Common Concurrency Problems

What Types Of Bugs Exist?

- **Non-deadlock and deadlock**
- **Study by Lu et al. [ASPLOS'08]**
  - Non-deadlock bugs make up a majority of concurrency bugs

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: **Bugs In Modern Applications**

Non-Deadlock Bugs

- **Atomicity-Violation Bugs**
  - The desired serializability among multiple memory accesses is violated
  - i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution

```
Thread 1::
if (thd->proc_info) {
        ...
        fputs(thd->proc_info, ...);
        ...
}

Thread 2::
thd->proc_info = NULL;
```

```
pthread_mutex_t proc_info_lock
            = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
pthread_mutex_lock(&proc_info_lock);
if (thd->proc_info) {
        ...
        fputs(thd->proc_info, ...);
        ...
}
pthread_mutex_unlock(&proc_info_lock);

Thread 2::
pthread_mutex_lock(&proc_info_lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&proc_info_lock);
```

Non-Deadlock Bugs

- **Order-Violation Bugs**
  - The desired order between two memory accesses is flipped

```
Thread 1::
void init() {
...
mThread = PR_CreateThread(mMain, ...);
...
}

Thread 2::
void mMain(...) {
...
mState = mThread->State;
...
}
```

Thread 2 seems to assume that the variable mThread has already been initialized (and is notNULL);
➔ To enforce ordering, use **condition variables**

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
int mtInit = 0;

Thread 1::
void init() {
        ...
        mThread = PR_CreateThread(mMain, ...);

        // signal that the thread has been created...
        pthread_mutex_lock(&mtLock);
        mtInit = 1;
        pthread_cond_signal(&mtCond);
        pthread_mutex_unlock(&mtLock);
        ...
}

Thread 2::
void mMain(...) {
        ...
        // wait for the thread to be initialized...
        pthread_mutex_lock(&mtLock);
        while (mtInit == 0)
                pthread_cond_wait(&mtCond, &mtLock);
        pthread_mutex_unlock(&mtLock);
        mState = mThread->State;
        ...
}
```

Non-Deadlock Bugs: Summary

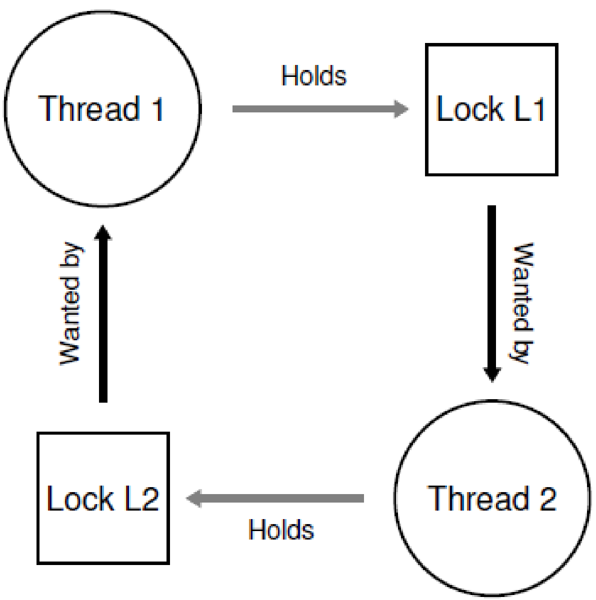
- A large fraction (97%) of non-deadlock bugs studied by Lu et al. are either atomicity or order violations.
  - By carefully thinking about these types of bug patterns, programmers can likely do a better job of avoiding them.
  - Automated code-checking tools will focus on these two types of bugs
- Unfortunately, not all bugs are as easily fixable as the examples
- Some require a deeper understanding of what the program is doing, or a larger amount of code or data structure reorganization to fix.

Deadlock Bugs

```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

```
Thread 2:
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```

deadlock does not necessarily occur; rather, it may occur

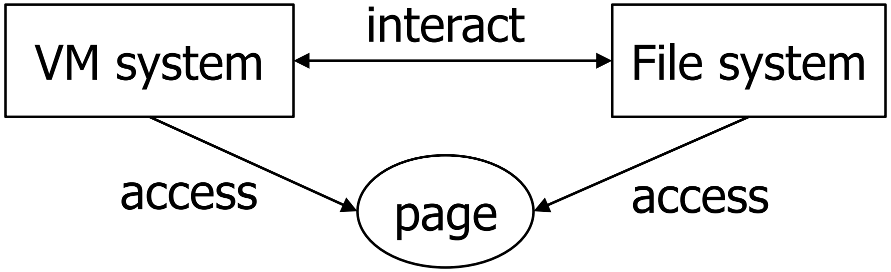


Deadlock dependency graph

How about if Thread 1 and 2 both made sure to grab locks in the same order?

Why Do Deadlocks Occur?

- Complex dependencies between components



- Encapsulation
  - hide details of implementations and thus make software easier to build in a modular way
  - E.g., Java Vector class and its method AddAll().

```
Vector v1, v2;
v1.AddAll(v2);
```

•Internally, because the method needs to be multi-thread safe, locks for both v1 and v2 need to be acquired.

•The routine acquires the locks in some arbitrary order

– lock(v1) then lock(v2)

•If some other thread calls v2.AddAll(v1) at nearly the same time, lock(v2) then lock(v1)

•Hidden from the calling application.

Conditions for Deadlock

- **Mutual exclusion**
  - Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock)
- **Hold-and-wait**
  - Threads hold resources allocated to them while waiting for additional resources
- **No preemption**
  - Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- **Circular wait**
  - There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

- If any of these four conditions are not met, deadlock cannot occur.
- To prevent deadlock, prevent one of the above conditions

Prevention - CircularWait

- **Total ordering** on lock acquisition.
  - For example, if there are only two locks in the system (L1 and L2), you can prevent deadlock by always acquiring L1 before L2.
- **Partial ordering** can be a useful in more complex systems
  - Linux File Memory Map Code
  - Ten different groups of lock acquisition orders

```
“i_mutex ➔ i_mmap_mutex”
“i_mmap_mutex ➔ private_lock ➔ swap_lock ➔ mapping->tree
  lock”
```

- **Enforce Lock Ordering by Lock Address** (solve encapsulation problem)

```
if (m1 > m2) { // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
```

- Ordering require careful design of locking strategies and must be constructed with great care.
- Ordering is just a convention, and a sloppy programmer can easily ignore the locking protocol and potentially cause deadlock

Partial lock ordering in /mm/filemap.c

```
/*
 * Lock ordering:
 *
 *  ->i_mmap_rwsem              (truncate_pagecache)
 *    ->private_lock            (__free_pte->__set_page_dirty_buffers)
 *      ->swap_lock             (exclusive_swap_page, others)
 *        ->mapping->tree_lock
 *
 *  ->i_mutex
 *    ->i_mmap_rwsem            (truncate->unmap_mapping_range)
 *
 *  ->mmap_sem
 *    ->i_mmap_rwsem
 *      ->page_table_lock or pte_lock  (various, mainly in memory.c)
 *        ->mapping->tree_lock   (arch-dependent flush_dcache_mmap_lock)
 *
 *  ->mmap_sem
 *    ->lock_page               (access_process_vm)
 *
 *  ->i_mutex                   (generic_perform_write)
 *    ->mmap_sem                (fault_in_pages_readable->do_page_fault)
 *
 *  bdi->wb.list_lock
 *    sb_lock                   (fs/fs-writeback.c)
 *    ->mapping->tree_lock      (__sync_single_inode)
 *
 *  ->i_mmap_rwsem
 *    ->anon_vma.lock           (vma_adjust)
 *
 *  ->anon_vma.lock
 *    ->page_table_lock or pte_lock  (anon_vma_prepare and various)
 *
 *  ->page_table_lock or pte_lock
 *    ->swap_lock               (try_to_unmap_one)
 *    ->private_lock            (try_to_unmap_one)
 *    ->tree_lock               (try_to_unmap_one)
 *    ->zone_lru_lock(zone)     (follow_page->mark_page_accessed)
 *    ->zone_lru_lock(zone)     (check_pte_range->isolate_lru_page)
 *    ->private_lock            (page_remove_rmap->set_page_dirty)
 *    ->tree_lock               (page_remove_rmap->set_page_dirty)
 *    bdi.wb->list_lock         (page_remove_rmap->set_page_dirty)
 *    ->inode->i_lock           (page_remove_rmap->set_page_dirty)
 *    ->memcg->move_lock        (page_remove_rmap->lock_page_memcg)
 *    bdi.wb->list_lock         (zap_pte_range->set_page_dirty)
 *    ->inode->i_lock           (zap_pte_range->set_page_dirty)
 *    ->private_lock            (zap_pte_range->__set_page_dirty_buffers)
 *
 * ->i_mmap_rwsem
 *   ->tasklist_lock            (memory_failure, collect_procs_ao)
 */
```

Prevention - Hold-and-wait

- Acquiring all locks at once, atomically

```
pthread_mutex_lock(prevention); // begin lock acquisition
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

- Drawback
  - Must know exactly which locks must be held and acquire them ahead of time.
  - Decrease concurrency as all locks must be acquired early on (at once) instead of when they are truly needed.

Prevention - No Preemption

- `pthread_mutex_trylock()`
  - either grabs the lock (if it is available) and returns success
  - or returns an error code indicating the lock is held
    - Can try again later if you want to grab that lock.

```
top:
        pthread_mutex_lock(L1);
        if (pthread_mutex_trylock(L2) != 0) {
                pthread_mutex_unlock(L1);
                goto top;
        }
```

- **Problems**
  - Livelock
    - two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks
    - add a random delay before looping back and trying the entire thing over again, thus decreasing the odds of repeated interference among competing threads
  - Encapsulation
    - How to release the resource allocated before calling the routine?

Prevention - Mutual Exclusion

- In general, this is difficult, because the code we wish to run does indeed have critical sections
- **lock-free** (and related **wait-free**) approaches
  - using powerful hardware instructions such as compare&swap

```
int CompareAndSwap(int *address, int expected, int new) {
    if (*address == expected) {
        *address = new;
        return 1; // success
    }
    return 0; // failure
}
```

```
void AtomicIncrement(int *value, int amount) {
    do {
        int old = *value;
    } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

Prevention - Mutual Exclusion

```
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    pthread_mutex_lock(listlock); // begin critical section
    n->next = head;
    head = n;
    pthread_mutex_unlock(listlock); // end critical section
}
```

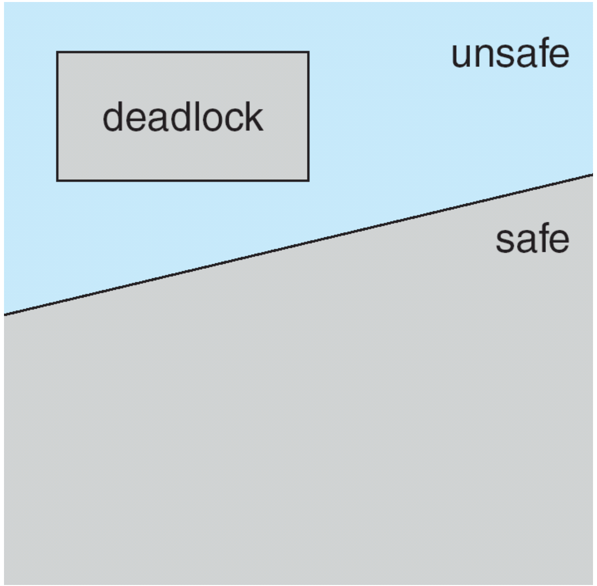
Why did we grab the lock so late, instead of right when entering insert()?

```
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
     assert(n != NULL);
    n->value = value;
    do {
          n->next = head;
    } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

This will fail if some other thread successfully swapped in a new head in the meanwhile; then retry!

Deadlock Avoidance

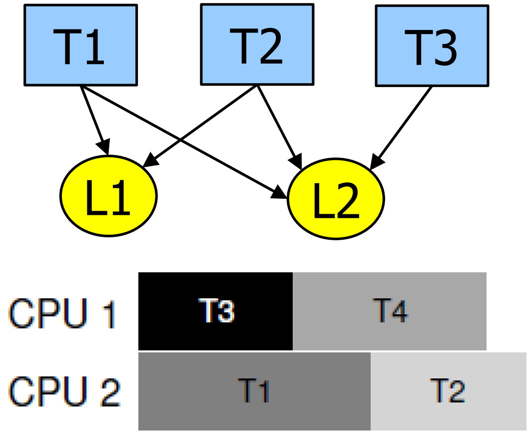
- **Deadlock avoidance method**
  - Monitor system states continuously
    - Dynamically examines the resource allocation state
  - Let the system always be in states that can allocate resources to each process in some order and still avoid a deadlock
  - Always keep the system in safe states

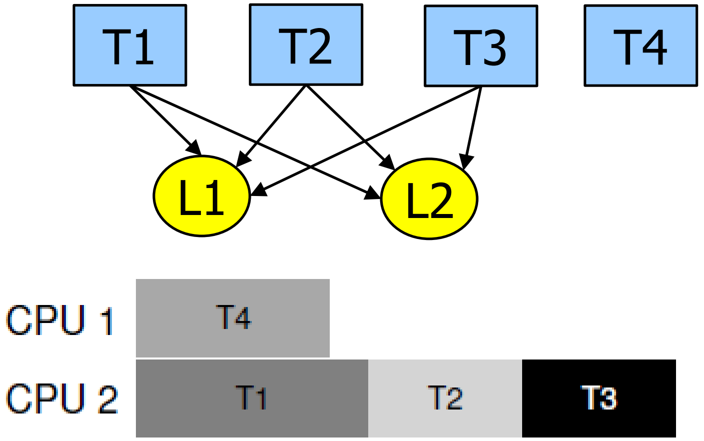


If a system is in safe state ⇒ no deadlocks
If a system is in unsafe state ⇒ possibility of deadlock
Avoidance ⇒ ensure that a system will never enter an unsafe state.

Deadlock Avoidance via Scheduling

- Schedules threads in a way as to guarantee no deadlock can occur
- A smart scheduler could make two threads are not run at the same time if they might cause a deadlock
- e.g., Dijkstra's Banker's Algorithm
- Conservative approach
  - it may have been possible to run these tasks concurrently
  - Limit concurrency ➔ High cost
- Useful in very limited environments, Not widely used
  - Need full knowledge of the entire set of tasks

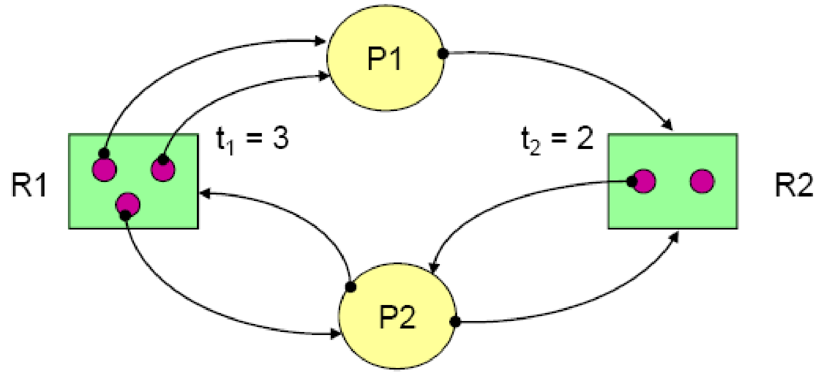




Detect and Recover

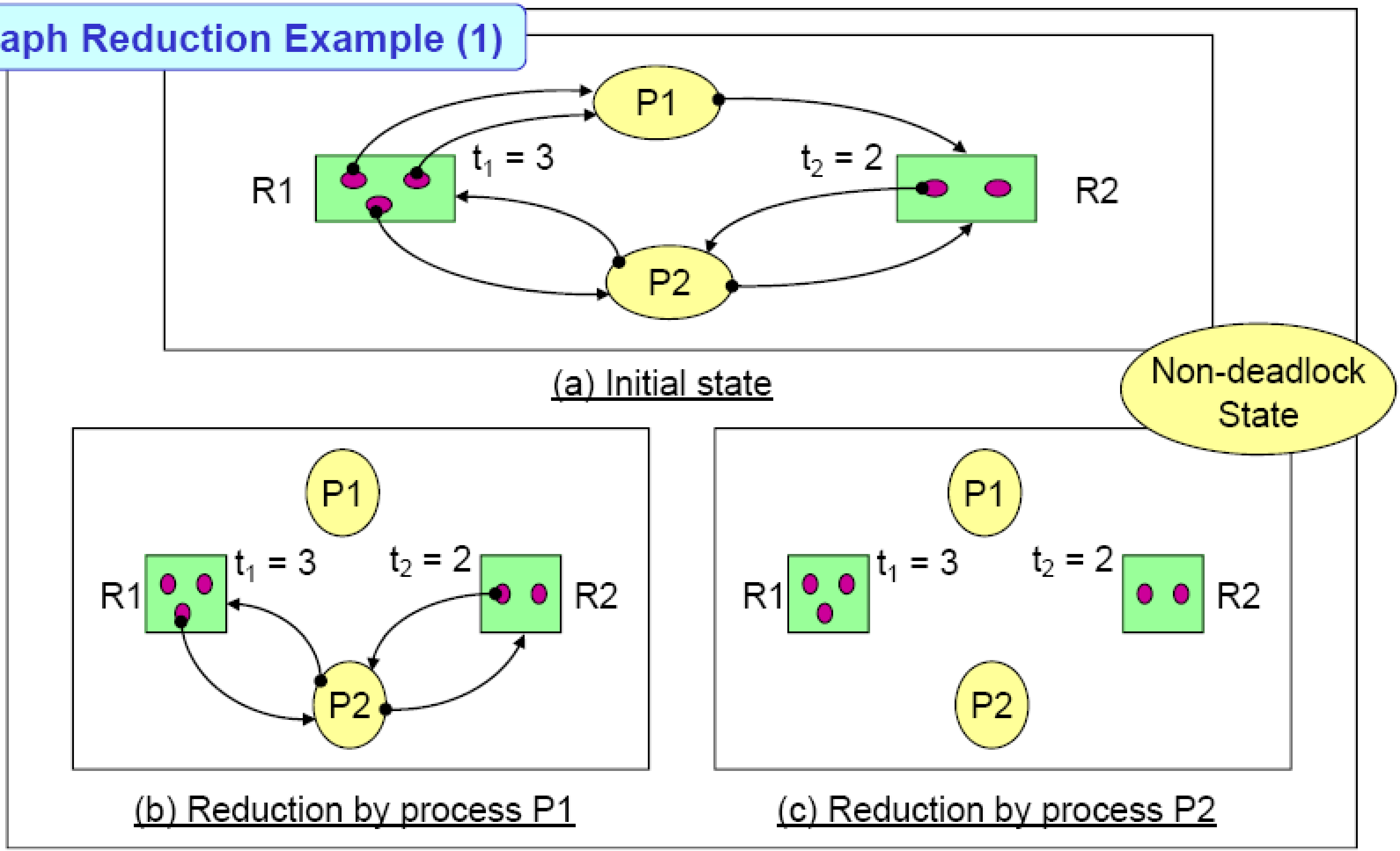
- Allow deadlocks to occasionally occur, and then take some action once such a deadlock has been detected

- If deadlocks are rare, it is better to use a pragmatic solution
  - "Not everything worth doing is worth doing well" - Tom West
  - If an OS froze once a year, you would just reboot it
  - Many database systems employ a deadlock detector, which runs periodically, building a resource graph and checking it for cycles. In the event of a cycle (deadlock), the system needs to be restarted.

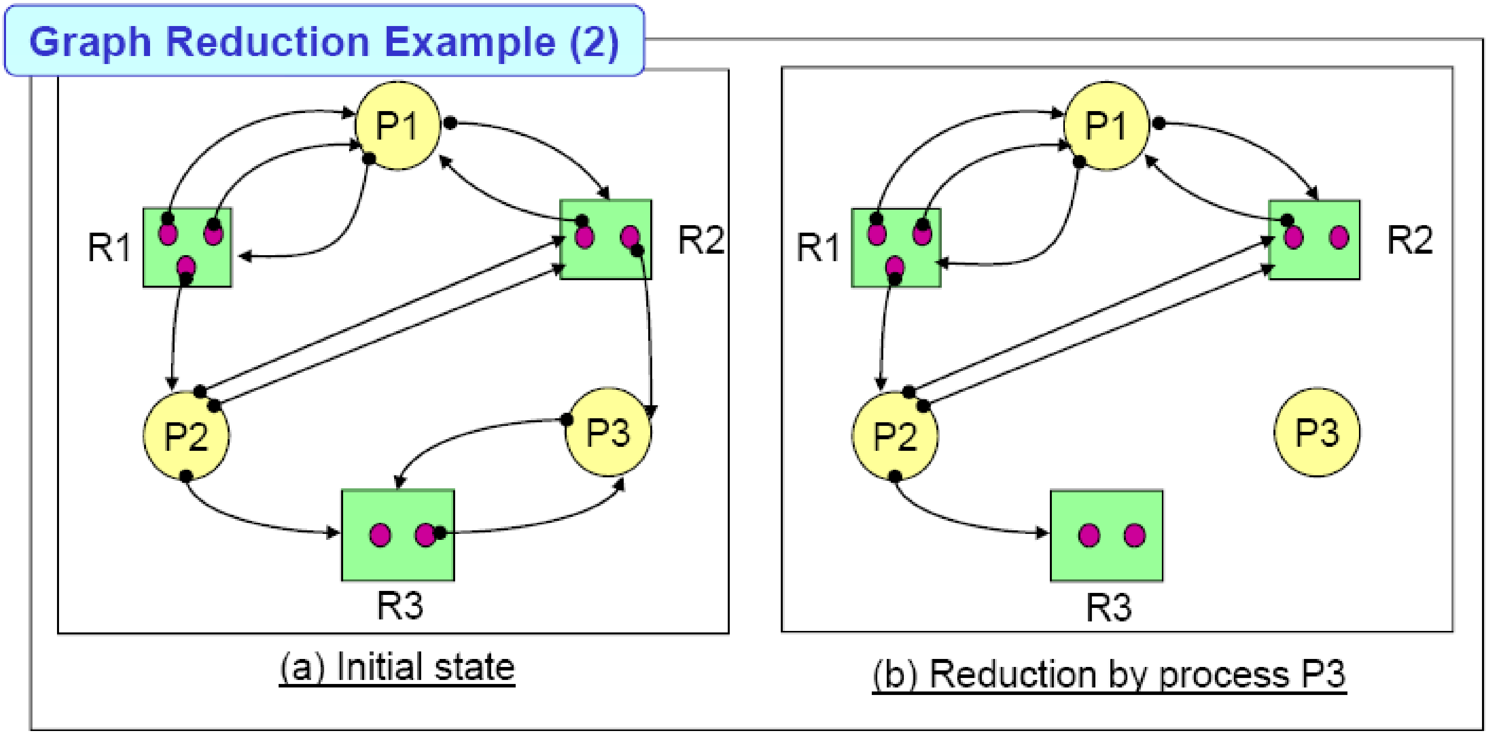


Deadlock Detection

Graph Reduction Example (1)



Deadlock Detection



Process P1 and P2 are blocked state in State-(b)
- Reduction is not possible any more
- Initial state is deadlocked

Deadlock Recovery

- **Process termination**
  - Terminates (abnormally) one or more processes to break the circular wait
    - Terminated processes are restarted or rolled back afterwards
- **Resource preemption**
  - Preempts some resources from processes that currently owns them and gives these resources to other processes until the deadlock cycle is broken
    - Election of the resources to be preempted to eliminate the deadlock
    - Preemption and reassignment of the resources

Summary

- Deadlock prevention
  - Deny a necessary condition for deadlocks
  - Deadlock cannot occur in the system
  - Serious resource waste
  - Not practical
- Deadlock avoidance
  - Need full knowledge of the entire set of tasks
  - Considers worst case
- Deadlock detection & recovery
  - Checks whether current state has deadlocked processes or not

Homework

- Homework in Chap 32 (Deadlock)