COS40003 Concurrent Programming

Lecture 9: Concurrency Bugs

CRICOS 00111D
TOID 3059

Outline

- Concurrency Bugs
- Non-deadlock bugs
  - Two popular ones
- Deadlock
  - Introduction of Deadlock
  - Deadlock conditions
  - Deadlock Prevention
  - Deadlock Detect and Recover

One interesting question:

- A question by a student in lab
  - What we have learned are low level programming primitives.
  - When I go to work, do I really need to program at such a level?
  - I assume those controls are encapsulated already? So what is the point of learning these low level primitives.

One interesting question:

- Why to study low level primitives?
- Answer:
  - Make you knowledgeable and more skillful
  - Avoid writing bad code
  - Be able to find out and be able fix bugs caused by bad concurrency control
- Do we really have that many bugs?

What types Of concurrency bugs exist in software?

- [Lu+08] examined major and important open-source software
  - MySQL (database management system), Apache (web server), Mozilla (client browser), OpenOffice (office suite)
- Concurrency bugs that have been found and fixed in each of these code bases are shown in the table

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

- [L+08] “Learning from Mistakes—A Comprehensive Study on Real World Concurrency Bug Characteristics”
  - Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou

Outline

- Concurrency Bugs
- Non-deadlock bugs
  - Two popular ones
- Deadlock
  - Introduction of Deadlock
  - Deadlock conditions
  - Deadlock Prevention
  - Deadlock Detect and Recover

Non-Deadlock Bugs

- Using two examples, we study:
  - What types of bugs are they?
  - How do they arise?
  - How can we fix them?

Non-Deadlock Bugs: example 1(a)

- Found in MySQL:

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

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

- Type 1: Atomicity violation

What is the problem?
If Thread 1 performs the check, but then is interrupted before the call to fputs, there is a context switch. Thread 2 then runs and set the pointer to NULL; When Thread 1 resumes, it will use a NULL pointer.
How to fix it?

Non-Deadlock Bugs: example 1(b)

```
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);
```

Question:

In Thread 2, do we need a lock for the single instruction:

“thd->proc_info = NULL;”

Yes

“thd->proc_info = NULL;”

Consists of:

(1) Move thd->proc to CPU;

(2) Set the pointer value to Null;

(3) Write the Null value back to thd->proc in memory

Non-Deadlock Bugs: example 2(a)

```
Thread 1::
void init() {
    ...
    mThread = PR_CreateThread(f1, ...);
    ...
}

Thread 2::
 {
   ...
   mState = mThread->State;
   ...
}
```

- Type 2: order violation

What is the problem?

Thread 2 assumes that the variable mThread has already been initialized;

However, if Thread 2 runs before Thread 1, it will likely crash with a NULL pointer.

How to fix it?

Non-Deadlock Bugs: example 2(b)

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

Thread 1::
   void init() {
   ...
   mThread = PR_CreateThread(f1, ...);

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

```
Thread 2::
  {
  ...
  // 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;
  ...
}
```

Solution: enforce ordering

Use condition variable(s) or semaphore(s)

Non-Deadlock Bugs: summary

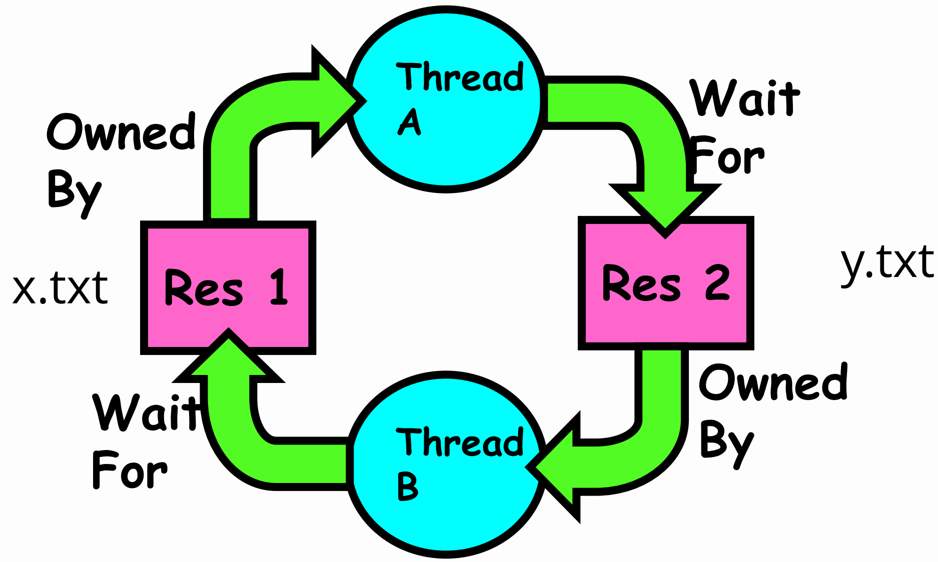
- A large fraction (97%) of non-deadlock bugs studied by Lu et al. are either atomicity or order violations.
- Not all bugs are as easily fixable as the given examples. Some require a deeper understanding of what the program is doing, or a larger amount of code or data structure reorganization to fix.
- [L+08] "Learning from Mistakes—A Comprehensive Study on Real World Concurrency Bug Characteristics"
  - Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou

Outline

- Concurrency Bugs
- Non-deadlock bugs
  - atomicity or order violations
- Deadlock
  - Introduction of Deadlock
  - Deadlock conditions
  - Deadlock Prevention
  - Deadlock Detect and Recover

Deadlock

Thread A owns Res 1 and is waiting for Res 2
Thread B owns Res 2 and is waiting for Res 1



Thread A has opened x.txt and wants to write some results into y.txt

Thread B has opened y.txt and wants to write some results into x.txt

Deadlocks - A real world example

Deadlock

Each car can be regarded as an individual thread and the roundabout (street) is the shared object (resource)

Why Do Deadlocks Occur?

- Reason One:
  - In large code bases, complex dependencies arise between components.
- Reason Two: due to the nature of **encapsulation:**
  - As software developers, we are taught to hide details of implementations and thus make software easier to build in a modular way.
  - Unfortunately, such modularity does not mesh well with locking.

Why Do Deadlocks Occur?

- For example, take the Java Vector class and the method AddAll().

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

- AddAll() acquires the locks of v1 and v2 in arbitrary order (say v1then v2) in order to add the contents of v2 to v1.
- If some other thread calls v2.AddAll(v1) at nearly the same time, we have the potential for deadlock
- This is hidden from the calling application

Outline

- Concurrency Bugs
- Non-deadlock bugs
  - atomicity or order violations
- Deadlock
  - Introduction of Deadlock
  - Deadlock conditions
  - Deadlock Prevention
  - Deadlock Detect and Recover

Deadlock conditions

- **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 (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
- **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.
- **Summary:** If any of these four conditions is not met, deadlock cannot occur.

Outline

- Concurrency Bugs
- Non-deadlock bugs
  - atomicity or order violations
- Deadlock
  - Introduction of Deadlock
  - Deadlock conditions
  - Deadlock Prevention
  - Deadlock Detect and Recover

Deadlock Prevention

- Recall four necessary conditions:

- Preventing: Circular Wait
- Preventing: Hold-and-wait
- Preventing: No preemption
- Preventing: Mutual exclusion

Deadlock prevention: Circular Wait

- Avoid circular wait: the most practical prevention technique

- How?

- Solution:
  - Enforce a **total ordering** on lock acquisition.
  - Enforce a **partial ordering** on lock acquisition is sufficient sometimes

Deadlock prevention: Circular Wait

- How can we ensure the order?
  - Better understand your programs, docs
  - A tip: a clever programmer can use the address of each lock as a way of ordering lock acquisition, regardless of which order they are passed in.

```
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);
}
```

Assume lock addresses are static,

may cause problems after memory fragment reorganization

Deadlock Prevention

- Recall four necessary conditions:

- Preventing: Circular Wait
- Preventing: Hold-and-wait
- Preventing: No preemption
- Preventing: Mutual exclusion

Deadlock prevention: Hold-and-Wait

- How?
- Candidate solution: atomically, acquire all locks at once.

```
pthread_mutex_lock(prevention);
// begin lock acquistion

pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...

pthread_mutex_unlock(prevention);
// end
```

Problem of acquire all locks at once.

- One: this technique is likely to decrease concurrency as all locks must be acquired early on (at once) instead of when they are truly needed.

- Two: recall, we have encapsulation: this means, when calling a routine, this approach requires us to know exactly which locks must be held and to acquire them ahead of time.

Deadlock Prevention

- Recall four necessary conditions:

- Preventing: Circular Wait
- Preventing: Hold-and-wait
- Preventing: No preemption
- Preventing: Mutual exclusion

Deadlock prevention: no preemption

- Idea: impose preemption
- How?
- Solution: pthread_mutex_trylock()
  - either grabs the lock (if it is available) and returns success
  - or returns an error code indicating the lock is held;

Deadlock prevention: no preemption

- Solution: pthread_mutex_trylock()

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

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

Now : deadlock-free!

Any possible problem?

Problem: livelock

It is possible (though perhaps unlikely) that two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks.

Solution: add a random delay before attempting again, avoiding the odd

Deadlock prevention: no preemption

- Difficulty of using a trylock approach.

- If one of these locks is buried in some routine that is getting called, the jump back to the beginning becomes more complex to implement. If the code had acquired some resources (other than L1) along the way, it must make sure to carefully release them as well;

- For example, if after acquiring L1, the code had allocated some memory, it would have to release that memory upon failure to acquire L2, before jumping back to the top to try the entire sequence again

Deadlock Prevention

- Recall four necessary conditions:

- Preventing: Circular Wait
- Preventing: Hold-and-wait
- Preventing: No preemption
- Preventing: Mutual exclusion

Deadlock prevention: mutual exclusion

- Idea: avoid the need for mutual exclusion at all
- How this can be possible?
- Recall:

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

- How about avoid using locks?

Deadlock prevention: mutual exclusion

- Solution:
  - lock-free approaches
  - using powerful hardware instructions, we can build data structures in a manner that does not require explicit locking.
- Recall: CompareAndSwap() - atomic

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

Deadlock prevention: mutual exclusion

- Increment a value by a certain amount

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

- Correct:
  - Instead of acquiring a lock, doing the update, and then releasing it,
  - The approach repeatedly tries to update the value to the new value and using the compare-and-swap.
  - In this manner, no lock is acquired, and no deadlock can arise (though livelock is still possibe).

Deadlock prevention: mutual exclusion

- A more complex example:
  - Lock-free list insertion using compare-and-swap
  - Page 10, Chapter 32, Operating Systems: Three Easy Pieces

- Building a lock-free data structure is non-trivial

Deadlock Prevention

- Recall four necessary conditions:

- Preventing: Circular Wait
- Preventing: Hold-and-wait
- Preventing: No preemption
- Preventing: Mutual exclusion
  - Hardware support
  - Careful scheduling

Deadlock Avoidance via Scheduling

- **Idea**: schedule threads in a way as to guarantee no deadlock can occur

- **Condition**: requires some global knowledge of which locks different threads might grab during their execution

Example: deadlock avoidance via scheduling

- Threads: T1, T2, T3, T4
- Locks: L1, L2
- Processors: P1, P2
- Which threads need which locks (the table below):

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

Example I: deadlock avoidance via scheduling

- Threads: T1, T2, T3, T4
- Locks: L1, L2
- Processors: P1, P2
- Which threads need which locks (the table below):

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

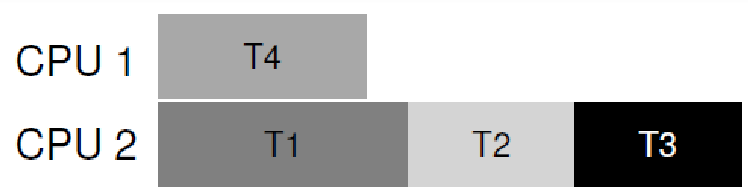
- Possible scheduling:

Example II: deadlock avoidance via scheduling

- Threads: T1, T2, T3, T4
- Locks: L1, L2
- Processors: P1, P2
- Which threads need which locks (the table below):

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

- Possible scheduling:



Summary: deadlock avoidance via scheduling

- Unfortunately, this type of approaches are only useful in very limited environments, because

- (1) one has to have full knowledge of the entire set of tasks and the locks that they need.
- (2) Such approaches can limit concurrency.

- Thus, avoidance of deadlock via scheduling is not a widely-used general-purpose solution.

Outline

- Concurrency Bugs
- Non-deadlock bugs
  - atomicity or order violations
- Deadlock
  - Introduction of Deadlock
  - Deadlock conditions
  - Deadlock Prevention
  - Deadlock Detect and Recover

Deadlock: Detect and Recover

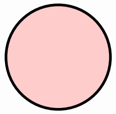
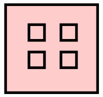
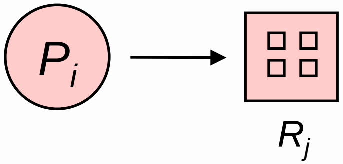
- One final general strategy is to allow deadlocks to occasionally occur, and then take some action once such a deadlock has been detected
  - First solution: simply reboot
  - Second solution: in many database systems, a deadlock detector runs periodically, building a resource graph and checking it for cycles

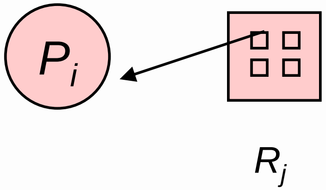
Deadlock detection

- System Model Setup:
- Resource types R_1, R_2, . . ., R_m

  CPU cycles, memory space, I/O devices
- Each resource type R_i has *1 or more* instances
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

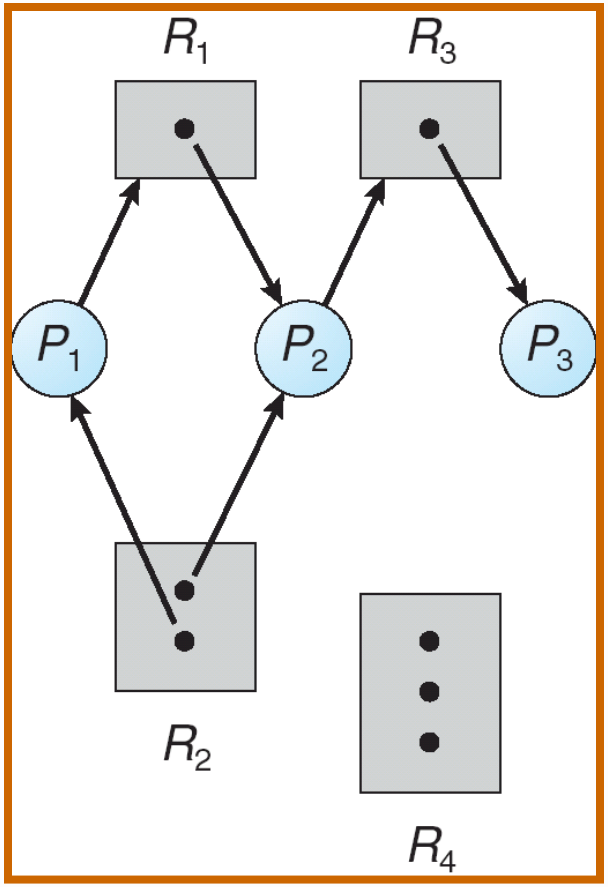
Resource-Allocation Graph

- Thread 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

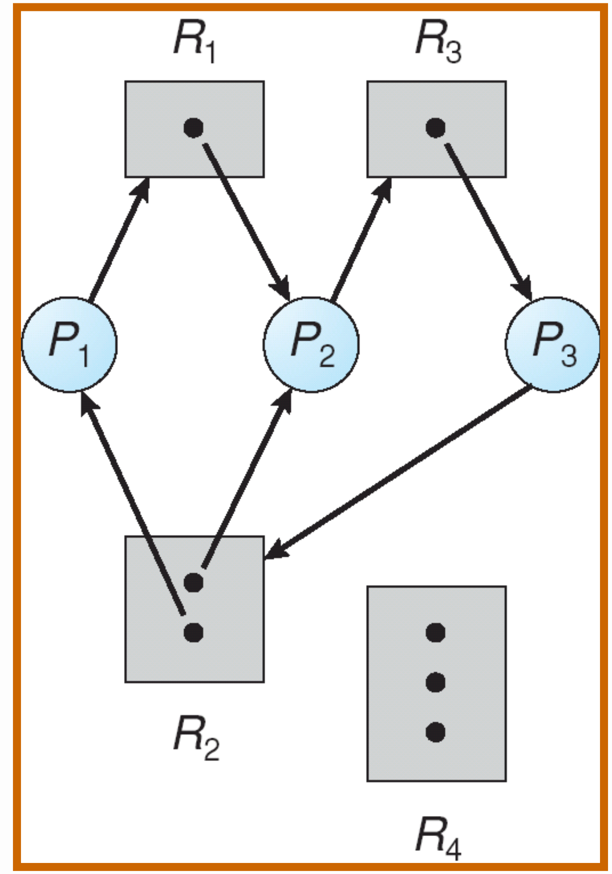
- P_i is holding an instance of R_j 

Resource allocation graph with a deadlock

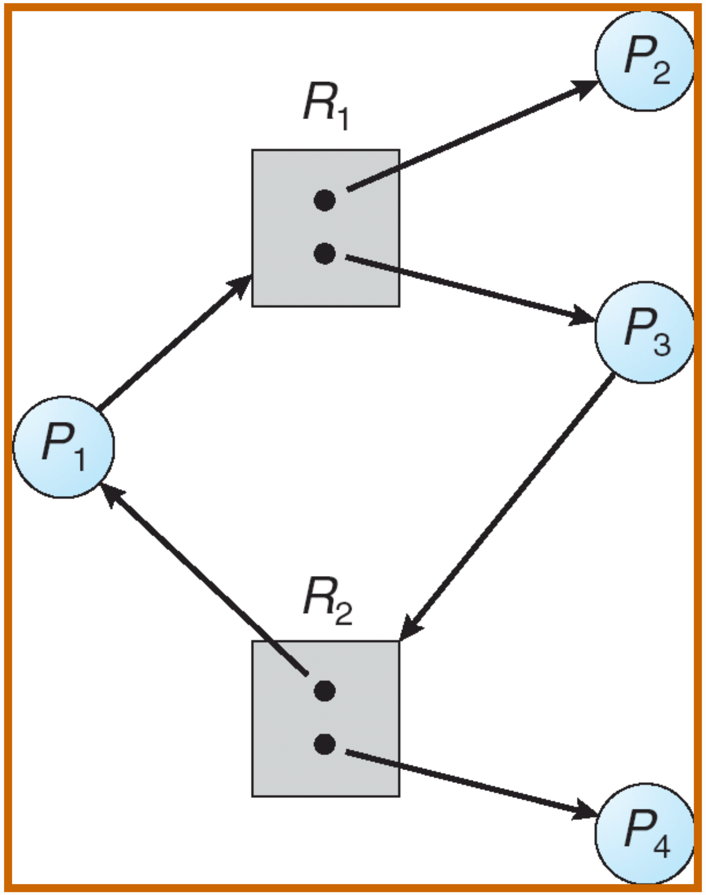
Before P_3 requested an instance of R_2



After P_3 requested an instance of R_2



Graph with a cycle but no deadlock



Thread P_4 may release its instance of resource type R_2. That resource can then be allocated to P3, thereby breaking the cycle.

Relationship of cycles to deadlocks

- Simple conclusions:
- If a resource allocation graph contains **no** cycles $\Rightarrow$ no deadlock
- If a resource allocation graph contains a cycle and if **only one** instance exists per resource type $\Rightarrow$ deadlock
- If a resource allocation graph contains a cycle and and if **several** instances exists per resource type $\Rightarrow$ possibility of deadlock

Acknowledgement

- Chapter 32
  - Operating Systems: Three Easy Pieces

Questions?