Khancurrency

- Value of concurrency – speed & economics
- But few widely-accepted concurrent programming languages (Ada and Java are exceptions)
- Few concurrent programming paradigm
  - Each problem requires careful consideration
  - There is no common model
- OS tools to support concurrency tend to be “low level”

Critical Sections

shared double balance;

Code for p₁:  
balance = balance + amount;
balance = balance - amount;

Code for p₂:  
balance+=amount
balance-=amount

Critical Sections (2)

Execution of p₁:
load R1, balance
load R2, amount

Timer interrupt
load R1, balance
load R2, amount
sub R1, R2
store R1, balance

Execution of p₂:
load R1, balance

Timer interrupt
add R1, R2
store R1, balance

Critical Sections (3)

- Mutual exclusion: Only one process can be in the critical section at a time
- There is a race to execute critical sections
- The sections may be defined by different code in different processes
  - cannot easily detect with static analysis
- Without mutual exclusion, results of multiple execution are not determinate
- Need an OS mechanism so programmer can resolve races

Some Possible Solutions

- Disable interrupts
- Software solution – locks
- Transactions
- FORK(), JOIN(), and QUIT() [Chapter 2]
  - Terminate processes with QUIT() to synchronize
  - Create processes whenever critical section is complete
  - See Figure 8.7
- ... something new ...

Disabling Interrupts

shared double balance;

Code for p₁:
disableInterrupts();
balance = balance + amount;
enableInterrupts();

Code for p₂:
disableInterrupts();
balance = balance - amount;
enableInterrupts();

- Interrupts could be disabled arbitrarily long
- Really only want to prevent p₁ and p₂ from interfering with one another; this blocks all pᵢ
- Try using a shared “lock” variable
Using a Lock Variable

shared boolean lock = FALSE;
shared double balance;

Code for p1
/* Acquire the lock */
while(lock) { }
lock = TRUE;
/* Execute critical sect */
balance = balance + amount;
/* Release lock */
lock = FALSE;

Code for p2
/* Acquire the lock */
while(lock) { }
lock = TRUE;
/* Execute critical sect */
balance = balance - amount;
/* Release lock */
lock = FALSE;

Busy Wait Condition

shared boolean lock = FALSE;
shared double balance;

Code for p1
/* Acquire the lock */
while(lock) ;  
lock = TRUE;
/* Execute critical sect */
balance = balance + amount;
/* Release lock */
lock = FALSE;

Code for p2
/* Acquire the lock */
while(lock) ;  
lock = TRUE;
/* Execute critical sect */
balance = balance - amount;
/* Release lock */
lock = FALSE;

Unsafe “Solution”

shared boolean lock = FALSE;
shared double balance;

Code for p1
/* Acquire the lock */
while(lock) ;  
lock = TRUE;
/* Execute critical sect */
balance = balance + amount;
/* Release lock */
lock = FALSE;

Code for p2
/* Acquire the lock */
while(lock) ;  
lock = TRUE;
/* Execute critical sect */
balance = balance - amount;
/* Release lock */
lock = FALSE;

• Worse yet … another race condition …
• Is it possible to solve the problem?

Atomic Lock Manipulation

enter(lock) {
  disableInterrupts();
  lock = TRUE;
  while(lock) {
    lock = FALSE;
    enableInterrupts();
  }
  lock = TRUE;
  enableInterrupts();
}

exit(lock) {
  disableInterrupts();
  lock = TRUE;
  while(lock) {
    lock = FALSE;
    enableInterrupts();
  }
  lock = FALSE;
  enableInterrupts();
}

• Bound the amount of time that interrupts are disabled
• Can include other code to check that it is OK to
  assign a lock
• … but this is still overkill …

Deadlock (2)

shared boolean lock1 = FALSE;
shared boolean lock2 = FALSE;
shared list L;

Code for p1
/* Enter CS to delete elt */
enter(lock1);  
<delete element>
<intermediate computation>
enter(lock2);  
<update length>
exit(lock2);  
... 

Code for p2
/* Enter CS to update len */
enter(lock2);  
<update length>
<intermediate computation>
enter(lock1);  
<update length>
<add element>
exit(lock1);  
... 

Processing Two Components

shared boolean lock1 = FALSE;
shared boolean lock2 = FALSE;
shared list L;

Code for p1
/* Enter CS to delete elt */
enter(lock1);  
<delete element>
exit(lock1);  
... 

Code for p2
/* Enter CS to update len */
enter(lock2);  
<update length>
<intermediate computation>
exit(lock2);  
<add element>
... 

Code for p3
/* Enter CS to update len */
enter(lock3);  
<update length>
<intermediate computation>
exit(lock3);  
...
Transactions

- A transaction is a list of operations
  - When the system begins to execute the list, it must execute all of them without interruption, or
  - It must not execute any at all
- Example: List manipulator
  - Add or delete an element from a list
  - Adjust the list descriptor, e.g., length
- Too heavyweight – need something simpler

Dijkstra Semaphore

- Invented in the 1960s
- Conceptual OS mechanism, with no specific implementation defined (could be enter()/exit())
- Basis of all contemporary OS synchronization mechanisms

Solution Constraints

- Processes p_i & p_j enter critical sections
- Mutual exclusion: Only one process at a time in the CS
- Only processes competing for a CS are involved in resolving who enters the CS
- Once a process attempts to enter its CS, it cannot be postponed indefinitely
- After requesting entry, only a bounded number of other processes may enter before the requesting process

Notation

- Let fork(proc, N, arg_1, arg_2, ..., arg_N) be a command to create a process, and to have it execute using the given N arguments

- Canonical problem:
  
  ```
  proc_0() {
    proc_1() {
      while(TRUE) {
        while(TRUE) {
          <compute section>
        }
        <critical section>
      }
    }
  }
  ```

  <shared global declarations>
  <initial processing>
  fork(proc_0, 0);
  fork(proc_1, 0);

Solution Assumptions

- Memory read/writes are indivisible (simultaneous attempts result in some arbitrary order of access)
- There is no priority among the processes
- Relative speeds of the processes/processors is unknown
- Processes are cyclic and sequential

Dijkstra Semaphore Definition

- Classic paper describes several software attempts to solve the problem (see problem 4, Chapter 8)
- Found a software solution, but then proposed a simpler hardware-based solution
- A semaphore, s, is a nonnegative integer variable that can only be changed or tested by these two indivisible functions:
  
  ```
  V(s): [s = s + 1]
  P(s): [while(s == 0) (wait); s = s - 1]
  ```
### Solving the Canonical Problem

```c
proc_0() {
    while(TRUE) {
        <compute section>
        P(mutex);
        <critical section>
        V(mutex);
    }
}

proc_1() {
    while(TRUE) {
        <compute section>
        P(mutex);
        <critical section>
        V(mutex);
    }
}

semaphore mutex = 1;
fork(proc_0, 0);
fork(proc_1, 0);
```

### Shared Account Balance Problem

```c
Proc_0() {
    /* Enter the CS */
    P(mutex);
    balance += amount;
    V(mutex);
    ...}
}

Proc_1() {
    /* Enter the CS */
    P(mutex);
    balance -= amount;
    V(mutex);
    ...}
}

semaphore mutex = 1;
fork(proc_0, 0);
fork(proc_1, 0);
```

### Sharing Two Variables

```c
proc_A() {
    while(TRUE) {
        <compute section A1>
        update(x);
        /* Signal proc_B */
        V(s1);
        <compute section A2>
        /* Wait for proc_B */
        P(s2);
        retrieve(y);
    }
}

semaphore s1 = 0;
semaphore s2 = 0;
fork(proc_A, 0);
fork(proc_B, 0);
```

### Device Controller Synchronization

- The semaphore principle is logically used with the busy and done flags in a controller
- Driver signals controller with a V(busy), then waits for completion with P(done)
- Controller waits for work with P(busy), then announces completion with V(done)

### Bounded Buffer Problem

```c
producer() {
    buf_type *next, *here;
    while(TRUE) {
        produce_item(next);
        /* Claim an empty */
        P(empty);
        P(mutex);
        here = obtain(empty);
        V(mutex);
        copy_buffer(next, here);
        P(mutex);
        release(here, fullPool);
        V(mutex);
        /* Signal a full buffer */
        V(full);
    }
}

consumer() {
    buf_type *next, *here;
    while(TRUE) {
        consume_item(next);
        /* Claim full buffer */
        P(full);
        here = obtain(full);
        V(mutex);
        copy_buffer(here, next);
        P(mutex);
        release(here, emptyPool);
        V(mutex);
        /* Signal an empty buffer */
        V(empty);
        consume_item(here);
    }
}

semaphore mutex = 1;
semaphore full = 0;     /* A general (counting) semaphore */
semaphore empty = N;    /* A general (counting) semaphore */
buf_type buffer[N];
fork(producer, 0);
fork(consumer, 0);
```

### Bounded Buffer Problem (2)

```c
producer() {
    buf_type *next, *here;
    while(TRUE) {
        produce_item(next);
        /* Claim an empty */
        P(empty);
        P(mutex);
        here = obtain(empty);
        V(mutex);
        copy_buffer(here, next);
        P(mutex);
        release(here, fullPool);
        V(mutex);
        /* Signal a full buffer */
        V(full);
    }
}

consumer() {
    buf_type *next, *here;
    while(TRUE) {
        consume_item(next);
        /* Claim full buffer */
        P(full);
        here = obtain(full);
        V(mutex);
        copy_buffer(here, next);
        P(mutex);
        release(here, emptyPool);
        V(mutex);
        /* Signal an empty buffer */
        V(empty);
        consume_item(here);
    }
}

semaphore mutex = 1;
semaphore full = 0;     /* A general (counting) semaphore */
semaphore empty = N;    /* A general (counting) semaphore */
buf_type buffer[N];
fork(producer, 0);
fork(consumer, 0);
```
Bounded Buffer Problem (3)

```c
producer() {  
  buf_type *next, *here;
  while(TRUE) {  
    produce_item(next);
    /* Claim an empty */
    P(empty);
    P(mutex);
    here = obtain(empty);
    V(mutex);
    copy_buffer(next, here);
    P(mutex);
    release(here, fullPool);
    V(mutex);
    /* Signal a full buffer */
    V(full);
  }
}

consumer() {  
  buf_type *next, *here;
  while(TRUE) {  
    consume_item(next);
    /* Claim full buffer */
    P(full);
    P(mutex);
    here = obtain(full);
    V(mutex);
    copy_buffer(here, next);
    P(mutex);
    release(here, emptyPool);
    V(mutex);
    /* Signal an empty buffer */
    V(empty);
    }  
}
```

Readers-Writers Problem

Readers-Writers Problem (2)

```c
Readers-Writers Problem (3)

Readers-Writers Problem (4)

First Solution

```c
reader() {  
  while(TRUE) {  
    <other computing>
    P(mutex);
    readCount++;
    if(readCount == 1)  
      P(writeBlock);
    V(mutex);
    /* Critical section */
    access(resource);
    P(mutex);
    readCount--;
    if(readCount == 0)  
      V(writeBlock);
    V(mutex);
  }
}

writer() {  
  while(TRUE) {  
    <other computing>
    P(writeBlock);
    /* Critical section */
    access(resource);
    V(writeBlock);
  }
}
```

• First reader competes with writers
• Last reader signals writers
First Solution (2)

```c
reader() {
while(TRUE) {
    <other computing>
    P(mutex);
    readCount++;
    if(readCount == 1) P(writeBlock);
    V(mutex);
    /* Critical section */
    access(resource);
    P(mutex);
    readCount--;
    if(readCount == 0) V(writeBlock);
    V(mutex);
}
}
resourceType *resource;
int readCount = 0;
semaphore mutex = 1;
semaphore writeBlock = 1;
fork(reader, 0);
fork(writer, 0);
```

Writer Precedence

```c
reader() {
while(TRUE) {
    <other computing>
    P(readBlock);
    P(mutex);
    readCount++;
    if(readCount == 1) P(writeBlock);
    V(mutex);
    V(readBlock);
    /* Critical section */
    access(resource);
    P(mutex);
    readCount--;
    if(readCount == 0) V(writeBlock);
    V(mutex);
}
}
```

Writer Precedence (2)

```c
reader() {
while(TRUE) {
    <other computing>
    P(writePending);
    P(mutex);
    readCount++;
    if(readCount == 1) P(writeBlock);
    V(mutex);
    V(writePending);
    /* Critical section */
    access(resource);
    P(mutex);
    readCount--;
    if(readCount == 0) V(writeBlock);
    V(mutex);
}
}
```

The Sleepy Barber

- Barber can cut one person's hair at a time
- Other customers wait in a waiting room

```c
customer() {
while(TRUE) {
    customer = nextCustomer();
    if(emptyChairs == 0) continue;
    P(mutex);
    emptyChairs--;
    P(chair);
    P(mutex);
    emptyChairs--;
    takeCustomer();
    if this.name() = "B.Barker" {
        kill();
    }
    V(mutex);
    V(chair);
    }
}
semaphore mutex = 1, chair = N, waitingCustomer = 0;
fork(customer, 0);
fork(barber, 0);
```

Cigarette Smoker’s Problem

- Three smokers (processes)
- Each wish to use tobacco, papers, & matches
  - Only need the three resources periodically
  - Must have all at once
- 3 processes sharing 3 resources
  - Solvable!
Implementing Semaphores

- Minimize effect on the I/O system
- Processes are only blocked on their own critical sections (not critical sections that they should not care about)
- If disabling interrupts, be sure to bound the time they are disabled

Implementing Semaphores: enter() & exit()

class semaphore {
    int value;
    public:
        semaphore(int v = 1) { value = v; }
        P(){
            disableInterrupts();
            while(value == 0) {
                enableInterrupts();
                disableInterrupts();
            }
            value--;
            enableInterrupts();
        }
        V(){
            disableInterrupts();
            value++;
            enableInterrupts();
        }
};

Implementing Semaphores: Test and Set Instruction

- TS(m): [Reg_i = memory[m]; memory[m] = TRUE;]

(a) Before Executing TS
(b) After Executing TS

Using the TS Instruction

boolean s = FALSE;

. . .

while(TS(s)) {
    . . .
    P(s);
    . . .
    <critical section>
    s = FALSE;
    . . .
    <critical section>
    V(s);
    . . .

Active vs Passive Semaphores

- A process can dominate the semaphore
  - Performs V operation, but continues to execute
  - Performs another P operation before releasing the CPU
  - Called a passive implementation of V
- **Active** implementation calls scheduler as part of the V operation.
  - Changes semantics of semaphore!
  - Cause people to rethink solutions