Semaphores

A semaphore is an object that contains a nonnegative integer variable $S$ and two atomic operations: $P$ and $V$

- $P(S):$ while $(S \leq 0)$ ;
  
  $S = S - 1;$

- $V(S):$ $S = S + 1;$

Implementing Critical Sections

```c
semaphor mutex = 1; /* binary semaphore */

P(mutex); /* entryProtocol */
critical section;
V(mutex); /* exitProtocol */
```

Implementing Resource Allocation

```c
semaphor count = n; /* counting semaphore */

P(mutex); /* entryProtocol */
resource usage;
V(mutex); /* exitProtocol */
```
Implementing Semaphores

- Semaphores can be implemented at the user level using busy waiting or at the kernel level using system calls.

Kernel Level Implementation

\[ P(S) : \text{trap to the kernel;} \]
\[ \quad \text{disable interrupts;} \]
\[ \quad \text{if } (S > 0) S = S - 1; \]
\[ \quad \text{else} \{ \]
\[ \quad \quad \text{queue the thread on } S; \]
\[ \quad \quad \text{change its state to blocked;} \]
\[ \quad \quad \text{schedule another thread;} \]
\[ \quad \} \]
\[ \quad \text{enable interrupts;} \]
\[ \quad \text{return;} \]

Kernel Level Implementation

\[ V(S) : \text{trap to the kernel;} \]
\[ \quad \text{disable interrupts;} \]
\[ \quad \text{if } (S == 0 && \text{queue on } S \text{ is not empty}) \{ \]
\[ \quad \quad \text{pick a thread from the queue on } S; \]
\[ \quad \quad \text{change its state from blocked to ready;} \]
\[ \} \text{ else } S = S + 1; \]
\[ \quad \text{enable interrupts;} \]
\[ \quad \text{return;} \]

Java Semaphores

- Java only supports implicit binary semaphores.
- Each Java object has a binary semaphore.
- A block of code can be made a critical section using the keyword synchronized.
\[ \text{synchronized (object) \{ \]
\[ \quad \text{critical section;} \]
\[ \} \]
Java Semaphores

type method (...){
    synchronized (this) {
        critical section;
    }
}
synchronized type method (...){
    critical section;
}

An Example

private long fn(long j, int k) {
    long total = j;
    for (int i = 1; i <= k; i++) total += i;
    return total;
}

public void run() {
    for (int m = 1; m <= M; m++)
        synchronized (this) {
            sum = fn(sum, m);
        }
}

Synchronizations

- Mutual exclusion: enforcing the mutual exclusion of critical sections
  semaphor mutex = 1;
  t; P(mutex);
  critical section;
  t; V(mutex);

- Condition synchronization: delaying the execution of a thread until some condition is satisfied
  semaphor mutex = 0;
  t; if (!condition) P(mutex);
  t; V(mutex);
Semaphore Classes

- The author provides two semaphore classes: BinarySemaphore and CountingSemaphore
- These two semaphore classes support two operations: P() and V()
- The author provides a class MyObject that support two operations: P(semaphore) and V(semaphore)

An Example

- This program starts three threads, Pa, Pb, Pc
- Pa repeatedly prints “A”, Pb “B”, Pc “C”
- Use two binary semaphores and one counting semaphore such that
- A “B” must be output before any “C”
- “B” and “C” must alternate in output
- The total number of “B” and “C” cannot exceed the number of “A” at any time

An Example

```java
class ABCs extends MyObject {
    protected static final BinarySemaphore B = new BinarySemaphore(0);
    protected static final BinarySemaphore C = new BinarySemaphore(1);
    protected static final CountingSemaphore sum = new CountingSemaphore(0);
    // main()
}
```

```java
public static void main(String[] args) {
    Thread pa = new Thread(new Pa());
    Thread pb = new Thread(new Pb());
    Thread pc = new Thread(new Pc());
    pa.start(); pb.start(); pc.start();
    nap(9000);
    pa.stop(); pb.stop(); pc.stop();
    System.out.println();
    System.out.println("B = " + B + ", C = " + C + ", sum = " + sum);
    System.exit(0);
}
```
An Example

class Pa extends ABCs implements Runnable {
    public void run () {
        while (true) { nap(1+(int)(random(500)));
            System.out.print("A"); System.out.flush();
            V(sum);
        }
    }
}

An Example

class Pb extends ABCs implements Runnable {
    public void run () {
        while (true) { nap(1+(int)(random(800)));
            P(C); P(sum);
            System.out.print("B"); System.out.flush();
            V(B);
        }
    }
}

An Example

class Pc extends ABCs implements Runnable {
    public void run () {
        while (true) { nap(1+(int)(random(800)));
            P(B); P(sum);
            System.out.print("C"); System.out.flush();
            V(C);
        }
    }
}

Classical Problems

- Bounded Buffer
- Sleeping barber
- Dining philosophers
- Readers and writers
Bounded Buffer

- This problem involves a buffer with a finite number of slots
- A producer places items into the buffer and a consumer takes items out of the buffer
- The producer suspends when the buffer is full and the consumer suspends when the buffer is empty
- There is only one thread accessing a buffer slot at a time

```java
class BoundedBuffer extends MyObject {
    private int numSlots = 0;
    private double[] buffer = null;
    private int putIn = 0, takeOut = 0;
    private int count = 0;
    private BinarySemaphore mutex = null;
    private CountingSemaphore elements = null;
    private CountingSemaphore spaces = null;
    // BoundedBuffer()
    // deposit()
    // fetch()
}
```

```java
public BoundedBuffer(int numSlots) {
    super("BoundedBuffer with " + numSlots + " slots");
    if (numSlots <= 0)
        throw new IllegalArgumentException("numSlots<=0");
    this.numSlots = numSlots;
    buffer = new double[numSlots];
    mutex = new BinarySemaphore(1);
    elements = new CountingSemaphore(0);
    spaces = new CountingSemaphore(numSlots);
    System.out.println("BoundedBuffer alive, numSlots=" + numSlots);
}
```

```java
public void deposit(double value) {
    P(spaces);
    buffer[putIn] = value;
    putIn = (putIn + 1) % numSlots;
    P(mutex);
    count++;
    System.out.println(" after deposit, count=" + count + ", putIn=" + putIn);
    V(mutex);
    V(elements);
}
```
public double fetch() {
    double value;
    P(elements);
    value = buffer[takeOut];
    takeOut = (takeOut + 1) % numSlots;
    P(mutex);
    count--;
    System.out.println(" after fetch, count= " + count
    + ", takeOut=" + takeOut);
    V(mutex);
    V(spaces);
    return value;
}

Bounded Buffer

class Producer extends MyObject implements Runnable {
    private int pNap = 0;
    private BoundedBuffer bb = null;

    public Producer(String name, int pNap, BoundedBuffer bb)
    {
        super(name);
        this.pNap = pNap;
        this.bb = bb;
    }
    // run()
}

Bounded Buffer

public void run() {
    double item;
    int napping;
    while (true) {
        napping = 1 + (int) random(pNap);
        nap(napping);
        item = random();
        bb.deposit(item);
    }
}

Bounded Buffer

class Consumer extends MyObject implements Runnable {
    private int cNap = 0;
    private BoundedBuffer bb = null;

    public Consumer(String name, int cNap, BoundedBuffer bb)
    {
        super(name);
        this.cNap = cNap;
        this.bb = bb;
    }
    // run()
}
Parallel Programming

**Bounded Buffer**

```java
public void run() {
    double item;
    int napping;
    while (true) {
        napping = 1 + (int) random(cNap);
        nap(napping);
        item = bb.fetch();
    }
}
```

**Using Bounded Buffers**

```
sin(sqrt(x)) * sin(sqrt(x))
A: sqrt(x) → B: sin(x) → C: x * x
```

```java
public void run() { // thread A
double work;
    for (int i = 0; i < limit; i++) {
        work = Math.sqrt(work);
        nap(1+(int)random(2000)); //ABbb.deposit(work);
    }
}
```

**Bounded Buffer**

```java
class ProducerConsumer extends MyObject {
    public static void main(String[] args) {
        int numSlots = 20;
        int pNap = 3, cNap = 3, runTime = 60;
        BoundedBuffer bb = new BoundedBuffer(numSlots);
        Thread producer = new Thread(new Producer("PRODUCER", pNap*1000, bb));
        Thread consumer = new Thread(new Consumer("Consumer", cNap*1000, bb));
        producer.start(); consumer.start();
        nap(runTime*1000);
        producer.stop(); consumer.stop();
        System.exit(0);
    }
}
```

**Using Bounded Buffers**

```
public void run() { // thread B
double work;
    for (int i = 0; i < limit; i++) {
        work = ABBb.fetch(); work = Math.sin(work);
        nap(1+(int)random(2000)); //BCbb.deposit(work);
    }
}
```

```java
public void run() { // thread C
double work;
    for (int i = 0; i < limit; i++) {
        work = BCbb.fetch(); work = work * work;
        nap(1+(int)random(2000));
    }
}
```
Unbounded Buffer

class QueueItem extends MyObject {
    protected double value = 0;
    protected QueueItem nextItem = null;

    protected QueueItem() { super(); }
    protected QueueItem(double value) {
        this.value = value;
    }
}

Unbounded Buffer

class UnboundedBuffer extends QueueItem {
    private int count = 0;
    private CountingSemaphore elements =
        new CountingSemaphore(0);
    private BinarySemaphore mutex =
        new BinarySemaphore(1);
    private QueueItem head = null;
    private QueueItem tail = null;
    // deposit()
    // fetch()
}

Unbounded Buffer

public void deposit(double value) throws CorruptedQueueException {
    P(mutex);
    QueueItem oldTail = null;
    QueueItem newOne = new QueueItem(value);
    if (tail == null) {
        if (head != null) throw new CorruptedQueueException();
        else { head = newOne; tail = newOne; }
    } else {
        oldTail = tail; tail = newOne; oldTail.nextItem = newOne;
    }
    count++;
    V(mutex);
    V(elements);
}

Unbounded Buffer

public double fetch() throws CorruptedQueueException {
    double value; QueueItem oldOne = null;
    P(elements);
    P(mutex);
    if (head == null & & tail == null)
        throw new CorruptedQueueException();
    else if (head != null & & tail != null) {
        if (head == tail) { oldOne = head; head = null; tail = null; }
        else { oldOne = head; head = oldOne.nextItem; }
        count--;
    } else throw new CorruptedQueueException();
    value = oldOne.value;
    V(mutex);
    return value;
}
Parallel Programming

Sleeping Barber

- This problem involves a **barber chair** and several **waiting chairs** in a barber shop
- When the barber finishes a customer’s haircut, he fetches the next customer in the waiting chairs if there is one; otherwise, he waits for new customers
- A **customer** goes to the barber chair and wakes up the barber if no other customers in the shop
- A **customer** goes to a waiting chair if there are other customers and there are waiting chairs left
- A **customer** returns and comes back later if there is no waiting chair left

```
Sleeping Barber
class Barber extends MyObject implements Runnable {
    private Salon salon = null;
    public Barber(String name, Salon salon) {
        super(name);
        this.salon = salon;
        new Thread(this).start();
    }
    public void run() {
        while (true) {
            salon.wantToCut(); // service the next customer
        }
    }
}
```

```
Sleeping Barber
class Customer extends MyObject implements Runnable {
    private int id = 0;
    private Salon salon = null;
    private int cutTime = 0, growTime = 0;
    public Customer(String name, int id, int cutTime,
        int growTime, Salon salon) {
        super(name + " " + id);
        this.id = id;
        this.salon = salon;
        this.cutTime = cutTime; this.growTime = growTime;
        new Thread(this).start();
    }
    // run()
}
```

```
Sleeping Barber
public void run() {
    int napping;
    while (true) {
        napping = 1 + (int)random(growTime);
        nap(napping);
        salon.wantHairCut(id, cutTime);
    }
}
```
**Sleeping Barber**

```java
class Salon extends MyObject {
    private int numChairs = 0;  private int waiting = 0;
    private CountingSemaphore customers = null;
    private CountingSemaphore barber = null;
    private BinarySemaphore mutex = null;
    private CountingSemaphore cutting = null;
    public Salon(String name, int numChairs) {
        super(name); this.numChairs = numChairs;
        customers = new CountingSemaphore(0);
        barber = new CountingSemaphore(0);
        mutex = new BinarySemaphore(1);
        cutting = new CountingSemaphore(0);
    }  // methods
}

public void wantToCut() {
P(customers);  // wait for new customers
P(mutex);
waiting--;
V(barber);  // fetch the next customer
V(mutex);
P(cutting);  // wait for the finish of haircut
}
```

```java
class SleepingBarber extends MyObject {
    public static void main(String[] args) {
        int numChairs = 5;  int numCustomers = 10;
        int cutTime = 2; int growTime = 4;  int runTime = 60;
        Salon salon = new Salon("Salon", numChairs);
        new Barber("Barber", salon);
        for (int i = 0; i < numCustomers; i++)
            new Customer("Customer", i, cutTime*1000,
            growTime*1000, salon);
        nap(runTime*1000);
        System.exit(0);
    }
}
```

**Sleeping Barber**

```java
class SleepingBarber {
    public void wantHairCut(int i, int cutTime) {
        int napping;
        P(mutex);
        if (waiting < numChairs) {
            waiting++;
            V(customers);  // new customer comes in
            V(mutex);
            P(barber);  // wait for haircut
            napping = 1 + (int)random(cutTime);
            nap(napping);
            V(cutting);  // finish haircut
        } else {
            V(mutex);
        }
    }
}
```
Dining Philosophers

- This problem involves five philosophers sitting at a round table
- Five chopsticks interleave between them
- A bowl of spaghetti sits in the center of table
- Philosophers think for a while, then try to eat
- Philosophers eat a while, then ready to think
- A philosopher needs two chopsticks to eat

Dining Philosophers

- Only one philosopher at a time uses a chopstick
- A philosopher cannot hold only one chopstick and refuse to release it
- A hungry philosopher eats if the two chopsticks aside him are not used by others
- No philosopher starves forever

```java
class DiningServer extends MyObject {
    private static final int THINKING = 0, HUNGRY = 1, EATING = 2;
    private int numPhilhs = 0;
    private int[] state = null;
    private BinarySemaphore[] self = null;
    private BinarySemaphore mutex = null;
    public DiningServer(int numPhilhs) {
        super("DiningServer for " + numPhilhs + " philosophers");
        this.numPhilhs = numPhilhs;
        state = new int[numPhilhs];
        for (int i = 0; i < numPhilhs; i++) state[i] = THINKING;
        self = new BinarySemaphore[numPhilhs];
        for (int i = 0; i < numPhilhs; i++) self[i] = new BinarySemaphore(0);
        mutex = new BinarySemaphore(1);
    }

    private final int left(int i) { return (numPhilhs + i - 1) % numPhilhs; }
    private final int right(int i) { return (i + 1) % numPhilhs; }

    public void takeForks(int i) {
        P(mutex);
        state[i] = HUNGRY;
        test(i);
        V(mutex);
        P(self[i]);
    }
```
Dining Philosophers

public void putForks(int i) {
    P(mutex);
    state[i] = THINKING; test(left(i)); test(right(i));
    V(mutex);
}

private void test(int k) {
    if (state[left(k)] != EATING && state[right(k)] != EATING && state[k] == HUNGRY) {
        state[k] = EATING;
        V(self[k]);
    }
}
}

Dining Philosophers

class Philosopher extends MyObject implements Runnable {
    private int id = 0;  private int napThink = 0;  private int napEat = 0;  private DiningServer ds = null;

    public Philosopher(String name, int id, int napThink, int napEat, DiningServer ds) {
        super(name + " " + id);
        this.id = id;
        this.napThink = napThink;
        this.napEat = napEat;
        this.ds = ds;
        new Thread(this).start();
    }
}

Dining Philosophers

public void run() {
    while (true) {
        think();
        ds.takeForks(id);
        eat();
        ds.putForks(id);
    }
}

Dining Philosophers

private void think() {
    int napping;
    napping = 1 + (int) random(napThink);
    nap(napping);
}

private void eat() {
    int napping;
    napping = 1 + (int) random(napEat);
    nap(napping);
}
Readers and Writers

- This problem involves concurrent accesses to a database
- Multiple readers may read a database simultaneously as long as no writer writes
- Only one writer at a time may write
- Prevent starvation

---

class Database extends MyObject {
    private int numReaders = 0;
    private BinarySemaphore mutex = new BinarySemaphore(1);
    private BinarySemaphore ok = new BinarySemaphore(1);
    public Database() { super("Database"); }
    public void startRead(int i) {
        P(mutex);
        numReaders++;
        if (numReaders == 1) P(ok);
        V(mutex);
    }
    // methods
}

Readers and Writers

public void endRead(int i) {
    P(mutex);
    numReaders--;
    if (numReaders == 0) V(ok);
    V(mutex);
}
public void startWrite(int i) {
    P(ok);
}
public void endWrite(int i) {
    V(ok);
}

---

class Reader extends MyObject implements Runnable {
    private int id = 0;
    private int rNap = 0;
    private Database db = null;
    public Reader(String name, int id, int rNap, Database db) {
        super(name + id);
        this.id = id;
        this.rNap = rNap;
        this.db = db;
        new Thread(this).start();
    }
    // run()
}
Readers and Writers

```java
public void run() {
    int napping;
    while (true) {
        napping = 1 + (int) random(rNap);
        nap(napping);
        db.startRead(id);
        napping = 1 + (int) random(rNap);
        nap(napping);
        db.endRead(id);
    }
}
```

Readers and Writers

```java
class Writer extends MyObject implements Runnable {
    private int id = 0;
    private int wNap = 0;
    private Database db = null;
    public Writer(String name, int id, int wNap, Database db) {
        super(name + id);
        this.id = id;
        this.wNap = wNap;
        this.db = db;
        new Thread(this).start();
    }
    // run()
}
```

Readers and Writers

```java
public void run() {
    int napping;
    while (true) {
        napping = 1 + (int) random(wNap);
        nap(napping);
        db.startWrite(id);
        napping = 1 + (int) random(wNap);
        nap(napping);
        db.endWrite(id);
    }
}
```

Counting Semaphores from Binary

```java
class CountingSemaphoreFromBinary extends MyObject {
    private int count = 0;
    private BinarySemaphore mutex = new BinarySemaphore(1);
    private BinarySemaphore blocked = new BinarySemaphore(0);
    public CountingSemaphoreFromBinary(int n) {
        count = n;
    }
    // down() and up()
}
```
public void down() {
    P(mutex);
    count--;
    if (count < 0) {
        V(mutex);
        P(blocked);
    } else {
        V(mutex);
    }
}

public void up() {
    P(mutex);
    count++;
    if (count <= 0) {
        V(mutex);
        P(blocked);
    } else {
        V(mutex);
    }
}

class CountingSemaphoreFromBinary extends MyObject {
    private int count = 0;
    private BinarySemaphore mutex = new BinarySemaphore(1);
    private BinarySemaphore blocked = new BinarySemaphore(0);
    private BinarySemaphore serial = new BinarySemaphore(1);
    public CountingSemaphoreFromBinary(int n) {
        count = n;
    }
    // down() and up()
}