CSCI 234

Design of Internet Protocols: Concurrency

George Blankenship

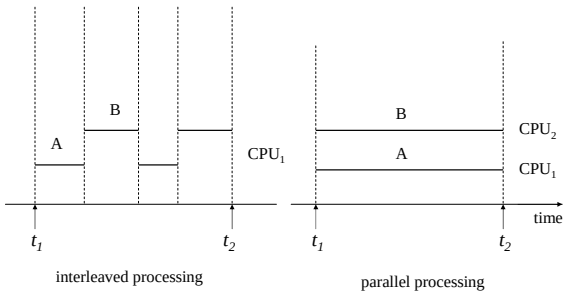
Outline

- Concurrent Processes
- Locks
- Synchronization
- Semaphores
- Producer/Consumer Algorithms

Introduction

- What is a Concurrent Process?
  - Multiple users access databases and use computer systems simultaneously.
  - An airline reservation system used by travel agents and reservation clerks concurrently.
- Why Concurrent Process?
  - Better transaction throughput and response time
  - Better utilization of resource

Concurrent Transactions



Oops, something's wrong

- Reserving a seat for a flight
- If concurrent access to data in DBMS, two users may try to book the same seat simultaneously

time

Agent 1 finds seat 35G empty

Agent 2 finds seat 35G empty

Agent 1 sets seat 35G occupied

Agent 2 sets seat 35G occupied

Debit Transactions

- Problems can occur when concurrent transactions execute in an uncontrolled manner.
- A, originally equals to 100, after the execution of T1 and T2, A is supposed to be 100+10-8=102

Add 10 To A	Minus 8 from A
T1	**T2**
Read(A)	
A=A+10	
	Read(A)
	A=A-8
Write(A)	
	Write(A)

Value of A on the disk
100
100
100
100
110
92

Concurrency Control Through Locks

- Lock: variable associated with each data item
  - Describes status of item with regard to operations that can be performed on it
- Binary locks:
  - Locked
  - unlocked
- Multiple-mode locks:
  - Read lock
  - Write lock
  - Unlocked
- Each data item can be in only one of the lock states

Concurrency George Blankenship 7

Two Operations

T1	T2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Let's assume serial schedule S1: T1;T2

Initial values: X=20, Y=30 → Result: X=50, Y=80

Concurrency George Blankenship 8

Locks Alone Don't Do the Trick!

Let's run T1 and T2 in interleaved fashion

Schedule S

T1	T2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y:=X+Y;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
X:=X+Y;	
write_item(X);	
unlock(X);	

unlocked too early!

Non-serializable! Result: X=50, Y=50

Concurrency George Blankenship 9

Basic Two-Phase Locking

- Consistent locking order imperative to avoid concurrency problems
- Two phases to execution schedules
  - Phase I – acquire locks
  - Phase II – unlock locks
  - Once Phase II has begun, no locks may be acquired
- Residual issues
  - reduction of concurrency
  - Deadlock

Example

T1’
read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y);
read_item(X);
X:=X+Y;
write_item(X);
unlock(X);

T2’
read_lock(X);
read_item(X);
write_lock(Y);
unlock(X);
read_item(Y);
Y:=X+Y;
write_item(Y);
unlock(Y);

- Both T1’ and T2’ follow the 2PL protocol
- Any schedule including T1’ and T2’ is guaranteed to be serializable
- Limits the amount of concurrency

Conservative Two-Phase Locking

- Lock all items needed BEFORE execution begins by predeclaring its read and write set
- If any of the items in read or write set is already locked (by other transactions), transaction waits (does not acquire any locks)
- Deadlock free but not very realistic

Strict Two-Phase Locking

- Transaction does not release its write locks until AFTER it aborts/commits
- Not deadlock free but guarantees recoverable schedules
- strict schedule: transaction can neither read/write X until last transaction that wrote X has committed/aborted
- Most popular variation of two-phase locking

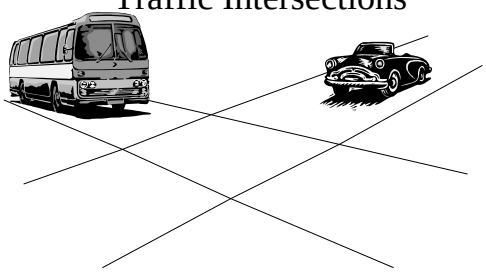
Concurrency George Blankenship 13

Residual Issues

- Concurrency control subsystem is responsible for inserting locks at right places into your transaction
  - Strict two-phase locking is widely used
  - Requires use of waiting queue
- All two-phase locking protocols guarantee serializability
- Does not permit all possible serial schedules

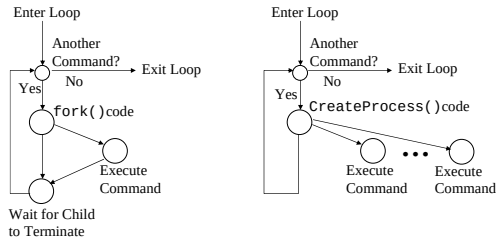
Concurrency George Blankenship 14

Traffic Intersections



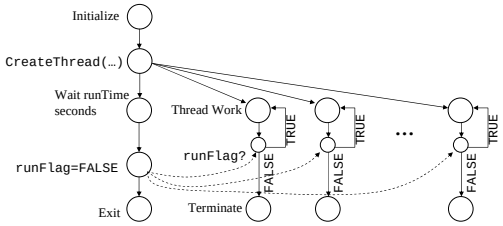
Concurrency George Blankenship 15

Command Execution



(a) UNIX Shell (b) Windows Command Launch

Synchronizing Multiple Threads with a Shared Variable



Critical Sections

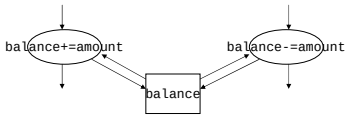
```
shared double balance;
```

Code for p_1

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

Code for p_2

```
 . . .
balance = balance - amount;
 . . .
```



Time Slice Execution

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
…
```

Timer interrupt

```
add   R1, R2
store R1, balance
…
```


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

Disabling Interrupts

```
shared double balance;
```

Code for p_1

```
disableInterrupts();
balance = balance + amount;
enableInterrupts();
```

Code for p_2

```
disableInterrupts();
balance = balance - amount;
enableInterrupts();
```

- Interrupts could be disabled arbitrarily long
- Really only want to prevent p_1 and p_2 from interfering with one another; this blocks all p_i
- Try using a shared “lock” variable

Using a Lock Variable

```
shared boolean lock = FALSE;
shared double balance;
```

Code for p_1

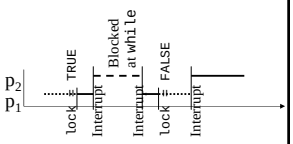
```
/* Acquire the lock */
  while(lock) ;
  lock = TRUE;
/* Execute critical sect */
  balance = balance + amount;
/* Release lock */
  lock = FALSE;
```

Code for p_2

```
/* 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 p_1

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

Code for p_2

```
/* 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 p_1

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

Code for p_2

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

```
exit(lock) {
  disableInterrupts();
  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 …

Concurrency George Blankenship 25

Processing Two Components

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

Code for p_1

```
  . . .
/* 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);
  . . .
```

Code for p_2

```
  . . .
/* 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);
  . . .
```

Concurrency George Blankenship 26

Deadlock from Strict Two-Phase Locking

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

Code for p_1

```
  . . .
/* 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 p_2

```
  . . .
/* 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);
  . . .
```

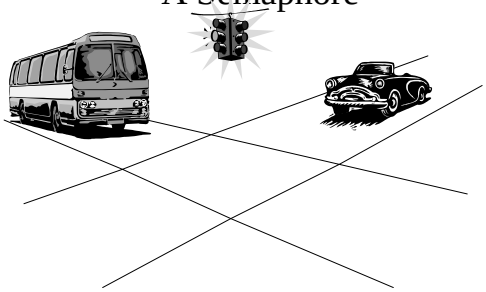
Concurrency George Blankenship 27

Atomic 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

Concurrency George Blankenship 28

A Semaphore



Concurrency George Blankenship 29

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

Concurrency George Blankenship 30

Solution Constraints

- Processes P_0 and P_1 enter critical sections
- Mutual exclusion
  - Only one process at a time in the critical section
  - Only processes competing for a critical section are involved in resolving who enters the section
- Fairness
  - Once a process attempts to enter its critical section, it cannot be postponed indefinitely
  - After requesting entry, only a bounded number of other processes may enter before the requesting process

Concurrency George Blankenship 31

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

Concurrency George Blankenship 32

Dijkstra Semaphore Definition

- Classic paper describes several software attempts to solve the problem
  - Offered a software solution
  - 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]
```

Concurrency George Blankenship 33

Solving the Canonical Problem

```
Proc_0() {                      proc_1() {
  while(TRUE) {                   while(TRUE {
    <compute section>;              <compute section>;
    P(mutex);                       P(mutex);
      <critical section>;             <critical section>;
    V(mutex);                       V(mutex);
  }                               }
}                               }

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

Concurrency George Blankenship 34

Shared Account Balance Problem

```
Proc_0() {                      proc_1() {
  . . .                           . . .
/* Enter the CS */              /* Enter the CS */
  P(mutex);                       P(mutex);
    balance += amount;              balance -= amount;
  V(mutex);                       V(mutex);
  . . .                           . . .
}                               }

semaphore mutex = 1;

fork(proc_0, 0);
fork(proc_1, 0);
```

Concurrency George Blankenship 35

Sharing Two Variables

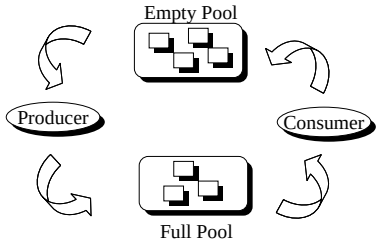
```
proc_A() {                      proc_B() {
  while(TRUE) {                   while(TRUE) {
    <compute section A1>;         /* Wait for proc_A */
    update(x);                      P(s1);
  /* Signal proc_B */               retrieve(x);
    V(s1);                          <compute section B1>;
    <compute section A2>;           update(y);
  /* Wait for proc_B */           /* Signal proc_A */
    P(s2);                          V(s2);
    retrieve(y);                    <compute section B2>;
  }                               }
}                               }    semaphore s1 = 0;
                                     semaphore s2 = 0;
                                     fork(proc_A, 0);
                                     fork(proc_B, 0);
```

Concurrency George Blankenship 36

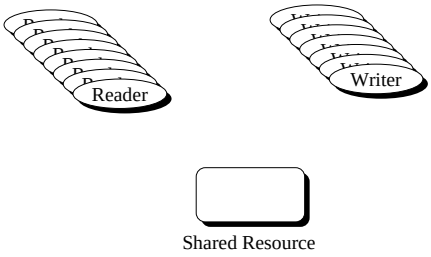
Device Controller Synchronization

- 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



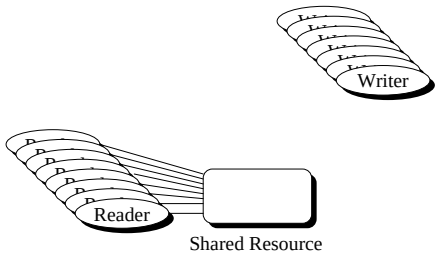
Readers-Writers Problem



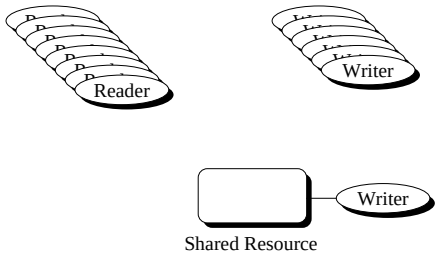
Readers/Writers Problem

- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it

Readers Can Share Resource



Writers Must Have Exclusive Use

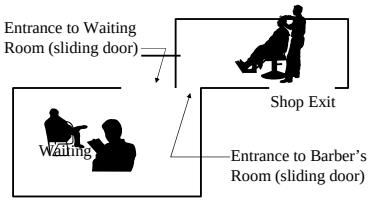


Solution Parameters

- First reader and writers compete for resource
- Last reader returns resource to open competition
- Writers must wait for all readers to finish
- Readers can starve writers
- Updates can be delayed without bound
- Need to give write precedence

The Sleepy Barber

- Barber can cut one person's hair at a time
- Other customers wait in a waiting room
- Barber might not check waiting room



Cigarette Smoker's Problem

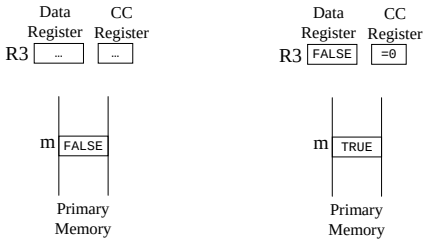
- Three smokers (processes)
- Each wish to use tobacco, papers, & matches
  - Only need the three resources periodically
  - Must have all at once
- 3 processes sharing 3 resources
  - Solvable, but difficult

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: 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)) ;
  <critical section>
  s = FALSE;
  . . .
```

```
semaphore s = 1;
  . . .
  P(s) ;
  <critical section>
  V(s);
  . . .
```

General Semaphore

```
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;
}
```

Concurrency George Blankenship 49

Active/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

Concurrency George Blankenship 50

Producer/Consumer Problem

- One or more producers are generating date and placing these in a buffer
- A single consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time

Concurrency George Blankenship 51

Buffer Based Solution

- Two semaphores are used
  - one to represent the amount of items in the buffer
  - one to signal that it is all right to use the buffer
- Set flag to enter critical section before check other processes
- If another process is in the critical section when the flag is set, the process is blocked until the other process releases the critical section
- Deadlock is possible when two process set their flags to enter the critical section. Now each process must wait for the other process to release the critical section

Concurrency George Blankenship 52

Fairness

- Each process gets a turn at the critical section
- If a process wants the critical section, it sets its flag and may have to wait for its turn

Concurrency George Blankenship 53

Producer Function

```
producer:
repeat
  produce item v;
  b[in] := v;
  in := in + 1
forever;
```

Concurrency George Blankenship 54

Consumer Function

```
consumer:
repeat
  while in <= out do { nothing
  };
  w := b[out];
  out := out + 1;
  consume item w
forever;
```

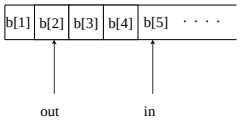

Producer with Circular Buffer

```
producer:
repeat
  produce item v;
  while ( (in + 1) mod n = out)
  do { nothing };
  b[in] := v;
  in := (in + 1) mod n
forever;
```


Consumer with Circular Buffer

```
consumer
repeat
  while in = out do { nothing
  };
  w := b[out];
  out := (out + 1) mod n;
  consume item w
forever;
```

Infinite Buffer



Note: shade area indicates portion of buffer that is occupied

Message Passing

- Enforce mutual exclusion
- Exchange information

```
send (destination, message)
receive (source, message)
```


Message Passing - Synchronization

- Sender and receiver may or may not be blocking (waiting for message)
- Blocking send, blocking receive
  - both sender and receiver are blocked until message is delivered
  - called a rendezvous

Message Passing - Synchronization

- Nonblocking send, blocking receive
  - sender continues processing such as sending messages as quickly as possible
  - receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive

Concurrency George Blankenship 61

Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive could know ahead of time which process a message is expected
- Receive primitive could use source parameter to return a value when the receive operation has been performed

Concurrency George Blankenship 62

Indirect Addressing

- Messages are sent to a shared data structure consisting of queues
- Queues are called mailboxes
- One process sends a message to the mailbox and the other process picks up the message from the mailbox

Concurrency George Blankenship 63

General Message Format