Design of a Mutual Exclusion and Deadlock Algorithm in PCBSD – FreeBSD

Libertad Caicedo Acosta, Camilo Andrés Ospina Acosta, Nancy Yaneth Gelvez García, Oswaldo Alberto Romero Villalobos.

Universidad Distrital Francisco José de Caldas, Bogotá, Colombia

***Abstract* — This paper shows the implementation of mutual exclusion in PCBSD-FreeBSD operating systems on SMPng environments, providing solutions to problems like investment priority, priority propagation, interlock, CPU downtime, deadlocks, between other. Mutex Control concept is introduced as a solution to these problems through the integration of the scheduling algorithm of multiple queues fed back and mutexes.**

***Keywords* — Mutex, PcBSD, SMPng, FreeBSD, Operating Systems.**

I. Introduction

OVER time operating systems have evolved to reach the progress that can be seen today: starting batch processing, which involved planning the next job to run on a treadmill until multiprogramming systems in which many users waited to be served. With the advent of personal computers has been generally allowing one active process and more resources to which access, then with the integration of more than one processor on a machine, appeared multiprocessing and therefore the concept of parallelism, which involves making one or many processes running on different processors at the same time, being assigned a process per processor. Such evolution is generated from finding that a perceived performance and user satisfaction is optimal.

One of the main functions of the operating system is making decisions about allocating resources to the various processes are in ready state and require access to the same resource; process scheduler uses the scheduling algorithm to make such decisions. Scheduling algorithms implemented in the kernel of the system depending on the environment in which they are seeking to improve the response time, proportionality, predictability, fairness and prevent data loss. [1]

In environments such as real-time or interactive problems may be found when concurrency occurs one or more processors; where processes wish to share the same resource difficulties are encountered when defining the time and the conditions under which each process makes use of the resource, looking in critical section only able to stay a process, ie, that the final result depends on who is running and when it does. This situation leads to problems usually involving shared memory, files, and resources in general (a resource is a hardware device or a piece of information) are generated, which leads to data loss or downtime CPU.

II. Mutual Exclusion And Deadlock

Mutual exclusion is born from the generation of the problems listed above with concurrent programming, seeking to ensure that if a process makes use of a shared resource processes exclude others do the same. However, sometimes the processes are performing internal calculations and other things that do not involve access to the critical section, ie, the part of the program that accesses the shared memory. What is desired is that the processes can operate in parallel to data sharing is optimized over time, as long as only one is in critical section. There are some considerations when performing a mutual exclusion algorithm using critical regions:

- There can only be a process critical section at a time.
- Must know the speeds or the number of CPU's.
- Only the process is in critical section may block other.
- There should be no downtime CPU, because no process can wait infinitely to be executed.

Another mechanism that avoids mutual exclusion is partially disabling interrupts on a CPU; however do user processes and may not be re-enabled that would kill the system, or if the CPU multiprocessing disabling performed cease to function. In the same way the method operates lock variables, in which when the lock is 0, the process can access critical section and when no one is 1; however has problems as to the mechanism mentioned above.

One of the latest implementations of mutual exclusion algorithms are the *mutex*, which allow to manage a resource or piece of code; is very helpful for thread sets that are implemented in user space. The mutex variable is a padlock that can be open or closed, which is represented by 0 if it is open and any other value if it is closed. When a process requires entering critical section checks whether the padlock is open and if so hard to run if it is not blocked until the process in that section is released, ie, the padlock opens. If several blocked threads, then one is selected at random to be the next to access critical section.

The mutex can be viewed from its behavior. As shown in Figure 1, a mutex is a padlock variable that can be open or closed and which may have one of the following behaviors:

DOI: 10.9781/ijimai.2014.316

- Sppining: When you constantly look at the state of the lock to see if the resource is already released. In this case there are no interruptions, then time is wasted waiting for lengthy processes that took control of the resource.
- Blocking: When the resource is not changed to block and awaits the call of the appeal process had, by the time it releases.
- Sleeping: If the resource is not available, it puts the process to sleep until the resource is enabled.

For exceptional cases, the programmer can create a few extra conditions under which a process sends to sleep while the presented problem is solved.

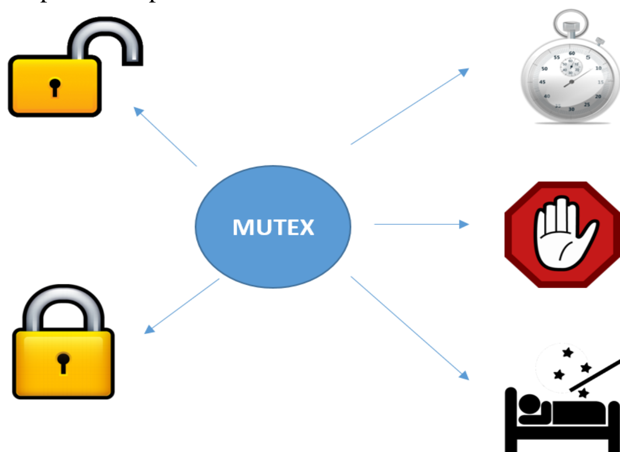

Fig. 1 - Representation of mutual exclusion.

In many cases computers have resources that can be used by only one process in the same space of time; if they run more, inconsistencies or errors occur in the information manipulated. Operating systems temporarily attach to a process exclusive access to certain resources; in cases in which a process one needs more than one resource, first makes the request, but when you need to access another and find that it is occupied by a process 2 which requires use of the resource using process 1; as both are waiting for the other release to free the resource you are running a deadlock occurs. This phenomenon could also occur between machines in the same network with shared devices such as scanners, printers, external drives, etc. [2]. Figure 2 shows more clearly the deadlock problem, with both processes P1 and P2, and two R1 and R2 resources that are left in a standby cycle, and retention of the release of resources.

Deadlocks are usually not preemptive resource linked, ie they may leave without being run over. In some cases there is a quantum that allows equality between processes and subtract the lifetime of the running process and then leave critical section; if it has been completed is deleted, if not, back to the tail of "ready" to run below the remaining time. Such resources may also have an associated priority, which means that in case a higher priority process needs the resource that is being used, use it and send the running process to a suspended list.

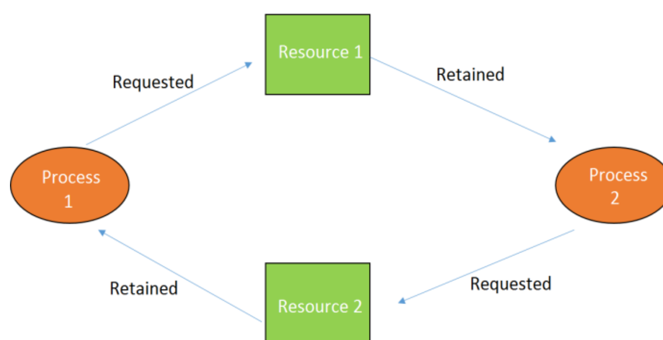

Fig. 2 - Representation of deadlock.

Currently the FreeBSD kernel supports symmetric multiprocessing (SMP), in which all Central Processing Units (CPU's) have a single connection to a non-uniform memory, implementing the Mutex Synchronization strategy as the primary method to manage short-term threads synchronization [3]. One of the desired characteristics for the mutexes design is that acquiring and releasing uncontested mutexes should be as fast as possible, which is one of the reasons for designing an algorithm that works in junction with the FreeBSD-PCBSD mutexes.

III. DESIGN

On operating systems such as FreeBSD PcBSD have been implementing mutual exclusion algorithms increasingly trying to minimize the problems that arise concerning the allocation of resources to processes.

PcBSD is a desktop operating system based on FreeBSD, which provides stability and security in server environments; makes use of window managers and open source application installers of the same type. It is currently in version 10 call PcBSD Joule, which is used in the following analysis and implementation to be discussed later. [4]

A. Mutual Exclusion Algorithm in PcBSD

The implementation of the scheduling algorithms in pcbsd involves evolution in terms of versioning, as currently implemented in some types of which a mutex some extra functionality integrated avoid problems such as investment priority, priority propagation and interlock.

The current implementation emerges from the problems presented with KSE (Kernel Scheduling Entity) in multiprocessing environments that generated downtime CPU and deadlocks [5]; therefore, from version 5.0 of FreeBSD kernel restructuring is done in the way of working threads in such environments, implementing mutexes that lead to a kernel SMPng according to [6].

As mentioned in section 2, there are three types of behaviors associated with mutexes which are kept in the model proposed by the creators of FreeBSD development, ie that there are shared mutex as Spinning, Blocking or Sleeping. Additionally mutexes define four types of [7], which are defined below:

- Mutex: When access to data is located on 1 CPU and accessed by a single thread.
- RW Lock: When access to data is made with several threads on several CPU's, in which reading and writing is permitted, however, only one process can be in write mode, while many in read mode.
- Lock RM: is equal to the RW Lock, only varies in the fact that the reading time is optimized.
- Waitchannel: When a thread requires the use of another thread is assigned a stop expected to sleeping.

It is preferable to use a mutex Blocking that a mutex Spinning in most cases, there are only a few exceptions where the other is better.

Below in Figures 3, 4 and 5 shows the implementation of mutexes, which are encoded in the kernel of PCBSD.

```
struct mtx {
        struct lock_object      lock_object;    /* Common lock properties. */
        volatile uintptr_t      mtx_lock;       /* Owner and flags. */
};
```

Figure 3 - Defining the mutex structure.

The above structure is the /usr/include/sys/_mutex.h system directory.

```
/* Lock a normal mutex. */
#define __mtx_lock(mp, tid, opts, file, line) do {                      \
        uintptr_t _tid = (uintptr_t)(tid);                              \
                                                                        \
        if (!_mtx_obtain_lock((mp), _tid))                              \
                _mtx_lock_sleep((mp), _tid, (opts), (file), (line));    \
        else                                                            \
                LOCKSTAT_PROFILE_OBTAIN_LOCK_SUCCESS(LS_MTX_LOCK_ACQUIRE, \
                    mp, 0, 0, (file), (line));                          \
} while (0)
```

Fig. 4 - Definition of a mutex lock.

The lock and unlock functions are in the C++ library mutex.h, which are located in the /usr/include/sys filesystem.

```
/* Unlock a normal mutex. */
#define __mtx_unlock(mp, tid, opts, file, line) do {
        uintptr_t _tid = (uintptr_t)(tid);

        if (!_mtx_release_lock((mp), _tid))
                _mtx_unlock_sleep((mp), (opts), (file), (line));
} while (0)
```

Fig. 5 - Defining unlock a mutex.

B. Design of mutexes in multiple queues fedback to an environment PCBSD-FreeBSD

In conjunction with the scheduling algorithm of multiple queues fed back [8], we propose an extra control for handling mutual exclusion and deadlock through mutexes which we call Mutex Control. For the specific case of pcbsd-FreeBSD, this control within each scheduling algorithm was implemented as shown in Figure 6, allowing at the time of assessment step in a process for critical section, a mutex is assigned to it. All this for the purpose to have a better management and resource allocation to avoid problems CPU timeouts, deadlocks, interlocking, among others.

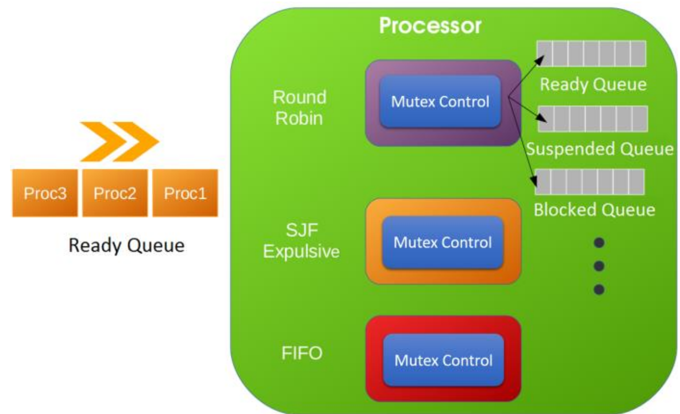

Fig. 6 - Environment and location Mutex Control

Design of Control made Mutex involves a simulated environment algorithm fed back tails, specifically built to allow the integration of mutexes proposed PCBSD-FreeBSD. In Figure 7, was able to visualize the proposed multiple queues fed back flowchart Mutex Control behavior.

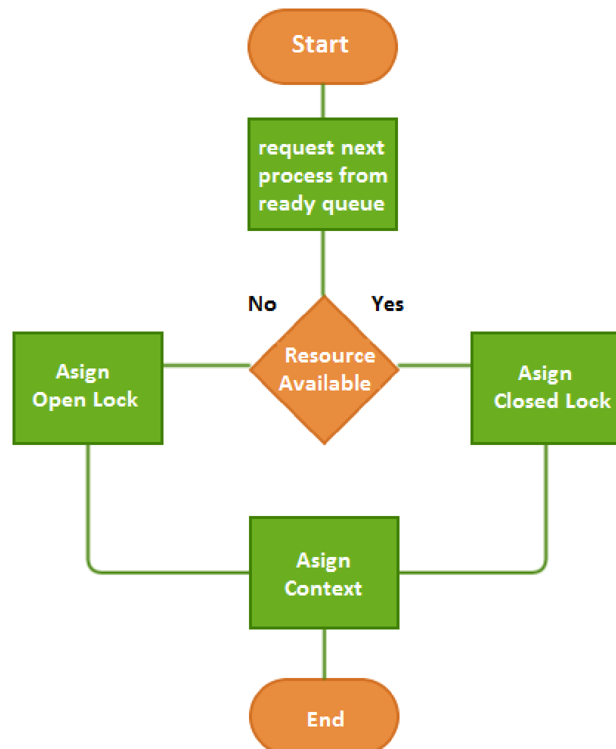

Fig.7 - Mutex Control Operation

The structure mutex proposal to integrate itself to queues fed back algorithm, is given as an adaptation of the libraries implemented in C++ Kernel PCBSD-FreeBSD, reflecting the main behavior that they exhibit a SMPng environment. Representation is in terms of Objects Oriented Programming and structural, in which a Mtx class, mtx_lock function and mtx_unlock function similar to structures in C++, mtx, __mtx_lock and __mtx_unlock respectively is created and seen previously in Figures 3, 4 and 5.

```
class mtx():
    '''
    Simple mutex class for locking resources
    '''
    def __init__(self, resource=None, owner=None, lock=0):
        self.lock_object={"resource":resource}
        self.mtx_lock={"owner":owner, "lock":lock} #0 lock else unlock}


def mtx_lock(mtx):
    if mtx.mtx_lock["lock"]:
        mtx.mtx_lock["lock"]=0

def mtx_unlock(mtx):
    if not mtx.mtx_lock["lock"]:
        mtx.mtx_lock["lock"]=1
```

Fig. 8 - Mtx code implementation.

Integrating mutex controls the scheduling algorithms is performed just after receiving the next process in the ready queue and is given by an assessment of the resources needed by each process, allocation and release locks as can be seen in Figure 8.

```
proc=self.core1.roundRobinAlgo.readyProcesses.next()

#Create New Mutex
new_mtx=mtx(proc.resource, proc.name)
#Eval locking
if bIP.resource["type"] not in (core1resource,core2resource,core3resource):
    mtx_lock(new_mtx)
else:
    mtx_unlock(new_mtx)
#Assign Mutex to Process
proc.mtx= new_mtx

self.core1.roundRobinAlgo.blockedProcesses.append(proc)
```

Fig. 9 - Code implementing the Mutex Control

IV. Mutex Simulation

In order to simulate the coupled behavior of the mutex controller proposed, it was necessary to create an application by multiple queues fed back to incorporate into their calling from the processes queue on each core to mutex control in order to make the decision to change the context of each process (blocked, suspended and critical section).

Then several screens showing action working together mutex control through multiple queues fed back (multiprocessor) can be observed. It should be noted that although the simulation was designed thinking of ways to perform mutual exclusion in PcBSD-FreeBSD, an implementation of a mutex control style could be proposed in other operating systems (improving the effectiveness of the algorithms for SMPng environments).

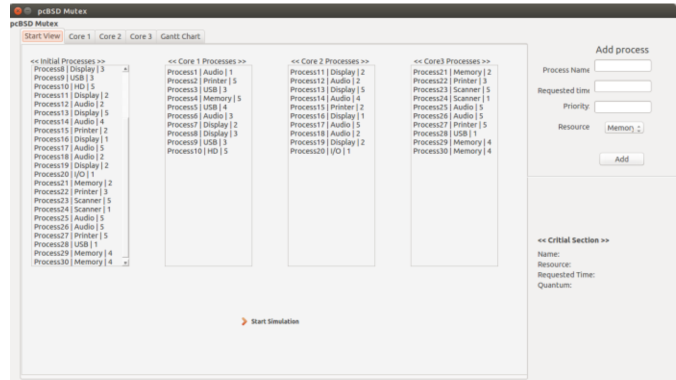

Fig. 10 - Assigning each core processes

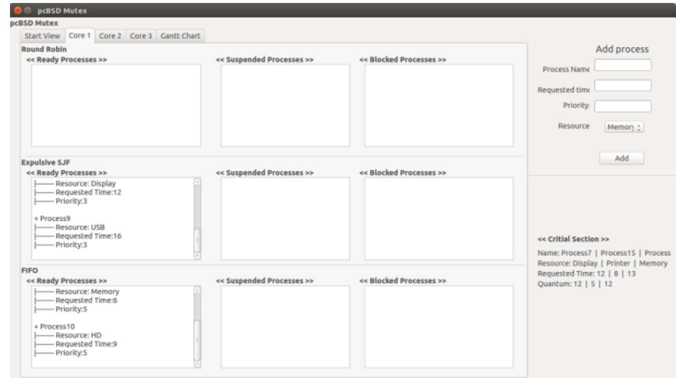

Fig. 11 - Core 1 running

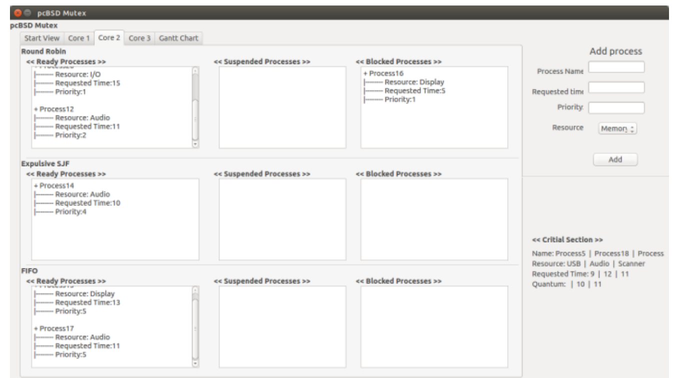

Fig. 12 - Core 2 running

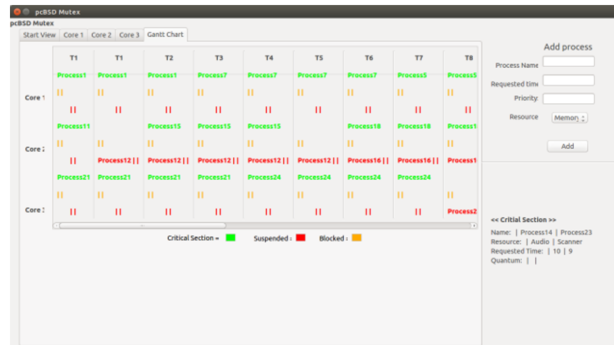

Fig.13 - Gantt execution of algorithms

V. CONCLUSIONS

As previously mentioned, FreeBSD-PCBSD implements the Mutex Synchronization strategy as the primary method to manage short-term threads synchronization which in junction with the proposed algorithm improves the OS desired characteristics for the mutexes.

For each operating system there is a way to implement mutual exclusion that is best suited to your operation. Not always the most complex has better benefits.

Scheduling algorithms and mutual exclusion, require adaptations to environments smpng because it worked very well on a single processor environment, tends to have problems or inefficiencies in the management of resources and response times Multiprocessor.

The mutex control minimizes the problems that are presented to the planning algorithms in environments SMPng such as investment priority, priority propagation, interlock, CPU timeouts, and deadlocks, among others.

Integrating mutexes to implement multiple exclusion in PcBSD-FreeBSD operating systems for SMPng environments, represents major advantages implementation over other mutual exclusion algorithms.

REFERENCES

[1] A. S. Tanenbaum, “Sistemas Operativos Modernos”, Pearson Educación, 3ra edición. pp. 146-160. México 2009.

[2] Zobel, “Operating Systems Review”, Automatica. Vol. 6. Number 1/2, June, 1972.

[3] Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M, Watson, “The Design and Implementation of the FreeBSD Operating System”.

[4] PcBSD. (2014, Julio 13). iXsystems, Inc. [En línea]. Disponible en: http://www.pcbsd.org/es/.

[5] FreeBSD. (2014, Julio 13).” KSE”. [En línea]. Disponible en: http://www.freebsd.org/cgi/man.cgi?query=kse&sektion=2&manpath=FreeBSD+5.0-RELEASE.

[6] Baldwin John. (2014, Julio 13).” How SMPng Works and Why It Doesn't Work The Way You Think ”. [En línea]. Disponible en: http://people.freebsd.org/~jhb/papers/smp/slides.pdf.

[7] Rao, Attilio. “FreeBSD src/ committer since 2007.” AsiaBSDCon 2009 Paper Session. The first part of the two. [En línea] Disponible en: https://www.youtube.com/watch?v=a3XLROUjXic

[8] JRA, “Planificación de Procesos”, Departamento de Informática, Facultad de Ingeniería.Universidad Nacional de la Patagonia “San Juan Bosco” 2010.

Libertad Caicedo Acosta is a final year student from Computers and Science Engineering at The District University Francisco José de Caldas of Bogotá, Colombia. She is one of the founders of the Python programming group created in 2011 at the same University, was an academic assistant for the Computers and Science program in 2013 and for the Complexity Group since 2011. She is currently working to get his B.Sc. It belongs to the research group GIIRA. lcaicedoa@correo.udistrital.edu.co

Camilo Andrés Ospina Acosta is a final year student from Computers and Science Engineering at The District University Francisco José de Caldas of Bogotá, Colombia. He is one of the founders of the Python programming group created in 2011 at the same University, and was an academic assistant for the Computers and Science program in 2013, He is currently working to get his B.Sc. It belongs to the research group GIIRA. caospinaa@correo.udistrital.edu.co

Nancy Yaneth Gelvez García was born in Pamplona, Colombia. She received the B.S. Systems Engineer in Colombian School of Industrial Careers ECCI and M.S. Science in information and communications degrees; Professor and member of the Curriculum Project in Systems Engineering from the University District and GIIRA research group. nygelvezg@udistrital.edu.co

Oswaldo Alberto Romero Villalobos was born in Bogotá, Colombia. He received the B.S. Systems Engineer also S.P. Software Engineer, S.P. Roads Design, Traffic and Transportation and M.S. Industrial Engineering degrees; Professor and member of the Curriculum Project in Systems Engineering from the University District and GIIRA research group. Mr. Romero has been Technical Director well as consultant and advisor to various agreements between the District Department of Transportation in Bogotá and the University District and municipalities in Colombia, also has been a consultant and architect for various software development companies.
oromerov@udistrital.edu.co