Chapter 32: Common Concurrency Problems

32.2: Non-Deadlock Bugs

- Non-deadlock bugs make up a majority of concurrency bugs
- **Atomicity Violation Bugs**
  - “The desired serializability among multiple memory accesses is violated
  - A common solution is to simply put locks around the shared area or critical section
- **Order Violation Bugs**
  - Example: One thread can be initialized and another thread can change that thread’s state by dereferencing the pointer
    - This may not be viable because one thread can dereference a null pointer to cause a segmentation fault
  - The desired order between two memory accesses is flipped, meaning that A should always be executed before B, but the order is not enforced during execution
  - Using condition variables is an easy and robust way to add this style of synchronization

32.3: Deadlock Bugs

- Example of a deadlock
  - Thread 1 is holding a lock L1 and waiting for another one. Unfortunately, thread 2 that holds lock L2 is waiting for L1 to be released

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

  - The above code can cause a deadlock but not always.
    - If thread 1 grabs lock L1 and then a context switch occurs, to thread 2, and then thread 2 grabs L2 and tries to acquire L1, we can have a deadlock
- Why do Deadlocks occur?

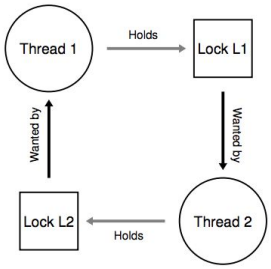
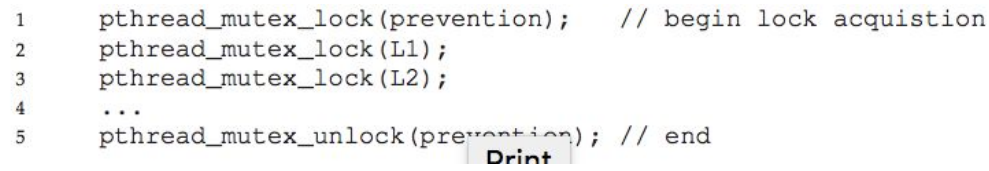


Figure 32.2: **The Deadlock Dependency Graph**

  - One reason is that in large code bases, complex dependencies arise between components.

        - OS: Virtual memory system may need to access the file system in order to page in a block from disk -> file system might subsequently require a page of memory to read the block into and thus contact the virtual memory system
    - **Encapsulation:** We are taught to use information hiding to make software easier to build and more modular -> however, easy to deadlock
- Conditions for Deadlock
    - Mutual Exclusion: Threads claim exclusive control of resources that they require
    - 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 cannot be forcibly removed from threads that are holding them
    - Circular waits: There exists a circular chain of threads such that each thread holds one or more resources that are being requested by the next thread in the chain
- Prevention
    - **Circular Wait**
        - Write lock code so that you never produce a circular wait
        - Example: If there are two locks L1 and L2, we can ensure that one lock is always acquired before another one
        - **Partial Ordering:** Linux includes ten different groups of lock acquisition orders, including simple ones such as “i_mutex” before “i_map_mutex”
        - Tip: Enforce Lock Ordering By Lock Address
            - ex: do_something(mutex_t *m1, mutex_t *m2) can cause a deadlock because one thread can call do_something and always grab m1 before m2, one thread can call do_something(L1, L2) and another can call (L2, L1)
            - You can use the lock address to always grab the higher (or lower) order address
    - Hold and Wait
        - Hold and wait requirements for deadlock can be avoided by acquiring all locks at once, atomically
        - 

        - By first grabbing lock prevention, the code can guarantee that no untimely thread switch can occur in the midst of lock acquisition and thus deadlock can be avoided
        - This requires that anytime a thread grabs a lock, it must first acquire the global prevention lock

        - ex: if another thread was trying to grab locks L1 and L2 in a different order, it would be OK because it would hold the prevention lock while doing so
    - Extra care is needed because we must know when each lock must be required at which exact time
- No Preemption
    - Because we generally view locks as held until unlock is called, multiple lock acquisition often gets us into trouble **because when waiting for one lock, we are holding another.**
    - Ex: pthread_mutex_trylock() either grabs the lock (if it is available) and returns success or returns an error code indicating that the lock is held

```
1   top:
2     pthread_mutex_lock(L1);
3     if (pthread_mutex_trylock(L2) != 0) {
4       pthread_mutex_unlock(L1);
5       goto top;
6     }
```

    -
    - Another thread can call the locks in a different order and the program would still be deadlock free
    - **Livelock:** It is possible that two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks.
        - In this case, both systems are running through this code sequence over and over again, but progress is not being made
        - As opposed to a deadlock where there is no progress being made because the threads are in a constant state of waiting
        - One solution: One could add a random delay before looping back and trying the entire thing over again, thus decreasing the odds of repeated interference among competing threads
- Mutual Exclusion
    - Avoid the need for mutual exclusion at all
    - Using powerful hardware instructions, you can build data structures in a manner that does not require explicit locking

```
1  int CompareAndSwap(int *address, int expected, int new) {
2    if (*address == expected) {
3      *address = new;
4      return 1; // success
5    }
6    return 0; // failure
7  }
```

Imagine we now wanted to atomically increment a value by a certain amount. We could do it as follows:

```
1  void AtomicIncrement(int *value, int amount) {
2    do {
3      int old = *value;
4    } while (CompareAndSwap(value, old, old + amount) == 0);
5  }
```

    -

        - instead of acquiring the lock, doing the update, then releasing, we repeatedly try to update the value to the new amount through the compare and swap method
- Deadlock Avoidance via Scheduling
    - Avoidance requires some global knowledge of which locks various threads might grab during their execution and schedules threads so that no deadlocks can occur
    - ex: T1 grabs locks L1 and L2, T2 grabs locks L1 and L2, T3 grabs just L2, and T4 grabs no locks at all
    - This means that a smart scheduler can say that as long as T1 and T2 are not run at the same time so no deadlock occurs

    -
    - If T3 grabs both L1 and L2, a scheduler can make sure that those two threads are not run at the same time -> costs performance
- Detect and Recover
    - Allow deadlocks to occasionally occur, and then take some action once the deadlock has been detected
    - If deadlocks are rare, we can simply just reboot
    - Deadlock detector that runs periodically, building a resource graph and checking for its cycles (deadlocks)

Deadlock Avoidance

- We commonly have situations where
    - mutual exclusion is fundamental
    - hold and block are inevitable
    - preemption is unacceptable
    - the resource dependency networks are imponderable
- Deadlock situation
    - main memory is exhausted
    - we need to swap some processes out to secondary storage to free up memory
    - swapping processes out involves the creation of new I/O requests descriptors, which must be allocated from main memory
    - Problem is that we exhausted a critical resource
        - similar situations
            - process will free up resources when it completes
            - but the process needs more resources in order to complete

Reservations

- Declining to grant requests that would put the system into an unsafely resource depleted state is enough to prevent deadlock.
    - Difficult to gracefully handle the failure of a random allocation request mid-operation (critically low resource)
    - Therefore, it is common to ask processes to reserve their resources before they actually need them
- Ex: sbrk(2) system call
    - does not actually allocate more memory to the process
    - requests the OS to change the size of the data segment in the process' virtual address space
    - The actual memory assignments will not happen until the process begins referencing those newly authorized pages
- We can refuse to create new files when file system space gets low
- we could refuse to create new processes if we found ourselves thrashing due to pressure on main memory
- we could refuse to create or bind sockets when network traffic saturates our service level agreement

Over-Booking

- It is often considered relatively safe to grant somewhat more reservations than we actually have the resources to fulfill
- The reward for overbooking is that we can get more work done with the same resources
- Danger is that there might be a demand that we cannot gracefully handle
    - Airlines occasionally offer cash and free trips to anyone who is willing to take a later flight
    - Required network bandwidth is routinely estimated based on expected traffic distribution
    - In OS, the notion of killing random processes is so abhorrent that most OS simple refuse to over book -> common to underbook (e.g. reserve the last 10% for emergency/super use)

Dealing with Rejection

- What should a process do when some resource allocation request fails?
    - Simple program: log an error message and exit
    - Stubborn program: might continue to retry the request
    - Robust program: return errors for requests that cannot be processed but continue to try to serve new requests
    - Civic-minded program: might attempt to reduce its resource use (and therefore the number of requests it can serve)

Java Synchronization

- Synchronized methods and synchronized statements

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

- It is not possible for two invocations of synchronized methods on the same object to interleave
- When one thread is executing a synchronized method for an object, all other threads that invoke the synchronized method for the same object must wait until the first thread is done
- Changes to the state of the object are visible to all threads (broadcasted)

Java Intrinsic Locks and Synchronization

- Every object has an intrinsic lock associated with it
- By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them

Java Locks in Synchronized Statements

- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

- addName needs to synchronize changes to lastName and nameCount, but also needs to avoid synchronizing invocations of other objects' methods
- synchronized statements are useful for improving concurrency with fine-grained synchronization

- if we have two variables that are never used together, we can simply call a synchronized statement on each of them instead of making a new synchronized method

Reentrant Synchronization

- A thread can acquire a lock that it already owns
- Allowing a thread to acquire the same lock more than once enables reentrant synchronization
- Synchronized code invokes a method that also contains synchronized code, and both sets of code use the same lock
- Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block

Monitors

- A synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to come true
- Also have a mechanism for signaling other threads that the condition has been met
- Contains a mutex and a condition variable (condition variable is basically a container of threads that are waiting for a certain condition)
- Another definition: a thread-safe class, object, or module that wraps around a mutex in order to safely allow access to a method or variable by more than one thread
    - methods are executed with mutual exclusion
    - By using condition variables, it provides the ability for threads to wait on a certain condition, hence “monitor”
    - Can be referred to as a thread-safe object/class/module
    - The monitor queue prevents starvation so it is intrinsically fair
    - complete mutual exclusion is assured

Health Monitoring and Recovery

- How would we identify if the system was deadlocked?
    - Identify all of the blocked processes
    - identify the resource on which each process is blocked
    - identify the owner of each blocking resource
    - determine whether or not the implied dependency graph contains any loops

Health Monitoring

- How do we know whether or not the system is making progress? (definition of deadlock)
    - Have an internal monitoring agent watch message traffic or a transaction log to determine whether or not work is continuing
    - By asking clients to submit failure reports to a central monitoring service when a server appears to have become
    - By having each server send periodic heart-beat messages to a central health monitoring service

    - By having an external health monitoring service send periodic test requests to the service that is being monitored, and ascertain that they are being responded to correctly and in a timely fashion
    - Heartbeat messages: Can only tell us that the node and application are still up and running -> cannot tell if the application is actually serving requests
    - An external health monitoring service can determine whether or not the monitored application is responding to requests. But this does not tell us whether other requests have not been deadlocked/wedged
    - An internal monitoring agent might be able to monitor logs or statistics to determine that the service is processing requests at a reasonable rate. However, if the internal monitor fails, it may not be able to detect and report errors
    - Many systems use a combination of this
        - first line of defense is an internal monitoring agent that closely watches key applications to detect failures and hangs
        - If the internal monitoring agent is responsible for sending heartbeats (health status reports) to a central monitoring agent, a failure of the internal monitoring system could be detected
        - An external test service that periodically generates test transactions. Provides an independent assessment that might include external factors that would not be tested by internal and central monitoring services

Managed Recovery

- HIghly available services must be designed for restart, recovery, and failover in the case where a system has hung or failed
    - software should be designed so that any process in the system can be killed/restarted at any time. When a process restarts, it should be able to resume communication with other processes with minimal disruption
    - Multiple levels of restart
        - warm-start: restore the last saved state and resume service
        - cold-start: ignore any saved state and restart new operations from scratch (state may be corrupted)
        - reset and reboot: reboot the entire system and cold start the applications
    - Progressively escalating restarts
        - restart only a single process, and expect it to resync with the other processes when it comes back up
        - maintain a list of all of the processes involved in the delivery of a service, and restart all processes in that group
        - restart all of the software on a single node
        - restart a group on nodes, or the entire system

False Reports

- Say that a central monitoring service notes that it has not received a heartbeat from process A

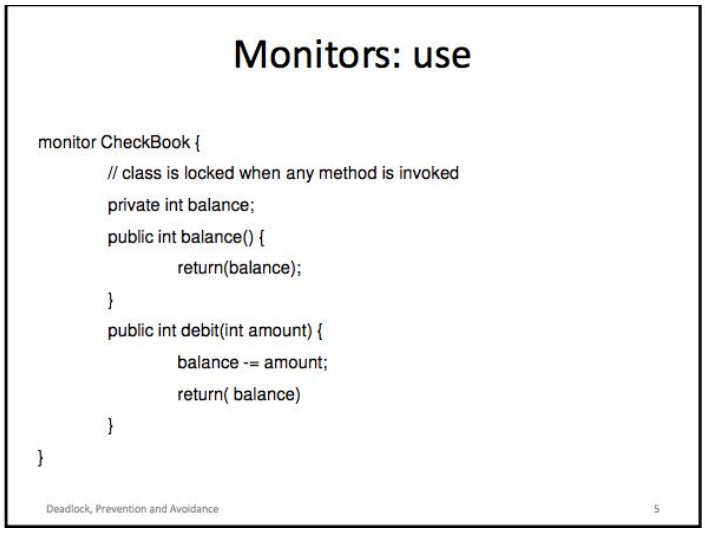
    - it might mean that the process A's node has failed
    - It might mean that the process A has failed
    - it might mean that the process A's system is loaded and the heartbeat message was delayed
    - it might mean that a network error prevented or delayed the delivery of a heartbeat message
    - it might mean there was a problem with the central monitoring system
- We don't want to start an expensive fire-drill unless we are pretty sure that a process has actually failed
    - the best option would be for a failing system to detect its own problem, inform its partners, and shut down clearly
    - if the failure is detected by a missing heartbeat, it may be wise to wait for multiple heart beats
    - to distinguish a problem with a monitored system from a problem in the monitoring infrastructure, we might want to wait for multiple processes/nodes to notice and report the problem
- Trade-off
    - May suffer unnecessary service disruptions from forcing unnecessary failovers of health servers
    - may prolong service outage by waiting too long
    - we can mis-diagnose the cause of the problem and completely restart the wrong components

Other Managed Restarts

- Non-disruptive rolling upgrades
    - if a system is capable of operating without some of its nodes, it is possible to achieve non-disruptive rolling software upgrades
    - We can take nodes down, one-at-a-time, upgrade each to a new software release, and then integrate them into the service
        - new software must be upwards compatible with the old software
        - if the new upgrade does not seem to be working, there must be a fallback feature to go back to the previous working release
- Prophylactic reboots
    - Automatically restart every system at a regular interval

Monitors: Protected Classes

- Each monitor class has a semaphore
    - automatically acquired on method invocation
    - automatically released on method return
    - automatically released/acquired around CV waits
- Good encapsulation
    - developers need not identify critical sections
    - clients need not be concerned with locking

    - protection is completely automatic
- high confidence of adequate protection
- 

Evaluating Monitors

- Correctness
    - complete mutual exclusion is assured
- Fairness
    - semaphore queue prevents starvation
- Progress
    - Inter-class dependencies can cause deadlocks (bad)
- Performance
    - coarse grained locking is not scalable (bad)

Evaluating Java Synchronized Methods

- Correctness
    - correct if developer implements them in the right way
- Fairness
    - priority thread scheduling that could lead to potential starvation of resources
- Progress
    - Prevents single threaded application deadlocks -> multithreaded applications can still deadlock
- Performance
    - fine grained (per object) locking that allows for good performance, however we must be able to identify the critical sections

Encapsulated Locking

- opaquely encapsulate implementation details
    - make class easier to use for clients
    - preserves the freedom to change it later

- locking is entirely internal to class
- critical sections involve only class resources
- critical sections do not span multiple operations
- no possible interactions with external resources
- Encapsulating within the class does not always solve everything

Client Locking

- Class cannot correctly synchronize all users
- critical section spans multiple class operations
    - updates in a higher level transaction
- client-dependent synchronization needs
    - locking needs depend on how object is used
    - client may control access to protected objects
    - client may select best serialization method
- potential interactions with other resources
    - deadlock prevention must be at a higher level
- Not preferred but sometimes necessary

Spin Lock vs Atomic Update Loops (Test and Set vs Compare and Swap)

- both involve spinning on an atomic update
    - but they are not the same
- a spin lock
    - spins until the lock is released
    - this could essentially take a very long time to complete
- An atomic update loop
    - spins until there is no conflict during the update
    - impossible to be preempted while holding the lock
    - conflicting updates are actually very rare

Evaluating Lock Free Operations

- Effectiveness/Correctness
    - effective against all conflicting updates
    - cannot be used for complex critical section (bad)
- Progress
    - no possibility of deadlock or convoy
- Fairness
    - small possibility of brief spins -> not guaranteed bounded time but very close
- Performance
    - expensive instructions but cheaper than syscalls

What is a Deadlock?

- Two or more processes
    - cannot complete without all required resources

    - each holds a resource the other needs
- No progress is possible
    - each is blocked, waiting for another to complete
- Related Problems: livelock
    - processes not blocked, but cannot complete
- Related problem: priority inversion
    - high priority actor blocked by low priority actor

Why Study Deadlocks?

- A major peril in cooperating parallel processes
    - they are relatively common in complex applications
    - they result in catastrophic system failures
- Finding them through debugging is very difficult
    - they happen intermittently and are hard to diagnose
    - they are much easier to prevent at design time
- Once you understand them, you can avoid them
    - most deadlocks result from careless/ignorant designs
    - an ounce of prevention is worth a pound of core

Commodity Resource Problems

- Memory deadlock
    - we are out of memory
    - we need to swap some processes out
    - we need memory to build the I/O request to swap the processes out
- Critical resource exhaustion
    - a process has faulted for a new page
    - there are no free pages in memory
    - there are no free pages on the swap device

Avoidance - Reservations

- advance reservations for commodities
    - resource manager tracks outstanding reservations
    - only grants reservations if resources are available
- over subscriptions are detected early
    - before processes ever get the resources
- client must be prepared to deal with failures
    - but these do not result in deadlocks
- dilemma over-booking vs under utilization
- malloc() is an example of reservation
    - it reserves the memory until the process gets the resources, preventing oversubscriptions from happening
    - you can refuse to allocate the resource when you don’t think you can handle it

Real Commodity Resource Management

- Advanced Reservation mechanisms are common
    - unix setbreak system call to allocate more memory
    - disk quotas, quality of service contracts
- once granted, reservations are guaranteed
    - allocation failures only happen at reservation time … hopefully before the new computation has begun
    - failures will not happen at request time
    - system behavior more predictable, easier to handle
    - if reservations cannot be guaranteed, they will not be allocated
- but clients must deal with reservation failure

Dealing with Rejection

- Need to gracefully handle errors
    - we want to minimize the impact that this has on the least amount of people
- reservations eliminate difficult failures
    - recovering from a failure in mid-computation
    - may involve awkward and complex unwinding
- graceful handling of reservation failures
    - fail new request, but continue running
    - try to reserve essential resources at start-up time
- keep trying until it works … not so good
    - may impose unbounded delay on requestor
    - freeing resources or shedding load could help
- We want to have responsible behavior instead of trying over and over
    - ex: if there are a bunch of old channels that we haven't used in a while
        - we can close those channels and then start again in order to handle the issue responsibly

Overbooking vs Under Utilization

- problem reservations overestimate requirements
    - clients seldom need all resources all the time
    - all clients won't need max allocation at the same time
- question: can one safely overbook resources?
    - for example, seats on an airplane
- what is a safe resource allocation?
    - one where everyone will be able to complete
    - some people may have to wait for others to complete
    - we must be sure there are no deadlocks
- As long as nothing bad happens if we overbook, we can overbook and still be fine
    - if we know that we can overbook a resource, it should be okay

*You must lock the list before you lock the buffer

Deadlock - Practical Examples