Classic Problems of Synchronization

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining Philosophers Problem
- Monitors
Readers-Writers Problem

Readers

Writers
Problem Definition

- Database to be shared among several concurrent processes
- Some processes want to read-only
- Some processes want to read-write
- Several different versions
  - First readers-writers problem
  - Second readers-writers problem
Readers-Writers Problem (Cont.)

Reader

Writer

Shared Resource
Readers-Writers Problem (Cont.)

Concurrent readers
Readers-Writers Problem (Cont.)

Shared Resource

Reader

Writer

Exclusive writer
First Solution

Reader() {
    while(TRUE) {
        wait(mutex);
        readCount++;
        if(readCount==1)
            wait(wrt);
        signal(mutex);
        read(resource);
        wait(mutex);
        readCount--;
        if(readCount == 0)
            signal(wrt);
        signal(mutex);
    }
}

Writer() {
    while(TRUE) {
        wait(wrt);
        write(resource);
        signal(wrt);
    }
}

resourceType *resource;
int readCount = 0;
semaphore mutex = 1;
semaphore wrt = 1;

• First reader competes with writers
• Last reader signals writers
First Solution (Cont.)

Reader() {
    while(TRUE) {
        wait(mutex);
        readCount++;
        if(readCount == 1)
            wait(wrt);
        signal(mutex);
        read(resource);
        wait(mutex);
        readCount--;
        if(readCount == 0)
            signal(wrt);
        signal(mutex);
    }
}

Writer() {
    while(TRUE) {
        wait(wrt);
        write(resource);
        signal(wrt);
    }
}

• First reader competes with writers
• Last reader signals writers
• Any writer must wait for all readers
• Readers can starve writers
• “Updates” can be delayed forever
Second Solution: Writer Precedence

Reader() {
    while(TRUE) {
        wait(rd);
        wait(mutex1);
        readCount++;
        if(readCount == 1)
            wait(wrt);
        signal(mutex1);
        signal(rd);

        read(resource);
        wait(mutex1);
        readCount--;
        if(readCount == 0)
            signal(wrt);
        signal(mutext1);
    }
}

writer() {
    while(TRUE) {
        wait(mutex2);
        writeCount++;
        if(writeCount == 1)
            wait(rd);
        signal(mutex2);
        wait(wrt);

        write(resource);
        signal(wrt);
        wait(mutex2)
        writeCount--;
        if(writeCount == 0)
            signal(rd);
        signal(mutex2);
    }
}

int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore rd = 1, wrt = 1;
The Dining Philosophers Problem

- A classical synchronization problem
- 5 philosophers who only eat and think
- Each need to use 2 forks for eating
- There are only 5 forks
- Illustrates the difficulty of allocating resources among process without deadlock and starvation
The Dining Philosophers Problem

- Each philosopher is a process
- One semaphore per fork:
  - Fork: array[0..4] of semaphores
  - Initialization:
    fork[i].count:=1 for i:=0..4
- A first attempt:
  - Deadlock if each philosopher starts by picking his left fork!

```c
P_i() {
    while(TRUE) {
        think;
        wait(fork[i]);
        wait(fork[i+1 mod 5]);
        eat;
        signal(fork[i+1 mod 5]);
        signal(fork[i]);
    }
}
```
The Dining Philosophers Problem

- Idea: admit only 4 philosophers at a time who try to eat
- Then, one philosopher can always eat when the other 3 are holding one fork
- Solution: use another semaphore T to limit at 4 the number of philosophers “sitting at the table”
- Initialize: T.count:=4

\[ P_i() \{
    \text{while}(\text{TRUE}) \{
        \text{think};
        \text{wait}(T);
        \text{wait}(\text{fork}[i]);
        \text{wait}(\text{fork}[i+1 \mod 5]);
        \text{eat};
        \text{signal}(\text{fork}[i+1 \mod 5]);
        \text{signal}(\text{fork}[i]);
        \text{signal}(T);
    \}
\} \]
Recall: Problems with Semaphores

- Semaphores are a powerful tool for enforcing mutual exclusion and coordinate processes.
- Problem: `wait(S)` and `signal(S)` are scattered among several processes:
  - It is difficult to understand their effects.
  - Usage must be correct in all processes.
  - One bad (or malicious) process can fail the entire collection of processes.
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```plaintext
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (...) { .... }
  ...
  procedure Pn (...) {......}

  Initialization code ( ....) { ... }
}
```
Monitors

- Is a software module containing:
  - one or more procedures
  - an initialization sequence
  - local shared data variables

- Characteristics:
  - Local shared variables accessible only by monitor’s procedures
  - a process enters the monitor by invoking one of its procedures
  - only one process can be in the monitor at any one time

- The monitor ensures mutual exclusion
  - no need to program this constraint explicitly

- Shared data are protected by placing them in the monitor
  - The monitor locks the shared data on process entry
Condition Variables

- Process synchronization is done using condition variables, which represent conditions a process may need to wait for before executing in the monitor

- `condition x, y;`

- Local to the monitor (accessible only within the monitor)

- Can be accessed and changed only by two functions:
  - `x.wait()`: blocks execution of the calling process on condition `x`
    - the process can resume execution only if another process executes `x.signal()`
  - `x.signal()`: resume execution of some process blocked on condition `x`.
    - If several such processes exists: choose any one
    - If no such process exists: do nothing
Monitors

- Awaiting processes are either in the entrance queue or in a condition queue.
- A process puts itself into condition queue $cn$ by issuing $cn.wait()$.
- $cn.signal()$ brings into the monitor one process in condition $cn$ queue.
- *signal-and-wait* and *signal-and-continue*
P/C: Finite Circular Buffer of Size k

- Can consume only when the number _full_ of (consumable) items is at least 1
- Can produce only when the number _empty_ of empty spaces is at least 1
Solution of P/C: Finite Circular Buffer of Size $k$

**Initialization:**

- `mutex.count := 1;`
- `in := 0;`
- `full.count := 0;`
- `out := 0;`
- `empty.count := k;`

**Producer:**

```plaintext
while (TRUE) {
    produce item;
    wait (empty);
    wait (mutex);
    append (item);
    signal (mutex);
    signal (full);
}
```

**Critical sections**

- `append(v):`
  - `b[in] := v;`
  - `in := (in+1) mod k;`

- `take():`
  - `w := b[out];`
  - `out := (out+1) mod k;`
  - `return w;`

**Consumer:**

```plaintext
While (TRUE) {
    wait (full);
    wait (mutex);
    item = take();
    signal (mutex);
    signal (empty);
    consume (item);
}
```
Producer/Consumer Using Monitors

- Two types of processes:
  - producers
  - consumers

- Synchronization is now confined within the monitor

- `append(.)` and `take(.)` are procedures within the monitor: are the only means by which P/C can access the buffer

- If these procedures are correct, synchronization will be correct for all participating processes

```
Producer:
while (TRUE) {
   produce item;
   append(item);
}

Consumer:
while (TRUE) {
   item = take();
   consume item;
}
```
Monitor for the Bounded P/C Problem

- **Buffer:**
  - *buffer*: array[0..k-1] of items;

- **Buffer pointers and counts:**
  - *nextin*: points to next item to be appended
  - *nextout*: points to next item to be taken
  - *count*: holds the number of items in the buffer

- **Condition variables:**
  - *notfull*: notfull.signal() indicates that the buffer is not full
  - *notempty*: notempty.signal() indicates that the buffer is not empty
Monitor for the Bounded P/C Problem

Monitor boundedbuffer {
    Item buffer[k];
    integer nextin, nextout, count;
    condition notfull, notempty;

    Append(v){
        if (count==k) notfull.wait();
        buffer[nextin] = v;
        nextin = (nextin+1) mod k;
        count++;
        notempty.signal();
    }

    initialization_code(){
        nextin=0; nextout=0; count=0;
    }

    Item Take(){
        if (count==0)
            notempty.wait();
        v = buffer[nextout];
        nextout =
            (nextout+1) mod k;
        count--;
        notfull.signal();
        return v;
    }
}
Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable
Linux Synchronization

- **Linux:**
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
  - Disable kernel preemption on single processors

- **Linux provides:**
  - semaphores
  - spin locks (mostly on SMP machines)
Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks
Transactional Memory

- Multi-core systems

- A memory transaction
  - Sequence of memory read-write operations that are atomic

- If all operations are completed, transaction is committed

- Otherwise, rolled back

- As the number of threads increase, the use of traditional locking doesn’t scale well