Weather Prediction Using Semaphores

# Members:

# Rohit Kriplani

Namita Nair

Rohan Keswani

**Question:**

A weather station computer system automatically predicts the weather based on input from three measuring devices. Each measuring device is running a separate process that performs some observation and when finished, places the data into a Shared buffer. When all three measuring processes have placed their observations into the buffer a calculation process is signaled. The calculation process takes the observations,performs some calculation and predicts the weather. All 4 started at same time.All measuring processes use the following code.

main(){

Shared buffer \*buffer;

Result \*results=make\_observation();

add\_observation(buffer,results);

}

The calculation process uses the following code.

main(){

Shared buffer \*buffer;

Result \*results1,\*results2,\*results3;

get\_results(buffer,results1,results2,results3);

calculate(results1,results2,results3);

}

Using semaphore modify the above code so that mutual exclusion on shared buffer is guaranteed and it forces the calculation process to wait for observation processes to finish.

**Theory:**

**Semaphores:**

In programming, especially in Unix systems, semaphores are a technique for coordinating or synchronizing activities in which multiple processes compete for the same operating system resources. A semaphore is a value in a designated place in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.  
  
Semaphores are commonly use for two purposes: to share a common memory space and to share access to files. Semaphores are one of the techniques for interprocess communication (IPC). The C programming language provides a set of interfaces or "functions" for managing semaphores.

**Binary semaphores** are binary, they can have two values only; one to represent that a process/thread is in the critical section(code that access the shared resource) and others should wait, the other indicating the critical section is free.

The max value that a **counting semaphore** can take is the the number of processes you want to allow into the critical section at the same time.

Again, you might have a case where you want exclusion over a certain resource, yet you know this resource can be accessed by a max number of processes (say X), so you set a counting semaphore with the value X.

This would allow the X processes to access that resource at the same time; yet the process X+1 would have to wait till one of the processes in the critical section gets out.

A **mutex** is essentially the same thing as a binary semaphore and sometimes uses the same basic implementation. The differences between them are in how they are used. While a binary semaphore may be used as a mutex, a mutex is a more specific use-case, in that only the thread that locked the mutex is supposed to unlock it. This constraint makes it possible to implement some additional features in mutexes:

1. Since only the thread that locked the mutex is supposed to unlock it, a mutex may store the id of thread that locked it and verify the same thread unlocks it.
2. Mutexes may provide [priority inversion](https://en.wikipedia.org/wiki/Priority_inversion) safety. If the mutex knows who locked it and is supposed to unlock it, it is possible to promote the priority of that thread whenever a higher-priority task starts waiting on the mutex.
3. Mutexes may also provide deletion safety, where the thread holding the mutex cannot be accidentally deleted.
4. Alternately, if the thread holding the mutex is deleted (perhaps due to an unrecoverable error), the mutex can be automatically released.
5. A mutex may be recursive: a thread is allowed to lock it multiple times without causing a deadlock.

**Pseudo code:**

Global definitions are :

init( mutex, 1);

init( ready, 0);

The code for the measuring process:

void measuring()

{

Shared\_Buffer \*buffer;

Result \*results;

Semaphore mutex, ready;

results=make\_observation();

wait(mutex)

add\_observation(buffer, results);

signal( mutex);

signal( ready);

}

The calculation process uses the following code:

void calculation()

{

Shared\_Buffer \*buffer;

Results \*results1, \*results2, \*results3;

Semaphore mutex, ready;

wait ( ready );

wait ( ready );

wait ( ready );

wait ( mutex );

get\_results(buffer, results1, results2, results3 );

signal( mutex );

Calculate (results1, results2, results3 );

}

**Code:**

import java.io.\*;

import java.lang.\*;

import java.util.concurrent.Semaphore;

class osmini1

{

public static void main(String args[])

{

int i;

Weather w[]=new Weather[4];

for(i=0;i<3;i++)

w[i]=new Weather(i,"machine");

w[3]=new Weather(3,"calculator");

for(i=3;i>=0;i--)

w[i].start();

}

}

class Weather extends Thread

{

static Semaphore mutex=new Semaphore(1);

static Semaphore ready=new Semaphore(0);

static int buffer[]=new int[4];

int id;

String type;

static int Mcount=0;

Weather(int i,String s)

{

id=i;

type=s;

}

public void run()

{

if(type=="machine")

{

machine();

}

else

{

calculator();

}

}

public void machine()

{

try{

mutex.acquire();

buffer[Mcount]=1;

Mcount++;

addObservation();

mutex.release();

ready.release();

finished();

}

catch (InterruptedException e)

{

System.out.println(e.getMessage());

}

}

public void calculator()

{

while(Mcount!=3)

{

System.out.println("Calculator is waiting.");

try

{

ready.acquire();

}

catch (InterruptedException e)

{

System.out.println(e.getMessage());

}

}

try

{

mutex.acquire();

getResults();

mutex.release();

}

catch (InterruptedException e)

{

System.out.println(e.getMessage());

}

}

public void addObservation()

{

System.out.println("Machine "+ id + " is processing.");

}

public void getResults()

{

System.out.println("Calculator is taking values from the buffer.");

System.out.println("Performing calculations.");

try

{

Thread.sleep(1000);

}

catch (InterruptedException e)

{

System.out.println(e.getMessage());

}

System.out.println("Output predicted.");

}

public void finished()

{

System.out.println("Machine "+ id + " has finished processing. Data placed inside buffer.");

}

}

**Output:**



