International Journal of Applied Engineering Research ISSN 0973-4562 Volume 10, Number 17 (2015) pp 37768-37774
© Research India Publications. http://www.ripublication.com

Proposal for a busy wait bolt semaphore algorithm controlled to handle interlocking in Android

Nancy Y. Gélvez G
Department of System Engineering Distrital Francisco José de Caldas University
Bogotá, Colombia (South America) nygelvezg@udistrital.edu.co

Danilo A. López S
Department of Electronic Engineering Distrital Francisco José de Caldas University
Bogotá, Colombia (South America) dalopezs@udistrital.edu.co

Jhon F. Herrera C
Department of System Engineering Distrital Francisco José de Caldas University
Bogotá, Colombia (South America) jfherrerac@udistrital.edu.co

Abstract
In this paper the busy wait bolt semaphore algorithm is introduced, used by the Android operating system to address the problem of interlocking processes. Below a description of the mechanisms is made, which in principle, are used by all operating systems. Similarly, emphasis on Linux is made because like Android, it is one of UNIX distributions. The algorithm operation is described, its flowchart is depicted and its behavior is simulated on a three-processor system performed in javascript. Finally the busy wait and controlled bolt semaphore algorithm is proposed, a modification on the algorithm currently used in Android also described and shown in the flowchart. The results show that the proposed algorithm reduces system performance by approximately 3.42%, which is not significant compared to the benefits obtained; therefore, it is an alternative to improve the characteristics of the current Android kernel.

Keywords: Android, prevention and resolution of interlocking, semaphore, lock, algorithm simulation.

Introduction
Various techniques have been used to try to solve the deadlock problem in operating systems, including both software and hardware. Its importance lies in the impact on operating systems, as well as computer systems in general which have difficulty sincronyzing; the need is clearest in database systems or communications networks because of multiple accessibility and constant data updating.
Operating systems face access to resources required by the applications that run on it. In this case in particular, the Android operating system is studied; a UNIX distribution that implements the semaphores algorithm with busy wait bolt, a special mechanism to prevent and confront deadlock, as opposed to other distributions focused solely on prevention. Based on this algorithm, a proposal by the name Semaphore with busy wait and controlled bolt is performed in order to, as in [1], provide mobile phones with immunity, as well as good performance and memory cost reduction.

Race Conditions
This term refers to the conditions that arise when two or more processes try to access a memory or resource at the same instant in time, a typical example is that of an Airline reservation system that tries to allocate the last seat on the plane to several applicants; in computer systems we may illustrate the situation by the conflict between two processes that seek to occupy the same memory location [2] instantly, the variables associated with memory locations are used by both processes and in the end one of the values or states of variables will be lost and one of the process outputs will never take place.

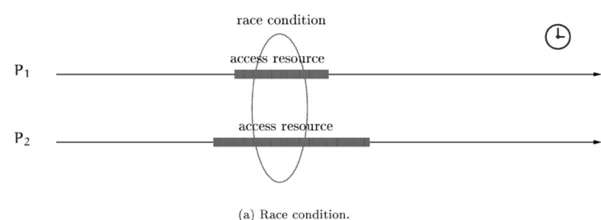


Fig.1. Race condition [3].

Shows the race condition (Figure 1), sequence representation of 2 processes and their access to a shared resource. The figure shows a violation of one of the race conditions: mutual exclusion.

Race conditions are the following:
1. When a process is in its critical section, no other process has access to it; there are no concurrent critical sections. (Mutual exclusion).
2. When none of the processes is in its critical section, any process that requires entering its critical section must be able to do so.
3. When multiple processes compete to enter their critical sections, one should be allowed (only one) to enter its critical section for a finite time.
4. There should always be one process able to access its critical section.

Network Settings to Simulate Mutual Exclusion

Race conditions were observed in item II, one of them is mutual exclusion. In the early 1960s (1962) the problem of mutual exclusion [4] was presented; when a process accesses a resource it must do so in such a way as to exclude others that also require its use at the same time; it is a resource protection mechanism that processes must use; it is called critical section when a process is using a resource exclusively; the other processes on hold are in non-critical section.

This non-critical section consists of 3 sections: input, in which processes request access to the critical section, output, and remaining [5]. Mutual exclusion exists to solve race condition problems. When a process uses resources, it is said to be in critical section, which is defined as part of the code that ensures mutual exclusion; while a process is in critical section the others are in their internal processes; achieving this during all the time the processes are running is what prevents race conditions. The code that performs this task must be done in such a way that [6]:

Solutions that Address the Problem of Critical Section and Mutual Exclusion

A. Dekker Algorithm

The first suggested method to solve the critical section problem guaranteeing fulfillment of the race conditions was the Dekker algorithm written for 2 processes; Edsger Dijkstra tested the proposed algorithm and generalized it for more than 2 processes [7].

With regard to this generalization some authors say it is not "fair" in the sense that not all processes have equal probability of accessing the critical section as they should have [8].

The basic idea behind the Dekker algorithm implementation for two processes is that a process enters immediately to critical section when the other process is not competing to enter it [9], Dekker performs 4 attempts which are described below:

i. Attempt 1

```
int processNumber = 0;
void processZero()
{while (TRUE) do {
while (processNumber == 1) do {} //spin-wait
CriticalRegionZero;
processNumber = 1;
OtherStuffZero; }}
void processOne()
{while (TRUE) do {
while (processNumber == 0) do {} //spin-wait
CriticalRegionOne;
threadNumber = 0;
OtherStuffOne;}} [10]
```

The processNumber variable indicates the process number that alternates in execution 0 for process zero or 1 process for process 1. For the processszero method the while cycle that compares the condition processNumber == 1 "puts process 1 in wait, giving process 0 access to its critical region, CriticalRegionZero, after it completes this critical section the variable processNumber changes its value to 1, process 0 OtherStuffZero tasks are performed, processsOne method executes the same instructions but obviously with the purpose of leaving proceso1 in critical section, in this way the concept of flag is observed which in this case is processNumber; the logic of the first attempt is based on this variable. A strict dependence between processes is generated, which leads to the slower ones "delaying" the others.

ii. Attempt 2

This effort seeks to solve the synchronization problem of the two processes by means of two indicators or flags for each process; these are process0inside and process1inside in the pseudocode considered; before entering its critical section each process asks if the other process is running, if so, a circular wait is executed until the other process is ended, the current process can enter critical section first giving value 1 to its corresponding flag, then executing its critical section, and finally updating its flag to 0. The algorithm does not solve the problem in general and may lead to a violation of the Mutual Exclusion principle, the two processes could enter critical section; if the following sequence [11] takes place in steps 5 and 6 the Mutual Exclusion is broken:

- process 0 evaluates process1inside in the while conditional
- Process 0 finds that process1inside == 0 (the condition is FALSE).
- Process 1 evaluates process0inside in the while conditional
- Process 1 finds that process0inside == 0 (the condition is FALSE).
- Process 0 enters the critical section.
- Process 1 enters the critical section

iii. Attempt 3

In contrast to the previous solution in which processes behave in a "selfish" way (since each uses a flag to indicate that it will enter its critical section), the third attempt takes into account if the other "wishes to enter" critical section. Although it looks similar to the previous attempt, it should be clearly noted that the flags process0WantsToEnter or process1WantsToEnter take the corresponding value before assessing whether the other process needs to enter critical section, if so, it remains in wait, otherwise it proceeds to its critical section and finally changes its indicator to 0.

It may happen that both remain in a kind of courtesy wait where each awaits for the other to release the critical section; a classic case of deadlock, both are waiting for the other to change the flag indicating its intention to enter the critical section to false.

iv. Attempt 4

In the fourth attempt, some instructions are defined in the latency period; they alternate the "desire" of entering critical section for each process through a delay, trying for the other process to enter if both are in simultaneous wait thus avoiding the deadlock.

```
int process0WantsToEnter = 0;
int process 1WantsToEnter = 0;
void process Zero()
{while (TRUE) do {
```

process 0WantsToEnter = 1;
while (process 1WantsToEnter) do { // not quite a spin-wait
process 0WantsToEnter = 0;
delay(someRandomCycles);
process 0WantsToEnter = 1;}
CriticalRegionZero;
process 0WantsToEnter = 0;
OtherStuffZero;}}
void processOne()
{while (TRUE) do {
process1WantsToEnter = 1;
while (process0WantsToEnter) do {
process1WantsToEnter = 0;
delay(someRandomCycles);
Thread1WantsToEnter = 1;}
CriticalRegionOne;
process1WantsToEnter = 0;
OtherStuffOne;}}

The case may be that both delay times are the same, situation that would lead to the deadly embrace of the third attempt, but in this respect it can be said that the probability of this event occurring is minimal.
However, a succession of events is possible in which a process is not granted entry to critical section, although conditions are given for one of the processes to leave critical section, the incoming process will not be able to enter, falling into starvation.

B. The Dijkstra generalization
The mathematician from Rotterdam proposed an implementation of the algorithm for more than 2 processes; although he did not consider his solution as a generalization of the Dekker algorithm, said algorithm is shown in Figure 2.

1. $status[i] := competing;$
2. **do**
3. {
4. **while**$(turn \neq i)$
5. {
6. $status[i] := out;$
7. **if** $status[turn] = out$ **then** $turn := i;$
8. }
9. $status[i] := cs;$
10. }**while**$(status[other] = cs);$
11. **critical section;**
12. $status[i] := out;$

Fig.2. Dijkstra algorithm for n processes [12].

In this algorithm, it should be noted that the variable turn (change) has not initialized, and remains unchanged after each execution in critical section.
This solution is not without flaws and is also susceptible to generate starvation. Priority access is given to the latest process, if this process (which now has the turn in critical section) is continuously interested in entering critical section it will lead to starvation. [13].

C. The Peterson Solution
In 1981 G. L. Peterson devised a simple algorithm to solve the problem of mutual exclusion; the latest attempt by Dekker had become obsolete [14].

```
#define FALSE 0
#define TRUE 1
#define N   2

int turno;
int interesado[N];

void entrar_region(int proceso);
{
    int otro;

    otro = 1 – proceso;
    interesado[proceso] = TRUE;
    turno = proceso;
    while (turno == proceso && interesado[otro]
}

void salir_region(int proceso)
{
    interesado[proceso] = FALSE;
}
```

Fig.3. Peterson algorithm to achieve mutual exclusion [14].

Two processes are defined for the algorithm shown in Figure 3; the variable turn will tell us who's turn it is, the integer variable INTERESTED is 0 at the start, the process is 1 or 0; variable OTHER indicates the number of the other process, variable INTERESTED changes its value to TRUE (the current process wants to enter critical section) , assigns the flag (TURN) to the process that should be run; a wait is made if the condition is given that even if the turn belongs to the current process the other process is interested in entering (INTERESTED [other] = TRUE).
Finally, the algorithm runs exit_region (int process) to indicate that the current process is no longer interested in entering its critical section; the value of the turn variable is overwritten, the last assigned value is selected. When the while cicle is reached, it's the turn of the current process and the other is not interested in accessing therefore the critical section of the current process is run and the other process is excluded.

International Journal of Applied Engineering Research ISSN 0973-4562 Volume 10, Number 17 (2015) pp 37768-37774
© Research India Publications. http://www.ripublication.com

Deadlock

The problem of mutual exclusion is shown in the previous item; deadlock is also a problem faced by computer systems which consists in: when two or more processes are running simultaneously and require one or more resources there may be situations in which both processes need to use the same resource or more resources; let`s assume there are two processes A and B, both recording a document on a CD; A starts using the scanner (its critical section) and B chooses to use the CD recorder, then A requires the recorder, but B does not release it and to complicate the situation, requests the Scanner; now both processes are deadlocked.

Programs that make use of several critical sections are prone to deadlocks.

This situation can happen in many scenarios and systems, data bases, networks of computers (a classic example is that of the shared printer).

In short it is a situation in which one or more processes are waiting for an event which never happens. The most common situation is when there are two processes, each with a synchronization object that another process wants, and there is no way for a process to release the object that it has to allow the other process to continue.

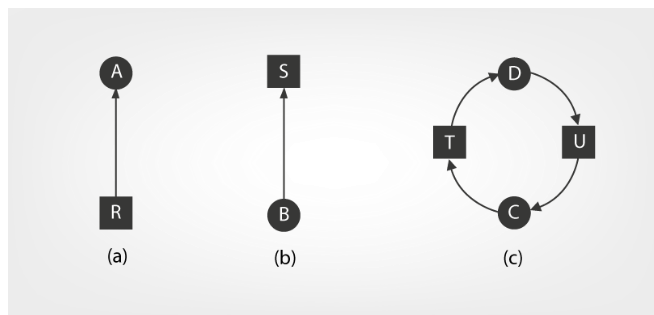


Fig.4. Resource allocation [15].

Figure 4, shows resource allocation processes, where (a) is the resource contention. (b) The request for a resource, and (c) Deadlock [15].

A. Coffman Conditions

In 1970 Coffman and Shoshani set the 4 conditions necesary for deadlock to occur between processes [16]:

1. Mutual Exclusion.
2. Retention and wait: threads that are already withholding some resources may try to keep new resources.
3. No preference condition (No expropriation); once the process is holding a resource, that resource can only be removed when the process that is holding it, voluntarily releases the resource, there is no resource appropriation by an "incoming" process.
4. Circular Wait. There may be a circular chain of processes that request the resources in use by the next process in the chain.

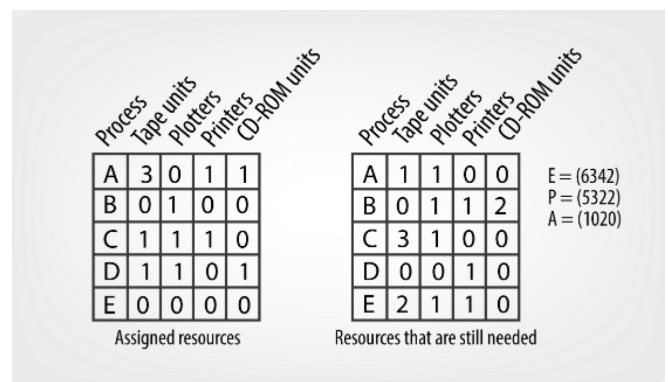
None of the processes can run none of them can release resources and none can be awakened [17].

How to Avoid Deadlocks

Before a state of deadlock arises, many ways of preventing and detecting it can be performed (static analysis), prevention consists in breaking one of the four necessary conditions for deadlock, as for instance, to remove the circular wait of the process configuration, but prevention can negatively influence the system behavior.

A. Banker's Algorithm

Dijkstra proposed a solution called the banker's algorithm. A process after being created, reports the maximum number of resources of each type that can be requested during execution. This information is then taken into account in deciding whether to allow later processes to enter the System [18].



Process	Tape units	Plotters	Printers	CD-ROM units
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Assigned resources

Process	Tape units	Plotters	Printers	CD-ROM units
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources that are still needed

E = (6342)
P = (5322)
A = (1020)

Fig.5. Example of the two arrays that the banker algorithm uses [19].

This algorithm is to satisfy resource requests made by processes as their deployment progresses, this takes into account the initial allocation of resources established when all processes reported the amount of resources needed for their execution (in Figure 5 the parent resource allocation is observed).

The algorithm iterates until all processes are executed and the final safe state is achieved.

Unix. Intrinsic Android Algorithms

Since Android is a distribution of UNIX, its algorithmic foundations are there. In the context of deadlock, according to Vishal Pokarne [20], UNIX pretends it does not exist, as does Windows, to which Ayush Jaiswal [21] adds how inexpensive this technique is, because algorithms require several computational resources to run constantly. Priyanshu Jha Tapasweni Pathak and [21] [22] claim that the execution of an algorithm to deal with deadlock, reduces performance by 20%, so it is unnecessary, especially taking into account that it has an approximate occurrence probability of 0.01%.

With current requirements, simple techniques are born in UNIX and implemented in Android as the hardware upgrades. UNIX takes the least expensive path by not detecting deadlock; it only handles requests that can not be met [23]. According to [24] it has the Helgrind tool to detect synchronization errors tracing effects more accurately therefore handling Deadlock implicitly.

International Journal of Applied Engineering Research ISSN 0973-4562 Volume 10, Number 17 (2015) pp 37768-37774
© Research India Publications. http://www.ripublication.com

Currently, algorithms known as the semaphore and the lock are used together with the lockdep configuration that detects blocking in real time [25]. Lockdep is a validator of the kernel that segments lock acquisition patterns, which according to Ingo Molner [26] has greatly reduced locking and according to [27] has delivered a more stable core. This code fragment comes from <Linux / lockdep.h> and prevents recurrences: current-> lockdep_recursion ++;.

Android Processing

Android is a Linux-based operating system [28] [29] with a 2.6. x kernel simplified to handle most tasks. To activate the execution processes, written in C++, it must be ensured that memory and input/output areas are activated when the system reboots and that the table of processes, drivers, the Dalvik VM, the Zygote, classes and necesary resources have started [21].

Dalvik [28] is the platform that gives Android possibilities in real time. Multiple independent processes with memory addresses and own areas can be run. Each Android application is assigned as a Linux process (task) [29] and runs within its own instance of Dalvik in such a way that Android disclaims all memory and process management liability. It runs Java applications on top of it and is based on infinite records. It requires 35% more bytes in the instruction stream and 30% less instructions [30].

This operating system has three types of processes [29]: 1) active processes with a user interaction, 2) visible processes for partial and transparent activity, and 3) empty processes to improve system performance. The information linked with each process is stored in a process descriptor (trask_struct), a data structure defined in the file <linux / shed.h>; It also stores all processes in a double linked circular list, call process list [31]. Android ensures application response stopping or killing processes that impede the flow, and releases resources for higher priority applications [29].

Android Algorithm

There are 3 techniques developed to synchronize control paths in the kernel [32]: 1) the non-preemptive kernel, used by the traditional UNIX kernel in which a process execution can not be interrupted. 2) Disable interruptions, detected when leaving critical section. 3) The third technique used in Android, is that of semaphores with busy wait bolt, a mechanism used by multiprocessors, with a limited number of semaphores and using the ascending order request protocol.

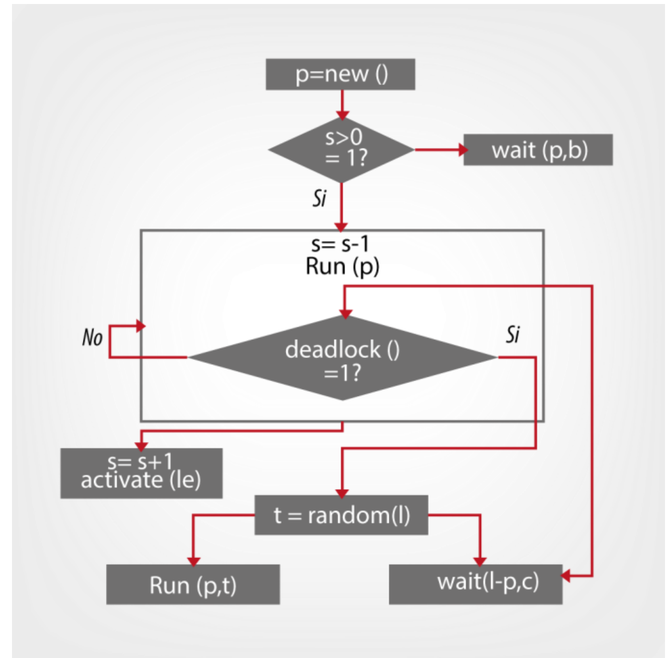


Fig.6. Semaphores algorithm flowchart with busy wait bolt [31].

This algorithm, which is shown in Figure 6, is described by name. First the semaphores algorithm developed by Dijkstra in 1965 [31] is performed. This algorithm consists of two functions: P () and V () working with an initial value for the semaphore, and it is increased or decreased according to system conditions; however it is limited, not being mandatory [33], and remains in busy wait. In the case of Android, functions are called down () and up () respectively.

As for the busy wait bolt (Spinlock [33]), it comprises another semaphore shared by processes with binary values, [34] (two options: acquire () and release () [35]) and running atomic operations [38]. In this case the process remains in standby until permission is granted [36], taking into account that this standby uses up processor time [31]. When Deadlocks are used for mutual exclusion, they are usually called mutex, whose initial value is 1 [31]. Busy wait comprises a number of synchronization mechanisms that provide access to critical sections in short term [37] so caution is necesary, especially with reading and writing mechanisms for which the following solution has been proposed: multi-threaded systems in order to protect resources and prevent inconsistencies [38] as in the case of Android, since it can run actions parallel to the kernel [31].

Figure 6 describes more clearly the Android algorithm to prevent and solve the deadlock. P represents a process, b represents the blocked area, s represents the semaphore, l indicates a list of processes involved in the deadlock, le indicates the waiting list, c refers to critical zone and l - p means all elements of list l with the exception of p.

Proposal

According to [21], the algorithms for handling interlock should be considered good practice: a design which minimizes

processing time and is also dynamic and adaptable "challenge in manufacturing the core."

Following this idea a change in the algorithm is proposed when selecting the turn of processes that are in deadlock. Instead of a random selection, a deductive action is performed, based on two main criteria: the priority of processes and the remaining runtime. Figure 7 shows the selection priority flowchart, where l2 is the auxiliary list that will have the highest priority processes (priority 1 being the highest and 3 the lowest).

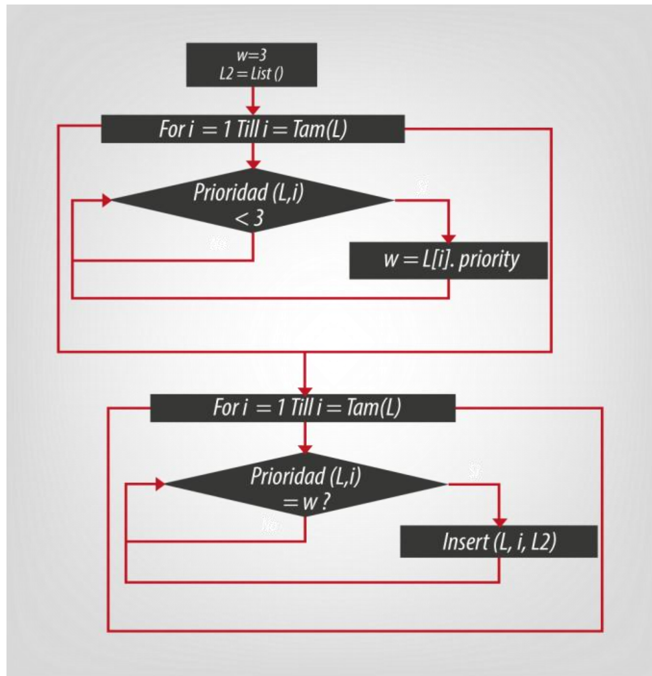


Fig.7. Flowchart of evaluation by priority on the proposed semaphore algorithm with busy wait and controlled bolt.

Replacing the random function of Figure 6 by the homunculus function gives the following simulation code:

**function homunculus(l){
var s = eval_prioridad(l);
s = s.tamL() == 1 ? s.obtDato(1) : eval_t_ejecucion(s); return s;}**

Results and Analysis

Table 1. Runtime comparison on the simulation, between the Android algorithm and the proposed for deadlock, measured in milliseconds.

	P1	P2	P3	P4	Pr1	Pr2
Normal	101	196	209	208	204,33333	178,5
Controlled	1000	192	201	199	197,33333	398

The simulation for the proposed algorithm (Figure 7) and for the algorithm valid in Android is performed, the results, measured in milliseconds, are shown in Table 1.

Column P1 shows test 1, performed on a computer with an Intel Core i 2 Duo processor and Windows 7-32 bit operating system. Columns P2, P3 and P4 refer to tests performed on a computer with an Intel Celeron processor, and Windows 8.1 64 bit operating system. Pr1 indicates the average for the P2, P3 and P4 tests and Pr2 shows the average of the 4 tests.

According to the average values obtained it is proven that the proposed algorithm performs efficiently for existing systems with large processing capabilities; however with older, less updated systems it is clearly deficient.

Similarly, [1] introduces a Dimmunix scheme to give the entire Android platform stagnation immunity with efficient results, generating costs between 4 and 5% in performance and 4% in memory; similar to simulation results.

Conclusion

Applying an algorithm to deal with the deadlock problem implies greater processing and computational expense; however, with current hardware technologies (as mentioned in [38]) there is sufficient capacity to support a control mechanism.

It was noted that the proposed algorithm on systems without enough processing capacity, produced a deficiency of 122.96%, however systems with current processing capabilities, show a deficiency of 3.42%, which is almost insignificant and therefore it is useful to implement in reality.

References

[1] H. Jula, T. Rensch, G. Candea, Platform-wide deadlock immunity for mobile phones, 7th Workshop on hot topics in system dependability, pp. 6, 2011.

[2] P. Maya, A. Kanade and R. Majumdar , Race detection for android applications, Indian Institute of Science, ACM paper, June 2014.

[3] A. Tanenbaum, Sistemas Operativos Modernos, Ámsterdam: Pearson Educación, 3ra Edition, pp. 118, 2009.

[4] E. Dijkstra, Solution of a problem in concurrent programming control, CACM, Volume 8, Issue 9, pp. 569, 1965.

[5] A. Silberschatz, P. Baer. Fundamentos de Sistemas Operativos, McGraw- Hill, 7 Edition, pp. 173, 2006.

[6] R. Jacob, Synthesis of mutual exclusion based on binary semaphores, IEEE journal and magazines, pp.560, 1989.

[7] A. Martin, A New Generalization of Dekker's algorithm for mutual exclusion, Pasadena CA Institute of Technology of California, pp.51, 1985.

[8] A. Martin, A New Generalization of Dekker's Algorithm for mutual exclusion" Pasadena CA Institute of Technology of California, pp. 70-72, 1985.

[9] K. Alagarsamy, Some myths about famous mutual exclusion algorithms, British Columbia Canada, ACM Publication, pp. 77, August 2005.
[10] Clay Beshears, "The art of concurrency", California: O'reilly, 1ra Edition, pp. 54-55, 2009.
[11] Clay Beshears, "The Art of Concurrency", California: O'reilly 2009 1ra Edition, pp. 59-61.
[12] K. Alagarsamy, Some myths about famous mutual exclusion algorithms, British Columbia Canada, ACM Publication, pp. 97, August 2005.
[13] K. Alagarsamy, Some Myths About Famous Mutual Exclusion Algorithms, British Columbia Canada, ACM Publication, August 2005, pp. 105-106.
[14] A. Tanenbaum, Sistemas Operativos Modernos, Ámsterdam: Pearson Educación 2009, 3ra Edition. P. 121-122.
[15] Clay Beshears, The art of Concurrency, California: O'reilly 1ra Edition, pp. 61-62, 2009.
[16] A. Tanenbaum, Sistemas Operativos Modernos, Ámsterdam: Pearson Education, 3ra Edition, pp. 320-330, 2009.
[17] A. Tanenbaum, Sistemas Operativos Modernos, Ámsterdam: Pearson Education, 3ra Edition, pp. 427-437, 2009.
[18] C. Sanchez C, G, Christopher, Efficient distributed deadlock avoidance with liveness guarantees, Stanford University and Washington University, ACM Publications, 2006.
[19] A. Tanenbaum, Sistemas Operativos Modernos, Ámsterdam: Pearson Education, 3ra Edition, pp. 448, 2009.
[20] Quora, Which technique is used for handling deadlocks in UNIX and Windows?. [Online], accessed 05/07/2014, http://www.quora.com/Which-technique-is- used-for-handling-deadlocks-in-unix-and-windows.
[21] Quora, Why do modern operating systems pretend that deadlocks do not occur?. [Online], accessed 07/07/2014, http://www.quora.com/Why-do-modern- operating-systems-pretend-that-deadlocks-do-not-occur.
[22] Quora, Why don't Windows and UNIX recover from deadlock? Why is there no deadlock detection and recover algorithm used?. [Online], accessed 18/08/2014, http://www.quora.com/Why-dont-Windows-and- Unix-recover-from-deadlock-Why-is-there-no-deadlock-detection- and-recover-algorithm-used
[23] Instituto Politécnico Worcester, Deadlock. [Online], accessed 09/03/2014, http://web.cs.wpi.edu/~cs3013/a06/week8 dead.pdf.
[24] Stack Exchange, How to find deadlock reasons for a processon UNIX practically?. [Online], accessed 11/03/2014, http://stackoverflow.com/questions/7922538/how-to-find-deadlock- reasons-for-a-process-on-unix-practically.
[25] Stack Exchange, Kernel: Dealing with deadlocks in UNIX. [Online], accessed 24/05/2014, http://stackoverflow.com/questions/13658645/kernel-dealing-with- deadlocks-in-unix
[26] S. Lockwood-Childs, Liclockdep: userspace lockdep. [Online], accessed 01/03/2014 http://www.vctlabs.com/posts/2014/Jul/09/liblockdep/
[27] J. Corbet, User-space lockdep. [Online], accessed 15/09/2014, https://lwn.net/Articles/536363/.
[28] J. Pan, S. Chen, N. Nguyen, Intelligent information and database systems, Springer, Alemania, pp. 58, 2010.
[29] Gestión de Memoria en Android. [Online], accessed 22/03/2014, http://www.sozpic.com/gestion-de-memoria-en-android/
[30] C. Maia, L. Nogueira, L. Pinho, Evaluating Android OS for embedded real-time systems. Instituto de Porto. [Online], accessed 17/03/2014, http://www.utdallas.edu/~cxl137330/courses/fall13/RTS/papers/4a. pdf.
[31] I. Cano, B. Carraté, M. Rodríguez, Establecimiento externo de límites de procesos en Linux. Universidad Complutense de Madrid. [Online], accessed 04/05/2014 http://eprints.ucm.es/8919/1/memoria.pdf.
[32] J. Gómez, Sistemas Operativos II – Guía Didáctica. Editorial Copicentro, pp. 58-135, España, 2010.
[33] J. Gómez, Sincronización y comunicación, Universidad de Granada. [Online], accessed 08/04/2014, http://lsi.ugr.es/jagomez/sisopi_archivos/3Sincronizacion.pdf .
[34] S. Candela, R. García, Quesada, A. Santana, F, Santos, Fundamentos de Sistemas Operativos, Thomson, PP.110, España, 2007.
[35] G. Romero, Exclusión mutua, Universidad de Granada. [Online], accessed 18/07/2014 http://geneura.ugr.es/~gustavo/aco/teoria/exclusion/exclusion.pdf .
[36] Semáforos. [Online], accessed 28/05/2014, https://1984.lsi.us.es/wiki-ssoo/index.php/Sem%C3%A1foros
[37] Sistemas Operativos II – 2 Control de Procesos y Sincronización, Universidad del País Vasco. [Online], accessed 28/05/2014, http://www.sc.ehu.es/acwlaroa/SO2/Apuntes/Cap2.pdf.
[38] O. Campos, Programando módulos para el Kernel de Linux. Concurrencia en el Kernel. [Online], accessed 11/05/2014, http://www.genbetadev.com/software-libre-y-licencias/programando-modulos-para-el-kernel-de-linux- concurrencia-en-el-kernel.