Sistemi in tempo reale
Anno accademico 2006 - 2007
Concorrenza - II

Giuseppe Lipari
`http://feanor.sssup.it/~lipari`

Scuola Superiore Sant'Anna

The need for concurrency

There are many reason for concurrency
- functional
- performance
- expressive power

Functional
- many users may be connected to the same system at the same time
  - each user can have its own processes that execute concurrently with the processes of the other users
- perform many operations concurrently
  - for example, listen to music, write with a word processor, burn a CD, etc...
  - they are all different and independent activities
  - they can be done *at the same time*

The need for concurrency (2)

Performance
- take advantage of blocking time
  - while some thread waits for a blocking condition, another thread performs another operation
- parallelism in multi-processor machines
  - if we have a multi-processor machine, independent activities can be carried out on different processors are the same time

Expressive power
- many control application are inherently concurrent
- concurrency support helps in expressing concurrency, making application development simpler

Concurrency model

- a system is a set of concurrent activities
  - they can be processes or threads
- they interact in two ways
  - access the hardware resources (processor, disk, memory, etc.)
  - exchange data
- these activities compete for the resources and/or cooperate for some common objective

Resources

- a resource can be
  - a HW resource like a I/O device
  - a SW resource, i.e. a data structure
  - in both cases, access to a resource must be regulated to avoid interference
- Example 1
  - if two processes want to print on the same printer, their access must be sequentialised, otherwise the two printing could be intermangled!
- Example 2
  - if two threads access the same data structure, the operation on the data must be sequentialized otherwise the data could be inconsistent!

Interaction model

Activities can interact according to two fundamental models
- **shared memory**
  - All activities access the same memory space
- **message passing**
  - All activities communicate each other by sending messages through OS primitives
- we will analize both models in the following slides

Shared memory

Shared memory communication
- it was the first one to be supported in old OS
- it is the simplest one and the closest to the machine
- all threads can access the same memory locations
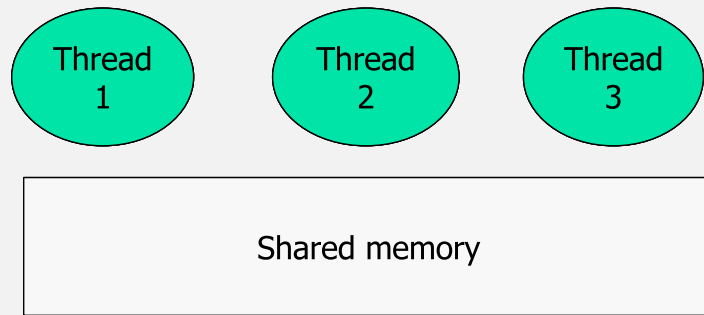
Mutual Exclusion Problem

- We do not know in advance the relative speed of the processes
  - hence, we do not know the order of execution of the hardware instructions
- Recall the example of incrementing variable x
  - incrementing x is not an atomic operation
  - atomic behavior can be obtained using interrupt disabling or special atomic instructions

Example 1

```
/* Shared memory */
int x;
```

```
void *threadA(void *)
{
  ...;
  x = x + 1;
  ...;
}
```

```
void *threadB(void *)
{
  ...;
  x = x + 1;
  ...;
}
```

- Bad Interleaving:

```
...
LD    R0, x     (TA)  x = 0
LD    R0, x     (TB)  x = 0
INC   R0        (TB)  x = 0
ST    x, R0     (TB)  x = 1
INC   R0        (TA)  x = 1
ST    x, R0     (TA)  x = 1
...
```

Example 2

```
// Shared object (sw resource)
class A {
    int a;
    int b;
public:
    A() : a(1), b(1) {};
    void inc() {
       a = a + 1; b = b +1;
    }
    void mult() {
       b = b * 2; a = a * 2;
    }
} obj;
```

```
void * threadA(void *)
{
    ...
    obj.inc();
    ...
}
```

```
void * threadB(void *)
{
    ...
    obj.mult();
    ...
}
```

Consistency: After each operation, a == b

Resource in a non-consistent state!!

```
a = a + 1;    TA    a = 2
b = b * 2;    TB    b = 2
b = b + 1;    TA    b = 3
a = a * 2;    TB    a = 4
```

Consistency

- for any resource, we can state a set of consistency properties
  - a consistency property C_i is a boolean expression on the values of the internal variables
  - a consistency property must hold before and after each operation
  - it does not need to hold during an operation
  - if the operations are properly sequentialized, the consistency properties will always hold
- formal verification
  - let R be a resource, and let $C(R)$ be a set of consistency properties on the resource
  - $C(R) = \{C_i\}$
  - A concurrent program is correct if, for every possible interleaving of the operations on the resource, $\forall C_i \in C(R)$, C_i holds.

# Example: Circular Array
Implementation of a FIFO queue.

```c
struct CA {
  int array[10];
  int head, tail, num;
}

void init(struct CA *ca) {
  ca->head=0; ca->tail=0;
  ca->num=0;
}
boolean insert(struct CA *ca, int elem) {
  if (ca->num == 10) return false;
  ca->array[ca->head] = elem;
  ca->head = (ca->head + 1) % 10;
  ca->num ++;
  return true;
}
boolean extract(struct CA *ca, int *elem) {
  if (ca->num == 0) return false;
  *elem = ca->array[ca->tail];
  ca->tail = (ca->tail + 1) % 10;
  ca->num--;
  return true;
}
```

Example: empty queue
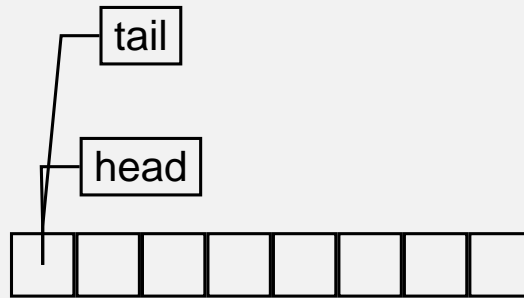
- head: index of the first free element in the queue
  - here will be inserted the next element
- tail: index of the first occupied element in the queue
  - will be the one that will be extracted next time
- the queue is empty, hence head == tail
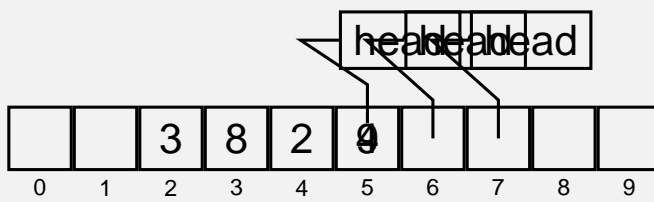
Example: insert


```
boolean insert(struct CA *ca,
               int elem)
{
  if (ca->num == 10)
      return false;
  ca->array[ca->head] = elem;
  ca->head = (ca->head+1)%10;
  ca->num++;
  return true;
}
```

- num = (head - tail) % 8 → num = 4;
- insert(ca, 9);
- head and num have been increased

Example: concurrent insert


```
boolean insert(struct CA *ca,
               int elem)
{
   if (ca->num == 10)
      return false;
   ca->array[ca->head] = elem;
```

- Two threads, the first calls insert(9), the second calls insert(4);
- thread 1 calls insert(ca, 9);
- preemption by second thread
- second thread completes
- there is a hole! At some point, the extract will read a 4 and a random value, instead of a 9 and a 4.

```
...
boolean insert(struct CA *ca,
               int elem)
{
   if (ca->num == 10)
      return false;
   ca->array[ca->head] = elem;
   ca->head = (ca->head+1)%10;
   ca->num++;
   return true;
}
...
```

```
   ca->head = (ca->head+1)%10;
   ca->num++;
   return true;
}
```

Consistency properties for struct CA

1. when the queue is empty, or when the queue is full,
   `head == tail`
2. `num` is equal to the number of times insert has been called minus the number of times that extract has been called
3. . . .
4. if element x has been inserted, eventually it must be extracted with an appropriate number of extracts
5. Every element that is extracted, has been inserted sometime in the past.

Last two can also be expressed as:

- Let $(x_1, x_2, \ldots, x_k)$ be the sequence of inserted elements, and let $(y_1, y_2, \ldots, y_k)$ be the sequence of extracted elements;
- then $\forall i = 1, \ldots, k \quad y_i = x_i$

Correctness of Circular Array implementation

- The previous program is not correct, as the last property is not verified
  - the sequence of elements extracted does not correspond to the sequence of elements inserted
  - The problem is that the first thread was preempted while updating the data structure in a critical point.
  - we must prevent thread 2 from accessing the data structure while another thread is completing an operation on it
- Proving non-correctness is *easy*, in the sense that we must find a counterexample
- Proving correctness is a very complex task!
  - it is necessary to prove the correctness for every possible interleaving of every operation, for every possible input data, and for every possible internal state

Insert and Extract

Let's assume that increments and decrements are atomic operations

- Producer: thread that inserts elements
- Consumer: thread that extracts elements

- It can be proved that interleaving exactly one producer and one consumer does not bring any problem
  - proof: if $0 < num < 10$, `insert()` and `extract()` are independent
  - if `num==0`
    - if `extract()` begins before `insert`, it immediately returns false,
    - if `insert` begins before, `extract` will still return false, so it cannot interfere with `insert`
  - same thing when num==10
- correctness is guaranteed for one consumer and one producer.

Insert and Extract - II

- What happens if we exchange the sequence of instructions in insert?

```
boolean insert(struct CA *ca,
               int elem)
{
  if (ca->num == 10)
     return false;
  ca->num++;
  ca->array[ca->head] = elem;
  ca->head = (ca->head+1)%10;
  return true;
}
```

- It is easy to prove that in this case `insert()` cannot be interleaved with `extract`

Critical sections

- the shared object where the conflict may happen is a resource
- the parts of the code where the problem may happen are called critical sections
- a critical section is a sequence of operations that cannot be interleaved with other operations on the same resource
- two critical sections on the same resource must be properly sequentialized
- we say that two critical sections on the same resource must execute in MUTUAL EXCLUSION
- there are three ways to obtain mutual exclusion
  - implementing the critical section as an atomic operation
  - disabling the preemption (system-wide)
  - selectively disabling the preemption (using semaphores and mutex)

Implementing atomic operations

- In single processor systems
  - disable interrupts during a critical section
  - non-voluntary context switch is disabled!

  ```
  CLI;
  <critical section>
  STI;
  ```

  - Limitations:
    - if the critical section is long, no interrupt can arrive during the critical section
      - consider a timer interrupt that arrives every 1 msec.
      - if a critical section lasts for more than 1 msec, a timer interrupt could be lost
      - It must be done only for very short critical section;
    - Non voluntary context switch is disabled during the critical section
      - Disabling interrupts is a very low level solution: it is not possible in user space.

Atomic operations on multiprocessors

- Disabling interrupts is not sufficient
  - disabling interrupts on one processor lets a thread on another processor free to access the resource
- Solution: use `lock()` and `unlock()` operations
  - define a flag `s` for each resource, and then surround a critical section with `lock(s)` and `unlock(s)`;

  ```
  int s;
  ...
  lock(s);
  <critical section>
  unlock(s);
  ...
  ```

Disabling preemption

- On single processor systems
  - in some scheduler, it is possible to disable preemption for a limited interval of time
- problems:
  - if a high priority critical thread needs to execute, it cannot make preemption and it is delayed
  - even if the high priority task does not access the resource!

```
disable_preemption();
<critical section>
enable_preemption();
```

no context switch may happen during the critical section,
but interrupts are enabled

Software mutual exclusion

- In the past, many algorithms have been proposed that only use a software approach to synchronization.
- Such algorithms are not very general, and not very efficient. They are useful for didactic purpouse, as the readr may get some insight into the problem

Dekker's algorithm

Global data

```
bool f0 = false;
bool f1 = false;
int turn = 0;
```

Process 0

```
f0 = true;
while (f1) {
    if (turn != 0) {
        f0 = false;
        while (turn != 0);
        f0 = true;
    }
}

// critical section
...
turn = 1
f0 = false
```

Process 1

```
f1 = true;
while (f0) {
    if (turn != 1) {
        f1 = false;
        while (turn != 1);
        f1 = true;
    }
}

// critical section
...
turn = 0
f1 = false
```

Peterson's algorithm

Global data

```
bool flag[0]   = false;
bool flag[1]   = false;
int turn       = 0;
```

Thread 1

```
flag[0] = true;
turn = 1;
while(flag[1] && turn == 1);

// critical section
...
flag[0] = false;
```

Thread 2

```
flag[0] = true;
turn = 1;
while(flag[1] && turn == 1);

// critical section
...
flag[0] = false;
```

Lamport's bakery algorithms

Global data

```
bool  Entering[N];    //  init to false
int   Number[N];      // init to 0
```

Lock for thread i

```
void lock(int i) {
    Entering[i] = true;
    Number[i] = 1 + max(Number[1],
    Entering[i] = false;
    for (j = 1; j <= N; j++) {
        while (Entering[j]);
        while ((Number[j] != 0) &&
    }
}
```

Unlock for thread i

```
unlock(integer i) {
    Number[i] = 0;
}
```

Thread code

```
Thread(integer i) {
    ...
    lock(i);
    // The critical section goes here...
    unlock(i);
    ...
}
```

Critical sections: a general approach

- General techniques exists to protect critical sections
  - Semaphores
  - Mutex
- Properties:
  - Interrupts always enabled
  - Preemption always enabled
- Basic idea:
  - if a thread is inside a critical section on a given resource
  - all other threads are blocked upon entrance on a critical section on the same resource
- We will study such techniques in the following

Producer / Consumer model

- mutual exclusion is not the only problem
  - we need a way of synchronise two or more threads
- example: producer/consumer
  - suppose we have two threads,
  - one produces some integers and sends them to another thread (PRODUCER)
  - another one takes the integer and elaborates it (CONSUMER)
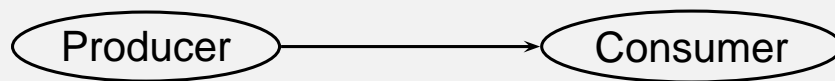
Producer ⟶ Consumer

Implementation with the circular array

- Suppose that the two threads have different speeds
  - for example, the producer is much faster than the consumer
  - we need to store the temporary results of the producer in some memory buffer
  - for our example, we will use the circular array structure

Producer/Consumer implementation

```
struct CA qu;
```

```
void *producer(void *)
{
  bool res;
  int data;
  while(1) {
    <obtain data>
    while (!insert(&qu, data));
  }
}
```

```
void *consumer(void *)
{
  bool res;
  int data;
  while(1) {
    while (!extract(&qu, &data));
    <use data>
  }
}
```

- Problem with this approach:
  - if the queue is full, the producer waits *actively*
  - if the queue is empty, the consumer waits *actively*

A more general approach

- we need to provide a general mechanism for synchonisation and mutual exclusion
- requirements
  - provide mutual exclusion between critical sections
    - avoid two interleaved insert operations
    - (semaphores, mutexes)
  - synchronise two threads on one condition
    - for example, block the producer when the queue is full
    - (semaphores, condition variables)