Operating systems and concurrency B06

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

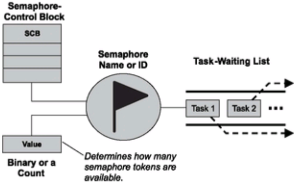
A semaphore is a kernel object that one or more tasks can acquire or release for the purposes of synchronisation or mutual exclusion.

- 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

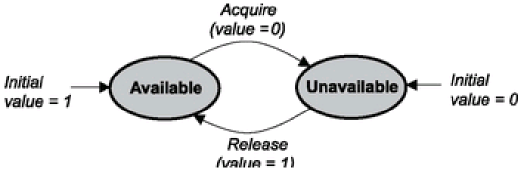
- Semaphore value initially 1
- Task calling `acquire(s)` when `s == 1` acquires the semaphore and s becomes 0
- Task calling `acquire(s)` when `s == 0` is suspended
- Task calling `release(s)` makes ready a previously suspended task if there are any
- Task calling `release(s)` restores value of s to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

Counting semaphores (Carel Scholten)

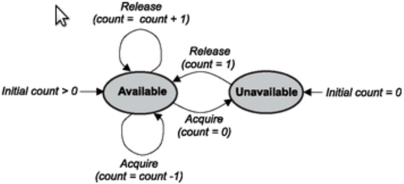


- Idea of binary semaphore can be generalised to counting semaphore (car park example)
- Each `acquire(s)` decreases value of s by 1 down to 0
- Each `release(s)` increases value of s by 1 up to some maximum
- Task waiting list used for tasks waiting on unavailable semaphore
- Waiting list may be FIFO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

Semaphore state diagrams



Binary semaphore



Counting semaphore

uC/OS-II semaphores: Create

- Must create a semaphore before using it
  `OS_EVENT *OSSemCreate(INT16U count);`
  - `count` specifies the initial value of the semaphore
  - `OSSemCreate` creates and returns a pointer to an `OS_EVENT` block that the OS uses to store info about the state of the semaphore
- Example

```
OS_EVENT *lcdSem;

...

lcdSem = OSSemCreate(1);
```

uC/OS-II semaphores: Pend

- Acquire the semaphore

```
void OSSemPend(OS_EVENT *pevent,
               INT32U timeout,
               INT8U *perr);
```

  - `pevent` must be a pointer to the `OS_EVENT` representing the semaphore that you want to acquire
  - `timeout` specifies how many ticks to wait before giving up waiting for the semaphore (if `timeout` is 0, then wait as long as it takes)
  - `perr` is a pointer to an integer that the OS can use to tell the caller whether the operation was successful or not

- Example

```
INT8U error;

OSSemPend(lcdSem, 0, &error);
```

uC/OS-II semaphores: Post

- Release the semaphore
  `INT8U OSSemPost(OS_EVENT *pevent);`
  - `pevent` must be a pointer to the `OS_EVENT` representing the semaphore that you want to release
  - the result returned is an integer that the OS can use to tell the caller whether the operation was successful or not
- Example

```
error = OSSemPost(lcdSem);
```

- Suspended tasks are made ready by `OSSemPost` in priority order

Mutual exclusion using semaphores

```c
static void appTaskCount1(void *pdata) {
  uint8_t error;

  while (true) {

    OSSemPend(lcdSem, 0, &error);

    count1 += 1;
    display(1, count1);
    total += 1;

    error = OSSemPost(lcdSem);

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
}
```

(See mutexsem.c)

Mutual exclusion using semaphores

```
static void appTaskCount1(void *pdata) {
  uint8_t error;

  while (true) {

    OSSemPend(lcdSem, 0, &error);

    count1 += 1;
    display(1, count1);
    total += 1;

    error = OSSemPost(lcdSem);

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
}
```

ENTRY PROTOCOL

(See mutexsem.c)

Mutual exclusion using semaphores

```
static void appTaskCount1(void *pdata) {
  uint8_t error;

  while (true) {

    OSSemPend(lcdSem, 0, &error);

    count1 += 1;
    display(1, count1);
    total += 1;

    error = OSSemPost(lcdSem);

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
}
```

ENTRY PROTOCOL

CRITICAL SECTION

(See mutexsem.c)

Mutual exclusion using semaphores

```
static void appTaskCount1(void *pdata) {
  uint8_t error;

  while (true) {

    OSSemPend(lcdSem, 0, &error);

    count1 += 1;
    display(1, count1);
    total += 1;

    error = OSSemPost(lcdSem);

    if ((count1 + count2) != total) {
      flashing = true;
    }
    OSTimeDlyHMSM(0,0,0,20);
  }
}
```

ENTRY PROTOCOL

CRITICAL SECTION

EXIT PROTOCOL

(See mutexsem.c)

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
 */
OS_EVENT *s = OSSemCreate(5);

/* Wait for parking space */
OSSemPend(s, 0, &error);

/* park car */

/* Leave parking space */
error = OSSemPost(s);
```

Signalling using semaphores

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

Task A	Task B

`/* await Task B */`
`OSSemPend(s, 0, &error);`
 	
`doSomeStuffLater();`
	`doSomeStuffEarlier();`
	
	`/* signal Task A */`
	`error = OSSemPost(s);`

- 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();`
`error = OSSemPost(aArrived);`
`OSSemPend(bArrived, 0, &error);`
`someA2stuff();`
	`someB1stuff();`
	`error = OSSemPost(bArrived);`
	`OSSemPend(aArrived, 0, &error);`
	`someB2stuff();`

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

Acknowledgements

- Labrosse, J., MicroC/OS-II: The Real-time Kernel, CMP, 2002
- Li, Q. and Yao, C., Real-time concepts for embedded systems, CMP, 2003
- Cooling, N, Semaphores Part 1 and 2, Sticky Bits Blog, 2009