# Computer Networks, Security, and Operating Systems

David Kendall

Northumbria University

# Introduction

- 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

# Semaphores

### Semaphore definition

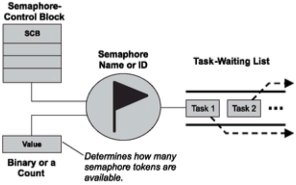
A semaphore is a kernel object that one or more tasks can acquire or release for the purposes of synchronisation or mutual exclusion.

- 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

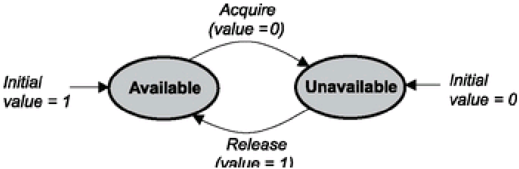
- Semaphore value initially 1
- Task calling `acquire(s)` when `s == 1` acquires the semaphore and s becomes 0
- Task calling `acquire(s)` when `s == 0` is suspended
- Task calling `release(s)` makes ready a previously suspended task if there are any
- Task calling `release(s)` restores value of s to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

# Counting semaphores (Carel Scholten)

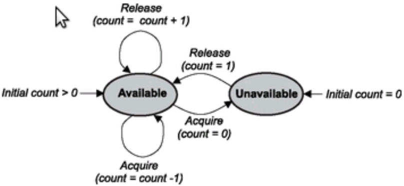



- Idea of binary semaphore can be generalised to counting semaphore (car park example)
- Each `acquire(s)` decreases value of s by 1 down to 0
- Each `release(s)` increases value of s by 1 up to some maximum
- Task waiting list used for tasks waiting on unavailable semaphore
- Waiting list may be FIFO 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);

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

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

ENTRY PROTOCOL

CRITICAL SECTION

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 | Task B |
|---|---|
| `/* await Task B */`<br>`rc = sem_wait(&sem);`<br><br>`doSomeStuffLater();` | `doSomeStuffEarlier();`<br><br>`/* signal Task A */`<br>`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 | Task B |
| --- | --- |
| `someA1stuff();`<br>`rc = sem_post(&aArrived);`<br>`rc = sem_wait(&bArrived);`<br>`someA2stuff();` | `someB1stuff();`<br>`rc = sem_post(&bArrived);`<br>`rc = sem_wait(&aArrived);`<br>`someB2stuff();` |

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

# Producer/consumer problem

- Very often in OS and concurrent applications programs, we have one or more tasks that produce data that must be used (consumed) by some other task(s).
- The rate at which data is produced may be, occasionally, greater than the rate at which data can be consumed
- A buffer can be useful to smooth out the differences in the rates of production and consumption

# Producer/consumer problem

## Real-world analogy

Imagine two people washing up. One person (the producer) washes the dishes and puts them in the dish rack (the buffer). The other person (the consumer) takes the dishes from the dish rack and dries them. If the dish rack fills up, the washer has to wait until the drier takes a dish from the rack. If the rack is empty, the drier has to wait for the washer to wash another dish and put it in the rack. (from *[Goetz et al., 2006]*)

- Our problem is to implement the dish rack ...
- ...and to ensure that the washer-up and drier use it properly

# Naive circular buffer (.h)

```
#ifndef __BUFFER_H
#define __BUFFER_H

enum {
  BUF_SIZE = 6UL
};

typedef struct message {
  unsigned int data;
} message_t;

void putBuffer(message_t const * const);
void getBuffer(message_t * const);

#endif
```

# Naive circular buffer (.c)

```
#include <buffer.h>

static message_t buffer[BUF_SIZE];
static unsigned int front = 0;
static unsigned int back = 0;

void putBuffer(message_t const * const msg) {
  buffer[back] = *msg;
  back = (back + 1) % BUF_SIZE;
}

void getBuffer(message_t * const msg) {
  *msg = buffer[front];
  front = (front + 1) % BUF_SIZE;
}
```

# Naive circular buffer (Use)

```
/* producer */
#include <buffer.h>

message_t msg;

...
msg.data = 27;
putBuffer(&msg);


/* consumer */
#include <buffer.h>

message_t msg;

...
getBuffer(&msg);
lcdWrite(''%u'', msg.data);
```

# Problems with a naive buffer

- Interference between producer(s) and consumer(s)
  - Imagine two producers (P1 and P2) concurrently executing `putBuffer`: P2 does `buffer[back] = ...` and is then descheduled; P1 starts and finishes `putBuffer(...)`; P2 finishes `putBuffer(...)`.
  - What has gone wrong? Draw the state of the buffer.
- Attempt to put data into a full buffer
  - No room on the dish rack – must wait.
- Attempt to get data from an empty buffer
  - No dishes to dry – must wait.

# Elements of a solution

- Enforce mutual exclusion to avoid interference
  - Semaphore `bufMutex` initialized to the value 1
- Enforce producer wait if no buffer slots are empty
  - Semaphore `emptySlot` initialized to the value `BUF_SIZE`
- Enforce consumer wait if no buffer slots are full
  - Semaphore `fullSlot` initialized to the value 0

# Structure of producer

Pseudo-code

```
while (true)

  //    produce an item

  wait (emptySlot);
  wait (bufMutex);

  //  add the item to the  buffer

  post (bufMutex);
  post (fullSlot);

}
```

# Structure of consumer

Pseudo-code

```
while (true) {
  wait (fullSlot);
  wait (bufMutex);

  // remove an item from buffer

  post (bufMutex);
  post (emptySlot);

  // consume the item

}
```

# 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