Lecture 13 — Semaphores

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 4, 2018

Mutual Exclusion through Messages

The earlier definition of mutual exclusion was informal.

There are additional desirable properties that will be used to evaluate any solution:

- Mutual exclusion must apply.
- A thread that halts outside the critical section must not interfere with other threads.
- It must not be possible for a thread requiring access to a critical section to be delayed indefinitely.
- When no thread is in the critical section, a thread that requests access should be allowed to enter right away.
- No assumptions are made about what the threads will do or the number of processors in the system.
- A thread remains inside the critical section for a finite time only.

Back to Springfield

Recall from earlier the example of the employees Alice and Bob who worked at the Springfield Nuclear Power Plant in Sector 7G.

Suppose there is a third employee at the power plant, Charlie, who works on the day shift at the same time as Alice.

Safety rules say that at least one of them has to monitor the safety of the reactor at all times and therefore they cannot both take lunch at the same time.

If we cannot predict when lunch begins or how long it will last, how can Alice and Charlie co-ordinate to make sure they don't take lunch at the same time?

At the Power Plant

Before Alice gets up from her desk to go for lunch, she calls Charlie.

If he does answer, she may proceed.

If Charlie does not answer, Alice will know he is not at his desk.
Therefore she cannot leave at the moment.

She can call again, constantly, until she reaches Charlie (busy-waiting), but this ties up a phone line nonstop and is effort intensive for Alice.

If she doesn't want to do that, at this point she has two options:
Wait some period of time (perhaps 15 minutes) and call again. Or
Leave a message in Charlie's voice mail box, asking him to call her back.

Then Alice can go about her work until she gets a call from Charlie and as soon as that happens, she may step out for lunch.

Power Plant to Computer World

Busy waiting has already been found inadequate as a solution.

It wastes CPU time that another thread could be putting to productive use.

The approach of “wait 15 minutes and try again” might be adequate for Alice as a human, but for the computer it is not ideal.

If *A* fails to get in, then sleeps for 2000 ms before trying again, if *B* is finished after 20 ms, then thread *A* waits unnecessarily for 1980 ms.

What we want is something that resembles the call-when-finished semantics of Alice leaving a message and Charlie calling her back.

Semaphore

A semaphore is a system of signals used for communication.

Before ships had radios, when two friendly ships were in visual range, they would communicate with one another through flag semaphores.

Each ship had someone holding certain flags in a specific position.

Thus the two ships could co-ordinate at (visual range) distance.

This worked dramatically better than many alternatives (e.g., shouting).

The Computer Semaphore

The computer semaphore was invented in 1965 by Edsger Dijkstra.

He described a data structure that can be used to solve synchronization problems via messages.

Although the version we use now is not exactly the same as the original description, even 50 years later, the core idea is unchanged.

Binary Semaphore

The binary semaphore: this is a variable that has two values, 0 and 1.

It can be initialized to 0 or to 1.

The semaphore has two operations: `wait` and `signal`.

In the original paper, `wait` was called `P` and `signal` was called `V`, but the names in common usage have become a little more descriptive.

Note: `signal` is also called `post` in many textbooks.

Binary Semaphore: wait

The `wait` operation on the semaphore is how a program tries to enter the critical section.

When `wait` is called, if the semaphore value is 1, set it to 0 and this thread may enter the critical section and continue.

If the semaphore is 0, some other thread is in the critical section and the current thread must *wait* its turn.

The thread that called `wait` will be blocked by the operating system, just as if it asked for memory or a disk operation.

This is sometimes referred to as decrementing the semaphore (because the value changes from 1 to 0).

Binary Semaphore: signal

The `signal` operation is how a program *signals*, or indicates, it is finished with the critical section.

When this is called, if the semaphore is 1, do nothing.

If the semaphore is 0 and there is a task blocked awaiting that semaphore, that task may be unblocked.

Otherwise, set the semaphore to 1.

This is also sometimes called incrementing the semaphore.

Caffeine - the Stuff of Life?

Analogy: you like coffee, and going to a particular coffee shop because there you can get your drink exactly the way you like it.

“Half caf, no whip, extra hot, extra foam, two shot, soy milk latte.”

After this beverage it may be the case that you need to use the washroom.

The washroom is locked at such places, so to get in you will need the key, which is available by asking one of the employees.

Programmers Convert Caffeine into Code

If nobody is currently in the washroom, you will get the key and can proceed.

If it is currently occupied, you will have to wait.

When the key is returned, if anyone is waiting, the employee will give the key to the first person in line for the washroom.

Otherwise he or she will put the key away behind the counter.

Semaphores: OS Support

Observe that the operating system is needed to make this work.

If thread *A* attempts to wait on a semaphore that some other thread already has, it will be blocked.

The operating system knows not to schedule it to run until it is unblocked.

When thread *B* is finished and signals the semaphore it is holding, that will unblock *A* and allow it to run again.

No Checking

The semaphore does not provide any facility to “check”the current value.

A thread doesn’t know in advance if it will block when it waits on a semaphore.

It can only give it a shot. Either it will be blocked or proceed directly.

Signalling

When a thread signals a semaphore, it likewise does not know if any other thread(s) are waiting on that semaphore.

There is no facility to check this, either.

When thread *A* signals a semaphore, we don't know what thread will continue execution.

Semaphore: Bad Behaviour

Nothing in the semaphore as defined protects against certain bad behaviour.

Suppose thread *C* would like to enter the critical section.

The programmer of this task is malicious as well as impatient: “my task is FAR too important to wait for those other processes and threads,” he says.

He implements his code: before he waits on the semaphore, he signals it.

Even though *A* or *B* might be in the critical section, the semaphore is signalled.

So he is fairly certain that his program will now get to enter the critical section.

Semaphore: Bad Behaviour

It's not foolproof: if there are other threads waiting, they might get woken up to proceed instead of *C*; much depends on the scheduler.

Nevertheless, this is really bad: one process can wreak all kinds of havoc by letting another process into the critical section.

Though the example here makes the author of thread *C* a scheming villain, such a situation may occur if it is simply the result of a programming error.

Preventing Bad Behaviour

The problem identified is usually solved by supplementing the basic binary semaphore.

A data structure called a mutex is a binary semaphore with an additional rule enforced: only the thread that has called `wait` may `signal` that semaphore.

This adds a small amount of extra bookkeeping to the semaphore, but this is a reasonable price to pay.

Semaphore Example: Linked List Integrity

```
typedef struct single_node {
  void *element;
  struct single_node *next;
} single_node_t;

typedef struct single_list {
  single_node_t *head;
  single_node_t *tail;
  int size;
} single_list_t;


void single_list_init( single_list_t *list ) {
  list->head = NULL;
  list->tail = NULL;
  list->size = 0;
}
```

Semaphore Example: Linked List Integrity

```
bool push_front( single_list_t *list, void *obj ) {
  single_node_t *tmp = malloc( sizeof( single_node_t ) );

  if ( tmp == NULL ) {
    return false;
  }

  tmp->element = obj;
  tmp->next = list->head;
  list->head = tmp;

  if ( list->size == 0 ) {
     list->tail = tmp;
  }

   ++( list->size );
   return true;
}
```

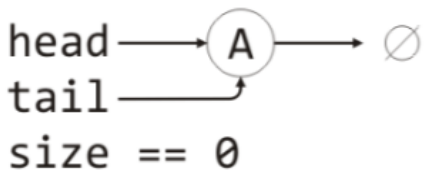
Using the Linked List

If only one thread access this data structure, we do not have a problem.

Suppose a thread tries to add an element *A* to the list using `push_front`.

Right before the increment of the `size` field there is a process switch.

At this point, the new node has been allocated and initialized, the pointers of `head` and `tail` have been updated, but `size` is 0.

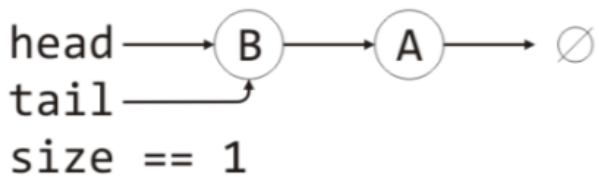


Using the Linked List

Now, the second thread executes and wants to add *B* to the linked list.

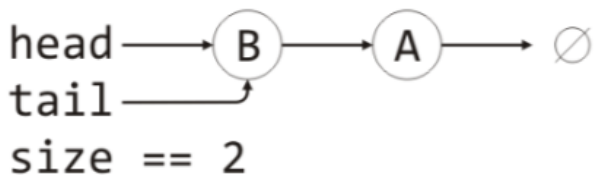
In the conditional statement, `list->size == 0` evaluates to true.

Thus, the `tail` pointer is updated.



When the first thread gets to run again, it will resume where it left off.

It increments the `size` integer, leaving the final state: `head` and `tail` both point to element *B*, even though there is element *A* in the list.



Inconsistent State

This is an inconsistent state.

The linked list has two elements in it but the `tail` pointer is wrong.

An attempt to remove an element from the list will reveal the problem.

Remove the front element? Check if `head` and `tail` are equal. That may give the mistaken impression that *B* is the last element in the list. We lost *A*!

Or the `head` pointer will be updated but `tail` will still point to *B* even after it has been freed, which can result in a segmentation fault or invalid access.

Semaphore Syntax

The linked list problem just discussed can be solved easily if a semaphore is employed.

Binary semaphores are useful, and we can generalize this concept to what is known as a *counting* or *general* semaphore.

General Semaphores

If a thread attempts to wait on a semaphore and the decrement operation makes the integer value negative, the calling thread is blocked.

If initialized with 5 and the current value is 2, a thread that waits on that general semaphore will not be blocked.

In some operating systems there is no distinction between binary and counting semaphores (create a general semaphore with a value of 1).

Semaphore Syntax

The syntax we will use is as follows:

```
sem_init( sem_t* semaphore, int shared, int initial_value);
sem_destroy( sem_t* semaphore )
sem_wait( sem_t* semaphore )
sem_post( sem_t* semaphore )
```

Applying the Semaphore to the Linked List

```
typedef struct single_node {
  void *element;
  struct single_node *next;
} single_node_t;

typedef struct single_list {
  single_node_t *head;
  single_node_t *tail;
  int size;
  sem_t sem;
} single_list_t;

void single_list_init( single_list_t *list ) {
  list->head = NULL;
  list->tail = NULL;
  list->size = 0;

  sem_init( &( list->sem ), 0, 1 );
}
```

Applying the Semaphore to the Linked List

```
bool push_front( single_list_t *list, void *obj ) {
  single_node_t *tmp = malloc( sizeof( single_node_t ) );

  if ( tmp == NULL ) { return false; }
  tmp->element = obj;

  sem_wait( &( list->sem ) ); {
    tmp->next = list->head;
    list->head = tmp;

    if ( list->size == 0 ) {
       list->tail = tmp;
    }
    ++( list->size );

  } sem_post( &( list->sem ) );
  return true;
}
```

Applying the Semaphore to the Linked List

The critical section here just encloses the modification of the shared linked list.

In theory one might put the wait and signal operations at the start and end of the entire function, respectively.

This is, however, suboptimal: it forces unnecessary waiting.

Including the call to `malloc` is especially bad; the memory allocation itself can block if insufficient memory is available.