

Semaphores and Other Synchronization Primitives

CS 111

Summer 2025

Operating System Principles

Peter Reiher

Outline

- Semaphores
- Mutexes and object locking
- Getting good performance with locking

Semaphores

- A theoretically sound way to implement locks
 - With important extra functionality critical to use in computer synchronization problems
- Thoroughly studied and precisely specified
 - Not necessarily so usable, however
- Like any theoretically sound mechanism, could be gaps between theory and implementation

Computational Semaphores

- Concept introduced in 1968 by Edsger Dijkstra
 - Cooperating sequential processes
- THE classic synchronization mechanism
 - Behavior is well specified and universally accepted
 - A foundation for most synchronization studies
 - A standard reference for all other mechanisms
- More powerful than simple locks
 - They incorporate a FIFO waiting queue
 - They have a counter rather than a binary flag

Semaphores - Operations

- Semaphore has two parts:
 - An integer counter (initial value unspecified)
 - A FIFO waiting queue
- P (proberen/test) ... “wait”
 - Decrement counter, if count ≥ 0 , return
 - If counter < 0 , add process to waiting queue
- V (verhogen/raise) ... “post” or “signal”
 - Increment counter
 - If queue non-empty, wake one of the waiting process

Using Semaphores for Exclusion

- Initialize semaphore count to one 1
 - Count reflects # threads allowed to hold lock
- Use P/wait operation to take the lock
 - The first wait will succeed 0
 - Subsequent waits will block -1
- Use V/post operation to release the lock
 - Increment semaphore count to indicate one less waiting request 0
 - If any threads are waiting, unblock the first in line

Using Semaphores for Notifications

- Initialize semaphore count to zero 0
 - Count reflects # of completed events
- Use P/wait operation to await completion
 - If already posted, it will return immediately
 - Else all callers will block until V/post is called -1
- Use V/post operation to signal completion
 - Increment the count -1
 - If any threads are waiting, unblock the first in line
- One signal per wait: no broadcasts

Counting Semaphores

- Initialize semaphore count to ... 3
 - The number of available resources
- Use P/wait operation to consume a resource
 - If available, it will return immediately 0
 - Else all callers will block until V/post is called -1
- Use V/post operation to produce a resource
 - Increment the count 0
 - If any threads are waiting, unblock the first in line
- One signal per wait: no broadcasts

Semaphores For Mutual Exclusion

```
struct account {  
    struct semaphore s;      /* initialize count to 1, queue empty, lock 0 */  
    int balance;  
    ...  
};  
  
int write_check( struct account *a, int amount ) {  
    int ret;  
    wait( &a->semaphore );   /* get exclusive access to the account */  
  
    if ( a->balance >= amount ) { /* check for adequate funds */  
        amount -= balance;  
        ret = amount;  
    } else {  
        ret = -1;  
    }  
  
    post( &a->semaphore );   /* release access to the account */  
    return( ret );  
}
```

Limitations of Semaphores

- Semaphores are a very basic mechanism
 - They are simple, and have few features
 - More designed for proofs than synchronization
- They lack many practical synchronization features
 - Prone to certain synchronization problems
 - One cannot check the lock without blocking
 - They do not support reader/writer shared access
 - No way to recover from a wedged V operation
 - Interaction issues with priority scheduling
- Nonetheless, most OSs support them

Locking to Solve High Level Synchronization Problems

- Mutexes and object level locking
- Problems with locking
- Solving the problems

Mutexes

- A Linux/Unix locking mechanism
- Intended to lock sections of code
 - Locks expected to be held briefly
- Typically for multiple threads of the same process
- Low overhead and very general

Object Level Locking

- Mutexes protect code critical sections
 - Brief durations (e.g., nanoseconds, milliseconds)
 - Other threads operating on the same data
 - All operating in a single address space
- Persistent objects (e.g., files) are more difficult
 - Critical sections are likely to last much longer
 - Many different programs can operate on them
 - May not even be running on a single computer
- Solution: lock objects (rather than code)
 - Typically somewhat specific to object type

Linux File Descriptor Locking

int flock(*fd*, *operation*)

- Supported *operations*:
 - LOCK_SH ... shared lock (multiple allowed)
 - LOCK_EX ... exclusive lock (one at a time)
 - LOCK_UN ... release a lock
- Lock applies to open instances of same *fd*
 - Lock passes with the relevant fd
 - Distinct opens are not affected
- Locking with flock() is purely advisory
 - Does not prevent reads, writes, unlinks

Advisory vs Enforced Locking

- Enforced locking
 - Done within the implementation of object methods
 - Guaranteed to happen, whether or not user wants it
 - May sometimes be too conservative
- Advisory locking
 - A convention that “good guys” are expected to follow
 - Users expected to lock object before calling methods
 - Gives users flexibility in what to lock, when
 - Gives users more freedom to do it wrong (or not at all)
 - Mutexes and flocks() are advisory locks

Linux Ranged File Locking

int lockf(*fd, cmd, offset, len*)

- Supported *cmds*:
 - F_LOCK ... get/wait for an exclusive lock
 - F_ULOCK ... release a lock
 - F_TEST/F_TLOCK ... test, or non-blocking request
 - *offset/len* specifies portion of file to be locked
- Lock applies to file (not the open instance)
 - Process specific
 - Closing any fd for the file releases for all of a process' fds for that file
- Locking may be enforced
 - Depending on the underlying file system

Locking Problems

- Performance and overhead
- Contention
 - Convoy formation
 - Priority inversion

Performance of Locking

- Locking often performed as an OS system call
 - Particularly for enforced locking
- Typical system call overheads for lock operations
- If they are called frequently, high overheads
- Even if not in OS, extra instructions run to lock and unlock

Locking Costs

- Locking used when you need to protect critical sections to ensure correctness
- Many critical sections are very brief
 - In and out in a matter of nano-seconds
- Overhead of the locking operation may be much higher than time spent in critical section

What If You Don't Get Your Lock?

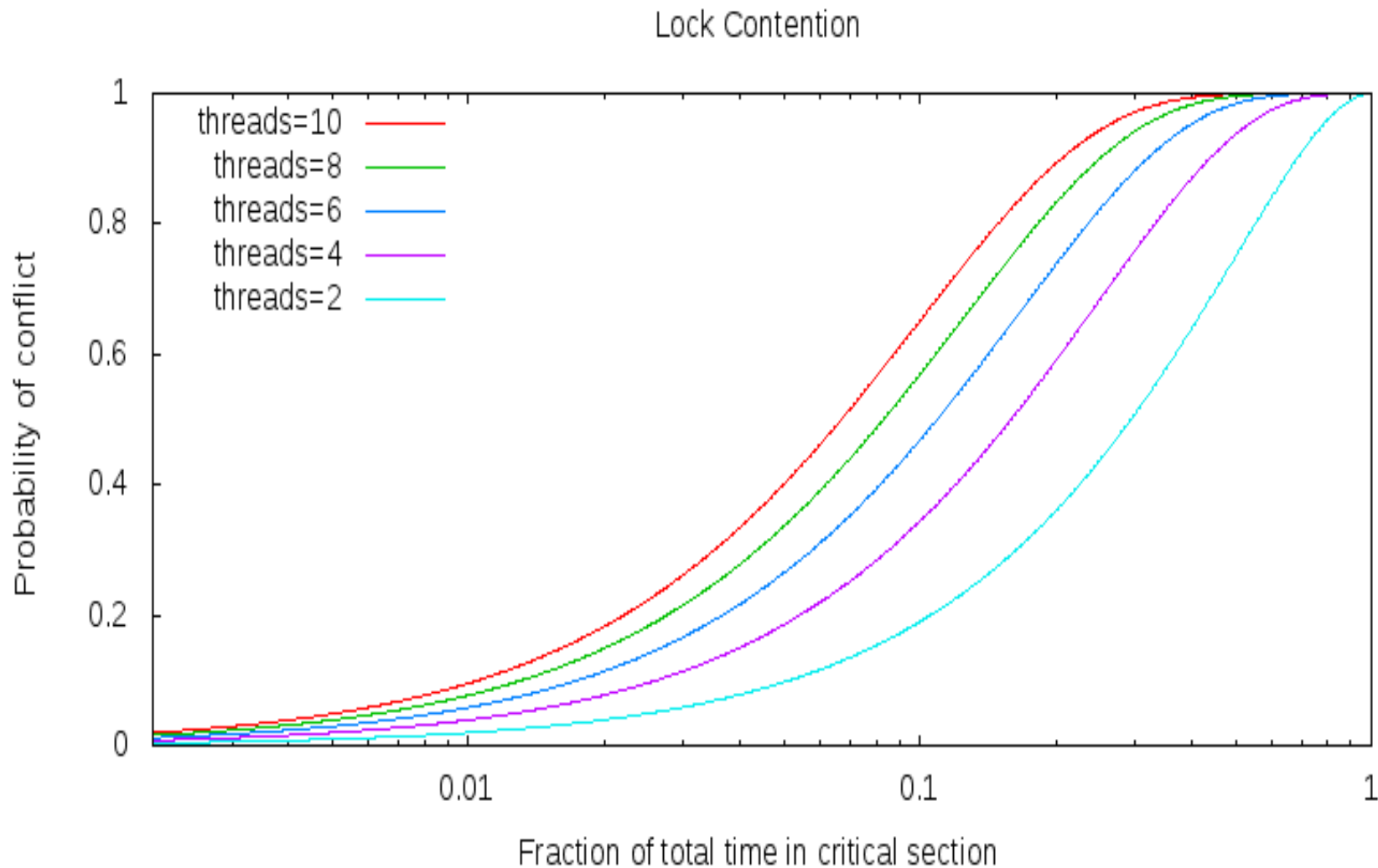
- Then you block
- Blocking is much more expensive than getting a lock
 - E.g., 1000x
 - Micro-seconds to yield and context switch
 - Milliseconds if swapped-out or a queue forms
- Performance depends on conflict probability

$$C_{\text{expected}} = (C_{\text{block}} * P_{\text{conflict}}) + (C_{\text{get}} * (1 - P_{\text{conflict}}))$$

What If Everyone Needs One Resource?

- One process gets the resource
- Other processes get in line behind him
 - Forming a *convoy*
 - Processes in a convoy are all blocked waiting for the resource
- Parallelism is eliminated
 - B runs after A finishes
 - C after B
 - And so on, with only one running at a time
- That resource becomes a *bottleneck*

Probability of Conflict



Convoy Formation

- In general

$$P_{\text{conflict}} = 1 - \underbrace{(1 - (T_{\text{critical}} / T_{\text{total}}))}_{\text{Nobody else in critical section at the same time}}^{\text{threads}}$$

Nobody else in critical section at the same time

- Unless a FIFO queue forms

$$P_{\text{conflict}} = 1 - (1 - ((T_{\text{wait}} + T_{\text{critical}}) / T_{\text{total}}))^{\text{threads}}$$

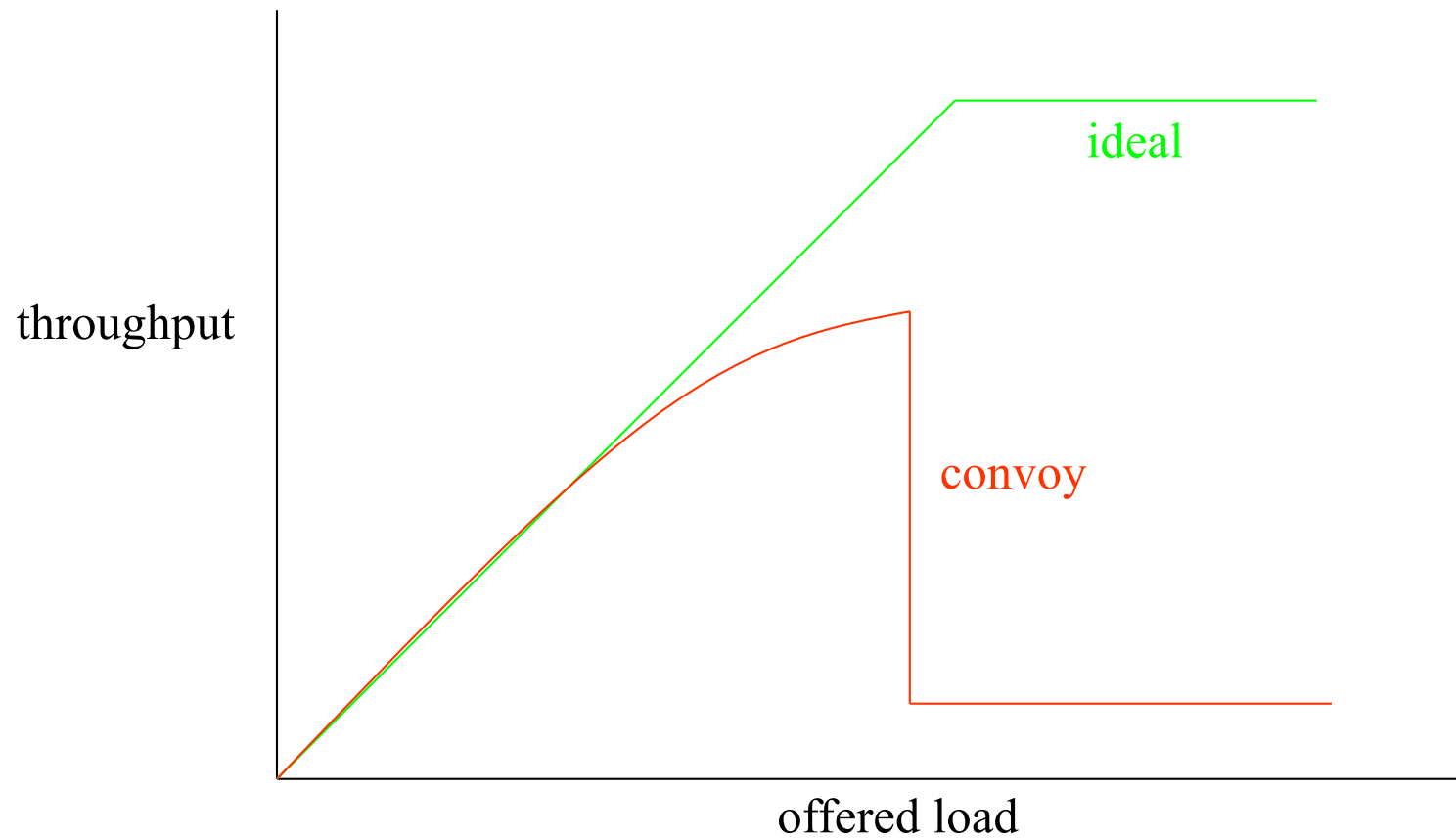
Newcomers have to get into line

And an (already huge) T_{wait} gets even longer

- If T_{wait} reaches the mean inter-arrival time

The line becomes permanent, parallelism ceases

Performance: Resource Convoys



Priority Inversion

- Priority inversion can happen in priority scheduling systems that use locks
 - A low priority process P1 has mutex M1 and is preempted
 - A high priority process P2 blocks for mutex M1
 - Process P2 is effectively reduced to priority of P1
- Depending on specifics, results could be anywhere from inconvenient to fatal

Priority Inversion on Mars



- A real priority inversion problem occurred on the Mars Pathfinder rover
- Caused serious problems with system resets
- Difficult to find

The Pathfinder Priority Inversion

- Special purpose hardware running VxWorks real time OS
- Used preemptive priority scheduling
 - So a high priority task should get the processor
- Multiple components shared an “information bus”
 - Used to communicate between components
 - Essentially a shared memory region
 - Protected by a mutex

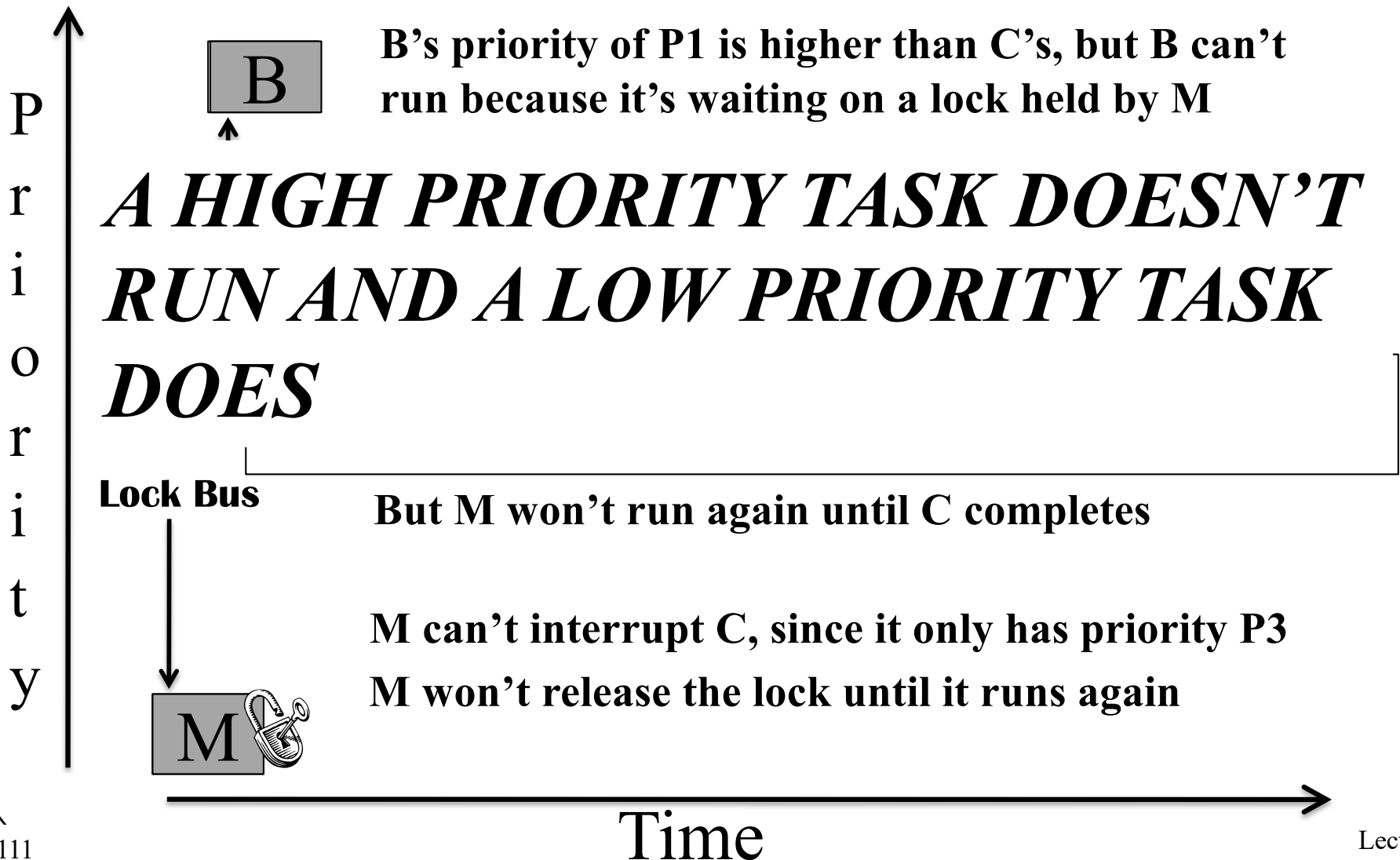
A Tale of Three Tasks

- A high priority bus management task (at P1) needed to run frequently
 - For brief periods, during which it locked the bus
- A low priority meteorological task (at P3) ran occasionally
 - Also for brief periods, during which it locked the bus
- A medium priority communications task (at P2) ran rarely
 - But for a long time when it ran
 - But it didn't use the bus, so it didn't need the lock
- $P1 > P2 > P3$

What Went Wrong?

- Rarely, the following happened:
 - The meteorological task ran and acquired the lock
 - And then the bus management task would run
 - It would block waiting for the lock
 - Don't pre-empt low priority if you're blocked anyway
- Since meteorological task was short, usually not a problem
- But if the long communications task woke up in that short interval, what would happen?

The Priority Inversion at Work



The Ultimate Effect

- A watchdog timer would go off every so often
 - At a high priority
 - It didn't need the bus
 - A health monitoring mechanism
- If the bus management task hadn't run for a long time, something was wrong
- So the watchdog code reset the system
- Every so often, the system would reboot

Handling Priority Inversion Problems

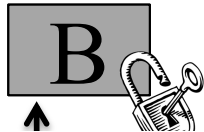
- In a priority inversion, lower priority task runs because of a lock held elsewhere
 - Preventing the higher priority task from running
- In the Mars Rover case, the meteorological task held a lock
 - A higher priority bus management task couldn't get the lock
 - A medium priority, but long, communications task preempted the meteorological task
 - So the medium priority communications task ran instead of the high priority bus management task

Solving Priority Inversion

- Temporarily increase the priority of the meteorological task
 - While the high priority bus management task was blocked by it
 - So the communications task wouldn't preempt it
 - When lock is released, drop meteorological task's priority back to normal
- *Priority inheritance*: a general solution to this kind of problem

The Fix in Action

P ↑
r

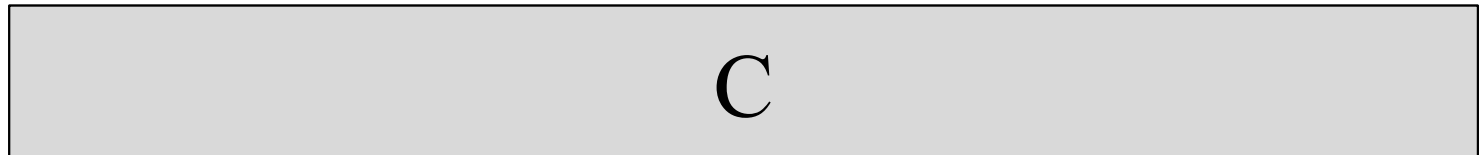


When M releases the
lock it loses high

...

***Tasks run in proper priority order and
Pathfinder can keep looking around!***

r
i
t
y



B now gets the lock
and unblocks



Time

Solving Locking Problems

- Reducing overhead
- Reducing contention

Reducing Overhead of Locking

- Not much more to be done here
 - Unless, perhaps, you wrote your own locking code
- Locking code in operating systems is usually highly optimized
- Certainly typical users can't do better

Reducing Contention

- Eliminate the critical section entirely
- Eliminate preemption during critical section
- Reduce time spent in critical section
- Reduce frequency of entering critical section
- Reduce exclusive use of the serialized resource
- Spread requests out over more resources

Eliminating Critical Sections

- Eliminate shared resource
 - Give everyone their own copy
 - Find a way to do your work without it
- Use atomic instructions
 - Only possible for simple operations
- Great when you can do it
- But often you can't

Eliminate Preemption in Critical Section

- If your critical section cannot be preempted, no synchronization problems
- May require disabling interrupts
 - As previously discussed, not always an option

Reducing Time in Critical Section

- Eliminate potentially blocking operations
 - Allocate required memory before taking lock
 - Do I/O before taking or after releasing lock
- Minimize code inside the critical section
 - Only code that is subject to destructive races
 - Move all other code out of the critical section
 - Especially calls to other routines
- Cost: this may complicate the code
 - Unnaturally separating parts of a single operation

Reduced Frequency of Entering Critical Section

- Can we use critical section less often?
 - Less use of high-contention resource/operations
 - Batch operations
- Consider “sloppy counters”
 - Move most updates to a private resource
 - Costs:
 - Global counter is not always up-to-date
 - Thread failure could lose many updates
 - Alternative:
 - Sum single-writer private counters when needed

Remove Requirement for Full Exclusivity

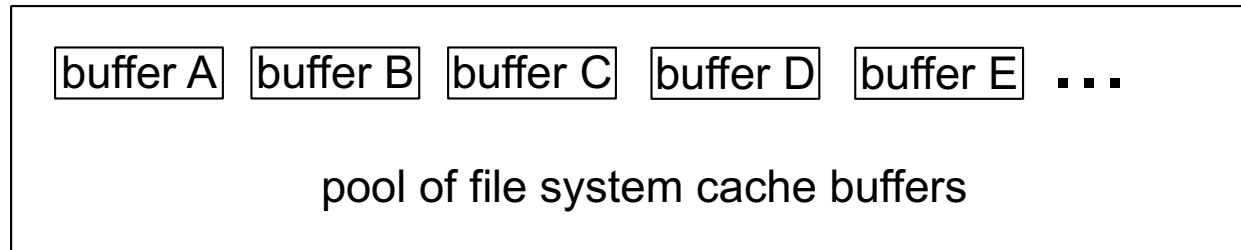
- Read/write locks
- Reads and writes are not equally common
 - File reads and writes: reads/writes > 50
 - Directory search/create: reads/writes > 1000
- Only writers require exclusive access
- Read/write locks
 - Allow many readers to share a resource
 - Only enforce exclusivity when a writer is active
 - Policy: when are writers allowed in?
 - Potential starvation if writers must wait for readers

Spread Requests Over More Resources

- Change lock granularity
- Coarse grained - one lock for many objects
 - Simpler, and more idiot-proof
 - Greater resource contention (threads/resource)
- Fine grained - one lock per object (or sub-pool)
 - Spreading activity over many locks reduces contention
 - Dividing resources into pools shortens searches
 - A few operations may lock multiple objects/pools
- TANSTAAFL
 - Time/space overhead, more locks, more gets/releases
 - Error-prone: harder to decide what to lock when

Lock Granularity – Pools vs. Elements

- Consider a pool of objects, each with its own lock



- Most operations lock only one buffer within the pool
- But some operations require locking the entire pool
 - Two threads both try to add buffer AA to the cache
 - Thread 1 looks for buffer B while thread 2 is deleting it
- The pool lock could become a bottle-neck, so
 - Minimize its use
 - Reader/writer locking
 - Sub-pools ...

The Snake in the Garden

- Locking is great for preventing improper concurrer
- With care perform v be made to
- But that c
- If we arer locking can
- lead to ou
- Deadlock

