Ensuring Conflict Serializability

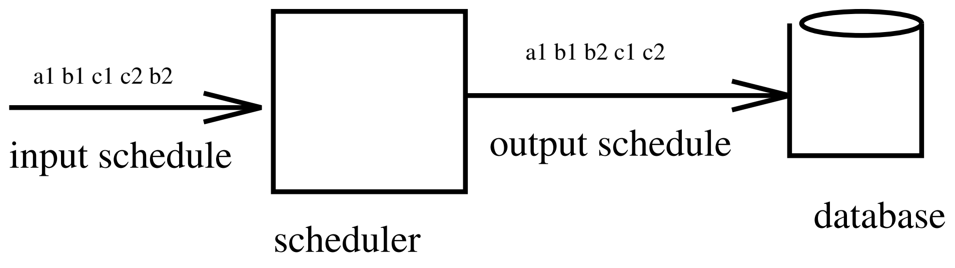
Topics:

- Schedulers.
- Basic Two Phase Locking.
- Non-locking schedulers– timestamping, SGT, optimistic protocols.

Readings:

- Kung and Robinson, On optimistic concurrency control, Chapter 3, Stonebraker.
- R. Agarwal et. al., Models of studying concurrency control performance: alternatives and implications, SIGMOD '85.
- Escrow Locking paper.
- Survey on deadlocks in distributed systems.

Schedulers



- A scheduler ensures that the schedules resulting from the concurrent execution of transactions is serializable.

- To process an operation o_i a scheduler may:
  - Execute o_i immediately.
  - Reject o_i causing transaction T_i to abort.
  - Delay the execution of o_i.

- The degree of concurrency permitted by the scheduler is the set of input schedules it can process without delaying or rejecting any operation.

Basic Two Phase Locking

- With each data item, two types of locks are associated–(S)hared and e(X)clusive.
- Lock Compatibility Matrix:

	Granted Mode		
requested mode	none	S	X
S	✓	✓	
X	✓		

- Locks that are not compatible are said to conflict.
- To read x, transaction T_i requests an S mode lock on x.
- To write x, transaction T_i requests an X mode lock on x.
- Lock request of T_i is granted, if no other transaction T_j holds a conflicting lock.
- Else, if T_j holds a conflicting lock, T_i is delayed.
- T_i must hold an approppriate lock during the execution of the operation.

Basic Two Phase Locking

- *Once a transaction releases any lock it acquired, it does not acquire locks on any data item.*
- Thus, each transaction has a *growing phase* in which it acquires all the locks it needs. It then has a *shrinking phase* in which it releases all locks.
- The following schedule could be generated by the 2PL protocol:a

$$w_1(x) \;\; w_2(x) \;\; w_2(y)$$

- However, the following schedule could not be generated by the 2PL protocol:

$$w_1(x) \;\; w_2(x) \;\; w_3(y) \;\; w_1(y)$$

Correctness of Two Phase Locking

If each transaction follows the 2PL protocol, the resulting schedules are CSR.

Proof: by contradiction.

- Let S be a schedule generated using the 2PL protocol that is not serializable.
- Since S not serializable, SG contains a cycle $T_1, T_2, \ldots T_n, T_1$.
- T_1 contains an operation o_1 that conflicts with some operation o_2 in T_2.
- Hence T_1 releases some lock before T_2 acquires all its locks.
- Since T_1 is two-phased, T_1 acquires all its locks before T_2 acquires all its locks.
- Similarly, T_2 acquires all its locks before T_3, ..., T_n acquires all its locks before T_1.
- Hence, T_1 acquires all its locks before it acquires all its locks– a contradiction!

Distributed Two-Phase Locking

- A distributed DBMS consists of multiple different sites each with its own scheduler.
- The schedulers at each site, taken together, is a distributed scheduler.
- Each scheduler maintains locks for the data on its site.
- A transaction T_i to execute an operation o_i at site s_j must request an appropriate lock from the scheduler at s_j.
- *T_i does not release locks at any of the sites it executes until it has acquired all the locks it needs at every site it executes on.*
- To ensure this property, the schedulers at different sites need to synchronize.

Timestamp Ordering

- Each transaction T_i when it begins execution it requests a timestamp $TS(T_i)$.
- No two transactions have the same timestamp.
- With each data item x two timestamps are associated:
  - **read-timestamp**: the largest timestamp of any transaction that executed a read operation on x.
  - **write-timestamp**: the largest timestamp of any transaction that executed a write operation on x.
- These timestamps are updated whenever a new read and a write operation on x is performed.
- The timestamp protocol ensures that any conflicting read and write operations are executed in the timestamp order:
- A $r_i(x)$ is permitted if $TS(T_i)$ is greater than or equal to the write timestamp of x. Else, transaction T_i is aborted.
- A $w_i(x)$ is permitted if $TS(T_i)$ is greater than than or equal the read timestamp and write timestamp of x. Else T_i is aborted.

Correctness of Timestamp Ordering Protocol

TO protocol ensures that schedules are conflict serializable.

Proof:

- Consider the serialization graph SG(S).
- For each edge $T_i \rightarrow T_j$, there exists conflicting operations $p_i(x)$ and $q_j(x)$, such that $p_i \prec_S q_j$.
- However, in that case, $TS(T_i) < TS(T_j)$.
- Thus, if $T_i \rightarrow T_j$ in SG(S), then $TS(T_i) < TS(T_j)$.
- Thus, if SG(S) contains a cycle $T_1, T_2, \ldots, T_n, T_1$, then $TS(T_1 < TS(T_1)$ which is a contradiction.

Serialization Graph Testing (SGT)

- The concurrency control mechanism maintains a graph SSG in which nodes correspond to transactions.
- Before an operation o_i (belonging to T_i) executes, an edge T_j to T_i is added if there exists an operation o_j belonging to T_j in the schedule that executed before o_i.
- If addition of the edges to the graph SSG results in a cycle the transaction is aborted and the node T_i along with all its incident edges are deleted from SSG.
- Else, if the addition of the edges does not result in the cycle in SSG, the operation is scheduled.
- The SGT protocol trivially ensures that the resulting schedules are CSR.
- Note that in the above protocol, we did not discuss when can the node corresponding to a transaction be deleted if it does not abort.

Deleting Nodes from SSG in the SGT Scheme

- Can the node be deleted when a transaction commits?
- Consider a schedule S:

$$w_1(x)\ \ w_2(x)\ \ w_2(y)\ \ c_2\ \ r_3(y)\ \ r_3(z)\ \ w_1(z)$$

- Such a schedule is not serializable. However, it could be generated by the SGT scheme if nodes were deleted from the graph as soon as the corresponding transaction commits.
- A node corresponding to a transaction T_i can be deleted if it is certain that T_i can no longer be part of a cycle in the SSG anytime in the future.
- Any node in a cycle has at least one incoming edge and at least one outgoing edge.
- So one safe time to delete a node is when it has terminated and is a source in the SSG (that is, it has no incoming edges).
- Can this condition be relaxed?

Conservative vrs Aggressive Scheduling

- We have seen three different concurrency control mechanisms that ensure CSR – 2PL, TO and SGT.
- It is possible to come up with multiple variations of of the above three mechanisms that guarantee CSR.
- These variations can be classified as either being more *conservative* or more *aggressive*.
- A conservative variation tends to delay operations to prevent transactions from aborting.
- Usually a very conservative version of each of the above three mechanims (that is, 2PL, TO, and SGT) can be constructed if there is prior knowledge of the database operations that will be executed by the transaction.
- An aggressive variation on the other hand goes on scheduling operations at the risk of later aborting the transaction.

Example Conservative Scheme: Conservative 2PL

- Assume that a transaction predeclares all the database operations it will execute.
- A transaction tries to acquire all the locks it will need before it accesses any data item.
- If it cannot acquire all locks, then it releases the locks it has been able to acquire and tries again later.
- Once the transaction is able to acquire all the locks it executes the operations and releases them according the two-phase locking rules.
- Unlike 2PL, *conservative 2PL never aborts a transaction.*
- Similar to conservative 2PL, conservative versions of SGT and TO that never abort transactions can also be developed.

Example Agressive Scheme: 2PL Certifier

- For each transction T_i in progress, the scheduler maintains two sets Read-scheduled(T_i) and write-scheduled(T_i) are maintained.
- Read and write request of transactions are processed without any delay.
- When T_i requests a commit operation, its commit is processed if for all active transactions T_j:
  - read-scheduled(T_i) $\cap$ write-scheduled(T_j) $= \emptyset$.
  - write-scheduled(T_i) $\cap$ write-scheduled(T_j) $= \emptyset$.
  - write-scheduled(T_i) $\cap$ read-scheduled(T_j) $= \emptyset$.
- The 2PL certifier produces CSR schedules.
- The order in which transactions are serialized is analogous with the order in which they are certified.
- The committed projection of a schedule produced by a 2PL certifier could be generated by the basic 2PL protocol.

Optimistic Concurrency Control

- Each transaction T_i is assigned a transaction identifier.
- T_i consists of three phases one after another:
  - **read phase**: in which T_i reads data items. All updates are done on a local copy maintained by transaction.
  - **validation phase**: in which it is decided if T_i should commit or its commitment may violate serializability
  - **write phase**: in which effects of T_i are reflected to the database.
- During validation phase, the system guarantees that if $T_i < T_j$, then either of the following must hold:
  - T_i completes its write phase before T_j starts its read phase.
  - write-set$(T_i) \cap$ read-set$(T_j) = \emptyset$ and T_i completes its write phase before T_j starts its write phase.
  - write-set$(T_i) \cap$ write-set$(T_j) = \emptyset$ and write-set$(T_i) \cap$ read-set$(T_j) = \emptyset$ and T_i completes its read phase before T_j completes its read phase.

Problem with 2PL: Deadlocks

- If transactions follow 2PL, they may get deadlocked.
- Consider the following schedule:

$$w_1(x) \;\; w_2(y)$$

- Assume that at this stage T_1 wishes to read a value of y and T_2 wishes to read the value of x.
- T_1 waits for T_2 to release the lock on y.
- T_2 waits for T_1 to release the lock on x.
- System must either prevent deadlocks or detect them.
- Once the deadlock is detected, it must be resolved.
- Deadlocks are resolved by aborting a transaction.
- Deadlocks are normally detected using either ***timeout*** or by constructing a ***wait-for graph***.

Preventing Common Deadlocks

- One common reason for a deadlock is *lock conversion*.
- Consider the schedule S:

$$r_1(x) \;\; r_2(x)$$

- In S T_1 and T_2 are holding S locks on data items x.
- Assume that both T_1 and T_2 wish to write on x. They request a upgrade to an X lock. This situation will result in a deadlock.
- Such a deadlock can be prevented if the transaction that could potentially result in an update on x had acquired an X lock on x.
- However, acquiring an X lock will even prevent other transactions that only wish to read x from accessing x.

Update Mode Locking

- Another way to prevent the deadlock due to conversion without reducing concurrency is by supporting another type of a lock– the (U)pdate lock.
- Lock Compatibility Matrix:

	Granted Mode			
requested mode	none	S	U	X
S	✓	✓	✓	
U	✓	✓		
X	✓			

- A transaction that needs to read x but may later require to write x, requests a U lock on x. This prevents other transactions from acquiring a U lock but permits other transactions to acquire a S lock.

Deadlock Prevention Schemes

- *Non-Blocking 2PL*: A scheduler aborts the transaction T_i, if T_i's lock request cannot be immediately granted.
  - Pessimistic– will result in a large number of unnecessary abortions.
  - Possibility of *livelocks*.

- *Priority Based Scheduling*:
  - With each transaction T_i a priority $p(T_i)$ is associated.
  - The scheduler decides whether or not a transaction T_i waits for a lock from T_j based on the priorities of T_i and T_j.
  - *wait-die strategy*:

    if $p(T_i) > p(T_j)$, then T_i *wait*, else *abort* T_i
  - *wound-wait strategy*:

    if $p(T_i) > p(T_j)$, then *abort* T_j, else T_i *wait*.

Livelocks with Priority Based Scheduling

- Priority based scheduling may result in *livelocks*.
- A livelock is a situation in which each member of a livelock set ends up wanting to wait for another member of the set resulting in cyclic restarts.
- Assume that wound-wait strategy is being used:

  if $p(T_i) > p(T_j)$, then ***abort*** T_j, else T_i ***wait***.
- T_1: $w_1(x)$ $w_1(y)$.
- T_2: $w_2(x)$ $w_2(z)$.
- $p(T_1)$ be 1, and $p(T_2)$ be 2.
- Assume that $w_1(x)$ has executed when T_2 requests a $w_2(x)$ operation.
- T_1 is aborted and T_2 gets the lock on x.
- System restarts T_1 with priority 3 and T_1 requests a lock on x.
- T_2 is aborted and T_1 gets a lock on x.
- T_2 is restarted with priority 4.

⋮

Preventing Livelocks using Timestamps for Priority

- One way to prevent livelocks is if the system assigns each transaction T_i a timestamp value $ts(T_i)$.
- Timestamps are unique and increase monotonically.
- Priority of the transaction is the inverse of the timestamp.
- That is, older the transaction, higher its priority.
- Timestamp of the transaction is not modified when a transaction is restarted.
- *wait-die strategy*:

  if $ts(T_i) < ts(T_j)$, then T_i *wait*, else *abort* T_i
- *wound-wait strategy*:

  if $ts(T_i) > ts(T_j)$, then *abort* T_j, else T_i *wait*.
- Using timestamps prevents cyclic restarts since the oldest transaction is never aborted.

Detecting Deadlocks

- One commonly used mechanism is to use *timeouts*.
- Another strategy is to construct a wait-for-graph (WFG) and to check for cycles in it.
- Nodes in WFG correspond to transactions.
- There is an edge from T_i to T_j if T_i waits for T_j for a lock.
- A cycle in WFG corresponds to a deadlock.
- To detect deadlocks, the scheduler periodically checks the WFG for a cycle.
- Once a cycle is detected, the scheduler chooses a *victim* transaction to abort using a *victim selection strategy*.

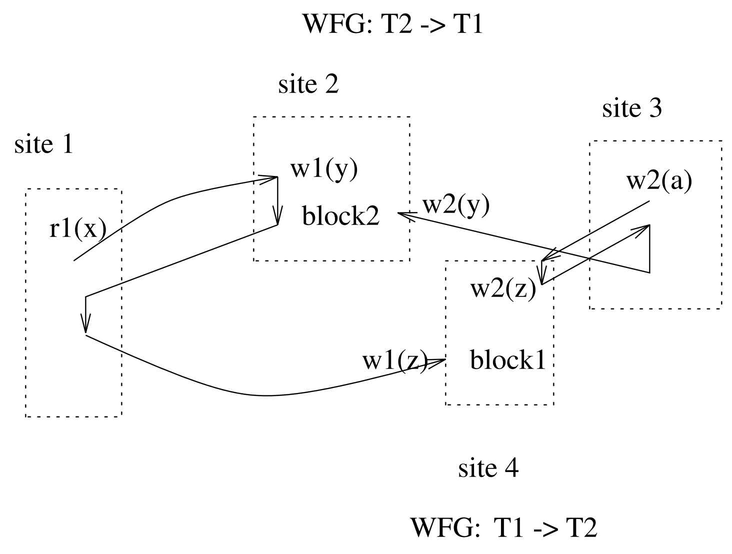
Distributed Deadlock Detection

- In a distributed environment, the WFG is distributed over different sites.
- ***Global deadlock detection***: one site collects WFG from other sites and constructs a global WFG.
- Global WFG is periodically analyzed for cycles.
- Problems:
  - High communication overhead.
  - Long delays before a deadlock is detected.

Path Pushing

- Each site s_k looks for a cycle in its local WFG.
- For each cycle it selects a victim and aborts.
- It then lists all paths $T_i \rightarrow \ldots \rightarrow T_j$ not in a cycle.
- It sends the path to every site s_l on which T_j may be blocked, waiting for a lock.
- A site on receipt of a path from another site, updates its WFG. If its WFG is modified, it identifies all the changed paths and iterates the logic.
- How does s_k know which sites T_j could be blocked at?
- Assume that each transaction is active only at one site at a time (no parallelism within a transaction).
- In this case, s_k sends path $T_i \rightarrow \ldots \rightarrow T_j$ to site on which T_j's control went after it left s_k.

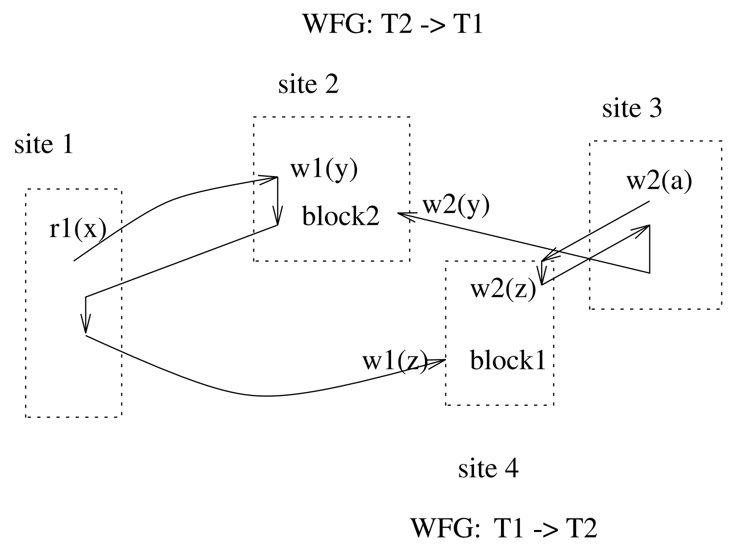
Example



Global Deadlock Detection

- One site will be chosen as the coordinator– say site 1.
- Every other site will send its WFG to site 1.
- Site 1 will construct a global WFG from the local WFG of each site.
- Once site 1 receives the WFG from site 2 and site 4, its global WFG will contain a cycle T_1, T_2, T_1.
- Site 1 will choose a victim and inform the other sites.

Example



Path Pushing Strategy

- WFG of site 4 contains a path $T_1 \rightarrow T_2$, and WFG of site 2 contains a path $T_2 \rightarrow T_1$.
- Site 4 sends its path $T_1 \rightarrow T_2$ to site 1.
- Site 1 updates its WFG and sends the path $T_1 \rightarrow T_2$ to site 4.
- Site 4 detects a deadlock.

Frequency of Deadlocks

- Let the number of data items in the database be R.
- Let the number of concurrently executing transactions be n.
- Let the size of each transaction be r.
- Assumption: $nr << R$.
- Probability that a step o_i of T_i waits (PW) is $\frac{nr}{2R}$.
- Probability that a transaction waits in its lifetime $PW(T)$ is $(1-(1-PW)^n)$.
- Hence $PW(T) \approx rPW = \frac{nr^2}{2R}$.
- Probability that T experiences a cycle of length $2 = \frac{PW(T)^2}{n}$.
- Probability that a transaction experiences a cycle of length i is proportional to $PW(T)^i$.
- Thus, if the probability of waiting is rare most cycles will be of length 2.
- Ignoring larger cycles, probability a transaction deadlocks is: $\frac{nr^4}{4R^2}$.
- Probability of a deadlock is $\frac{n^2r^4}{4R^2}$.

Deadlock Summary

- Since blocking in databases is rare, deadlocks are very rare.
- Most deadlocks involve cycles in WFG of size 2.
- A large number of strategies of detecting deadlocks in distributed databases have been developed in the literature.
- We studied 2 strategies – global deadlock detection, path pushing.
- Path pushing may be preferable since it has a lower communication overhead, and it detects smaller cycles faster than the global deadlock approach.
- Most common strategy of deadlock detection is timeout since it is simple and very efficient.
- Implementing an actual deadlock detector is considered too complex a task for something that is very rare.
- Some common causes of deadlocks are lock conversion, lock escalation, and hot spots.
- However, special mechansisms can be used to avoid such deadlocks.

Problem with 2PL: Hot Spots

- Hot spots are data items that are accessed and updated by a large number of transactions.
- Using the 2PL protocol, when transactions are modelled as a sequence of read/write operations will result in poor performance.
  - Assume each transaction holds the lock on the hot spot for 1 sec. In this case, the maximum transactions per second is 1.
- There are different kinds of hot spots and there is no single technique that will work for all of the situations.
- The typical solution to the hot spot problem requires the scheduler to exploit the semantics of the operation on the data items.
- For example, a transaction is not modelled as simply a sequence of read and write operations, but may consist of other semantically richer operations.

Hot Spot Situation 1

- Say that *total* is the running sum of all the accounts in a bank.
- Each deposit and withdraw transaction will either increment or decrement *total*.
- Thus, *total* will be a hot spot.
- If each transaction were to acquire a transaction duration X lock on *total* system performance will degrade.

Hot Spot Situation 1

- Instead, the system supports 2 new types of locks– (I)ncrement and (D)ecrement locks.
- A transaction wishing to increment the account acquires an I lock on total.
- A transaction wishing to decrement the total, acquires a D lock on total.
- Lock Conflict Matrix:

	R	W	I	D
R	√			
W				
I			√	√
D			√	√

- Thus, a transaction does not block other transactions from performing increments and decrements to *total* by acquiring a X lock on *total*.

Hot Spot Situation 2

- Consider an airline database in which for each flight there is a data item *seats-available*.
- Transactions that reserve a flight decrement the *seats-available* by the number of seats reserved.
- In such a scenario two decrements of seats may not be commutative.
- For example, if at an instance 19 seats are available. Then two decrement operations that decrement the *seats-available* by 10 each are not commutative (since they will result in different return values if executed in different order).
- However, two decrements that decrement the *seats-available* by 1 each are commutative and should not result in blocking each other.
- Such a hot spot can be solved by using *escrow* locking.

Escrow Locking

- Associated with the hot spot ***seats-available*** is a range of values denoted by a closed interval [m,n].
- The interval [m,n] implies that the value of ***seats-available*** is in between m and n.
- A decrement whose return value does not change for any value of seats-available in between [m, n] can be permitted since it commutes with all the previously executed decrement operations.
- A decrement whose return value changes depending upon the number of seats available cannot be permitted to execute since it does not conflict with the previously executed decrement operations.

Hot Spot Situation 3: serial number problem

- An application domain requires that each committed transaction acquire a *serial number*.
- The serial number is monotonically increasing– that is, if T_1 commits after T_2, then the serial number of T_1 is less than the serial number of T_2.
- The application that receives a serial number may do some processing based on the serial number it receives. Depending upon its processing the transaction may abort.
- Serial numbers are dense; that is, no gaps are allowed.
- Tens to hundreds of transactions may be trying to acquire a serial number concurrently.

Other Hot Spots

- Other hot spots in DBMSs include– index structures, B-trees, hash tables, etc.
- 2PL will perform very bad in presence of hot spots. So application specific methods that exploit semantics are used to overcome issue of hot spot. Most solutions are very hacky!
- For example, most index structures have a few mechanisms using which they are traversed (B-tree traversal, hash table operations, etc.) are fixed. So solutions exploit this structure for relaxing 2PL requirements.

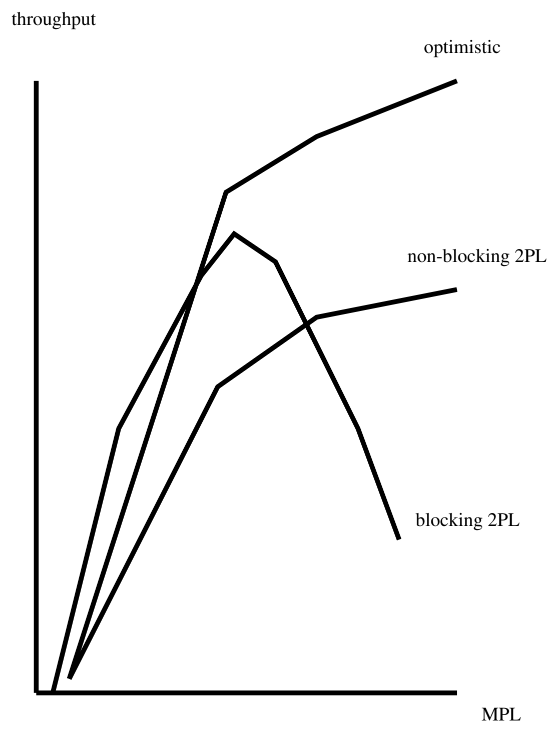
Summary of Schedulers

- A large number of schedulers that ensure CSR have been developed in the literature.
- Two-Phase locking, timestamp ordering, optimistic concurrency control.
- How should we compare the various schedulers?
  - *Efficiency:* overhead of scheduling an operation. In most schedulers, the overhead is not very high and thus efficiency is not a good measure of comparison.
  - *Degree of concurrency:* number of input schedules permitted by the scheduler without rejecting or delaying any operation. Not a good measure since it does not provide a total order over the schedulers. For example, it will not help choose between TO and 2PL since there are 2PL schedules that are not TO and vice versa.
- One way to compare the schedulers is to simulate the scheduler and study effect of scheduling policy on *throughput* and *response time.*
- Read [Aggrawal, et. al. 85] for simulation studies of concurrency control protocols.

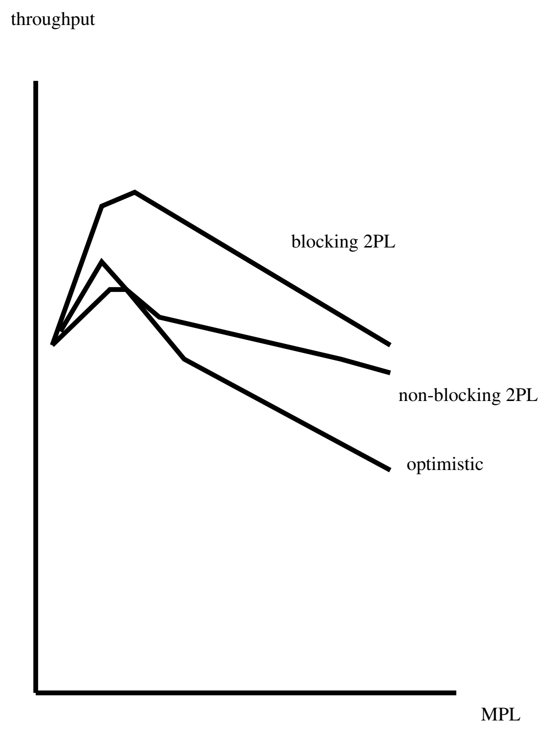
Data and Resource Contention

- Throughput of a system does not increase linearly with the number of users due to two reasons:
  - Resource Contention: different processes competing for resources like memory, disk, CPU, etc.
  - Data Contention: different transactions competing to access data.
- Both data contention as well as resource contention could cause *thrashing*.
- During thrashing, increasing the number of transactions (or MPL) causes throughput to reduce.
- Thrashing due to resource contention may be caused, for example, when the system is busy transferring pages in and out of memory.
- Thrashing due to data contention may be caused, if for example, adding more transactions causes more restarts in the optimitic scheme, or causes more transactions to block in the basic 2PL scheme.

Performance of Locking and Optimistic Protocols



INFINITE RESOURCES



LIMITED RESOURCES

- Under infinite resources, both the optimistic and the non-blocking 2PL outperform the basic blocking 2PL. The reason is that DC thrashing sets in early in blocking 2PL.
- However, when resources are not infinite, the loss of work that results due to increased restarts in optimistic and non-blocking 2PL, cause it to perform worse that basic 2PL.