Operating Systems

Threads

Sven Dziadek

October/November 2022

EPITA

Pthreads

POSIX Threads: Pthreads

Link with `-pthread`

```
#include <pthread.h>
```

Create threads

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
```

Race Conditions

Race Conditions

Race Condition

Result depends on timing

If some possible results are undesirable $\Rightarrow$ bug

Example (Too-much-milk problem)

Consider the following algorithm in a shared flat:

- Check fridge for milk
- Go to buy milk
- Put milk into fridge

Race Condition: Demo

Start **two threads** that increment a shared variable `c`:

```
for(i=0;i<1000;i++)
    c++
```

What is the final value of `c`?

See race.c

Race Condition: Demo

c++ translates to:

```
0x11a3 <fnC+30>:  mov    0x2eb3(%rip),%eax        # load 0x405c <c> into %eax
0x11a9 <fnC+36>:  add    $0x1,%eax                # %eax = %eax + 1
0x11ac <fnC+39>:  mov    %eax,0x2eaa(%rip)        # store %eax to 0x405c <c>
```

Thread 1	**Thread 2**
load c $=0$	
increment c $=1$	
	load c $=0$
store c $=1$	
	increment c $=1$
	store c $=1$

Synchronization Tools

Hardware: Atomic Operations

Atomic Operations in Hardware

CPUs can block memory bus for a few instructions

GCC provides (for any integral scalar or pointer type as type):

```
type __atomic_add_fetch (type *ptr, type val, int memorder);
type __atomic_fetch_add (type *ptr, type val, int memorder);
bool __atomic_compare_exchange (type *ptr, type *expected, type *desired, bool weak,
                                int success_memorder, int failure_memorder);
```

Also for more operations

See `https://gcc.gnu.org/onlinedocs/gcc-10.2.0/gcc/_005f_005fatomic-Builtins.html`

Atomic Operations: Demo

race.c

Pthreads: Mutex

Lock for **mut**ual **ex**clusion:

- lock()
- `critical section...`
- unlock()

```c
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Mutex Demo

race.c

Locking: Notions

acquire lock

critical section

release lock

Critical Section

Block of code (accessing shared resources) that must only be entered **at most once** at a time

Effective goal: serialization

Contention

Try to acquire a lock that is already locked

Acquire a free lock = uncontended

Note: High contention decreases performance

$\Rightarrow$ keep critical sections short / use **finegrained** locking

Pthreads: Spinlock

Like a mutex but
does not block

To be used for short duration only
(less than time of two context switches)
with real-time scheduling policies

```
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Pthreads: Semaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

Value indicates how many resources are free

Blocks if value would become negative

See `sem_overview(7)`

Monitors

Monitor

Construct or **abstract data type**

- for mutual exclusion
- with ability to wait for conditions

Example: Java's **synchronized** methods

Condition variables allow to

- wait for conditions to be fulfilled
- block on false conditions
  and release of lock while waiting

Pattern to be solved:

```
lock()
while (!condition) { //must be atomic
    unlock()
    sleep() //release lock while waiting
    lock()
}
/* critical section */
unlock()
```

Monitor Example: Producer-Consumer Problem

Producer-Consumer Problem

Resource synchronization problem
to distribute work packages/data

Producer threads distribute work packages
Consumer threads collect work packages

Assume a ring buffer of fixed size

Producer threads can push to buffer
    if not full

Consumer threads can pop from buffer
    if not empty

Note:

Check for empty/full buffer
only while holding **lock**

If condition not fulfilled
$\Rightarrow$ release lock while waiting
$+$ blocking while waiting preferred

Solution:

Use **mutex** together
with **condition variables**

Pthreads: Condition Variables

```c
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Note that wait unlocks `mutex` while waiting

Use `signal` or `broadcast` depending on how many threads should be woken up

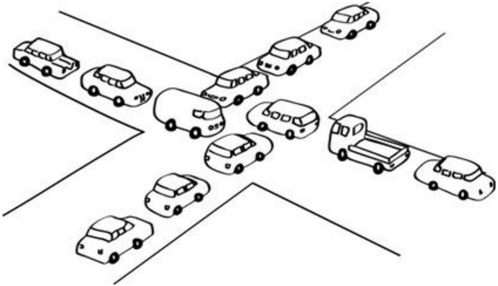
Deadlocks

Deadlock

Deadlock

Threads are blocking each other

Must be prevented at all costs

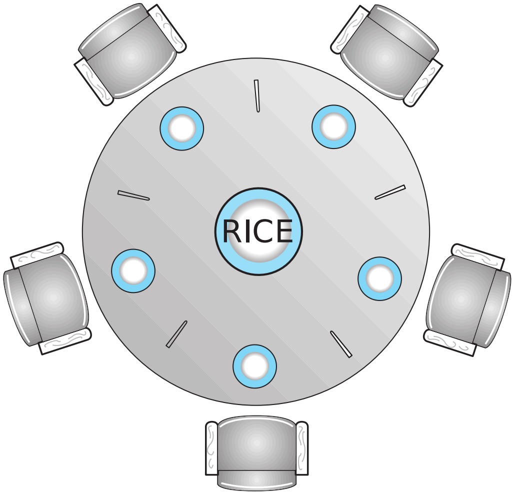


Note: Locking mutex twice can also lead to deadlock (Pthreads allows recursive mutex)

Deadlock: Necessary Conditions

- Mutual Exclusion
- Hold one and wait for other resource
- Resources cannot be preempted
- Circular wait

Dining Philosopher's Problem



Algorithm for each of the five philosophers:

- Think and then take left chopstick
- Think and then take right chopstick
- Eat some rice
- Put left chopstick back
- Put right chopstick back
- Repeat

Possible solutions: Break symmetry or atomically pick both chopsticks

Linux Kernel Support

Futex

Fast Userspace Mutex

`System call futex(2) only called in contended case`

Fast in uncontended case (only atomic operation)

Contended case: kernel manages wait queue

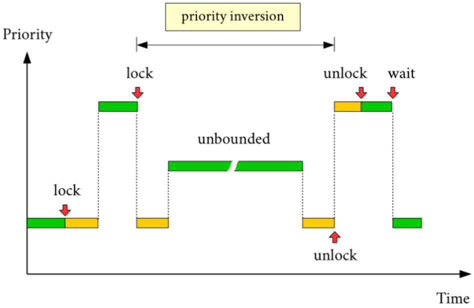
Not normally used directly

but used in Pthread library

See `futex(7)`

More Theory

Priority Inversion in Real-Time Scheduling



Priority Inversion (PI)

Blocking of a high-priority job by a lower-priority job

Unbounded PI must be prevented

Peterson's Algorithm (For two threads – thread 0 and thread 1)

```
int turn = 0; char flag[2] = {0, 0};

void lock(int me) {
    int other = 1-me;

    flag[me] = 1;                      /* I want to enter */
    turn = other;                      /* You can enter next */

    while (flag[other]                 /* If you want to enter */
           && turn == other)           /* and it's your turn */
        ; //busy wait                  /* I'll wait */
}

void unlock(int me) {
    flag[me] = 0;                      /* I don't want to enter anymore */
}
```

Peterson's Algorithm

Provably correct for mutual exclusion

Guarantees progress and bounded waiting

Why not to build your own synchronization tools

Demo: Peterson's Algorithm does not work on modern machines

Compiler & Processor Optimizations

Compiler can reorder instructions

CPU reorders memory accesses

Memory barriers can help: `__atomic_thread_fence(__ATOMIC_SEQ_CST)`

Demo: Peterson's Algorithm works with memory barrier

Note: Writing synchronization tools is hard $\Rightarrow$ use existing tools!