, one of the waiting processes is selected to complete the P(s) operation.

4.2 Semaphores

P and V operations on a semaphore

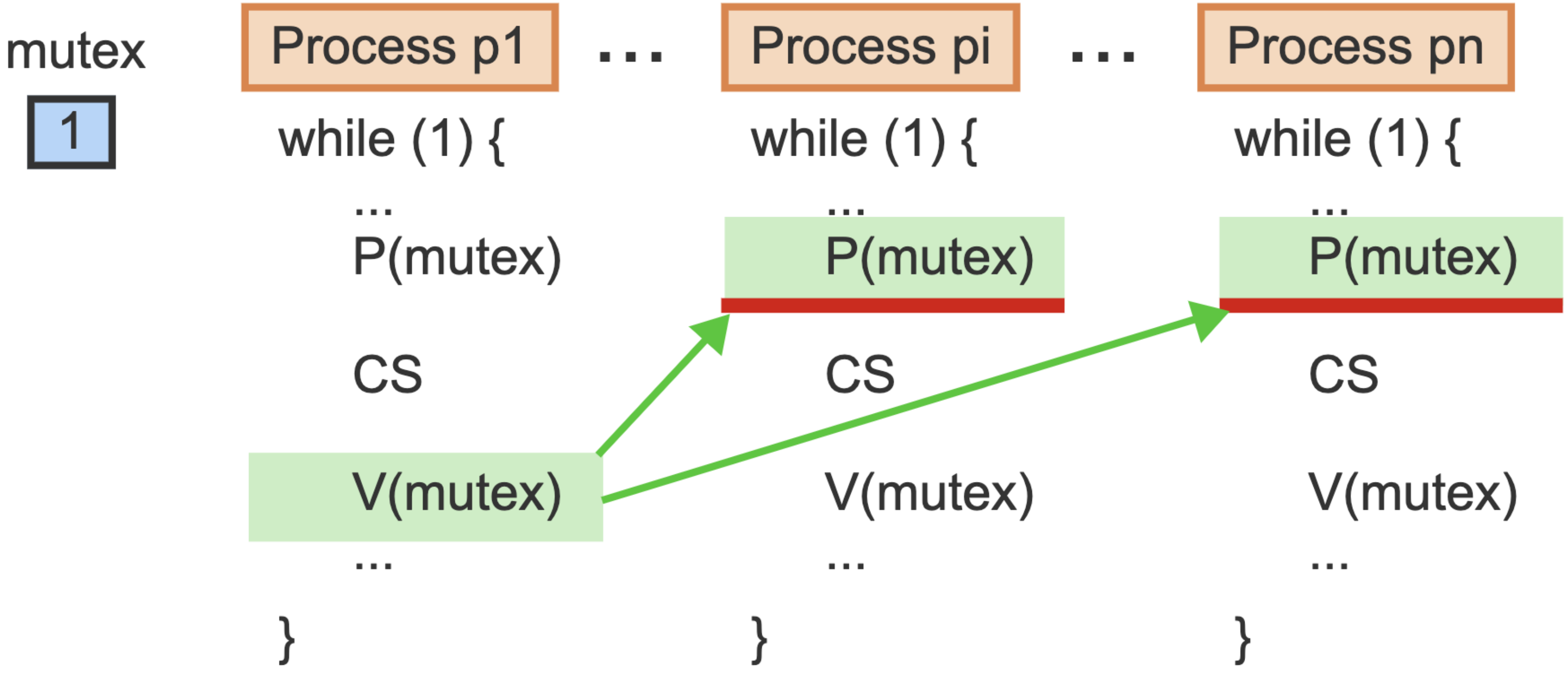




4.2 Semaphores

The CS problem using semaphores

- A single semaphore, initialized to 1, is sufficient to solve the problem for any number of processes.



4.2 Semaphores

The bounded-buffer problem using semaphores

- The semaphore **f** represents the number of full buffer slots and is incremented each time the producer places a new data item into the buffer and decremented each time the consumer removes an item from the buffer.
- The semaphore **e** represents the number of empty slots and is analogously modified by the producer and consumer.

Initially: f = 0; e = n;

```
Producer
while (1) {
    produce data
    P(e)
    place data into buffer
    V(f)
}
```

```
Consumer
while (1) {
    P(f)
    remove data from buffer
    V(e)
    process data
}
```

Outline

4.1 Process interactions

4.2 Semaphores

4.3 Implementation of semaphores

4.4 Monitors

4.5 Classic synchronization problems

4.3 Implementation of semaphores

Hardware support for synchronization

- The ***test-and-set instruction*** (***TS***) copies a variable into a register and sets the variable to zero in one indivisible machine cycle.
- Test-and-set has the form TS(R, x) where R is a register and x is a memory location and performs the following operations:

  - Copy x into R
  - Set x to 0

- A ***lock*** is a synchronization barrier through which only one process can pass.

4.3 Implementation of semaphores

test-and-set instruction (***TS***)

```
boolean test_and_set(int *target) {
        boolean oldValue = *target;
        *target = true;
        return oldValue;
}
```

Shared boolean variable `lock`, initialized to `false`

```
do {
     while(test_and_set(&lock)){
       /* do nothing */
     }

       /* critical section */

       lock = false;
       /* remainder section */
} while(true);
```

4.3 Implementation of semaphores

Binary semaphores

- A ***binary semaphore*** can take only the values 0 or 1.
- Pb and Vb are the simplified P and V operations that manipulate binary semaphores.
- Vb(sb): sb = 1
- Pb(sb): sb = 0
- ***Busy-waiting*** is repeatedly executing a loop while waiting for some condition to change.
- Implementing Pb(sb) using TS is very simple but suffers from the drawback of busy-waiting.

4.3 Implementation of semaphores

Synchronization problems using binary semaphores

Critical section (CS) problem

Initially: mutex = 1

Process p1 ··· Process p2

```
while (1) {
    ...
    Pb(mutex)

    CS

    Vb(mutex)
    ...
}
```

```
while (1) {
    ...
    Pb(mutex)

    CS

    Vb(mutex)
    ...
}
```

Bounded buffer problem

Initially: empty = 1, full = 0

Producer

```
while (1) {

    produce data

    Pb(empty)

    place data
    into buffer

    Vb(full)
}
```

Consumer

```
while (1) {

    Pb(full)

    remove data
    from buffer

    Vb(empty)

    process data
}
```

4.3 Implementation of semaphores

Implementing P and V operations on general semaphores

- A general semaphore s can be implemented using a regular integer variable manipulated by the functions P(s) and V(s).
- A binary semaphore guarantees that only one operation can access and manipulate s.
- The variable s serves a dual purpose:
  - When s is greater or equal to 0, s represents the semaphore value.
  - Whenever s falls below 0, the value represents the number of processes blocked on the semaphore.
- To avoid busy-waiting inside P(s) when s falls below 0, the process performs a blocking request, which places the process on a waiting list associated with s.
- A V(s) operation increments s and reactivates one process if one or more are blocked on s.

4.3 Implementation of semaphores

Implementation of general semaphores


```
P(s) {
    Pb(ms)
    s = s - 1
    if (s < 0)
        Vb(ms)
        block self on s
    Vb(ms)
}
```

```
V(s) {
    Pb(ms)
    s = s + 1
    if (s <= 0)
        reactivate a process
    else
        Vb(ms)
}
```

Outline

4.1 Process interactions

4.2 Semaphores

4.3 Implementation of semaphores

4.4 Monitors

4.5 Classic synchronization problems

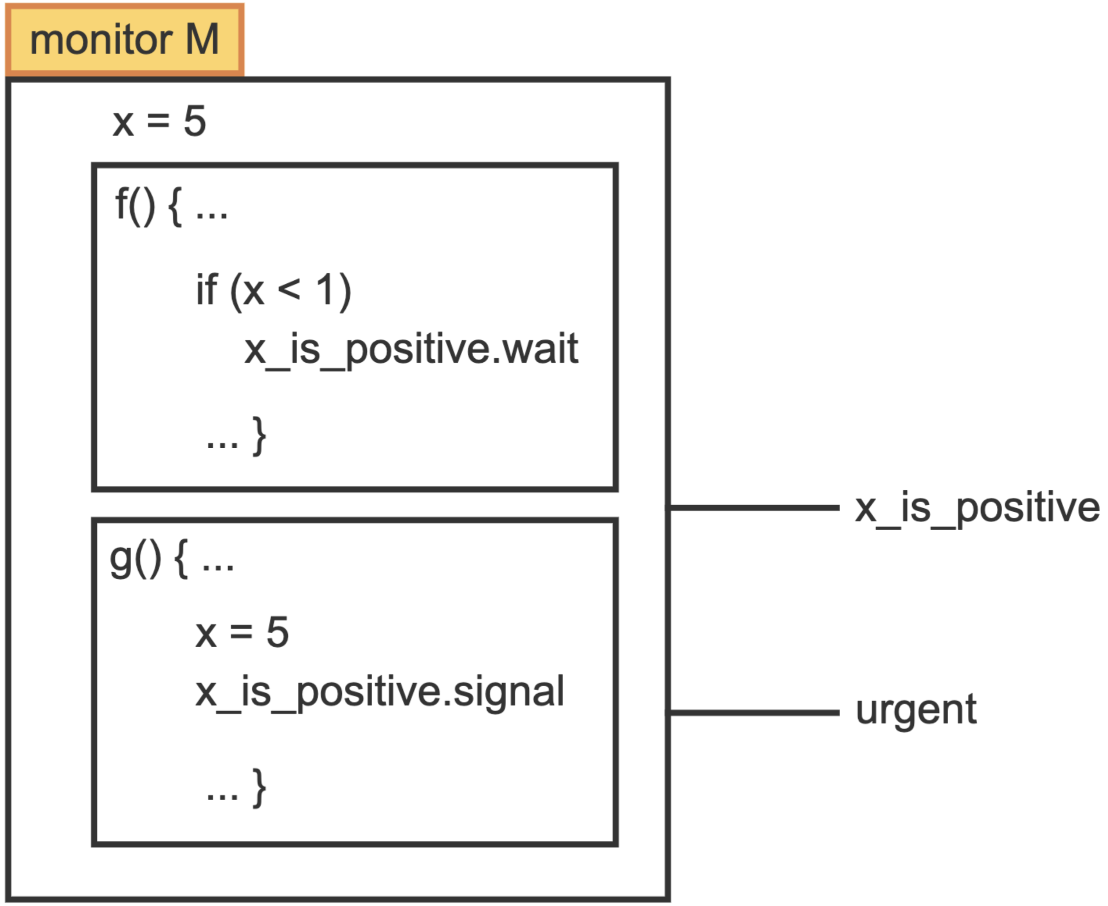
4.4 Monitors

Basic principles of monitors

- A ***monitor*** is a high-level synchronization primitive implemented using P and V operations.
- A ***condition variable*** is a named queue on which processes can wait for some condition to become true.
- The monitor must guarantee that the functions are mutually exclusive.
- A condition variable c is accessed using two special operations:
  - ***c.wait*** causes the executing process to block and be placed on a waiting queue associated with the condition variable c.
  - ***c.signal*** reactivates the process at the head of the queue associated with the condition variable c.

4.4 Monitors

Operation of a monitor



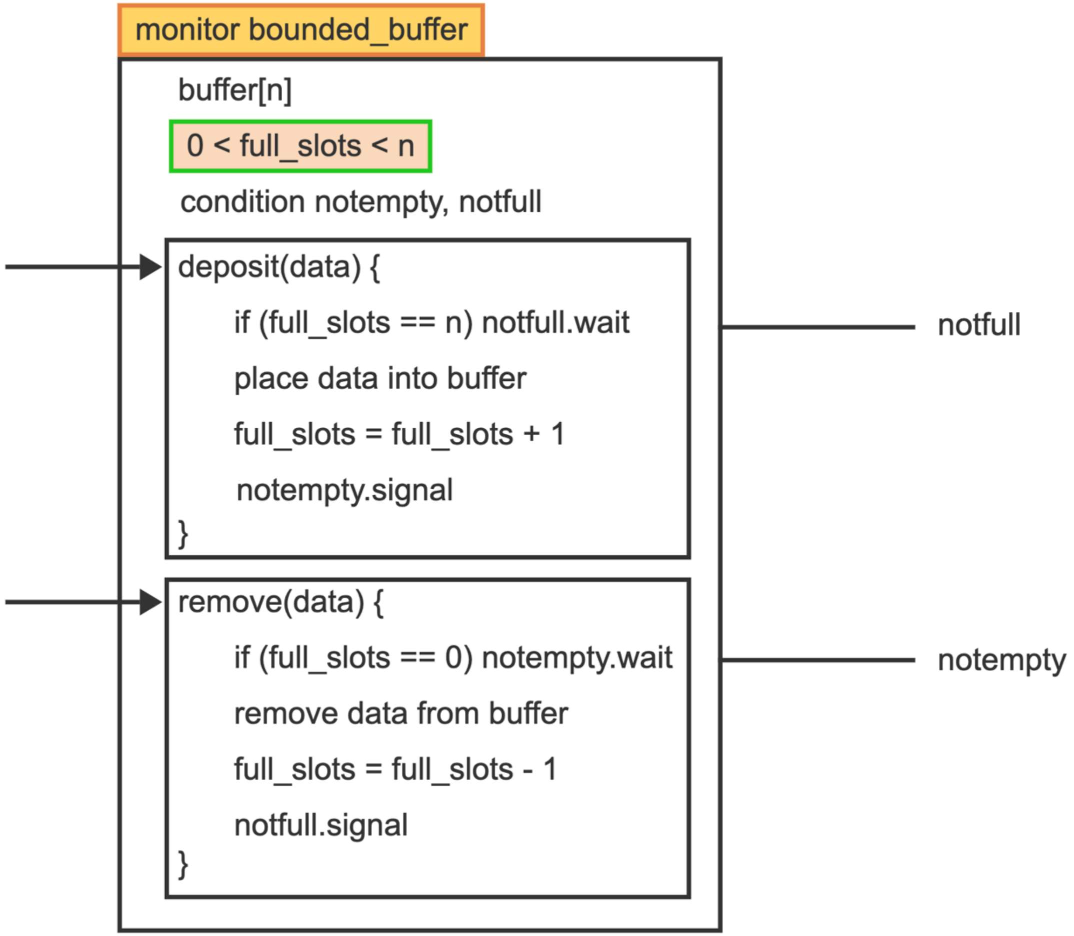
4.4 Monitors

A monitor implementation of the bounded-buffer problem

- Since the monitor guarantees mutual exclusion, depositing and removing data automatically become critical sections.
- Consequently, the solution works for multiple producers and multiple consumers.
- The producer uses the condition not full to wait when all buffer slots are full.
- Analogously, the consumer uses the condition not-empty to wait when all buffer slots are empty.
- A counter, full_slots, initially set to 0, is used to track how many slots are full.
- The counter is incremented by the producer and decremented by the consumer during each call to the monitor.

4.4 Monitors

Operation of the bounded buffer monitor



4.4 Monitors

Monitors with priority waits

- Usually, a queue associated with a conditional variable is processed in FIFO order.
- A ***priority wait*** has the form c.wait(p), where c is a conditional variable, and p is an integer specifying a priority according to which processes blocked on c are reactivated.

Outline

4.1 Process interactions

4.2 Semaphores

4.3 Implementation of semaphores

4.4 Monitors

4.5 Classic synchronization problems

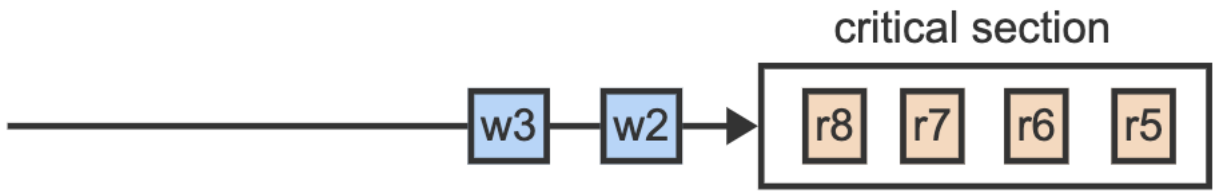
4.5 Classic synchronization problems

The readers-writers problem

- The main challenge is to guarantee maximum concurrency of readers while preventing the starvation of either type of process. Specifically, two rules must be enforced:

1. A reader can join others in the CS only when no writer is waiting. When the last reader exits the CS, the writer is allowed to enter.
2. All readers who have arrived while a writer is in the CS must be allowed to enter before the next writer.

- Rule 1 guarantees that writers cannot starve.
- Rule 2 guarantees that readers cannot starve. Jointly, the two rules guarantee maximum concurrency of readers.



4.5 Classic synchronization problems

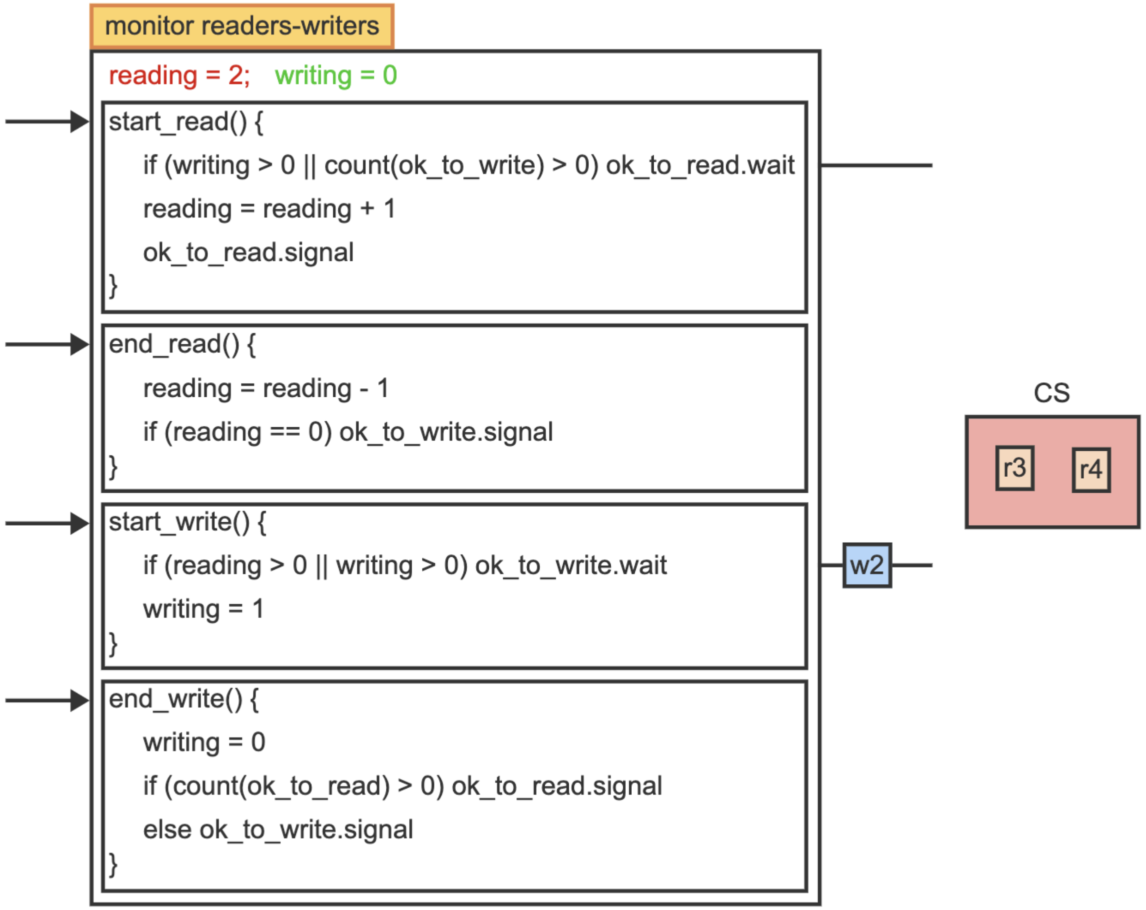
A monitor solution to the readers-writers problem

The monitor provides four functions:

- start_read is called by a reader to get permission to read
- end_read is called by a reader when finished reading
- start_write is called by a writer to get permission to write
- end_write is called by a writer when finished writing
- Two counters, reading and writing, are used to keep track of the number of readers and writers currently in CS, respectively.
- Two condition variables, ok_to_read and ok_to_write, are used to block readers and writers, respectively.
- A primitive count(c) is provided, which returns the number of processes blocked on c.

4.5 Classic synchronization problems

Readers-writers problem using a monitor



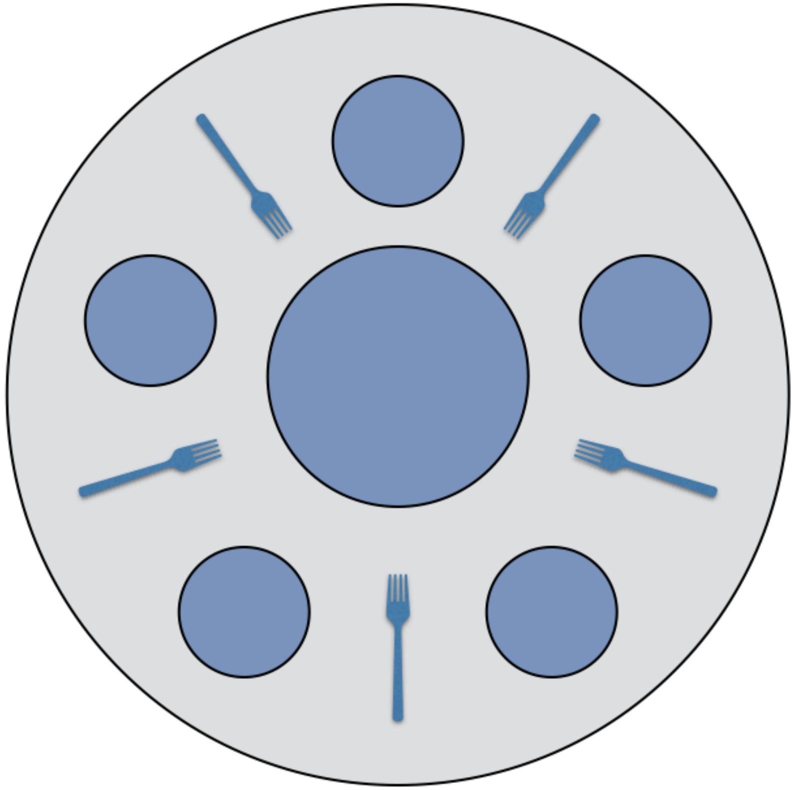
4.5 Classic synchronization problems

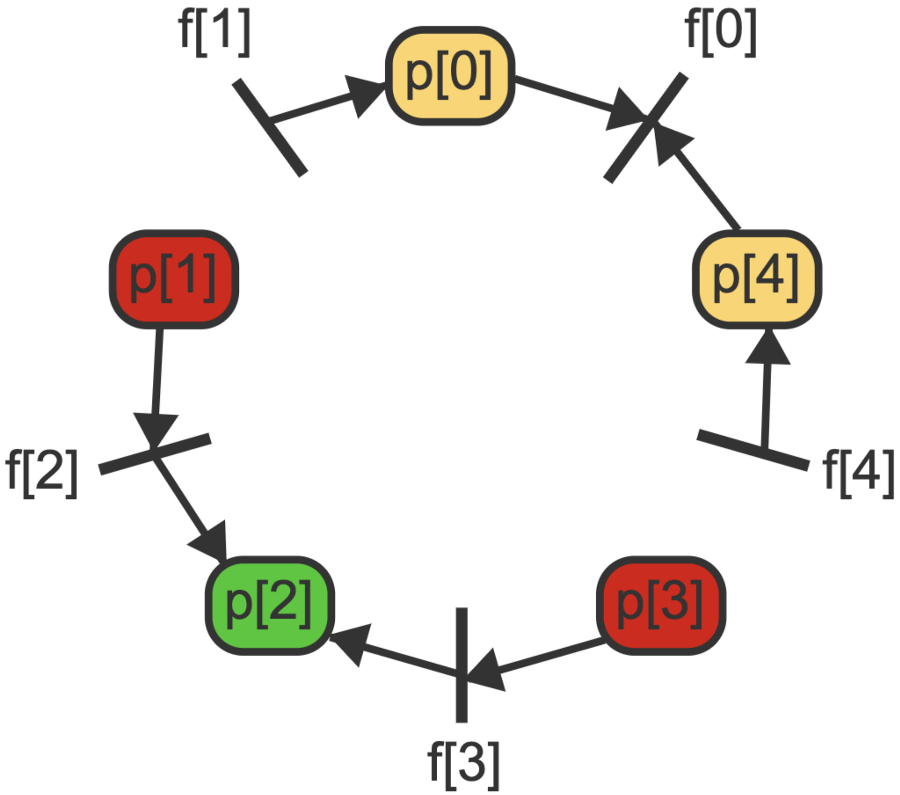
The dining-philosophers problem

- Five "philosophers," each representing a concurrent process, are seated around a table.
- Five "forks," each representing a resource, are placed on the table such that every two neighboring philosophers share one fork.
- Each philosopher alternates asynchronously between a phase of:
  - "thinking" – which represents execution not requiring any shared resources
  - "eating" – requires the prior acquisition of the two forks adjacent to the philosopher and shared with the two respective neighbors.
- The main challenge is preventing deadlock while guaranteeing that two nonadjacent philosophers can always eat concurrently.

4.5 Classic synchronization problems

The dining-philosophers problem





4.5 Classic synchronization problems

Approaches to preventing deadlock

- Before eating, p[i] requests the two adjacent forks and returns the forks when finished eating.
- The forks are 5 semaphores, f[0] through f[4], initialized to 1.
- P(f[i]) then corresponds to picking up fork f[i], and V(f[i]) corresponds to putting down fork f[i].
- This can lead to a deadlock since all philosophers can pick up the left fork f[i] concurrently and then block indefinitely on picking up the right fork.
- Approach 1: Request both forks simultaneously in a critical section.
- Approach 2: One philosopher picks up the forks in the opposite order from all other philosophers.

```
p(i) {
   while (1) {
          think
          P(f[i])
          P(f[i+1 mod 5])
          eat
          V(f[i])
          V(f[i+1 mod 5])
   }
}
```

End of Lecture

Thank you

Any questions?