OPERATING SYSTEMS

Introduction to Cooperating Processes

- Techniques
- Problems
- Solutions
  - Semaphore
  - Monitor

Introduction to Cooperating Processes

- Processes within a system may be independent or cooperating.
- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by other processes, including sharing data.
- Reasons for cooperating processes:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Cooperation among Processes by Sharing

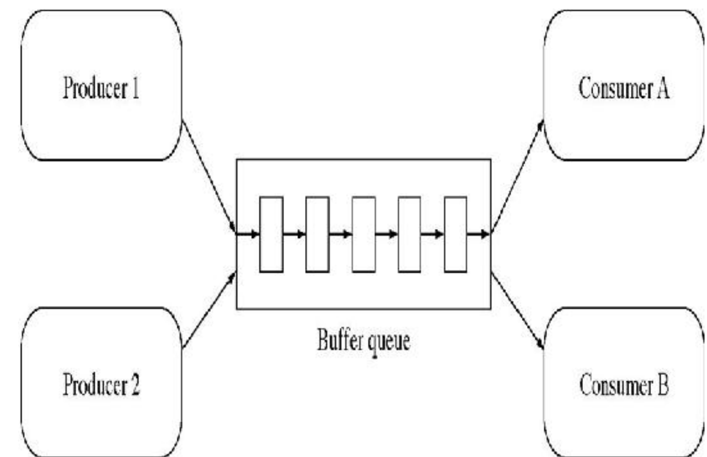
- Processes use and update shared data such as shared variables, memory, files, and databases.
- Writing must be mutually exclusive to prevent a race condition leading to inconsistent data views.
- Critical sections are used to provide this data integrity.
- A process requiring the critical section must not be delayed indefinitely; no deadlock or starvation.

Cooperation among Processes by Communication

- Communication by messages provides a way to synchronize, or coordinate, the various activities.
- Possible to have deadlock
  - each process waiting for a message from the other process.
- Possible to have starvation
  - two processes sending a message to each other while another process waits for a message.

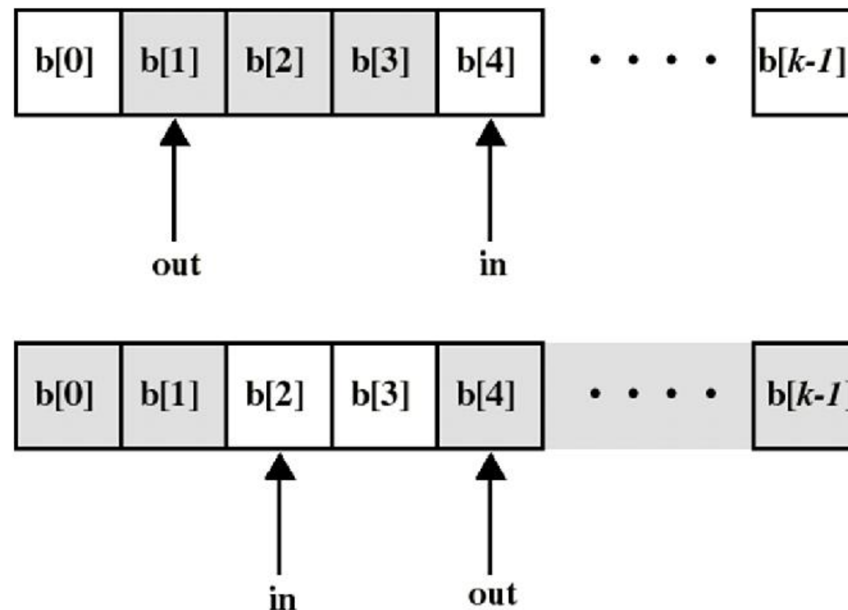
Producer/Consumer (P/C) Problem

- Paradigm for cooperating processes – *Producer* process produces information that is consumed by a *Consumer* process.
  - Example 1: a print program produces characters that are consumed by a printer.
  - Example 2: an assembler produces object modules that are consumed by a loader.
- There is need of buffer to hold items that are produced and later consumed:
  - unbounded-buffer places no practical limit on the size of the buffer.
  - bounded-buffer assumes that there is a fixed buffer size



Producer/Consumer Implementation

- The bounded buffer is implemented as a circular array with 2 logical pointers: **`in`** and **`out`**.
  - The variable **in** points to the next free position in the buffer.
  - The variable **out** points to the first full position in the buffer.



Problems with concurrent execution

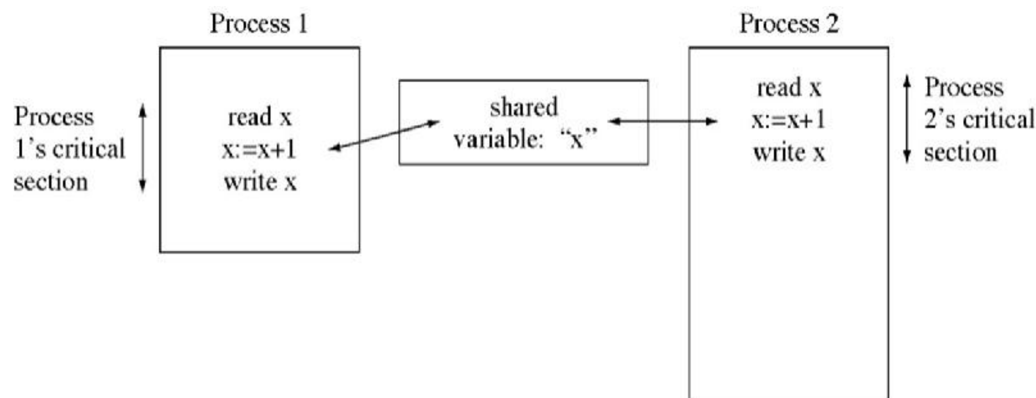
- Concurrent processes (or threads) often need to share data (maintained either in shared memory or files) and resources.
- If there is no controlled access to shared data, some processes will obtain an inconsistent view of this data.
- The action performed by concurrent processes will then depend on the order in which their execution is interleaved.

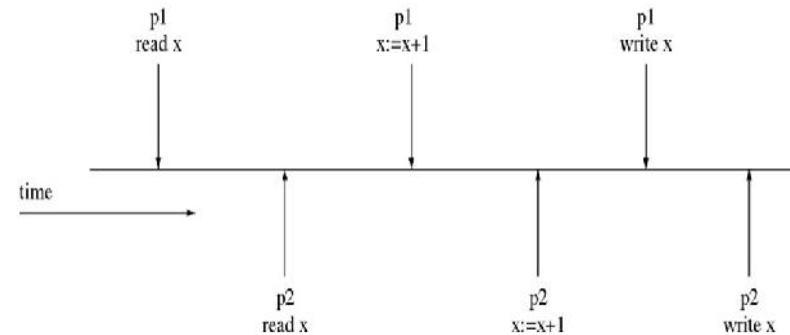
Data Consistency

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Solution to data consistency for consumer-producer problem can be provided by having an integer count that keeps track of the number of items in the buffer.
- Initially, the count is set to 0. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes a item.

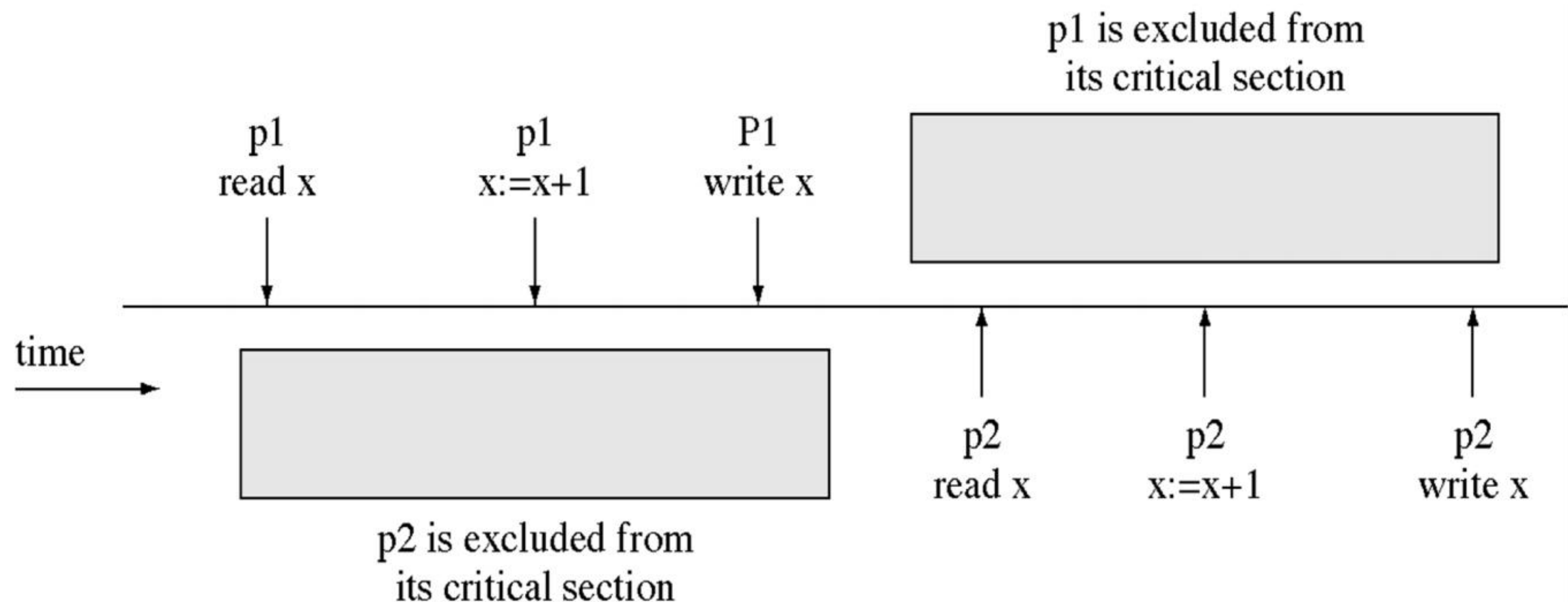
Race Condition

- Race condition: The situation where several processes access and manipulate shared data concurrently.
  - The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must coordinate or be synchronized.





Critical section to prevent a race condition



- Multiprogramming allows logical parallelism, uses devices efficiently but correctness is loosed when there is a race condition.
- So it is necessary to forbid logical parallelism inside critical section so some parallelism is loosed but correctness is regained.

Solution to Critical-Section Problem

- There are 3 requirements that must stand for a correct solution:
  1. **Mutual Exclusion**
  2. **Progress**
  3. **Bounded Waiting**
- All three requirements in each proposed solution, even though the non-existence of each one of them is enough for an incorrect solution.

Solution to CS Problem – Mutual Exclusion

1. **Mutual Exclusion** – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

- Implications:
  - Critical sections better be focused and short.
  - Better not get into an infinite loop in there.
  - If a process somehow halts/waits in its critical section, it must not interfere with other processes.

Solution to CS Problem – Progress

2. **Progress –** Processes should move towards its completion eventually
   - The process in the remainder section should execute normally
   - While, the process wish to enter into critical section should only participate for the decision
     - The decision for selecting a process to enter in critical section
   - i.e.
   - If only one process wants to enter, it should be able to.
   - If two or more want to enter, one of those should succeed.

Solution to CS Problem – Bounded Waiting

3. **Bounded Waiting** – A bound function
   - If a process requires to enter critical section while it is bypassed by other processes
   - The bound function should allow the bypassed process to enter in the critical section
   - Any process should not wait for a resource (critical section) for infinite time

Process failures problem

- If all 3 criteria (ME, progress, bounded waiting) are satisfied, then a valid solution will provide robustness against failure of a process in its remainder section (RS).
  - since failure in RS is just like having an infinitely long RS.
- However, no valid solution can provide robustness against a process failing in its critical section (CS).
  - A process Pi that fails in its CS does not signal that fact to other processes: for them Pi is still in its CS.

Synchronization Hardware

- Many systems provide hardware support for critical section code.
- Uniprocessors – could disable interrupts:
  - Currently running code would execute without preemption.
  - Generally too inefficient on multiprocessor systems.
- Modern machines provide special atomic (non-interruptible) hardware instructions:
  - Either test memory word and set value at once.
  - Or swap contents of two memory words.

Interrupt Disabling

- On a Uniprocessor: mutual exclusion is preserved but efficiency of execution is degraded: while in CS, we cannot interleave execution with other processes that are in RS.
- On a Multiprocessor: mutual exclusion is not preserved:
  - CS is now atomic but not mutually exclusive (interrupts are not disabled on other processors).

Special Machine Instructions

- Normally, access to a memory location excludes other access to that same location.
- Extension: designers have proposed machines instructions that perform 2 actions atomically (indivisible) on the same memory location. (e.g., reading and writing).
- The execution of such an instruction is also mutually exclusive (even on Multiprocessors).

Disadvantages of Special Machine Instructions

- Busy-waiting is employed, thus while a process is waiting for access to a critical section it continues to consume processor time.
- Starvation is possible when a process leaves a critical section and more than one process is waiting.
- They can be used to provide mutual exclusion but need to be complemented by other mechanisms to satisfy the bounded waiting requirement of the CS problem.

Semaphores

- Synchronization tool that does not require busy waiting.
- Data structure that is shared among all processes
- Semaphore is an integer variable and have two atomic and mutually exclusive functions to operate:
  - wait(S) (also down(S), P(S)), if S>0, assign thread to resource and decrement S. Else, put thread into queue
  - signal(S) (also up(S), V(S)), Increment S, if S>0 then de-queue a thread and call wait(S)
- Semaphore is set to N, when initialized (where N is number of threads treated by a resource at a time i.e. instances)
- Less Complicated

Semaphores

- Types:
  1. Binary semaphore – integer value can range only between 0 and 1 (really a Boolean)
     - For those resources which have only one instance
     - Have only two states:
       - Taken: Resource is taken, so threads are put in a queue
       - Not Taken: Resource is free, so a thread can be assigned from queue
  2. Counting semaphore – integer value can range over an unrestricted domain.
     - For those resources which have multiple instances
     - Can have several states, depends on no. of instances
     - If S is positive, means an instance of resource is free and no waiting threads are in queue
     - If S is zero, means all instances are being used and a threads can be put in queue
     - If S is negative, resources are busy and threads are waiting in queue.

Semaphores

- Must guarantee that two processes cannot execute wait() and signal() on the same semaphore at the same time.
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy-waiting in CS implementation:
    - But implementation code is short
    - busy-waiting possible if critical section rarely occupied.
- When a process must wait for a semaphore S, it is blocked and put on the semaphore's queue.
- Signal operation removes (assume a fair policy like FIFO) first process from the queue and puts it on list of ready processes.
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Deadlock and Starvation

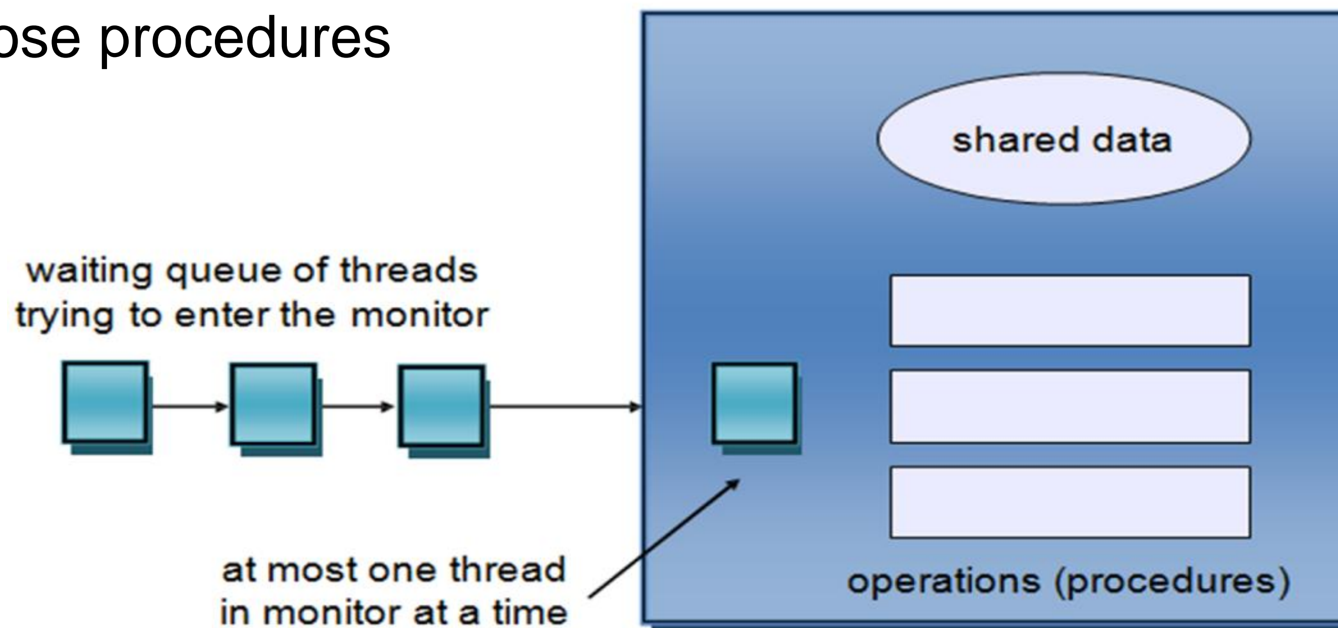
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of waiting processes.
- **Starvation** – indefinite blocking. A process may be removed after a very long time from the semaphore queue (say, if LIFO) in which it is suspended.
- **Priority Inversion** – scheduling problem when lower-priority process holds a lock needed by higher-priority process

Problems with Semaphores

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes.
- Usage must be correct in all the processes (correct order, correct variables, no omissions).
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- One bad (or malicious) process can fail the entire collection of processes.
- No control over usage or proper usage of resource.

Monitor

- It is better to use programming language support
- A *monitor* is a software module that encapsulates:
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent threads that invoke those procedures



Monitor

- Data for instances of resource can only be accessed from within the monitor
  - protects from unstructured access
- Synchronization code is added by compiler
- Addresses the key usability issues that arise with semaphores
- Automatic mutual exclusion, i.e. only one thread can be executing inside at any time
  - if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor

Monitor

- Once inside, a thread may discover it can’t continue, and may wish to block itself (or allow some other waiting thread to continue)
- It can wait and signal others to continue using Condition Variable
  - Condition variables can only be accessed from within monitor
  - A thread that waits “steps outside” the monitor (onto a wait queue associated with that condition variable)

Monitors

Condition Variable

- A place to wait; sometimes called a rendezvous point
- Three operations on condition variables
  - wait(c)
    - release monitor lock
    - Put a thread into associated wait queues w.r.t c
    - Allow another thread
  - signal(c)
    - wake up at most one waiting thread for condition c
    - if no waiting threads, signal is lost
  - broadcast(c)
    - wake up all waiting threads in c's waiting queue
    - Useful to notify all threads waiting for condition c (usually if condition has became invalid)
- Monitor Lock is a procedure to acquire or relinquish monitor
- and c posses a condition, such as no. of character in keyboard buffer etc.

Questions