I J C T A, 9(4), 2016, pp. 1843-1853

Comparative Analysis of IPC mechanisms for Linux and UC/OS-II

KavithaV.[1], **S. Ravi**[2] **and M. Anand**[3]

ABSTRACT

Great progress of the microelectronic industry for the last 15 years has caused a rapid growth in the field of embedded systems and has given a powerful impulse to the development of software for them, including the embedded operating system (OS). Real time performance analysis is critical during the design and integration of embedded software to guarantee that application time constraints will be met at run time. To select an appropriate for a specific application, OS services need to be compared. These OS services are compared and identified by parameters to form efficient operating system for an embedded system. Evolution of operating system specifically causes significant changes in interprocess communication methodologies. This paper provides a comparative analysis of Linux OS and uc/os-II in the lieu of interprocess communication that forms the main core of the design of an OS.

Keywords: semaphores**,** Linux, uc/os-II, mailbox

1. INTRODUCTION

Real Time Operating System (RTOS) is a multitasking operating system intended for real time applications. It facilitates the creation of real time system. Some key features of RTOS are the minimization of interrupt latency and thread switching latency. uC/OS II is a RTOS developed by Micrium. It uses pre-emption to run tasks and interrupt service routines. It provides semaphores, mutexs, message mailboxes, message queues, timers, event flags, and memory partition management. [2] Real-time systems are widely used in the construction of national defences, aerospace, industrial control and many other fields. As the most critical characters which may affect the whole system, real-time character is required to guarantee the performance of these systems. With the development of computer technology, electronic information technology, many systems are controlled by computer system currently. As the majority of general-purpose computers are using Microsoft's Windows or Linux which is open-source as operating system, which are not real-time operating systems, these systems have not enough realtime characters. At present, the vast majority of real-time systems are embedded systems while are built through the embedded processors with embedded operating system. In this paper, "real-time systems" mentioned in the following text are embedded systems.

Linux is a UNIX-like operating system that is open source and free to use under the GNU General Public License. uClinux is a port of the Linux operating system targeting embedded devices without a Memory Management Unit (MMU). It originally ported Linux 2.0, but has ports based on Linux 2.2 and Linux 2.6 now. There is support for many different processors from ARM to micro Blaze.

Early real-time systems have no operating system supported. To implement interprocess technologies, engineers must program code for specific practical application. Therefore, these particular software developments are less inheritance for code reuse, maintenance and upgrades which brought a lot of trouble.

[1] ECE Department, Dr. M.G.R University Assistant professor, Maduravoyal, Chennai, Tamil nadu, India, *Email: kavithasathish2006@gmail.com*

[2] ECE Department, Dr. M.G.R University HOD, Maduravoyal, Chennai, Tamil nadu, India, *Email: ravi_mls@yahoo.com*

[3] ECE Department, Dr. M.G.R University professor, Maduravoyal, Chennai, Tamil nadu, India, *Email: harshini.anand@gmail.com*

The emergence of real-time embedded operating system provides a powerful tool for real-time systems design and development because of its real-time kernel, multi-tasking, scheduling and fast interrupt response mechanism and so on. Such real-time characteristics can significantly reduce the workload of developers, improve development efficiency, and bring a lot of convenience for the maintenance and upgrading systems. It is crucial for the real-time operating system to adapt pre-emptive scheduling kernel, which is based on task priority. Some key features of RTOS are the minimization of interrupt latency and task switching latency. Embedded linux and ìC/OS-II are two specific operating systems use these features to implement its scheduling.

IPC is used in many applications and is aimed to solve mutual exclusion, synchronisation and data exchange among co-operating processes. Due to different application needs, different IPC mechanisms have evolved. For instance, applications that often perform synchronisation, need a mechanism that is optimised for that. If the mechanism also supports data exchange, it is more likely that it doesn't provide an optimal solution to synchronisation. For applications that both need synchronisation and data exchange yet another mechanism could be the best choice. Accordingly, there are IPC mechanisms that are optimal for solving one or two, or all of the tasks an IPC mechanism is supposed to solve i.e. mutual exclusion, synchronisation, pipes and data exchange. But that is not all. There are some implementation dependent attributes, which can be associated with the primitives. For example, consider an application that needs to exchange data between a group of processes in a synchronous way. The attributes here are collective and block, which are listed below together with some other attributes.

Data exchange attributes:

- Blocked. Synchronous communication.
- Non-blocked. Asynchronous communication.
- Partly blocked. Synchronous communication that timeouts after a specified time.
- Buffered. Holds data until completion.
  - Non-buffered. No buffering of data i.e. receives must precede sends of data.
  - Reliable. Reliable communication over network i.e. data will not be lost.
  - Unreliable. Unreliable communication over network i.e. data may be lost.
  - Collective. Communication between a group of processes i.e. broadcast or multicast communication.
  - Point-to-point. Communication between two processes.

Synchronisation and mutual exclusion attributes:

- Blocked.
- Non-blocked.
- Partly blocked.
- Collective. Synchronisation between a group of processes.
- Point-to-point. Synchronisation between two processes.
- Deadlock free.

IPC involves synchronisation, mutual exclusion and/or data exchange. There is a great variety of applications where IPC is used, e.g. telecom-, robotic-and control systems. Depending on the application type, different IPC mechanisms are needed. In some applications it is sufficient to use a shared storage, e.g.

RAM, for communication. Others make use of more sophisticated communication through e.g. message queues, pipes, signals etc. Common issues that a developer of applications using IPC must be aware of include:

- Race conditions
- Priority inversion
- Deadlock
- Starvation
- Livelock
- Boundedness of buffers

Not considering the above issues when designing applications may result in not (well) working software. To assist the application engineers in their design work, Operating System (OS) vendors have built in IPC functions in their Operating Systems. Many of the communication functions provided by an OS take care of race conditions, mutual exclusion and synchronisation. The other IPC issues are not generally handled, but some IPC mechanisms are powerful in the sense that they are dealing with most of the issues. An example of a mechanism that is deadlock-, starvation free and supports bounded priority inversion is the priority ceiling semaphore. Depending on the application and the type of IPC mechanism used, different methods must be used to achieve reliable communication. Accordingly, to prevent deadlocks, starvation etc. the application engineer must know how to use the communication functions that are provided.

2. DATA EXCHANGE ATTRIBUTES

2.1. Data exchange using memory management in Uc/os-II

Uc/os-II operating system provides OSMemGet () function to allocate the block of memory. Use of this function can create memory partitions for inter-process communication. Figure. 1 shows an example on the use of OSMemGet () to create an integer object shared between 2 processes. Figure. 2 depicts the

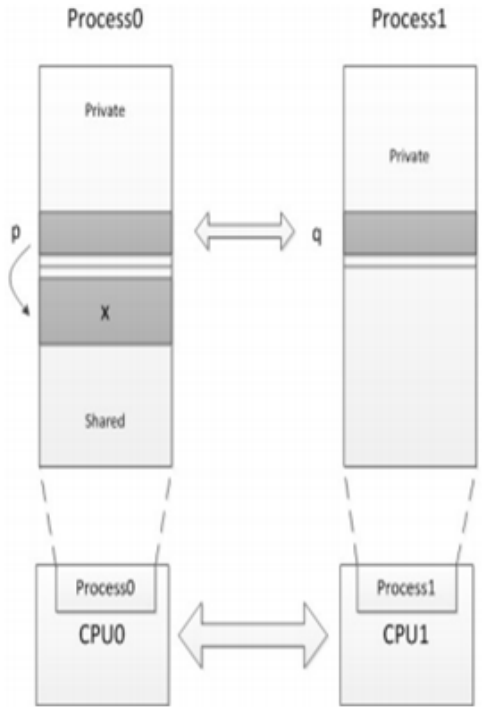

Figure 1: Data exchange between two processes

```
PROCESS 0                                   PROCESS 1
Int*p;                                      int*q;
P=OSMemGet(mem,&err);
msg_send (msg_receive(
    Group,                                      Group,
        1,                                          0,
MESSAGE_TAG,                                MESSAGE_TAG,
        P,                                          P=Q,
                                                        );
            );
```

Figure 2: Pseudocode for the data exchange operation

corresponding code within Process 0 and Process 1 [2]. There are two cores and each of them executes one process. Process 0 uses OSMemGet () to allocate a region of memory to communicate with other processes. The OSMemGet () function returns the address of the shared integer. This value is stored in an integer pointer p in Process 0. Then, the Process 0 sends content of p to Process 1 to exchange information. Process 1 stores this address in the integer pointer q.

After above steps, both Process 0 and Process 1 will be able to load from and store to this shared integer in the same way as a local variable. Any update to *q made by Process 1 can be seen by Process 0 using *p.

2.2. Data exchange using Pipes in Linux

In UNIX, a process has a set of I/O descriptors, from which one reads and to which one writes. Descriptors may refer to normal files, to devices (including terminals), or to communication channels. The use of a descriptor has three phases: its creation, its use for reading and writing, and its destruction. By using descriptors to write files, rather than simply naming the target file in the write call, one gains a surprising amount of flexibility. Often, the program that creates a descriptor will be different from the program that uses the descriptor. For example the shell can create a descriptor for the output of the 'ls' command that will cause the listing to appear in a file rather than on a terminal. Pipes are another form of descriptor that have been used in UNIX for some time. Pipes allow one-way data transmission from one process to another; the two processes and the pipe must be set up by a common ancestor.

Most users of UNIX know that they can pipe the output of a program "prog1" to the input of another, "prog2," by typing the command "prog1 | pro g2." This is called "piping" the output of one program to another because the mechanism used to transfer the output is called a pipe. When the user types a command, the command is read by the shell, which decides how to execute it. If the command is simple, for example, "prog1," the shell forks a process, which executes the program, prog1, and then dies. The shell waits for this termination and then prompts for the next command. If the command is a compound command, "prog1 | prog2," the shell creates two processes connected by a pipe. One process runs the program, prog1, and the other runs prog2. The pipe is an I/O mechanism with two ends, or sockets. Data that is written into one socket can be read from the other. Since a program specifies its input and output only by the descriptor table indices, which appear as variables or constants, the input source and output destination can be changed

```
#include <stdio.h>

#define DATA "Bright star, would I were steadfast as thou art . . ."

/*
 * This program creates a pipe, then forks.  The child communicates to the
 * parent over the pipe. Notice that a pipe is a one-way communications
 * device.  I can write to the output socket (sockets[1], the second socket
 * of the array returned by pipe()) and read from the input socket
 * (sockets[0]), but not vice versa.
 */
```

without changing the text of the program. It is in this way that the shell is able to set up pipes. Before executing prog1, the process can close whatever is at stout and replace it with one end of a pipe. Similarly, the process that will execute prog2 can substitute the opposite end of the pipe for stdin. Since a program specifies its input and output only by the descriptor table indices, which appear as variables or constants, the input source and output destination can be changed without changing the text of the program. It is in this way that the shell is able to set up pipes. Before executing prog1, the process can close whatever is at stdout and replace it with one end of a pipe. Similarly, the process that will execute prog2 can substitute the opposite end of the pipe for stdin.

The parent process makes a call to the system routine pipe (). This routine creates a pipe and places descriptors for the sockets for the two ends of the pipe in the process's descriptor table. Pipe () is passed an

```
/* Create a pipe */
if (pipe(sockets) < 0) {
      perror("opening stream socket pair");
      exit(10);
}

if ((child = fork()) == -1)
      perror("fork");
else if (child) {
      char buf[1024];

      /* This is still the parent.  It reads the child's message. */
      close(sockets[1]);
      if (read(sockets[0], buf, 1024) < 0)
            perror("reading message");
      printf("-->%s\n", buf);
      close(sockets[0]);
} else {
      /* This is the child.  It writes a message to its parent. */
      close(sockets[0]);
      if (write(sockets[1], DATA, sizeof(DATA)) < 0)
            perror("writing message");
      close(sockets[1]);
}
}
```

Figure 3: Pseudocode for the data exchange operation

array into which it places the index numbers of the sockets it created. The two ends are not equivalent. The socket whose index is returned in the low word of the array is opened for reading only, while the socket in the high end is opened only for writing. This corresponds to the fact that the standard input is the first descriptor of a process's descriptor table and the standard output is the second. After creating the pipe, the parent creates the child with which it will share the pipe by calling fork (). Figure. 3 illustrates the effect of a fork. The parent process's descriptor table points to both ends of the pipe. After the fork, both parent's and child's descriptor tables point to the pipe. The child can then use the pipe to send a message to the parent.

Figure. 4 shows sharing a pipe between parent and child where parent and child need to agree on which way to turn the pipe, from parent to child or the other way around. Using the same pipe for communication both from parent to child and from child to parent would be possible (since both processes have references to both ends), but very complicated. If the parent and child are to have a two-way conversation, the parent creates two pipes, one for use in each direction. (In accordance with their plans, both parent and child close the socket that they will not use. It is not required that unused descriptors be closed, but it is good practice.) A pipe is also a stream communication mechanism; that is, all messages sent through the pipe are placed in order and reliably delivered. When the reader asks for a certain number of bytes from this stream, he is given as many bytes as are available, up to the amount of the request. Note that these bytes may have come from the same call to write () or from several calls to write () which were concatenated.

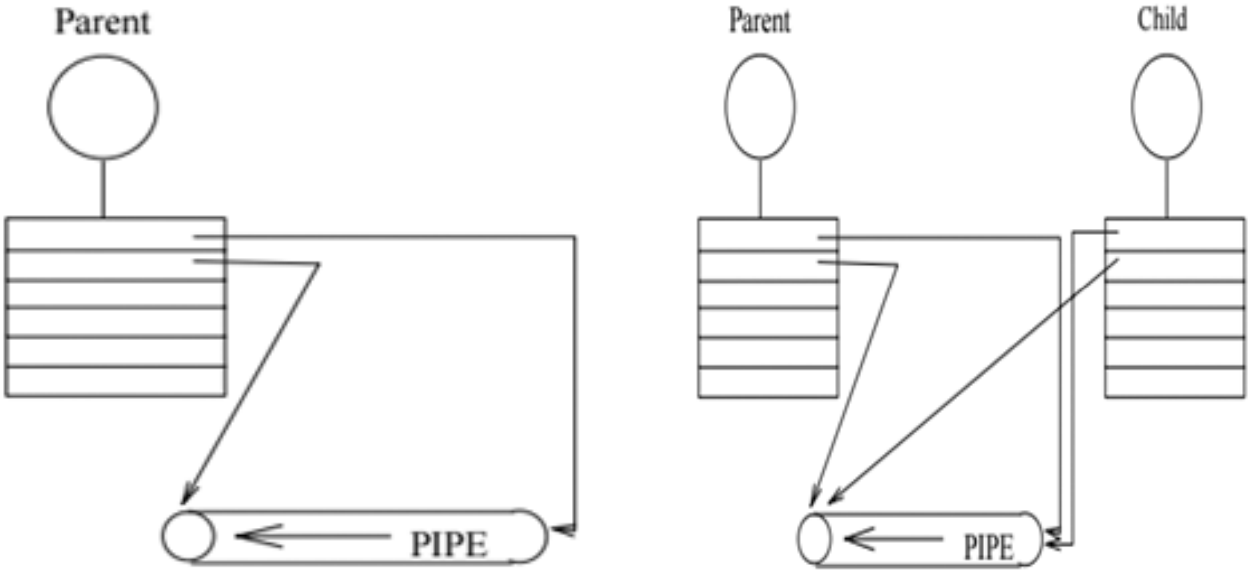

Figure 4: Sharing a pipe between parent and child

3. SYNCHRONIZATION MECHANISMS

3.1. Synchronization in UCOS-II

Synchronization is necessary when resources are shared by the concurrent tasks. Typically these resources are shared variables, but may also be other internal or external resources. All languages, program libraries, and operating systems that provide concurrency also have to offer some means of synchronization. Semaphores are used for synchronization between tasks and are assumed as atomic operations:

```
WAIT ( S ):
    while  ( S <= 0 );
    S = S - 1;
```

```
SIGNAL ( S ):
    S = S + 1;
```

As given here, these operations are defined, however, to be atomic (Protected by a hardware lock) The format of mutex is shown below.

```
FORMAT:
        wait ( mutex );          <-- Mutual exclusion: mutex init to 1.
         CRITICAL SECTION
        signal( mutex );
```

Semaphores can be used to force synchronization (precedence) if the preceeder does a signal at the end, and the follower does wait at beginning. For example, here P1 needs to be executed before P2.

```
P1:                          P2:
   statement 1;                 wait ( synch );
   signal ( synch );            statement 2;
```

3.2. Synchronization in Linux

For synchronization, threads may use a monitor-like critical region (of the type pthread_mutex_t) to which condition queues (of the type pthread_mutex_t) may be associated. Regions and condition queues must be initialized before they are used. Exclusive access to a critical region m is obtained by calling pthread_mutex_lock(m) at entry and pthread_mutex_unlock(m) at exit. There are also versions with timeout. A critical region is owned by the thread that has locked it and cannot be released by another thread. Further, a thread cannot acquire a region that it has already acquired.

```
#include <pthread.h>

class Sem {

  pthread_mutex_t mutex;
  pthread_cond_t queue;

  int S;

public:

  Sem() {
    S = 0;
    pthread_mutex_init(&mutex, null);
    pthread_cond_init(&queue, null);
  }

  void P() {
    pthread_mutex_lock(&mutex);
    while (S == 0) pthread_cond_wait(&queue,&mutex);
    S-;
    pthread_mutex_unlock(&mutex);
  }

  void V() {
    pthread_mutex_lock(&mutex);
    S++;
    pthread_cond_signal(&queue);
    pthread_mutex_unlock(&mutex);
  }

}
```

Figure 5: Pseudocode for mutual exclusion in Linux

4. PERFORMANCE ANALYSIS OF SEMAPHORES

4.1. Semaphore activation APIs used in linux and UC/OS-II

Semaphores provide a satisfactory solution related to concurrency and these are called sleeping locks in Linux [6]. Access to critical section is controlled by enforcing processes to hold a lock before entering into critical sections. They cause a process to sleep on contention, instead of spin; they are used in situations where the lock-held time may be long. Semaphores are represented by a structure, struct semaphore, which is defined in include/asm/semaphore.h. The structure contains a pointer to a wait queue and a usage count. The wait queue is a list of processes blocking on the semaphore. The usage count is the number of concurrently allowed holders. If it is negative, the semaphore is unavailable and the absolute value of the usage count is the number of processes blocked on the wait queue. The usage count is initialized at runtime via sema_init (), typically to 1 (in which case the semaphore is called a mutex).

Semaphores are manipulated via two methods: down (historically P) and up (historically V). The former attempts to acquire the semaphore and blocks if it fails. The later releases the semaphore, waking up any processes blocked along the way. Semaphore use is simple in Linux. To attempt to acquire a semaphore, call the down interruptible () function. This function decrements the usages count of the semaphore. If the new value is less than zero, the calling process is added to the wait queue and blocked. If the new value is zero or greater, the process obtains the semaphore and the call returns 0. The up() function, used to release a semaphore, increments the usage count. If the new value is greater than or equal to zero, one or more processes on the wait queue will be woken up:

```
struct semaphore mr_sem;
sema_init (&mr_sem, 1);     /* usage count is 1 */

if (down_interruptible (&mr_sem))
/* semaphore not acquired; received a signal... */

/* critical region (semaphore acquired)... */

up (&mr_sem);
```

In uc/os-II, semaphores consist of a waiting list and an integer counter, implemented using two functions-ossempend() and ossempost(). When ossempend() decrements the counter and if the value of semaphore is lesser than 0, the task is blocked and is moved to waiting list immediately. If the counter value is incremented by ossempost() and semaphore value is greater than or equal to zero, the corresponding task is removed from the wait list and rescheduling is done if necessary.

Priority inversion occurs when a low-priority task owns a semaphore, and a high-priority task is forced to wait on the semaphore until the low-priority task releases it. If, prior to releasing the semaphore, the low priority task is pre-empted by one or more mid-priority tasks, then unbounded priority inversion has occurred because the delay of the high-priority task is no longer predictable. Table I shows semaphore activation APIs used in linux and uc/os-II.

Table 1
Semaphore Apis In Linux And UC/OS-II

Get semaphore API	ossempend()	APIs used for semaphore activation in uc/os-II
Release semaphore API	ossempost()	
Get semaphore API	down_interruptible()	APIs used for semaphore activation in linux
Release semaphore API	up()	

4.2. Measuring semaphore get/release time

Semaphore is commonly used for synchronization primitive in RTOS. The times taken by get and release semaphore service call will be measured, and the time required to pass the semaphore from one task to another task will also be measured. Figure 6 illustrates the method used to measure semaphore get release time.

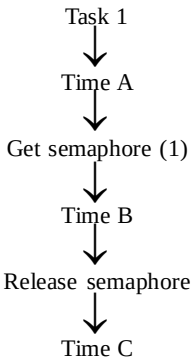

Figure 6: Measuring semaphore get/release time

Time to get semaphore: (A) to (B)

Time to release semaphore: (B) to (C)

There is only one task (Task1) and one binary semaphore (initialized to 1). Task1 will get the semaphore and then it will release it. Different RTOSes use different terms to describe the get/release of semaphore.

4.3. Measurement of semaphore passing time

To measure the performance of semaphore passing, the following measurement method as shown in Figure 7 is used.

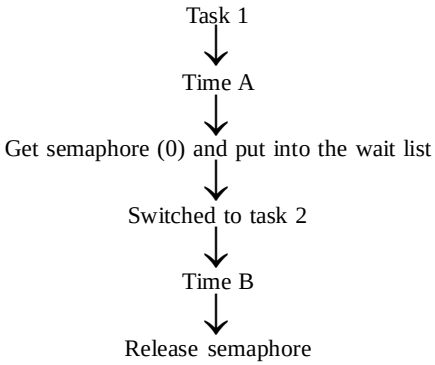

Figure 7: Measuring semaphore passing time

Semaphore passing time from task 1 to task B: (A) to (B)

There are two tasks: Task1 and Task2 with Task1having higher priority, and a binary semaphore (initialized to 0). At the beginning, Task1 is first executed and it tries to get the semaphore. Since the semaphore value is 0, Task1 will be put into a *sleep/inactive* state, waiting for the semaphore to be released. The current execution context will then be switched to Task2 which will release the semaphore. The semaphore, once released, will wake up Task1, and the execution context will be switched over to Task1.

Figure. 8 shows the measurement of execution time of linux and uc/os-II for different semaphore performance criteria. From the results shown, uc/os-II acquire and release time is the fastest whereas

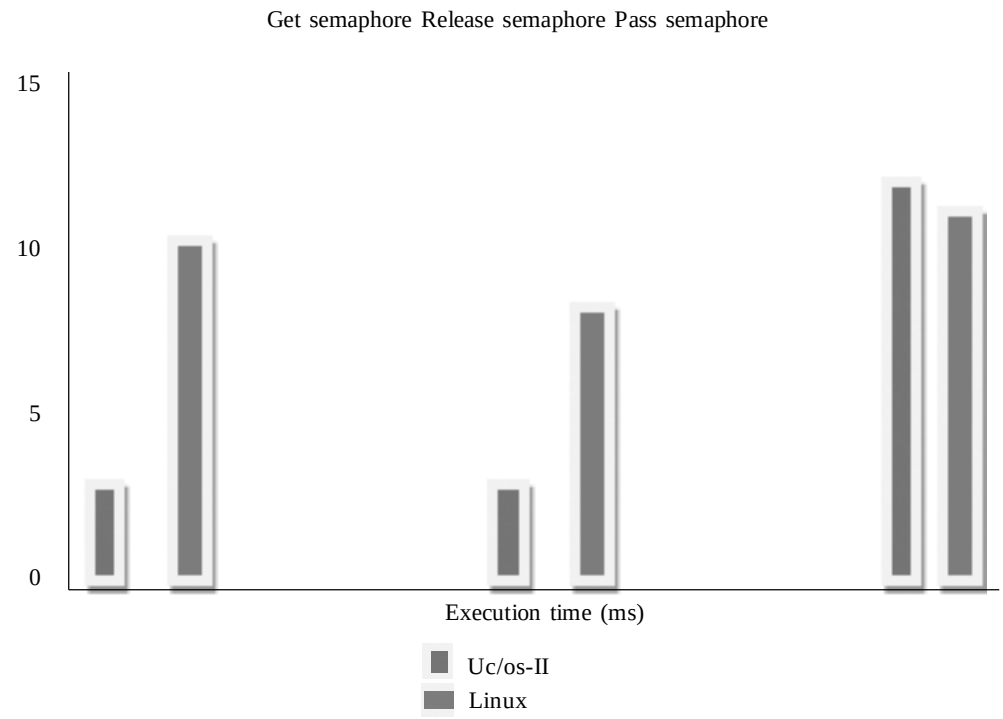

Figure 8. Comparative chart for various semaphore metrics

it is somewhat slower in Linux than uc/os-II.Pass semaphore time is somewhat faster in linux than uc/os-II.

A semaphore is a real-time mechanism that abstracts the control of access of shared resources by multiple tasks. A task should wait until a resource unit becomes available. Figure. 9 presents the pseudocodes for time performance analysis on a semaphore.

```
void Task1() { set period and make periodic; … while(1) {
release semaphore; start time measurement;
    }
        }
void Task2() { … while(1) { get semaphore; stop time
measurement;
    }
        }
```

Figure 9: Pseudocode for the Time Response of Semaphore

4.4. Race condition among tasks

The tasks of the system may compete for sharing resources. It will definitely cause some tasks to suspend and wait for the sharing resource. In preemptive scheduling kernel, priority inversion is a serious problem caused by race condition. A low-priority task which occupies critical resources has no right to implement, while a high-priority task has to wait a middle-priority task to release CPU to low-priority task. So the high-priority task is affected seriously and the task scheduling will become unstable and unpredictable. The realtime performance of system deteriorates rapidly. After all, the high-priority task can only seize the CPU from the low priority task. It can't seize the resources.

To eliminate race conditions, it is necessary to use priority inheritance, priority ceiling protocols and make operations atomic to resolve the problem, an atomic operation is indivisible and uninterruptible; once the operation starts, it will not be paused or interrupted until it completes, and no other operation will take place meanwhile. This section of code running uninterruptable is known as critical section.

- In ucos/II, race condition is avoided by implementing APIs-OS_ENTER_ CRITICAL () and OS_EXIT_CRITICAL () that locks CPU interrupts.
- Linux kernel contains an implementation of priority inheritance algorithm which is a part of the in-kernel Linux real time effort to make the entire Linux kernel fully pre-emptive. It waits until a high priority task requests a lock held by a low-priority task, and then boosts the low priority task priority up to the high priority level until the lock is released. This method does not prevent deadlocks, but does prevent priority inversion.
- The priority ceiling protocol which is an enhancement of basic priority inheritance is implemented in uc/os-II which assigns a "ceiling priority" to each semaphore, which is the priority of the highest job that will ever access that semaphore. A job then cannot pre-empt a lower priority critical section if its priority is lower than the ceiling priority for that section. This method can be sub-optimal, in that it can cause unnecessary blocking. However, it does prevent both priority inversion and deadlocks.

5. CONCLUSION

The selection of right operating system for a specific application has a great impact on performance of real-time system. In the embedded application the improvement of real-time performance of Linux kernel has far-reaching significance. The complex relationship between the tasks may cause heavy system consumption on internal communication between tasks. The Scheduling and separating the tasks in the beginning will also give us flexibility to apply different algorithms for different task queues. Linux assigns CPU time to each process fairly but real-time scheduling algorithms select the next process according to its schedule policy [3]. EDF shows the good scheduling performance in uc/os-II kernel but only considers deadline and it is not adequate for multimedia embedded systems [4]. Synchronization mechanism between tasks will decline the real-time performance of system. At last, how to use a real-time operating system to implement an actual application system is the key for all embedded system. As of February 2014, an additional scheduling policy known as *SCHED_DEADLINE* is on its way to be merged into the kernel mainline, offering earliest deadline first scheduling in the Linux kernel.

REFERENCES

[1] Mr. P. S. Prasad and Dr. Akhilesh Upadhyay," Scheduling Policy and its Performance for the Embedded Real time System*", International Journal of Advanced Research in Computer and Communication Engineering*, Vol. 2, Issue 4, April 2013.

[2] [2] uC/OS-II TM Real-Time Operating System, http://micrium.com/ newmicrium/uploads/file/datasheets/ucosii datasheet.pdf, Micrium, May. 2009

[3] OSKAR ÖRNVALL.,"Benchmarking Real-time Operating Systems for use in Radio Base Station applications", thesis, Department of Computer Science and Engineering, Göteborg., Sweden-March 2012.

[4] Byoungchul Ahn* Ji-Hoon Kim and Dong Ha Lee Sang Heon Lee, "A Real Time Scheduling Method for Embedded Multimedia Applications", Daegu Gyeongbuk Institute of Science & Technology, Daegu, Korea.

[5] Charulata Ingle and Rajshree Daryapurkar," Performance Analysis of Embedded Linux in Embedded System", *International Journal of Innovative Technology and Exploring Engineering, (IJITEE) ISSN: 2278-3075, Volume-2, Issue-2, January 2013.*

[6] Jayaraj P.B, Ranjith Gopalakrishnan and Vikas Jain2M," EFFECTIVE PERFORMANCE ANALYSIS AND VISUALIZATION ON EMBEDDED LINUX", proceedings of 10th LASTED international conference, software En gineering and applications, November 13 to 15,2010, Philips Innovation Campus No. 1, Murphy Road, Ulsoor, Bangalore.

[7] Samih M. Mostafa, S. Z. Rida & Safwat H. Hamad**,** "Finding Time Quantum of Round Robin CPU Scheduling", *IJRRAS 5 (1), October 2010.*

[8] Jae Hwan Koh and Byoung Wook Choi,"Real-time Performance of Real-time Mechanisms for RTAI and Xenomai in Various Running Conditions", *International Journal of Control and Automation Vol. 6, No. 1, February, 2013.*