Systemy Operacyjne :: Lista 6

Zadanie 1

> To “up” a futex, execute the proper assembler instructions that will cause the host CPU to atomically increment the integer. Afterward, check if it has in fact changed from 0 to 1, in which case there were no waiters and the operation is done. This is the noncontended case which is fast and should be common.
>
> In the contended case, the atomic increment changed the counter from -1 (or some other negative number). If this is detected, there are waiters. User space should now set the counter to 1 and instruct the kernel to wake up any waiters using the FUTEX_WAKE operation.
>
> Waiting on a futex, to “down” it, is the reverse operation. Atomically decrement the counter and check if it changed to 0, in which case the operation is done and the futex was uncontended. In all other circumstances, the process should set the counter to -1 and request that the kernel wait for another process to up the futex. This is done using the FUTEX_WAIT operation.

```
typedef int binarySemaphore;
binarySemaphore my_precious = 0;

bool
compare_and_swap( int *p, int old_val, int new_val ) {
    ATOMIC! {
        if *p != old_val {
            return false;
        }

        *p = new_val;
        return true;
    }
}

void
lock( binarySemaphore *s ) {
    if compare_and_swap( s, 0, 1 )
        return;

    do {
        if *s == 2 || compare_and_swap( s, 1, 2 ) {
            futex_wait( s, 2 );
        }
    } while( ! compare_and_swap( s, 0, 2 ) );
}

void
unlock( binarySemaphore *s ) {
    if ! compare_and_swap( s, 2, 1 ) {
        *s = 0;
        futex_wakeup( *s, 1 );
        return;
    }

    if ! compare_and_swap( s, 1, 0 ) {
        return;
    }
}
```

Zadanie 2

```
typedef struct {
    critsec: Mutex,
    wait: CondVar,
    count: int,
    waiters: int
} Semaphore;

void
init( Semaphore *s, int count ) {
    s -> critsec . init( );
    s -> wait . init( );
    s -> count = count;
    s -> waiters = 0;
}

void
wait( Semaphore *s ) {
    s -> critsec . lock();
    s -> count --;

    while( s -> count < 0 ) {
        if( s -> waiters < - (s -> count) ) {
            s -> waiters ++;
            s -> wait . wait( s -> critsec );
            s -> waiters --;
        } else {
            break;
        }
    }

    s -> critsec . unlock();
}

void
post( Semaphore *s ) {
    s -> critsec . lock();

    if( ++ s -> count <= 0 ) {
        s -> wait . signal( );
    } else {
       s -> critsec . unlock();
    }
}
```

Zadanie 3

```
typedef struct {
    critsec: Mutex,
    readers: int,
    noreaders: CondVar,
    nowriter: CondVar,
    writer: boolean
} RWLock;

void
init( RWLock *l ) {
```

```
    critsec . init( );
    readers = 0;
    noreaders . init( );
    nowriter . init( );
    writer = false;
}

void
rdlock( RWLock *l ) {
    l -> critsec . lock ();
    while( writer ) {
        l -> nowriter . wait( l -> critsec );
    }

    readers ++;
    l -> critsec . unlock ();
}

void
wrlock( RWLock *l ) {
    l -> critsec . lock( );

    while( l -> readers > 0 ) {
        l -> noreaders . wait( l -> critsec );
    }

    l -> writer = true
    l -> critsec . unlock( );
}

void
unlock( RWLock *l ) {
    l -> critsec . lock ();

    if( l -> writer ) {
        l -> writer = false;
        l -> noreaders . signal ();
    }
    else if( l -> readers > 0 ) {
        if( -- l -> readers == 0 ) {
            l -> noreaders . signal ();
        }
    }

    l -> critsec . unlock ();
}
```

Zadanie 4

Zasada działania

This is very useful synchronization mechanism to provide a lightweight and scalable lock for the scenario where many readers and a few writers are present. Sequence lock maintains a counter for sequence. When the shared data is written, a lock is obtained and a sequence counter is incremented by 1. Write operation makes the sequence counter value to odd and releasing it makes even. In case of reading, sequence counter is read before and after reading the data. If the values are the same which indicates that a write did not begin in the middle of the read. In addition to that, if the values are even, a write operation is not going on. Sequence lock gives the high priority to writers compared to readers. An acquisition of the write lock always succeeds if there are no

other writers present. Pending writers continually cause the read loop to repeat, until there are no longer any writers holding the lock. So reader may sometimes be forced to read the same data several times until it gets a valid copy(writer releases lock). On the other side writer never waits until and unless another writer is active.

Zadanie 5

The ability to wait until all readers are done allows RCU readers to use much lighter-weight synchronization—in some cases, absolutely no synchronization at all. In contrast, in more conventional lock-based schemes, readers must use heavy-weight synchronization in order to prevent an updater from deleting the data structure out from under them. This is because lock-based updaters typically update data items in place, and must therefore exclude readers. In contrast, RCU-based updaters typically take advantage of the fact that writes to single aligned pointers are atomic on modern CPUs, allowing atomic insertion, removal, and replacement of data items in a linked structure without disrupting readers. Concurrent RCU readers can then continue accessing the old versions, and can dispense with the atomic read-modify-write instructions, memory barriers, and cache misses that are so expensive on modern SMP computer systems, even in absence of lock contention.[14][15] The lightweight nature of RCU's read-side primitives provides additional advantages beyond excellent performance, scalability, and real-time response. For example, they provide immunity to most deadlock and livelock conditions.[note 4]

Of course, RCU also has disadvantages. For example, RCU is a specialized technique that works best in situations with mostly reads and few updates, but is often less applicable to update-only workloads. For another example, although the fact that RCU readers and updaters may execute concurrently is what enables the lightweight nature of RCU's read-side primitives, some algorithms may not be amenable to read/update concurrency.

Despite well over a decade of experience with RCU, the exact extent of its applicability is still a research topic.