Process Synchronization

Alok Jhaldiyal
Assistant Professor
SoCSE, UPES

Why needed?

- Processes may need to share data
  - More than one process reading/writing the same data (a shared file, a database record,…)
  - Output of one process being used by another
  - Needs mechanisms to pass data between processes
- Ordering executions of multiple processes may be needed to ensure correctness
  - Process X should not do something before process Y does something etc.
  - Need mechanisms to pass control signals between processes

Interprocess Communication (IPC)

- Mechanism for processes P and Q to communicate and to synchronize their actions Establish a communication link
  - Fundamental types of communication links
  - Shared memory
    - P writes into a shared location, Q reads from it and vice-versa
  - Message passing
    - P and Q exchange messages

Producer Consumer Problem

- Paradigm for cooperating processes
- *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.
- Basic synchronization requirement
  - Producer should not write into a full buffer
  - Consumer should not read from an empty buffer
  - All data written by the producer must be read exactly once by the consumer

Shared memory

- **Shared data**

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded Buffer Producer Process

```
item nextProduced;

while (1) {
while (((in + 1) % BUFFER_SIZE) == out)
    ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer Consumer Process

```
item nextConsumed;

while (1) {
        while (in == out)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
- Especially challenging with preemptive kernels

Critical Section

- Entry section: a piece of code executed by a process just before entering a critical section
- Exit section: a piece of code executed by a process just after leaving a critical section
- General structure of a process *Pi*

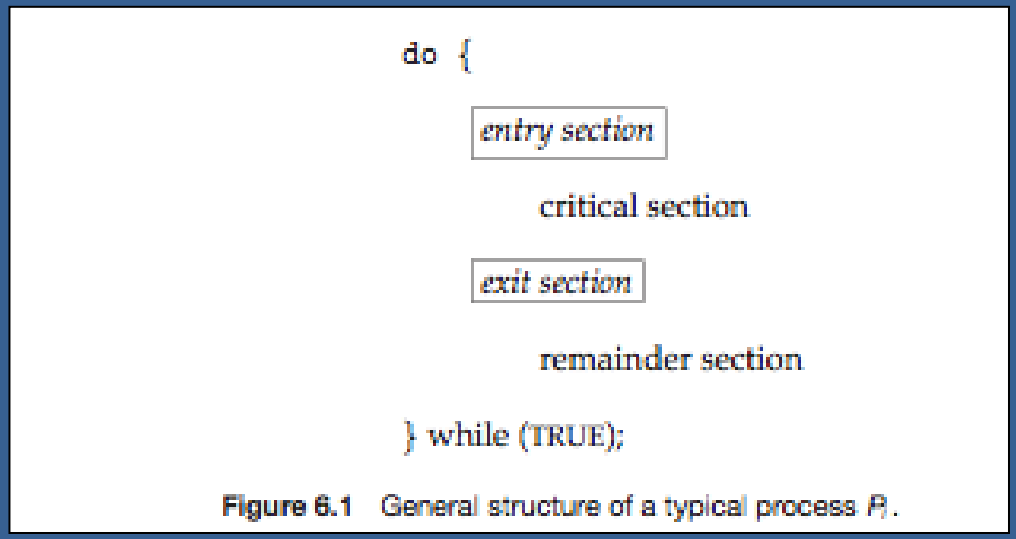


Figure 6.1 General structure of a typical process *Pi*.

Critical Section Solution

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the n processes

Mutex Locks

- Simplest synchronization tool.
- Mutex lock helps to protect critical regions and thus prevent race conditions.
- Process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- acquire() function acquires the lock, and the release() function releases the lock.
- Mutex lock has a boolean variable available whose value indicates if the lock is available or not.

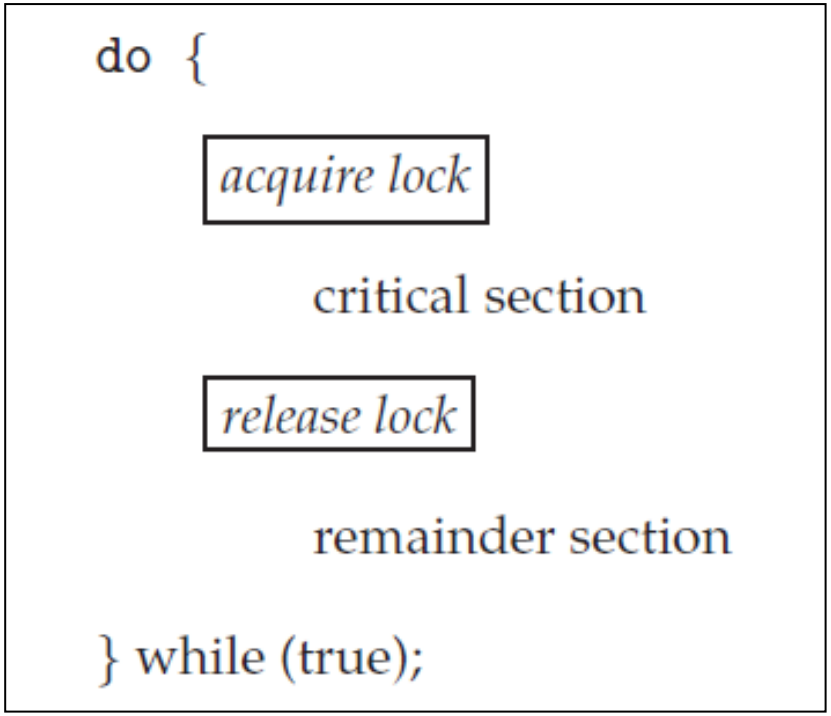
Implementation Mutex Locks

- acquire()

```
acquire() {
while (!available)
; /* busy wait */
available = false;;
}
```

- release()

```
release() {
available = true;
}
```



Busy Waiting

- Main disadvantage of using mutex lock is busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
- Continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared.
- Busy waiting wastes CPU cycles that some other process might be able to use productively

Semaphores

- Widely used synchronization tool
- Does not require busy-waiting
  - CPU is not held unnecessarily while the process is waiting
- A Semaphore S is
  - A data structure with an integer variable S.value and a queue S.q of processes
  - The data structure can only be accessed by two atomic operations, wait(S) and signal(S) (also called P(S) and V(S))
- Value of the semaphore S = value of the integer S.value

Wait and Signal Operations

- *wait* (*S*):

  if (S.value > 0) S.value --;

  else {

  add the process to S.q;

  block the process;

  }

- *signal* (*S*):

  if (S.q is not empty)

  choose a process from S.q and unblock it

  else S.value ++;

Semaphores Types

- Semaphores can be:
  - Counting Semaphore
  - Binary Semaphore
- Value of binary semaphore can range only between 0 and 1, behave similarly to mutex locks.
- Value of a counting semaphore can range over an unrestricted domain.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.

Semaphore Usage

- Each process that wishes to use a resource performs a wait() operation on the semaphore.
  - Also decrements the semaphore count.
- When a process releases a resource, it performs a signal() operation.
  - Increments the count
- When the count for the semaphore goes to 0, all resources are being used.
  - Processes that wish to use a resource will block until the count becomes greater than 0

Semaphore Implementation

- Semaphore operations is evolved approach and have overcome the need of busy waiting.
- This is implemented by modifying the basic definition of wait() and signal().
- Process executes the wait() operation and finds that the semaphore value is not positive, it must wait.
- Rather than engaging in busy waiting, the process can block itself.
- Block operation places a process into a waiting queue associated with the semaphore

Semaphore Implementation

- Once the process enters a semaphore waiting queue, control is transferred to the CPU scheduler.
- CPU scheduler picks another process from ready queue and assigns CPU to it.
- Process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- Process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state

Race Condition

- A scenario in which the final output is dependent on the relative speed of the processes
  - Example: The final value of the shared data *counter* depends upon which process finishes last
- Race conditions must be prevented
  - Concurrent processes must be synchronized
  - Final output should be what is specified by the program, and should not change due to relative speeds of the processes.

Atomic Operation

- An operation that is either executed fully without interruption, or not executed at all
  - The "operation" can be a group of instructions
    - Ex. the instructions for counter++ and counter--
- Note that the producer-consumer problem's solution works if *counter++* and *counter--* are made atomic
- In practice, the process may be interrupted in the middle of an atomic operation, but the atomicity should ensure that no process uses the effect of the partially executed operation until it is completed

Race Condition

Suppose that two processes A and B have access to a shared variable "Balance":

PROCESS A:	PROCESS B:
Balance = Balance - 100	Balance = Balance - 200

Further, assume that Process A and Process B are executing concurrently in a time-shared, multi-programmed system.

Concurrent Access to Shared Data

- The statement " Balance = Balance – 100" is implemented by several machine level instructions such as:
  - A1. LOAD R1, BALANCE // load Balance from memory into Register 1 (R1)
  - A2. SUB R1, 100 // Subtract 100 from R1
  - A3. STORE BALANCE, R1 // Store R1's contents back to the memory location of Balance.
- Similarly, "Balance = Balance – 200" can be implemented by the following:
  - B1. LOAD R1, BALANCE
  - B2. SUB R1, 200
  - B3. STORE BALANCE, R1

Race Condition

- Scenario 1:
- A1. LOAD R1, BALANCE
- A2. SUB R1, 100
- A3. STORE BALANCE, R1

  Context Switch!
- B1. LOAD R1, BALANCE
- B2. SUB R1, 200
- B3. STORE BALANCE, R1

Balance is effectively decreased by 300!

- Scenario 2:
- A1. LOAD R1, BALANCE
- A2. SUB R1, 100

  Context Switch!
- B1. LOAD R1, BALANCE
- B2. SUB R1, 200
- B3. STORE BALANCE, R1

  Context Switch!
- A3. STORE BALANCE, R1

Balance is effectively decreased by 100!

Race Conditions

When multiple processes are accessing shared data without access control the final result depends on the execution order creating what we call race conditions.

- **A serious problem for any concurrent system using shared variables!**
- **We need Access Control using code sections that are executed atomically.**
- **An Atomic operation is one that completes in its entirely without context switching (i.e. without interruption).**

Readers Writers Problem

- Classic problem for which design of synchronization and concurrency mechanisms can be tested.
- Other Problems:
  - Producer/Consumer problem
  - Dining Philosophers Problem

Readers/Writers Problem Defination

- There is a data area that is shared among a number of processes.
- Any number of readers may simultaneously write to the data area.
- Only one writer at a time may write to the data area.
- If a writer is writing to the data area, no reader may read it.
- If there is at least one reader reading the data area, no writer may write to it.
- Readers only read and writers only write
- A process that reads and writes to a data area must be considered a writer.

Home Work

- **Suggest a solution for reader/writers problem using semaphore.**