Basic Synchronization Principles

Concurrency

• Value of concurrency – speed & economics
• But few widely-accepted concurrent programming languages (Java, C# 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”
Command Execution

Enter Loop
Another Command? → Exit Loop
Yes → fork() code → Execute Command
No → Wait for Child to Terminate

UNIX Shell

Windows Command Launch

Synchronizing Multiple Threads with a Shared Variable

Initialize
CreateThread(...) → Wait runTime seconds → Thread Work
runFlag=FALSE → Exit
runFlag=TRUE → TRUE

RunFlag? → FALSE
TRUE
FALSE
TRUE
FALSE
TRUE

Terminate
Exit

Slide 8-4
Traffic Intersections

Critical Sections

shared double balance;

Code for $p_1$

. . .
balance = balance + amount;
amount;
. . .

Code for $p_2$

. . .
balance = balance -
amount;
. . .
Critical Sections

Execution of $p_1$

... 
load R1, balance 
load R2, amount 

Timer interrupt

Execution of $p_2$

... 
load R1, balance 
load R2, amount 
sub R1, R2 
store R1, balance 

... 
add R1, R2 
store R1, balance 
... 

Critical Sections

- **Mutual exclusion**: Only one process can be in the critical section at a time
- There is a *race* to execute critical sections (race condition)
- 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
Critical Sections

- **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()`
    - Terminate processes with `QUIT()` to synchronize
    - Create processes whenever critical section is complete
- … 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) {NULL;}
lock = TRUE;
/* Execute critical sect */
balance = balance + amount;
/* Release lock */
lock = FALSE;

**Code for p₂**

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

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

/* Acquire the lock */
while(lock){NULL;}
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

```c
enter(lock) {
    disableInterrupts();
    /* Loop until lock is TRUE */
    while(lock) {
        /* Let interrupts occur */
        enableInterrupts();
        disableInterrupts();
    }
    lock = TRUE;
    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 …

shared int lock = FALSE;
shared double amount,balance;

**Code for p1**

```c
/* Acquire the lock */
enter(lock);
/* Execute critical sect */
balance = balance + amount;
/* Release lock */
exit(lock);
```

**Code for p2**

```c
/* Acquire the lock */
enter(lock);
/* Execute critical sect */
balance = balance - amount;
/* Release lock */
exit(lock);
```

- 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 …
Deadlocked Pirates

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 CS to update len */
   enter(lock2);
   <update length>;
/* Exit both CS */
   exit(lock1);
   exit(lock2);
   ...

Code for p₂
...
/* Enter CS to update len */
   enter(lock2);
   <update length>;

   <intermediate computation>
/* Enter CS to add elt */
   enter(lock1);
   <add element>;
/* Exit both CS */
   exit(lock2);
   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 CS */
exit(lock1);
<intermediate computation>;
/* Enter CS to update len */
enter(lock2);
<update length>;
/* Exit CS */
exit(lock2);
<intermediate computation>;

Code for p₂

/* Enter CS to update len */
enter(lock2);
<update length>;
/* Exit CS */
exit(lock2);
<intermediate computation>;
/* Enter CS to add elt */
enter(lock1);
<add element>;
/* Exit CS */
exit(lock1);

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
A Semaphore

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 $p_0$ & $p_1$ 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**:

```c
Proc_0() {
    while(TRUE) {
        <compute section>;
        <compute section>;
        <critical section>;
        <critical section>;
    }
}

proc_1() {
    while(TRUE) {
        <compute section>;
        <compute section>;
    }
}

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

  \[ V(s) : [s = s + 1] \]
  \[ P(s) : [\text{while}(s == 0) \{ \text{wait} \}; s = s - 1] \]
Solving the Canonical Problem

Proc_0() {
    while(TRUE) {
        <compute section>;
        P(mutex);
        <critical section>;
        V(mutex);
    }
}

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

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

Shared Account Balance Problem

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

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

proc_1() {
    . . .
    /* Enter the CS */
    P(mutex);
    balance -= amount;
    V(mutex);
    . . .
}
Sharing Two Variables

```
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);
```

proc_B() {
    while(TRUE) {
        /* Wait for proc_A */
        P(s1);
        retrieve(x);
        <compute section B1>;
        update(y);
        /* Signal proc_A */
        V(s2);
        <compute section B2>;
    }
}

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

Bounded Buffer Problem (2)

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) {
        /* Claim full buffer */
        P(mutex);
        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(next);
    }
}

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) {
        /* 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);
        consume_item(next);
    }
}

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);
```

Readers-Writers Problem

```
Readers

Writers

Readers

Writers
```
Readers-Writers Problem (2)

Readers-Writers Problem (3)
Readers-Writers Problem (4)

```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);
    }
}
```

resourceType *resource;
int readCount = 0;
semaphore mutex = 1;
semaphore writeBlock = 1;
fork(reader, 0);
fork(writer, 0);

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

- 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
- May not be what we want

Writer Precedence

```c
reader() {  
  while(TRUE) {  
    <other computing>;
    P(readBlock);
    P(mutex1);
    P(mutex);  
    readCount++;  
    if(readCount == 1)  
      P(writeBlock);
    V(mutex1);
    V(readBlock);
    P(mutex1);
    access(resource);
    P(mutex1);
    readCount--;
    if(readCount == 0)  
      V(writeBlock);
    V(mutex1);
  }  
}
int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;
fork(reader, 0);
fork(writer, 0);
```

```c
writer() {  
  while(TRUE) {  
    <other computing>;
    P(mutex2);
    writeCount++;
    if(writeCount == 1)  
      P(readBlock);
    V(mutex2);
    P(writeBlock);
    access(resource);
    V(writeBlock);
    P(mutex2);
    writeCount--;  
    if(writeCount == 0)  
      V(readBlock);
    V(mutex2);
  }  
}
```
Writer Precedence (2)

```c
reader() {
    while(TRUE) {
        <other computing>;
        P(writePending);
        P(readBlock);
        P(mutex1);
        readCount++;
        if(readCount == 1)
            P(writeBlock);
        V(mutex1);
        V(readBlock);
        V(writePending);
        access(resource);
        P(mutex1);
        readCount--;
        if(readCount == 0)
            V(writeBlock);
        V(mutex1);
    }
}

writer() {
    while(TRUE) {
        <other computing>;
        P(mutex2);
        P(writePending);
        if(writeCount == 1)
            P(readBlock);
        V(mutex2);
        P(writeBlock);
        access(resource);
        V(writeBlock);
        P(mutex2);
        writeCount--;
        if(writeCount == 0)
            V(readBlock);
        V(mutex2);
    }
}

int readCount = 0, writeCount = 0;
semaphore mutex = 1, mutex2 = 1;
semaphore readBlock = 1, writeBlock = 1, writePending = 1;
fork(reader, 0);
fork(writer, 0);
```

The Sleepy Barber

- Barber can cut one person’s hair at a time
- Other customers wait in a waiting room
Sleepy Barber (aka Bounded Buffer)

customer() {
    while (TRUE) {
        customer = nextCustomer();
        if (emptyChairs == 0)
            continue;
        P(chair);
        P(mutex);
        emptyChairs--;
        takeChair(customer);
        V(mutex);
        V(waitingCustomer);
    }
}

barber() {
    while (TRUE) {
        P(waitingCustomer);
        P(mutex);
        emptyChairs++;
        takeCustomer();
        V(mutex);
        V(chair);
    }
}

semaphore mutex = 1, chair = N, waitingCustomer = 0;
int emptyChairs = N;
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, but difficult
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:

```cpp
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;]

Using the TS Instruction

```java
boolean s = FALSE;

... while(TS(s)) ;
<critical section>
  s = FALSE;
...  P(s) ;
<critical section>
...  V(s) ;
...
```
Implementing the General Semaphore

```c
struct semaphore {
    int value = <initial value>;
    boolean mutex = FALSE;
    boolean hold = TRUE;
};

shared struct semaphore s;

P(struct semaphore s) {
    while(TS(s.mutex)) ;
    s.value--;
    if(s.value < 0) {
        s.mutex = FALSE;
        while(TS(s.hold)) ;
    } else
        s.mutex = FALSE;
}

V(struct semaphore s) {
    while(TS(s.mutex)) ;
    s.value++;
    if(s.value <= 0) {
        while(!s.hold) ;
        s.hold = FALSE;
    } s.mutex = FALSE;
}
```

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