COMP90050 ADBS

授课老师：Rita

1 – JDBC

JDBC – Java Database Connection

```
public static void JDBCexample(String dbid, String userid, String passwd) {
    try (Connection conn = DriverManager.getConnection( url: "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
         Statement stmt = conn.createStatement(); )
    {
        // Do Actual Work
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle); }
}
```

id passwd

调用连接 statement object

执行

- Open a connection
- Create a statement object
- Execute queries using the statement object to send queries and fetch results
- Exception mechanism to handle errors

```
public void updateDatabase(Statement stmt) throws SQLException {
    try
    {
        stmt.executeUpdate( sql: "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
    }
    catch (SQLException sqle) {
        System.out.println("Could not insert tuple. " + sqle);
    }
    // Execute query and fetch results
    ResultSet rset = stmt.executeQuery( sql: "select dept_name, avg (salary) from instructor group by dept_name");
    while (rset.next()) {
        System.out.println(rset.getString( columnLabel: "dept_name") + " " + rset.getFloat( columnIndex: 2));
    }
}
```

帮助执行

异常捕获

check id 是否重复

primary key

Transactions in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically (not a good idea)
- Can turn off automatic commit on a connection - conn.setAutoCommit(false);
- Can turn on automatic commit on a connection - conn.setAutoCommit(true);
- Transactions must then be committed or rolled back explicitly
  - conn.commit(); 执行成功后⇒开始提交(automatically)
  - conn.rollback();

2 – Transaction

Transaction & Operation

Transaction - A unit of work in a database

- A transaction can have any number and type of operations in it
- Either happens as a whole or not
- Transactions ideally have four properties, commonly known as ACID properties

transaction VS operation

A->B转100 transaction

①A-100
②B+100
③记录A->B转账

operation.

ACID Properties

要么都干 要么都不干(100%)要么0

Atomicity - All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.

一致性:

Consistency - Data is in a ‘consistent’ state when a transaction starts and when it ends – in other words, any data written to the database must be valid according to all defined rules (e.g., no duplicate student ID, no negative fund transfer, etc.)

Isolation - transaction are executed as if it is the only one in the system.

Durability - the system should tolerate system failures and any committed updates should not be lost.

Type of Action

- **Unprotected actions** - No ACID property
- **Protected actions** - These actions are not externalised before they are completely done. These actions are controlled and can be rolled back if required. These have ACID property.
- **Real actions** - These are real physical actions once performed cannot be undone. In many situations, atomicity is not possible with real actions.

Embedded SQL

(Open Database Connectivity)

定义了4个变量

```
int main()
{           exec sql INCLUDE SQLCA; /*SQL Communication Area*/
            exec sql BEGIN DECLARE SECTION;
             /* The following variables are used for communicating
                      between SQL and C */
```

4个变量

```
              int OrderID; /* Employee ID (from user) */
              int CustID; /* Retrieved customer ID */
              char SalesPerson[10] /* Retrieved salesperson name */
              char Status[6] /* Retrieved order status */
            exec sql END DECLARE SECTION;
/* Set up error processing */
            exec sql WHENEVER SQLERROR GOTO query_error;
            exec sql WHENEVER NOT FOUND GOTO bad_number;
```

选3 attr => attribute name. (defined in table)

```
/* Prompt the user for order number */
      printf ("Enter order number: ");
      scanf_s("%d", &OrderID);
/* Execute the SQL query */
      exec sql SELECT (CustID, SalesPerson, Status)
         FROM Orders ->table           进入变量1
       WHERE OrderID = (:OrderID) // ":" indicates to refer to  C variable
       INTO :CustID, :SalesPerson, :Status;
/* Display the results */
      printf ("Customer number: %d\n", CustID);
      printf ("Salesperson: %s\n", SalesPerson);
      printf ("Status: %s\n", Status);
      exit();
query_error:
       printf ("SQL error: %ld\n", sqlca->sqlcode); exit();
bad_number:
       printf ("Invalid order number.\n"); exit(); }
```

host variable 是自己定义的

:variable

变量2 变量3 变量4

Host Variables - Declared in a section enclosed by the BEGIN DECLARE SECTION and END DECLARE SECTION. While accessing these variables, they are prefixed by a colon " :" . The colon is essential to distinguish between host variables and database objects (for example tables and columns).

Data Types - The data types supported by a DBMS and a host language can be quite different.

Error Handling - The DBMS reports run- time errors to the applications program through an SQL Communications Area (SQLCA) by INCLUDE SQLCA. The WHENEVER...GOTO statement tells the pre- processor to generate error-handling code to process errors returned by the DBMS.

Singleton SELECT - The statement used to return the data is a singleton SELECT statement; that is, it returns only a single row of data. Therefore, the code example does not declare or use cursors. 返回一组行数据

Embedded SQL

- Embedded SQL is a method that combines SQL with a high– level programming language's features.
- It enables programmers to put SQL statements right into the source code files used to set up an application.
- Database operations may be carried out effortlessly by developers by adding SQL statements to the application code.
- The source code files having embedded SQL statements should be **pre-processed before compilation** because of the issue of interpretation of SQL statements by the high– level programming languages.

Dynamic SQL

- Dynamic SQL involves the creation and execution of SQL statements at **runtime.**
- Dynamic SQL allows developers to generate SQL statements dynamically based on runtime conditions or user input.
- By combining changeable data, conditions, and dynamic database or column names, developers may quickly construct SQL queries using dynamic SQL.
- Because of its adaptability, dynamic SQL is a good choice when the SQL statements need to change in response to evolving needs or user inputs.
- Dynamic SQL queries are built at execution time, so the system chooses how to access the database and conduct the SQL queries.

Flat Transaction

开始符begin 结束符commit

- Everything inside the BEGIN WORK and COMMIT WORK is at the same level.
- The transaction will either survive together with everything else (commit)
- Or it will be rolled back with everything else (abort) – if some errors happen

id　当前余额　商品 单价

```
exec sql BEGIN WORK;
AccBalance = BuySomething(AccId, AccBalance,  Amount);
exec sql COMMIT WORK;
```

flat ~~trasc~~ transaction

```
Long BuySomething(long AccId,  long AccBalance, long Amount){
①Exec sql INSERT INTO history(AccId, Amount, time)
   VALUES( :AccId, :Amount, CURRENT);
②exec sql UPDATE accounts
   SET AccBalance =  AccBalance -  :Amount
   WHERE AccId = :AccId;
   return(AccBalance);
}
```

一个transaction中有两个operations

返回余额

```
exec sql BEGIN WORK;
AccBalance = BuySomething(AccId, AccBalance,  Amount);
if (AccBalance < 0){
exec sql ROLLBACK WORK;}
else{ exec sql COMMIT WORK;  }
```

(返回负和 就滚了)

Buy something → 检查 → 够 → 买
不够 → rollback ← 不往history中插数据
不 update account balance
☆要做全做要不全不做☆

Flat Transaction - Limitation

Flat transactions do not model many real applications.

- 个 flat transaction.

→ BEGIN WORK 要么都做要么都不做.

S1 : book flight from Melbourne to Singapore

S2: book flight from Singapore to London

S3: book flight from London to Dublin

→ END WORK

M→S→L→D ✗→ destination
→P → ✓

发现D到不了destination →但S可以到P再去destination

☆ 当前flat transaction→要全部回滚到M,再次订票时又要M→S走一遍, ☆ ⇒ waste.

From Dublin if we cannot reach our final destination instead we wish to fly to Paris from Singapore and then reach our final destination.

If we roll back we need to redo the booking from Melbourne to Singapore which is a waste.

相

Flat Transaction – Limitation Example

```
IncreaseSalary()
{
    real percentRaise;
    receive(percentRaise);
    exec sql BEGIN WORK;
    exec SQL UPDATE employee
    set salary = salary *  (1 + :percentRaise)
    exec sql COMMIT WORK;
}
```

涨工资 → 所有员工都涨.

how many employees?

许多员工 → connection to DB will be very long,

又或者connection断了 → update一半剩下一半未update → 也有问题 (failure)

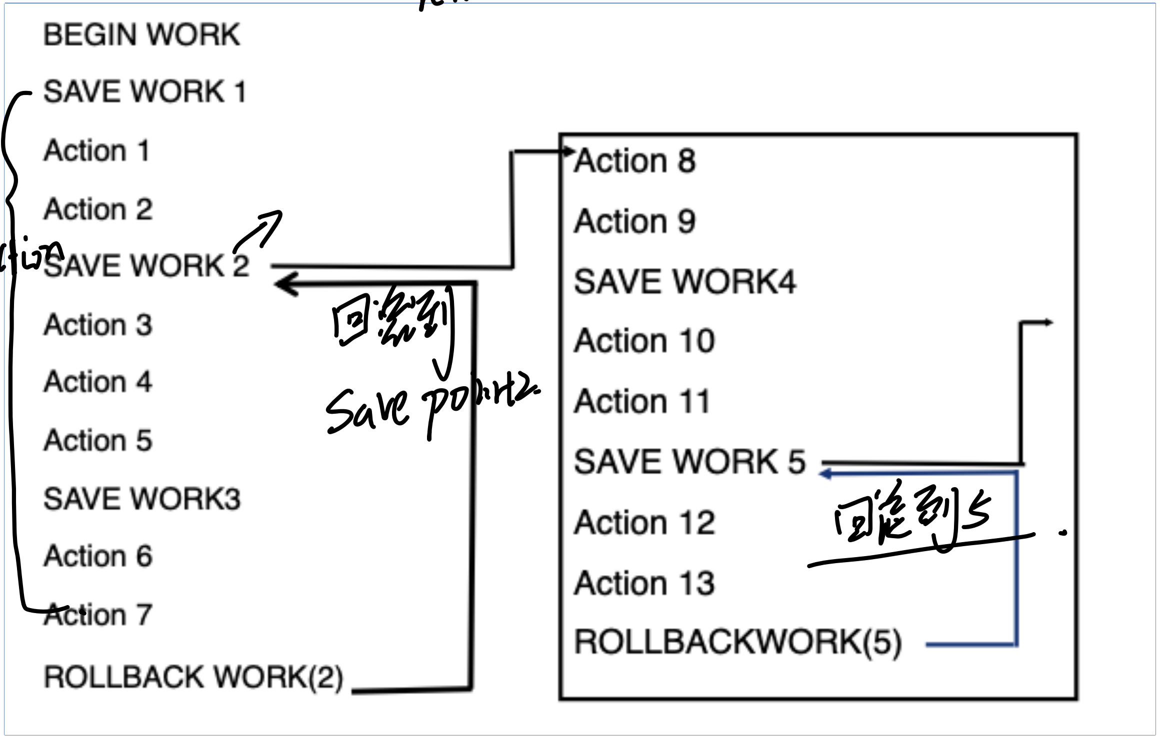
Flat Transaction – Solution: Save Points

→ 存点有点 在crush发生时，不需重新开始 只需回到 save point.
return to

- The only reason why an application program needs an identifier for a savepoint is that it may later want to re- establish (return to) that savepoint.

- To do that, the application invokes the ROLLBACK WORK function, but rather than requesting the entire transaction to be aborted, it passes the number of the savepoint it wants to be restored.

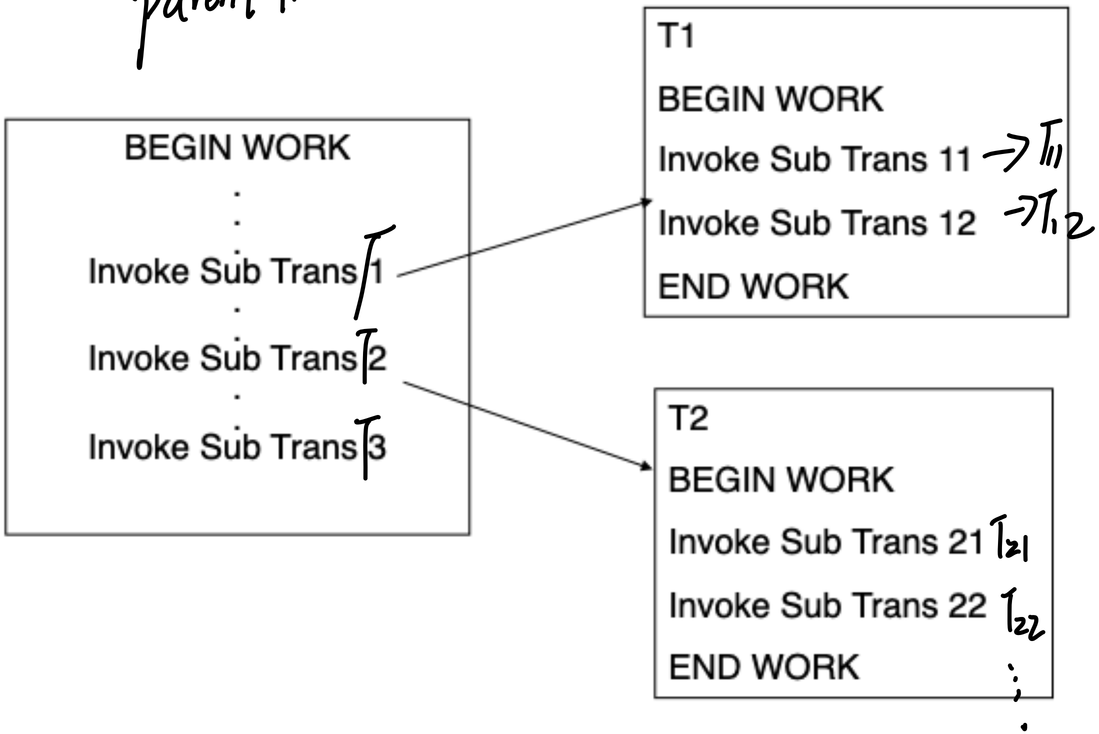
个 transaction

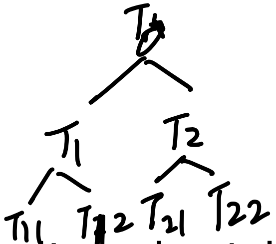


Nested Transaction 树状结构

parent transaction

sub-transaction





T1 rollback ⇒ T11, T12 both rollback.

(T11想commit ⇒ 要求T1、T0都要commit才行)

Commit rule

- A sub- transaction can either commit or abort, however, commit cannot take place unless the parent itself commits. (parent先commit, sub transaction才能commit)
  if it's a child, it's parent should commit before)
- Sub- transactions have Atomicity, Consistency, and Isolation properties but not have Durability property unless all its ancestors commit. 我要commit ⇒ 要求所有的祖先 commit. commit之后才有durability的要求
- Commit of a sub transaction makes its results available only to its parents

subtransaction commit 只有它的parent知道 (sibling不知道)

T11 commit T1知道但T12不知道.

Roll back rules

- If a sub- transaction rolls back all its children are forced to roll back

Visibility rules

- Changes made by a sub- transaction are visible to the parent only when the sub- transaction commits.
- All objects of parent are visible to its children.

(T1和T2同时进行

Implication of this is that the parent should not modify objects while children are accessing them. This is not a problem as parent is not run in parallel with its children.)

T0和T11 T12不同时进行

3 – TP Monitor

Transaction Processing Monitor (TP Monitor)

The main function of a TP monitor is to **integrate other system components and manage resources**.

存储

- TP monitors manage the transfer of data between clients and servers (数据传输
- Breaks down applications or code into transactions and ensures that all databases are updated properly
- It also takes appropriate actions if any error occurs

TP Monitor Services

多个DBs overall ACID property

Heterogeneity: If the application needs access to different DB systems, local ACID properties of individual DB systems is not sufficient. Local TP monitor needs to interact with other TP monitors to ensure the **overall ACID** property.

Control communication: If the application communicates with other remote processes, the local TP monitor should maintain the communication status among the processes to be able to recover from a crash. 一旦有remote communication 就有可能出现crash => crash recovery end user 用户

Terminal management: Since many terminals run client software, the TP monitor should provide appropriate ACID property between the client and the server processes.

User Interface · ~~界面~~

Presentation service: this is similar to terminal management in the sense it has to deal with different presentation (user interface) software - - e.g. X- windows

登陆网站 → 过了三小时 → session过期 自动退出

Context management: E.g. maintaining the sessions etc.

Start/Restart: There is no difference between start and restart in TP based system.

4 – Deadlock

For correct execution, we need to impose exclusive access to the shared variable counter by Task1 and Task2.

Concurrency Problem

Shared counter = 100;

Task1/Trans/Process/Thread	Task2/Trans/Process/Thread
counter = counter +10;	counter = counter +30;

Task1 and Task2 run concurrently. What are the possible values of counter after the end of Task1 and Task2?

Note: == means equals.

a) counter == 110	b) counter == 130	c) counter == 140;
Sequence of actions	Sequence of actions	Sequence of actions
T1: Reads counter == 100	T1: Reads counter == 100	T1: Reads counter == 100
T2: Reads counter == 100	T2: Reads counter == 100	T1: Writes counter == 100+10
T2: Writes counter == 100+30	T1: Writes counter == 100+10	T2: Reads counter == 110
T1: Writes counter == 100+10	T2: Writes counter == 100+30	T2: Writes counter == 110+30

结果差了 a), b) 都不 consistent.

140

对同一若干先读先改.

T2 100
T2 | 100 +30
T1 130
T1 | 130 +10
T1 = 140

Concurrency Control

- To resolve conflicts 冲突（同时修改先修改值）
- To preserve database consistency（保证一致性）

运行结果不受"顺序影响

Different ways for concurrency control

三种方式

– **Dekker's algorithm (using code)** - needs almost no hardware support, but the code is very complicated to implement for more than two transactions/ processes.

operated system

– **OS supported primitives (through interruption call)** - expensive, independent of number of processes, machine independent

– **Spin locks (using atomic lock/unlock instructions)** – most commonly used

Dekker's algorithm

```
int turn = 0 ; int wants[2];            // both should be initially 0
...
while (true) {
        wants[i] = true;                // claim desire
        while (wants[j]) {
                if (turn == j) {
                        wants[i] = false;     // withdraw intention
                        while (turn == j);
                        wants[i] = true;      // wait and reassert
                }
        }
        counter = counter + 1:          // resource we want mutex on
        turn = j;                       // assign turn
        wants[i] = false;
}...
```

Dekker's algorithm

- Needs almost no hardware support although it needs atomic reads and writes to main memory
- The code is very complicated to implement if more than two transactions/ process are involved
- Harder to understand the algorithm for more than two process
- Takes lot of storage space
- Uses busy waiting
- Efficient if the lock contention (that is frequency of access to the locks) is low

OS supported primitives – such as lock & unlock

- Through an interrupt call, the lock request is passed to the OS
- Need no special hardware
- Are very expensive (several hundreds to thousands of instructions need to be executed to save context of the requesting process.)
- Do not use busy waiting and therefore more effective

Spin Lock

- All modern processors do support some form of spin locks.
- Executed using atomic machine instructions such as test and set or compare and swap
- Need hardware support
- Use busy waiting
- Algorithm does not depend on number of processes
- Are very efficient for low lock contentions – all DB systems use them

Spin lock – Test and Set

```
testAndSet(int *lock)
{ /* the following is executed atomically, memory bus can be locked for up
    to two cycles (one for read and for writing*/
        if (*lock == 1){ * lock = 0; return (true)}
           else return (false);
    }
```

<u>Using test and set in spin lock for exclusive access</u>

int lock = 1; % initial value

T1	T2
/*acquire lock*/	/*acquire lock*/
while (!testAndSet(&lock);	while (!testAndSet(&lock);
/*Xlock granted*/	/*Xlock granted*/
//exclusive access for T1;	//exclusive access for T2;
counter = counter+1;	counter = counter+1;
/* release lock*/	/* release lock*/
lock = 1;	lock = 1;

Spin lock – Compare and Swap

```
boolean cs(int *cell, int *old, int *new)
{/* the following is executed atomically*/
if (*cell == *old)      { *cell = *new; return TRUE;}
else { *old = *cell; return FALSE;}
}
```

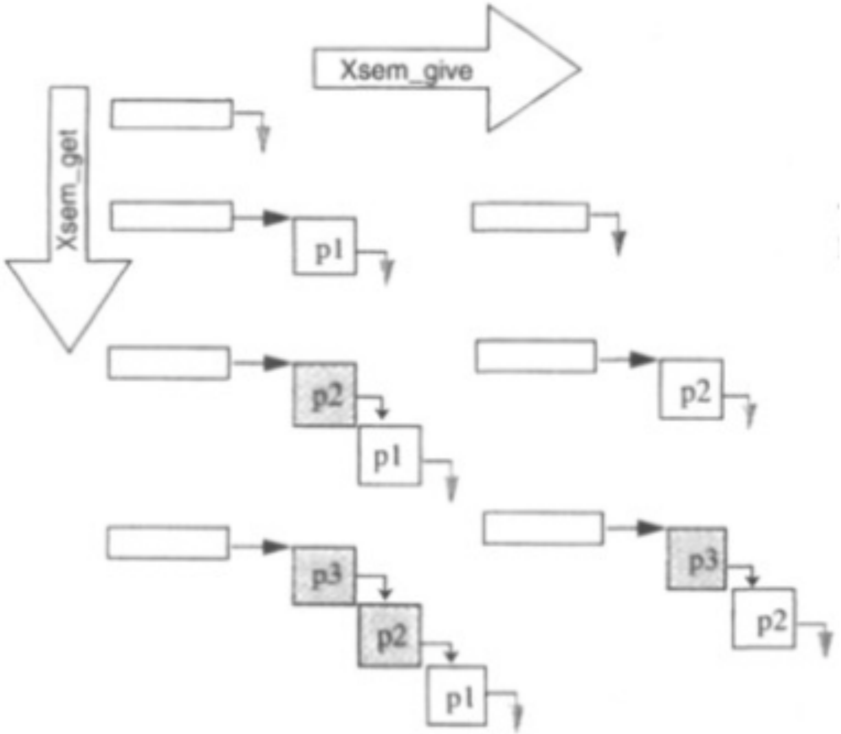
Semaphore

Derived from train and track

- Try to Get track, wait if track not clear
- If Get track was successful, use it (no other train will be able to use it at the same time)
- Once the train passes, the semaphore is set until the train exits that section of track

Computer semaphores have a **get()** routine that acquires the semaphore (perhaps waiting until it is free) and a **give()** routine that returns the semaphore to the free state, perhaps **signalling** (waking up) a waiting process.

Exclusive mode Semaphore



- Pointer to a queue of processes
- If the semaphore is busy but there are no waiters, the pointer is the address of the process that owns the semaphore.
- If some processes are waiting, the semaphore points to a linked list of waiting processes. The process owning the semaphore is at the end of this list.
- After usage, the owner process wakes up the oldest process in the queue (first in, first out scheduler)

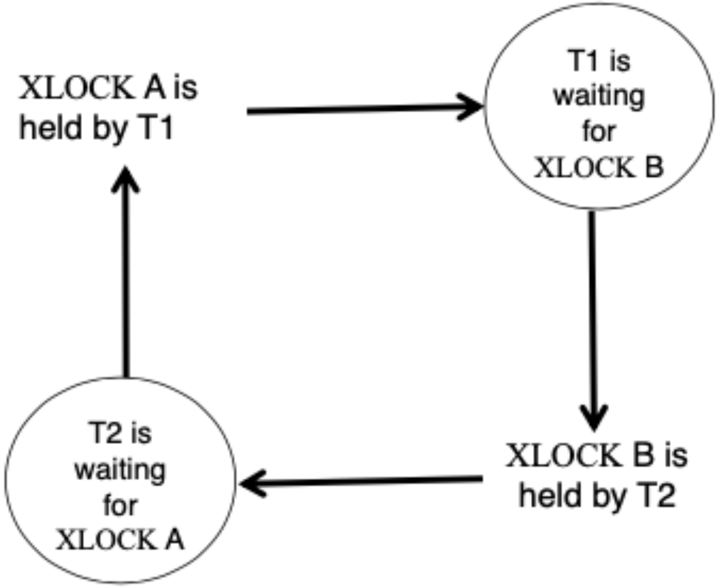
Deadlock

Deadlocks are rare, however, they do occur and the database has to deal with them when they occur

In a deadlock, each process in the deadlock is waiting for another member to release the resources it wants.

T1	T2
Begin	Begin
XLOCK(A)	XLOCK(B)
Write to A	Write to B
XLOCK(B)	XLOCK(A)
Write B	Write A
Unlock (A)	Unlock (A)
Unlock (B)	Unlock (B)
end	end

Resource Dependency Graph



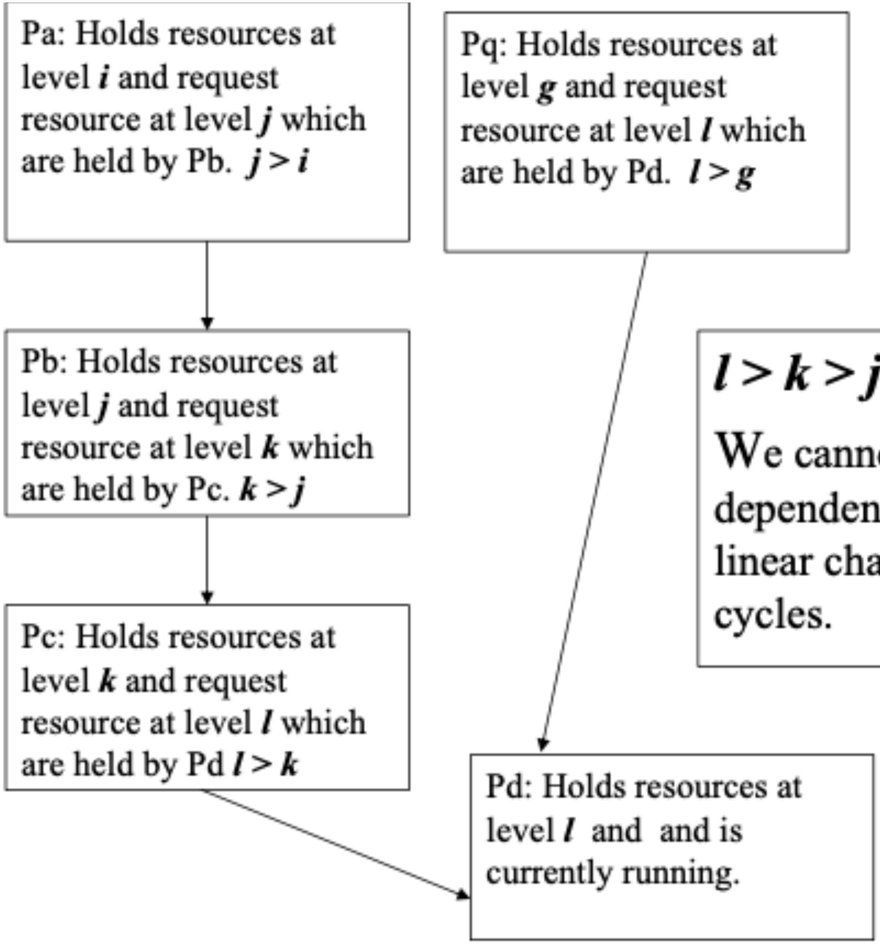
Deadlock - Solutions

- Have enough resources so that no waiting occurs – not practical

- Do not allow a process to wait, simply rollback after a certain time. This can create live locks which are worse than deadlocks.

- Linearly order the resources and request of resources should follow this order,
i.e., a transaction after requesting i^{th} resource can request j^{th} resource if $j > i$.
This type of allocation guarantees no cyclic dependencies among the transactions.

e.g.,



$l > k > j > i$ ***and*** $l > g$.

We cannot have loops. The dependency graph can be a tree or a linear chain and hence cannot have cycles.

Deadlock – Avoidance / Mitigation

- Pre- declare all necessary resources and allocate in a single request.

- Periodically check the resource dependency graph for cycles. If a cycle exists - rollback (i.e., terminate) one or more transaction to eliminate cycles (deadlocks). The chosen transactions should be cheap (e.g., they have not consumed too many resources).

- Allow waiting for a maximum time on a lock then force Rollback. Many successful systems (IBM, Tandem) have chosen this approach.

(Many distributed database systems maintain only local dependency graphs and use time outs for global deadlocks.)

Quiz

1. Which statement is correct?

A. Spinlocks do not need any hardware support in general

B. It is efficient to do concurrency control with Dekker's algorithm because the algorithm relies on a lot of hardware support

C. Test and set implementation of spinlock requires memory locking support

D. OS supported interrupt calls are fast and cheap because they are done at OS level

2. Which action is not suitable for nested transactions?

A. Sub- transactions do not have durability property until ancestors commit

B. Objects at parents can be made visible to children

C. All children as well as parent transaction can run in parallel together

D. Commit of a sub- transaction makes data available to parents

3. If a transaction A is running concurrently with transaction B, then the execution order should be equal to the outcome of which one of the following orders?

A. B running before A but not A running before B

B. A running before B or B running before A

C. It does not really matter for these two specific transactions

D. A running before B but not B running before A

4. Why do we really need to write code as Embedded or Dynamic SQL?

A. Because only this way we can write our queries and save them for later use as well

B. Because using another language in addition to SQL gives us more power to develop Applications

C. All of the choices given in this question are correct

D. Because this is the main way we can learn about errors in our SQL queries

5. Which of the following statements is true for transaction processing?

A. Running transactions concurrently but not in isolation with each other is not desirable.

B. Transactions are expected to be run obeying Chemical properties such as properties of acids and metals.

C. Durability is the least important ACID property.

D. SQL queries are the main atomic units of execution in relational DBMSs