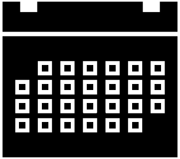
System Level Programming

Daniel Gruss

2024-03-07

My Code

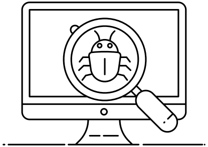


```c
void fun(size_t x)
{
  for (size_t i = 0; i < 1000000ULL; ++i)
    counter += (size_t)x;
}

int main()
{
  pthread_t t;
  pthread_create(&t,0,(void*(*)(void*))&fun,(void *)1);
  pthread_create(&t,0,(void*(*)(void*))&fun,(void *)2);
  mypause();
  printf("counter = %zu\n",counter);
  return 0;
}
```


I MULTITHREADED MY PROGRAM
THAT COMPUTES 1000000 + 2000000
THE RESULT IS 2999994
imgflip.com

With printf debugging



```
T1 adds 1 to 550209
T1 adds 1 to 550210
T1 adds 1 to 550211
T1 adds 1 to 550212
T1 adds 1 to 550213
T1 adds 1 to 550214
T1 adds 1 to 550215     <-- look
    T2 adds 2 to 550122 <-- at
    T2 adds 2 to 550125 <-- these
    T2 adds 2 to 550127
    T2 adds 2 to 550129
    T2 adds 2 to 550131
    T2 adds 2 to 550133
```

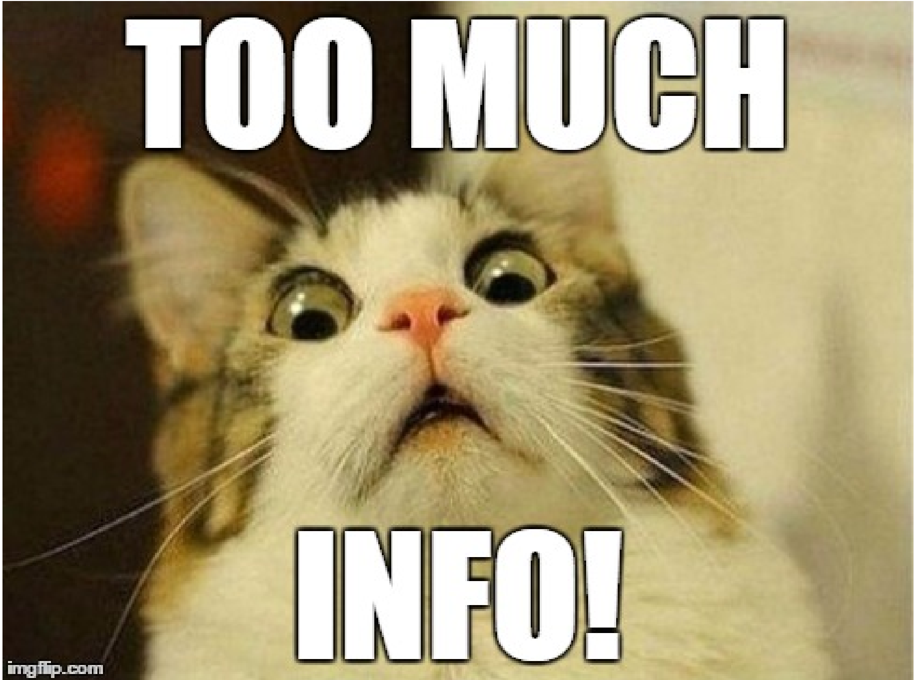
Intel® 64 and IA-32 A
Volumes: 1, 2A, 2B, 2

Last updated: November 16, 2020

› **File:**
325462-sdm-vol-1-2abcd-3abcd.pdf

› **Size:**
56.59 MB

Download


TOO MUCH
INFO!
imgflip.com

8.1.1 Guaranteed Atomic Operations

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte
- Reading or writing a word aligned on a 16-bit boundary
- Reading or writing a doubleword aligned on a 32-bit boundary

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line


INCREMENTS ARE NOT ATOMIC?
...
imgflip.com

How toilet locks work

How toilet locks work

How toilet locks work

How toilet locks work

1. Enter public toilet room

How toilet locks work

1. Enter public toilet room
2. Use toilet door

How toilet locks work

1. Enter public toilet room
2. Use toilet door
   2.1 Check color indicator (is it free?)

How toilet locks work

1. Enter public toilet room
2. Use toilet door
    2.1 Check color indicator (is it free?)
    2.2 If toilet is free:

How toilet locks work

1. Enter public toilet room
2. Use toilet door
   - 2.1 Check color indicator (is it free?)
   - 2.2 If toilet is free:
     - 2.2.1 Pass through door + **lock** door

How toilet locks work

1. Enter public toilet room
2. Use toilet door
    - 2.1 Check color indicator (is it free?)
    - 2.2 If toilet is free:
        - 2.2.1 Pass through door + **lock** door
    - 2.3 Else → back to step **2.1**

How toilet locks work

1. Enter public toilet room
2. Use toilet door
   - 2.1 Check color indicator (is it free?)
   - 2.2 If toilet is free:
     - 2.2.1 Pass through door + **lock** door
   - 2.3 Else → back to step **2.1**
3. Use toilet

How toilet locks work

1. Enter public toilet room
2. Use toilet door
    2.1 Check color indicator (is it free?)
    2.2 If toilet is free:
        2.2.1 Pass through door + **lock** door
    2.3 Else → back to step **2.1**
3. Use toilet
4. Use toilet door again

How toilet locks work

1. Enter public toilet room
2. Use toilet door
   - 2.1 Check color indicator (is it free?)
   - 2.2 If toilet is free:
     - 2.2.1 Pass through door + **lock** door
   - 2.3 Else $\rightarrow$ back to step **2.1**
3. Use toilet
4. Use toilet door again
   - 4.1 Pass through door + **unlock** door

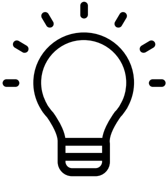
Let's code this!

Use toilet door (entry):

1. Check color indicator (is it free?)
2. If toilet is free:
   2.1 Pass through door + **lock** door
3. Else → back to step **2.1**

```
while (toilet_indicator != FREE)
{
  // busy wait - doing nothing
  // ugh, it's really urgent!
  // + i'm wasting time here
}
toilet_indicator = IN_USE;
```

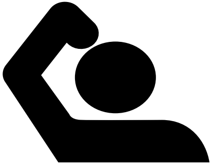
Spinlock



```
// return 0 if locking was successful
size_t lock(size_t* lock) {
  if (*lock == 0) // not locked
  {
    *lock = 1; // now locked
    return 0;
  }
  return 1;
}
```

POSIX: 0 means success!

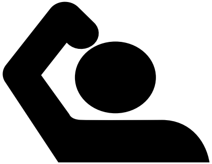
Spinlock



```
size_t lock(size_t* lock) {
  if (*lock == 0) // not locked
  {
    *lock = 1; // now locked
    return 0;
  }
  return 1;
}
```

Any problems here?

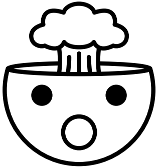
Spinlock



```
size_t lock(size_t* lock) {
  if (*lock == 0) // not locked
  {
    *lock = 1; // now locked
    return 0;
  }
  return 1;
}
```

Any problems here? It's not spinning!

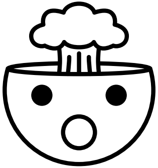
Spinlock



```
size_t lock(size_t* lock) {
  while (*lock == 1) // not locked
  {
    // busy wait
  }
  *lock = 1; // now locked
  return 0;
}
```

Any problems here?

Spinlock



```
size_t lock(size_t* lock) {
  while (*lock == 1) // not locked
  {
    // busy wait
  }
  *lock = 1; // now locked
  return 0;
}
```

Any problems here? It's not atomic!

POSIX to the rescue

```
#include <pthread.h>
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

POSIX to the rescue

```c
#include <pthread.h>
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

The `pthread_spin_lock()` function **locks** the spin lock referred to by `lock`. If the spin lock is currently unlocked, the calling thread acquires the lock immediately. If the spin lock is currently **locked** by another thread, the calling thread **spins**, testing the lock until it becomes available, at which point the calling thread acquires the lock.

Spinlocks are not efficient

Spinlocks are not efficient

Spinlocks are not efficient

Idea: Instead of busy waiting,

Spinlocks are not efficient

Idea: Instead of busy waiting,

- put thread to **sleep**,

Spinlocks are not efficient

Idea: Instead of busy waiting,

- put thread to **sleep**,
- keep a **list of sleeping threads**,

Spinlocks are not efficient

Idea: Instead of busy waiting,

- put thread to **sleep**,
- keep a **list of sleeping threads**,
- wake up a sleeping thread when unlocking.

Spinlocks are not efficient

Idea: Instead of busy waiting,

- put thread to **sleep**,
- keep a **list of sleeping threads**,
- wake up a sleeping thread when unlocking.

Spinlocks are not efficient

Idea: Instead of busy waiting,

- put thread to **sleep**,
- keep a **list of sleeping threads**,
- wake up a sleeping thread when unlocking.

→ We call this a Mutex!

Mutexes

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Mutexes

```c
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

The mutex object referenced by `mutex` shall be locked by a call to `pthread_mutex_lock()` that returns zero. If the mutex is already locked by another thread, the calling thread shall **block until** the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its **owner**.

How to implement events?

```
while (go_eat == 0)
{
  pthread_mutex_lock(&food_ready_mutex);
  if (food_ready)
    go_eat = 1;
  pthread_mutex_unlock(&food_ready_mutex);
}
goEat();
```

Wait, that's busy wait AGAIN!

Condition Variables

- Synchronization mechanism

Condition Variables

- Synchronization mechanism
- Not inherently thread-safe:

Condition Variables

- Synchronization mechanism
- Not inherently thread-safe:
    - Using mutex to make it thread-safe!

Condition Variables

- Synchronization mechanism
- Not inherently thread-safe:
  - Using mutex to make it thread-safe!
- Three main operations:

Condition Variables

- Synchronization mechanism
- Not inherently thread-safe:
  - Using mutex to make it thread-safe!
- Three main operations:
  1. `wait` - wait for an event

Condition Variables

- Synchronization mechanism
- Not inherently thread-safe:
  - Using mutex to make it thread-safe!
- Three main operations:
  1. `wait` - wait for an event
  2. `signal` - wake up 1 waiting thread

Condition Variables

- Synchronization mechanism
- Not inherently thread-safe:
  - Using mutex to make it thread-safe!
- Three main operations:
  1. `wait` - wait for an event
  2. `signal` - wake up 1 waiting thread
  3. `broadcast` - wake up ALL waiting threads

Condition Variables

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict
    mutex);
```

Condition Variables

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict
    mutex);
```

The `pthread_cond_wait()` functions shall block on a condition variable. The application shall ensure that these functions are called with `mutex` locked by the calling thread. These functions atomically release `mutex` and cause the calling thread to block on the condition variable `cond`. Upon return, the mutex shall have been locked and shall be owned by the calling thread.

pthread_cond_wait pseudo code

```
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict
    mutex)
{
  // atomic begin
  add_myself_to_sleepers_list();
  pthread_mutex_unlock(mutex);
  go_to_sleep();
  // atomic end
  // wait to be woken up
  pthread_mutex_lock(mutex);
}
```

Condition Variables

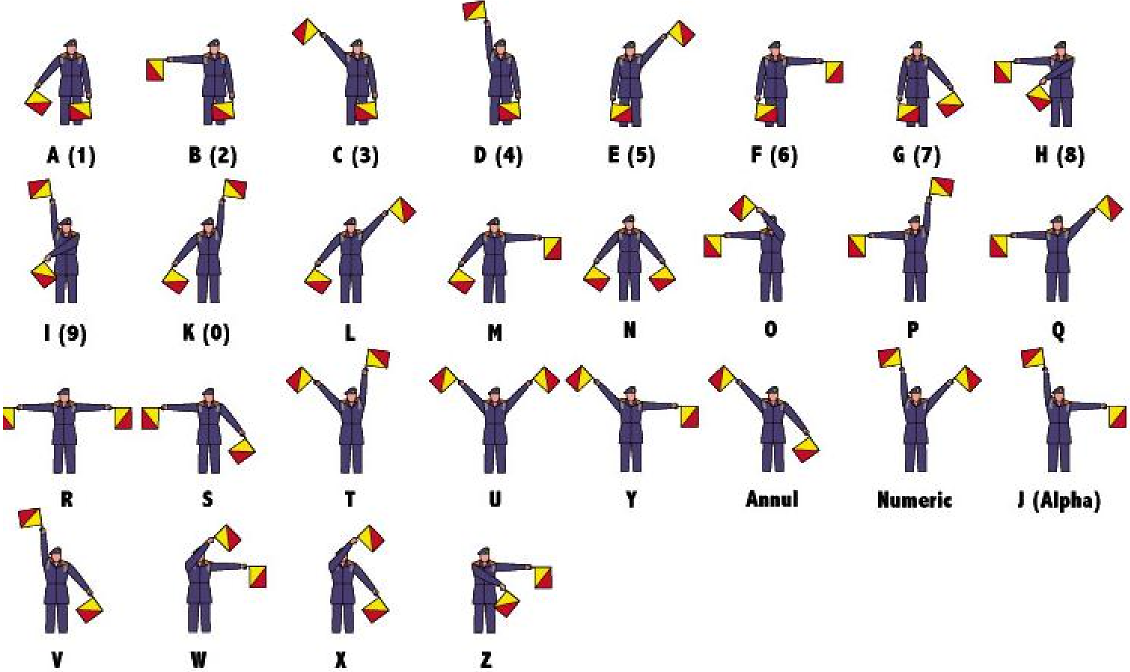
```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

Condition Variables

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

The `pthread_cond_broadcast()` function shall unblock all threads currently blocked on the specified condition variable `cond`.

The `pthread_cond_signal()` function shall unblock at least one of the threads that are blocked on the specified condition variable `cond` (if any threads are blocked on `cond`).


A (1)
B (2)
C (3)
D (4)
E (5)
F (6)
G (7)
H (8)
I (9)
K (0)
L
M
N
O
P
Q
R
S
T
U
Y
Annul
Numeric
J (Alpha)
V
W
X
Z

Semaphores

- stores a numerical value ≥ 0

Semaphores

- stores a numerical value ≥ 0
- two operations:

Semaphores

- stores a numerical value ≥ 0
- two operations:
  1. `wait` $=$ decrement

Semaphores

- stores a numerical value ≥ 0
- two operations:
    1. `wait` = decrement
    2. `post` = increment

Semaphores

- stores a numerical value ≥ 0
- two operations:
  1. `wait` = decrement
  2. `post` = increment

$\rightarrow$ what happens when decrementing at value 0?

Semaphores

- stores a numerical value ≥ 0
- two operations:
  1. `wait` = decrement
  2. `post` = increment

$\rightarrow$ what happens when decrementing at value 0?

$\rightarrow$ semaphore blocks

Semaphore vs Mutex

Mutex is basically a semaphore with

- numerical values 0 (locked) or 1 (free)

Semaphore vs Mutex

Mutex is basically a semaphore with

- numerical values 0 (locked) or 1 (free)
    1. `wait` = `lock`

Semaphore vs Mutex

Mutex is basically a semaphore with

- numerical values 0 (locked) or 1 (free)
  1. `wait` = `lock`
  2. `post` = `unlock`

Semaphore vs CVs

Synchronization of events with semaphores:

- semaphores are not *owned/held* by any thread

Semaphore vs CVs

Synchronization of events with semaphores:

- semaphores are not *owned/held* by any thread
    1. `wait` $\approx$ `cond_wait`

Semaphore vs CVs

Synchronization of events with semaphores:

- semaphores are not *owned/held* by any thread
    1. `wait` ≈ `cond_wait`
    2. `post` ≈ `cond_signal`