Proyecto Sistemas Operativos

Segundo Cuatrimestre 2019

Rau, Joaquín (LU: 117375)

Montenegro, Maximiliano (LU: 118190)

1. **Experimentación de Procesos y Threads con los Sistemas Operativos**

**Para el punto 1.1.1**:

● El ejecutable se encuentra en **/ProyectoSO-Rau-Montenegro/collatz/collatzHilos.sh**

● El archivo fuente se encuentra en  
 **/ProyectoSO-Rau-Montenegro/collatz/ejecicio\_hilos\_collatz.c**

**Estrategia**  
Se valida el dato de entrada y se crea el hilo encargado de generar la secuencia. Este hilo utiliza la función **collatz()** para dicha tarea que mediante el join, devuelve un **struct parametros** con la secuencia lista para ser mostrada por pantalla.

Para el punto 1.1.2:

● El ejecutable se encuentra en **/ProyectoSO-Rau-Montenegro/collatz/collatzProcesos.sh**

● El archivo fuente se encuentra en   
**/ProyectoSO-Rau-Montenegro/collatz/ejecicio\_procesos\_collatz.c**

**Estrategia**

Se valida el dato de entrada, se crea al proceso hijo encargado calcular la secuencia. Para esto, el proceso hijo utiliza la función **collatz()** pasando por el **pipe()** el resultado de esta. Luego de esto el padre imprime por pantalla la secuencia y el programa termina.

**Para el punto 1.1.2:**

● El ejecutable se encuentra en **/ProyectoSO-Rau-Montenegro/shell/shell.sh**

● El archivo fuente se encuentra en **/ProyectoSO-Rau-Montenegro/shell/shell.c**

**Estrategia**

El programa consiste de un bucle que le va pidiendo al usuario comandos hasta que se le ingresa un **exit.** Se tienen los programas compilados de cada uno de los comandos por separados, y cuando se selecciona un comando, **se realiza un fork()** y se cambia la imagen ejecutable del proceso hijo por la del comando seleccionado. Esto se realiza mediante la funcion **execl().**

**Para el punto 1.2.2.a**

● El ejecutable se encuentra en   
/**ProyectoSO-Rau-Montenegro/sincronizacion/alumnosIndependientes.sh**

● El archivo fuente se encuentra en   
**/ProyectoSO-Rau-Montenegro/sincronizacion/alumnos\_posgrado\_independientes.c**

**Estrategia**

Los alumnos son hilos independientes, cada uno de ellos tarda en hacer un curso lo especificado en el ejercicio.

**Para el punto 1.2.2.b**

●El archivo ejecutable se encuentra en  
 **/ProyectoSO-Rau-Montenegro/sincronizacion/alumnos.sh**

● El archivo fuente se encuentra en  
**/ProyectoSO-Rau-Montenegro/sincronizacion/alumnos\_posgrado.c**

**Estrategia**

Se utilizaron 5 semáforos, uno por cada alumno. Un alumno espera por el resto de alumnos antes de empezar un nuevo curso, esto lo realiza utilizando sem\_wait() por cada otro alumno. Cuando un alumno termina un curso, hace 4 sem\_post() avisando a los otros alumnos que ya termino.

**Para el punto 1.1.2.a**

● El archivo ejecutable se encuentra en  
**/ProyectoSO-Rau-Montenegro/sincronizacion/ascensorHilos.sh**

● El archivo fuente se encuentra en   
**/ProyectoSO-Rau-Montenegro/sincronizacion/ejercicio\_ascensor.c**

**Estrategia**

Estamos modelando un ascensor que está bajando y subiendo constantemente. Se detiene en los pisos en los cuales hay gente esperando o hay gente por bajar. Para simular las personas se utiliza un arreglo de threads.

Se utilizan cuatro arreglos de semáforos, dos para modelar el comportamiento de subida de personas al ascensor y los otros dos para modelar el comportamiento de bajada de las personas.

Los arreglos **cant\_gente\_esperando** y **cant\_gente\_bajar** modelan la cantidad de personas que se tienen o bajar en cada piso respectivamente. El arreglo de semáforos **esperando\_bajar** modela el momento en el cual el ascensor llega al piso al cual la persona se quiere bajar. Esto se hace mediante la utilización de la función **sem\_post()** lo cual desbloquea a la persona para que pueda descender del ascensor.

El arreglo de semáforos **esperando\_ascensor** posee un comportamiento similar al anterior pero este se utiliza para permitirle a una persona que está esperando en un determinado piso, subirse al ascensor cuando este se encuentra en el piso.

El comportamiento de una **persona** es el siguiente:

● Aleatoriamente tiene un piso inicial, y un piso destino

● Espera a que el ascensor llegue a su piso inicial, aumentando la cantidad de personas esperando en ese piso

● Una vez que llega el ascensor, aumenta la cantidad de gente esperando por bajar en el ascensor y espera a llegar a su destino.

El comportamiento del **ascensor** es el siguiente:

● Si está en un piso donde se tiene que bajar gente, el ascensor las despacha. Por cada persona que tiene que bajar, el ascensor habilita a bajar a estas personas

● Y luego, habilita a las personas esperando por subir en ese piso, a que se suban

**Para el punto 1.1.2.b**

● El archivo ejecutable se encuentra en   
**/ProyectoSO-Rau-Montenegro/sincronizacion/ascensorProcesos.sh**

● El archivo fuente se encuentra en   
**/ProyectoSO-Rau-Montenegro/sincronizacion/ascensor\_colas.c**

**Estrategia:**

**2. Problemas**

**Punto 1:**

1. Sandboxing es un mecanismo de seguridad que permite a los programas ejecutarse en espacios aislados. Una “caja de arena” es un ambiente donde el programa tiene acceso limitado a los recursos del sistema. El Sandboxing está basado en la prevención en lugar de la detección.   
   La mayoría de los métodos de Sandboxing proveen un enfoque basado en aislamiento, donde el programa en ejecución está totalmente aislado de los recursos afuera de la “caja de arena” y también de los programas fuera de esta.
2. Para la implementación del sandbox, los proveedores de software confían en

Características implícitas de seguridad del sistema operativo. De este modo, cada software tiene una implementación de sandbox diferente para el sistema operativo subyacente.

1. La similitud que encuentro entre el concepto de máquina virtual y el concepto de sandbox es que ambos hacen “creer” al programa algo que no es. Por ejemplo en una máquina virtual, los programas corren sobre un hardware que no existe, y en un sandbox, los programas corren sin saber que existe fuera del sandbox y accediendo a un conjunto limitado de recursos.
2. Como mencionamos anteriormente, el Sandboxing es un ambiente controlado donde los programas se ejecutan. Y este, provee acceso restringido de un limitado número recursos del sistema, como puede ser memoria, acceso a redes, o a dispositivos de entrada, Un programa que se está ejecutando en un sandbox tiene solo los permisos que necesita, sin tener permisos que no necesita. De esta manera, el sandbox prevé que algún malware perjudique al resto del sistema.  
   El principal objetivo del Sandboxing es:
   * Prevenir que algún malware inyecte código en el proceso o en el kernel.
   * No permitir la ejecución de código de malware que ha sido ligado a algún área del proceso o kernel

La mayoría de los métodos de Sandboxing proveen un enfoque basado en aislamiento, donde el programa ejecutando en el sandbox está enteramente aislado de los recursos y de los programas ejecutando fuera del sandbox. En lugar de aislar totalmente a los programas, existe un enfoque donde se puede permitir a las aplicaciones que compartan recursos.

El paper presenta una lista de los objetivos de del diseño de un sandbox:

* El sandbox debe restringir el acceso a archivos fuera del sandbox.
* El sandbox debe restringir recursos , en especial la memoria y el tiempo de CPU, para prever que el contenido malicioso bloquee al sistema por completo.
* El sandbox debe limitar el número total de procesos para evitar una sobrecarga en el planificador de tareas.
* El sandbox debe monitorear y controlar la comunicación entre procesos.
* El sandbox debe controlar la comunicación del proceso en la red.
* El sandbox debe controlar el tiempo de acceso al sistema de los procesos.

Además el paper presenta mecanismos de sandboxing específicos. Hay diferentes enfoques que se pueden usar a la hora e implementar un mecanismo de sandbox, los que se presentan son los siguientes:

* Chroot
* Seccop-bbpf
* System Call Interposition
* Resource limit
* Linux security modules
* Namespaces
* Cgroups
* Capabilities
* Hypervisor

**Chroot:** en este mecanismo se cambia el directorio raíz a cualquier otro directorio. El programa, en esencia confinado a un ambiente, ve el directorio dado como la nueva raíz y no puede tener acceso a los archivos afuera de esta carpeta.

**Seccop:** este mecanismo se encarga de que la aplicación ejecutándose sobre el sandbox solo pueda realizar limitadas llamadas al sistema. Si el proceso quiere realizar alguna otra llamada al sistema, este proceso termina.

**System Call Interposition:** es un poderoso mecanismo para monitorear y regular el comportamiento de un programa interceptando sus llamadas al sistema. Esto ayuda al programador para tomar control del proceso.

**Cgroups:** es una característica de seguridad que limita y aísla el uso de los recursos. Básicamente es un grupo de procesos con un específico límite de recursos.

**Capabilities:** es un mecanismo donde los procesos no privilegiados consiguen un limitado set de permisos de root. De esta manera no se le conceden la totalidad de privilegios de root a procesos que no los necesitan en su totalidad.

**Resource Limit:** es una característica de seguridad donde cada recurso tiene asociado un par de límites, el límite blando, y el límite duro.

**Namespace:** esto provee la habilidad de aislar grupos de procesos, En Linux cada proceso está asociado a un namespace, y solo puede ver o acceder a los recursos asociado a este namespace

**Hypervisor based virtualization**: está considerada como el mecanismo más seguro para aplicaciones de sandboxing. En este tipo de virtualización, el hardware está multiplexado entre varios ambiente virtuales de operación llamados nodos virtuales. Cada nodo contiene un sistema operativo separado.

**Container Based Virtualization**: es una alternativa ligera para hypervisor based virtualization. Este mecanismo virtualiza el espacio de recursos del usuario, permitiendo una instancia separada del ambiente virtual llamado container, este comparte el mismo kernel.

**Linux security module**: este mecanismo es de propósito general, es un framework de control de accesos que activa diferente módulos para ser conectados al kernel.

Como hemos visto el sandboxing es un importante mecanismo que al acompañar a los métodos de seguridad tradicionales, añade defensa extra al sistema. En definitiva, protege tanto a las aplicaciones como al sistema operativo.

**Punto 2:**

El núcleo FDMP se maneja la exclusión mutua utilizando spinlocks.  
Para cada procesador se definieron dos tipos de bloqueos. Uno es llamado “bloqueo de tareas”, y el otro es llamado “bloqueo de objetos”. El primero de estos es usado para la exclusión mutua de las estructuras de datos relacionadas al control de tareas. Esta estructura de datos incluye el bloque de control de tareas (TCB). El segundo tipo de bloqueo, se usa para la exclusión mutua de las estructuras de datos relacionadas a la comunicación y sincronización. Estas estructuras de datos incluyen semáforos, flags de eventos y colas de datos.

**B)**

Los Sistemas multiprocesadores en Chip (MPSoCs “Multiprocessor systems-on-chip”) se han vuelto muy comunes en el diseño de sistemas embebidos para lograr un alto rendimiento y un bajo consumo de energía. Los MPSoCs se clasifican en tres tipos: por un lado se encuentran los MPSoCs simétricos con memoria compartida, donde todos los procesadores son homogéneos y comparten todos los recursos, incluidos la memoria principal y los periféricos. El segundo tipo son los MPSoCs asimétricos. Los procesadores pueden ser homogéneos o heterogéneos. Cada procesador tiene memoria local, a la que puede acceder cualquier procesador, a costa de un mayor tiempo de acceso.  
El tercer tipo son los MPSoCs con pasaje de mensajes, donde cada procesador no comparte memoria con el otro. En este tipo, cada procesador tiene su propia memoria privada, en la cual no puede acceder ningún otro procesador.   
TOPPERS/FDMP Kernel, para MPSoCs asimétricos, está basado en la especificación de ITRON, uno de los sistemas operativos de tiempo real (SOTR) más populares en varios lugares del mundo. A diferencia de ITRON, FDMP amplia ligeramente la especificación ITRON hacia MPSoCs asimétricos.

Para el desarrollo de FDMP se utilizaron ciertas técnicas, que son:   
- Bloques de control independientes: las estructuras de datos para controlar los objetos del kernel se denominan *bloques de control*. En el kernel FDMP, estos objetos son asignados estáticamente a un procesador específico. Es decir, cada procesador tiene su propio bloque de control.

* Llamadas al sistema entre procesadores: Aquí se tienen dos técnicas que se pueden utilizar. Un método es por *manipulación directa* donde un procesador accede directamente al bloque de control del otro procesador. El otro método está basado en *remote call*, donde el procesador envía una solicitud al otro procesador.   
  FDMP emplea la técnica de manipulación directa.
* Lock Unit: Al utilizar el método de manipulación directa, se debe garantizar la exclusión mutua entre los procesadores. El kernel FDMP utiliza spinlocks para asegurar la exclusión mutua. Una unidad de bloqueo (lock unit), representa un conjunto de recursos que son controlados por un bloqueo de exclusión mutua.
* Evitar Deadlock: Si todas las llamadas al sistema siguen cierto orden para adquirir un bloqueo no se produce un deadlock, sin embargo, en muchas llamadas al sistema esto no es posible. Para estas llamadas al sistema, primero se debe adquirir un bloqueo de tareas y luego un bloqueo de objetos. En el kernel FDMP, primero se adquiere el bloqueo de la tarea y se identifica el bloque de control del objeto al que se debe acceder. En este punto, se libera el bloqueo de tareas. Luego, se adquiere el bloqueo de objetos, y finalmente el bloqueo de tarea es adquirido.
* Exclusión mutua entre procesadores e inter-procesadores: Para la ejecución de ciertas llamadas al sistema, se requiere exclusión mutua para inter-procesadores e intra-procesadores. FDMP se da cuenta de la exclusión mutua en inter-procesadores usando spinlocks, y se da cuenta en intra-procesadores deshabilitando las interrupciones.

En cuanto a la evaluación del kernel FDMP se obtuvieron los siguientes resultados:

* Tamaño del Código:  
  Al comparar FDMP con el kernel TOPPERS/JSP, se pudo ver que el código de FDMP es un 60% más largo que el de JSP. Esto se debe a que, para cada llamada al sistema que se realiza, se agregó una rutina para adquirir y liberar bloqueos. También se agregó una nueva rutina para evitar deadlocks. Por otro lado, hubo un incremento trivial en las secciones de datos y bss.
* Performance:  
  Para realizar esta evaluación, se usaron dos llamadas al sistema, ambas en las mismas dos condiciones. En comparación con JSP, los tiempos son más largos incluso en el caso de llamadas entre procesadores. Esto se debe a que la rutina adicional para la exclusión mutua y las estructuras de datos son más complicadas.

En el kernel FDMP las tareas son asignadas estáticamente a los procesadores. La migración dinámica de tareas no es compatible. Por un lado, la migración dinámica es efectiva para balancear la carga a través de los distintos procesadores, de manera que ninguno este sin trabajar y se pueda aumentar la performance. Por el otro lado, sin embargo, la migración de tareas automática hace que sea difícil analizar y limitar el peor rendimiento.   
Basado en FDMP, se desarrolló otro SOTR llamado TOPPERS/FMP, que soporta la migración dinámica de tareas. Para no degradar el tiempo de respuesta del peor caso, este kernel no migra automáticamente las tareas a través de los procesadores. En lugar de esto, provee llamadas al sistema para migrar las tareas, por lo tanto es responsabilidad del programador decidir qué y cuándo una tarea debe migrar a otro procesador.

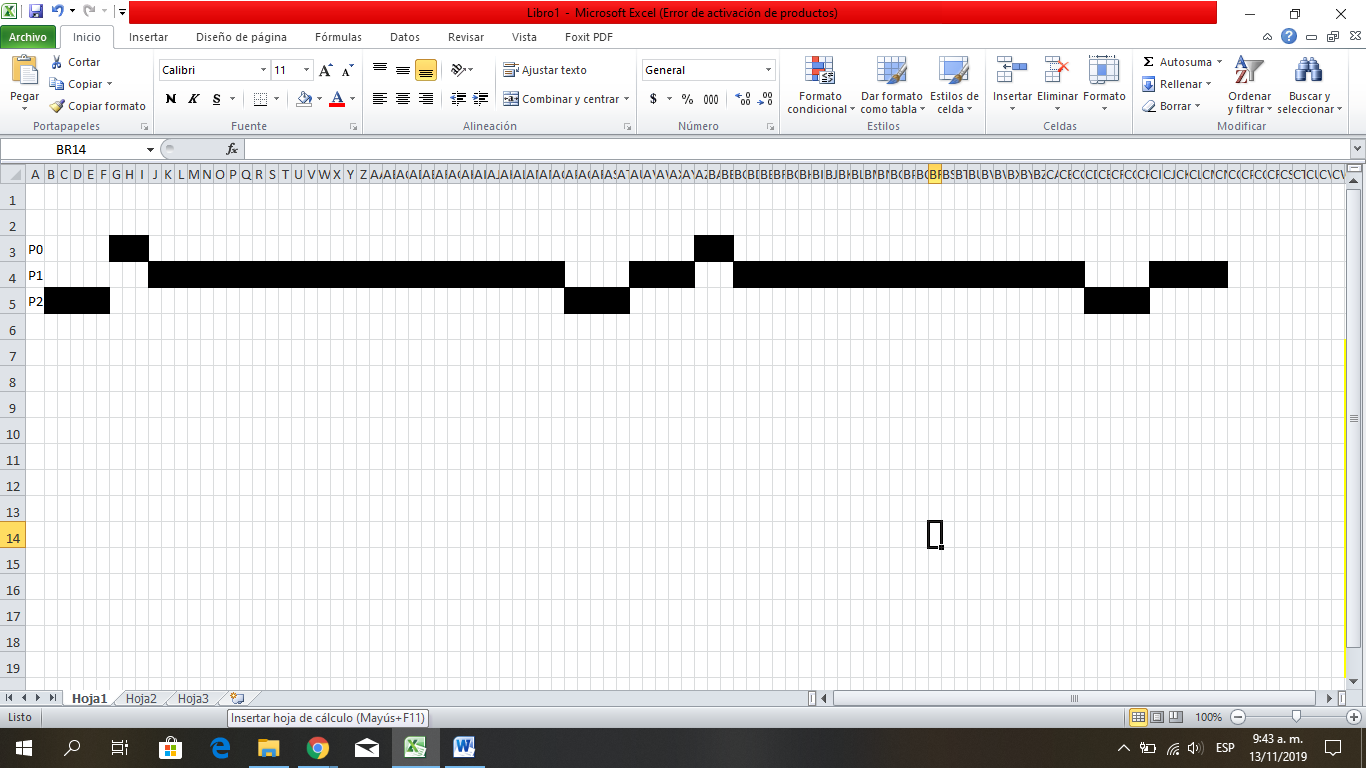
**Punto 3:**  
Conjunto 1  
**P0** t0 = 3 p0 = 50 t0/p0 = 0,06  
**P1** t1 = 70 p1 = 500 t1/p1 = 0,14  
**P2** t2 = 5 p2 = 40 t2/p2 = 0,125  
prior(P1)<prior(P0)<prior(P2)  
  
Conjunto 2  
**P0** t0 = 15 p0 = 50 t0/p0 = 0,3  
**P1** t1 = 5 p1 = 10 t1/p1 = 0,5  
**P2** t2 = 1 p2 = 4 t2/p2 = 0,25  
prior(P0)<prior(P1)<prior(P2)  
  
Conjunto 3  
**P0** t0 = 5 p0 = 20 t0/p0 = 0,25  
**P1** t1 = 7 p1 = 10 t1/p1 = 0,7  
**P2** t2 = 4 p2 = 100 t2/p2 = 0,04  
prior(P2)<prior(P0)<prior(P1)

1. - Para alcanzar una planificación EDF se tiene que asegurar que ⅀ ti/pi ≤ 1, donde i es la cantidad de procesos que se quieren planificar.  
   - Para alcanzar una planificación RM se debe garantizar la misma condición que en EDF y además para que un conjunto de procesos se pueda planificar con RM se debe asegurar que ⅀ ti/pi ≤ n(21/n-1), donde n es la cantidad de procesos que se quieren planificar. Esta condición es suficiente pero no necesaria, podría existir algún caso en el que esta condición sea falsa, y aun así el conjunto de procesos fuera planificable.  
     
   Para este ejercicio n = 3 y por lo tanto n(21/n-1) = 0,779   
   Conjunto 1:  
   ⅀ ti/pi = 0,325  
   Se puede planificar tanto en RM como en EDF ya que 0,325 < 0,779 < 1  
     
   Conjunto 2:   
   ⅀ ti/pi = 1,05  
   En este caso, no se puede planificar bajo ningún esquema ya que el resultado de la sumatoria es mayor a 1 y por lo tanto no cumple la restricción de EDF ni de RM.  
     
   Conjunto 3:  
   ⅀ ti/pi = 0,99  
   Bajo el esquema de planificación EDF se puede planificar sin problemas, sin embargo, bajo el esquema RM podría o no ser planificable ya que 0,99 no es menor a 0,779. En este caso, los procesos son planificables en RM.
2. **Planificaciones:**

**Conjunto 1:**   
Planificación con Rate Monotonic

prior(P0)<prior(P1)<prior(P2)

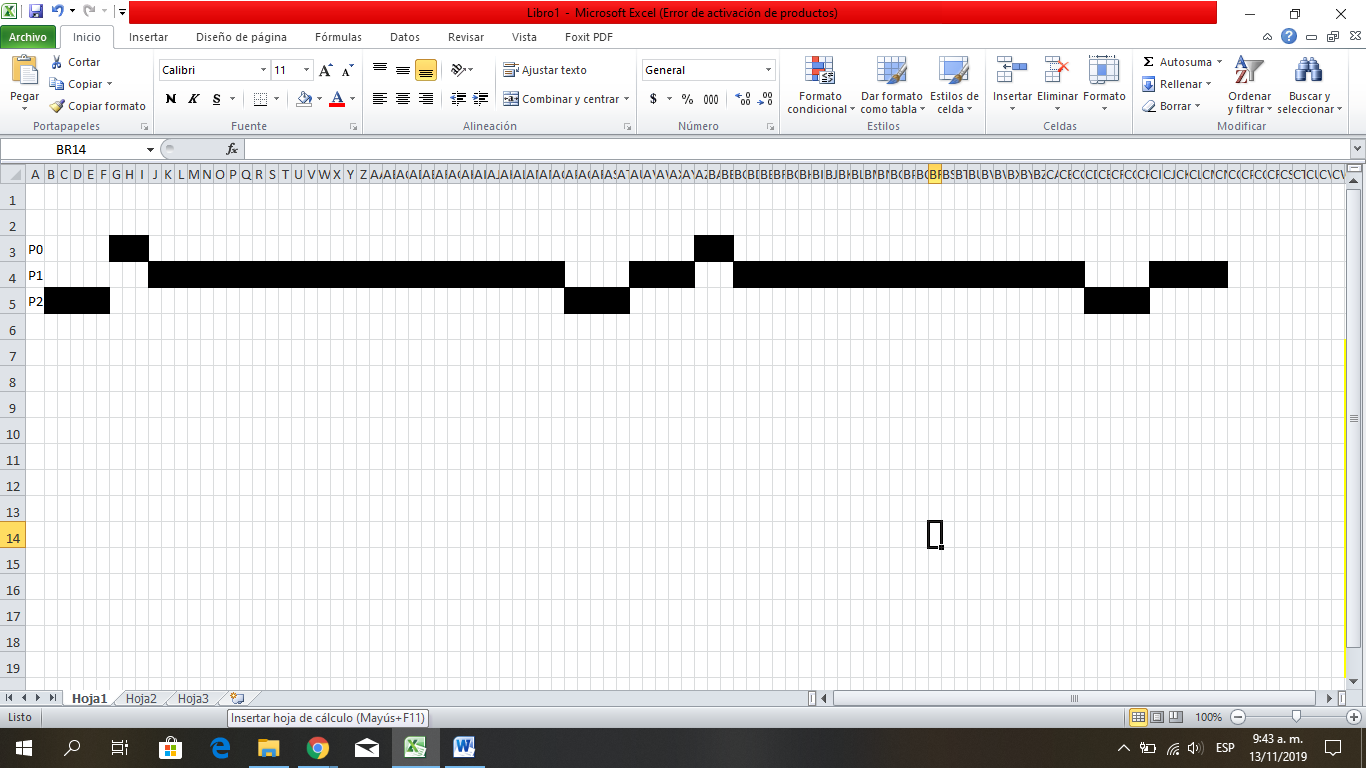
(P0,P1,P2) (P2) (P0) (P2) (P0)



0 10 20 30 40 50 60 70 80 90 100

Planificación con Earliest Deadline First  
Inicialmente prior(P0)<prior(P1)<prior(P2)

(P0,P1,P2) (P2) (P0) (P2) (P0)



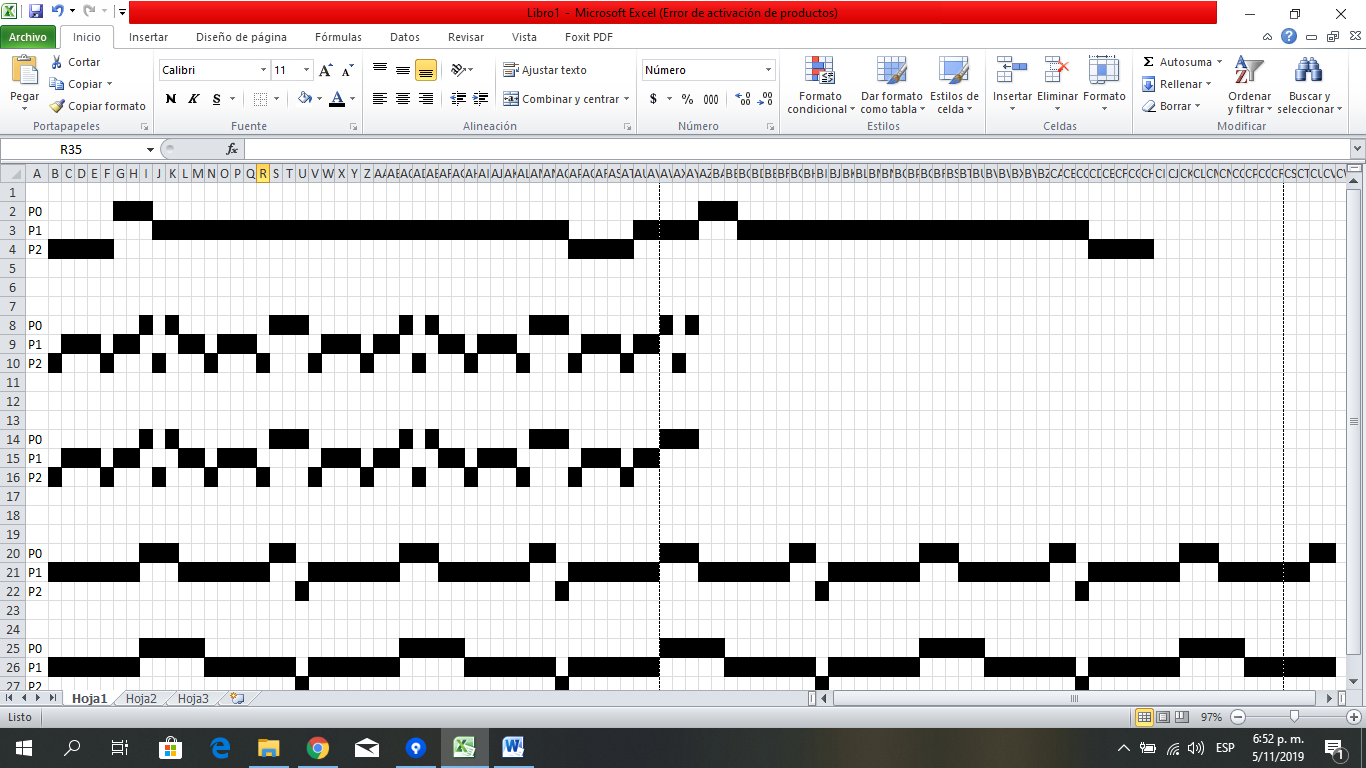
0 10 20 30 40 50 60 70 80 90 100

**Conjunto 2:**

Planificación con Rate Monotonic:

prior(P0)<prior(P1)<prior(P2)

(P0,P1,P2) (P2) (P2) (P1)(P2) (P2) (P2,P1) (P2) (P2)(P1) (P2) (P2) (P2,P1) (P2) (P2)(P0,P1)

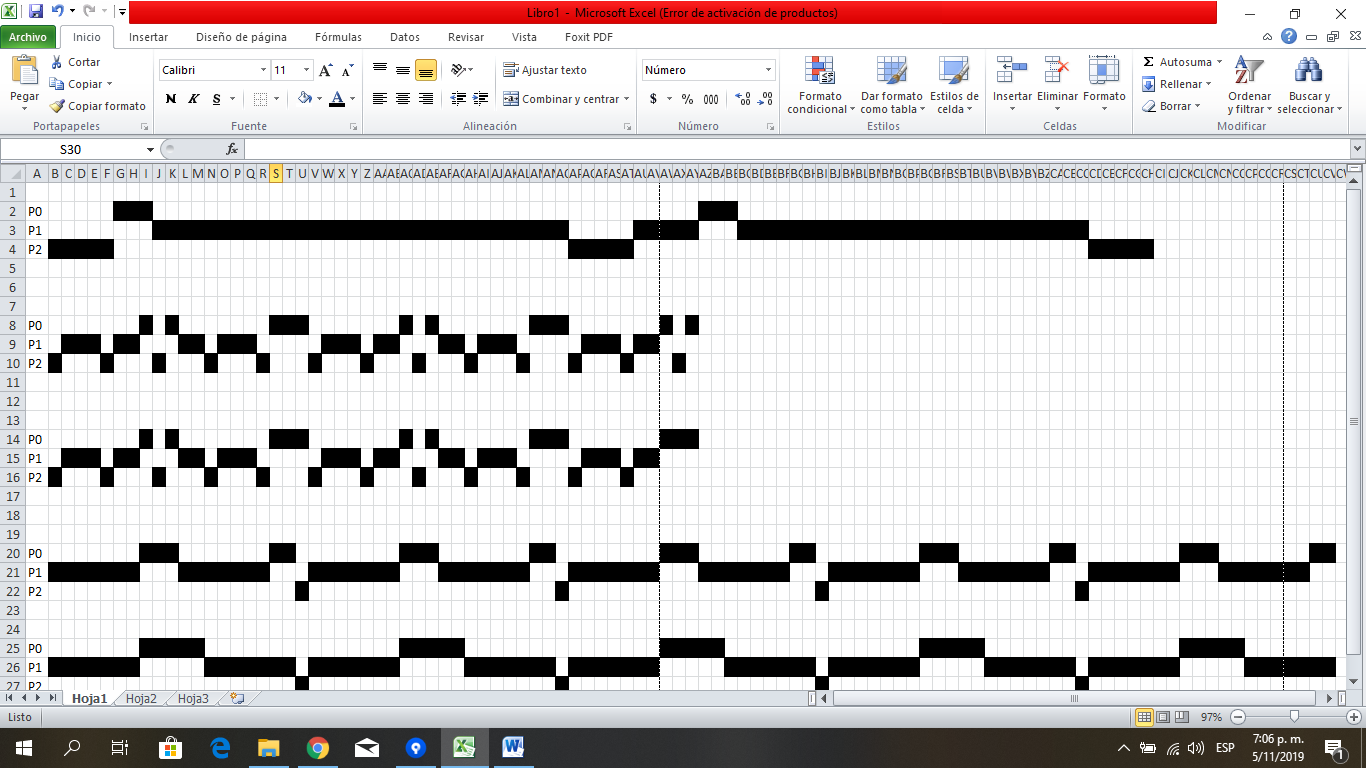


0 10 20 30 40 50\*

(\*) Aquí es donde no se respeta el deadline del proceso P0, ya que cuando llega el nuevo periodo de este proceso, aún falta ejecutar 3 unidades de tiempo del periodo anterior.

Planificación con EDF:  
Inicialmente prior(P0)<prior(P1)<prior(P2)

(P0,P1,P2) (P2) (P2) (P1)(P2) (P2) (P2,P1) (P2) (P2)(P1) (P2) (P2) (P2,P1) (P2) (P2)(P0,P1)



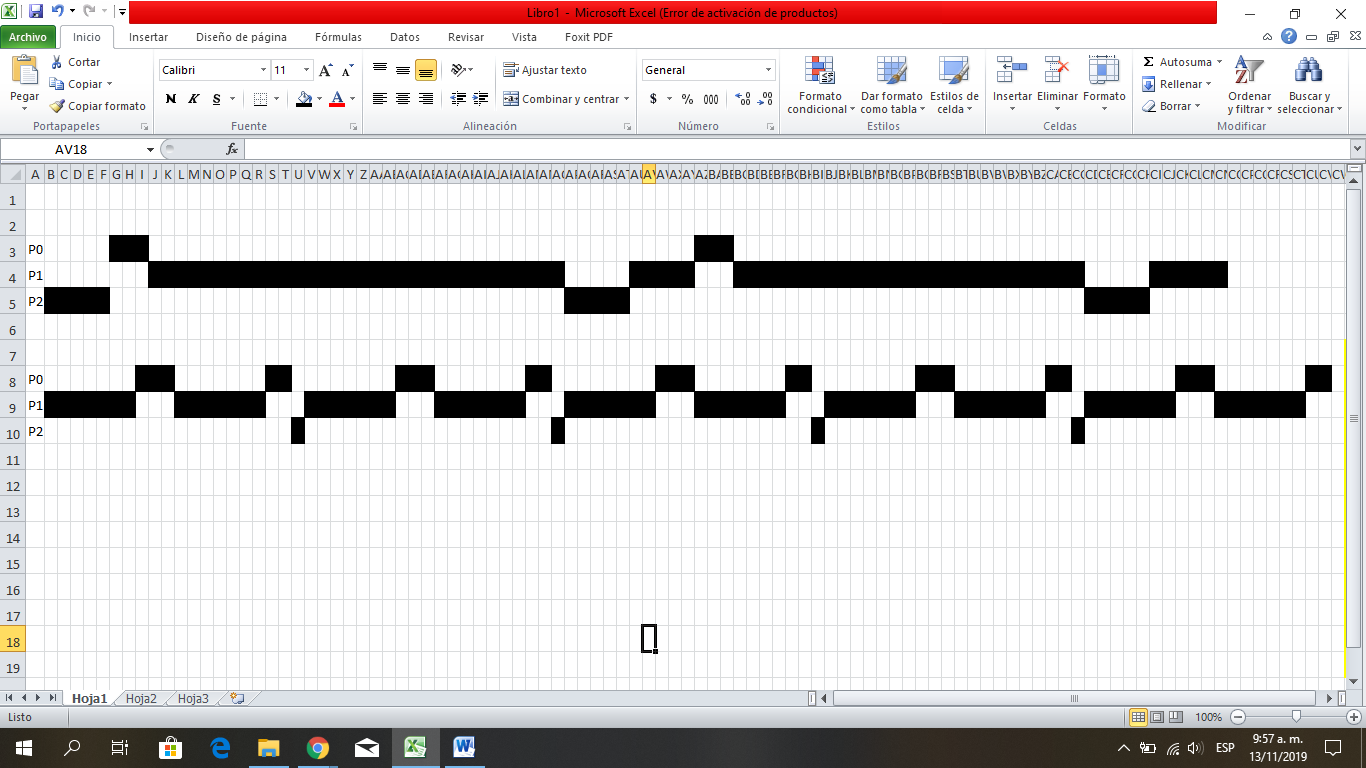
0 10 20 30 40 50\*

(\*) En este punto es donde no se cumple el deadline del proceso P0, aunque con este esquema se mejora un poco más que con el esquema anterior, ya que en este caso, quedan dos unidades de tiempo sin ejecutar antes de que llegue el nuevo periodo el proceso.

**Conjunto 3:**   
Planificación con Rate Monotonic

prior(P2)<prior(P0)<prior(P1)

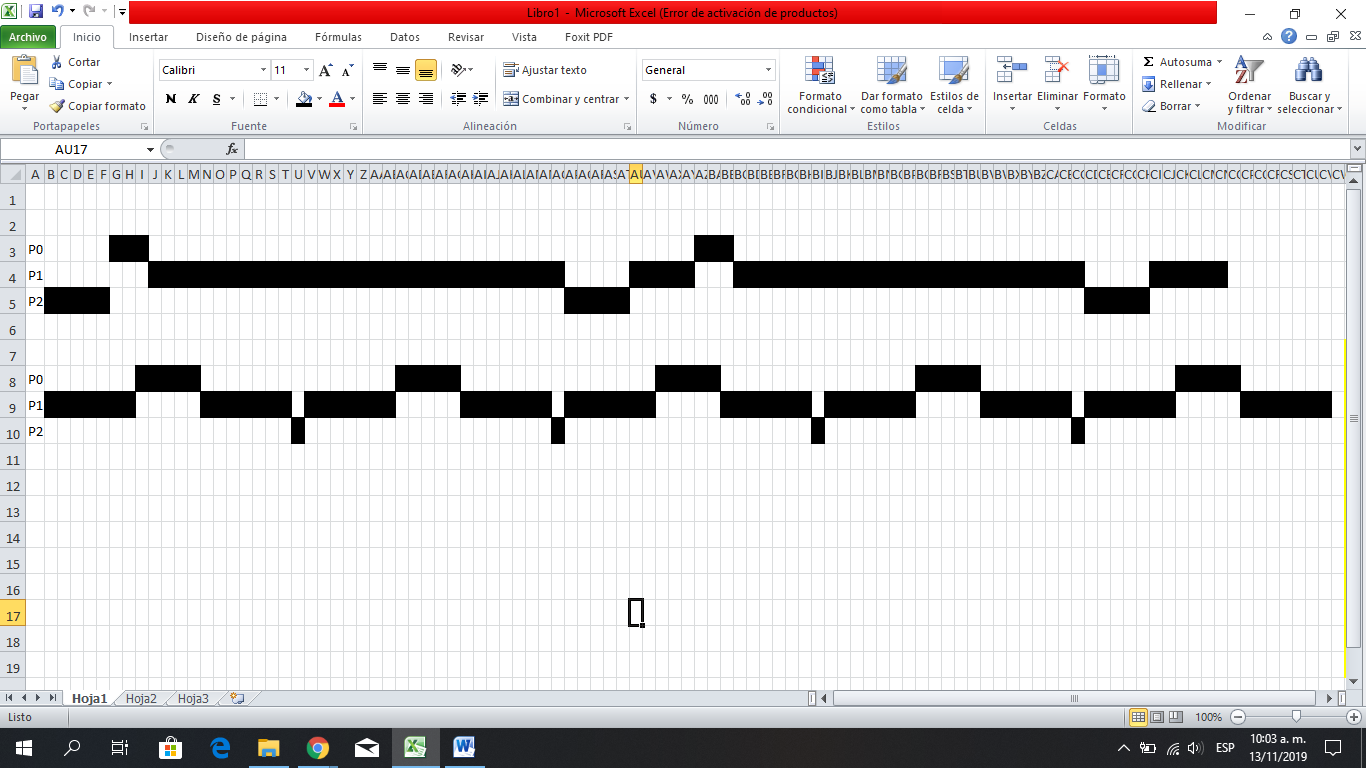
(P0,P1,P2) (P1) (P0,P1) (P1) (P0,P1) (P1) (P0,P1) (P1) (P0,P1) (P1) (P0,P1,P2)



0 10 20 30 40 50 60 70 80 90 100

Planificación con Earliest Deadline First:   
Inicialmente prior(P2)<prior(P0)<prior(P1)

(P0,P1,P2) (P1) (P0,P1) (P1) (P0,P1) (P1) (P0,P1) (P1) (P0,P1) (P1) (P0,P1,P2)



0 10 20 30 40 50 60 70 80 90 100

**Punto 4:**

a)

Como se necesitan 4 bits para movernos entre los dos niveles de páginas, los bits restantes de la dirección, esto es 12 bits, corresponden al desplazamiento dentro de cada página. Por lo tanto cada página tendrá un tamaño de 212 bytes = 4KB. Los marcos tienen el mismo tamaño que las páginas.

b)

El espacio virtual utilizado es 8 (paginas) x 4 KB = 32 KB.

La memoria residente en bytes es 4 (frames) x 4 KB = 16 KB = 16384 B.

c)

1)

Situación inicial:

**PTBR**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | v |
|  | v |
| 0x00 | i |
| 0x00 | i |

|  |  |
| --- | --- |
| 0xFD | v |
| 0x64 | v |
| 0xAA | i |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | v |
| 0xA1 | v |
| 0xAA | i |
| 0xA1 | i |

* Los primeros cinco accesos que se realizan, no cambian el estado inicial de la tabla de páginas, ya que son exitosos.
* En el siguiente acceso, que es el (1,3,11), se produce un miss ya que esta entrada en la tabla de páginas es invalida. Por lo tanto, se debe hacer uso del algoritmo de reemplazo LRU, y cambiar el contenido de la entrada (1,3) por el contenido de la página menos usada.

**PTBR**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | v |
|  | v |
| 0x00 | i |
| 0x00 | i |

|  |  |
| --- | --- |
| 0xFD | v |
| 0x64 | v |
| 0xAA | i |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | v |
| 0xA1 | i |
| 0xAA | i |
| 0xA1 | v |

* En el siguiente acceso, (1,1,F0), se vuelve a producir un miss, y se debe hacer uso del algoritmo de reemplazo. Luego de las modificaciones, el estado de la tabla es el siguiente.

**PTBR**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | v |
|  | v |
| 0x00 | i |
| 0x00 | i |

|  |  |
| --- | --- |
| 0xFD | v |
| 0x64 | v |
| 0xAA | i |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | i |
| 0xAA | v |
| 0xAA | i |
| 0xA1 | v |

* Con el siguiente acceso, (1,2,40), se produce nuevamente un miss.

**PTBR**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | v |
|  | v |
| 0x00 | i |
| 0x00 | i |

|  |  |
| --- | --- |
| 0xFD | v |
| 0x64 | i |
| 0xAA | i |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | i |
| 0xAA | v |
| 0x64 | v |
| 0xA1 | v |

* Con los siguientes dos accesos se produce un hit, por lo tanto el estado de la tabla de paginas no se modifica.
* Con el último acceso se produce un miss. El estado de la tabla de paginas resultante es el siguiente.

**PTBR**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | v |
|  | v |
| 0x00 | i |
| 0x00 | i |

|  |  |
| --- | --- |
| 0xFD | v |
| 0xA1 | v |
| 0xAA | i |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | i |
| 0xAA | v |
| 0x64 | v |
| 0xA1 | i |

2)

Situacion Inicial:

**PTBR**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | v |
|  | v |
| 0x00 | i |
| 0x00 | i |

|  |  |
| --- | --- |
| 0xFD | v |
| 0x64 | v |
| 0xAA | i |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | v |
| 0xA1 | v |
| 0xAA | i |
| 0xA1 | i |

* Luego de los primeros 5 accesos, el estado de la tabla de páginas es el mismo, ya que todos los accesos son exitosos.
* Con el acceso (1,3,11), se produce un miss, ya que esta entrada es invalida en la Tabla de páginas. Como se está usando el algoritmo de reemplazo optimo, la mejor entrada a reemplazar seria la entrada (1,0) ya que en la secuencia de accesos, esta entrada no se accede nunca. Por lo tanto, el estado de la tabla de páginas seria el siguiente.

**PTBR**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | v |
|  | v |
| 0x00 | i |
| 0x00 | i |

|  |  |
| --- | --- |
| 0xFD | v |
| 0x64 | v |
| 0xAA | i |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | i |
| 0xA1 | v |
| 0xAA | i |
| 0xAA | v |

* El estado de la tabla de páginas no se modifica con el siguiente acceso ya que es un hit.
* Con el acceso (1,2,40), se produce un miss. En este caso, la mejor entrada para reemplazar seria la entrada (1,1) o la entrada (1,3) ya que ninguna de las dos se vuelve a acceder en la secuencia. En este caso elegimos la entrada (1,1).

**PTBR**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | v |
|  | v |
| 0x00 | i |
| 0x00 | i |

|  |  |
| --- | --- |
| 0xFD | v |
| 0x64 | v |
| 0xAA | i |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | i |
| 0xA1 | i |
| 0xA1 | v |
| 0xAA | v |

* Con los siguientes 3 accesos, (0,0,20), (0,0,39) y (0,1,3), no se produce ningún cambio en la tabla de páginas ya que todas las entradas son válidas.

3)

En la parte 1, donde se utiliza el algoritmo de reemplazo LRU, se producen 4 fallos.

En la parte 2, donde se utiliza el algoritmo de reemplazo optimo, se producen 2 fallos.

d)

Si el proceso realiza la operación M(3,3,33), se produce una violación de acceso, ya que en las entradas 2 y 3 este proceso tiene prohibido acceder.

e)

Luego de que el proceso solicita la nueva página, el estado de la tabla de páginas es el siguiente.

**PTBR**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | v |
|  | v |
|  | v |
| 0x00 | i |

|  |  |
| --- | --- |
| 0xFD | v |
| 0xA1 | v |
| 0xAA | i |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | i |
| 0xAA | v |
| 0x64 | v |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | i |
| 0x00 | i |
| 0x00 | i |
| 0x00 | i |

Luego de cargar la nueva página e intentar acceder a esta nueva entrada en la tabla de páginas, se produce un miss, por lo que se debe utilizar el algoritmo de reemplazo para cargar la página correspondiente. El estado final de la tabla de páginas es el siguiente.

**PTBR**

|  |
| --- |
|  |

|  |  |
| --- | --- |
|  | v |
|  | v |
|  | v |
| 0x00 | i |

|  |  |
| --- | --- |
| 0xFD | v |
| 0xA1 | v |
| 0xAA | i |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | i |
| 0xAA | i |
| 0x64 | v |
| 0xA1 | i |

|  |  |
| --- | --- |
| 0xAA | v |
| 0x00 | i |
| 0x00 | i |
| 0x00 | i |