



# Proyecto

06.11.2024

---

Universidad Nacional del Sur

Sistemas Operativos

Alumnos: Juan Pablo Antonelli, Gerónimo Infante

Grupo: 12



# Índice

## Procesos, threads y comunicación

- [Pumper Nic con pipes](#)
- [Pumper Nic con cola de mensaje](#)
- [Minishell](#)

## Sincronización

- [Taller de motos](#)
- [Santa Claus](#)

## Problemas conceptuales

- [Traducción con sistema de paginación](#)
- [Traducción con sistema de segmentación](#)
- [Tabla de páginas y algoritmo de reemplazo LRU](#)

# Experimentación de Procesos y Threads con los Sistemas Operativos

## 1.1 Procesos, threads y comunicación

### *1.PUMPER NIC.*

Antes de comenzar con la explicación y el desarrollo de los incisos de dicho punto, fue de gran utilidad identificar las siguientes entidades en el problema, que luego se desarrollaron como procesos :

- Usuario y usuarioVIP: Por diseño, ambos fueron implementados mediante la misma función; se determina si es VIP o no a través del valor de una variable local.
- Despachador: Encargado de tomar pedidos e indicar, según la orden, qué empleado debe realizar la tarea correspondiente.
- Empleado encargado de hamburguesas.
- Empleado encargado de menú vegano.
- Dos empleados encargados de las papas.

### **a) Implementación utilizando procesos y pipes para la comunicación entre los participantes**

Para utilizar pipes como medio de comunicación entre procesos, es necesario establecer una relación de padre-hijo. El proceso padre se encarga de crear todos los pipes antes de generar a sus procesos hijos, de modo que cada uno de ellos pueda acceder a estos pipes posteriormente.

Teniendo esto en cuenta y habiendo identificado las entidades que caracterizan el ejercicio, se decidió que el proceso padre sería el despachador. Antes de cumplir su tarea principal, el despachador crea los pipes necesarios y los procesos empleados (empleadoHamburguesa, empleadoVegano, empleadoPapas1 y empleadoPapas2), así como a los procesos de los clientes (cuatro de ellos).

La comunicación entre procesos mediante pipes se ilustra en la [Figura 1](#), donde cada rectángulo representa un proceso, y las conexiones muestran los pipes. Las flechas indican la dirección de la comunicación, partiendo desde el proceso que escribe en el pipe y apuntando hacia el proceso que lee.

### Funcionamiento:

Ambos tipos de clientes crean una orden y la envían a través de su pipe correspondiente, según si son VIP o no, hacia el despachador. El despachador, siempre que haya una orden de un cliente VIP, la tomará primero; si no hay órdenes de clientes VIP, tomará una de los clientes regulares. Esto es posible gracias a que el pipe *OrdenClienteVip* (marcado en rojo en la figura) se implementó como un pipe no bloqueante.

Una vez que el despachador recibe una orden, determina lo que el cliente solicitó y envía a cada empleado los mensajes necesarios a través del pipe correspondiente, según la cantidad de hamburguesas, menús veganos o papas pedidas por el cliente.

Al finalizar la preparación de los alimentos, los empleados envían mensajes a través de pipes compartidos con todos los clientes, indicando que el alimento está listo para retirarse. Si dos o más clientes solicitaron el mismo alimento, deberán competir para ver quién lo toma primero. Una vez que un cliente ha recolectado todo lo que pidió, se retira.

Como elemento adicional especificado en el enunciado, y debido a la complejidad de determinar cuántos clientes están esperando en un pipe, se implementó una variable entera local para cada cliente. Antes de hacer un pedido, cada cliente consulta esta variable; si cumple ciertos criterios, se considera que hay demasiada gente en la cola de pedidos y el cliente se retira, de lo contrario, procede a hacer su pedido.

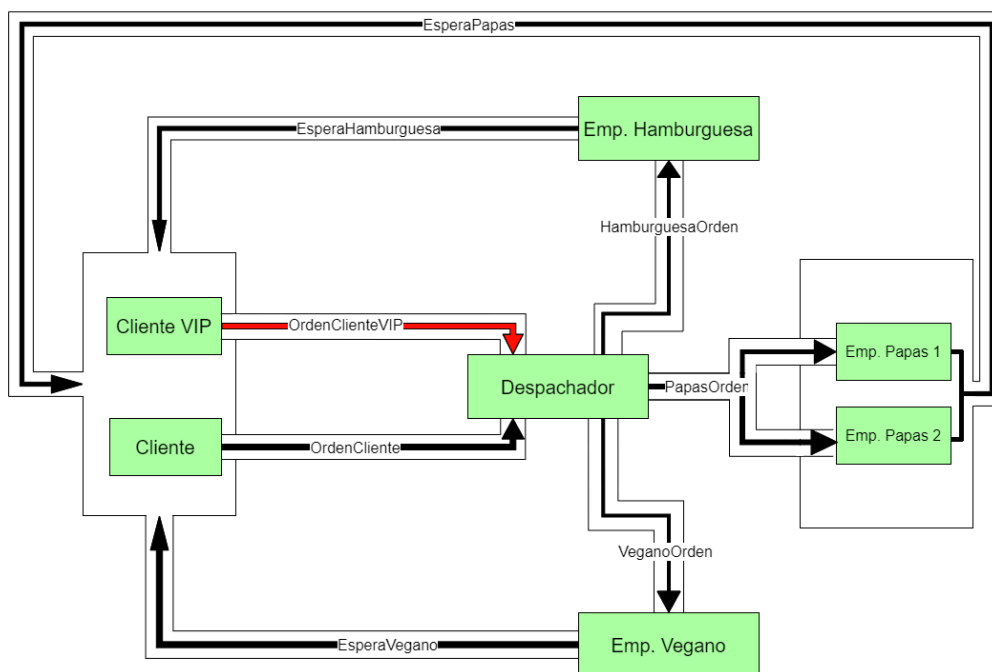


Figura 1: Esquema de comunicación mediante pipes

### Possible problema:

Dado que solo uno de los pipes está configurado como no bloqueante, puede surgir una situación de deadlock en el programa. Esto ocurre cuando, en un instante  $t$ , no hay clientes VIP, por lo que el despachador lee una orden de un cliente no vip. Al finalizar la atención, el despachador vuelve a verificar si hay clientes VIP para atender; al no encontrar ninguno, pasa a revisar si hay clientes no vips en el instante  $t+1$ . Si en este momento no llega ningún cliente no vip adicional y solo llegan clientes VIP, estos no serán atendidos, ya que el despachador permanecerá bloqueado esperando por clientes normales, generando así una situación de deadlock.

Una posible solución sería implementar ambos pipes, tanto *ordenCliente* como *ordenClienteVIP*, en modo no bloqueante. Sin embargo, esto generaría una espera ocupada, lo cual es muy costoso en términos de rendimiento, por lo que no es una solución ideal. Además, dado que en una situación real la proporción de clientes no VIP suele ser mayor que la de clientes VIP, esta situación tendría una baja probabilidad de ocurrir. Por lo tanto, no se considera necesario tomar precauciones adicionales al respecto.

### Compilacion y ejecucion:

Ejecutar el script "scriptEjecutarPipes.sh", el cual compila el archivo *pumperNicPipes.c* y lo ejecuta.

## **b) Implementación utilizando procesos y cola de mensaje para la comunicación entre los participantes**

En esta implementación, se utilizó una única cola de mensajes para la comunicación entre procesos. Para lograr la sincronización de tareas, se desarrollaron diferentes tipos de mensajes: cinco tipos para la coordinación de tareas y uno adicional que permite determinar si hay demasiadas personas en la cola, permitiendo a los clientes abandonar la espera si es necesario. La estructura de esta comunicación se muestra en la [Figura 2](#), donde las flechas indican los mensajes enviados para la sincronización, y el tipo de mensaje correspondiente se muestra con un número en la parte inferior derecha de la imagen.

Un proceso principal ejecuta el código *PumperNicColaMensajes.c*, encargándose de crear los procesos para el despachador, empleadoHamburguesa, empleadoVegano, empleadoPapas1, empleadoPapas2 y 12 clientes. Además, envía 10 mensajes a la cola con el tipo de mensaje *COLAESPERA*, indicando que el número máximo de clientes que un cliente puede tolerar en la cola de espera es de 10, antes de decidir irse sin hacer un pedido.

### Funcionamiento:

Antes de realizar un pedido, todo cliente, ya sea VIP o no VIP, verifica si hay mensajes de tipo *COLAESPERA*. Si no encuentra mensajes de este tipo, el cliente decide retirarse. De lo contrario, prepara su orden y la envía a la cola de mensajes, especificando el tipo (*CLIENTEVIP* para clientes vip o *CLIENTE* para clientes regulares). Para dar prioridad a las órdenes de clientes VIP, el despachador lee la cola con el tipo *ATENDERCONPRIORIDAD*, cuyo valor es -2, lo que permite tratar la cola como una cola de prioridad, donde 1 corresponde a *CLIENTEVIP* y 2 a *CLIENTE*. De esta forma, siempre que haya una orden de un cliente VIP, el despachador la toma primero; si no hay mensajes de clientes VIP, entonces toma una orden de cliente regular. Tras recibir una orden, el despachador envía tantos mensajes como productos solicitados, utilizando los tipos *EMPLEADOHAMBURGUESA*, *EMPLEADOPAPAS* y *EMPLEADOVEGANO*, según corresponda.

Finalmente, dado que cada proceso posee un PID único en todo el sistema, se decidió que la notificación de los alimentos preparados por los empleados se envíe al cliente mediante mensajes cuyo tipo corresponda al PID del cliente que realizó la orden. De esta manera, si el cliente A y el cliente B pidieron ambos una hamburguesa, no habrá competencia por el mismo producto, sino que cada cliente recibirá los alimentos según el orden en que se procesaron sus pedidos. De ello cada cliente una vez que pide la orden espera por mensajes con su respectivo PID como tipo. Para identificar el tipo de alimento recibido, el mensaje contiene una variable *entrega* de tipo char, que puede tener los siguientes valores:

- H → Se entrega una hamburguesa.
- V → Se entrega un combo vegano.
- P → Se entrega una porción de papas.

Esto permite que el cliente, al recibir un mensaje, pueda verificar el valor de la variable *entrega* y entender qué alimento ha recibido. Así, cada cliente espera tantos mensajes con su PID como la suma de productos que pidió; una vez que ha recibido todos los alimentos solicitados, el cliente se retira.

### Compilacion y ejecucion:

Ejecute el script *scriptEjecutar.sh*, el cual se encarga de compilar los archivos, creando los ejecutables necesarios y configurando las rutas utilizadas por el proceso principal *PumperNicColaMensajes.c* para asignar estos ejecutables al crear los procesos hijos. Una vez realizada la compilación, para ejecutar, Para ejecutar el programa, ingrese el comando *./main* directamente en la consola. Si *./main* se incluye dentro de un script, es posible que la consola se cierre al finalizar *PumperNicColaMensajes.c*, que es el proceso encargado de crear los demás procesos y la cola de mensajes. Ejecutando *./main* manualmente en la consola, se evita que el cierre la consola temporal y permite que el programa funcione correctamente.

Además, se incluye el archivo *eliminadorCola.c*, que también es compilado por el script y produce un ejecutable denominado *"rm"*. En caso de ser necesario ejecutar el programa varias veces, se debe eliminar la cola de mensajes creada al final de cada ejecución para asegurar su correcto funcionamiento, para ello ejecute *"/rm"*.

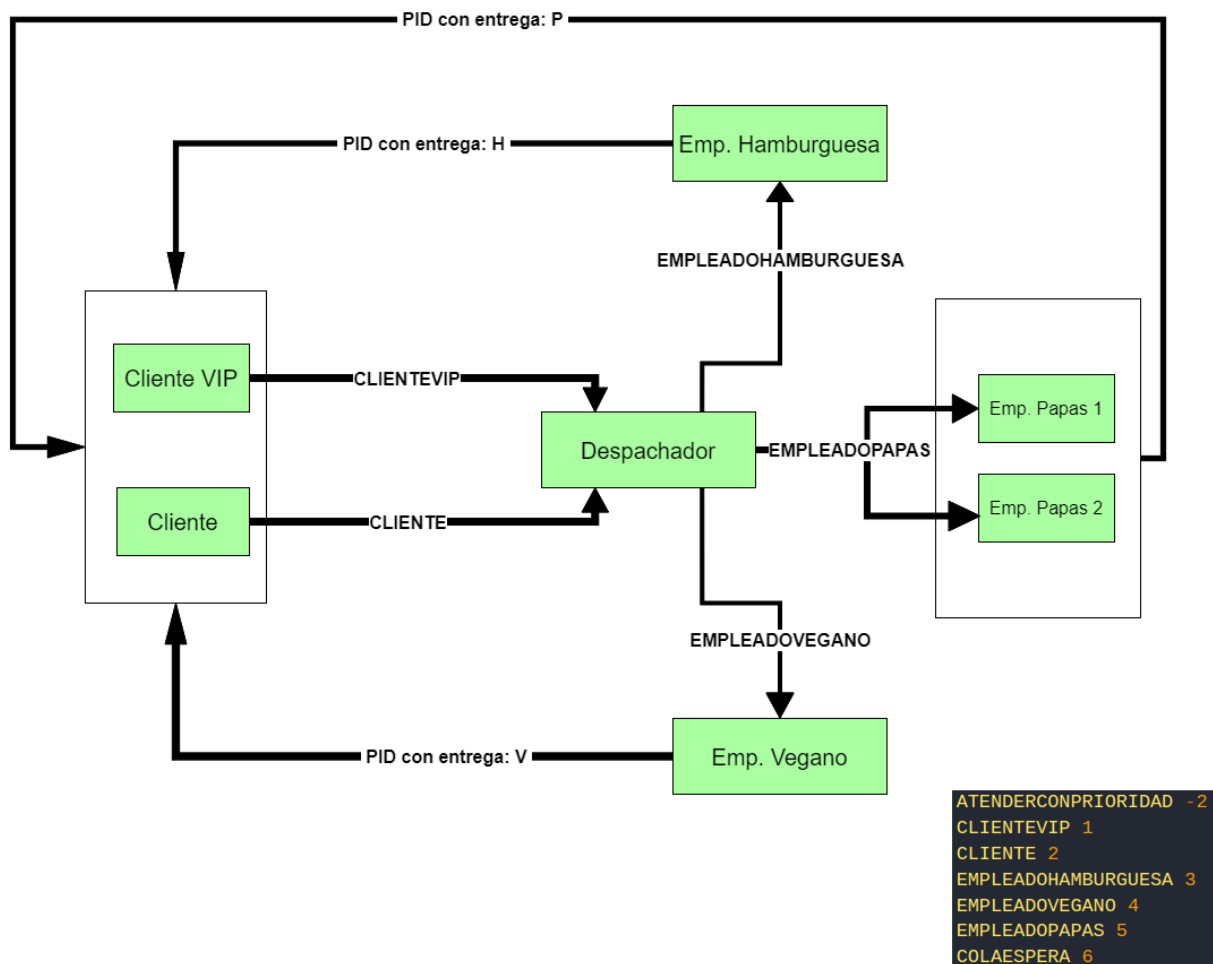


Figura 2: Esquema de comunicación mediante cola de mensajes

#### Ventajas sobre pipes:

- **Facilidad para implementar prioridades en la lectura de mensajes:** La función `msgrcv` permite leer mensajes en orden de prioridad simplemente indicando un tipo de mensaje negativo, lo que facilita la atención preferencial a los clientes VIP.
- **Optimización en el uso de recursos:** Con una cola de mensajes, no es necesaria una relación padre-hijo entre procesos, lo que permite que cada proceso, una vez

creado, cargue su propia imagen ejecutable. Esto evita que todos los procesos sean clones del mismo proceso padre.

- **Mejor comunicación y escalabilidad:** La cola de mensajes permite que el programa se amplíe sin realizar cambios significativos. Si se desean agregar nuevos empleados o menús, solo es necesario conectarlos a la cola de mensajes y definir un nuevo tipo de mensaje para la sincronización, en lugar de crear nuevos pipes y gestionar los extremos en cada proceso que no los utiliza, lo cual incrementaría la complejidad.
- **Claridad en el código:** Cada entidad tiene su propia imagen ejecutable, lo que facilita la depuración y la lectura del código al hacerlo más modular y comprensible.

## 2.MINISHELL

El objetivo de ésta minishell es proveer un conjunto limitado de comandos básicos, sin usar la función `system`. Para ello se implementó cada uno de los comandos usando llamadas al sistema. A su vez es importante notar que una shell por sí sola se encarga de comprender el comando requerido por el usuario, y crea un proceso cargando la imagen ejecutable correspondiente al programa de ese comando. Las decisiones de diseño para cada comando fueron las siguientes:

- Ayuda: Muestra una lista de los posibles procesos con una breve descripción de cada uno, como así los parámetros que este necesite entre '`<`' '`>`'.
- Crear un directorio: Caracterizado por el nombre `mkdir`, se ejecuta con un parámetro, el `path/nombre` que llevará el directorio a crear. Para su implementación se utilizó la llama al sistema `mkdir(path, permiso)` la cual se encuentra disponible en la librería `sys/stat.h`, donde por defecto se setean todos los permisos (0777).
- Eliminar un directorio: Caracterizado por el nombre `rmdir`, el cual requiere como parámetro el nombre del directorio a eliminar. Por decisión en el desarrollo, se implementó que si el mismo posee archivos en su interior, también serán eliminados. Para su implementación se utilizaron las funciones `readdir()`, para determinar lo que tenga en su interior, de forma que si posee elementos no directorios, los elimine con `remove()` y si se trata de directorios se vuelva a llamar recursivamente. En caso de que el directorio está vacío lo elimina con la llamada al sistema `rmdir(nombreDirectorio)`.
- Crea un archivo: Caracterizado por nombre `crear_archivo`, requiere como parámetro, el `path/nombre` que llevará el archivo. Para su implementación se utilizó la llamada `open()`, donde se indicó que en caso de que no exista el archivo lo cree con permisos de lectura y escritura (0666). Pero si existe lo abre con los permisos de escritura.



- Listar el contenido de un directorio: Caracterizado por el nombre `ls`, no requiere ningún parámetro. Para su implementación se utilizó la llamada `readdir()`, la cual se indica el directorio actual y retorna una estructura de tipo `dirent`, la cual en su interior posee una variable que indica el nombre del archivo o directorio que leyó dentro del directorio actual. De esta forma se invoca tantas veces la llamada hasta que devuelva NULL, indicando que ya leyó todo lo que contenía el directorio.
- Mostrar el contenido de un archivo: Caracterizado por el nombre `cat`, requiere como parámetro el nombre del archivo a leer. Para su implementación se utilizó la llamada `open()`, con el archivo indicado abriéndolo solo como lectura, luego para la muestra del contenido, se utilizó la llamada `read()` sobre el archivo y mostrándolo por consola mediante la llamada `write()`.
- Modificar los permisos de un archivo: Caracterizado por el nombre `chmod`, requiere como parámetro, el path/name del archivo a cambiar sus permisos, como así los permisos a indicar. Para su implementación se utilizó la llamada al sistema `chmod`.

### Compilacion y ejecucion:

Ejecutar el script "scriptEjecutarShell.sh", el cual compila a todos los archivos correspondientes a los comandos, como así al proceso principal shell.c.

## 1.2 Sincronización

### 1. Taller de motos:

Este ejercicio de sincronización consiste en coordinar la secuencia correcta para la fabricación de dos motos. La secuencia sigue el siguiente orden: primero se fabrican dos ruedas, luego se arma el chasis, se agrega el motor, se pinta la moto, y este ciclo se repite para la segunda moto. A esta segunda moto se le agrega un ítem extra, y luego el proceso vuelve a comenzar. La secuencia se representa como: "RRCMPRRCMPE", donde las letras significan:

- R → Fabricación de una rueda
- C → Armado del chasis
- M → Agregado del motor a la moto
- P → Pintado de la moto en color rojo(P\_rojo) o verde(P\_verde)
- E → Agregado del ítem extra a la moto

Cada paso de esta secuencia representa una entidad implementada como un hilo. Para lograr la sincronización adecuada entre los hilos, se utilizan seis semáforos:

- *SecuenciaSem* → Controla la producción de cuatro ruedas para coordinar que el fabricante de ruedas se detenga y permita al encargado del ítem extra continuar. Inicializado en 4.
- *RuedaSem* → Permite que el fabricante de ruedas produzca ruedas. Inicializado en 2.


- *ChasisSem*→Permite que el operario arme el chasis. Inicializado en 0.
- *MotorSem*→ Permite que el operario ponga el motor. Inicializado en 0.
- *PintorSem*→ Permite que los pintores compitan por el semáforo para pintar la moto de verde o rojo. Inicializado en 0.
- *ExtraSem*→ Permite que el operario encargado del ítem extra lo agregue, una vez finalizada la fabricación de dos motos. Inicializado en 0.

### Modelo:

<i>Rueda</i>	<i>Chasis</i>	<i>Motor</i>	<i>Pintor Verde</i>	<i>Pintor Rojo</i>	<i>Extra</i>
wait(sequenciaSem)	wait(chasisSem)	wait(motorSem)	wait(pintorSem)	wait(pintorSem)	wait(extraSem)
wait(ruedaSem)	wait(cuadroSem)	<b>Produce Motor</b>	<b>Pinta de Verde</b>	<b>Pinta de Rojo</b>	wait(extraSem)
<b>Produce Rueda</b>	<b>Arma Chasis</b>	signal(pintorSem)	signal(extraSem)	signal(extraSem)	<b>Agrega Extra</b>
signal(chasisSem)	signal(motorSem)		signal(CuadroSem)	signal(CuadroSem)	signal(sequenciaSem)
			signal(CuadroSem)	signal(CuadroSem)	signal(sequenciaSem)
					signal(sequenciaSem)
					signal(sequenciaSem)

### Funcionamiento:

Al inicial los semáforos contadores *ruedaSem* en 2 y *sequenciaSem* en 4, permite que se fabriquen dos ruedas. Al finalizar cada una, se avisa al operario encargado de armar el chasis, con 2 ruedas creadas se tienen 2 avisos de *chasisSem*, por lo tanto arma el chasis y avisa al empleado encargado del motor que lo ponga, a partir de la sincronización con el semáforo *motorSem*. Luego de poner el motor, éste avisa que se tiene que pintar la moto mediante el semáforo *pintorSem*, donde tanto el pintor de verde como el de rojo compiten para determinar quién pintará la moto, una vez pintada (primer moto), se avisa al operario encargado del ítem extra y se mandan a crear 2 ruedas más. Ya que *sequenciaSem* posee un valor restante de 2 y como el operario encargado del ítem extra necesita dos avisos sobre su semáforo para poder ejecutar, se ejecuta el fabricante de ruedas y fabrica dos más, dejando ahora si el semáforo *sequenciaSem* en 0, repitiendo la secuencia anterior



mencionada. Luego de pintar esta nueva moto (la segunda), se avisa por segunda vez al operario del ítem extra y al fabricante de ruedas, sin embargo este último no puede ejecutar ya que *secuenciaSem* posee valor 0. Por lo tanto ejecuta solo el operario encargado del ítem extra, agrega el ítem y setea el valor de *secuenciaSem* en 4 para que vuelva a ejecutar la secuencia.

### Compilacion y ejecucion:

Ejecute el script “scriptEjecutarTaller.sh”, el cual compila el archivo tallerMotos.c y lo ejecuta.

## 2.Santa Claus:

Para la implementación de este ejercicio, se consideraron tres entidades, cada una con su función correspondiente en el código:

- Santa
- Reno
- Elfo

Santa se implementó con un único hilo, mientras que para los renos se emplearon 9 hilos y, para los elfos, se optó por desarrollar 10 hilos. Para la sincronización entre estos hilos, se utilizaron los siguientes mecanismos:

### Mutex:

- *mutexReno* y *mutexElfo*: Estos mutex, junto con sus respectivos semáforos contadores(renos y elfos), permiten determinar cuándo el noveno reno llegó o el tercer elfo tiene problema. Los cuales son los encargados de despertar a Santa.

### Semáforos:

- *Despertar*: Utilizado por el último reno y el último elfo para llamar a santa y despertarlo. Inicializado en 0.
- *Renos*: Semáforo contador inicializado en 9. A medida que los renos llegan de vacaciones este va decrementando.
- *Elfos*: Semáforo contador inicializado en 3. Este semáforo decrece cada vez que un elfo tiene problemas en la fabricación de juguetes.
- *hay\_reno*: Semáforo binario inicializado en 0. Este semáforo es utilizado por el último reno al regresar de las vacaciones, antes de despertar a Santa. El reno realiza una señal (signal) de *hay\_reno*, permitiendo que, si llegan simultáneamente un reno y un elfo, Santa verifique primero si *hay\_reno*, si se verifica se atiende a los renos, de lo contrario se atienden a los elfos.
- *SantaReno* y *SantaElfo*: Los renos y los elfos deben esperar a que Santa los atienda para continuar con sus tareas (atar a los renos al trineo o ayudar a

los elfos). Por esta razón, se utilizan semáforos contadores inicializados en 0. Estos semáforos son empleados por Santa para indicar a los renos o elfos cuándo pueden proceder.

Modelo:

<i>Santa</i>	<i>Reno</i>	<i>Elfo</i>
<pre> wait(Despertar) if(waitry(hay_reno)==0){   <b>preparaReno</b>   for( 1 to 9){     <b>enganchar</b>     signal(SantaReno)   }   for( 1 to 9){     signal(Renos)   } }else{   for( 1 to 3){     <b>ayudaAElfo</b>     signal(SantaElfo)   }   for( 1 to 3){     signal(Elfos)   } } </pre>	<pre> <b>Volviendo de vacaciones</b> lock(mutexReno) wait(renos) if(WaiTry(renos) == -1){   signal(hay_reno)   signal(Despertar)   <b>Reno llama a Santa</b> }else{   signal(Renos) } unlock(mutexReno) <b>Reno espera que lo ate</b> wait(SantaReno) </pre>	<pre> <b>Elfo haciendo juguetes</b> if(ocurrePobrlemas){   <b>Elfo con problemas</b> <b>armando</b>   lock(mutexElfo)   wait(Elfos)   if(WaiTry(Elfos) == -1){     signal(Despertar)     <b>Elfo llama a Santa</b>   }else{     signal(Elfos)   }   unlock(mutexElfo)   <b>Elfo espera que lo ayude</b>   wait(SantaElfo) } </pre>

Funcionamiento:

1. *Renos:*

Al regresar de sus vacaciones, los renos deben ingresar de uno en uno. Para garantizar esto, se utiliza el *mutexReno*, que permite determinar quién será el último reno encargado de avisar a Santa. Si un reno no es el último, permanece esperando a que Santa lo ate. Por otro lado, cuando el último reno llega, éste realiza un signal en el semáforo *hay\_reno* y despierta a Santa, luego queda esperando junto con los demás renos a que Santa los ate.

2. *Elfos:*

Los elfos siempre están ocupados fabricando juguetes, pero si surge un problema con el armado, se utiliza el *mutexElfo* para identificar al último elfo con problemas. Si un elfo no es el último, espera a que Santa lo atienda. En cambio, si es el tercer elfo con problemas, será el encargado de despertar a Santa y se quedará esperando junto con los demás elfos hasta que Santa los ayude a todos. Si un nuevo elfo enfrenta un problema con el armado, deberá esperar hasta que se haya

atendido a los tres elfos anteriores, ya que el semáforo *Elfos* estará en 0 durante ese tiempo.

### 3. *Santa*:

Santa permanece dormido hasta que un reno o un elfo lo despierta. Al despertar, verifica si ha llegado el último reno utilizando un wait try con el semáforo *hay\_reno*. Si el último reno aún no ha llegado, Santa se encarga de ayudar a los tres elfos en espera, realizando tres señales (signal) en el semáforo *SantaElfo* y luego tres señales en el semáforo *Elfos*, permitiendo que aquellos nuevos elfos que enfrentaron un problema con el armado, puedan progresar en el código.

Si ha llegado el último reno, Santa los ata al trineo, enviando nueve señales al semáforo *SantaReno* y luego nueve señales al semáforo *Renos* para mantener la ciclicidad del proceso.

Si, tanto el último elfo como el último reno realizan un signal en el semáforo *Despertar* a la vez. La prioridad para atender a los renos se establece mediante el uso de un wait try sobre el semáforo *hay\_reno*. Santa atenderá primero a los renos y una vez que complete su tarea con ellos, pasará a atender a los elfos en el siguiente ciclo, ya que el semáforo *Despertar* tendrá un valor de 1.

### Compilacion y ejecucion:

Ejecute el script "scriptEjecutarSanta.sh", el cual compila el archivo *santaClaus.c* y lo ejecuta.

## 2.2 Problemas conceptuales

### 1 Traducción binaria de la dirección lógica (16 bits) 0011000000110011

#### a) Sistema de paginación

Ya que el tamaño de página es de 512 direcciones, tenemos que para el desplazamiento dentro de una página es de  $\log_2(512) = 9$  bits (*offset*), de ello los bits para determinar la página son  $16 - 9 = 7$  (*p*), contando con un total de  $2^7$  páginas. Por último sabemos que la dirección lógica consta de tupla  $\langle \text{númeroPágina}(p), \text{desplazamiento}(\text{offset}) \rangle$

- $p = 0011000_2 = 24_{10}$
- $\text{offset} = 000110011_2$

Para obtener la dirección física, se indexa la tabla de páginas por el número de página obtenida  $p$ , obteniendo el marco  $M = p/2 = 12_{10} = 0001100_2$ . Por último se concatena el número de marco obtenido con el offset de la memoria lógica, y se obtiene la dirección física, dicho procedimiento se puede ver en la [figura 3](#):

- Dirección física:  $0001100000110011_2$

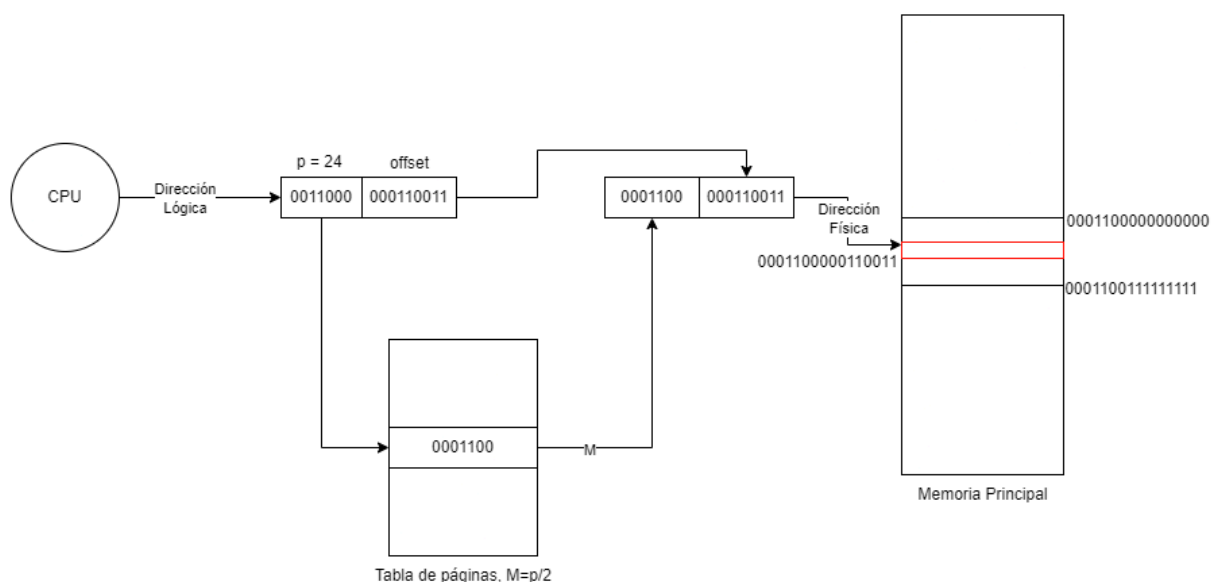


Figura 3: Diagrama de traducción de dirección de memoria con paginación

## b) Sistema de segmentación

Ya que el tamaño de segmento es de 2k es decir 2048 bytes, tenemos que para el desplazamiento dentro de un segmento es de  $\log_2(2048) = 11$  bits (*offset*), de ello los bits para determinar el segmento son  $16 - 11 = 5$  ( $s$ ), contando con un total de  $2^5$  segmentos. Por último sabemos que la dirección lógica consta de tupla  $\langle \text{númeroSegmento}(s), \text{desplazamiento}(\text{offset}) \rangle$

- $s = 00110_2 = 6_{10}$
- $\text{offset} = 00000110011_2 = 51_{10}$

Para obtener la dirección física, se indexa la tabla de segmentos por el número de segmento obtenido  $s$ , obteniendo la base:

- $\text{Base} = 20 + 4096 + s = 4122_{10} = 1000000011010_2$

Luego se compara el offset con el valor límite (tamaño del segmento), en este caso  $\text{offset} (51) < \text{límite} (2048)$ , es válido, y pasa a sumarse el offset con el valor base de la tabla

de segmentos, obteniendo así la dirección física, dicho procedimiento se puede ver en la [figura 4](#).

- Dirección física:  $\text{Base} + \text{offset} = 4122 + 51 = 4173_{10} = 1000001001101_2$

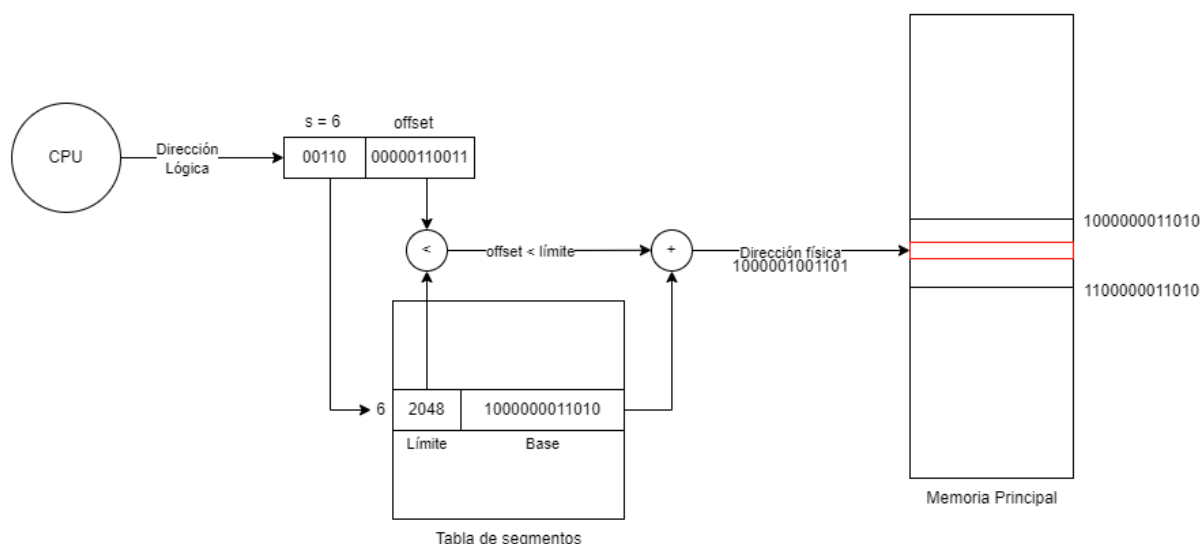


Figura 4: Diagrama de traducción de dirección de memoria con segmentación

## 2 Tabla de páginas y algoritmo de reemplazo LRU

### a) Traducción de direcciones de memoria

Ya que se posee un sistema de paginado, para poder traducir las direcciones virtuales a físicas, solo hay que indexar a la tabla de páginas mediante la página indicada en la dirección virtual y obtener el marco asociado. Una vez con el marco, este se concatena con el offset y se obtiene la dirección física.

Ya que las páginas poseen tamaño 4096 bytes, teniendo direccionamiento al byte, se obtiene que el desplazamiento sobre una página es de  $\log_2(4096) = 12$  bits (*offset*). Por lo que los restantes bits de la dirección lógica  $16 - 12 = 4$  bits son para determinar el número de página ( $p$ ), permitiendo así obtener un total de  $2^4 = 16$  páginas. De forma que a partir de la dirección lógica en hexadecimal los tres dígitos menos significativos corresponden al offset, mientras que el cuarto dígito (el más significativo) corresponde al número de página. Sabiendo esto se muestran las traducciones en hexadecimal:

- 0x621C: Se indexa la tabla de páginas con el número 6, y se obtiene el marco 8
  - Dirección física: 0x821C
- 0xF0A3: Se indexa la tabla de páginas con el número  $15_{10} = F_{16}$  y se obtiene el marco 2

- Dirección física: 0x20A3
- 0xBC1A: Se indexa la tabla de páginas con el número  $11_{10} = B_{16}$  y se obtiene el marco 4
  - Dirección física: 0x4C1A
- 0x5BAA: Se indexa la tabla de páginas con el número 5 y se obtiene el marco  $13_{10} = D_{16}$ 
  - Dirección física: 0xDBAA
- 0x0BA1: Se indexa la tabla de páginas con el número 0 y se obtiene el marco 9
  - Dirección física: 0x9BA1

Estableciendo el bit de referencia en 1, se obtuvo la siguiente tabla:

Página	Marco	Bit Referencia
0	9	<b>1</b>
1	-	0
2	10	0
3	15	0
4	6	0
5	13	<b>1</b>
6	8	<b>1</b>
7	12	0
8	7	0
9	-	0
10	5	0
11	4	<b>1</b>
12	1	0
13	0	0
14	-	0
15	2	<b>1</b>

### b) Dirección lógica que resulte en fallo de página

Para que ocurra un fallo de página, es necesario una dirección lógica cuya página no se encuentre en memoria principal, de ello al indexar en la tabla de páginas, en este caso encontraría un -. Por lo tanto una dirección la cual resulta en fallo de página es:



- 0x1BBC

### **c) Conjunto de marcos que elegirá el algoritmo de reemplazo LRU**

Ya que el algoritmo de reemplazo LRU con técnica de reemplazo local y determinando que no hay ningún frame libre, tendrá en cuenta aquellos marcos los cuales están asociados a páginas cuyos bit de referencia estén en 0. Por lo tanto el conjunto de marcos a seleccionar es:

- {0,1,5,6,7,10,12,15}