Frequently asked questions from the previous class survey

- Spinlock: Just busy waiting?
- Turnstile ultimately becomes a queue for threads blocked on a lock? Grows and decreases in size?
- Difference between blocking, waiting, and sleeping
- When would you use atomic integer operations?
- Log-based recovery
  - If T4 and T5 are in the same file, does T5 go first and then T4?
  - Order of redo and undo after a checkpoint
  - Checkpoints: In memory or disk?
- Importance of journaling in file systems

Topics covered in today’s lecture

- Atomic Transactions
  - Locking protocols
  - Timestamp protocols
- Deadlocks
- Deadlock characterization

Locking protocol governs how locks are acquired and released

- There are different modes in which data can be locked
  - A transaction acquires a lock on a data item in different modes
- Shared mode locks
  - \( T_i \) can read, but not write, data item \( Q \)
- Exclusive mode locks
  - \( T_i \) can read and write data item \( Q \)

Transactions must request locks on data items in the right mode

- To access data item \( Q \), \( T_i \) must first lock it
  - Wait if \( Q \) is locked in the exclusive mode
    - If \( T_i \) requests a shared-lock on \( Q \)
      - Obtain lock if \( Q \) is not locked in the exclusive mode
- \( T_i \) must hold lock on data item as long as it accesses it
Two-phase locking protocol: Locks and unlocks take place in two phases

- **Transaction’s growing phase:**
  - Obtain locks
  - Cannot release any lock

- **Transaction’s shrinking phase:**
  - Can release locks
  - Cannot obtain any new locks

Two-phase locking protocol: Conflict serializability

- Conflicts occur when 2 transactions access same data item; and 1 of them is a write
- A transaction acquires locks serially; **without** releasing them during the acquire phase
- Other transactions **must** wait for first transaction to start releasing locks.
- Deadlocks may occur

Order of conflicting transactions

- Two-phase locking
  - Determined at **execution time**

- How about selecting this order in **advance**?
  - **Timestamp based protocols**

Timestamp based protocols

- For each $T_i$, there is a fixed timestamp
  - Denoted $TS(T_i)$
  - Assigned before $T_i$ starts execution

- For a later $T_j$; $TS(T_i) < TS(T_j)$

- Schedule must be equivalent to schedule in which $T_i$ appears before $T_j$.

Each data item $Q$ has two values

- $W$-timestamp($Q$)
  - Largest timestamp of any transaction that successfully executed write()

- $R$-timestamp($Q$)
  - Largest timestamp of any transaction that successfully executed read()
Transaction issues a `read(Q)`

- If `TS(T_i) < W-timestamp(Q)`
  - Needs value that was already overwritten
  - The read is rejected and `T_i` is rolled back
- `TS(T_i) >= W-timestamp(Q)`
  - Operation is executed
  - `R-timestamp(Q) = max(TS(T_i), R-timestamp(Q))`

Transaction issues a `write(Q)`

- If `TS(T_i) < R-timestamp(Q)`
  - Value of `Q` produced by `T_i` needed previously
  - `T_i` assumed that this value would never be produced
  - The write is rejected and `T_i` is rolled back
- If `TS(T_i) < W-timestamp(Q)`
  - Trying to write an obsolete value of `Q`
  - The write is rejected and `T_i` is rolled back

What happens when a transaction is rolled back?

- Transactions `T_j` is assigned a new timestamp
- Restart

Schedule using the timestamp protocol:

- Transactions `T_2` and `T_3`
  - `T_2`: `read(B)`, `read(A)`
  - `T_3`: `write(B)`, `write(A)`

Timestamps are assigned to transactions before the start of the first instruction `TS(T_2) < TS(T_3)`

The Journey So Far ...

- Multiprogramming
- Processes
- Interprocess Communications
- Threads
- CPU Scheduling
- Synchronization & Coordination
- Deadlocks

DEADLOCKS

A waiting process is never again able to change state if it is waiting for resources held by other processes.
What we will look at …

Deadlocks

- Prevention
- Characterization
- Avoidance
- Detection & Recovery

Why?

System Model

Requirements

For many applications, processes need exclusive accesses to multiple resources

- Process A: Asks for scanner and is granted it
- Process B: Asks CD recorder first and is granted it.
- Process A: Now asks for CD recorder
- Process B: Now asks for Scanner

Both processes are blocked and will remain so forever!

Deadlock

Resource Deadlocks

- Major class of deadlocks involves resources
- Can occur when processes have been granted access to devices, data records, files, etc.
- Other classes of deadlocks: communication deadlocks, two-phase locking

Related concepts

- Livelocks and starvation

Other deadlock situations

- Distributed systems involving multiple machines
- Database systems
  - Process 1 locks record R1
  - Process 2 locks record R2
  - Then, processes 1 and 2 try to lock each other’s record
    - Deadlock
- Deadlocks can occur in hardware or software resources

Preemptable resources

- Can be taken away from process owning it with no ill effects
- Example: Memory
  - Process B’s memory can be taken away and given to process A
  - Swap B from memory, write contents to backing store, swap A in and let it use the memory

Non-preemptable resources

- Cannot be taken away from a process without causing the process to fail
- If a process has started to burn a CD
  - Taking the CD-recorder away from it and giving it to another process?
    - Garbled CD
    - CD recorders are not preemptable at an arbitrary moment
- In general, deadlocks involve non-preemptable resources
Some notes on deadlocks

- The OS typically does not provide deadlock prevention facilities
- Programmers are responsible for designing deadlock free programs

System model

- Finite number of resources
  - Distributed among competing processes
- Resources are partitioned into different types
  - Each type has a number of identical instances
  - Resource type examples: Memory space, files, I/O devices

A process must utilize resources in a sequence

- Request
  - Requesting resource must wait until it can acquire resource
  - request(), open(), allocate()
- Use
  - Operate on the resource
- Release
  - release(), close(), free()

For kernel managed resources, the OS maintains a system resource table

- Is the resource free?
  - Record process that the resource is allocated to
- Is the resource allocated?
  - Add to queue of processes waiting for resource
- For resources not managed by the OS
  - Use wait() and signal() on semaphores

Deadlock: Formal Definition

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

- Because all processes are waiting, none of them can cause events to wake any other member of the set
- Processes continue to wait forever

Deadlock Characterization
Deadlocks: Necessary Conditions (I)

- **Mutual Exclusion**
  - At least one resource held in nonsharable mode
  - When a resource is being used
    - Another requesting process must wait for its release

- **Hold-and-wait**
  - A process must hold one resource
  - Wait to acquire additional resources
    - Which are currently held by other processes

Deadlocks: Necessary Conditions (II)

- **No preemption**
  - Resources cannot be preempted
  - Only voluntary release by process holding it

- **Circular wait**
  - A set of \( \{P_0, P_1, ..., P_n\} \) waiting processes must exist
    - \( P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow ... \rightarrow P_n \rightarrow P_0 \)
  - Implies hold-and-wait

Resource allocation graph

- Used to describe deadlocks precisely
- Consists of a set of vertices and edges
- Two different sets of nodes
  - \( P \): the set of all active processes in system
  - \( R \): the set of all resource types in the system

Directed edges

- **Request edge**
  - \( P_i \) has requested an instance of resource type \( R_j \)
  - Directed edge from process \( P_i \) to resource \( R_j \)
  - Denoted \( P_i \rightarrow R_j \)
  - Currently waiting for that resource

- **Assignment edge**
  - Instance of resource \( R_j \) assigned to process \( P_i \)
  - Directed edge from resource \( R_j \) to process \( P_i \)
  - Denoted \( R_j \rightarrow P_i \)

Representation of Processes and Resources

- Processes
- Resources
  - A resource type may have multiple instances

Resource Allocation Graph example

- \( R_1 \)
- \( R_3 \)
- \( P_1 \)
- \( P_2 \)
- \( P_3 \)
- \( R_2 \)
- \( R_4 \)

- Request Edge
- Assignment Edge
Determining deadlocks

- If the graph contains no cycles?
  - No process in the system is deadlocked

- If there is a cycle in the graph?
  - If each resource type has exactly one instance
    - Deadlock has occurred
  - If each resource type has multiple instances
    - A deadlock may have occurred

Resource Allocation Graph:

<table>
<thead>
<tr>
<th>Cycle but not a deadlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁ (\rightarrow) R₁ (\rightarrow) P₂ (\rightarrow) R₂ (\rightarrow) P₁</td>
</tr>
<tr>
<td>P₄ may release instance of R₂ allocate to P₂ and break cycle</td>
</tr>
</tbody>
</table>

Methods for handling deadlocks

- Use protocol to prevent or avoid deadlocks
- Ensure system never enters a deadlocked state
- Allow system to enter deadlocked state; BUT
  - Detect it and recover
- Ignore problem, pretend that deadlocks never occur

Problems with undetected deadlocks

- Resources held by processes that cannot run
- More and more processes enter deadlocked state
  - When they request more resources
- Deterioration in system performance
  - Requires restart
When is ignoring the problem viable?

- When they occur infrequently (once per year)
  - Ignoring is the cheaper solution
  - Prevention, avoidance, detection and recovery
  - Need to run constantly

Law passed by Kansas Legislature ... early 20th Century

“When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone”

Dining philosophers problem: Necessary conditions for deadlock (1)

- Mutual exclusion
  - 2 philosophers cannot share the same chopstick
- Hold-and-wait
  - A philosopher picks up one chopstick at a time
  - Will not let go of the first while it waits for the second one

Dining philosophers problem: Necessary conditions for deadlock (2)

- No preemption
  - A philosopher does not snatch chopsticks held by some other philosopher
- Circular wait
  - Could happen if each philosopher picks chopstick with the same hand first

Is there a traffic deadlock here?
The traffic scenario: Necessary Conditions (1)

- Mutual Exclusion
  - A vehicle needs its own space
  - We can't stack automobiles on top of each other

- Hold-and-wait
  - A vehicle does not move and stays in place if it cannot advance

The traffic scenario: Necessary Conditions (2)

- No preemption
  - We cannot move an automobile to the side

- Circular-wait
  - Each vehicle is waiting for the one in front of it to advance

DEALING WITH DEADLOCKS

Four strategies for dealing with deadlocks

- Ignore the problem
  - May be if you ignore it, it will ignore you

- Detection and Recovery
  - Let deadlocks occur, detect them, and take action

- Deadlock avoidance
  - By careful resource allocation

- Deadlock prevention
  - By structurally negating one of the four required conditions

The contents of this slide-set are based on the following references
