CS 3214: Computer Systems
Lecture 17: Concurrency Bugs

Instructor: Huaicheng Li

Oct 18 2022

VIRGINIA TECH™

Concurrency Problems

- Concurrency bugs
- What types of bugs are there?
  - Deadlocks
  - Non-deadlocks

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

Source: Lu'08

Non-Deadlock Bugs

- What types are these?
  - Atomicity-Violation Bugs

Thread 1:

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

Thread 2:

```
thd->proc_info = NULL;
```

Non-Deadlock Bugs

- What types are these?
  - Atomicity-Violation Bugs

```
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

- What types are these?
  - Atomicity-Violation Bugs
  - Order-Violation Bugs

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

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

Non-Deadlock Bugs

- What types are these?
  - Atomicity-Violation Bugs
  - Order-Violation Bugs

Thread 1:

```
void init () {
    mThread = PR_CreateThread(mMain, ...);
ç
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
}
```

Thread 2:

```
void mMain(...) {
pthread_mutex_lock(&mtLock);
while (mtInit == 0)
    pthread_cond_wait(&mtCond, &mtLock);
pthread_mutex_unlock(&mtLock);
    mState = mThread->State;
}
```

Deadlock

- **Thread perspective**
  - A condition in which one or more related threads are blocked waiting for an event that will never occur because the blocked threads would be the ones to cause it.
- **Resource perspective**
  - Threads are blocked waiting for resources that will never be granted because they are held by threads currently requesting resources.
- Resource contention or lack of communication / signaling

Deadlocks

- An example

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

Typical for deadlock is that

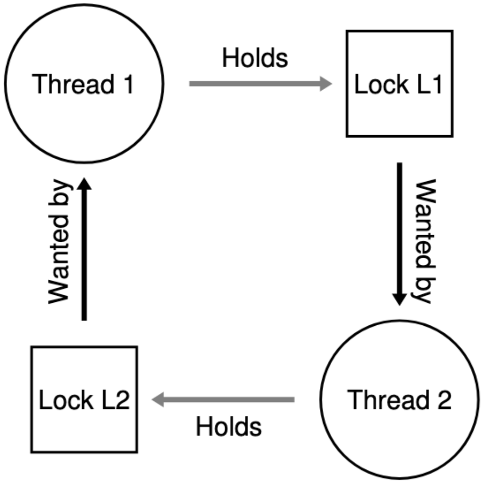
(a) threads cannot make forward progress
(b) threads cannot easily back out

Why Do Deadlocks Occur?

- In large code bases, complex dependencies arise …
  - e.g., Linux: mm <-> fs
- Nature of encapsulation (for modularity)
  - *e.g., Seemingly innocuous interfaces almost invite you to deadlock*

Java Vector class: Vector v1, v2;

Thread 1:	***Thread 2:***
v1.AddAll(v2);	v2.AddAll(v1);


Thread 1
Holds
Lock L1
Wanted by
Thread 2
Holds
Lock L2
Wanted by

Resource Deadlock Detection

- Will focus on deadlocks involving reusable resources (e.g., mutexes)
- Reliable after-the-fact deadlock detection requires access to resource allocation graph:
  - Nodes are either processes or resources with 2 types of edges
  - From resource R_i to process P_k: process P_k holds resource R_i
  - From process P_k to resource R_i: process P_k is trying to acquire resource R_i
- In practice, finding this graph can be difficult, though some debuggers provide it, e.g. Windows [URL]

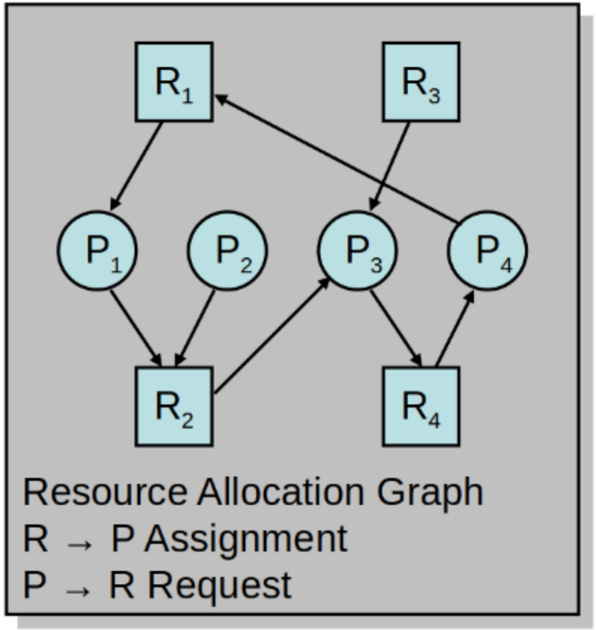


Figure 2: Resource Allocation Graph

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 (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.

Strategies for Dealing with Deadlocks

- Deadlock recovery
  - e.g., after the fact
  - Preempt access to resource (if possible)
  - Back process up: expensive, require checkpointing and/or transactions
  - Kill involved processes/threads until deadlock is resolved (Very tricky)
  - Kill all processes/threads involved
  - Reboot ...
- Deadlock prevention
  - e.g., remove one of the necessary conditions
  - ***Deadlock cannot occur if one of the necessary conditions is removed***
- Deadlock avoidance
  - e.g., adopt a strategy if none of the necessary conditions can be removed
  - Not practical, only therectial

Deadlock Prevention

- Circular Wait
  - Write your locking code to avoid circular wait
  - Total ordering on lock acqusition (e.g., always lock(L1) before lock(L2))
  - Partial ordering
    - Document locking order, (e.g., Linux mm code)
    - *Create a partial order of all resources that may be held simultaneously - e.g., by taking their addresses; example: C++17 std::scoped lock*
  - Both ***total and partial ordering*** require careful design of locking strategies and must be constructed with great care

Deadlock Prevention

- Hold-and-wait
  - How about acquiring all locks at once, atomically
    - No thread switch happening in the midst of lock acquisition
  - Unfortunately, solution is problematic ...
    - Encapsulation (against us): we must know beforehand which locks to acquire
    - Coarse granualarity, static vs. on-demand, decreased concurrency

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

Deadlock Prevention

- No Preemption
  - Preempt access to resource - difficult to write code that is robust in the presence of such preemption
  - A relaxed lock acquisition: pthread_mutex_trylock()
    - If lock available, grab the lock, return success
    - Else, returns an error code indicating lock held by others
  - However, one new problem arise: **livelock**
    - e.g., 2 threads executing the same trylock logics, repeatedly, but no progress …
    - Possible fix: add random delay before looping back
    - Even so, it's not a "preemption" of locks, rather gracefully to allow developer to back out

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

Deadlock Prevention

- Mutal Exclusion
  - Lock-free data structures!
    - Use hardware instructions to build structures requiring no explicit locking ...

```
int CompareAndSwap(int *addr, int expected, int new) {
    if (*addr == expected) {
        *addr = new;
        return 1; // success

    return 0; // failure
}
```

Deadlock Prevention

- Mutal Exclusion
  - Lock-free data structures!
    - Use hardware instructions to build structures requiring no explicit locking ...

```
int CompareAndSwap(int *addr, int expected, int new) {
    if (*addr == expected) {
        *addr = new;
        return 1; // success

    return 0; // failure
}
int AtomicIncrement(int *val, int amount)
    do {
        int old = *val;
    } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

Deadlock Prevention

- Mutal Exclusion
  - Lock-free data structures!
    - Use hardware instructions to build structures requiring no explicit locking …

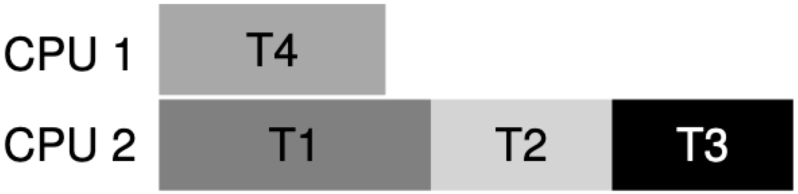
```
void insert(int val) {
    nod_t *n = malloc(sizeof(node_t);
    assert(n);
    n->value = val;
    pthread_mutex_lock(listlock); // begin
    n->next = head;
    head – n;
    pthread_mutex_unlock(listlock); // end
}
```

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

Deadlock Avaoidance via Scheduling

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

Bad



Good

Practical Strategies

- **Minimize likelihood of deadlock by applying prevention strategies wherever possible:**
  - avoid unnecessarily fine-grained locking (share a lock)
  - define locking order if not possible
  - use tools that flag when locking order is violated
  - have clear signaling strategies
- **Allow for deadlock recovery**
  - Design system to minimize the amount of work that is lost or must be repeated if deadlock recovery necessitates killing of processes

Deadlock vs. Starvation

Starvation

Apparent lack of progress that could be fixed with a proper scheduling strategy:

- Strict priority scheduler might starve lower priority thread if higher priority threads are always READY
- Reader-writer locks may assign lock to only readers, starving writers

Deadlock

There is no scheduling policy that would allow forward progress

See Levine [2] for an attempt to extend the definition of deadlock to other lack of progress states

SIGOPS OS Review, 37(1):54-64, Jan 2003

- Deadlock is a state where a set of threads is blocked waiting for a resource or event that could be produced only by a thread in the set
- For reusable resources, can be analyzed with a resource allocation graph
- Employ strategies for
  - Deadlock Detection & Recovery
  - Deadlock Prevention
- In general, risk of deadlock increases with finer granularity of locking: scalability vs robustness trade-off