

1. Comunicación entre procesos.

1. Resolver las siguientes secuencias:

a. La secuencia permitida es:

A-A-B-C-D-E-A-B-C-D-E-E

b. La secuencia permitida es:

A-B-(C o D)-E-A-B-(C o D)-E-F

- **Solución utilizando PIPES:**

Se opto por que exista un PIPE por cada carácter que se desee mostrar, y un proceso encargado de “escuchar” el PIPE de su carácter y “escribir” en el PIPE que “escucha” el siguiente proceso. Cada proceso solo lee de un único PIPE (el del carácter que debe mostrar), y solo escribe en el PIPE del proceso siguiente. Inicialmente el único PIPE con un valor es el de A.

- **Solución utilizando COLA DE MENSAJES:**

Se creo una única cola de mensajes que todos los procesos podrían sabiendo de antemano la “clave” para acceder a dicha cola. Nuevamente, se definió un proceso para imprimir un carácter a la vez. Ademas se creo la estructura básica de un “mensaje”, que seria la estructura que se pasarían entre procesos para comunicarse, con el único atributo “tipo” que es un entero que representa que carácter se esta transmitiendo. Para hacer mas practico la asignación del “tipo”, optamos por definir como constantes los siguientes valores:

```
#define A 1
#define B 2
#define C 3
#define D 4
#define E 5
#define size sizeof(mensaje)-sizeof(long)
```

Figura 1 – Captura de pantalla de la definición de las constantes

Cada proceso se “conecta” a la cola de mensajes, esperando su mensaje para imprimir su carácter y enviar el mensaje que espera el siguiente proceso, así sucesivamente.

- **Capturas de pantalla.**

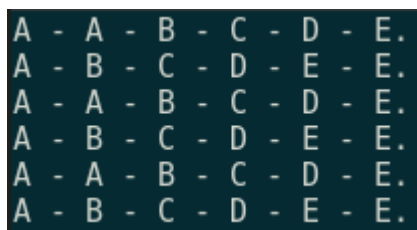


Figura 2 – Ejecución de la primera secuencia.

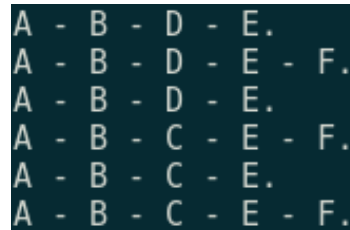


Figura 3 – Ejecución de la segunda secuencia.

2. Las abejas y el oso.

- **Solución utilizando MEMORIA COMPARTIDA:**

Definimos un struct llamado “**memoria**” al que los procesos involucrados (abejas y oso), podrían acceder. Esta estructura cuenta con los atributos “**porciones_vaciás**”, que contabiliza la cantidad de lugares disponibles en la “**fuelle**”, “**estado_oso**” que indica el estado en el que se encuentra el oso (**durmiendo** o **despierto**), y “**mutex**” para sincronizar a las abejas y asegurar que solo una de ellas este cocinando una porción de miel. Además de manera global se definieron para hacer mas practico e intuitivo, las constantes **despierto** y **durmiendo**.

```
#define durmiendo 0
#define despierto 1
#define n 10
#define size sizeof(memoria)
```

Figura 4 – Definición de las constantes del problema.

Para representar el accionar de una abeja, se crea el proceso **abeja**, por la cantidad de abejas que se desee tener, en este caso optamos por 3. Las abejas para poder cocinar una porción, deben previamente haber consumido el semáforo **mutex**. Una vez haya cocinado, la abeja habilita a que cualquier otra pueda cocinar haciendo **post(&mutex)**. Una vez **porciones_vaciás** sea igual a 0 (se lleno de porciones la fuente), la siguiente abeja que intente cocinar no podrá porque ya se lleno la fuente, entonces modifica el valor del atributo **estado_oso** a **despierto**, y habilita a que otra abeja pueda intentar cocinar. Las abejas permanecerán constantemente intentando cocinar mientras el **estado_oso** sea igual a **despierto**.

El oso, espera a que el valor de **estado_oso** cambie a **despierto**, para poder comenzar a consumir las porciones de la fuente. Una vez consumidas en su totalidad, cambia el valor de **estado_oso** a **despierto**, lo que permite a las abejas volver a cocinar, y comenzar nuevamente el ciclo.

- **Solución utilizando COLA DE MENSAJES:**

Definimos la estructura “**mensaje**”, con los atributos “**tipo**” que es hacia quien esta destinado el mensaje (oso, o abeja), y el atributo “**fuelle**”, que contabiliza la cantidad de porciones que faltan para llenar el tarro. Además de definir globalmente las constantes **oso** y **abeja** para hacer mas visible a quien va esperar por ese mensaje.

```
#define n 10 //CANTIDAD DE PORCIONES
#define m 3 //CANTIDAD DE ABEJAS
#define size sizeof(mensaje) - sizeof(long)
#define oso 1
#define abeja 2
```

Figura 5 – Definición de las constantes del problema.

Se crean dos procesos llamados, “oso” y “abejas”, donde cada uno se conecta, sabiendo la “clave” previamente, a la cola de mensajes donde estará esperando por el mensaje que la hará accionar.

El proceso “abejas” espera a que le llegue el atributo **tipo** con el valor de **abeja** (valor constante definido de manera global). Una vez el mensaje recibido es con el valor de **tipo** deseado, la abeja cocina una porción. Si, la porción preparada es la ultima que le faltaba a la fuente para llenarse (**fuelle** == 0), envía un mensaje de **tipo** igual a **oso** (valor constante definido de manera global) para que el oso pueda comenzar a consumir las porciones.

El proceso “oso” espera recibir el mensaje de **tipo** igual a **oso**. Una vez el oso recibe ese mensaje, procede a consumir las porciones hasta llegar al final. Cuando consume la ultima porción, envía el mensaje con **tipo** igual a **abeja**, para que las abejas vuelvan a cocinar y iniciar nuevamente el ciclo.

```
ABEJA: Cocinando porcion: 0 de miel.  
ABEJA: Cocinando porcion: 1 de miel.  
ABEJA: Cocinando porcion: 2 de miel.  
ABEJA: Cocinando porcion: 3 de miel.  
ABEJA: Cocinando porcion: 4 de miel.  
ABEJA: Cocinando porcion: 5 de miel.  
ABEJA: Cocinando porcion: 6 de miel.  
ABEJA: Cocinando porcion: 7 de miel.  
ABEJA: Cocinando porcion: 8 de miel.  
ABEJA: Cocinando porcion: 9 de miel.  
OSO: Comiendo porcion: 0 de miel.  
OSO: Comiendo porcion: 1 de miel.  
OSO: Comiendo porcion: 2 de miel.  
OSO: Comiendo porcion: 3 de miel.  
OSO: Comiendo porcion: 4 de miel.  
OSO: Comiendo porcion: 5 de miel.  
OSO: Comiendo porcion: 6 de miel.  
OSO: Comiendo porcion: 7 de miel.  
OSO: Comiendo porcion: 8 de miel.  
OSO: Comiendo porcion: 9 de miel.
```

Figura 6– Ejecución del programa “Osos & Abejas”.

2. Problemas

2. Memoria

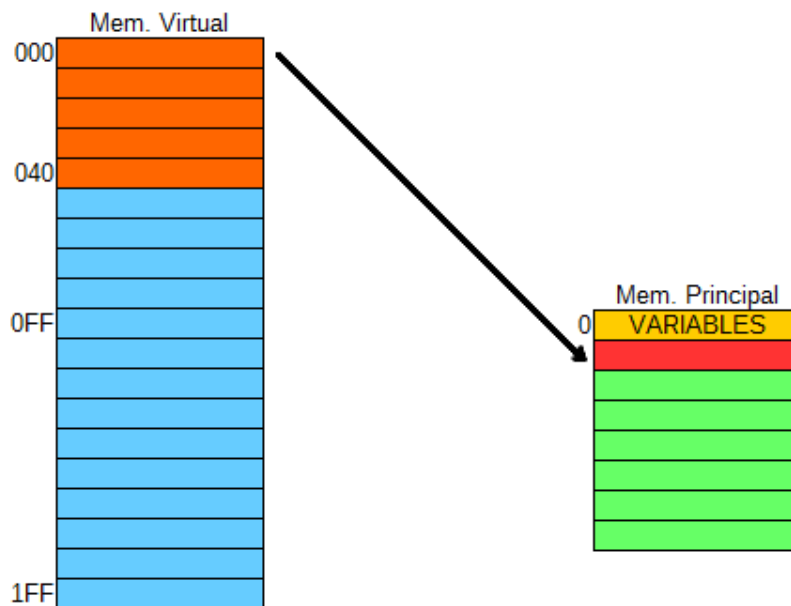


Figura 7 – Representación gráfica del problema.

Para poder acceder a un elemento a un bloque de memoria virtual se necesita el siguiente direccionamiento.



Figura 8 – Representación del direccionamiento de la memoria del problema.

Al ser 256 Bytes por pagina, y cada entero ocupar 1 Byte, puedo alocar 256 enteros en cada pagina. Por lo tanto las direcciones constan de 3 dígitos hexadecimales que diseccionan a la pagina, y 2 dígitos para direccionar a un elemento de la pagina.

Nota: cabe aclarar que el dígito mas significativo que direcciona a la memoria virtual solo puede ser 0 o 1, debido a que la memoria virtual tiene 1024 paginas.

Cada pagina en memoria virtual puede albergar hasta a dos filas enteras de datos. Por ejemplo en la pagina color **NARANJA** están alocados los elementos con indice [000 000, 001 127] (los indices de elementos son de la matriz creada i,j).

Mem. Virtual	
000 000	,,,000 127
001 000	,,,001 127
002 000	,,,002 127
003 000	,,,003 127

Figura 9 – Representación gráfica de los elementos alocados en las paginas.

Direcciones:

- Primer pagina de datos:

Virtual			Offset	
0	0	0	0	0

- Segunda pagina de datos:

Virtual			Offset	
0	0	1	0	0

- Ultima pagina de datos:

Virtual			Offset	
0	4	0	0	0

Fila 1°

- Primer elemento de la fila 1°:

Virtual			Offset	
0	0	0	0	0

- Ultimo elemento de la fila 1°:

Virtual			Offset	
0	0	0	7	F

Fila 2°

- Primer elemento de la fila 2°:

Virtual			Offset	
0	0	0	8	0

- Ultimo elemento de la fila 2°:

Virtual			Offset	
0	0	0	F	F

Fila 128°

- Primer elemento de la fila 128°:

Virtual			Offset	
0	4	0	0	0

- Ultimo elemento de la fila 128°:

Virtual			Offset	
0	4	0	F	F

Algoritmo A:

```
VAR Tabla: ARRAY [1..128] OF ARRAY [1..128] OF INTEGER;  
...  
BEGIN  
  FOR columna:= 1 TO 128 DO  
    FOR fila:= 1 TO 128 DO  
      Tabla[fila][columna]:= 0;  
    END  
  END
```

Sucesiones de accesos a paginas:

La serie de accesos a memoria para traer las paginas es:

000-001-002-...-039-040-001-002-...

Figura 10 – Sucesión de accesos a memoria (hexadecimal) en busca de paginas.

Esta sucesión ocurrirá 128 veces en total, debido a que al traer una pagina con dos filas completas en su interior, solo requerirá dos elementos de ella, debido a que este algoritmo recorre los elementos de una misma columna. Por lo tanto cada dos elementos habrá que hacer un cambio de pagina.

Faltante de paginas:

La cantidad de faltantes de pagina es: 8193 fallos. Esto es porque de cada pagina solo se pueden utilizar dos elementos, y se hace esto por cada columna.

1 Faltante (cargar las variables) +(128 filas / 2 elementos que lee por pagina) x 128 columnas =
8192 fallos

Algoritmo B:

```
VAR Tabla: ARRAY [1..128] OF ARRAY [1..128] OF INTEGER;  
...  
BEGIN  
  FOR fila:= 1 TO 128 DO  
    FOR columna:= 1 TO 128 DO  
      Tabla[fila][columna]:= 0;  
    END  
  END
```

Sucesiones de accesos a paginas:

La serie de accesos a memoria para traer las paginas es:

000-001-002-...-038-039-040

Figura 9 – Sucesión de accesos a memoria (hexadecimal) en busca de paginas.

Debido a que una vez traída la pagina no volverá a ser necesario volver a traerla, ya que en cada pagina se alocan dos filas enteras.

Faltante de paginas:

La cantidad de faltantes de pagina es: 64 fallos. Esto es porque en cada pagina (256 Bytes) entran dos filas enteras de datos, por lo tanto

$$1 \text{ Faltante (para cargar las variables)} + 128 \text{ filas} / 2 \text{ filas} \times \text{bloque} = 65 \text{ faltantes de pagina}$$

Conclusión del ejercicio:

En este problema se hace evidente que el algoritmo que hace un recorrido mas eficiente, en cuanto a accesos a memoria es el **Algoritmo B**, que tiene la menor cantidad de faltantes de paginas. Esto se debe a que recorre todos los elemento de una fila, antes de cambiar a la siguiente, aprovechando al máximo el modo en que fueron almacenados en memoria. Mientras que el **Algoritmo A**, al recorrer todos los elementos de una misma columna genera que cada 2 elementos accedidos haya un faltante de pagina.

3. Simulación.

- **Buddy:**

En esta simulación se definió un struct para abstraer el comportamiento de un proceso, con los atributos: “**id**”, que representa el identificar del proceso y “**size**” que representa su tamaño. La memoria esta simulada con un árbol binario, donde cada partición de memoria es un nodo hoja del árbol. La decisión de optar por esta estructura radica en que es la que mejor representa el comportamiento a la hora de compactar particiones, ya que las particiones “hermanas” conocen a su padre, y pueden unirse de manera mas practica.

Cada nodo del árbol mantiene información de: “**estado**”, los estados son:

- **Lleno:** cuando ambos hijos están ocupados por un proceso.
- **Ocupado:** cuando tiene un proceso, o cuando tiene uno de sus hijos ocupados.
- **Libre:** ambos hijos libres o no tiene un proceso asignado.

Ademas cada nodo del árbol, tiene los atributos: “**size**” que representa el tamaño de la partición, “**disponibles**”

- **Primer Lugar:**

Para realizar esta simulación se opto por definir una lista doblemente enlazada que representa una partición de memoria. Mantiene información del tamaño (en KB), si esta o no ocupado, identificación de la partición, identificación del proceso que esta alocado (en caso de estar libre este valor es -1), y un puntero a la partición siguiente y anterior.

Inicialmente existe un nodo (partición), libre y con 64 MB libres, a continuacion se generan aleatoriamente requerimientos para procesos de tamaños variables. A medida que estos requerimientos la lista cambia, crea modifica o destruye particiones.

Los métodos empleados son:

- **insertar(int id, int tam_pro):** es quien maneja la lógica para insertar procesos de ID=id y tamaño=tam_pro. En caso de no poder lo notifica.

- ***liberar(int id)***: busca la partición donde este alocada el proceso con ID=id, y libera el espacio de la partición, seguido de una “compactacion”, que si existen dos particiones libres contiguas las une.
- ***recorrer()***: recorre la lista para mostrar por pantalla el estado actual de cada partición.
- ***compactar()***: recorre la lista en busca de particiones libres aledañas que pueda unir.
- **Mejor Lugar:**

Para la simulación del “***mejor lugar***”, se utilizo la misma estructura que la del “***primer lugar***”, solo que modifiko la inserción de un elemento para que permita buscar de todas las particiones disponibles el proceso se aloque, si es que es posible, en la partición mas optima. El resto de la implementación es la misma.

Para todos las simulaciones esta definido un método:

- ***darRequerimientos(int n)***: se encarga de generar “n” requerimientos, incluye la creación de procesos y su eliminación.

Conclusión del ejercicio.

Como se puede observar en las simulaciones, el algoritmo de ***Buddy*** es el único que genera fragmentación interna y externa a la vez, en comparación a los algoritmos de ***Mejor Lugar*** y ***Primer Lugar***, el algoritmo de ***Buddy*** hace un uso mucho menos eficiente de la memoria, ya que termina con mas cantidad de memoria desperdiciada producto de la fragmentación interna y externa.

Entre ***Primer Lugar*** y ***Mejor Lugar***, no existe gran diferencia en cuanto a uso eficiente de memoria, los resultados para ambos fueron similares en cuanto a fragmentación externa. Una forma de mejorar este problema es realizar una desfragmentación de la memoria periódicamente, aunque tendría un costo muy alto, ya que se perdería mucho tiempo de computo realizando tareas de sistema y no de usuario.

