Chapter 8: Deadlocks

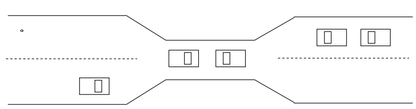
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

System Model

- Resource types:
  - Preemptable (can be taken away with no ill effect)
  - Nonpreemptable (cannot be taken away from current owner)
- Resource types $R_1, R_2, \ldots, R_m$
  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Introduction to Deadlocks

- 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
- Usually the event is release of a currently held resource
- None of the processes can …
  - run
  - release resources
  - be awakened

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Example

```
Void process_A(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
Void process_b(void) {
 down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
```

```
Void process_A(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
Void process_b(void) {
 down(&resource_2);
  down(&resource_1);
  use_both_resources();
  up(&resource_1);
  up(&resource_2);
}
```


Resource-Allocation Graph

A set of vertices V and a set of edges E.

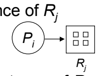
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

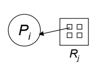
- Process

- Resource Type with 4 instances

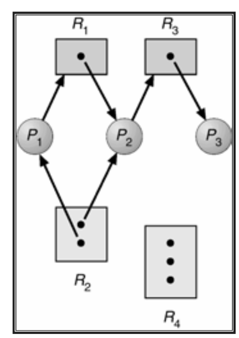
- P_i requests instance of R_j



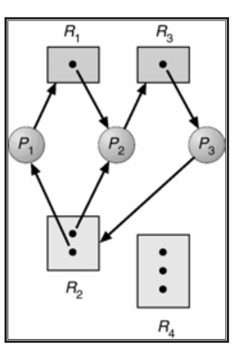
- P_i is holding an instance of R_j



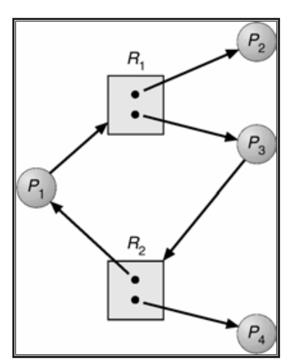
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

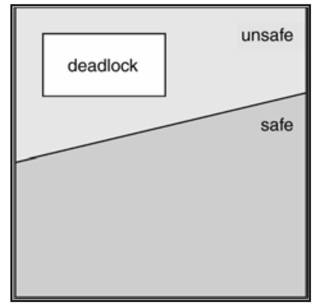
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence <P_1, P_2, …, P_n> is safe if for each P_i, the resources that *Pi* can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<I$.
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

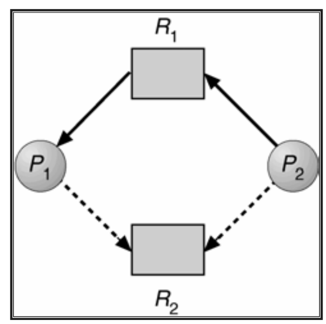
Safe, Unsafe , Deadlock State



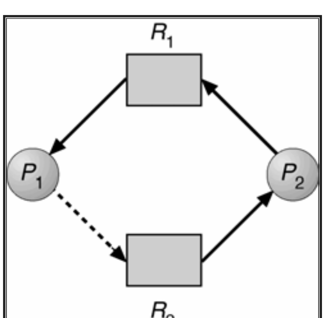
Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Safe and Unsafe States (1)

(a)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(b)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(c)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(d)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(e)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

Demonstration that the state in (a) is safe

Safe and Unsafe States (2)

(a)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(b)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(c)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(d)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

Demonstration that the sate in b is not safe

The Banker's Algorithm for a Single Resource

(a)

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(b)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(c)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

- Three resource allocation states
  - safe
  - safe
  - unsafe

Banker's Algorithm for Multiple Resources

Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Banker's Algorithm

1. Look for a row R, whose unmet resources are smaller than or equal to available resources A. If no such row exists, the system will eventually deadlock, since no process can run to completion
2. Assume the process of the row chosen requests all the resources it needs an finishes. Mark that process as terminated and add all its resources to the A vector.
3. Repeat until all processes are marked terminated, in which case the initial state was safe, or until a deadlock occurs, in which case it was not.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- *Available:* Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.
- *Max:* $n \times m$ matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- *Allocation:* $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.
- *Need:* $n \times m$ matrix. If *Need*$[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j].$$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
   Work = Available
   Finish [*i*] = *false* for *i* - 1,3, …, *n*.
2. Find and *i* such that both:
   (a) *Finish* [*i*] = *false*
   (b) $Need_i \le Work$
   If no such *i* exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.
4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $Request_i$ [*j*] = *k* then process P_i wants *k* instances of resource type R_j.

1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \le Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:
   $Available = Available = Request_i;$
   $Allocation_i = Allocation_i + Request_i;$
   $Need_i = Need_i - Request_i;$
   - *If safe* $\Rightarrow$ *the resources are allocated to* P_i.
   - *If unsafe* $\Rightarrow$ P_i *must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types *A* (10 instances), *B* (5instances, and *C* (7 instances).
- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix. Need is defined to be Max – Allocation.

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Example P_1 Request (1,0,2) (Cont.)

- Check that Request $\le$ Available (that is, $(1,0,2) \le (3,3,2) \Rightarrow$ *true*.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

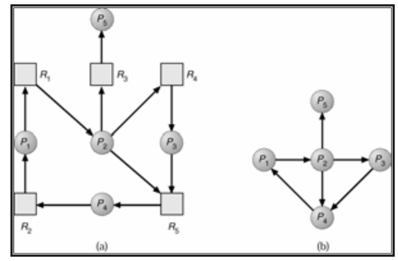
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

- *Available:* A vector of length *m* indicates the number of available resources of each type.
- *Allocation:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- *Request:* An *n* x *m* matrix indicates the current request of each process. If *Request* [i_j] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work = Available*
   (b) For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   If no such *i* exists, go to step 4.

Detection Algorithm (Cont.)

3. *Work = Work +* $Allocation_i$
   Finish[*i*] = *true*
   go to step 2.
4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type C.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

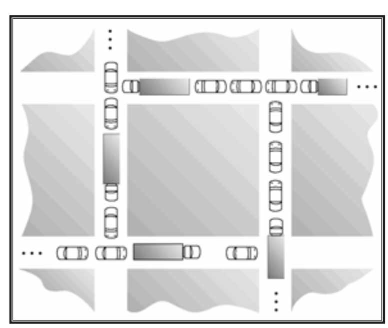
- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection

  allowing the use of the optimal approach for each of resources in the system.
- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.

Traffic Deadlock for Exercise 8.4



Synchronizer Class

```
typedef pthread_t threadId;
typedef int synchronizerId;

class Synchronizer {
public:
  Synchronizer();
  virtual ~Synchronizer();
  virtual void acquire();
  virtual void release();
private:
  pthread_mutex_t M;
  // deliberately not implemented
  Synchronizer(const Synchronizer &);
  Synchronizer &operator=(const Synchronizer &);
};
```


Instrumented Synchronizer

```
class instrumentedSynchronizer : public Synchronizer {
public:
  instrumentedSynchronizer();
  ~instrumentedSynchronizer();
  void acquire();
  void release();
private:
  synchronizerId myid;
  // deliberately not implemented
  instrumentedSynchronizer(const instrumentedSynchronizer &);
  instrumentedSynchronizer &operator=(const instrumentedSynchronizer &);
};

extern threadId CurThreadId();
extern bool havePotentialDeadlock();
```


Instrumented Synchronizer

```
void instrumentedSynchronizer::acquire() {
  threadId t = CurThreadId();
  GlobalLockMonitor()->observeAcquireAttempt(myid,t);
  Synchronizer::acquire();
}

void instrumentedSynchronizer::release() {
  threadId t = CurThreadId();
  GlobalLockMonitor()->observeRelease(myid,t);
  Synchronizer::release();
}
```


Lock Monitor

```
class LockMonitor : private Synchronizer {
public:
  LockMonitor() ;
  bool isAcyclic() ;
  int assignId() ;
  void observeAcquireAttempt(synchronizerId sid, threadId t) ;
  void observeRelease(synchronizerId sid, threadId t) ;
private:
  directedGraph<synchronizerId> priorSyncMap;
  synchronizerId nextId;
  incidenceMap<synchronizerId,threadId> syncOwnedBy;
};
```


Lock Monitor

```
void observeAcquireAttempt(synchronizerId sid, threadId t) {
  acquire();
  if(!syncOwnedBy.isPair(sid,t)) {
    vector<synchronizerId> *u = syncOwnedBy.left(t);
    if(u!=NULL) {
      for(unsigned i = 0;i<u->size();++i) {
        priorSyncMap.insertPair(sid,(*u)[i]);
      }
      delete u;
    }
    syncOwnedBy.insertPair(sid,t);
  }
  release();
};
void observeRelease(synchronizerId sid, threadId t) {
  acquire();
  syncOwnedBy.erasePair(sid,t);
  release();
};
```


Example

```
static instrumentedSynchronizer a,b,c;
static void *R1(void *) {
  cout << "R1 Thread " << CurThreadId() << " try a.acquire();\n";
  a.acquire();
  cout << "R1 Thread " << CurThreadId() << " try b.acquire();\n";
  b.acquire();
  sleep(2);
  cout << "R1 Thread " << CurThreadId() << " try a.release();\n";
  a.release();
  cout << "R1 Thread " << CurThreadId() << " try b.release();\n";
  b.release();
  return NULL;
}
static void *R2(void *) {
  sleep(1);
  cout << "R2 Thread " << CurThreadId() << " try b.acquire();\n";
  b.acquire();
  cout << "R2 Thread " << CurThreadId() << " try c.acquire();\n";
  c.acquire();
  cout << "R2 Thread " << CurThreadId() << " try c.release();\n";
  c.release();
  cout << "R2 Thread " << CurThreadId() << " try b.release();\n";
  b.release();
  return NULL;
}
```

Example, continued

```
static void *R3(void *) {
  sleep(1);
  cout << "R3 Thread " << CurThreadId() << " try a.acquire();\n";
  a.acquire();
  cout << "R3 Thread " << CurThreadId() << " try c.acquire();\n";
  c.acquire();
  cout << "R3 Thread " << CurThreadId() << " try c.release();\n";
  c.release();
  cout << "R3 Thread " << CurThreadId() << " try a.release();\n";
  a.release();
  return NULL;
}
static void *R4(void *) {
  sleep(3);
  cout << "R4 Thread " << CurThreadId() << " try c.acquire();\n";
  c.acquire();
  cout << "R4 Thread " << CurThreadId() << " try a.acquire();\n";
  a.acquire();
  cout << "R4 Thread " << CurThreadId() << " try c.release();\n";
  c.release();
  cout << "R4 Thread " << CurThreadId() << " try a.release();\n";
  a.release();
  return NULL;
}
```


Example, continued

```
int main(int argc, char *argv[]) {
  bool fail = (argc>1);
  pthread_t T1,T2,TX;
  pthread_create(&T1,NULL,R1,NULL);
  pthread_create(&T2,NULL,R2,NULL);
  if(!fail) {
    pthread_create(&TX,NULL,R3,NULL);
  } else {
    pthread_create(&TX,NULL,R4,NULL);
  }
  sleep(5);
  dumpstat();
  return 0;
}
```


Version 1

R1 Thread 1026 try a->acquire();
R1 Thread 1026 try b->acquire();
R2 Thread 2051 try b->acquire();
R3 Thread 3076 try a->acquire();
R1 Thread 1026 try a->release();
R1 Thread 1026 try b->release();
R2 Thread 2051 try c->acquire();
R2 Thread 2051 try c->release();
R2 Thread 2051 try b->release();
R3 Thread 3076 try c->acquire();
R3 Thread 3076 try c->release();
R3 Thread 3076 try a->release();
do not see potential deadlock

Version 2

R1 Thread 1026 try a->acquire();
R1 Thread 1026 try b->acquire();
R2 Thread 2051 try b->acquire();
R1 Thread 1026 try a->release();
R1 Thread 1026 try b->release();
R2 Thread 2051 try c->acquire();
R2 Thread 2051 try c->release();
R2 Thread 2051 try b->release();
R4 Thread 3076 try c->acquire();
R4 Thread 3076 try a->acquire();
R4 Thread 3076 try c->release();
R4 Thread 3076 try a->release();
see potential deadlock