Vol 5 Issue 7 Aug 2015

ISSN No : 2230-7850

International Multidisciplinary Research Journal

# Indian Streams Research Journal

**Welcome to ISRJ**

**RNI MAHMUL/2011/38595** **ISSN No.2230-7850**

Indian Streams Research Journal is a multidisciplinary research journal, published monthly in English, Hindi & Marathi Language. All research papers submitted to the journal will be double - blind peer reviewed referred by members of the editorial board.Readers will include investigator in universities, research institutes government and industry with research interest in the general subjects.

**Address:-Ashok Yakkaldevi 258/34, Raviwar Peth, Solapur - 413 005 Maharashtra, India**
**Cell : 9595 359 435, Ph No: 02172372010 Email: ayisrj@yahoo.in Website: www.isrj.org**

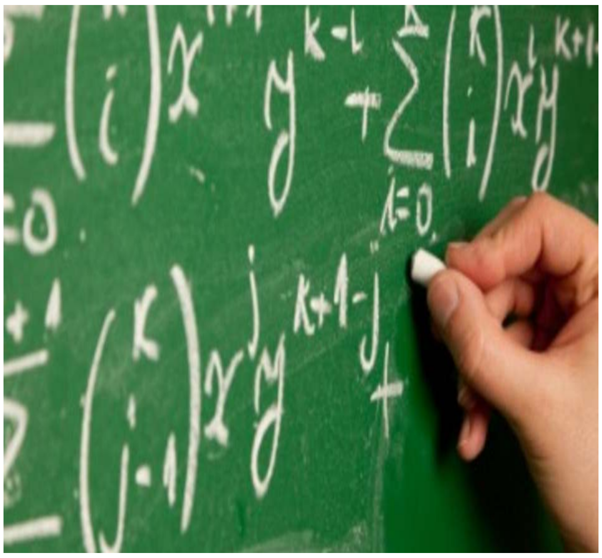
Indian Streams Research Journal
ISSN 2230-7850

Impact Factor : 3.1560(UIF)
Volume - 5 | Issue - 7 | Aug - 2015

# ALGORITHM IMPLEMENTATION TO PREVENT DEADLOCKS USED BY IOS IN MULTIPLE FED BACK QUEUES

Nancy Yaneth Gelvez Garcia[1], Danilo Alfonso López Sarmiento[2], JhonFrancined Herrera Cubides[3]
[1,2,3]Faculty of Engineering, Full time professor at Universidad Distrital Francisco José de Caldas, Bogotá, Colombia.

Danilo Alfonso López Sarmiento
Faculty of Engineering, Full time professor at Universidad Distrital Francisco José de Caldas, Bogotá, Colombia.



ABSTRACT

This paper introduces a study of criteria established in the Mac operating system, better known as MacOS for handling deadlock processes within the system. This document describes each of the features considered for its operation; similarly, it shows the algorithm implementation used by IOS to handle mutual exclusion and deadlock applied to the scheduling process algorithm: fed back multiple queues, noting that the solution proposed by MacOS is efficient in solving the process deadlock..

KEYWORDS :*Fed back multiple queues, mutual exclusion, deadlock, iPhone OS.*

## 1. INTRODUCTION

An operating system is mainly in charge of implementing processes; efficient and timely process deployment depends on it. These processes, managed and handled by the operating system are dependent on the resources for their development; processes will need many resources, and often times, maybe always, they will require them simultaneously. However, these resources can only be used by one process at a time; the system will not operate properly when two processes access the same resource. One of the most important tasks of the operating system is to ensure optimum time for the process to use the required resources.

One might assume that the task is simple, the operating system only has to assign the proper timing for each process and there will be no more problems. However, a process took place that got the operating systems and their behavior in trouble: deadlock. So, they had to propose several ways to prevent these events (to be described later), we will explain why they happen and how they are opposed in the Apple operating system for mobile phones.

## 2. WHAT IS THE DEADLOCK?

We have a process A that needs resources A and B; the operating system allows use of the first, while process B, that needs both, uses the resource available for now (B). In order to continue, A needs the resource being used by B, and, on the other hand B needs the resource not yet released by A. Thus, both are on hold, and therefore the deadlock occurs. This is the most common type of deadlock; there are other types that are not specifically related to E/S resources [1].

A deadlock is nothing more than a "set of processes where each is waiting for an event that can only be caused by another process in the whole" [2]. As both are waiting, none will produce an action to wake up the others, so they will remain in that state indefinitely [3] [4].

Four conditions must take place for the deadlock to occur, if one of them is not present there is no deadlock [5] [6]:

1.Mutual exclusion condition: each process is assigned to only one process.
2.Containment and waiting: processes request new resources that have already been assigned
3.Non preemptive condition: the granted resources cannot be removed by force
4.Circular wait: each process expects a resource occupied by the next process in the circle.

## 3.WHAT IS A SEMAPHORE?

In operating systems, it is a form of synchronization for systems with shared [7] memory. It is an object with an integer value that declares two atomic methods: wait and signal. Its implementation is as follows:

```
Wait(s){
        s =s – 1;
        if(s<0)
        block process}
signal(s) {
        s =s +1;
        if(s<=0)
        unlockprocess blocked bywait}
```

When the value is less than or equal to zero, any wait operation will block the process. When positive, any process running wait will not be blocked (as shown in Figure 1).

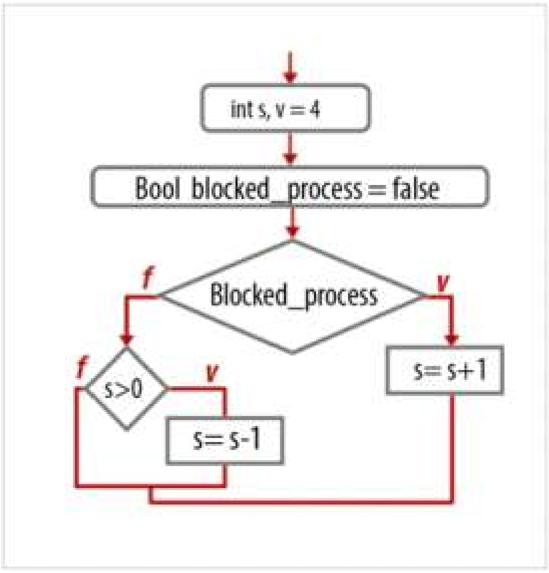


Figure 1. Traffic light algorithm flowchart [7]

## 4.HANDLING CONCURRENCY IN IOS: THE BIRTH OF GRAND CENTRAL DISPATCH.

Apple, creator of the operating system for mobile phones from the same company (IOS), developed a technology for applications that require or demand some form of concurrency. Concurrency is simply the notion of multiple things happening at the same time. With the birth of multiple core CPUs, software application developers need new ways to take advantage of this. Although the IOS operating system is able to run several programs at the same time, many of them run underneath and perform tasks that require continuous processing time. If an application has a lot of work but only uses a fraction of the available cores, the other resources are being wasted.

In the past, concurrency required the programmer to instantly think in threads. However, this was a challenge for them because they are low-tech tools that must be handled manually. In addition, the number of an application's threads can change dynamically (depending on hardware and system performance in real time), therefore, implementing the correct solution using threads is not an easy task. In this way, IOs adopted a more asynchronous strategy for the execution of concurrent tasks than the thread-based system. Instead of directly creating the threads, applications only need to define specific tasks and leave the system to carry them out. In this way, applications are able to reach a level of scalability that was impossible with threads; also, application developers achieve a more efficient programming model.

To achieve this goal, the Grand Central Dispatch (GCD) technology was developed. This technology takes the thread management code that the programmer usually builds for his applications and puts it at the system level. Then, the tasks to be run are defined and added to a dispatch queue. GCD creates the necessary threads and programs the execution of processes in those threads [8].

Also, within the GCD operation, three technologies can be used to develop the applications.

Operation queues, based on Objective-C, can asynchronously encapsulate the work that needs to be deployed.

Dispatch Sources, which is a kind of fundamental data that coordinates the processing of specific low-level system events, which are the most important.

Dispatch queues,They are an easy way to perform tasks concurrently and asynchronously in applications, being a task that an application needs to perform. Dispatch queues serve as an object of a structure, which behaves as a FIFO, so that additional tasks are performed in the same order as they were added. There are three types of queues, serial running one task at a time, concurrent running one or more tasks at the same time, and the most important, that executes the tasks of the main thread in the application.

GCD provides several technologies that can be used to help manage the code in addition to these queues.

Dispatch groups: Provide a useful synchronization mechanism for the code that depends on completion of other tasks.
Dispatch semaphores: If the tasks being sent to the dispatch queues access certain finite resources, semaphores may be needed to regulate the number of tasks simultaneously accessing that resource.

## 5.SEMAPHORES IN GRAND CENTRAL TECHNOLOGY DISPATCH

Dispatch semaphores work like ordinary semaphores with one exception. When the resource is available, it takes less time to obtain access to a dispatch semaphore than a traditional system of semaphores. This is because the GCD does not call within the kernel for this particular case. The only time it does is when the resource is not available and the system needs to stop the thread until the semaphore is in sign state [9].

The semantics for the use of semaphores is as follows:

•When the semaphore is created, it is performed using the dispatch_semaphore_create function. The number of processes available is specified with an integer.
•In each task, the dispatch_semaphore_wait function is called, which puts the semaphore on hold.
•When the semaphore is not in standby, the resource is taken and used.
•When the task is done with the resource, it is released and the semaphore is marked with the function dispatch_semaphore_signal.

The number of available resources is specified when the light is created. This value becomes the initial semaphore variable. Whenever the semaphore is put on hold, the function for this decreases this variable by one. If the result is negative, the function tells the kernel to block the thread. On the other hand, the signal function increases the variable by one to indicate that the resource has been released; if there are blocked tasks waiting for a resource, one will be unlocked and allowed to do its work.

## 6.ADVANTAGES OF THE "IOS" OPERATING SYSTEM COMPARED TO USING "GCD" AND "THREADS" [10]

•It provides a simple and clear programming interface.
•It provides an automatic and holistic way to handle threads.
•It is more efficient in terms of memory because storing batteries is costly to the application memory space.
•It does not trap the kernel when loading.
•Eliminates the need to create and configure threads.
•Eliminates the need to handle and schedule the threads' work.

In summary the operating system handles deadlocks as follows (Figure 2):

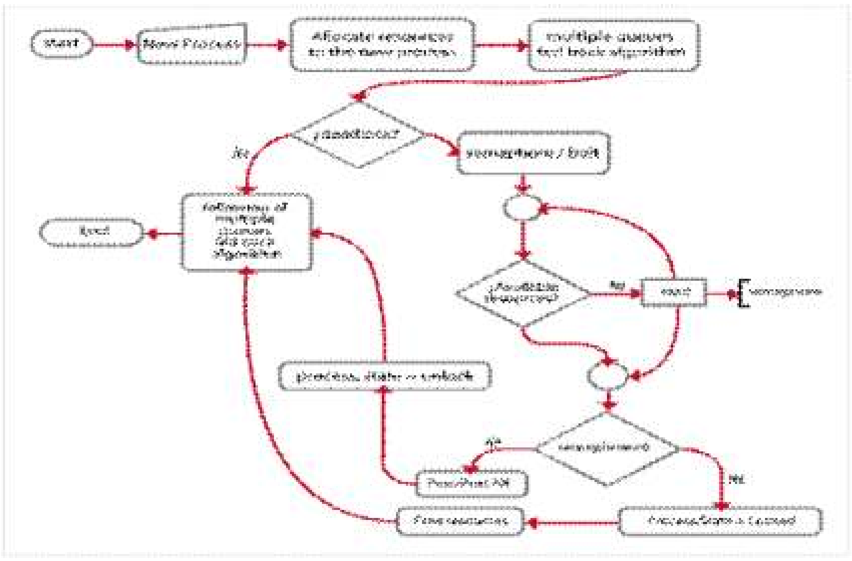
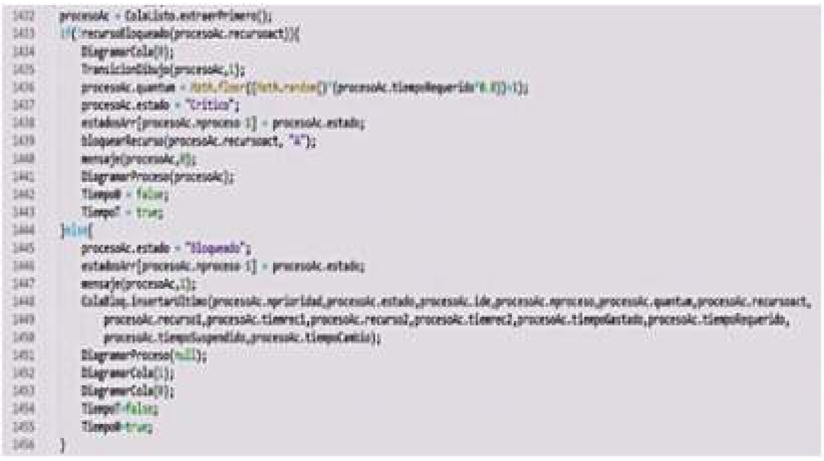


Figure 2. Implemented algorithm flowchart

## 7.IMPLEMENTATION

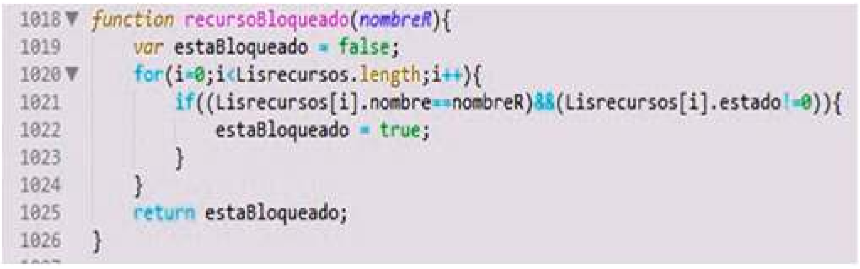
Each process is configured within the solution so that when it starts, its situation is analyzed before entering the critical zone; the lock is adjusted so that the resource needed initially is not blocked by another process (see Figure 3); should it be free, the process will enter the critical zone, otherwise it will be blocked until its resource is available.



```
procesoAc = ColaListo.extraerPrimero();
if(!recursoBloqueado(procesoAc.recursoact)){
    DiagramarCola(0);
    TransicionDibujo(procesoAc,1);
    procesoAc.quantum = Math.floor((Math.random()*(procesoAc.tiempoRequerido*0.8))+1);
    procesoAc.estado = "Critico";
    estadosArr[procesoAc.nproceso-1] = procesoAc.estado;
    bloquearRecurso(procesoAc.recursoact, "A");
    mensaje(procesoAc,0);
    DiagramarProceso(procesoAc);
    Tiempo0 = false;
    TiempoT = true;
}else{
    procesoAc.estado = "Bloqueado";
    estadosArr[procesoAc.nproceso-1] = procesoAc.estado;
    mensaje(procesoAc,1);
    ColaBloq.insertarUltimo(procesoAc.nprioridad,procesoAc.estado,procesoAc.id,procesoAc.nproceso,procesoAc.quantum,procesoAc.recursoact,
        procesoAc.recurso1,procesoAc.tiemrec1,procesoAc.recurso2,procesoAc.tiemrec2,procesoAc.tiempoGastado,procesoAc.tiempoRequerido,
        procesoAc.tiempoSuspendido,procesoAc.tiempoCambio);
    DiagramarProceso(null);
    DiagramarCola(1);
    DiagramarCola(0);
    TiempoT=false;
    Tiempo0=true;
}
```

Figure 3. Lock setup

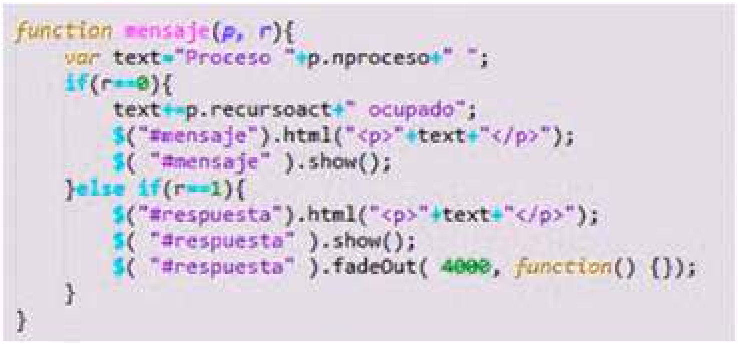
In this condition the state that owns the resource will be analyzed, or if available, 1) if it is in use by another process, or 2) if it is not available due to external failures as shown in Figure 4.



```
function recursoBloqueado(nombreR){
    var estaBloqueado = false;
    for(i=0;i<Lisrecursos.length;i++){
        if((Lisrecursos[i].nombre==nombreR)&&(Lisrecursos[i].estado!=0)){
            estaBloqueado = true;
        }
    }
    return estaBloqueado;
}
```

Figure 4. Setting the initial verification of a blocked resource

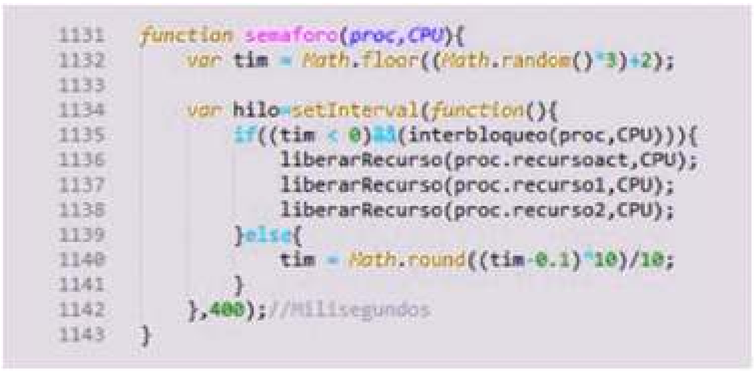
To handle messages and notes it is configured so that each processor shows the process or resource currently in use, as shown in Figure 5, which allows detailed information on system behavior.



```
function mensaje(p, r){
    var text="Proceso "+p.nproceso+" ";
    if(r==0){
        text+=p.recursoact+" ocupado";
        $("#mensaje").html("<p>"+text+"</p>");
        $( "#mensaje" ).show();
    }else if(r==1){
        $("#respuesta").html("<p>"+text+"</p>");
        $( "#respuesta" ).show();
        $( "#respuesta" ).fadeOut( 4000, function() {});
    }
}
```

Figure 5. Notification settings

A function is configured to manage the semaphore so that if there is a deadlock, a counter between 2 and 5 seconds is generated for the affected process and if it becomes negative and the deadlock remains, the process will be expelled from the critical zone and its resources released to other processes. The implementation is shown in figure 6.



```
function semaforo(proc,CPU){
    var tim = Math.floor((Math.random()*3)+2);

    var hilo=setInterval(function(){
        if((tim < 0)&&(interbloqueo(proc,CPU))){
            liberarRecurso(proc.recursoact,CPU);
            liberarRecurso(proc.recurso1,CPU);
            liberarRecurso(proc.recurso2,CPU);
        }else{
            tim = Math.round((tim-0.1)*10)/10;
        }
    },400);//Milisegundos
}
```

Figure 6. Semaphore settings

## 5.CONCLUSIONS

According to Apple, the use of threads to handle concurrency is difficult and tedious for the programmer, and therefore they posit a new technology to make their job easier and to handle efficiently and optimally the need to work with simultaneous tasks in mobile applications.

Deadlock is a situation to which all operating systems must be prepared for, thus each of them offers different strategies to prevent the four conditions leading to it from occurring so that processes can perform their tasks properly.

Semaphores enable processes to not embrace resources, and to not be waiting for the resource (busy waiting), but instead, the semaphore will enable them to know exactly when it has been released and is available for use.

Semaphores are an easy deployment to perform and understand when handling planning algorithms where processes have more than one resource, avoiding a deadlock between them.

Semaphore handling has a significant impact on the operating system, because it prevents two processes with similar requirements from interlocking, seeking to fulfill the needs and expectations of the system as long as they are feasible with existing technologies.

## REFERENCES

[1]. TORRES JIMÉNEZ, JOSÉ. Conceptos de Sistemas Operativos. Editorial Trillas Pre-edición.

[2] Carretero PerezJesus, “Sistemas Operativos: Un visión Aplicada.” McGrawHill.,SegundaEdición, Aravaca Madrid. 238-240.

[3]. ECHAIZ, JAVIER. Departamento de Ciencias e Ingeniería de la Computación. Universidad Nacional del Sur. Interbloqueo, Bloqueo Mutuo. [En línea y PDF]. [Online], Accessed 01/06/2015. http://cs.uns.edu.ar/~jechaiz/sosd/clases/slides/05- Deadlocks-extra.pdf.

[4]. RÍOS, CARLOS. Sistemas Operativos. Procesos Concurrentes. [Online], Accessed 01/06/2015. http://ucinformaticacarlos.blogspot.com/.

[5]. CAMPOS, ANA GABRIELA; BUSTOS BERNAL, JOSÉ RICARDO; ROMERO RAMOS, RAYNIEL. Sistemas Operativos. Principios Generales de la Concurrencia. [Online], Accessed 01/06/2015. http://infoautomaticaso.blogspot.com/2010_11_01_archiv e.html.

[6]. REY, ELOY ANGUIANO; CARRO, ROSA M.; GONZÁLEZ, ANA. Escuela Politécnica Superior. Universidad Autónoma de Madrid. Sistemas Operativos.
[7] Taenbaum Andrew, sistemas operativos modernos. Pearson, Tercera edición. México, 2009. Pag 433 – 438.
[8]. MILLER, BARTON P.; COOKSEY, GREGORY; MOORE, FREDRICK. Computer Sciences Departament.University of Wisconsin. Madison, USA. An Empirical Study of the Robustness of MacOS Applications Using Random Testing.ACM. Proceedings of the First International Workshop on Random Testing (RT'06). Pág.46-54. Julio 20 de 2006, Portland, ME, USA.
[9]. ROGERS, MICHAEL P. Northwest Missouri State University, Mayville, MO. There and Back Again: Leveraging iOs Development on Mac Os X. ACM. CCSC: Central Plains Conference. Pág.174-180. 2011, USA.
[10] Librería de desarrollo de Mac. "Concurrency Programming Guide". [Online], Accessed 28/05/2015.https://developer.apple.com/library/mac/documentation/General/Conceptual/ConcurrencyProgramingGuide/Introduction/I ntroduction.html#//apple_ref/doc/uid/TP40008091-CH1-SW1.