Operating systems and concurrency B07

David Kendall

Northumbria University

Introduction

- Semaphores can be used to solve a number of classical synchronisation problems
- We will consider:
  - Producer/consumer problem
  - Readers/writers problem

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)

```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

}
```

Readers and writers

- Multiple tasks require access to a shared data structure, database or filesystem
- Some tasks only need to read the data (readers)
- Other tasks only need to write the data (writers)
- We need to avoid interference (how might this occur?)
- Full mutual exclusion may be inefficient (why?)
- So we require:
    1. Any number of readers can be allowed to read simultaneously
    2. A writer must have exclusive access (ie no other writers and no readers can access the data at the same time as the writer)

Elements of a solution

- Ensure mutually exclusive access to the data when writing
  - Semaphore `writeMutex` initialised to 1
- Keep a count of the number of readers
  - `unsigned int nReaders` initialised to 0
- Ensure mutually exclusive access to the readers count
  - Semaphore `nReadersMutex` initialised to 1

Structure of writer

Pseudo-code

```
while (true) {
  wait(writeMutex);

 /* write the data */

  post(writeMutex);

 /* do non-critical stuff */
}
```

- Protocol for a writer is very simple...
- ...it needs exclusive access to the data

Structure of a reader

Pseudo-code

```
while (true) {
  wait(nReadersMutex);
  nReaders += 1;
  if (nReaders == 1) {
    wait(writeMutex);
  }
  post(nReadersMutex);

  /* read the data */

  wait(nReadersMutex);
  nReaders -= 1;
  if (nReaders == 0) {
    post(writeMutex);
  }
  post(nReadersMutex);

  /* do non-critical stuff */
}
```

- We keep track of the number of readers
- The first reader to arrive needs to ensure that there are no writers
  `wait(writeMutex)`
- The last reader to finish should release any waiting writers `post(writeMutex)`
- Readers that arrive while the first reader is waiting for a writer to finish are suspended by the first
  `wait(nReadersMutex)`

Problem with this solution

- There's a problem with this solution that we'll consider in a later session.

Summary

- Semaphores allow us to solve a variety of synchronisation problems
- Care is required to avoid a number of problems . . .
- . . . to be considered later

Acknowledgements

- Silberschatz, Galvin, Gagne, *Operating System Concepts*, John Wiley, 2013
- B. Goetz with T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*, Addison Wesley, 2006