Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Operating Systems

Part 2: Concurrency – 6) Threads and Locks

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

BTI1341 / Fall 2022-23

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

Outline

Introduction

In computing, concurrency occurs whenever a *resource* could get *accessed concurrently* (at the same time) by more than one party:

- Two or more processes running on multiple cores (CPUs)
- Two or more threads running within a single process
- OS kernel threads

While often considered as an application programming issue, it makes sense to treat the topic in this course:

- Support from the OS is required for multi-threading in applications.
- The OS itself makes heavy usage of concurrency to implement scheduling, virtual memory management etc.
- The mechanisms for synchronization introduced in this part are required by the OS for handling timer interrupts.

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Threads

Threads as Further Abstraction

So far, we have introduced the abstraction of CPU and main memory (using processes and address spaces). A thread is another abstraction within a process:

> Definition
>
> A thread is an abstraction for the current *point of execution* (i.e. the program counter, PC/IP). It enables a process to run concurrently at different code locations. Every thread of a process shares the *same virtual address space*, *open file descriptors* and other things. However, it is running with its *own PC and registers*. Also, every thread has its *own stack* and may also have access to other data exclusively, i.e. to thread-local storage .

⚠ *If two threads run on a single core, a context switch is also required when switching from one to the other!*

Address Spaces

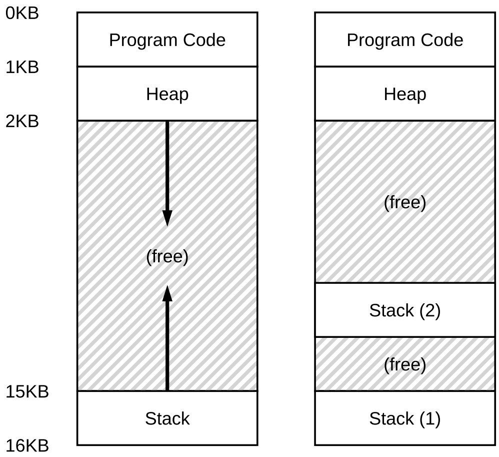

Figure: Single- and Multi-Threaded Address Spaces

Courtesy of [ADAD18], Own Modification

Why Use Threads?

To leverage the power of modern multi-core CPUs, multi-threaded applications are required. However, writing such applications tends to be more difficult. Before using threads, alternatives like multiprogramming and IPC should also be considered.

Here are some typical tasks, where using threads make sense:

- *Parallel computing*: Distributing computation on large data structures to multiple cores (benefit: shared address space). Example: matrix multiplication.
- *Blocking operations*, e.g. waiting for I/O. With threads, a process can do other work while waiting for completion.
- *Periodical work* can be scheduled by a process and performed in the background.

Thread Creation

There are different thread APIs. In this course, **POSIX threads (pthreads)** are used.[1] In pthreads, a thread is created using

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

The arguments are:

`thread`	Handle used for interacting with the thread.
`attr`	Attributes of the thread (e.g. stack size), or `NULL`.
`start_routine`	*Function pointer* to start thread.
`arg`	Arguments to pass to the function, or `NULL`.

On success, 0 is returned; otherwise an error number.

[1] See "`man pthreads`" and [ADAD18, Chapter 27] for more details.

Waiting for a Thread

Using `pthread_join()`, a thread can wait for another to complete:

```
int pthread_join(pthread_t thread,
                 void **retval);
```

The arguments are:

`thread`	Handle of the thread to wait for (obtained from `pthread_create()`).
`retval`	Location to store the result returned by the awaited thread.

On success, 0 is returned; otherwise an error number.

⚠ Pthreads Dangers and Pitfalls

While the POSIX API is quite straightforward, there are some things to watch out for. Most importantly:

- Thread creation is not a function call! Scheduling may take place and the order of execution is not guaranteed.
- Return codes of the `pthread_*` functions shall be checked![2]
- Do not pass data from the stack (e.g. references to automatic variables) to a thread!
- Never return thread-local data (e.g. variables on the thread's stack) from a thread!
- To compile a program using pthreads, it must be linked against the `pthread` library, use "`gcc -pthread`" for this.

[2] Always check the return code of any function!

Example: Thread Creation

```c
#include <assert.h>
#include <pthread.h>
#include <stdio.h>

void *mythread(void *arg) {
  printf("%s\n", (char *)arg);
  return NULL;
}

int main(void) {
  printf("main: begin\n");

  pthread_t t1, t2;

  assert(0 == pthread_create(&t1, NULL, mythread, "A"));
  assert(0 == pthread_create(&t2, NULL, mythread, "B"));

  // join waits for the threads to finish
  assert(0 == pthread_join(t1, NULL));
  assert(0 == pthread_join(t2, NULL));

  printf("main: end\n");
}
```

Example Execution Trace 1

main	t1	t2
runs		
“`main: begin`”		
creates t1		
creates t2		
waits for t1		
	runs	
	“`A`”	
	returns	
waits for t2		
		runs
		“`B`”
		returns
“`main: end`”		

Example Execution Trace 2

main	**t1**	**t2**
runs		
"`main: begin`"		
creates t1		
	runs	
	"`A`"	
	returns	
creates t2		
		runs
		"`B`"
		returns
waits for t1		
returns immediately		
waits for t2		
returns immediately		
"`main: end`"		

Example Execution Trace 3

main	**t1**	**t2**
runs		
“`main: begin`”		
creates t1		
creates t2		
		runs
		“`B`”
		returns
waits for t1		
	runs	
	“`A`”	
	returns	
waits for t2		
returns immediately		
“`main: end`”		

Example: Sharing Data

```c
#include <assert.h>
#include <pthread.h>
#include <stdio.h>

volatile int counter = 0;

void *mythread(void *arg) {
  printf("%s: begin\n", (char *)arg);
  for (int i = 0; i < 1e7; i++) {
    counter++;
  }
  printf("%s: done\n", (char *)arg);
  return NULL;
}

int main(void) {
  printf("main: begin (counter = %d)\n", counter);

  pthread_t t1, t2;

  assert(0 == pthread_create(&t1, NULL, mythread, "A"));
  assert(0 == pthread_create(&t2, NULL, mythread, "B"));

  // join waits for the threads to finish
  assert(0 == pthread_join(t1, NULL));
  assert(0 == pthread_join(t2, NULL));

  printf("main: done (counter = %d)\n", counter);
}
```

Example: Sharing Data (Output)

```
$ ./threads2
main: begin (counter = 0)
A: begin
B: begin
B: done
A: done
main: done (counter = 10977157)

$ ./threads2
main: begin (counter = 0)
A: begin
B: begin
B: done
A: done
main: done (counter = 11124703)

$ ./threads2
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done (counter = 11046906)
```

Problem: Uncontrolled Scheduling

To increment `counter`, the following instructions are executed:

```
1192:  mov     0x2eb4(%rip),%eax      # 404c <counter>
1198:  add     $0x1,%eax
119b:  mov     %eax,0x2eab(%rip)      # 404c <counter>
```

Here is what possibly happened:

1. `t1` is run and loads `counter` into `eax` (e.g. 50).
2. It adds one to the register, `eax` is now 51.
3. *A timer interrupt occurs!*
4. `t2` is scheduled by the OS.
5. `t2` executes and *completes* the same instructions, `counter` is now 51.
6. *Later*, `t1` resumes and also increments `counter` to 51!

Problem: Uncontrolled Scheduling (cont.)

OS	t1	t2	PC	EAX	counter
	before crit. sect.		1192	0	50
	`mov`		1198	**50**	50
	`add`		119b	**51**	50
Interrupt					
Restore t2			1192	0	50
		`mov`	1198	**50**	50
		`add`	119b	**51**	50
		`mov`	119f	51	**51**
Interrupt					
Restore t1			119b	51	51
	`mov`		119f	51	**51**

Critical Sections

The issue we have now identified with *shared data* in concurrent environments is a race condition , or in this case a data race . The outcome is *not deterministic* anymore (i.e. unpredictable) and depends on the timing of code execution. We call such a piece of code a critical section:

> Definition
> A critical section is a section of code which accesses a shared resource and may be prone to race conditions.

The property of mutual exclusion solves this type of issues; it guarantees that only a single thread at a time is executing code from a given critical section.

Synchronization Primitives and Atomicity

A possible solution for race conditions are atomic operations : Supported by the HW with a more powerful instruction, incrementing a counter in memory could be reduced to a *single* instruction.

Such instructions are not available for all kinds of critical sections, as implementing them in HW is too costly in general.

Instead, synchronization primitives will be used. They build on simple atomic operations in HW and some support from the OS to create powerful mechanisms for dealing with critical sections.

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Locks

Motivation

Most of the difficulties in writing concurrent application comes from critical sections and the lack of atomicity:

- Operations on shared data require more than one instruction.
- On multiple cores, more than one thread is possibly executing the same piece of code at the same time.
- Even on systems with only a single core, threads can be interleaved due to timer interrupts and scheduling.

The basic idea of a lock or locking mechanism is to ensure, that only a *single thread* is executing code *in a critical section* at any time.

Using locks, execution order can be controlled by the programmer, overriding otherwise uncontrollable scheduling by the OS.

Mutex: A Basic Lock

A mutual exclusion lock or mutex is basically just a variable holding the *state* of the lock:

- **available** (unlocked): no thread holds the lock.
- **acquired** (locked): *exactly one* thread holds the lock and is in a critical section (hopefully).

Internally, a lock variable may store additional data, e.g. which thread holds the lock etc.

Example:

```
lock_t mutex;
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Note: `lock()` *only returns after the lock has become available and has been locked by the thread requesting it!*

Coarse- and Fine-Grained Locking

Different locking strategies are possible: coarse-grained locking and fine-grained locking .

Consider the following example:

```
for (int i = 0; i < 1e7; i++) {
  lock(&mutex);
  counter++;
  unlock(&mutex);
}
```

Compared to:

```
lock(&mutex);
for (int i = 0; i < 1e7; i++) {
  counter++;
}
unlock(&mutex);
```

. . . which one is better?

Coarse- and Fine-Grained Locking (cont.)

Coarse-grained locking also occurs when the same lock is used for multiple critical sections. Thus, often more than one lock is used:

```
lock_t lock_s1, lock_s2;
...
lock(&lock_s1);
// critical section 1
unlock(&lock_s1);
...
lock(&lock_s2);
// critical section 2
unlock(&lock_s2);
...
```

In general, larger critical sections or multiple sections using the same lock tend to reduce concurrency.

Pthread Mutexes

The `pthread` library supports mutexes for locking:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

`mutex` is the lock variable, it must be *initialized* prior to usage:[3]

```
// Either static initializer:
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

// Or dynamically:
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

Finally, when a lock is not used anymore, it should be *destroyed:*

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

[3]The dynamic variant allows to configure further attributes of the lock. For default values, `pthread_mutex_init(&lock, NULL)` is the same.

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

HW Locking Primitives

Building and Evaluating Locks

We will now look at different ways a lock can be implemented with support from the HW. To evaluate locks, we will use the following criteria:

Correctness	Does the lock work correctly? Does it provide mutual exclusion?
Fairness	Does a thread waiting for the lock get a fair chance of obtaining it? Can a thread *starve*?
Performance	How much is the added overhead when using the lock? Especially, when a single thread runs the same code with- and without the lock?

1) Disabling Interrupts

An early approach to implement mutual exclusion was to simply disable interrupts for a moment:

```
void lock() {
  DisableInterrupts();
}

void unlock() {
  EnableInterrupts();
}
```

This approach is not practical (except for OS kernel code):

- Disabling and enabling interrupts uses privileged instructions.
- The OS loses control (no timer interrupt anymore!).
  - A process may lock and then use as much CPU as it likes.
  - If an endless loop occurs, the OS never regains control.
- Does not work with multiple CPUs (interrupts are per CPU).
- Interrupts may get lost if critical sections take too long.

2) Flag Variable

Idea: Use a simple variable to denote if a lock is available.

```c
// flag: 0 -> lock available, 1 -> lock acquired
struct lock {
  int flag;
};

void init(struct lock *lck) { lck->flag = 0; }

void lock(struct lock *lck) {
  while (lck->flag == 1)
    ; // spin wait!
  lck->flag = 1;
}

void unlock(struct lock *lck) { lck->flag = 0; }
```

Why Flag Variables Fail

The flag variable *is not correct*, it is itself prone to race conditions:

Thread 1	Thread 2
call `lock()`	
...`while (flag==1)`	
Interrupt, switch to Thread 2	
	call `lock()`
	...`while (flag==1)`
	...flag = 1;
	Interrupt, switch to Thread 1
...flag = 1;	

It is also *not fair*: there is no guarantee, that a thread will obtain a lock at a certain point in the future (depends on scheduling).

Finally, there is a *performance issue*: spin-waiting wastes scheduled time, possibly preventing the other thread to release the lock.

HW: Test-and-Set Instructions

The first HW instruction which enables constructing a lock is test-and-set or atomic exchange (x86: "`xchg`"). It atomically *returns* the old value at a given location and sets it to a new value.

In C, it would look as follows:

```
int TestAndSet(int *old_ptr, int new) {
  int old = *old_ptr;
  *old_ptr = new;
  return old;
}
```

3) A Valid Spin Lock

Using `TestAndSet`, the flag variable lock can be transformed in a valid spin lock :

```
// flag: 0 -> lock available, 1 -> lock acquired
struct lock {
  int flag;
};

void init(struct lock *lck) { lck->flag = 0; }

void lock(struct lock *lck) {
  while (TestAndSet(&lck->flag, 1) == 1)
    ; // spin wait!
}

void unlock(struct lock *lck) { lck->flag = 0; }
```

Evaluating Spin Locks

Spin locks *are correct* and provide mutual exclusion due to the atomicity of the `TestAndSet` instruction.

However, they are *not fair*: Depending on scheduling, a thread may simply spin forever, leading to starvation.

Performance depends on the system used:

- For *single core* systems, performance impacts may be severe:
  1. Assume the thread holding the lock is preempted.
  2. In the worst case, all other $N - 1$ threads are scheduled first, and each of them spins during its entire time slice...
- On *multi-core* systems, performance is reasonable:
  - If number of threads $\approx$ number of cores.
  - Assuming that threads are scheduled on different cores and critical sections are short.

HW: Compare-And-Swap

Another HW primitive is compare-and-swap or compare-and-exchange (x86: “`cmpxchg`”). It atomically *tests* a value at a given location and *replaces* it when it matches an expected value:

In C, it would look as follows:

```c
int CompareAndSwap(int *ptr, int expected, int new) {
  int current = *ptr;
  if (current == expected)
    *ptr = new;
  return current;
}
```

`CompareAndSwap` can also be used to implement a spin lock:

```c
void lock(struct lock *lck) {
  while (CompareAndSwap(&lck->flag, 0, 1) == 1)
    ; // spin wait!
}
```

HW: Load-Linked, Store-Conditional

A different approach is load-linked , combined with store-conditional (e.g. found on ARM). Here, two instructions work together atomically: the first only writes a given location iff it has not been modified since being read using the second.

In C, it would look as follows:

```
int LoadLinked(int *ptr) {
  return *ptr;
}

int StoreConditional(int *ptr, int value) {
  if (UNMODIFIED) {
    // *ptr has not been modified since last LoadLinked()
    *ptr = value;
    return 1; // success
  } else {
    return 0; // failure
  }
}
```

HW: Load-Linked, Store-Conditional (cont.)

A spin lock can also be implemented using LoadLinked and StoreConditional:

```
void lock(struct lock *lck) {
  while(true) {
    while (LoadLinked(&lck->flag) == 1)
      ; // spin wait!
    if (StoreConditional(&lck->flag, 1) == 1)
      return;
  }
}
```

HW: Fetch-And-Add

Finally, the HW instruction fetch-and-add offers interesting options. It atomically *increments* a location and returns the old value.

In C, it would look as follows:

```c
int FetchAndAdd(int *ptr) {
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

4) Ticket Locks

With `FetchAndAdd`, ticket locks , a different kind of lock, can be implemented:

```
struct lock {
  int ticket;
  int turn;
};

void init(struct lock *lck) {
  lck->ticket = 0;
  lck->turn = 0;
}

void lock(struct lock *lck) {
  int myturn = FetchAndAdd(&lck->ticket);
  while (lck->turn != myturn)
    ; // spin wait!
}

void unlock(struct lock *lck) { lck->turn++; }
```

Evaluating Ticket Locks

Ticket locks *are correct* and provide mutual exclusion, also due to the atomicity of the HW instruction used.

Compared to spin locks, ticket locks *are fair*: The ticket mechanism ensures that a thread will be scheduled at some point in the future (assuming that all prior threads properly release the lock).

Regarding performance however, things have not changed: ticket locks are a form of spin locks and thus have the same issues.

Comparing Locks

To conclude, the following table gives a brief overview of the lock types introduced so far and how they match the criteria defined on Slide 28:

Lock Type	Correctness	Fairness	Performance
1) Disabled Interrupts[4]	✓	✗	✗
2) Flag Variables	✗	✗	✗
3) Spin Locks	✓	✗	✗[5]
4) Ticket Locks	✓	✓	✗[5]

As can be seen, none of the locks has reasonable performance. In the next part, we will investigate further locking concepts, which help to solve this problem.

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

[5] Reasonable on multi-core systems only.

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`.