Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Operating Systems

Part 2: Concurrency – 7) More Locking, Condition Variables

Revision: `master@d08e360 (20220915-171922)`

BTI1341 / Fall 2022-23

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

Outline

Efficient Locks

Comparing Locks

Remember our overview from last time:

Lock Type	Correctness	Fairness	Performance
1) Disabled Interrupts[1]	✔	✖	✖
2) Flag Variables	✖	✖	✖
3) Spin Locks	✔	✖	✖[2]
4) Ticket Locks	✔	✔	✖[2]

Let's try to fix lock performance...

[1] Not usable in practice, included for completeness only.

[2] Reasonable on multi-core systems only.

The Problem With Spin Locks

Example situation:

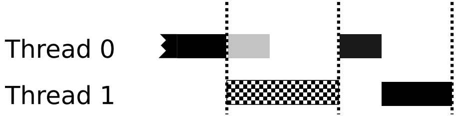

Figure: Spin-Waiting

1. Thread 0 is in critical section (*lock held*).
2. It is *interrupted* (e.g. scheduling decision).
3. Thread 1 is scheduled and spin-waits for the lock. It *uses up* all of its scheduling quantum!
4. Thread 0 is scheduled again and can finish its work.
5. Thread 1 can now acquire the lock and continue.

Now think about multiple threads waiting for the lock...

Another Problem: Priority Inversion

A different problem with spin locks, which *affects correctness*, is priority inversion :

1. Assume two threads: T0 (low priority), T1 (high priority).
2. T1 is blocked, T0 is scheduled.
3. T0 acquires a spin lock.
4. T1 is now unblocked, wants to acquire the lock and starts spinning.

Due to the fact that T1 has higher priority, T0 will *never be scheduled* again and thus cannot release the lock. The system effectively blocks forever!

NB: priority inversion can also occur without spin locks when having multiple threads with different priorities.

A First Solution: `yield()`

To increase performance, hardware support alone is not sufficient, the OS has to provide support for yielding .

`yield()` is a system call, which moves the current thread from *running* to *ready*. This allows the OS then to schedule another thread/process without the current thread wasting its whole quantum; i.e., the thread de-schedules itself:

```
void init() {
  flag = 0;
}

void lock() {
  while (TestAndSet(&flag, 1) == 1)
    yield();   // <==
}

void unlock() {
  flag = 0;
}
```

A First Solution: `yield()` (cont.)

Using `yield()`, the example could look like this:

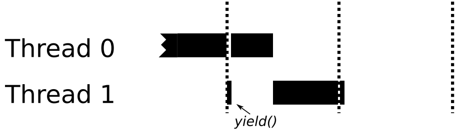

Figure: Using `yield()`

There are two issues with this solution:

1. It *does not scale*: E.g. for 100 threads, in the worst case, 99 have to yield first (including a context switch every time!).
2. It *does not ensure fairness*: Depending on scheduling etc., a thread may be caught in an endless loop yielding (starvation), while others are running.

Using Queues to Sleep

The cause for the issues with `yield()` is the scheduler: It has no knowledge about thread dependencies and may repeatedly make *"wrong" scheduling decisions*. To fix this, we require some control about which thread will be scheduled next.

A *queue* can be used to track the *order* in which to wake up the threads. Some more OS support is then required to wake up the correct thread.

Basic idea: syscalls `park()` and `unpark(thread_id)`.[3]

`park()` puts the current thread to sleep (like `yield()`), `unpark(thread_id)` wakes up the thread with the *given ID*.

[3] As they make the basic algorithm easier to understand, we use these Solaris syscalls for the example.

Queue Example

```c
struct lock {
  int flag;
  int guard;
  queue_t q;
};

void lock(struct lock lck) {
  while (TestAndSet(&lck->guard, 1) == 1)
    ;   // spin lock, only for flag (short)
  if (lck->flag == 0) {
    lck->flag = 1;   // lock acquired!
    lck->guard = 0;
  } else {
    queue_add(lck->q, gettid());   // add self to queue
    lck->guard = 0;
    park();   // go to sleep
  }
}
```

Queue Example (cont.)

```c
void unlock(struct lock lck) {
  while (TestAndSet(&lck->guard, 1) == 1)
    ;   // spin lock, only for flag (short)
  if (queue_empty(lck->q))
    lck->flag = 0;    // no threads waiting for lock
  else
    // keep lock for next thread and unpark it
    unpark(queue_remove(lck->q));
  lck->guard = 0;
}
```

Wake-up/Waiting Race

The code given is prone to a wake-up/waiting race : Assume a thread gets *interrupted just before park()*. If the thread holding the lock is then scheduled and releases it, the former thread will never wake up! The problem is here:

```
...
queue_add(lck->q, gettid());
lck->guard = 0;
park();
```

A solution for this is another syscall, `setpark()`:

```
...
queue_add(lck->q, gettid());
setpark();    // new
lck->guard = 0;
park();
```

If a thread is interrupted and `unpark()` is called, the subsequent `park()` then *returns immediately*.

Two-Phase Locks

A hybrid idea used in different systems are two-phase locks . As the name implies, such a lock consists of two phases:

1. For a *short time* period, perform *spin-waiting*.
2. If, during that period, the lock cannot be acquired, *yield* / go to sleep.

The idea here is, that it might be advantageous to keep a thread scheduled (running) in order to *avoid switching overhead* if the lock will be released shortly.

For pthreads, `pthread_spinlock_t` offers locks which perform spin-waiting. However, with modern schedulers, using spin locks has become less attractive.

Linux Futexes

The low-level mechanism used by *Linux NPTL*[4] is the futex (see "`man 2 futex`"), which is basically an arbitrary 32-bit number (variable) in *user memory*.[5]

Futexes support different operations, the two most important ones are:

`FUTEX_WAIT` Compares the variable to a given value and blocks the thread if equal. If blocking, it later returns with 0, otherwise it immediately returns an error.
This operation is atomic and the kernel guarantees its total order.

`FUTEX_WAKE` Wakes a specified number of the waiting threads.

[4] The *Native POSIX Threads Library*, i.e. the standard implementation of pthreads since Linux 2.6

[5] Also on 64 bit systems.

Mutexes Using Futexes

To conclude, we outline the basic idea of implementing a mutex using futexes:[6]

```
void lock(int lck) {
  int current;
  while ((current = FetchAndAdd(&lck)) != 0) {
    futex_wait(&lck, current+1);
  }
}

void unlock(int lck) {
  lck = 0;
  futex_wake(&lck, 1); // wake 1 thread
}
```

See [Dre04] for a good introduction to futexes and a correct mutex implementation.

[6] ⚠ This code is flawed and important parts are missing. Also, there are no `futex_wait()` and `futex_wake()` functions, the syscall must be invoked directly.

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Concurrent Data Structures

Overview

This section provides an overview of concurrent data structures, also called thread-safe data structures:

- How to add locks to some example data structures.
- How to deal with performance issues.
- The focus lies on *ideas* used and the *type of thinking* required.
- Concurrent data structures are a field of its own, we only give a *broad overview*.

⚠ *From here on, all code using the pthreads API is shown without error handling for brevity. In real code, return values must always be checked!*

A Simple Counter

The following is a simple, *non-concurrent* counter:

```c
struct counter {
  int value;
};

void init(struct counter *c) {
  c->value = 0;
}

int get(struct counter *c) {
  return c->value;
}

void increment(struct counter *c) {
  c->value++;
}

void decrement(struct counter *c) {
  c->value--;
}
```

A Counter With Locks

This is the same counter *with locking* added:

```
struct counter {
  int value;
  pthread_mutex_t lock;
};

void init(struct counter *c) {
  c->value = 0;
  pthread_mutex_init(&c->lock, null);
}

int get(struct counter *c) {
  pthread_mutex_lock(&c->lock);
  return c->value;
  pthread_mutex_unlock(&c->lock);
}
```

A Counter With Locks (cont.)

```c
void increment(struct counter *c) {
  pthread_mutex_lock(&c->lock);
  c->value++;
  pthread_mutex_unlock(&c->lock);
}

void decrement(struct counter *c) {
  pthread_mutex_lock(&c->lock);
  c->value--;
  pthread_mutex_unlock(&c->lock);
}
```

Approximate Counting

The way the locks are added to the counter scales poorly and has a *large performance impact*.

A possible solution is an approximate counter, it works as follows:

1. Have a single *counter per CPU core*, each with its own lock.
2. There is a *global counter*, with a global lock.
3. Every thread only updates the counter for its core.
4. When a local counter reaches a certain *threshold*, its value is transferred to the global counter (and it is reset afterwards).

This is a *tradeoff*: The smaller the threshold, the more accurate the global counter is. At the same time, the larger the threshold, the better the performance. A counter with a small threshold behaves more like a normal counter.

Approximate Counter Example

The following trace shows a hypothetical counting process for an approximate counter on 4 cores:

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	$5 \rightarrow 0$	1	3	4	5
7	0	2	4	$5 \rightarrow 0$	10

Approximate Counter Benchmark

The following figure shows measurements for 1 to 4 threads, each updating a counter to 1 million (on an Intel Core i5 CPU with 2.7 GHz). It compares the two counter types given before:

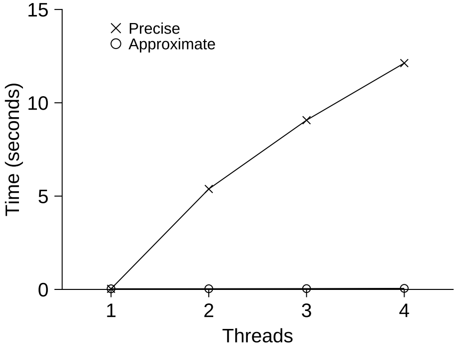

Figure: Counter With Locks vs. Approximate Counter

Courtesy of [ADAD18]

The Impact of Threshold

In the following figure, the impact of varying sizes for the threshold value on the same experiment can be seen:

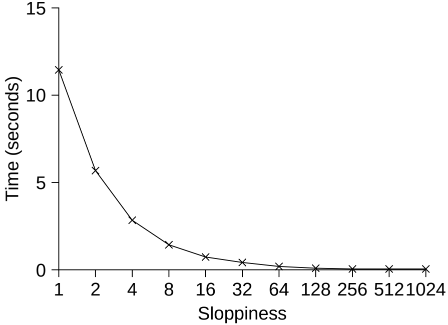

Figure: The Impact of Threshold

Courtesy of [ADAD18]

Linked Lists (Poor Quality)

The following is an example of a concurrent linked list with poor code quality:

```
struct node {
  int key;
  struct node *next;
};

struct list {
  struct node *head;
  pthread_mutex_t lock;
};
```

Linked Lists (Poor Quality) (cont.)

```c
int insert(struct list *lst, int key) {
  pthread_mutex_lock(&lst->lock);
  struct node *new = malloc(sizeof(struct node));
  if (new == NULL) {
    perror("malloc");
    pthread_mutex_unlock(&lst->lock);
    return -1;   // unable to insert key
  }
  new->key = key;
  new->next = lst->head;
  lst->head = new;
  pthread_mutex_unlock(&lst->lock);
  return 0;   // key inserted
}
```

Linked Lists (Poor Quality) (cont.)

```
int lookup(struct list *lst, int key) {
  pthread_mutex_lock(&lst->lock);
  struct node *current = lst->head;
  while (current) {
    if(current->key == key) {
      pthread_mutex_unlock(&lst->lock);
      return 0;    // key found!
    }
    current = current->next;
  }
  pthread_mutex_unlock(&lst->lock);
  return -1;    // key not found!
}
```

Problem: Exceptional Control Flows

In the previous example, a lock was released at *different* locations, due to different / *exceptional control flow*:

```
pthread_mutex_lock(&lst->lock);
...
if (some condition) {
  pthread_mutex_unlock(&lst->lock);
  ...
}
pthread_mutex_unlock(&lst->lock);
```

⚠ **Such "unmatched" locking/unlocking is an important source of errors!**

Often, code can be *rearranged* such that lock/unlock only encompasses the critical section. Error-handling and other exceptional control flow (e.g. returning from functions) is handled outside.

Linked Lists (Better)

Here, `insert()` and `lookup()` are more cleanly written:

```
int insert(struct list *lst, int key) {
  // no synchronization required
  struct node *new = malloc(sizeof(struct node));
  if (new == NULL) {
    perror("malloc");
    return -1;   // unable to insert key
  }
  new->key = key;

  // only lock critical section
  pthread_mutex_lock(&lst->lock);
  new->next = lst->head;
  lst->head = new;
  pthread_mutex_unlock(&lst->lock);

  return 0;   // key inserted
}
```

Linked Lists (Better) (cont.)

```c
int lookup(struct list *lst, int key) {
  int retval = -1;

  pthread_mutex_lock(&lst->lock);
  struct node *current = lst->head;
  while (current) {
    if(current->key == key) {
      retval = 0;
      break;
    }
    current = current->next;
  }
  pthread_mutex_unlock(&lst->lock);
  return retval;
}
```

More Concurrency $\neq$ Faster

For the linked list, concurrency could be increased by having a lock per node and doing hand-over-hand locking. In practice, this seldom achieves more performance than locking the entire list. Thus:

- *Just because a data structure is more concurrent, it is not necessarily more efficient!*
- This is especially the case when it requires to acquire and release locks *frequently*.
- Always *start simple*, i.e. with a big lock, and then try to improve things, *only if required*.

"Premature optimization is the root of all evil." [Knu74]

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Condition Variables

Motivation

Locks enable mutual exclusion. But this is not the only form of synchronization required in practice. Often, a thread wants to *wait for a condition* to become true, e.g. for a child to complete some task. An option could be a shared variable:

```c
volatile int done = 0;

void *child(void *arg) {
  ...
  done = 1;
  return NULL;
}

void parent() {
  pthread_t c;
  assert(0 == pthread_create(&c, NULL, child, NULL));
  while (done == 0)
    ; // spin wait :-(
      // parent should sleep *until* child is ready!
  ...
}
```

Condition Variables With Pthreads

Using a condition variable, threads can put themselves on a queue and wait (sleep) until a condition is true. Another thread can then wake one or more waiting threads.

For pthreads, the type `pthread_cond_t` is a condition variable. There are a couple of functions for working with it, the most important ones are:

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_destroy(pthread_cond_t *cond);
```

Condition Variables With Pthreads (cont.)

As for mutexes, condition variables must also be *initialized*, either using `pthread_cond_init()` or the static initializer `PTHREAD_COND_INITIALIZER`. After use, they should be *destroyed* with `pthread_cond_destroy()`.

`pthread_cond_wait()` puts the thread to *sleep* and requires a locked mutex. The mutex is atomically released when the thread is put to sleep, before the function returns, the lock is reacquired.

`pthread_cond_signal()` *wakes a single* (indeterminate) thread, `pthread_cond_broadcast()` *wakes all* waiting threads.

☞ *It is good practice to also lock the mutex when signaling or broadcasting, however that is not strictly required.*

Example: Waiting for Child

```c
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
  pthread_mutex_lock(&m);
  done = 1;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}

void thr_join(){
  pthread_mutex_lock(&m);
  while (done == 0)
    pthread_cond_wait(&c, &m);
  pthread_mutex_unlock(&m);
}
```

Example: Waiting for Child (cont.)

```c
void *child(void *arg) {
  printf("child\n");
  thr_exit();
  return NULL;
}

int main(void) {
  printf("parent: begin\n");
  pthread_t chld;
  pthread_create(&chld, NULL, child, NULL);
  thr_join();
  printf("parent: end\n");
  return 0;
}
```

Why the Shared Variable?

When `pthread_cond_t` is a condition variable used for signaling, why do we need a separate, shared variable (e.g. “`done`”)?

Consider the following code:

```
void thr_exit() {
  pthread_mutex_lock(&m);
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}

void thr_join(){
  pthread_mutex_lock(&m);
  pthread_cond_wait(&c, &m);
  pthread_mutex_unlock(&m);
}
```

If the child runs first, there is *no thread to signal*. Later, the parent goes to sleep and will never be woken up.

Why the Mutex?

As the condition variable already provides some synchronization, why do we need a separate mutex?

Consider the following code:

```
void thr_exit() {
  done = 1;
  pthread_cond_signal(&c);
}

void thr_join(){
  while (done == 0)
    pthread_cond_wait(&c, &m);
}
```

There is a possible *race condition*:

1. Assume parent runs first and checks `done == 0` → true.
2. Parent is interrupted.
3. Child runs, sets `done = 1` and signals → no thread is waiting.
4. Parent continues, calls `pthread_cond_wait()` and goes to sleep forever!

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Producer/Consumer Problems

Introduction

A producer/consumer problem (or also bounded buffer problem) consists of 1…N *producer threads* putting items into the buffer, and 1…N *consumer threads* retrieving items from it.

For illustration, our buffer is just a single `int` variable. The producer stores increasing numbers in it and the consumer retrieves them. This happens endlessly:

```
void *producer(void *arg) {
  int i = 0;
  while (1) {
    put(i++);
  }
}

void *consumer(void *arg) {
  while (1) {
    int item = get();
    printf("%d\n", item);
  }
}
```

Introduction (cont.)

Here is an implementation of the buffer (`put()` and `get()`), checking if the buffer is full (or empty) before adding (removing) an item:

```
int buffer;
int full;

void put(int value) {
  assert(full == 0);
  full = 1;
  buffer = value;
}

int get() {
  assert(full == 1);
  full = 0;
  return buffer;
}
```

Clearly, the buffer is a *shared resource* and access to it must be synchronized.

Synchronization, First Attempt

```
pthread_cond_t cond;
pthread_mutex_t mutex;

void *producer(void *arg) {
  int i = 0;
  while (1) {
    pthread_mutex_lock(&mutex);
    if (full == 1)   // wait until buffer is empty
      pthread_cond_wait(&cond, &mutex);
    put(i++);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
}
```

Synchronization, First Attempt (cont.)

```
void *consumer(void *arg) {
  while (1) {
    pthread_mutex_lock(&mutex);
    if (full == 0)    // wait until buffer is full
      pthread_cond_wait(&cond, &mutex);
    int item = get();
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    printf("%d\n", item);
  }
}
```

Problem: More Threads

With only one producer and consumer, this code works. Problems arise with *more than one* producer/consumer, e.g. one producer (P) and two consumers (C0 and C1):

1. *C0 runs*, acquires the lock, finds the buffer empty and *waits* (releasing the lock).
2. *P runs*, acquires the lock, produces an item and *signals*.
   $\rightarrow$ *C0 is ready to run — but not yet run.*
3. *P continues* to loop, notices that the buffer is full and in turn *waits* itself (releasing the lock).
4. Now, *C1 runs* and acquires the lock, notices that there is an item available and **consumes** it, then sends the *signal* and releases the lock.
5. Finally, *C0 runs*, returns from `pthread_cond_wait()` (holding the lock) and **fails trying to consume** the item!

Synchronization, Second Attempt

Signaling wakes up a thread and is *only a hint* that something has changed. There is no guarantee that the state of the system then corresponds to the state when signaling took place!

The solution is thus to *re-check the state* of the system:

```
void *producer(void *arg) {
  int i = 0;
  while (1) {
    pthread_mutex_lock(&mutex);
    while (full == 1)   // wait until buffer is empty
      pthread_cond_wait(&cond, &mutex);
    ...
  }
}

void *consumer(void *arg) {
  while (1) {
    pthread_mutex_lock(&mutex);
    while (full == 0)   // wait until buffer is full
      pthread_cond_wait(&cond, &mutex);
    ...
  }
}
```

⚠ *With condition variables, always use `while` instead of `if` for checking! Additionally, this also fixes spurious wake-ups .*

Problem: Only One Condition Var.

There is a different issue with our second attempt, again with one producer (P) and two consumers (C0 and C1):

1. There is no item yet, *both C0 and C1 run* and then *wait*.
2. *P runs*, produces an item and *signals*.
   → *This wakes one of the two consumers, e.g. C0.*
3. *P continues* to loop, notices that the buffer is full and in turn also *waits*.
4. *C0 runs*, it now *rechecks* the condition (buffer is full), empties the buffer and *signals*.
   → *As the condition variable is used for both, consumers and producers, C1 is waken accidentally.*
5. *C1 runs*, *rechecks* the condition and sees that the buffer is empty. It then goes back to *wait*. **All threads are now effectively waiting forever!**

Final Solution

The solution here is to use *dedicated condition variables* for consumers and producers:

```
pthread_cond_t empty, filled;
pthread_mutex_t mutex;

void *producer(void *arg) {
  int i = 0;
  while (1) {
    pthread_mutex_lock(&mutex);
    while (full == 1)   // wait until buffer is empty
      pthread_cond_wait(&empty, &mutex);
    put(i++);
    pthread_cond_signal(&filled);
    pthread_mutex_unlock(&mutex);
  }
}
```

Final Solution (cont.)

```c
void *consumer(void *arg) {
  while (1) {
    pthread_mutex_lock(&mutex);
    while (full == 0)   // wait until buffer is full
      pthread_cond_wait(&filled, &mutex);
    int item = get();
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);
    printf("%d\n", item);
  }
}
```

Covering Conditions

Another situation can occur when it is *unclear which thread* to wake up. Given the following, hypothetical memory allocator:

```c
int freebytes = MAX_MEMORY;
pthread_cond_t c;
pthread_mutex_t m;

void *allocate(int size) {
  pthread_mutex_lock(&m);
  while (freebytes < size)
    pthread_cond_wait(&c, &m);
  void *ptr = ... // allocate memory
  freebytes -= size;
  pthread_mutex_unlock(&m)
  return ptr;
}
```

Covering Conditions (cont.)

```c
void free(void *ptr, int size) {
  pthread_mutex_lock(&m);
  bytesLeft += size;
  pthread_cond_signal(&c);    // but whom to signal?
  pthread_mutex_unlock(&m);
}
```

Covering Conditions (cont.)

Consider the following scenario:

1. `freebytes` is 0 (no memory free).
2. *Two threads request* memory: T0 wants 100 bytes, T1 wants 10 bytes.
   → *Both enter sleep on `pthread_cond_wait()`.*
3. Now, another thread, *T2 releases* 50 bytes.
4. *If T0* is woken up, it goes back to sleep, the system is *blocked*.

As there is *no way to control* which thread wakes up, the situation can only be solved using `pthread_cond_broadcast()` and waking *all threads*. This is a covering condition ("all cases are covered").

⚠ *This is inefficient. In general, if a program only works when changing signals to broadcasts, there is probably a design issue!*

Appendix

Bibliography

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 ed., Arpaci-Dusseau Books, August 2018, Available online: `http://ostep.org`.

[Dre04] Ulrich Drepper, *Futexes Are Tricky*.

[Knu74] Donald E. Knuth, *Structured programming with go to statements*, Computing Surveys 6 (1974), 261–301.