

Operating systems and concurrency B06

David Kendall

Northumbria University

- Concurrent tasks that share resources can interfere with each other
- Interference can lead to incorrect behaviour
- Interference can be avoided by identifying **critical sections** and enforcing **mutual exclusion**
- Mutual exclusion protocols considered so far involve **busy waiting**
- Busy waiting is bad
- This lecture is about how to enforce mutual exclusion **without busy waiting**

Semaphore definition

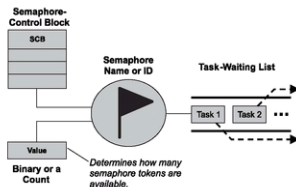
A **semaphore** is a kernel object that one or more tasks can acquire or release for the purposes of synchronisation or mutual exclusion.

- Binary semaphore proposed by Edsger Dijkstra in 1965 as a mechanism for controlling access to critical sections
- Two operations on semaphores:
 - **acquire** (aka: pend, wait, take, P)
 - **release** (aka: post, signal, put, V)

Semaphore operations

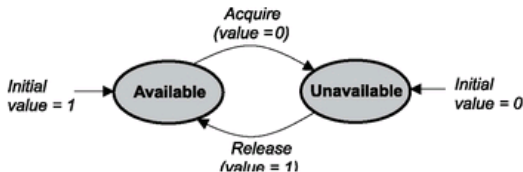
- Semaphore value initially 1
- Task calling `acquire(s)` when `s == 1` acquires the semaphore and `s` becomes 0
- Task calling `acquire(s)` when `s == 0` is **suspended**
- Task calling `release(s)` makes ready a previously suspended task if there are any
- Task calling `release(s)` restores value of `s` to 1 if there are no suspended tasks

Counting semaphores (Carel Scholten)

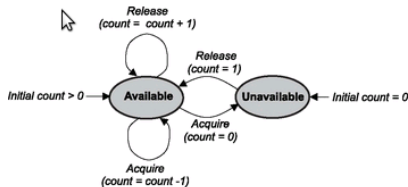


- Idea of binary semaphore can be generalised to **counting** semaphore (car park example)
- Each `acquire(s)` decreases value of `s` by 1 down to 0
- Each `release(s)` increases value of `s` by 1 up to some maximum
- Task waiting list used for tasks waiting on unavailable semaphore
- Waiting list may be FIFO or priority-ordered or ...
 - ... implementation dependent (important to know what your particular implementation does here)

Semaphore state diagrams



Binary semaphore



Counting semaphore

POSIX semaphores: Create

- Must **create** a semaphore before using it

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

- `sem` is a pointer to a semaphore variable
- if `pshared` is 0 then the semaphore is to be shared by threads of the same process, if `pshared` is non-zero then the semaphore is to be shared between processes
- `value` specifies the initial value of the semaphore
- `sem_init` creates an unnamed semaphore
- It returns 0 on success and -1 on error

- Example

```
#include <semaphore.h>
```

```
sem_t sem;
```

```
int rc;
```

```
...
```

```
rc = sem_init(&sem, 0, 1);
```

POSIX semaphores: wait

- Acquire the semaphore

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

- `sem_wait` decrements (locks) the semaphore pointed to by `sem`
- If the value of the semaphore's counter is greater than zero then it is decremented and the call succeeds, otherwise the call is blocked until some other task increments the counter and it becomes greater than zero
- If the caller of `sem_wait` is blocked then it is suspended, allowing other tasks to proceed, its TCB is stored in a queue of tasks waiting for the semaphore, so that it can be rescheduled when the semaphore becomes available — notice **NO BUSY WAITING**

- Example

```
sem_t sem;
```

```
int rc;
```

```
rc = sem_wait(&sem);
```

```
assert(rc == 0);
```


POSIX semaphores: post

- Release the semaphore

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

- `sem_post` increments (unlocks) the semaphore pointed to by `sem`
- If the semaphore's value becomes greater than zero then another task blocked waiting for the semaphore can be woken up and scheduled for execution
- `sem_post` returns 0 on success, on error -1 is returned, the value of the semaphore is unchanged and `errno` is set to indicate the error

- Example

```
sem_t sem;  
int rc;
```

```
rc = sem_post(&sem);  
assert(rc == 0);
```

Mutual exclusion using semaphores

```
void *count1_thr(void * arg) {  
    int rc;  
  
    while (!flashing) {  
  
        rc = sem_wait(&sem);  
        assert(rc == 0);  
  
        count1 += 1;  
        total += 1;  
        if ((count1 + count2) != total) {  
            flashing = true;  
        }  
  
        rc = sem_post(&sem);  
        assert(rc == 0);  
    }  
}
```

Mutual exclusion using semaphores

```
void *count1_thr(void * arg) {  
    int rc;  
  
    while (!flashing) {  
  
        rc = sem_wait(&sem);  
        assert(rc == 0);  
  
        count1 += 1;  
        total += 1;  
        if ((count1 + count2) != total) {  
            flashing = true;  
        }  
  
        rc = sem_post(&sem);  
        assert(rc == 0);  
    }  
}
```

ENTRY PROTOCOL

Mutual exclusion using semaphores

```
void *count1_thr(void * arg) {  
    int rc;  
  
    while (!flashing) {  
  
        rc = sem_wait(&sem);  
        assert(rc == 0);  
  
        count1 += 1;  
        total += 1;  
        if ((count1 + count2) != total) {  
            flashing = true;  
        }  
  
        rc = sem_post(&sem);  
        assert(rc == 0);  
    }  
}
```

ENTRY PROTOCOL

CRITICAL SECTION

Mutual exclusion using semaphores

```
void *count1_thr(void * arg) {  
    int rc;
```

```
    while (!flashing) {
```

```
        rc = sem_wait(&sem);  
        assert(rc == 0);
```

ENTRY PROTOCOL

```
        count1 += 1;  
        total += 1;  
        if ((count1 + count2) != total) {  
            flashing = true;  
        }
```

CRITICAL SECTION

```
        rc = sem_post(&sem);  
        assert(rc == 0);
```

EXIT PROTOCOL

```
    }
```

Resource access using semaphores

- Imagine a system to control access to a limited number of resources (e.g. parking spaces)

```
/*  
 * Initialise a semaphore to total  
 * number of parking spaces  
 */  
rc = sem_init(&sem, 0, 5);  
  
/* Wait for parking space */  
rc = sem_wait(&sem);  
  
/* park car */  
  
/* Leave parking space */  
rc = sem_post(&sem);
```

Signalling using semaphores

- Imagine we want to ensure some ordering between functions in 2 tasks

Task A

```
/* await Task B */  
rc = sem_wait(&sem);  
  
doSomeStuffLater();
```

Task B

```
doSomeStuffEarlier();  
  
/* signal Task A */  
rc = sem_post(&sem);
```

- Task A must wait for task B, ie B must be allowed to execute `doSomeStuffEarlier()` before A is allowed to execute `doSomeStuffLater()`

Rendezvous using semaphores

Task A

```
someA1stuff();  
rc = sem_post(&aArrived);  
rc = sem_wait(&bArrived);  
someA2stuff();
```

Task B

```
someB1stuff();  
rc = sem_post(&bArrived);  
rc = sem_wait(&aArrived);  
someB2stuff();
```

- Task A has to wait for task B and vice versa

Summary

- A semaphore is a data structure managed by the operating system, consisting of an integer counter and a queue of TCBs
- Semaphores allow us to solve the mutual exclusion problem **without busy waiting**
- Semaphores can be used to solve other synchronisation problems such as
 - resource allocation
 - signalling
 - rendezvous