Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Operating Systems

Part 2: Concurrency – 8) Semaphores, Common Problems

Revision: `master@323240b (20230907-115823)`

BTI1341 / Fall 2023/24

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

Outline

Semaphores

Reader-Writer Locks

Typical Problems: Dining Philosophers

Typical Problems: Deadlocks

Other Typical Problems

Appendix

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Semaphores

Introduction

We have seen that two different kinds of synchronization primitives are required for solving typical concurrency issues:

- Locks (mutexes)
  Protect *critical sections*, ensuring that only a single thread at a time is executing code in it (*mutual exclusion* property).
- Condition variables
  Used when *signaling* between threads or *ordering* (thread scheduling) is required.

A more generic primitive, the semaphore , can be used for both.

Semaphores

Definition

A semaphore ([Dij68]) is an object (a “variable”) with an *integer value*. It is typically manipulated using two *atomic* operations:

- **Increment**: `sem_post()` or `V()`
  Wakes up a *single* thread if there is at least one waiting.
- **Decrement**: `sem_wait()` or `P()`
  Blocks the calling thread if the semaphore becomes negative.

The function/behavior of a semaphore is defined by its *initial value*.

Note: Practical semaphores do not really become negative. It may however help to think of negative values as the number of threads waiting.

Semaphores with Pthreads

For Pthreads, the relevant functions are defined in `semaphore.h`, the most important are:[1]

```
int sem_init(sem_t *sem, int pshared,
             unsigned int value);
```

Initializes a semaphore to a given `value`. `pshared` is set to 0, except if the semaphore is to be shared between multiple processes.

```
int sem_post(sem_t *sem);
```

Increments the semaphore, unblocks a waiting thread.

```
int sem_wait(sem_t *sem);
```

Decrements the semaphore, possibly blocking the caller.

[1] See "`man 7 sem_overview`" for further details.

Binary Semaphores

Semaphores can be used in place of locks/mutexes. For this, the initial value has to be *set to 1*. Such a semaphore is called a binary semaphore .

```
sem_t sem;
sem_init(&sem, 0, 1);   // <-- init: 1

sem_wait(&sem);

// critical section

sem_post(&sem);
```

Binary Semaphore: 1 Thread

Sem.	Thread 0	State
1	…	running
1	call `sem_wait()`	running
0	`sem_wait()` returns immediately	running
0	*critical section…*	running
0	call `sem_post()`	running
1	`sem_post()` returns	running
1	…	running

Binary Semaphore: 2 Threads

Sem.	Thread 0	State	Thread 1	State
1	…	running		ready
1	call `sem_wait()`	running		ready
0	`sem_wait()` returns	running		ready
0	*critical section…*	running		ready
0	*interrupt, switch to T1*	ready	…	running
0		ready	call `sem_wait()`	running
-1		ready	`sem_wait()` blocks	sleeping
-1		ready	*switch to T0*	sleeping
-1	*critical section…*	running		sleeping
-1	call `sem_post()`	running		sleeping
0	`sem_post()` returns	running		ready
0	*interrupt, switch to T1*	ready		running
0		ready	`sem_wait()` returns	running
0		ready	*critical section…*	running
0		ready	call `sem_post()`	running
1		ready	`sem_post()` returns	running
1		ready	…	running

Semaphores for Ordering

Semaphores can also be used in place of condition variables, e.g. to have a thread wait for another thread.

For this, the initial value has to be *set to 0*. The first call to `sem_wait()` then immediately blocks and another thread can signal by calling `sem_post()`. There are two possible execution flows:

- The thread calling *`sem_wait()` is run first*.
  The semaphore is decremented to -1 and the thread is blocked.
- The thread calling *`sem_post()` is run first*.
  The semaphore is incremented to 1, the other thread can continue to run when calling `sem_wait()`.

In both cases, the waiting thread will only *run after* the first thread has invoked `sem_post()`!

Ordering Example

Example of using a semaphore instead of `pthread_join()`:

```
sem_t sem;

void *child(void *arg) {
  printf("child\n");
  sem_post(&sem);
  return NULL;
}

int main(void) {
  sem_init(&sem, 0, 0);   // <-- init: 0
  printf("parent: begin\n");
  pthread_t thread;
  pthread_create(&thread, NULL, child, NULL);
  sem_wait(&sem);
  printf("parent: end\n");
}
```

Producer/Consumer Problem

Let's revisit the producer/consumer problem from last time. This time, the *buffer may hold up to `MAX` elements* and there can be multiple concurrent producers and consumers.

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
  buffer[fill] = value;
  fill = (fill + 1) % MAX; // ring buffer
}

int get() {
  int rv = buffer[use];
  use = (use + 1) % MAX;
  return rv;
}
```

Producer/Consumer Problem (cont.)

Producer and consumer threads:

```
void *producer(void *arg) {
  int i = 0;
  while (1) {
    sem_wait(&empty);
    put(i++);
    sem_post(&full);
  }
}

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

Producer/Consumer Problem (cont.)

The `main()` function, which initializes the semaphores:

```
sem_t empty, full;

int main(void) {
  // ...
  sem_init(&empty, 0, MAX);
  sem_init(&full, 0, 0);
  // ...
}
```

Problem: Race Condition

The code given above only works correctly when the buffer just stores a single element (`MAX = 1`). In this case, the semaphores provide mutual exclusion implicitly. Otherwise, there are two race conditions:

```
buffer[fill] = value;
fill = (fill + 1)
```

and:

```
int rv = buffer[use];
use = (use + 1) % MAX;
```

These two blocks (specifically the variables `fill` and `use`) are critical sections and need to be protected.

Solution (Wrong): Add a Mutex

```c
sem_t mutex;

void *producer(void *arg) {
  int i = 0;
  while (1) {
    sem_wait(&mutex)   // <--
    sem_wait(&empty);
    put(i++);
    sem_post(&full);
    sem_post(&mutex)   // <--
  }
}

void *consumer(void *arg) {
  while (1) {
    sem_wait(&mutex)   // <--
    sem_wait(&full);
    int item = get();
    sem_post(&empty);
    sem_post(&mutex)   // <--
    printf("%d\n", item);
  }
}
```

Next Issue: Deadlock

This first solution for the race condition may introduce a new problem: a deadlock! Here is what could happen with two threads:

1. Consumer *runs first* and acquires `mutex`.
2. It then calls `sem_wait()` on `full`.
   - `full` is 0 at this point (no items produced yet).
   - Consumer blocks, `mutex` is *not unlocked*!
3. Now, producer runs:
   - Tries to acquire `mutex` (held by consumer).
   - Producer *blocks*!

Although the producer could wake the consumer, both threads will effectively be *blocked forever*!

Solution: Lock Ordering

In this case, the solution is simple: change the *lock order*.

```
void *producer(void *arg) {
  int i = 0;
  while (1) {
    sem_wait(&empty);
    sem_wait(&mutex)   // <--
    put(i++);
    sem_post(&mutex)   // <--
    sem_post(&full);
  }
}

void *consumer(void *arg) {
  while (1) {
    sem_wait(&full);
    sem_wait(&mutex)   // <--
    int item = get();
    sem_post(&mutex)   // <--
    sem_post(&empty);
    printf("%d\n", item);
  }
}
```

When to Use Semaphores?

Semaphores look like a generalization of locks and condition variables. Which one should be used?

- It is in general possible to replace locks and condition variables with semaphores.
- However, replacing specific constructions with generalizations is not always the best thing to do.[2]
- Consider semantics and *expressiveness*:
  - A mutex says probably more than a semaphore initialized to 1.
  - For synchronizing access to limited quantities (e.g. for bounded buffers), semaphores provide natural semantics (they "count" the number of available items).
- In the end, this is also a question of personal taste...

[2] E.g. it has been shown that building correct condition variables using semaphores is difficult (see [Bir]).

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Reader-Writer Locks

Introducing R-W Locks

A different lock type not treated so far are reader-writer locks: Often, *concurrent reading* of a data structure is possible, as long as *no thread is writing* at the same time. The basic idea is simple:

- A *write lock* is acquired when writing to the structure *or* by the first reading thread (thus excluding any writers).
- An (unlimited) number of additional threads may still obtain a *read lock*.
- No thread can write to the structure as long as the write lock is held.

→ From this follows, that every thread wishing to write must first *wait for all* reading threads to release the lock!

R-W Locks with Semaphores

Example: implementation of an R-W lock using semaphores.

Note: *Fairness is not ensured*; readers may easily starve writers! How could this be fixed?

```
struct rwlock {
  sem_t lock;      // mutex for writelock manipulation
  sem_t writelock; // mutex for writing access
  int readers;     // current number of readers
};

void rwlock_init(struct rwlock *lck) {
  lck->readers = 0;
  sem_init(&lck->lock, 0, 1);
  sem_init(&lck->writelock, 0, 1);
}
```

R-W Locks with Semaphores (cont.)

```
void rwlock_acquire_readlock(struct rwlock *lck) {
  sem_wait(&lck->lock);
  lck->readers++;
  if (lck->readers == 1) // first reader acquires writelock
    sem_wait(&lck->writelock);
  sem_post(&lck->lock);
}

void rwlock_release_readlock(struct rwlock *lck) {
  sem_wait(&lck->lock);
  lck->readers--;
  if (lck->readers == 0) // last reader releases writelock
    sem_post(&lck->writelock);
  sem_post(&lck->lock);
}

void rwlock_acquire_writelock(struct rwlock *lck) {
  sem_wait(&lck->writelock); }

void rwlock_release_writelock(struct rwlock *lck) {
  sem_post(&lck->writelock); }
```

R-W Locks With Pthreads

Pthreads also offers support for reader-writer locks with an API similar to the other lock types.

To initialize/destroy a R-W lock:

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t
                              *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

To acquire a read lock:

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

To acquire a write lock:

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

For unlocking both types of locks:

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Typical Problems: Dining Philosophers

Introduction

For modeling systems in which there is competition for exclusive access to a limited set of resources, the classical problem of the dining philosophers ([Dij71]) is often used:

- A given number of philosophers sits at a round table.
- There is the same amount of forks on the table.
- The philosophers act in loops: either they think or they would like to eat.
- In order to eat, a philosopher needs exactly two forks.

Introduction (cont.)

An example of the dining philosopher's problem with 5 philosophers/forks:

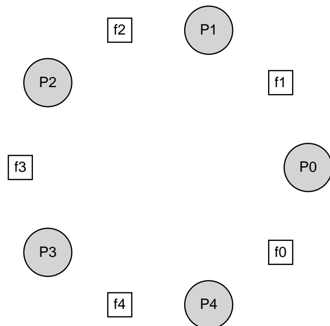

Figure: The Dining Philosophers

Courtesy of [ADAD18]

Introduction (cont.)

The basic per-philosopher algorithm would be as follows:

```
while (1) {
  think();
  getforks();
  eat();
  putforks();
}
```

Without precautions, a *race condition* may occur in `getforks()` and/or `putforks()`, when the philosophers try to claim two forks for eating.

Solution: First Attempt

The basic idea is to have *one semaphore per fork* in order to ensure mutual exclusion when it is accessed: `sem_t forks[5]`.[3]

```
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}

void putforks() {
  sem_post(forks[left(p)]);
  sem_post(forks[right(p)]);
}
```

However, this naive implementation is prone to *deadlocks*!

[3] In this example code, forks and philosophers are numbered and two functions `int left(int p)` and `int right(int p)` are given, which return the number of the fork left (or right) of the given philosopher `p`.

Solution: Second Attempt

The problem with the first solution occurs when *all philosophers* are interrupted after taking the left fork, but before being able to take the right fork (or vice versa).

The simplest solution (also proposed by Dijkstra) is to break the dependency between the philosophers by *changing the order* of acquisition for at least one philosopher:

```
void getforks() {
  if (p == 0) {   // which p does not matter
    sem_wait(forks[right(p)]);
    sem_wait(forks[left(p)]);
  } else {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
  }
}
```

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Typical Problems: Deadlocks

Introduction

We have seen that deadlocks are a frequent problem in concurrent programming (e.g. on slides 17 and 29). There are many situations potentially leading to them, here is an example:

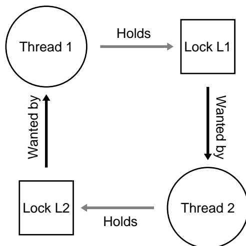

Figure: Deadlock with Two Threads

Courtesy of [ADAD18]

```
// Thread 1
pthread_mutex_lock(L1)
pthread_mutex_lock(L2)

// Thread 2
pthread_mutex_lock(L2)
pthread_mutex_lock(L1)
```

Reasons and Conditions

Such deadlocks seem easily preventable, however in practice this is more difficult due to *code size/complexity* and *encapsulation*. These can hardly be avoided in general.

Already early, *four necessary conditions* for deadlocks could be identified:[4]

Mutual exclusion	Threads need to claim exclusive control over a resource.
Hold-and-wait	Threads hold on to resources while waiting for additional resources.
No preemption	Resources cannot be forcibly reclaimed from threads.
Circular wait	There is a circular chain of threads holding and requesting resources.

[4] See [CES71].

Preventing Circular Wait

The *most practical* solution is to avoid circular waiting in code:

- If possible, have a *total ordering* on locking and always acquire all locks in the same order.
- Else, have a partial ordering depending on the lock, be sure to *document it*!

Example from [lin]:

```
/*
 * Lock ordering:
 *
 *  -> i_mmap_rwsem            (truncate_pagecache)
 *    -> private_lock          (__free_pte->__set_page_dirty_buffers)
 *      -> swap_lock           (exclusive_swap_page, others)
 *        -> i_pages lock
 *
 *  -> i_mutex
 *    -> i_mmap_rwsem          (truncate->unmap_mapping_range)
 *
 *  -> mmap_sem
 *    -> i_mmap_rwsem
 *      -> page_table_lock or pte_lock (various, mainly in memory.c)
 *        -> i_pages lock      (arch-dependent flush_dcache_mmap_lock)

...
```

Idea: Lock Ordering by Address

A possibly clever idea is the use of the effective *memory address* of the locks to enforce lock ordering:

```
if (L1 > L2) { // pointer comparison!
  pthread_mutex_lock(L1)
  pthread_mutex_lock(L2)
} else {
  pthread_mutex_lock(L2)
  pthread_mutex_lock(L1)
}
```

Preventing Hold-and-Wait

Hold-and-wait can be avoided by acquiring all locks *at once*, atomically:

```
pthread_mutex_lock(atomic)
pthread_mutex_lock(L1)
pthread_mutex_lock(L1)
// ...
pthread_mutex_unlock(atomic)
```

There are some possible issues with that approach:

- It must be exactly known which locks are to be acquired.
  $\rightarrow$ Typically difficult to achieve in complex code bases.
- Has a negative impact on concurrency in general.

Implementing Preemption

There is no preemption with locks, i.e. “taking away” a lock from a thread is not possible.

An option might be using `pthread_mutex_trylock()`, which returns an error instead of blocking when a lock is already held, e.g:

```
while (1) {
  pthread_mutex_lock(L1);
  if (0 != pthread_mutex_trylock(L2)) {
    pthread_mutex_unlock(L1);
  } else {
    break;
  }
}
```

Possible issues:

- No real preemption (other thread continues to run).
- A livelock may occur (all threads run, none manages to acquire both locks).
- Also difficult to implement in complex code bases.

Preventing Mutual Exclusion

Will be discussed in 09-concurrency-4!

Detect and Recover

A final strategy might also be to *detect and recover*. This is an option if:

- Deadlocks rarely occur
- Fixing them is very difficult

For this, some kind of (separately running) *deadlock detector* is required, which may e.g restart the application if a deadlock occurs. Such approaches are sometimes used by databases and other distributed systems.

Clearly, this is not a real solution but might be a proper and *pragmatic* engineering workaround.

Other Typical Problems

Atomicity Violations

Atomicity violations occur when some part of code is assumed to be *running atomically* as a whole, while in reality it might get preempted:

```
// Thread 0
...
if (fileptr) {   // <-- assumed to be run together...
  ...
  fputs("data", fileptr);   // <-- ...with this
  ...
}

// Thread 1
...
fclose(fileptr);
...
```

Preventing Atomicity Violations

Often, atomicity violations can easily be solved by *proper locking*:

```
pthread_mutex_t fileptr_lock = PTHREAD_MUTEX_INITIALIZER;

// Thread 0
...
pthread_mutex_lock(&fileptr_lock);
if (fileptr) {
  ...
  fputs("data", fileptr);
  ...
}
pthread_mutex_unlock(&fileptr_lock);

// Thread 1
...
pthread_mutex_lock(&fileptr_lock);
fclose(fileptr);
pthread_mutex_unlock(&fileptr_lock);
...
```

Order Violation Bugs

Order violation bugs occur when a certain *order of execution* between threads is assumed:

```
struct my_resource rs;

// Thread 0
...
init_resource(&rs);   // <-- assumed to be run before...
...

// Thread 1
...
print_resource(&rs);  // <-- ... this
...
```

Preventing Order Violation Bugs

The solution for order violation bugs in general is to *enforce ordering* using condition variables or semaphores:

```
pthread_mutex_t rs_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t rs_cond = PTHREAD_COND_INITIALIZER;
int rs_init = 0;

struct my_resource rs;

// Thread 0
...
init_resource(&rs);

pthread_mutex_lock(&rs_mutex);
rs_init = 1;
pthread_cond_signal(&rs_cond);
pthread_mutex_unlock(&rs_mutex);
...

// Thread 1
...
pthread_mutex_lock(&rs_mutex);
while (rs_init == 0)
  pthread_cond_wait(&rs_cond, &rs_mutex);
pthread_mutex_unlock(&rs_mutex);

print_resource(&rs);
...
```

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

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`.

[Bir] Andrew Birrell, *Implementing Condition Variables with Semaphores*, `https://www.microsoft.com/en-us/research/wp-content/uploads/2004/12/ImplementingCVs.pdf`.

[CES71] E. G. Coffman, M. J. Elphick, and A. Shoshani, *System Deadlocks*, Computing Surveys **3** (1971), 67–78.

[Dij68] Edsger W. Dijkstra, *Cooperating sequential processes*, 1968, `http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF`.

[Dij71] ______, *Hierarchical ordering of sequential processes*, Acta Informatica **1** (1971), 115–138.

[lin] *Linux kernel, filemap.c*, `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/mm/filemap.c`.