MODULE 12 & 14 :

Process Synchronization and Deadlocks

Process Synchronization and Deadlocks – Slide Plan

- Why do you need synchronization? The Critical-Section Problem – Slides 3-9
- How do you synchronize? Synchronization Hardware, Locks, Mutexes and Semaphores – Slides 10-17
- Synchronization Examples – Slides 18-21
- Barrier synchronization – Slides 22-25
- Synchronization problems – Slides 26-27
- Deadlocks, dealing with deadlocks – Slides 28-35
- Solutions to Synchronization Problems (including deadlocks) – Slides 35-50

Why do you need Synchronization?

- Critical sections – register coherence, variable (queue) coherence, atomicity preservation, interrupt register coherence
- Solution – semas, variety of locks, interrupt masking
- Application semantic (phase co-ordination) coherence – matrix multiplication, barrier wait
- Sync going awry – deadlocks, sema priority inversion, basic application-level logical solutions, OS-level solutions – sema priority inversion (avoidance), watchdog (recovery)

Why do you need Synchronization?

- Fundamentally, the need for synchronization of multiple threads/processes (we will use ‘threads’ here on because that is simpler) is because serial semantics are expected even in multithreaded scenarios, i.e. we expect the same results from concurrent execution that we would have got in completely serialized execution scenarios

- Basically, what we are expecting is serial execution results in a shorter time

Why do you need Synchronization?

- Let us look at a basic concurrent/multithread execution scenario:

Expected results after a run each of T1 and T2, assuming Counter is initially 0: Counter =2

Why do you need Synchronization? (contd.)

- If you look at this closer, you will notice that a load of the variable ‘counter’ from memory (value 0 in this case) will need to be done into a register before an increment is done.

- Assuming that T1 got swapped out at this point by say, a higher priority T2, the register would be backed up on T1’s stack and restored at some point when T1 get swapped back in.

- Meanwhile in this T1 swap-out time window, assume T2 that just got swapped in, also went in and loaded ‘0’ (which was what the value of the variable would still be because T1’s incrementing has not been commited yet to memory). The ‘0’ is incremented to 1 and commited to memory.

Why do you need Synchronization? (contd.)

- When T1 is eventually scheduled back in, it will also increment 0 (in its post-context switch restored register) to 1 and push it back to memory.
- Thus, we have missed a write to ‘counter’ and haven’t got the serial semantics that we required.
- This is the case because, the threads didn’t get mutually exclusive increment access to ‘counter’, the register providing a space for the lack of mutual exclusion to exploit.
- In other words, the register contents across the context switch of T1 have effectively become stale, you could call it a lack of register coherence. A seemingly indivisible, atomic increment at assembly instruction level, proving not to be so.

Why do you need Synchronization? (contd.)

- Let’s look at another loss of atomicity, this time at the high-level language level, in an enqueuing/dequeueing scenario.
- Enqueueing: Prev = Tail; Tail = new (Node); Tail → next = Prev
- Need for mutual exclusion is because if a second process did the same as above for a different node, depending on which enqueueing (Tail = new (Node)) happened last, that would overwrite the first, if Prev=Tail and Tail = new (Node) happened in an overlapped manner.
- For dequeue: Head = Head.next involves a load, a change and a store, which needs to be mutually exclusive, else you might miss a dequeue.
- This is a lack of atomicity as three high-level language steps have lost their sequential coherence.

Why do you need Synchronization? (contd.)

- Let’s look at a case of nested interrupts.
- There is a stage in early interrupt/exception processing where some special purpose registers that support interrupts and interrupt returns need to be saved on the interrupt/exception stack. If interrupts are not masked in this time window, a nested interrupt can cause return from interrupts to be incoherent.
- Therefore, interrupts are masked by processor architecture in this window, and need to be re-enabled by interrupt handler code to allow for nested interrupts after this window.

Why do you need Synchronization? (contd.)

- Let’s look at a case of barrier synchronization:

- When there’s a scenario that needs multiple threads to wait after a phase of parallel operation, before continuing on to the next phase of parallel execution, we use what’s termed barrier synchronization

- Examples:
  - Matrix Merge sort: Sort smallest arrays, wait (barrier sync), merge sorted small arrays
  - Multiplication: Divide the matrices into tiles, do multiplication within the tiles, wait (barrier sync), do the additions across tiles, wait (barrier sync), print result

How do you Synchronize?

- There are a few processor instructions that implement atomicity at the hardware level (at the lowest register-bus level of coherence, atomicity is after all a hardware issue).

- Operating system-level solutions like a variety of locks, semaphores, mutexes, condition variables are built on top of these processor-level primitives, to safeguard coherence and help obtain expected serial results.

- And, in some extreme cases like the interrupt handling case, masking of interrupts is done to achieve coherence. And, in the multiprocessor context, masking of interrupts is replaced by spinlocks.

Locks, Mutexes and Semaphores

Locks, Mutexes and Semaphores are based on an atomic processor instruction or an in-effect atomic pair of instructions that have the following semantics:

- Test-and-set-lock – Atomically, test a memory location for a value (say, 1) and set it to 1. The instruction “returns” True by setting a flag/register, if the old value before the set, was 0. If it was already 1, then it would return False.

  - ✓ On x86 processors, similar functionality is provided by the cmpxchg with lock prefix instruction pair (this is more compare-and-swap, but broad idea remains the same even if with a slightly different implementation).

  - ✓ On RISC processors like the PowerPC, it is the ll(load linked), sc (store conditional) pair that provide the same functionality

Basic building blocks of Mutual Exclusion

With cmpxchg:

Load eax, address /*old value*/

Mov eax, ebx

Increment ebx

Lock, Cmpxchg (ebx, address)

/*if address still contains old eax,

move current ebx there*/

If cmpxchg returns success, go forward, else loop back to load eax point & try again

Basic building blocks of Mutual Exclusion (Contd.)

With ll-sc:

P1: Loop: Load-linked (reg, addr);
reg=reg+1;
If Store-conditional (addr, reg) = False, Loop.
Else (Exit loop)

P2: Loop: Load-linked (reg, addr);
reg=reg+1;
If Store-conditional (addr, reg) = False, Loop.
Else (Exit loop)

Locks

There are a variety of locks that build on the basic atomic primitives described in the previous slides, here's a basic lock of the busy-waiting variety using ll-sc:

```
void lock(atomic32_p lock_p
        {
                int32 old_val;
                repeat {
                        old_val = lwarx(lock_p);
                } until(old_val == 0 && stwcx (lock_p, 1));
                isync();
                return;
        }
```

Locks (contd.)

```
void unlock(atomic32_p lock_p)
    {
        int32 old_val;
        sync();
        lock_p->value = 0;
    return;
    }
```

Note: The isync, sync is in order to ensure sequential consistency in case of processors that have more relaxed consistency models

Locks, Mutexes and Semaphores(contd.)

- The implementation of the basic lock on the previous slides involved a busy-waiting loop to get the lock.
- Although useful as proof-of-concept, that’s too time consuming to be of interest in practical OS-level implementations.
- Therefore, in practical implementations, instead of looping/busy-waiting in attempts to acquire locks, mutexes, or semaphores, you block the task and as part of the releases, you unblock a task that is waiting(for instance, at the head of the wait queue) on the lock, mutex or semaphore.

Locks, Mutexes and Semaphores (contd.)

- The distinctions between locks, mutexes and semaphores are at the OS-level – at the very lowest level, they all use the basic mutual exclusion implementing processor instructions.

- One crucial difference between mutexes and semaphores at the OS-level is that a mutex can only be released by its owner, whereas a semaphore can be released by threads that don't own it. Hence, semaphores can be used purely as a signalling mechanism say, from kernel level to a user-level thread.

Posix Mutexes - example

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void *arg)
{
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Job %d has finished\n", counter);
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

Posix Mutexes – example (contd.)

```
int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while(i < 2)
    {
        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]", strerror(error));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

Expected output:

Job 1 has started
Job 2 has started
Job 2 has finished
Job 2 has finished

Q: What if mutexes weren't used?

Posix Semaphores - example

```
// C program to demonstrate working of Semaphores
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}


int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

Posix Semaphores – example (contd.)

```
int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

Use gcc sema.c -lpthread –lrt to compile

Expected Output:
Entered..

Just Exiting...

Entered..

Just Exiting...

Questions: What is the expected output? What if there was no semaphore used?

Barrier Synchronization

A Simple Centralized Barrier

- **Shared counter maintains number of processes that have arrived**
  - **increment when arrive (lock), check until reaches numprocs**
  - **Problem?**

```
struct bar_type {int counter; struct lock_type lock;
                              int flag = 0;} bar_name;
BARRIER (bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)
        bar_name.flag = 0;              /* reset flag if first to reach*/
    mycount = bar_name.counter++;       /* mycount is private */
    UNLOCK(bar_name.lock);
    if (mycount == p) {                 /* last to arrive */
        bar_name.counter = 0;           /* reset for next barrier */
        bar_name.flag = 1;              /* release waiters */
    }
    else while (bar_name.flag == 0) {};     /* busy wait for release */
}
```

Barrier Synchronization (contd.)

- Increment of counter needs to be atomic
- Increment of counter and check needs to be atomic – why?
- What is the problem in the above? Set of the flag (which is the exit criterion for the busy wait) is done just before the exit, but since the reset of the flag is done on entry into the barrier a second time (a re-entry for the same process), if processes other than the last get swapped out in the middle of the exit criterion check loop and come back, there is a chance that they will get to see the newly reset value of flag. Thus, ensuring they never exit the barrier, resulting in a deadlock.

Barrier Synchronization (contd.)

- This can be fixed using a local sense variable that demarcates the first iteration of the barrier from the second, by introducing a local sense of which iteration you are in. The global view of this local sense is consistent except for the time where the new (second) entrant to the barrier resets it and the others are still in the first barrier. Once everyone enters the barrier a second time, there is consistency once more.

- Any option other than thread-local sense variable? Yes, a separate counter, but that makes things clumsy, as you need a semaphore to protect this, etc.

Barrier with Sense Reversal

```
bar_name.flag = 1;
thread_local local_sense  = 1;
BARRIER (bar_name, p) {

    local_sense = !(local_sense); /* toggle private sense variable */
    LOCK(bar_name.lock);
    bar_name.counter++;
    if (bar_name.counter == p)
        {UNLOCK(bar_name.lock);
        bar_name.counter = 0;
        bar_name.flag = local_sense; }/* release waiters*/
    else
        { UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) {}; }
    }
```

Problems when Synchronization is done in a wrong way

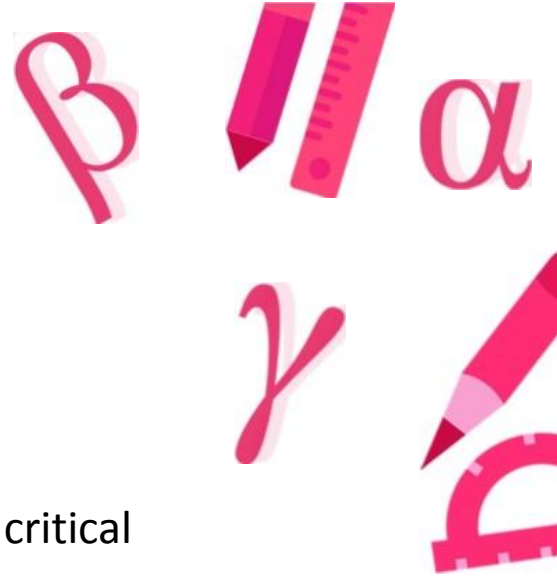
The most common problems faced when using lock-based synchronization methods are:

- Starvation or a skewed rate of progress among the tasks that are being synchronized due to a lack of guaranteed fairness in the lock acquire primitive
- Deadlocks due to either wrong order followed in acquire and release of locks in case of nested locks or issues like priority inversion
- Issues like priority inversion and convoying

Problems when Synchronization is done in a wrong way

- If applications crash within locked regions, you might be left with data structures that are inconsistent

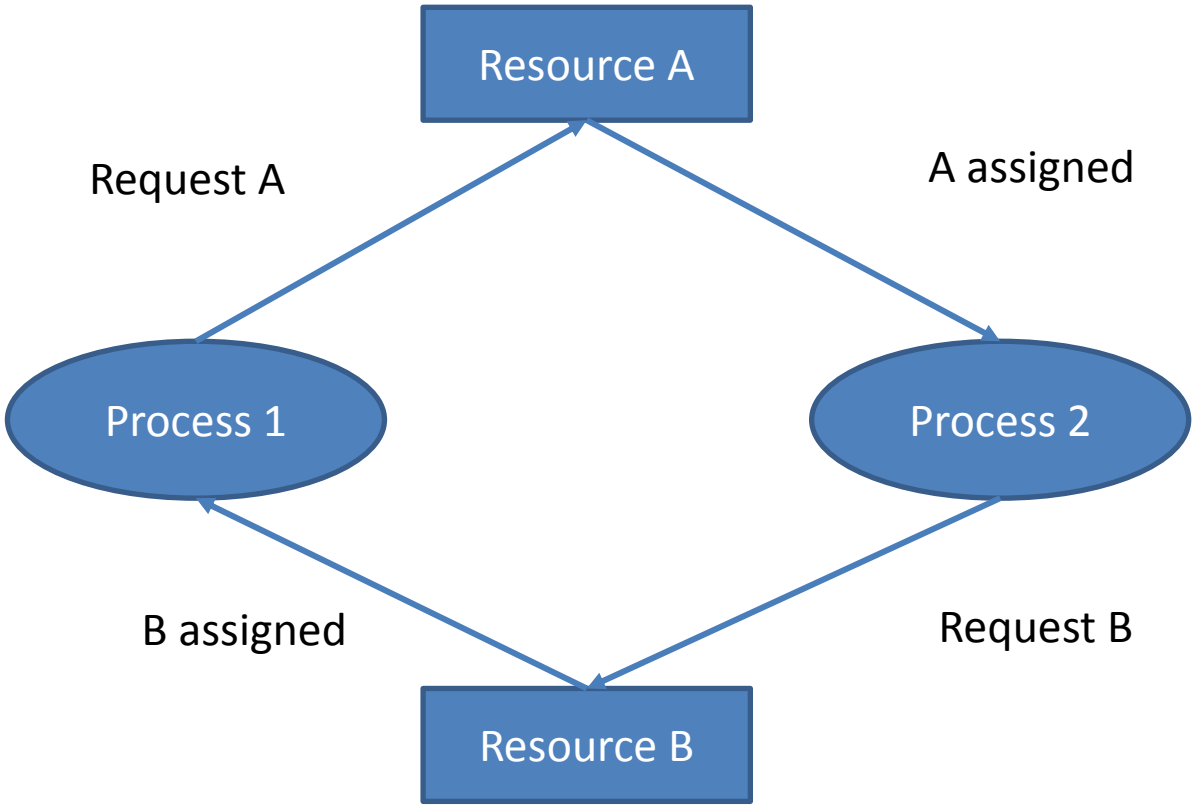
- Lack of proper classification of synchronization scenarios – there might be cases where for instance, you need to identify a scenario as a frequent reader, infrequent writer case and are doing over-serialization of readers.



Deadlocks

- Deadlocks occur in lock-unlock scenarios that are intended to protect critical sections where shared resources are accessed concurrently
- Deadlocks happen due to different orders followed (by different users of the lock-unlock pairs) in acquire and release of locks in case of nested locks
- There are issues like priority inversion and process crashes while holding onto locks that result in starvation of tasks for potentially long periods. Although the effects are somewhat similar to that of a deadlock scenario, these are technically different from deadlocks – deadlocks result only when there are circular waits for critical resources

Deadlocks (contd.)



Resource A has been assigned to Process 2, and Resource B has been assigned to Process 1. Process 1 now requests for Resource A, and Process 2 now requests for Resource B, resulting in a circular dependency or a deadlock

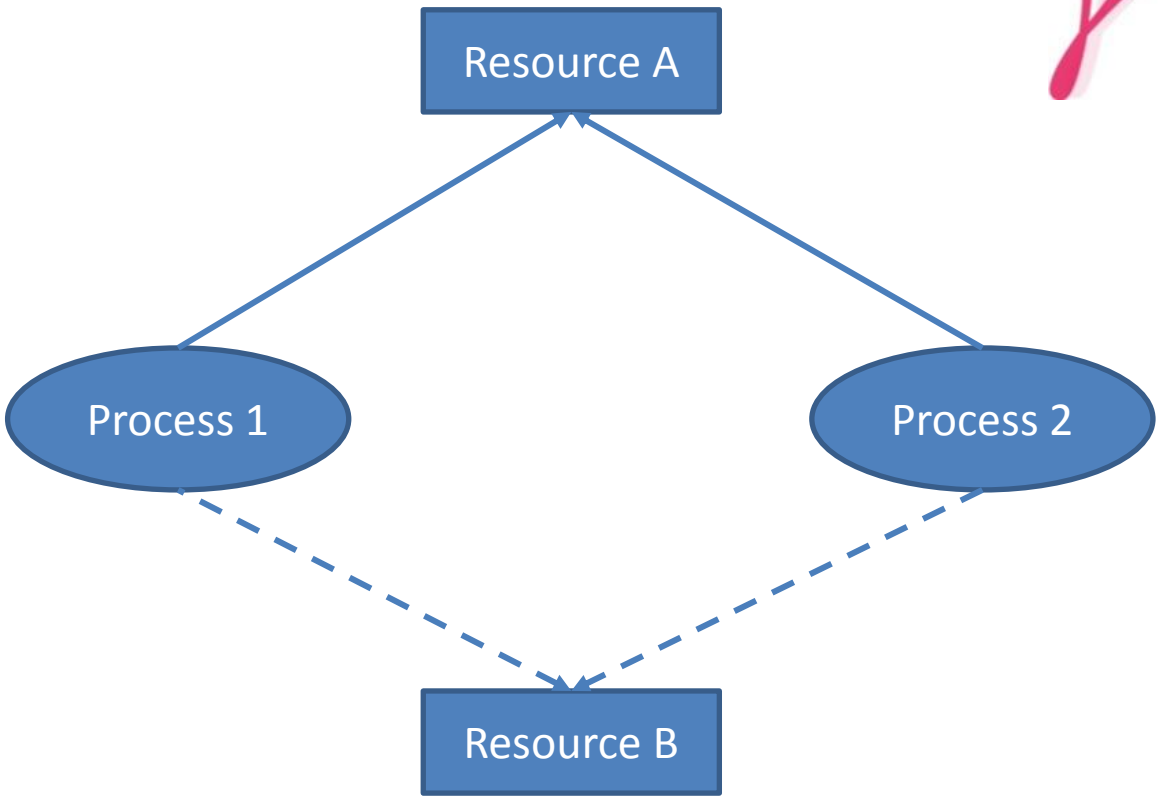
Dealing with deadlocks

Solutions range from:

- Deadlock avoidance by using wait or lock free data structures (dealt with in detail in the section on how to avoid poor synchronization problems)
- Deadlock avoidance by detection at resource acquire/release time - Resource graph checking solutions, Banker’s algorithm type solutions
- Tolerance and recovery using OS mechanisms like audit tasks

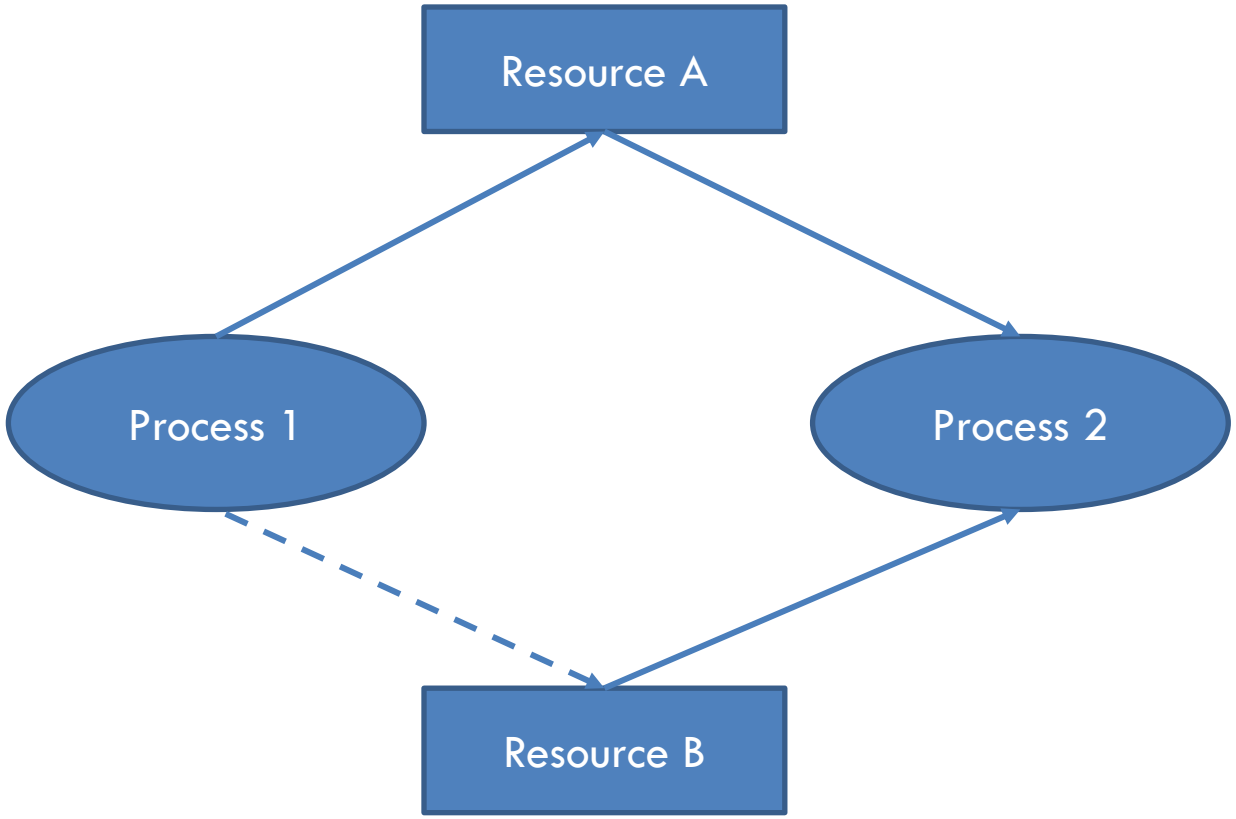
Dealing with deadlocks – Resource allocation

- One instance of each resource
- Request edges, Assignment edges
- Claim edges
- On request, claim edge becomes a request edge
- No claims that lead to cycles are serviced



Resource allocation graph for deadlock avoidance

Dealing with deadlocks – Resource allocation



An unsafe state in a resource allocation graph

Dealing with deadlocks – Banker's algorithm

- For resources that have more than one instances, the resource allocation graph isn't enough.

- Safety algorithm + resource allocation algorithm

- Safety algorithm
  - Check of allocation, need and request limits to determine safety of state

- Resource allocation algorithm:
  If Request causes Request to exceed Need – decline with error
  If Request would have caused Available to go negative, then Wait
  Resource granted if the following steps are safe (if not, then Wait):
  Available = Available - Request
  Allocation = Allocation + Request
  Need = Need – Request

Dealing with deadlocks – Banker's algorithm

- Not very practical – as it is well nigh impossible to know demand for resources at process creation time (application code and requirements grow over time with code maintenance), number of processes grow as well in a real-world system.

- Which is why, these types of algorithms are very rarely used in OS kernel code. In Linux kernel code, strict ordering of acquiring and releasing locks is followed to avoid deadlocks, or techniques like RCU (Read-Copy-Update – Slide 43) are relied on

Dealing with deadlocks – Tolerance and recovery

- OS-level audit tasks can be designed to check progress of select tasks/all tasks by periodically checking their stacks for function/subroutine execution progress.

- Suspect tasks can be logged and after a certain threshold, terminated.

Solutions to problems of poor Synchronization

- Solutions range from problem avoidance by using wait or lock free data structures to solve synchronization problems or problem detection and recovery by OS mechanisms like watchdog tasks

- Using multiple reader-single writer type locks like read-copy-update (RCU) primitives that are used in the Linux kernel

Lock-free Queues

```
public class Node {
public T value;
public AtomicReference<Node> next;
public Node(T value) {
value = value;
next = new AtomicReference<Node>(null);
public void enq(T value) {
Node node = new Node(value);
while (true) {
Node last = tail.get();
Node next = last.next.get();
if (last == tail.get()) {
if (next == null) {
if (last.next.compareAndSet(next, node)) {
tail.compareAndSet(last, node);
return;
}
 } else { tail.compareAndSet(last, next); } } }
```

Lock-free Queues (contd.)

- First operation = Try pointing old tail → next to new node, if success, try pointing tail to new node.
- Return even if failure on second point (tail to new node) attempt as the job will be completed by a helper.
- The helper is a failed first attempter, who needs to (if needed), try and complete the half enqueue of the successful enqueuer(if we fail, it means someone else has succeeded, either way, the half enqueue needs to be done), to proceed with its own enqueue.

Lock-free Queues (contd.)

```
public T deq() throws EmptyException {
while (true) {
Node first = head.get();
Node last = tail.get();
Node next = first.next.get();
if (first == head.get()) {
if (first == last) {
if (next == null) {
throw new EmptyException();
}
tail.compareAndSet(last, next);
} else {
T value = next.value;
if (head.compareAndSet(first, next))
return value;} } } }
```

Lock-free Queues (contd.)

- The first C&S is to deal with the head = tail situation, and a concurrent enqueuer(if last=first and last.next != NULL).

- The C&S attempts to finish the job (if we fail, it means someone else has done the job), and then does the actual head pointing using the second C&S that deals with concurrent dequeues

Concurrent deq, enq

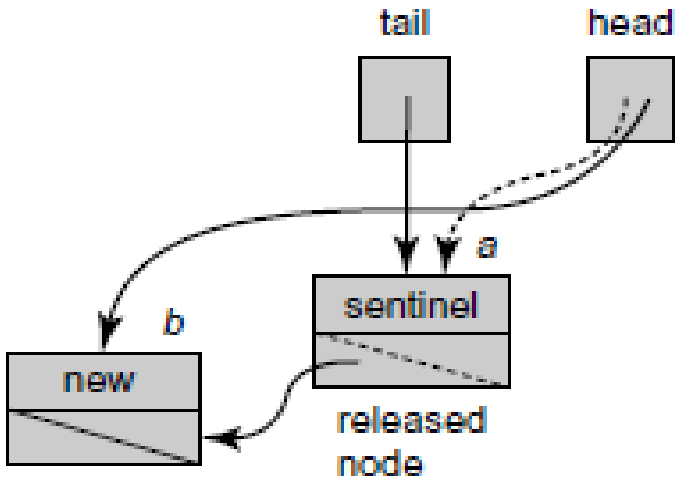


Figure 10.13 Why dequeuers must help advance `tail` in Line 35 of Fig. 10.11. Consider the scenario in which a thread enqueuing node *b* has redirected *a*'s `next` field to *b*, but has yet to redirect `tail` from *a* to *b*. If another thread starts dequeuing, it will read *b*'s value and redirect `head` from *a* to *b*, effectively removing *a* while `tail` still refers to it. To avoid this problem, the dequeuing thread must help advance `tail` from *a* to *b* before redirecting `head`.

ABA Problem

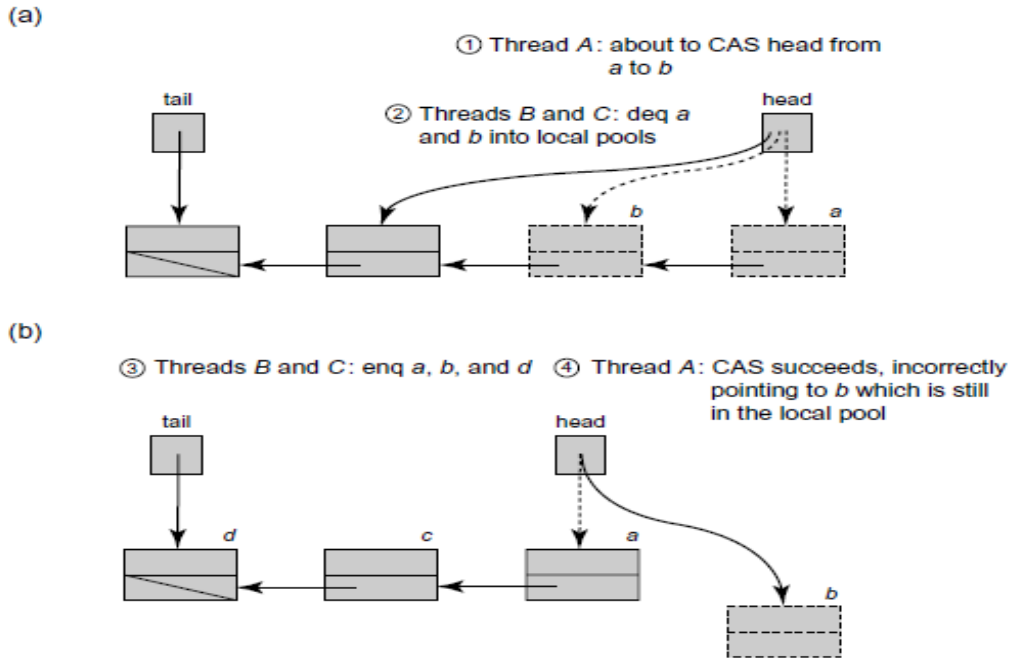


Figure 10.14 An ABA scenario: Assume that we use local pools of recycled nodes in our lock-free queue algorithm. In Part (a), the dequeuer thread *A* of Fig. 10.11 observes that the sentinel node is *a*, and next node is *b*. (Step 1) It then prepares to update `head` by applying a `compareAndSet()` with old value *a* and new value *b*. (Step 2) Suppose however, that before it takes another step, other threads dequeue *b*, then its successor, placing both *a* and *b* in the free pool. In Part (b) (Step 3) node *a* is reused, and eventually reappears as the sentinel node in the queue. (Step 4) thread *A* now wakes up, calls `compareAndSet()`, and succeeds in setting `head` to *b*, since the old value of `head` is indeed *a*. Now, `head` is incorrectly set to a recycled node.

Erratum: enq a, b c, and d

No problems with garbage collecting languages as they won't free b as long as there is a reference to it. Can be solved using a counter.

RCU (Read Copy Update)

- Lock-free programs are tough to write for more complicated data structures and critical sections – they are very good in certain situations as we have seen, but not in others
- Serialization is a given in a frequent-writer, everyone-is-a-writer situation
- But, what about the infrequent writer, frequent reader condition?
- What can be said about readers?
  - Can they return stale, but consistent values?
  - Do they need to be consistent with respect to each other in terms of time (i.e. when they do the read)? Need we demarcate between readers of a certain epoch and a different one?

RCU (contd.)

- Read-write locks serialise writes (obviously), lock for readers only to keep a limit on the number of readers (like a counting semaphore – no blocking), or if lock is already held by writer. Writes wait if there are already readers, even if some of these are newer than the writer itself. Potential write starvation. You can bias things a bit more toward the writer by asking all new readers (newer than the writer's attempt to write) to wait till the writer in waiting gets the lock and finishes its write.

- Seqlocks serializes writes, but allows writes to go ahead without waiting for readers at all. Readers check for counters before and after the reads – if the counter value increments, they retry. Potential read starvation. Seqlocks can't be used for pointers because any writer could invalidate a pointer that a reader has already followed. To fix this you need, conceptually, a separate pointer copy + atomic pointer update like in RCU

- RCU allows readers to go through always. It is wait-free for readers. Readers aren't allowed to context-switch or be interrupted inside classic read RCU routines however.

RCU(contd.)

- RCU does not demarcate strictly between readers of different epochs as far as writing goes – whoever reads after an RCU pointer update (commit) reads the new value. Writes that do a synchronize RCU (on classic non-preempt RCU) make sure all readers have been context-switched on CPU's they were running on (they wouldn't be allowed context switches if inside RCU read critical sections – a context switch thus means they are done with their read-side critical sections) – in a toy implementation, this could be done by moving the caller of synchronize RCU across all CPU's. After this "grace period", RCU free is called to get rid of the old copy.

- What about with RCU preempt?

- Hence, RCU. RCU better for infrequent writer cases because writes serialize and are hence expensive

RCU(contd.)

- Hence, when lock-free structures are too tough to implement and especially when data is rarely written (below examples):

  - ✓ Use run-to-completion method for readers, allow readers to go ahead, but don’t allow them to sleep or be pre-empted

  - ✓ Writes need to be careful (i.e. Mutually exclusive write commits need to be atomic) when committing and

  - ✓ Freeing waits for readers to finish (rcu_synchronize)

RCU (contd.)

- Uses (in Linux): Networking stack: ARP cache update, routing table update, statistics-collecting structures - stale copy tolerance
- Toy implementation of RCU + W/R lock – RCU comparison
- Careful write-side routines – freeing in the case of delete element is protected by RCU sync. What about other cases like multiple connected updates? Copy and pointer assignment within a lock, unlock or CAS
- http://www.rdrop.com/users/paulmck/RCU/whatisRCU.html
- https://lwn.net/Articles/263130/
- http://www.rdrop.com/~paulmck/RCU/RCUdissertation.2004.07.14e1.pdf

RCU (contd.)

- Use rcu_read_lock() and rcu_read_unlock() to guard RCU read-side critical sections.
- Within an RCU read-side critical section, use rcu_dereference() to dereference RCU-protected pointers – it has barriers and error filtering
- Use some solid scheme (such as locks or semaphores or CAS) to keep concurrent updates from interfering with each other.

RCU (contd.)

- Use rcu_assign_pointer() to update an RCU-protected pointer. This primitive protects concurrent readers from the updater, not concurrent updates from each other! You therefore still need to use locking (or something similar) to keep concurrent rcu_assign_pointer() primitives from interfering with each other.

- Use synchronize_rcu() after removing a data element from an RCU-protected data structure, but before reclaiming/freeing the data element, in order to wait for the completion of all RCU read-side critical sections that might be referencing that data item.

  (From the What is RCU page.)

Thank You!