Operating Systems:
Internals and Design Principles
William Stallings

Chapter 5

Co tual d Synchroniza ont.)

Outline

- OS synchronization mechanisms
  - Semaphore
    - Binary and general semaphores
    - Implementation
  - Producer/con
  - Bounded buff
  - Monitor
    - Condition variables
  - Message passing
  - Reader/Writer problem

Common Concurrency Mechanisms

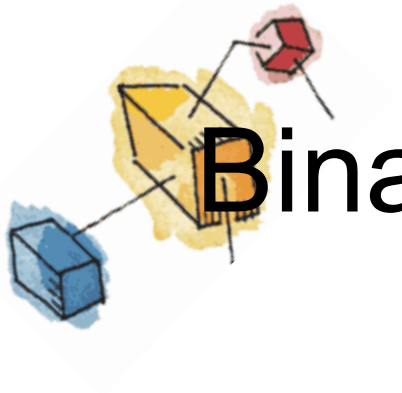
Semaphore

- A queue is used to hold processes waiting on the semaphore – eliminating busy-wait
- Semaphore:
  - An integer v among processes.
- Only three operations m ormed on a semaphore, all of which are atomic:
  - Initialize the integer,
  - decrement the value (`semWait`),
  - increment the value (`semSignal`)

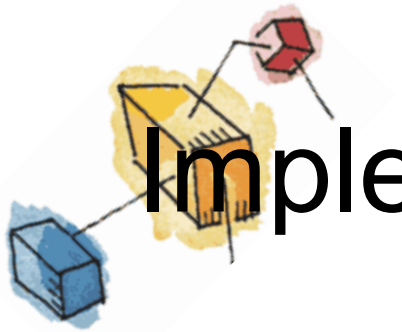
Semaphore

- The semaphore `s` is initially assigned a zero or positive value
- When a process issues a `semWait`, `s` is decremente
  - The process will be blocke negative
  - The negative value equals r of blocked processes waiting to be unblocked
- Each `semSignal`, incrementing `s`, unblocks one of the waiting processes when `s` is negative

Semaphore Primitives



Binary Semaphore Primitives



Implementation of Semaphores

- the semWait and semSignal operations themselves are critical sections and thus must be implemented as atomic primitives
- Use one of t ed schemes for mutual exclu

Strong/Weak Semaphores

- A queue is used to hold processes waiting on the semaphore – eliminating busy-wait
  - In what order are processes removed from the queue?

Strong Sem

- the process that has been bl gest is released from the queue first (FIFO)

Weak Semaphores

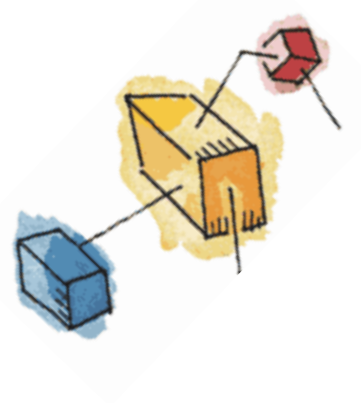
- the order in which processes are removed from the queue is not specified

Mutual Exclusion Using Semaphores

/* Initialize s to 1 */

Mutual Exclusion Using Semaphores

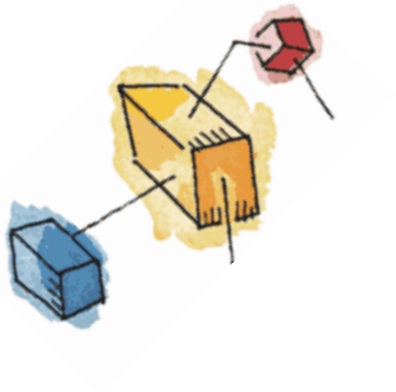
Synchronization Using Semaphores



```
Semaphore s = 0;        /* Initialize s to 0 */
Proc_0()                                    1() {
  . . .                                      .
  S1;                                        wait(s);
  semSignal (s);                             S2;
  . . .                                      . . .
}                                           }
```


Producer/Consumer Problem

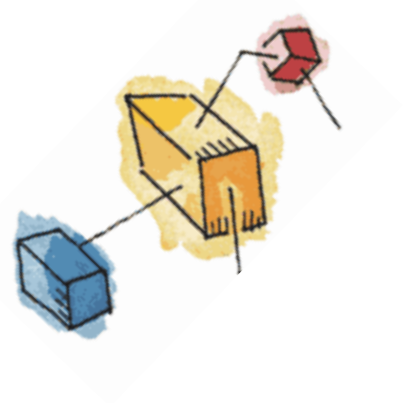
- General Situation:
  - One or more producers are generating data and placing these in a buffer
  - One or more consumers are taking items out of the buffer one at a time
  - Only one producer or consumer may access the buffer at any one time
- The Problem:
  - Ensure that the Producer can't add data into full buffer and consumer can't remove data from empty buffer



Buffer Structure

Binary Semaphores: Incorrect Solution

A producer can update n before if statement!



Binary Semaphores: Correct Solution

Can cause starvation!

Producers continue to append & make n > 1 always! →

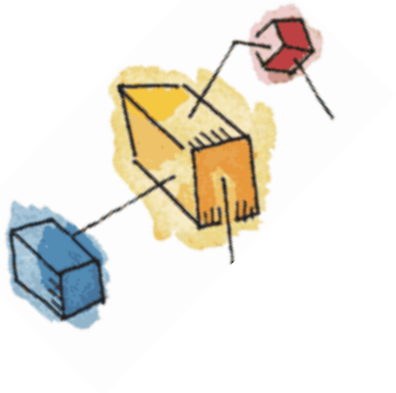
C1: delay = 0 →
Starved here!

C0: m > 0 →

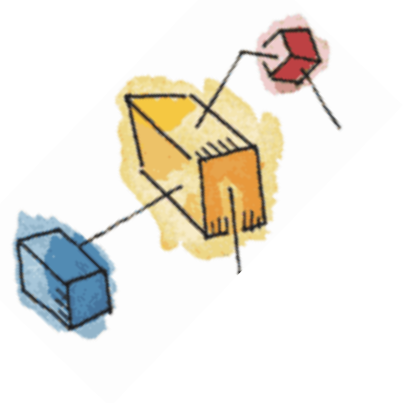
General Semaphores

Ensure consumer not to take items when the buffer is empty

For mutual exclusion



Bounded Buffer



Semaphores

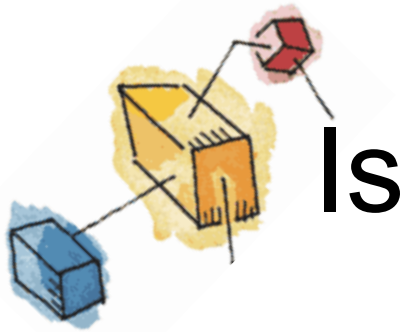
Ensure producer not to add items when the buffer is full

Producers wait here if the buffer is full

Producer informs Consumers that there are data items available

Issues with Semaphores

- Semaphores provide a powerful synchronization tool. However,
  - semSignal() and semWait() are scattered among se[illegible] refore, it is difficult to [illegible] cts
  - Usage must be correct [illegible] processes.
  - One bad process (or one programming error) can kill the whole system.



Issues with Semaphores

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
semWait (S);	emWait (Q);
semWait (Q);	emWait (S);
…	…
semSignal (S);	semSignal (Q);
semSignal (Q);	semSignal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Dining Philosophers Problem

Dining Philosophers Problem

Warning: This solution could deadlock!

Monitors

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java
- Software module consisting of one or more procedures, an initialization sequence, and local data

Monitor Characteristics

Local data variables are accessible only by the monitor's procedures and not by any external procedure

Only one process may be executing in the monitor at a time

Process enters monitor by invoking one of its procedures

Schematic View of a Monitor

ppen if one process in
needs to wait for some
cur and is unable to

Synchronization

- Synchronisation achieved by using **condition variables** that are contained within a monitor and only accessible within the monitor
- Condition va on by two functions:
  - cwait(c): Suspend execution g process on condition *c*
  - csignal(c) Resume execution of some process blocked after a cwait(c) on the same condition

Structure of a Monitor

Bounded Buffer Solution Using Monitor

Message Passing

- When processes interact with one another, there are two fundamental requirements:

synchroni	nication
• to enforce mutual exclusion	• ge n

- Message Passing is one approach to providing these functions
  - works with distributed systems *and* shared memory multiprocessor and uniprocessor systems

Message Passing

- The actual function of message passing is normally provided in the form of a pair of primitives:
  - send (destination, message)
  - receive (so
- **Exchange inf**
  - A process sends information in *message* to another process designated by a *destin*
  - A process receives information by executing the receive primitive, indicating the *source* and the *message*

Blocking send, Blocking receive

- Both sender and receiver are blocked until message is delivered
- **Synchronization**
  - Receiver the message is received
  - Sender cannot procee message arrives to the destination

Non-blocking Send, Non-blocking Receive

Nonblocking send, blocking receive

- sender continues on but receiver is blocked until the requested message arrives
- most useful
- sends one or more messages to various destinations as quickly as possible
- example -- a service process that wishes to provide a service or resource to other processes

Nonblocking send, nonblocking receive

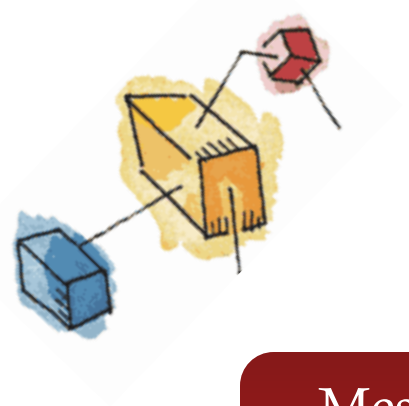
- neither party is required to wait

Addressing

- Sending process need to be able to specify which process should receive the message
  - Direct addressing
  - Indirect Add

Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive can be handled in one of two ways:
  - require that the process ex ignate a sending process
    - effective for cooperating concurrent processes
  - implicit addressing
    - source parameter of the receive primitive possesses a value returned when the receive operation has been performed



Indirect addressing

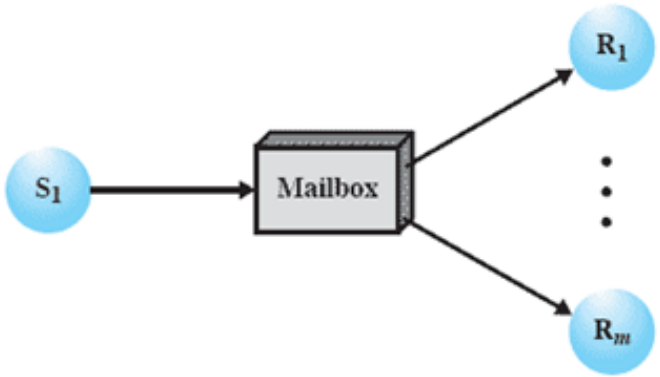
Messages are sent to a shared data structure consisting of queues that can temporarily hold messa

Queues are referred to as *mailboxes*

One process sends a message to the mailbox and the other process picks up the message from the mailbox

Allows for greater flexibility in the use of messages

Indirect Process Communication



(c) One to many

General Message Format

Readers/Writers Problem

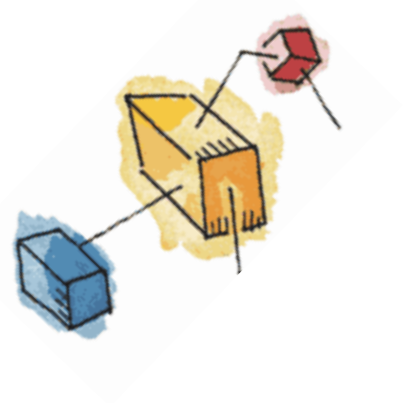
- A data area is shared among many processes
  - Some processes only read the data area (readers), some only write to the area (writers)
- Conditions t :
  1. Multiple re at once.
  2. Only one writer at a time
  3. If a writer is writing to the file, no reader may read it.

Readers have Priority

To ensure
1. only one writer can write at a time
2. no writer can write when there are readers

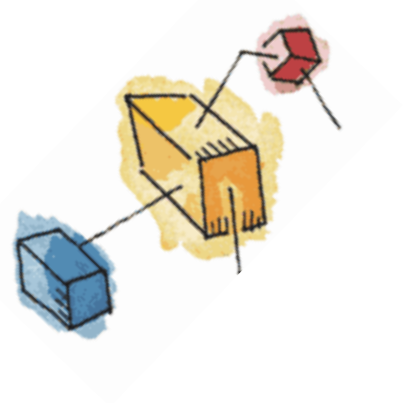
Readers continue to arrive & make readcount > 1 always!

Writers may get starved here!



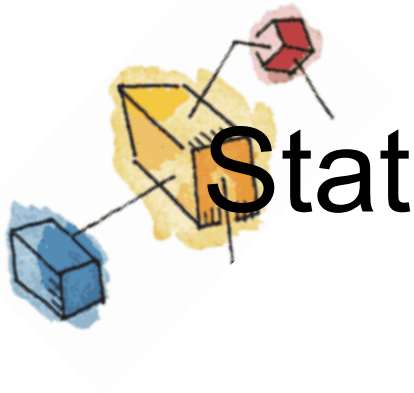
Writers have Priority

Readers and writers



Writers have Priority

"z" is used to make sure
only one reader is competing
"rsem" wi



State of the Process Queues

Summary

- Operating system themes are:
  - Multiprogramming, multiprocessing, distributed processing
  - Fundamental to these themes is concurrency
    - Issues of c ration arise
    - Effective s deadlock and starvation
- Mutual exclusion
  - Condition in which there is a set t processes, only one of which is able to access a given resource or perform a given function at any time
  - One approach involves the use of special purpose machine instructions – may not be efficient as it uses busy waiting

Summary

- Semaphores
  - Used for signalling among processes and can be readily used to enforce a mutual exclusion discipline
- Monitors
  - A programming construct that provides equivalent functionality to that of semaphores and is sometimes easier to control
- Messages
  - Useful for the enforcement of synchronization discipline
  - Exchange information