Recovery and Atomicity

● *Transaction* : a program unit to access or update various data items

● Transaction 수행시 failure 발생으로 인한 *data inconsistency*를 복구하는 방법

● Transaction atomicity : transaction의 수행이 완전히 수행되거나 또는 수행이 전혀 안되거나 하는 성질

● Transaction Properties

◉ *Atomicity*

◉ *Consistency(Seriability)*

◉ *Isolation*

◉ *Durability*

● Failure Classification

◉ Storage Types

- ***Volatile Storage :*** registers, main memory
- ***Nonvolatile Storage :*** secondary memory(정보손실 가능)
- ***Stable Storage :*** secondary memory(정보손실 없음)

◉ Failure Types

- ***Logical Errors***
- ***System errors :*** ex) Deadlock
- ***System Crash :*** H/W malfunction으로 인한 volatile storage 내의 정보 손실
- ***Disk failure :*** Disk Block Contents Loss for head crash or failure

● Disk Acces 기본 연산

◉ **input(X)** : X를 포함하는 disk의 physical block을 main memory로 전송

◉ **output(X)** : X를 포함하는 메모리의 buffer block를 disk로 전송

◉ **read(X, xi)** :

```
void function read(X, xi)  {
     if (X is not in memory)
       input(X) ;
     xi=X ;
     return ;
};
```

◉ **write(X,xi)** :

```
void function write(X, xi) {
     if (X is not in memory)
       input(X);
     X(in buffer block)=xi ;
     return ;
};
```

● Transaction Model

◉ *Transaction :* a program unit to access or update various data items

◉ **Inconsistency Example :** Transaction failure case

```
T: read(A,a1)
   a1 := a1 - 50
   write(A,a1)
   read(B,b1)     <---- failure 발생 시점
   b1 := b1+50
   write(B,b1)
```

◉ **Correctness and Atomicity of Transaction**

- *Correctness :* Each transaction must be a program that preserves database consistency
- *Atomicity :*

◉ *Transaction Abort*

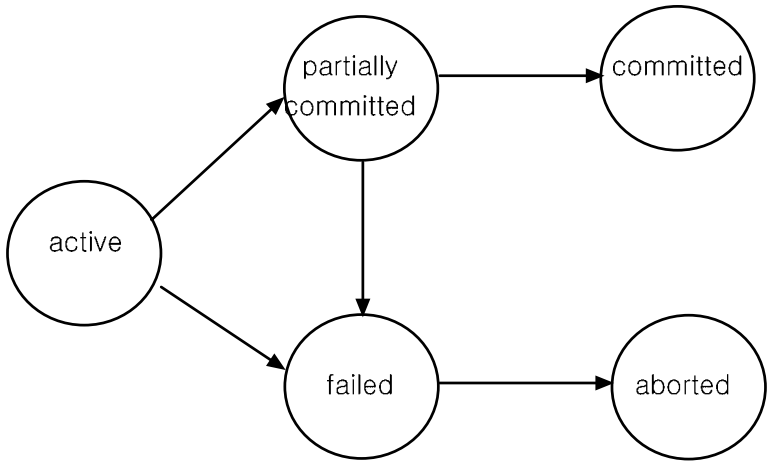
- Transaction의 성공적 실행의 실패
- abort되었을 때 transaction의 중간 결과는 DB에 반영되어서는 안됨
- rollback : transaction abort 발생시 그 transaction 수행 이전 상태로 되돌리는 작업

◉ *Transaction Commit*

- Transaction의 성공적인 실행 완성
- data update의 경우 DB를 새로운 상태로 변경함

◉ Transaction States

- *Active*
  - ⊙ 실행 상태
- *Partially Committed*
  - ⊙ transaction의 마지막 명령어 처리 수행
  - ⊙ 메모리 내의 변경 내용을 DB로 반영하지 않은 상태
- *Failed*
  - ⊙ 정상 실행이 불가능한 상태
- *Aborted*
  - ⊙ rollback을 통하여 transaction의 실행 이전 상태로 되돌린 상태
- *Committed*
  - ⊙ transaction의 마지막 명령어 처리 수행
  - ⊙ 메모리 내의 변경 내용을 DB로 반영한 상태



● Recovery Mechanisms

◉ Log-Based Recovery Mechanism

- ● *Database Log(Journal)*
  - ◉ **Log는 stable storage에 위치함**
  - ◉ **Log record로 구성됨(Log Record는 하나의 database write 연산을 나타냄)**
  - ◉ **Log record의 fields**
    - ● *Transaction Name*
    - ● *Data Item Name*
    - ● *Data Item의 Old Value*
    - ● *Data Item의 New Value*

◉ Logging 방법 :

- <Ti starts>
- <Ti, Xj, Vold, Vnew>
- <Ti commits>

◉ redo와 undo 연산

- **redo(Ti) : redo(redo(redo(Ti))) = redo(Ti)**

  Transaction Ti에 의해 update된 모든 data item Xj의 값을 Vnew값으로 set하는 연산

- **undo(Ti) : undo(undo(undo(Ti))) = undo(Ti)**

  Transaction Ti에 의해 update된 모든 data item Xj의 값을 Vold값으로 set하는 연산 (즉, rollback)

◉ Defered Database Modification

- 모든 DB의 변경들을 Log로 반영
- write에 의한 DB 변경은 transaction의 partially commit 이후로 지연
- partially commit 후 write에 의한 변경 내용을 DB로 반영
- Log record의 fields
  - ◉ *Transaction Name*
  - ◉ *Data Item Name*
  - ◉ *Data Item의 New Value*

● **Example**

수행전의 DB 내용 : A = 1000 B = 2000 C = 700

transactions :

```
T0 : read(A,a1)
     a1 := a1 - 50
     write(A,a1)
     read(B,b1)
     b1 := b1 + 50
     write(B,b1)
T1 : read(C,c1)
     c1 := c1 - 100   <-- crash 발생시점
     write(C,c1)
```

Log	Database
<T0 starts>	
<T0, A, 950>	
<T0, B, 2050>	
<T0 commits>	
	A = 950
	B = 2050
<T1 starts>	
<T1, C, 600>	
<T1 commits>	
	C = 600

● Transaction Failure 발생시의 recovery

```
integer function Defered_Logging_Recovery_Mechanism()  {
      if (<Ti starts> in Log and <Tl commits> in Log) {
         redo(Ti) ;
      } ;
      return;
};
```

◉ Immediate Database Modification

- Transaction이 active 상태일때 변경 사항을 DB 반영가능하게 함
- Log record의 fields
  - ◉ *Transaction Name*
  - ◉ *Data Item Name*
  - ◉ *Data Item의 Old Value*
  - ◉ *Data Item의 New Value*
- **Logging 방법 :**
  - ◉ **<Ti starts>**
  - ◉ **<Ti, Xj, Vold, Vnew>**
  - ◉ **<Ti commits>**

● Example

수행전의 DB 내용 : A = 1000 B = 2000 C = 700

Transactions	Log	DB
T0 :	<T0 starts>	
read(A,a1)		
a1 := a1 - 50		
write(A,a1)	<T0, A, 1000, 950>	
		A=950
read(B,b1)		
b1 := b1 + 50		
write(B,b1)	<T0, B, 2000, 2050>	
		B=2050
	<T0 commits>	
T1 :	<T1 starts>	
read(C,c1)		
c1 := c1 - 100 <----- **Failure 발생 시점**		
write(C,c1)	<T1, C, 700, 600>	
		C= 700
	<T1 commits>	

● **Recovery**

```
integer function Immediate_Logging_Recovery_Mechanism()  {
 If <Ti starts> in Log and  <Ti commits> in Log  {
     redo(Tl);
     return REDO ; }
 elseif <Tl starts> in Log and <Ti commits> not in Log {
     undo(Ti);
     return UNDO;
  };
};
```

◉ **Log Record Buffering**

- Performance를 목적으로 Log Record들을 main memory로 buffering
- Execution Rules
  1. 평상시 Log record를 main memory로 buffering
  2. Main Memory의 data block이 DB로 반영되기 직전에 그 block의 data와 관련된 모든 log record들을 log file로 기록
  3. <Ti commits> log record가 log로 출력되기 전에 Ti와 관련된 모든 log record들을 Log로 출력
  4. <Ti commits> log record가 출력된 후 Ti는 commit 상태로 됨

◉ **Database Buffering**

- Virtural memory 기법을 쓰는 시스템의 경우 page fault가 발생함
- Block B1을 swap-out하고 Bolck B2를 swap-in해야 할 경우
  1. Bolck B1에 관련된 모든 log record를 Log file에 기록
  2. Block B1을 DB 로 출력
  3. Block B2를 DB에서 main memory로 가져 옴

◉ Checkpoint Mechanism

● ***Active Transaction*(AT)**

= <Ti starts> in Log File and <Ti commits> not in Log File

● ***checkpoint record* = a set of ATs**

● ***Checkpoint 연산* : 주기적으로(수분마다 실행) 실행되는 다음의 연산**

1. Main Memory에 존재하는 모든 log record들을 Log FIle에 기록
2. 변경된 모든 buffer block들을 DB로 출력
3. checkpoint record를 Log File에 기록

● Recovery Procedure

```
void function chechpoint_recovery() {
  locate at the last checkpoint record in Log__File;
  UndoSet = a set of all ATs in checkpoint record;
  RedoSet = { };
  Locate file pointer at the last checkpoint record of Log__File;
  while !(eof(Log__File)) {
    skip current Log__file record;
    if (current Log__File record == <Ti starts>)
        UndoSet = UndoSet <union>  {Ti} ;
    elseif (current Log__File record == <Ti commits>) {
        UndoSet = UndoSet - {Ti};
       RedoSet = RedoSet union {Ti};
   };
  while UndoSet != { } {
    a = UndoSet의 member ;
    UNDO(a);
    UndoSet = UndoSet-{a};
   };
  while RedoSet != { } {
    a = RedoSet의 member ;
    REDO(a);
    RedoSet = RedoSet-{a};
   };
};
```

◉ Shadow Paging Mechanism

● Pages, Memory frames, page table을 갖는 Paging 기법을 이용하는 시스템에서 이용

● current page table과 shadow page table 이용

● 기법

1. Transaction시작시 두 table의 내용은 동일
2. write(X, xj) 연산 수행시
   1) if (X-i-th(page table의 i번째 요소는 X가 있는 page를 가리킴) not in main memory)
      input(X);
   2) if (연산이 X-i-th의 첫연산?) {
      find an unused page on disk ;
      current page table의 i 번째 요소가 새로 할당된 page를 reference하게 함;
      } ;
   3) Main memory buffer 내에서 X의 값을 xj로 변경;
3. commit 수행전
   1) transaction에 의해 변경된 main memory의 모든 buffer page들을 disk로 출력
   2) current page table을 disk로 출력
   3) current page table의 disk address를 shadow page table의 disk address를 각는 고정위치로 기록

Vitrutal Memory 기법