Scuola Superiore Sant’Anna

A design Methodology for Concurrent programs

Giuseppe Lipari
A DESIGN METHODOLOGY FOR SHARED MEMORY PROGRAMS
Methodology

• We present here a structured methodology for programming in shared memory systems
• The methodology leads to “safe” programs
• Not necessarily optimized!
• Programmers can always use their intelligence and come up with “elegant” solutions
Shared memory programming

• A program is a set of
  – threads that interact by accessing ...
  – data structures (shared resources)

• A data structure can be seen as
  – a set of variables
  – a set of functions operating on the variables

• Threads
  – access the data structures only through functions
Object Oriented programming

- In C++

```cpp
class SharedData {
    int array[10];
    int first, last;
    ...
    public:
        SharedData();
        int insert(int a);
        int extract();
};
```

- In C

```c
struct SharedData {
    int array[10];
    int first, last;
};
...

SharedData_init(struct SharedData *d);
int SharedData_insert(struct SharedData *d, int a);
int SharedData_extract(struct SharedData *d);
```
Shared Data structure

- Encapsulating the semaphores
  - The data structure
    - should already include the mechanisms for mutual exclusion and synchronization
    - for example, the CircularArray data structure
  - Functions on the data structure
    - should use the semaphores inside the function
  - Threads
    - can only access the data structure through functions
How to program the data structure?

• Some design considerations
  – First, design the interface: that is, which functions the threads need to call
  – Mutual exclusion
    • for simplicity, all functions on the same data structure should be in mutual exclusion
    • maybe, this is not optimized, but it is SAFE!
Mutual exclusion

- For each data structure,
  - define a mutual exclusion semaphore, initialized to 1
- For each function
  - just after starting the function, take the semaphore, and leave it before returning
Mutual exclusion

• Example

class MyData {
    ...;
    sem_t m;
    ...;
public:
    MyData() {
        ...
        sem_init(&m, 0, 1);
    }

    int myfun() {
        sem_wait(&m);
        ....
        sem_post(&m);
    }
};
Synchronization

• Design consideration

  – Specify the behavior of the functions
    • under which conditions a calling thread should be blocked?
  – Identify all different blocking conditions
Synchronization

• Other design considerations
  – identify unblocking conditions
    • when a blocked thread should be unblocked
    • mark the functions that should unblock those threads

• Putting all together
  – draw a state diagram for the resource
  – STATES: the various states of the resource
  – EVENTS: threads that call the functions
Example: CircularArray with Manager

• Problem explanation
• Design the data structure interface
• Identify blocking and unblocking conditions
• Draw the state diagram
Coding Rules

• For each blocking condition
  – a semaphore initialized to 0 (*blocking semaphore*)
  – an integer that counts the number of blocked threads on the condition (*blocking counter*) (init to 0)
• One (or more) integer variable(s) to code the state
• Write the initialization function (constructor in C++)
The functions

• Finally, code the functions
  – take the mutex at the beginning
  – check the blocking conditions (if any)
  – if the thread has to be blocked,
    • increment the blocking counter, signal on the mutex, wait on the blocking semaphore, wait on the mutex
  – perform the code, change state if necessary
  – check unblocking conditions
  – if a thread has to be unblocked
    • decrement the blocked counter, signal the semaphore
The code
A subtle error

• The “man in the middle” problem
• Solution:
  – check blocking conditions in a while() loop
Passing “le baton”

• A (more elegant) solution
MONITORS

PTHREADS – MUTEXES AND CONDITION VARIABLES
Monitors

- Monitors are a language structure equivalent to semaphores, but cleaner
  - A monitor is similar to an object in a OO language
  - It contains variables and provides procedures to other software modules
  - Only one thread can execute a procedure at a certain time
    ✓ Any other thread that has invoked the procedure is blocked and waits for the first threads to exit
    ✓ Therefore, a monitor implicitly provides mutual exclusion
Monitors

- Monitors support synchronization with *Condition Variables*
  - A condition variable is a blocking queue
  - Two operations are defined on a condition variable
    - \texttt{wait()} -> suspends the calling thread on the queue
    - \texttt{signal()} -> resume execution of one thread blocked on the queue

- Important note:
  - \texttt{wait()} and \texttt{signal()} operation on a condition variable are different from \texttt{wait} and \texttt{signal} on a semaphore!
  - There is not any counter in a condition variable!
  - If we do a signal on a condition variable with an empty queue, the signal is lost
Monitors in Java

• Java provides something that vaguely resembles monitors
  – the “synchronized” keyword allows to define classes with protected functions
  – every “synchronized” class has one implicit condition variable
  – you can “signal” one thread with notify(); and all threads with notifyAll();
Monitors in POSIX

• It is not possible to provide monitors in C
  – C and C++ do not provide any concurrency control mechanism; they are “purely sequential languages”

• POSIX allows something similar to Monitors through library calls
Slides on POSIX monitors
Exercises on POSIX monitors
COMPLEX SYNCHRONIZATION PROBLEMS
READERS - WRITERS
Readers/writers

- One shared buffer

Readers:
- They read the content of the buffer
- Many readers can read at the same time

Writers:
- They write in the buffer
- While one writer is writing no other reader or writer can access the buffer

Use semaphores to implement the resource
Simple implementation

class Buffer {
    Semaphore wsem;
    Semaphore x;
    int nr;

    public:
    Buffer() : wsem(1), x(1), nr(0) {}  
    void read();
    void write();
}

void Buffer::read() {
    x.wait();
    nr++;
    if (nr==1) wsem.wait();
    x.signal();
    <read the buffer>
    x.wait();
    nr--;
    if (nr==0) wsem.signal();
    x.signal();
}

void Buffer::write() {
    wsem.wait();
    <write the buffer>
    wsem.signal();
}
Problem: starvation

- Suppose we have 2 readers (R1 and R2) and 1 writer (W1)
  - Suppose that R1 starts to read
  - While R1 is reading, W1 blocks because it wants to write
  - Now R2 starts to read
  - Now R1 finishes, but, since R2 is reading, W1 cannot be unblocked
  - Before R2 finishes to read, R1 starts to read again
  - When R2 finishes, W1 cannot be unblocked because R1 is reading
Priority to writers!

class Buffer {
    Semaphore x, y, z, wsem, rsem;
    int nr, nw;

    public:
        Buffer() : x(1), y(1), z(1), wsem(1), rsem(1), nr(0), nw(0) {
        
    
    void Buffer::read() {
        z.wait();
        rsem.wait();
        x.wait();
        nr++;
        if (nr==1) wsem.wait();
        x.signal();
        rsem.signal();
        z.signal();
        <read the buffer>
        wsem.signal();
        y.wait();
        nr--;
        if (nr == 0) rsem.signal();
        y.signal();
    }

    void Buffer::write() {
        y.wait();
        nw++;
        if (nw==1) rsem.wait();
        y.signal();
        wsem.wait();
        <write the buffer>
        wsem.signal();
        y.wait();
        nw--;
        if (nw == 0) rsem.signal();
        y.signal();
    }
}
Problem

- Can you solve the readers/writers problem in the general case?
  - No starvation for readers
  - No starvation for writers

Solution

- Maintain a FIFO ordering with requests
  - If at least one writer is blocked, every next reader blocks
  - If at least one reader is blocked, every next writer blocks
  - One single semaphore!
Solution

class Buffer {
    int nbr, nbw;
    int nr, nw;
    Semaphore rsem, wsem;
    Semaphore m;

public:
    Buffer():
        nbw(0), nbr(0), nr(0), nw(0),
        rsem(0), wsem(0) {}
    void read();
    void write();
};
**Solution**

```cpp
void Buffer::read()
{
    m.wait();
    if (nw || nbw) {
        nbr++;
        m.signal(); rsem.wait(); m.wait();
        while (nbr>0)
            {nbr--; rsem.signal();}
    }
    nr++;
    m.signal();
    <read buffer>;
    m.wait();
    nr--;
    if (nbw && nr == 0) wsem.signal();
    m.signal();
}

void Buffer::write()
{
    m.wait();
    if (nw || nbw || nr || nbr) {
        nbw++;
        m.signal(); wsem.wait(); m.wait();
        nbw--;
    }
    nw++;
    m.signal();
    <read buffer>;
    m.wait();
    nw--;
    if (nbr) {nbr--; rsem.signal();}
    else if (nbw) wsem.signal();
    m.signal();
}
```
MESSAGE PASSING
Message passing

- Message passing systems are based on the basic concept of message

- Two basic operations
  - `send(destination, message);`
  - `receive(source, &message);`

- Two variants
  - Both operations can be synchronous or asynchronous
  - `receive` can be symmetric or asymmetric
Producer/Consumer with MP

- The producer executes `send(consumer, data)`
- The consumer executes `receive(producer, data)`;
- No need for a special communication structure (already contained in the send/receive semantic)
Synchronous communication

- Synchronous send/receive

producer:
  s_send(consumer, d);

consumer:
  s_receive(producer, &d);
Async send/ Sync receive

Asyncronous send / synchronous receive

producer:
   a_send(consumer, d);

consumer:
   s_receive(producer, &d);
Asymmetric receive

- Symmetric receive
  - `receive(source, &data);`

- Often, we do not know who is the sender
  - Imagine a web server;
    - the programmer cannot know in advance the address of the browser that will request the service
    - Many browser can ask for the same service

- Asymmetric receive
  - `source = receive(&data);`
Message passing systems

- In message passing
  - Each resource needs one threads manager
  - The threads manager is responsible for giving access to the resource

- Example: let’s try to implement mutual exclusion with message passing primitives
  - One thread will ensure mutual exclusion
  - Every thread that wants to access the resource must
    - send a message to the manager thread
    - access the critical section
    - send a message to signal the leaving of the critical section
Sync send / sync receive

```c
void * manager(void *)
{
    thread_t source;
    int d;
    while (true) {
        source = s_receive(&d);
        s_receive_from(source, &d);
    }
}

void * thread(void *)
{
    int d;
    while (true) {
        s_send(manager, d);
        <critical section>
        s_send(manager, d);
    }
}
```

ERI Gennaio 2008
With Async send and sync receive

```c
void * manager(void *)
{
    thread_t source;
    int d;
    while (true) {
        source = s_receive(&d);
        a_send(source, d);
        s_receive_from(source, &d);
    }
}

void * thread(void *)
{
    int d;
    while (true) {
        a_send(manager, d);
        s_receive_from(manager, &d);
        <critical section>
        a_send(manager, d);
    }
}
```

ERI Gennaio 2008
A different approach

- **Shared memory**
  - each resource is a class
  - threads ask for services through functions calls
  - they synchronize through mutexes and condition variable

- **Message Passing**
  - each resource has a manager thread
  - threads ask for services through messages and receive response through messages
  - they synchronize through blocking receives
For each resource

- a “manager” thread takes care of executing the services on behalf of the clients/threads
- general structure of a manager:
  - wait for a message
  - decode the message, execute the service
  - eventually, send back the response
- The manager sequentializes all services