Justin Ciocoi

Nov. 8, 2023

CSCI 375 Slide Notes

Chapter 7: Synchronization Examples

- In the world of synchronization, there are a few classical problems which are often used to test newly-proposed synchronization schemes
- These include the following
  - *Bounded-Buffer Problem*
  - *Readers and Writers Problem*
  - *Dining Philosophers Problem*

Bounded-Buffer Problem

- In the bounded-buffer problem, there are n buffers, and each is capable of holding one item
  - The finite number of buffers here places a limit on how many items can be stored waiting for consumption
- Three semaphores are used
  - Semaphore `mutex` is initialized to 1
    - This is a *binary semaphore*, (0 or 1 *only*) which protects the buffer, ensuring mutual exclusion when producers or consumers access it to add or remove an item
  - Semaphore `full` is initialized to 0
    - This is a *counting semaphore*, which tracks the number of items in the buffer, ensuring that a consumer does not try to consume an empty buffer
  - Semaphore `empty` initialized to n
    - This is a *counting semaphore* which tracks the number of empty slots in the buffer, ensuring producers don't produce when the buffer is full

- The structure of a *writer process* is as follows

- ```
  do
  {
          ...
          //produce item in next produced
          ...
      wait(empty);
      wait(mutex);
          ...
          //add next produced to buffer
          ...
      signal(mutex);
      signal(full);
  }
  while(true);
  ```

    - First off, the producer will create the item to be placed in the buffer without accessing any shared data
    - `wait(empty)` waits for empty to be non-zero and then decrements it since it will be adding an item to the buffer, thus reducing the number of empty slots
    - Then, `wait(mutex)` ensures that the producer has exclusive access to the buffer before beginning to write to it
    - The writer process will then add an item to the buffer
    - `signal(mutex)` will allow other processes access to the buffer
    - `signal(full)` increments the full semaphore, signaling to all processes that the buffer is full

- The structure of a *consumer process* is as follows
  ```

- ```
  do
  {
      wait(full);
      wait(mutex);
          ...
      //remove an item from buffer to next consumed
          ...
      signal(mutex);
      signal(empty);
          ...
      //consume the item in next consumed
          ...
  }
  while(true);
  ```
    - Before it can remove an element from the buffer, the consumer process must first perform two wait operations
    - `wait(full)` waits for a producer process to call `signal(full)` indicating to consumers that there is a buffer ready to be consumed
    - `wait(mutex)` then ensures exclusive access to the buffer, after which the consumer will remove the item from the buffer
    - Once the item is removed, shared data access is no longer needed, so the consumer can call `signal(mutex)` and `signal(empty)`, allowing producers to produce into the buffer
- In the bounded-buffer problem, it is important to note that the `mutex` semaphore exists to provide exclusive access to the buffer, and the `full` and `empty` semaphores control the flow of production based on the state of the buffer (whether it is full or empty)

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
    - *Readers*, who only read the data set and do not perform any updates
    - *Writers*, who can both read *and* write to the shared data set
- The problem is allowing as many readers as want to access the shared data, but only allow one writer at a time
- There are several variations of how readers and writers are considered, and all involve some form of priority
  ```

- Shared Data
    - Data set
    - Semaphore `rw_mutex` initialized to 1
        - This semaphore ensures mutual exclusion for the writer processes by not allowing other writers or readers to access a dataset that is currently accessed by a writer
    - Semaphore `mutex` initialized to 1
        - This semaphore protects the `read_count` variable, which keeps track of the number of readers
    - Integer `read_count` initialized to 0
- The structure of a *writer process* is as follows
- ```
  do
  {
      wait(rw_mutex);
          ...
          //writing is performed
          ...
          signal(rw_mutex);
  }
  while(true);
  ```
    - `wait(rw_mutex);` waits for permission to write to the dataset, blocking if another writer is currently writing
    - Once `rw_mutex` is acquired the writer can safely write to the dataset
    - `signal(rw_mutex)` releases the `rw_mutex`, allowing other writers or readers to access the dataset
- The structure of a *reader process* is as follows
  ```

- ```
  do
  {
          wait(mutex);
          read_count++;
          if(read_count==1)
              wait(rw_mutex);
      signal(mutex);
          ...
          //Reading is performed
          ...
      wait(mutex);
          read_count--;
          if(read_count==0)
              signal(rw_mutex);
          signal(mutex);
  }
  while(true);
  ```
    - `wait(mutex)` waits for permission to modify the `read_count` variable
    - Once `mutex` is acquired, the reader process increments the read count
        - If this is the first reader, the process will also wait to acquire `rw_mutex` if it is not the first reader, it can assume the first reader had already waited for `rw_mutex` and subsequently blocked any writers until all readers have exited
    - Since `read_count` has now been incremented, *and* `rw_mutex` has been acquired, the reader can `signal(mutex)`, allowing other reader processes to modify the read count, and proceed to the critical section (reading)
    - Once reading is finished, the reader once again calls `wait(mutex)` such that it can safely decrement the `read_count` variable
    - If `read_count==0`, we know the last reader has exited the critical section, and can thus `signal(rw_mutex)` allowing writers to now enter their critical sections
- In this processes, the `while(true)` condition is used to indicate an infinite loop where readers and writers will indefinitely continue their operation
    - In real-world applications, this condition would likely be replaced with a more specific one, such as having the reader/writer exit after reading/writing a certain number of entries

Dining Philosophers Problem
  ```

- In this problem, we assume that philosophers (*processes*) alternate between eating (*executing critical section*) and thinking (*waiting for access to critical section*)
- These philosophers do *not* interact with their neighbors, but they will occasionally try to pick up two chopsticks (one at a time) to eat from the bowl (*shared data*)
    - They need both chopsticks to eat, and then will release both when they are done
- In the case of the 5 philosophers, there is the following shared data
    - Bowl of rice (*dataset*)
    - Semaphore `chopstick[5]` initialized to 1
- The structure of philosopher i is as follows:
- ```
  do
  {
      wait(chopstick[i]);
      wait(chopstick[(i+1)%]);

          //eat

      signal(chopstick[i]);
      signal(chopstick[(i+1)%5]);

          //think
  }
  while(true);
  ```
- However, there is a problem with this algorithm
    - Since philosophers pick up chopsticks one at a time, a deadlock can occur if all philosophers simultaneously pick up one chopstick
  ```

- There are a few ways to avoid the deadlock presented here
    - *Resource Hierarchy*, such that the philosophers must first acquire the lower numbered chopstick, followed by the higher numbered one
    - *Resource Allocation Limit*, such that the total number of philosophers trying to eat is limited
    - *Chopstick Timeout*, such that a philosopher picks up one chopstick, it will release it after a set amount of time if the other chopstick cannot be picked up
        - This is the method Professor Dogshit mentioned in class