Process Synchronization

Topics:
Monitors
Simulation of Monitors
Checkpoint and Recovery

Monitors
High-level synchronization construct that allows the safe sharing of an abstract data-type using concurrent processes
class monitor-name {
variable declarations
method declarations
initialization
}

Monitors (continued)
To allow a process to wait within a monitor a condition variable must be declared as condition x,y;
The operation x.wait means that the process invoking the command waits (outside the monitor)
The operation x.signal resumes exactly one suspended process.
Dining Philosopher Example

monitor dining_philosopher {
int state[5] /*thinking, hungry and eating*/
condition self[5];
method pickup(int I) {state[I]=hungry;test(I); if (state[I]<>eating) self[I].wait();}
method putdown(int I) {
state[I]=thinking;test((I-1)%5);test((I+1)%5);}
}

Dining Philosopher Example (continued)

Method test(int k) { if (state[(k-1)%5] <> eating and state[k]==hungry and state[(k+1)%5]<>eating) {state[k]=eating;self[k].signal;}
Method init { for (I=0;I<5;I++) state[I]=thinking;}

Property of Monitor

- At most one process is inside the monitor
- We need to have mutual exclusion inside a monitor.
- When a process is blocked, it is made to wait outside the monitor.
- No process is assumed to die inside a monitor.
Monitor Implementation Using Semaphore

Variables: Semaphore
mutex(=1), next(=0);
int next-count(=0);
Each method F (inside a monitor will be replaced by:
wait(mutex); F; if (next-count>0)
signal(next); else signal(mutex);
Mutual Exclusion within a monitor is ensured

Monitor Implementation Using Semaphore

For each condition variable x, we have:
Semaphore x-sem(=0); int x-count(=0);
The operation x.wait can be implemented as:
x-count++; if (next-count > 0) signal(next)
else signal(mutex); wait(x-sem); x-count--;
Value of priority stored with the name of the process that is suspended.

Solaris 2 Operating System

Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
Uses adaptive mutexes for efficiency when protecting data from short-code segments
Uses condition variables and readers-writers locks when longer sections of code need to access data.
Atomic Transactions

Transaction - program unit that must be executed atomically; i.e., either all the operations associated with it are executed to completion or none are performed.

Must preserve atomicity despite possibility of failure.

Ensuring atomicity in an environment where failures result in the loss of information on volatile storage.

Log-Based Recovery

Write-ahead log - all updates are recorded on the log, which is kept in stable storage; log has following fields: 1:Transaction name 2:Data item name, old value, new value

The log has a record of <T_j starts> and either <T_j commits> or <T_j aborts.

Log-Based Recovery

Recovery algorithm uses two procedures:

undo(T_j) - restores value of all data updated by transaction T_j to the old values (log does not contain <T_j commits> but contains <T_j starts>)
redo(T_j)
Checkpoints-Reduce Recovery Overhead

- Output all log records currently residing in volatile storage onto stable storage.
- Output all modified data residing in volatile storage to stable storage.
- Output log-record <checkpoint> onto stable storage: Recovery routine examines log to determine the most recent transaction \( T_j \) that started executing before the most recent checkpoint took place.

Concurrent Atomic Transactions

1. Serial Schedule - the transactions are executed sequentially in some order.
   E.g., Transaction_0: read(a), write(a), read(b), write(b)
   Transaction_1: read(a), write(a), read(b), write(b)
   Transaction_0 is executed first

Concurrent Atomic Transactions

- Conflicting Operations - \( O_i \) and \( O_j \) conflict if they access the same data item, and at least one of the operations is a write operation
- Conflict Serializable schedule: schedule that can be swapped into a serial schedule by a series of non-conflicting operations
Example (continued)

- sequence: \( \{ \text{read}(a), \text{write}(a) \} T_0, \{ \text{read}(a), \text{write}(a) \} T_1, \{ \text{read}(b), \text{write}(b) \} T_0, \{ \text{read}(b), \text{write}(b) \} T_1 \)

- Locking Protocols: How locks are acquired and released.
  - Shared: \( T_I \) has obtained a lock for item \( Q \), then it can read but not write.
  - Exclusive: read and write \( Q \)

Locking Protocol

- Two phase locking protocol:
  - Growing phase: A transaction may obtain locks but not release any.
  - Shrinking phase: A transaction may release locks but not acquire new ones.
  - This protocol ensures conflict serializability but not dead-locks

Time Stamp Protocol

- Time-stamp ordering scheme - transaction ordering for determining serializability order. With each transaction \( T_j \) in the system, associate a unique fixed time stamp, denoted by \( TS(T_j) \). For any new transaction \( T_k \) that comes after \( T_j \), \( TS(T_k) > TS(T_j) \).
  - For each data item value, assign two timestamp values - \( W\text{-timestamp}(Q) \) - the largest timestamp of any transaction that executed \( Q \) successfully.