Designing shared objects: the Anderson/Dahlin method

Glenn Bruns
CSUMB

Lecture Objectives

After this lecture, you should be able to:

- Write simple, safe code for multi-threaded programs
- Understand best design practices
- Avoid common pitfalls in using synchronization primitives

Recap

- We're learning to write concurrent programs
- We're using C with the pthreads library
- We've written concurrent programs using synchronization primitives like locks and condition variables
- We know concurrent programming is tricky ☹

Recap: condition variables

A condition variable is a synchronization object that lets threads wait efficiently

`pthread_cond_wait(cond, lock)`

- lock is released, calling thread is suspended and put on the condition variable's waiting list
- lock is re-acquired before wait returns

`phtread_cond_signal(cond)`

- takes a thread off the waiting list and marks it as "ready"
- if no thread on the waiting list, does nothing

`phtread_cond_broadcast(cond)`

- like signal, but takes all threads off the waiting list

The Anderson/Dahlin approach

- Thomas Anderson and Michael Dahlin have a method for writing concurrent code
- They use only locks and condition variables – no semaphores
- Concurrent code is packaged as shared objects
- The method guides you through design of the shared object

Method, part 1: class design

- Identify classes
- Define interfaces, and identify state variables
- Implement methods

This is just how you would do class design with an OO language

Method, part 2: multi-threaded case

- add a single lock
- add code to acquire and release the lock
- add zero or more condition variables
- add wait calls within loops
- add signal and broadcast calls

We'll explore these through an example

Simple bounded buffer: class design

```
typedef struct {
  // state variables
  int cnt;      // 0 or 1, depending on whether buffer empty or not
  int val;      // value of item in buffer
} SBUF;

// create a new synchronized buffer
SBUF *sbuf_create();

// write to the buffer
void sbuf_write(SBUF *sbuf, int a);

// read from the buffer
int sbuf_read(SBUF *sbuf);
```

This illustrates how to do OO-style code (without inheritance) in C

Implement methods

```
typedef struct {
  int cnt;
  int val;
} SBUF;

// create a new synchronized buffer
SBUF *sbuf_create() {
  SBUF *sbuf = (SBUF *)malloc(sizeof(SBUF));
  sbuf->cnt = 0;
  sbuf->val = 0;
  return sbuf;
}

// write to the buffer
                                    ) {
  sbuf->cnt = 1;
}

// read from the buffer
  sbuf->cnt = 0;
  return(a);
}
```

to be done in lab

to be done in lab

No synchronization variables are used in this step

Add lock, and code to use it

```
typedef struct {
  // state variables
  int cnt;
  int val;
  // sync. variables
  pthread_mutex_t lock;
} SBUF;
```

```
void sbuf_write(SBUF *sbuf, int a) {
  pthread_mutex_lock(&sbuf->lock);
    to be done in lab
  pthread_mutex_unlock(&sbuf->lock);
}

int sbuf_read(SBUF *sbuf) {
    to be done in lab
  return(a);
}
```

- normally a shared object will have one lock
- each method begins and end with locking/unlocking

Add condition variables

```
typedef struct {
  // state variables
  int cnt;
  int val;
  // sync. variables
  pthread_mutex_t lock;
  pthread_cond_t read_go;
  pthread_cont_t write_go;
} SBUF;
```

- think about situations in which methods will have to wait
- if method never need to wait, no condition vars needed
- in this step the designer has a lot of freedom

Add wait calls inside loops

```
void sbuf_write(SBUF *sbuf, int a) {
    to be done in lab
  while (“buffer full”) {
    pthread_cond_wait(&sbuf->write_go, &sbuf->lock);
  }
    to be done in lab
}
```

```
int sbuf_read(SBUF *sbuf) {
    to be done in lab
}
```

The conditions are in English to make their meaning clear.

“buffer full” becomes `sbuf->cnt == 1`

“buffer empty” becomes `sbuf->cnt == 0`

Add signal or broadcast calls

We'll do this in lab.

Remember:

- Signalling is always done when lock is held
- Think about the condition that threads are waiting on

Benefits to using this method

- Threading code is hidden in a class
- Easier to get code right
- Code is easier to read
- No semaphores: they can be tricky

This method makes pthreads programming easier.
It's still not easy!

Summary

The Anderson/Dahlin method for designing shared objects:

1. start with normal class design
2. add a lock to the class; enclose method bodies with lock/unlock calls
3. add 0 or more condition variables to the class
4. add wait calls within loops
5. add signal and broadcast calls