Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- 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
  - Multiprocess machine, spin or block
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

6.9: Atomic Transactions

- Introduce notions of databases into operating systems
  - Challenge is that some of these operations are “heavy” and not necessarily fast
- Transaction:
  - A collection of operations that performs a single logical function. For example, changing the state and moving the process from waiting to ready state is one transaction
  - Transactions are atomic with all or nothing semantics
    - Committed transactions means, all the operations went through
    - Aborted transactions means, none of them went through
    - You cannot be in a middle state, e.g., changed state, removed it from waiting state but didn’t add to ready state
    - When a transaction aborts, we roll back

Storage states

- Storage to implement transactions:
  - Volatile storage: Does not survive system crash
  - Nonvolatile storage: Survives system crashes
  - Stable storage: Information is “never” lost. Uses nonvolatile storage and replication
- Log-based recovery:
  - Write-ahead logging, where we write all operations into a log in stable storage
    - <transaction name, data item name, old value, new value>
  - Transaction is made up of
    - <Ti, starts> set of transaction logs <Ti, commit>
    - If both starts and commit is there, then the transaction is committed. Else, it is rolled back
    - Logs are idempotent, you can apply it again and again in the same order without side effects

Checkpoints

- Logs keep growing. After every failure, we'd have to go back and replay the log. This can be time consuming.
- Checkpoint frequently
  - Output all log records currently in volatile storage onto stable storage
  - Output all modified data residing in volatile storage to the stable storage
  - Output a log record <checkpoint> into stable storage
- On failure, search backwards till we hit the first checkpoint. The first transaction start from the checkpoint (going back) is the start of replay

Serializability

- Transactions can be concurrent. Such concurrency may cause problems depending on the interleaving of the transactions. We introduce stricter notions of this phenomenon in order to predict system behavior
- Schedule is an execution sequence
- Serial schedule: Schedule where two concurrent transactions follow one after the other
  - For two transactions T1, T2: serial schedule is T1 then T2 or T2 then T1. For n transactions, we have n! choices, all of which is valid
  - Serial schedule cannot fully utilize the system resources and so we want to relax the schedule: non-serial schedule

Conflict

- We define a schedule to be in conflict if they both operate on the same data item and one of the operations is a write
- If there is no conflict, the schedule can be swapped.
- If after non-conflicting swaps we reach a serial schedule, then that schedule is called conflict serializable

Read(A)
Write(A)
Read(B)
Write(B)
read(A)
write(A)
read(B)
write(B)

Serial schedule

Read(A)
Write(A)
read(A)
write(A)
Read(B)
Write(B)
read(B)
write(B)

Conflict serializable schedule

Locking protocol to enforce order

- Shared: Transaction can read but not write
- Exclusive: Transaction can read and write

- Two phase protocol to ensure serializability:
  - Growing phase - transaction can obtain but not release locks
  - Shrinking phase - transaction can release lock but not acquire new ones

  - Ensures conflict serializability but is not free from deadlocks

Timestamp-based Protocols

- Timestamp transactions: Can be real wall clock time or logical clock
- The timestamp determines the serializability order
- For each data item (Q), associate two timestamps
  - W-timestamp denotes largest timestamp of any transaction that successfully executed write(Q).
  - R-timestamp for read(Q)
- Suppose Ti issues read(Q):
  - If TS(Ti) < W-timestamp(Q), rollback Ti
  - If TS(Ti) >= W-timestamp(Q), execute Ti, R-timestamp = maximum (R-timestamp(Q) and TS(Ti))
- Suppose Ti issues write(Q):
  - If TS(Ti) < R-timestamp(Q), rollback Ti
  - If TS(Ti) < W-timestamp(Q), rollback Ti
  - Execute write

Schedule possible under Timestamp

Read(B)

read(B)

write(B)

Read(A)

read(A)

write(A)