Sistemas operativos

Grupo OS (840) y Gpo 1 (1554)
6 to semestre. M.C. Laura Sandoval Montaño.

Evaluación.

- Trabajos (Ejercicios, cuestionarios, Indagación, etc.) 30 %
- Programas (Software) (Escritos en C, ANSI C bajo Linux) gcc 35 %
- Exámenes parciales 35%

Usuario: fredacg012 ; Password: No de cuenta

Escala

[3, 5.99] → 5
[6, 6.49] → 6
[6.5, 7.49] → 7
[7.5, 8.49] → 8
[8.5, 9.49] → 9
[9.5, ∞] → 10

3 < → NP

NO APLICA EXÁMENES FINALES

---

getchar()

Antecedentes asignaturas/temas.

- Estructura y programación de computadoras.
- Estructura de datos y algoritmos I y II.
- Linux, - Lenguaje C.

1: Introducción a los sistemas operativos

Software → Instructivos de arranque
- Programas
- Datos
  - Cuantos procesos puede ejecutar libremente.

TEMA I: Introducción a los sistemas operativos.

1.1: Concepto y propósito de los sistemas operativos.

Software operativo: Software que permite actar con un sistema de cómputo o de comunicación. (teléfono, Celular).

Sistema operativo: Administra todos los dispositivos y tareas que están conectados entre sí.

Internet de las cosas: los usuarios del internet son las cosas.

Definición formal de sistema operativo.

Un sistema operativo es software de base que administra los recursos de un sistema de cómputo-comunicación con el objetivo de proporcionar las funcionalidades que dicho sistema provee al usuario.

Generalmente cuenta con una interfaz hacia el usuario y aplicaciones.

Software de base: Cercano a los dispositivos físicos.

compiladores, sistema operativo.

Lógica alambrada

firmware: Programa que es físicamente un chip (inmerso en el dispositivo)

BIOS: firmware que se encarga de revisar que dispositivos están conectados.

* Manejador de base de datos Es SO → Falso

Recursos: Aquellos elementos o aplicaciones para que el SO pueda trabajar.

- Recursos
  - Tipo hardware
  - Tipo software

Recursos

Cómputo		Comunicaciones	
Hardware	Software	Hardware	Software
• Memoria.	• Programas del usuario.	• Modem.	• Mensajes.
• Teclado	• Bibliotecas.	• Altavoz/Bocina	• Llamadas
• Dispositivos de entrada y salida.	• Procesos.	• Drivers de hardware	• Drivers de software.
• Procesador			
• Disco			

Los más importantes.

Proceso: Un programa en ejecución.

```
int
#include <stdio.h>
int main()
 {
    fork();
    print("Hola amigos");
 }
```

driver: Programas de bajo nivel que generalmente se encargan de manipular el hardware, son invocados por el procesador cuando son de muy bajo nivel y con el sistema operativo.

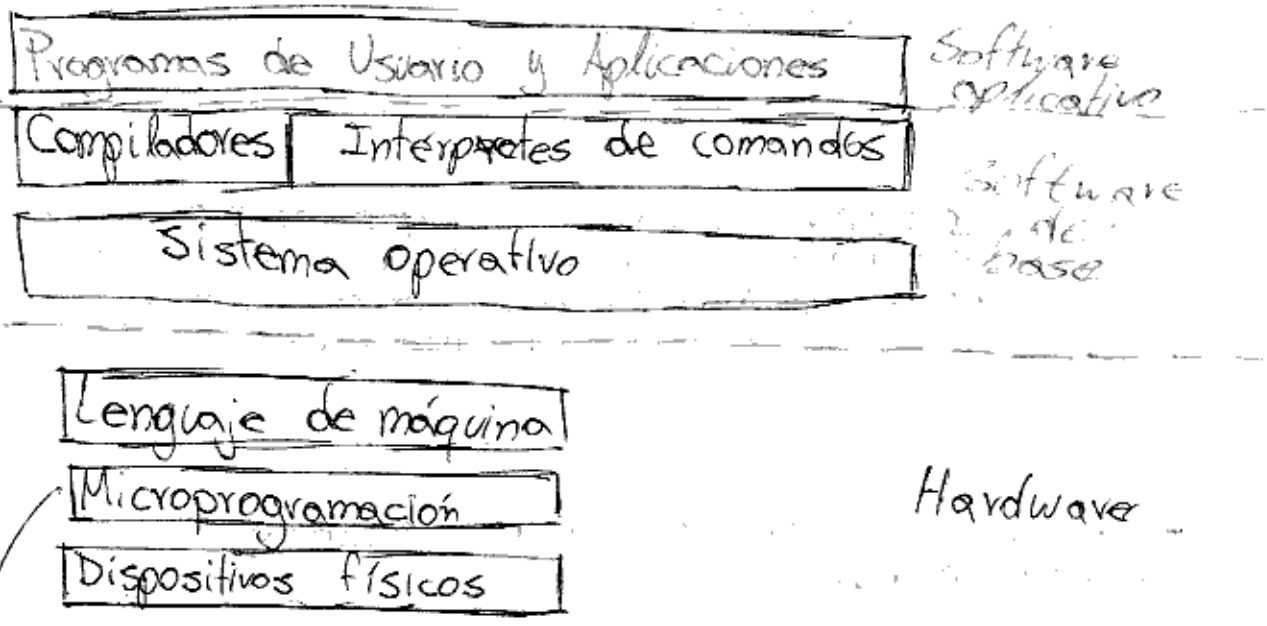
Funcionalidades de un sistema de cómputo-comunicación.

- Permite al usuario almacenar, recuperar y procesar datos.
- Administra las tareas del procesador para que ejecute programas del usuario y aplicaciones de manera eficaz.
- Utilizar el hardware del sistema de forma eficiente.
- Proporcionar una interfaz de acceso a dispositivos y datos remotos, conectados a través de líneas de comunicación.
- Generalmente cuenta una interfaz hacia el usuario y aplicaciones. → Puede ser a través de una interfaz de comando. También por interfaz gráfica.

Unix: cd .. /Alfredo/sisOperativos

MsDos: cd .. \alfredo \sistOperativos

- Las primeras computadoras no utilizaban sistemas operativos.
- Los primeros sistemas operativos solo podían hacer un proceso a la vez.



Instrucciones en lenguaje máquina que le dice al procesador que hacer, inmersa en algo físico (no se puede borrar) un ejemplo es el bios.

S.O: Sistema Operativos
O.S: Operate System

1.2 Evolución de los Sistemas Operativos.

- 1940: No existían sistemas operativos, entonces directamente el usuario a través en lenguaje máquina le indicaba a la computadora lo que debía hacer, en el lenguaje binario, generalmente para calculos, no para almacenar archivos.

- 1950: Surge el primer sistema operativo para una computadora IBM 1704 (en el año 1956). Ya no se depende que el usuario sea un especialista para utilizar el hardware.

  - Las computadoras eran electromécanicas, usaban bulbos.

	Hardware y software	Manejo de procesos
1950:	• Computadoras electromecánicas (bulbos) • S.O. IBM 704	• Una tarea a la vez (unitarea). • procesamiento por lotes (batch). .bat
1960	• Ya no son computadoras electromecánicas sino electrónicas porque utilizan transistores • S.O. IBM 360 (OS/360) • S.O. Multics → Por multitareas. • En 1969 surge UNIX	• Procesamiento por lotes. • Multiprogramación o multitareas. • Multiusuarios. • Usuarios interactivos. • Memoria virtual. • Sistemas de tiempo real.

Usuario interactivo: Interacción en línea.
Interacción en tiempo real: Respuesta inmediata.

1970	Hardware y Software	Manejo de procesos.
1970	• Auge de los circuitos integrados reduce el espacio de las computadoras y surgen las minicomputadoras. • Se inicia la creación de las microcomputadoras así como los microprocesadores (Todo se reduce en tamaño pero no en capacidad. • Surge memoria de semiconductores. • Almacenamiento masivo • Redes de área local • Se reescribe UNIX en lenguaje C.	• Múltiples modos de operación en un solo sistema. (Multiusuario y multiprogramación) ↓ Varios usuarios atendidos en el sistema

1.- Verdadero.
2.- Falso.
3.- Verdadero
4.- Verdadero
5.- Verdadero

Linux funciona en cualquier procesador.

	Hardware y software	Manejo de procesos
1980	· El auge de las computadoras personales (PC) (Personal computer). · Surgen estaciones de trabajo. y servidores · microprocesadores (Gran avance) · Redes de computadoras metropolitanas. · Surge el internet. · Ms Dos · Windows. · MacOS	- Además de las anteriores. - Surge el cómputo distribuido. - Surge el modelo Cliente-Servidor.
1990	· Surge la arquitectura masivamente paralela. · Utilizaban las supercomputadoras. · La www crece rápidamente. · Surge el sistema operativo Linux (en 1991.	~~Todos~~ - Se difunde más la computación distribuida. - Surgen los protocolos de sistemas abiertos.
2000	· Dispositivos inteligentes. · La nube (AWS)	→ Web
2010	· Internet de las cosas. · Auge de la inteligencia artificial (forma más global) para aplicaciones. · Open Source (Github y Redhat)	
2020	· Redes 5G (Incremento Iot)	

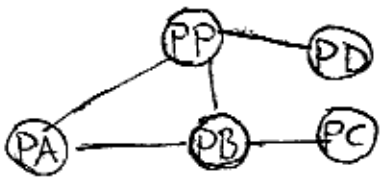
Actual

1.3.- Estructura de los sistemas operativos

*Estructura Monolítica

Un solo programa realiza las funciones básicas del sistema operativo.

- Está formado por funciones y procedimientos.

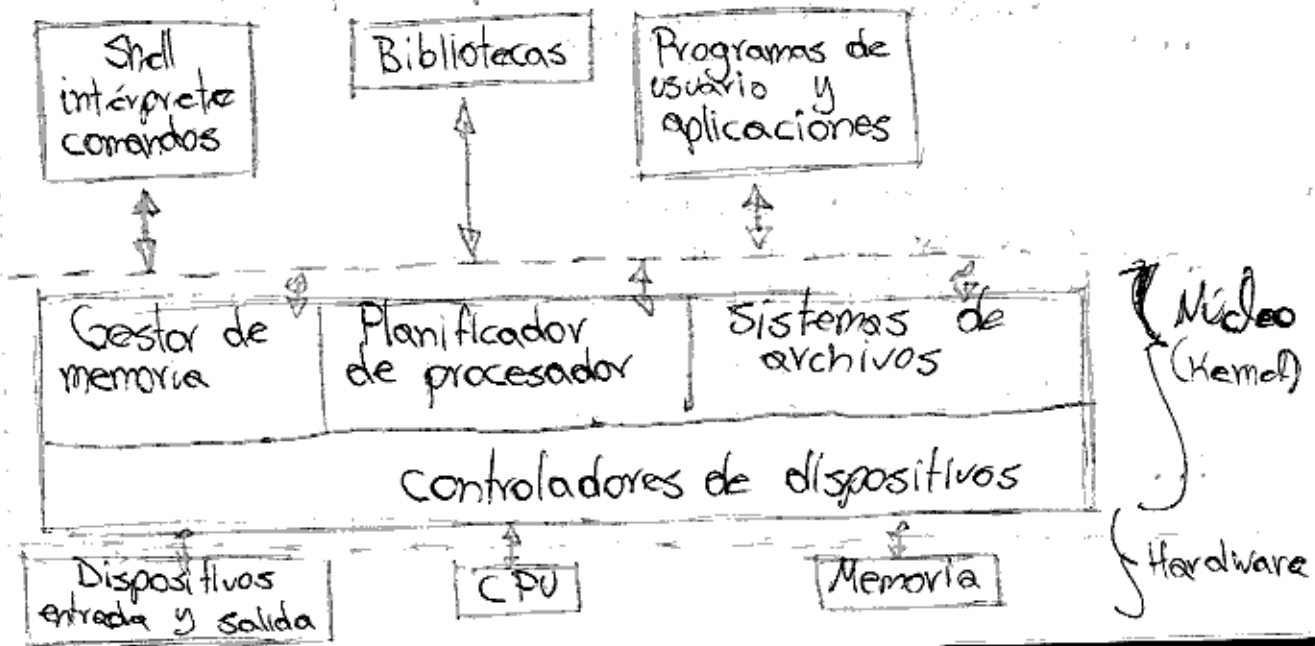


*Es un caos organizado, por ser un solo programa. Todo el código esta en memoria principal.

Ejemplo:

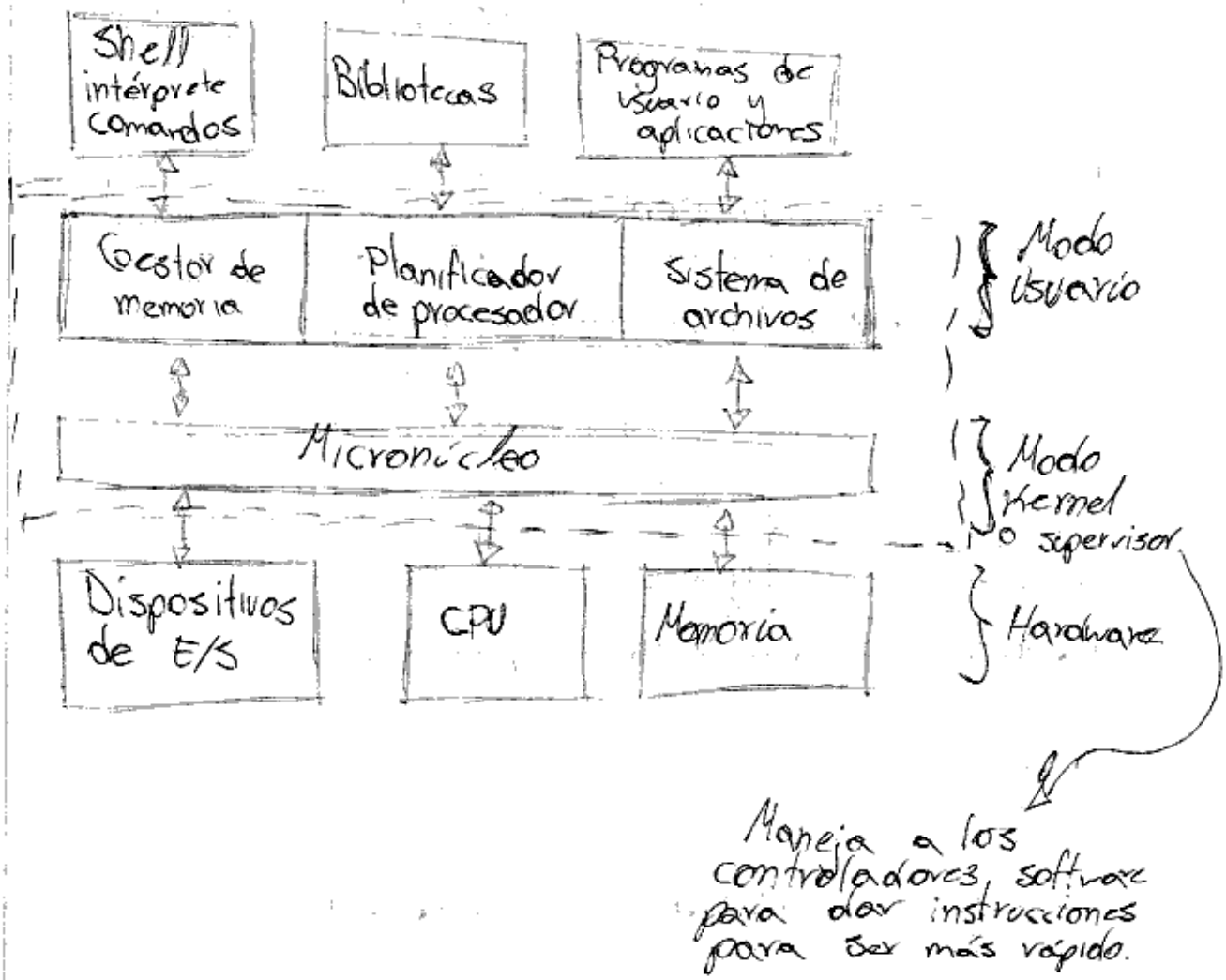
- MSDOS (Sistema operativo en disco).
- MsDOS → System operating
  ↓
  Microsoft Disk
- Linux | Unix

*Estructura por capas



Terminales: C - Shell
K - Shell
B - Shell

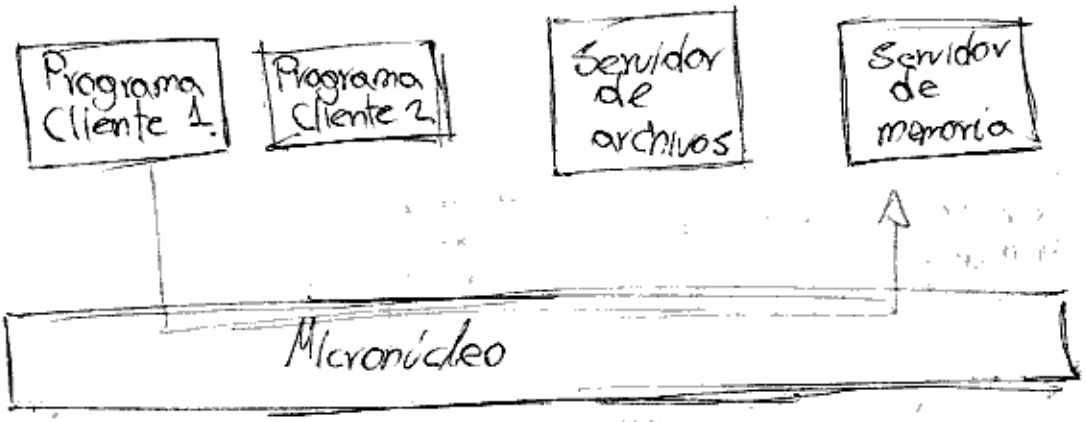
* Estructura de Micronúcleo.



// En Linux se pueden aumentar o disminuir el número de procesos del sistema operativo.

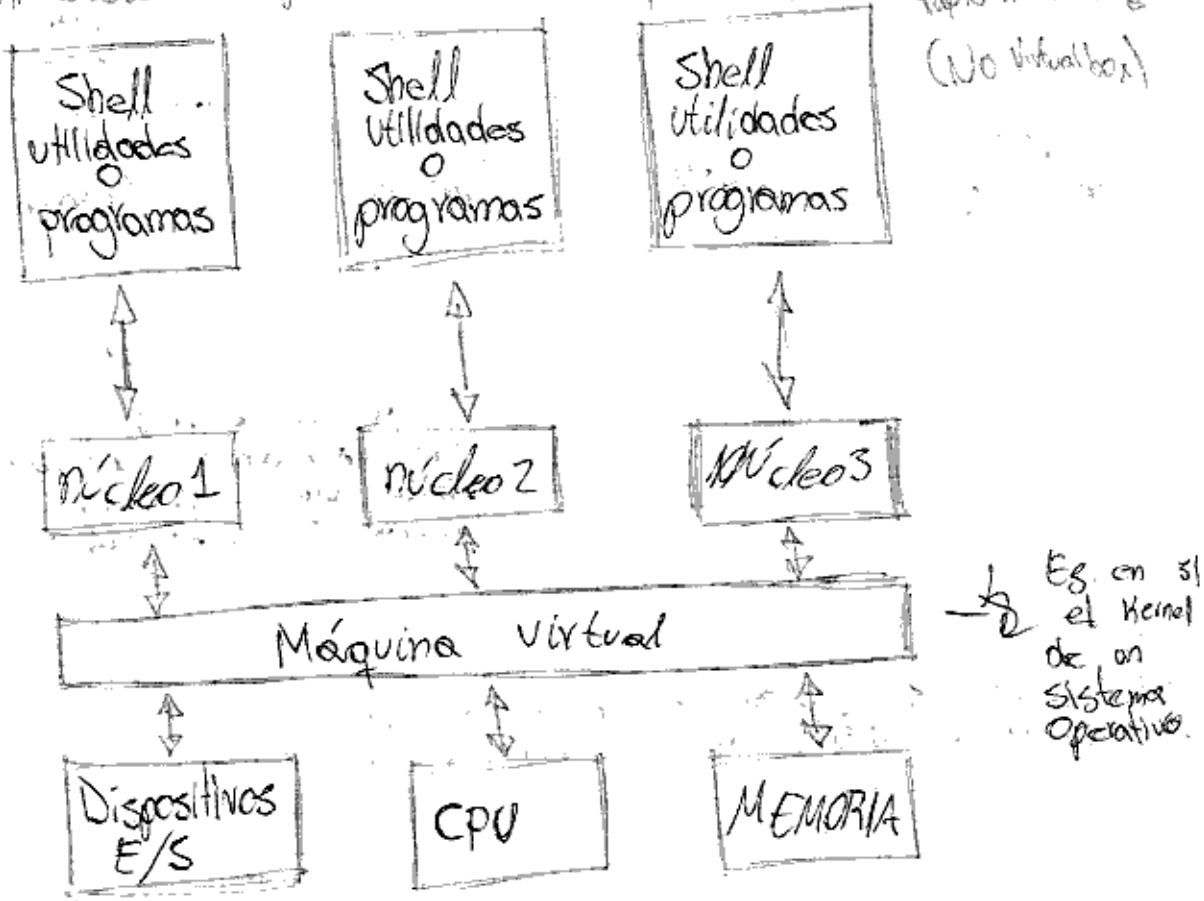
* Variante estructura micronúcleo.

// Solo se cargan por indicaciones del micronúcleo.



Modelo cliente-servidor.

* Sistemas operativos como máquinas virtuales de tipo hardware (No VirtualBox)



administrador = Gestor

Alfredo

* Máquina virtual de software

Para abstraer o virtualizar un ~~software y hardware~~ sistema total tanto software como hardware, cuyo objetivo es integrar varios sistemas operativos dando la sensación de ser varias máquinas diferentes.

1.4 Componentes de un sistema operativo actual

- El administrador de procesos

Provee recursos al proceso o procesos para que puedan realizar su tarea. (memoria, estados, cuando ingresan, cuando se suspenden)

- Planificador de procesos ~~y procesador~~

Se encarga de seleccionar que proceso va con el procesador para su ejecución y durante cuanto tiempo.

- Gestor de memoria.

Administra tanto los espacios libres como ocupados de la memoria principal.

- Sistema de archivos.

Se encargan de proporcionar una vista uniforme del almacenamiento de archivos (memoria secundaria).

- Manejadores de dispositivos

Oculta las características específicas de cada dispositivo y ofrece servicios comunes a todos.

// procesos se en memoria principal.

•Las computadoras personales las definio IBM.

•Llamadas al sistema.

Son rutinas que ordenan al sistema operativo que desempeñe un trabajo referente a manipulaciones detalladas del hardware y software.

•Sistemas de comunicaciones.

A través de estos sistemas de comunicaciones se encargan de controlar el envio y recepción de datos a través de las interfaces de red.

•Sistemas de protección.

Es el mecanismo que controla el acceso de los programas o los usuarios a los recursos del sistema o especificar los controles de seguridad a realizar.

1.5.- Consideraciones de diseño de un sistema operativo.

En el diseño se ve como va a ser la convivencia de los procesos que va a estar administrando el SO.

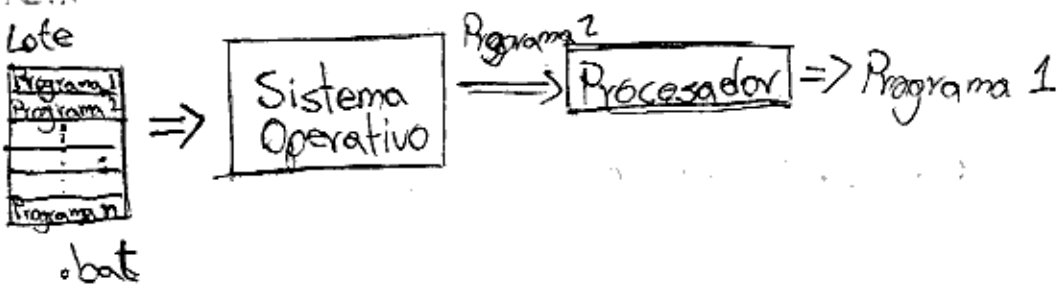
El más simple.

> Sistema operativo de tarea única y único usuario.

En la memoria hay un solo SO y un solo proceso.

Ejemplos: CP/M y primeras versiones de MsDOS

> Sistemas operativos por procesamiento por lotes o batch.



. multiprocesamiento ≠ multitarea

* El que ejecuta los proceso es el procesados
* El so decide que proceso se le asigna al procesador.

> Sistema operativo multitarea o multiprogramación

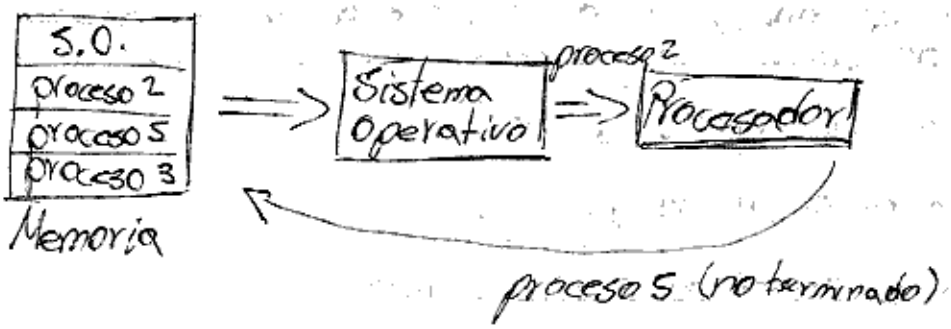
Se manejan varias tareas o procesos de manera simultánea o concurrente.

El sistema la memoria tiene los procesos, los procesos compiten por los recursos del sistema, incluso la memoria, pero el más importante es el tiempo del procesador.

- Con un solo procesador solo 1 proceso puede estar en ejecución.
- Con dos o más unidades de procesamiento puede haber dos o más procesos ejecutandos (1 en cada unidad de procesamiento)
  - Esto es Multiprocesamiento.

En todo sistema de multitarea ocurre lo que es el cambio de contexto

Cambio de contexto



---

* " " debe ser multitarea → verdadero.
* En un sistema de multiprocesamiento debe ser multiusuario → Falso

• Unidad de procesamiento: Puede ~~Los~~ núcleos del procesador.

- Registros.

- Program Counter: Apuntador de programa, continua después de donde se quedó.

Cambio de contexto: Continuar un proceso donde se quedó.

~~Continuar un proceso que ~~ no fue terminado.~~

Continuar un proceso que no fue terminado, justamente donde se quedó.

El cambio de contexto

Resguardar el estado del proceso saliente y reestablecer el estado del proceso entrante.

Cuando se realiza
Cambio de contexto.

- ~~Cuando se~~

1.- Del proceso saliente se guardan todos sus valores asociados, como los de los registros y direcciones de memoria implicados, en una estructura de datos llamada: bloque de control de procesos. (process control block (PCB)).

Un bloque para un proceso.

2.- Cambia el estado del proceso.

3.- Cambia el estado del proceso entrante.

4: Se cargan los valores de los registros y direcciones de memoria asociados al proceso entrante, localizados en su estructura PCB.

Se cargan los valores de los registros y direcciones de memoria asociados a los procesos en su estructura PCB

• Un usuario único puede mandar a ejecutar varias tareas.

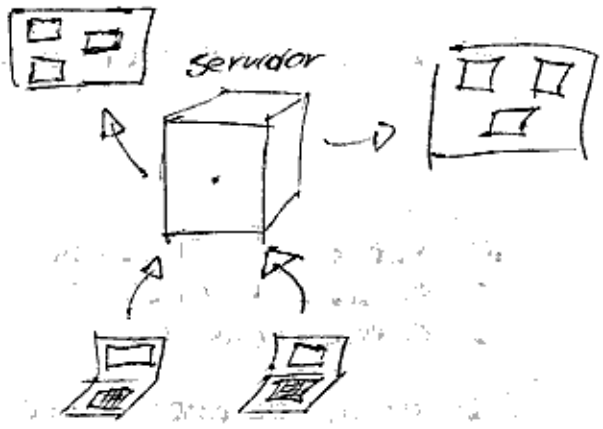
SO: Software que administra los recursos para que el sistema de cómputo sea funcional.

Procesos interactivos: Respuesta inmediata del usuario.

Multitarea
- Usuario único
- varios usuarios / aplicaciones interactivas.

Usuario único

~~Puede darse con~~



> Sistemas operativos distribuidos.

Trabajan sobre sistemas distribuidos.

~~Un sistema operativo~~

Sistema distribuido: Es una conexión de procesadores conectados en red. Comparte recursos, el más importante que comparte es el tiempo de procesador.

↓

Su finalidad es compartir recursos.

> Desempeña las mismas funciones que un sistema distribuidos normal pero sus recursos van a estar distribuidos en una red.

1.- Introducción a los sistemas operativos

> Sistemas operativos en tiempo real

- Por ser en tiempo real la respuesta debe ser inmediata.
- No manejan niveles altos de programación.

(Unitarea o tareas no tan exigentes).

- Sistemas embebidos.
- No se usan para manejo de mucha información en archivos.

Tarea 1:

1.- Elabora 5 enunciados de verdadero y falso del tema 1.1 referido al concepto y propósito de los sistemas operativos.

2.- Para las décadas 2000 y 2010 a la fecha incluye una tecnología de la época y un tipo de manejamiento de procesos.

3.- De la estructura de los sistemas operativos, indagar cual es la estructura del kernel en unix/linux.

(Decir como se comunican los componentes)

4.- Elaborar un cuadro sinóptico de los diferentes diseños de un sistema operativos (vistos en 1.5).

Tema 2. Administración de procesos

El objetivo es que el alumno clasifique los procesos mediantes los diferentes tipos de comunicación concurrentes.

- Interviene mucho la sincronización.
- Una llamada al sistema es una rutina.

2.1. Procesos. Concepto y estructura.

El proceso es el recurso software más importante que administra el sistema operativo.

- Generalmente se dice que un proceso es un programa en ejecución pero no todo proceso es un programa en ejecución.
- Otra definición: Un proceso es una actividad asíncrona.
- " " : Es el espíritu animado de un procedimiento.
- " " : Es el centro de control de un procedimiento en ejecución.
- " " : Es la entidad a la que se le asignan los procesadores.

¿Qué es un programa y que es un proceso?

- Es estático porque ~~estático~~ es un archivo, esta en el almacenamiento secundario.
- Cuando se ejecuta se vuelve dinámico porque pasa a memoria principal.

- Es dinámico, reside en memoria principal.

Tema 2: Administración de procesos

Creación de procesos

- De manera general hay cuatro formas.

1.- Como parte del arranque del sistema operativo.

// En linux el primer proceso que se crea es *init()*

2.- Dando la orde de ejecución de un programa o aplicación. (por línea de comando o interfaz gráfica).

3.- A través de un programa en ejecución utilizando llamadas al sistema.

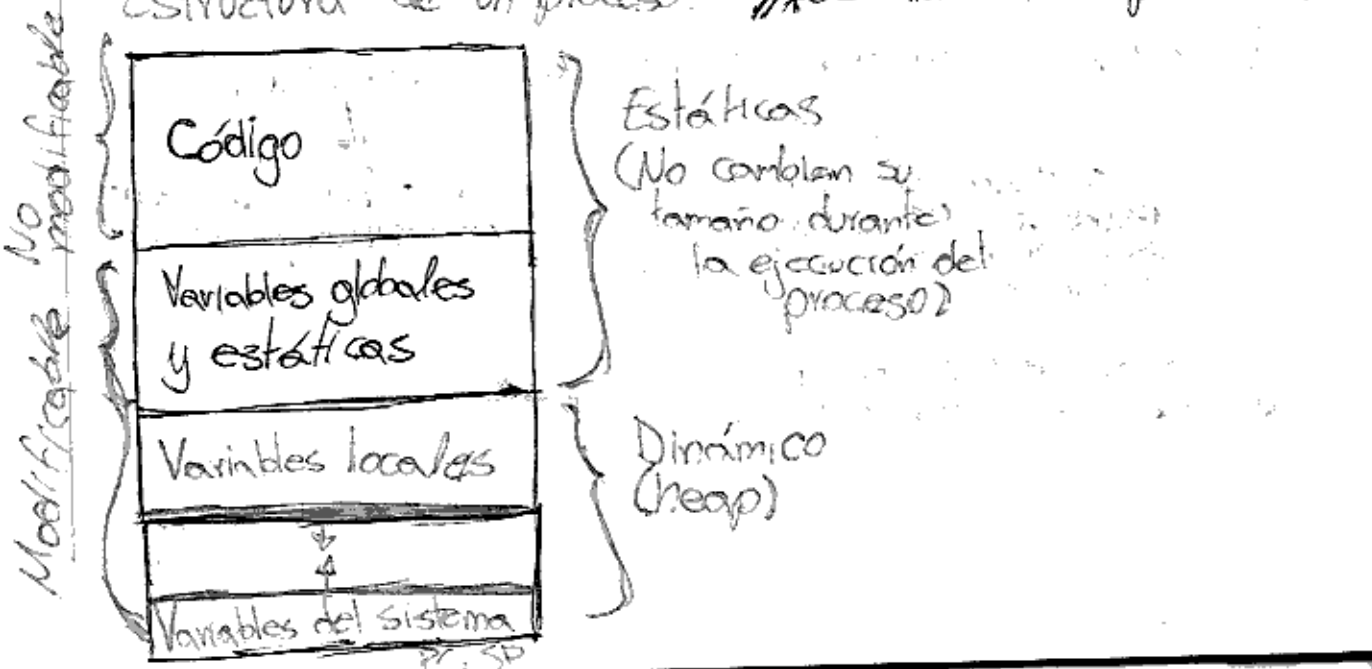
// Para crear un proceso se utiliza la llamada al sistema fork() de linux/unix *//

4.- Como parte del procedimiento por lotes en un sistema que lo realice de manera automática.

Cuando se crea un proceso se le asignan recursos del sistema, el tiempo de procesador (más importante), memoria principal, dispositivos de entrada y salida, entre otros.

Estructura de un proceso

// *Se divide en tres partes *//



BLOCK CONTROL PROCESS

Tema 2: Administración de procesos.

Bloque de control de procesos: (BCP o PCB).

// * Se utiliza en el cambio de contexto *//

// * Es un bloque por cada proceso *//

// * La información de los registros, de las variables del sistema estan en el PCB *//

Generalmente el BCP contiene:

1: El estado actual del proceso.

2: Un identificador único (PID) // * identificador de proceso *// a init() se le asigna el PID 1.

3: Un apuntador hacia el proceso padre. (El que lo creó).

Todo proceso es creado por un proceso excepto init(). Todos

4: Apuntadores a los procesos hijo

5: Prioridad del proceso. Generalmente un SO tiene procesos pri

6: Apuntadores hacia zonas de memoria del proceso.

7: Apuntadores a los recursos asignados al proceso

8: Una área de salvaguarda de los registros.

---

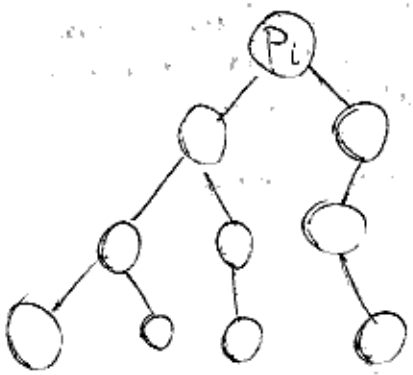
* Variables del sistema: Variables del sistema operativo para administrar el proceso. Ejemplo: program counter, stack counter

Tema 2- Administración de procesos.

9- El procesador en el que se está ejecutando.

La BCP es la entidad que define un proceso al sistema operativo. Es muy importante su manejo y de hecho muchos sistemas de cómputo tienen un registro de hardware que siempre apuntan hacia el PCB del proceso que se está ejecutando. Esto para que el acceso sea inmediato

Estructura jerrárquica de procesos.



Procesos en linux.

Los procesos en linux tienen la estructura antes vista (la que esta dividida en tres). El PCB se llama tabla de procesos, donde específicamente linux maneja los siguientes campos

1- Estado

2- Apuntadores a memoria ocupada.

3- Campo de señales.

4- Temporizadores para calculo de prioridad dinámica.

5- Parámetros de planificación.

6- Identificador de usuario. (UID)

Tema 2: Administración de procesos.

7: Identificador de proceso (PID).

8: Descriptores de eventos.

Todos los proceso excepto init(), son creados por la llamada al sistema fork().

```
#include <sys/types.h>
#include <unistd.h>     //Para que reconozca fork()
pid_t fork();           //No tiene argumentos y nos regresa un identificador de tareas.
```

```
#include <sys/types.h>
#include <unistd.h>     // Unix standar
#include <stdio.h>

int main()
{
    fork();/* Se crea proceso hijo */
    printf("hola amigos \n");
    /* El printf lo escribe el padre
    return 0; el hijo */
}
```

- Solo lo que está abajo del fork();
- No se sabe cual se ejecutará primero.

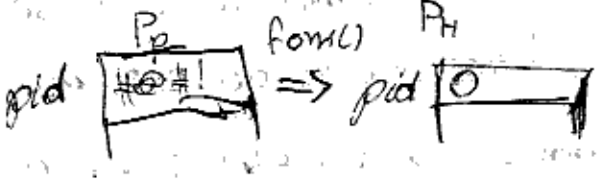
Tanto el proceso padre como el proceso hijo ejecutan el mismo código ubicado el la memoria original (No se duplica el código)

*Además del código comparten los archivos, hola amigo saldrá dos veces en el monito.

Tema 2: Administración de procesos

Tanto variables globales y locales se copian.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
  int pid;
  pid = fork();
// En este punto fork(), ya creó el proceso hijo. //
  if (pid != 0)
   {
    printf("Soy el padre");
   }
   else
   {
    printf("Soy el hijo");
   }
   return 0;
} //main
```

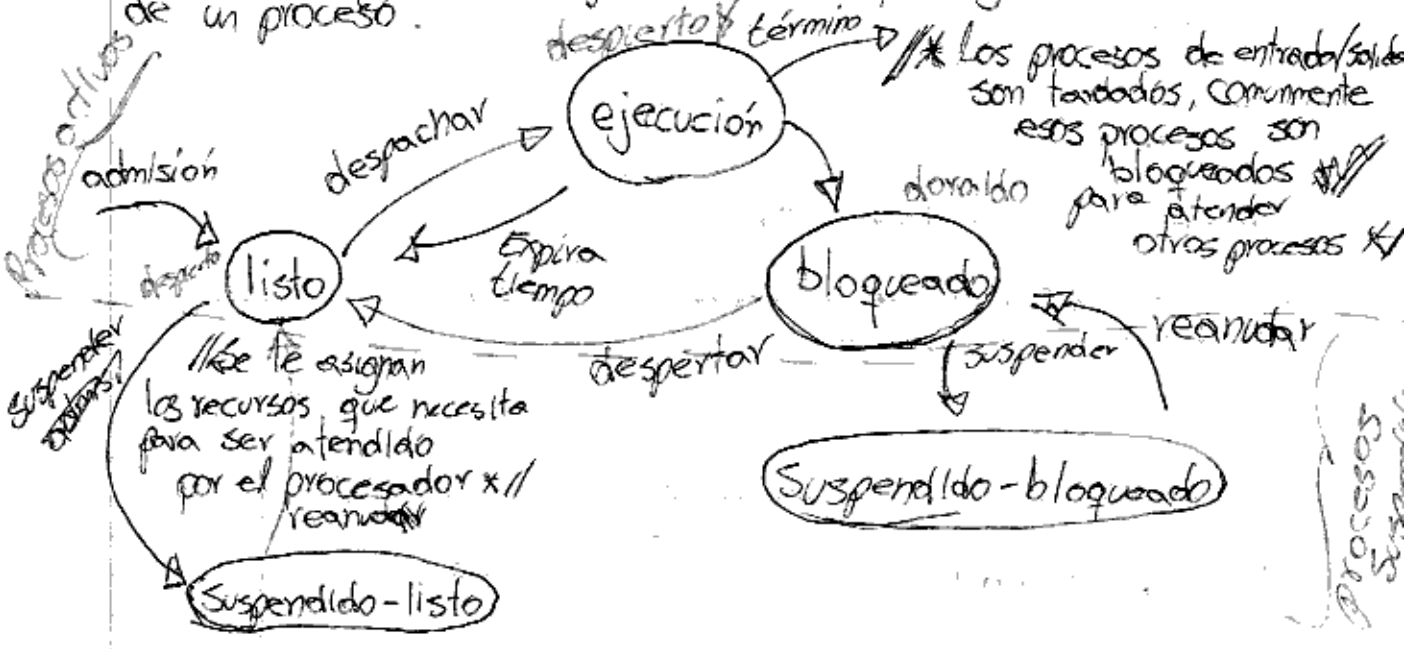


kill -9 : Para matar a un proceso de a de veras.

2. Administración de procesos

Estados de un proceso

De manera general se pueden decir que hay tres estados de un proceso.



- Un proceso suspendido no puede proceguir sino hasta que lo reanuda otro proceso
- Razones para suspender a un proceso:
  - * Se vea que el sistema está funcionando mal y pueda fallar.
  - * Un usuario que está al pendiente de los resultados que está dando su proceso, lo puede suspender para ver sus resultados parciales y posteriormente puede reanudarlos.

$ kill -9 1037

$ ps -x : Ver relación de procesos (Cuantos procesos se han lanzado) Todos los que han existido desde que encendiste la PC.

* Un proceso activo puede estar activo : Verdadero

2.- Administración de procesos

*Si se ve que hay mucha carga de trabajo y se requiere que un proceso importante termine, se regresa a suspender el proceso para que los importantes terminen su ejecución.

Tarea:

No más de una cuartilla.

Los comandos en linux para suspender y reanudar procesos.

PID	Estado actual	Transición	Estado final
215	bloqueado	~~despertar~~ ~~despertar~~ despachar	listo
217	listo	Suspender	suspendido-listo
215	listo	despachar	ejecución
218	suspendido-bloqueado	reanudar	bloqueado
213	ejecución	bloquear	bloqueado
220	Ejecución	expira tiempo	listo

ps -x : Estado zombie

2: Administración de procesos

- Razones para que los procesos terminen.

* Cuando un proceso termina, generalmente pasado un tiempo libera sus recursos.
* Salida normal (Cuando termina el proceso por su algoritmo).
* Cuando hay un error crítico del mismo proceso.

//*Violación de segmento (Acceder a localidades de memoria que están afuera del rango).

* Puedes ser por una condición de excepción.
* Por recibir señal de otro proceso (kill).

Una interrupción es un evento asíncrono, es decir, puede ocurrir en cualquier momento; no está sincronizado con el reloj del sistema ni con el ciclo de ejecución de instrucciones del procesador.

Cuando el SO detecta una interrupción que puede manipular, realiza lo siguiente:

1.* El sistema operativo toma el control ante la recepción de una señal de interrupción.
2. El SO guarda el estado del proceso interrumpido.
3. El estado se aloja en el PCB
3. El SO analiza la interrupción y transfiere el control a la rutina apropiada para atenderla.
4. La rutina del manejador de interrupciones procesa la interrupción.
5. Se restablece el estado del proceso interrumpido. (del que estaba en ejecución

// Cuando hay una llamada al sistema es una interrupción

2=Administración de procesos getpid Pn: Proceso nieto

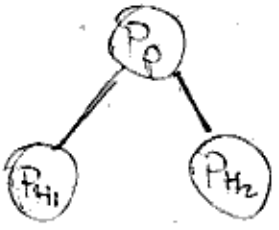
6: Continua la ejecución del proceso interrumpido.

Llamadas al sistema de manejo de procesos

fork() : • Crea un proceso con el mismo código del proceso que lo creó. (PID del hijo)

• Regresa un valor al proceso padre y otro al proceso hijo (0)

• Si no puede crear un proceso regresa -1



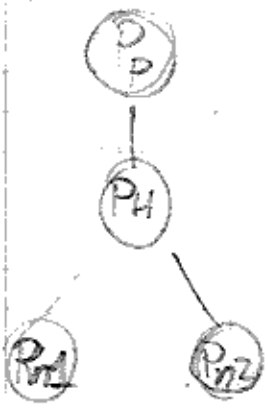
Este caso:

El hijo no sabe cual es su pid

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  int pidH1, pidH2;
  pidH1 = fork();
  if(pidH1)
   { pidH2= fork();
     if(pidH2)
       {
         printf("Soy el padre y ya cree dos hijos")
         printf("Sus PID son: %d y %d", pidH1, pidH2);
       }
0: falso
otro número: verdadero
```

```
else
{
 printf("Soy el hijo 2 y
   mi pid es: %d ,
                getpid();

}
else
{
 printf("Soy el hijo 1 y mi
  pid es: %d, getpid());

  return 0;
}// fin del main
```

· Correa González Alfredo
· Martínez Herrera Esteban




```
#include <sys/types.h>
#include <unista.h>
#include <stdio.h>
int main()
{
 int pidH, pidn1, pidn2;
 pidH = fork();
 pidn1 = fork();
 if(pidH)
  {printf("Soy el padre y ya cree un hijo y un nieto");
  printf("Sus pid son: %d y %d", pidH y pidn1);
 }else
  if(pidn2)
  {pidn2 = fork();
  if(pidn2)
   printf("Soy el hijo, mi pid es: %d", getpid());
  }else
   {printf("Soy el nieto 2, mi pid es: %d", getpid());
    }
```

```
else
 {
 printf("Soy el nieto 1: %d", getpid());
 }
 return 0;
}//Cierra el main()
```

//

2.- Administración de procesos

Características del proceso padre versus proceso hijo.

Al momento de crear un proceso con fork() al nuevo proceso se le asigna un identificador (PID), el cual debe de ser único.

- Qué comparten:
  - *Código. (la misma localidad de memoria).
  - *Archivos abiertos por el padre antes de crear al hijo.

- Qué no comparten:
  - *variables globales.
  - *variables locales.
  - *Variables del sistema.

  } El hijo ocupa otra área de memoria para estas variables.

- Si el padre termina, los hijos son adoptados por otro proceso de una jerarquía más grande.

Ej

2- Administración de procesos

Ejercicio

Elabora un programa en C cuya jerarquía de procesos sea de un padre y un hijo. El proceso padre definirá un arreglo de cinco enteros, el hijo escribirá el menor del arreglo

```
#
#

int main()
{
  int arr[5] = {25, 17, 8, 40, 13};
  int pidH;

  pidH = fork();
  if (pidH)
  {
    printf("Soy el padre \n");
    arr[3] = 1;  // Solo se modifica el del padre.
  }

  else
  {
    int i, min = arr[0];
    for (i = 1; i < 5; i++)
    {
      if (min > arr[i])
        min = arr[i];
    } // fin for
    printf("El menor es: %d \n", min);
  }//fin else

  return 0;
}// fin main
```

P_P	P_H
arr[0] = 25	arr[0] = 25
arr[1] = 17	arr[1] = 17
arr[2] = 8	arr[2] = 8
arr[3] = 40 1	arr[3] = 40
arr[4] = 13	arr[4] = 13
pidH = ~~0~~ 3061	pidH = 0
	i = ~~0~~ 1 2 3 4 5
	min = 25, 17 8

2.- Administración de procesos

Llamada wait.

wait() : A diferencia de fork() y getpid() este si necesita ~~regresa~~ argumentos, necesita la biblioteca

```
#include <sys/wait.h>
pid_t wait(int *<stat_loc>);
```

Espera a que termine a su hijo para terminar, si tiene varios hijos espera a que termine el primer hijo.

~~wait(null);~~ wait(NULL);

Llamada exit.

exit(): Requiere la siguiente biblioteca:

```
#include <stdlib.h>
void exit(int status);
                0-255
```

exit tiene dos funciones, cuando un proceso manda sentido del hijo al padre, solo una contiene de 0 a 255.

es el equivalente del return de una función, pero exit es para un proceso

Llamada pipe()

Crea un canal de comunicación entre proceso padre y proceso hijo.

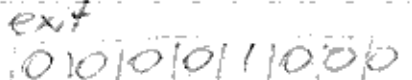
Se puede decidir que uno escriba y otro lea en cualquier orden

>> << : Corrimientos de bits

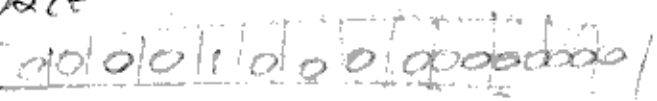
2: Administración de procesos

// Para que el padre lo imprima

exit



wait



```
int main()
{
    int arr[5] = {};
    int pid1, valor;

    pid1 = fork();
    if(pid1)
    {
      printf("Soy el padre");
      wait(&valor);
      Printf("El minimo es: %d", valor>>8);
    }
    else
    {
      int i, min = arr[0]; <- sleep(1);
      for(i=1; i<5; i++)
      {
        if(min>arr[i])
          min=arr[i];
      }// fin for
      exit(min);
    } // fin else

  return 0;
} // fin main
```

x=2

x = x >> 1 // x << 1 = 2;

// x será 1

equivalente

x=4

x = x << 2;

x será 16

Tarea: Elaborar un programa en C que como estructura de procesos sea el proceso padre y dos procesos hijo. Los procesos hijo deben coexistir // Luego luego el padre debe crear a los dos hijos. El proceso padre creará un arreglo de 10 elementos enteros, cuyos valores vayan del 0 al 255, el Ph1 calculará el mínimo, se lo enviará al padre para que lo escriba, Ph2 calculará el mayor y se lo enviará al padre para que lo escriba. // Deben de aparecer 2 : a lo más

// En un solo printf() el mayor y el menor.

2. Administración de procesos.

Clases de interrupciones

Existen seis clases de interrupciones.

1. Interrupciones del llamados al sistema o supervisor.

2. Interrupciones de entrada y salida

Cuando se va a usar un dispositivo de entrada y salida

Se generan al tratar de hacer una operación con dispositivos de entrada y salida. Pasa a bloqueado.

3. Interrupciones externas de entrada y salida.

Hay una incidencia, el movimiento del ratón.

4. Interrupciones de verificación de la máquina.

Se ocasiona por el mal funcionamiento del hardware, por ejemplo, cuando falla una sección del disco. ó que se llegue a dañar un módulo de memoria principal.

5. Interrupciones de verificación del programa.

Ocurre por mucha frecuencia entre los programadores, ejemplo: dividir entre cero, leer mal una cadena, ejecutar un código con algo inválido, localidad de memoria incorrecta.

6. Interrupciones críticas.

Por cuestiones externas

Se cae el café, se cae el equipo etc.

2. Administración de procesos

Otras llamadas al sistema

Vistas:

- fork()
- getpid()
- wait()
- exit()

otros

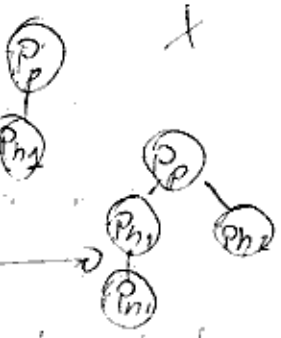
- getppid() → get pid parent: Regresa el pid del padre del que lo invoca.

Errores comunes

- No convivían los dos hijos

```
pid = fork();
if(pidh1 != 0)
{
    X wait()   ← Espera a que termine el hijo 1
    pidh2 = fork();
```

- No crearon bien la jerarquía de procesos




```
pidh1 = fork();
pidh2 = fork();
{
    → código de Pp y Ph2   → los dos los naren.
```

	Pp	Ph1	Pn1	Ph2
pidh1	13	0	0	13
pidh2	14	15 (Ph1)	0	0

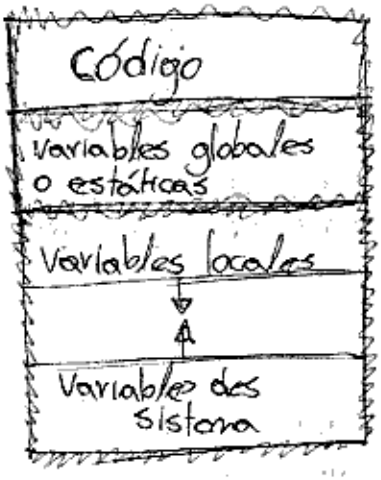
- Distinguir el valor regresado por el wait.

```
pidh1 = fork();
if(pidh1 != 0)
{
    pidh2 = fork();
    if(pidh2 != 0)
    {
        pidv1 = wait(&valor);
        if(pidv1 == pidh1)
            min == valor >> 8;
        else
            max = valor >> 8;
        pidv1 = wait(&valor1);
        if(pidv1 == pidh2)
            max == valor >> 8;
        else
            min = valor >> 8;
```

2-Administración de procesos.

Procesos ligeros o hilos

Un hilo o proceso ligero contiene la misma estructura que los procesos.



Además un hilo puede estar en cualquiera de los estados de un proceso.

El control de los hilos lo tiene el hilo principal.

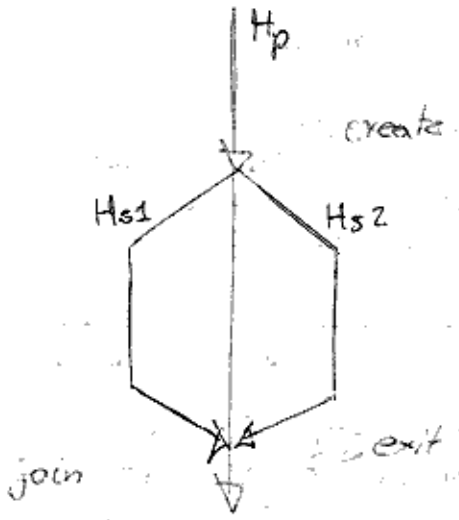
Se pueden tener variables de tipo hilo, a cada hilo se le da un nombre para que el hilo principal los pueda manipular.

- No se invocan por llamadas al sistema.
- Se invocan con la biblioteca Posix.
- Los hilos comparten el código, los archivos que haya abierto el hilo principal antes de crear los otros hilos y los de más importante: variables globales o estáticas.

Los procesos comparten:

- El código y los archivos

2.- Administración de procesos.



join: Es el equivalente a wait()
Hp: hilo principal,
Hs: hilo secundario.

Ejercicio.

- Mismo programa pero con hilos posix, donde el hilo principal cree dos hilos secundarios.

- El hilo secundario 1 obtendrá el mínimo de un arreglo.
- El hilo secundario 2 obtendrá el máximo de un arreglo.
- Los valores del arreglo los definirá el hilo principal y estarán en un rango de 0 a 1000.

Características de los hilos Posix

1.- Consumen menos memoria que los procesos creados con fork. (Ya no copia las variables globales) porque las variables globales son compartidas.

Sólo utiliza las variables locales de la función que va a ejecutar (solo los hilos secundarios).

2.- Cuesta menos crearlos (Ocupa menos tiempo al crearlo porque no pierde tiempo en buscar espacio para las variables globales y locales porque son compartidas).

3.- Ya tienen disponible un mecanismo de comunicación entre ellos. (a través de las variables globales).

2.- Administración de procesos

. Concurrencia (Varios proceso activos al mismo tiempo).

- Se presentan en Multitarea (Donde hay varios procesos activos).
- Considerando que los procesos compiten por los recursos del sistema.
- Tambien en multiprocesamiento. (Dos o más unidades de procesamiento).
- Cuando se tiene un sistema de multitareas y/o multiusuarios se deben manejar de manera correcta la sincronización para que los procesos realicen de manera adecuada su ejecución.

Podemos hablar de tres tipos de procesos concurrentes.

1.- Totalmente independiente.

Los procesos concurrentes pueden ser ejecutados sin que haya dependencia de otros procesos, por lo tanto no hay comunicación entre ellos.

2.- Asíncronos.

Hay cierta dependencia. Es cuando los procesos, en ocaciones requieren cierta sincronización y cooperación.

3.- Síncronos.

Los procesos requieren una total sincronización para que puedan ejecutarse en tiempo y en forma.

T

2- Administración de procesos.

Tres aspectos que hay que cuidar cuando son asíncronos y síncronos.

1.- La comunicación entre procesos. Definir bien el canal de comunicación para que la transferencia de información sea exitosa.

2.- La asignación de recursos. El objetivo de la asignación de recursos es que debe ser justa.

3.- Los procesos se ejecuten en cierto orden.

Situaciones de sincronización que se presentan en procesos concurrentes asíncronos y síncronos.

* Condición de carrera o competencia (Race condition)

Ocurre cuando dos o más procesos o hilos acceden a un recurso compartido sin control. No existe sincronía en estos procesos para acceder al recurso compartido, por lo que suele ocurrir resultados inconsistentes.

// Cada hilo va comprobando las variables globales.

// Si no se maneja bien esto puede haber inconsistencia de datos.

```
if (error == 1)
  {
    return(1);
  }
```

* → pthread_t

error = pthread_create(...);

2- Administración de procesos.

* Barreras

Son mecanismos que se requieren para que un proceso o hilo espere a que otro hilo o proceso termine su ejecución o cálculo, evitando así la condición de carrera

* El join() actua como una barrera.
* wait() para procesos actua como una barrera.

2- Administración de procesos

Regiones críticas y Exclusión Mutua

Código donde se hacen operaciones con datos compartidos entre dos o más procesos o hilos

- Técnicas para el control de acceso a las regiones críticas.

Problemas si no se hace el control de acceso a las variables o datos compartidos:

- Inconsistencia de los datos. (No da siempre el mismo resultado con el mismo problema).

Proceso

- Operaciones sin uso de variables compartidas.

Primitiva de entrada

- Operaciones en la región crítica.

Solo un proceso debe estar en la región crítica

Primitiva de salida

- Operaciones sin uso de variables compartidas.

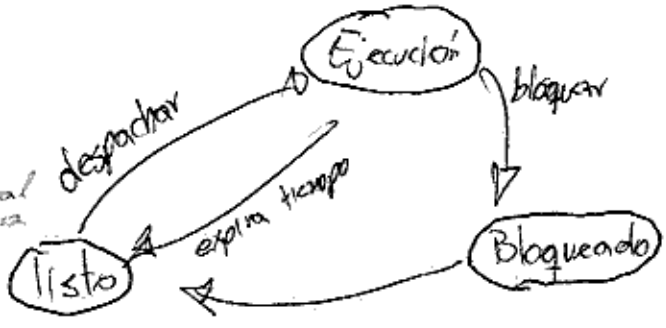
2. Administración de procesos

Algoritmo de Dekker (solo para dos procesos)

```
init num_proc;

proc_uno

void *proc_uno()
{
   while(1)
   {
      tareas_iniciales_uno();
      while(num_proc == 2);      // Primitiva de entrada
      region_critica_uno();
      numero_proc = 2;           // Primitiva de salida
      otras_tareas_uno();
   }
}
```

```
proc_dos

void *proc_dos()
{
   while(1)
   {
      tareas_iniciales_dos();
P.E   while(num_proc == 1);
      region_critica_dos();
P.S.  num_proc = 1;
      otras_tareas_dos();
   }
}
```

```
main()
{
  num_proc = 1;   // Se dice cual proceso entra primero
  <crea_proc_uno>
  <crea_proc_dos>
  <espera_proc_uno>
  <espera_proc_dos>
```



*Todas las diferencias entre procesos e hilos

Los procesos son adoptados

Si el hilo principal termina se lleva a los hilos secundarios

join() : Actua como barrera

2- Administración de procesos

LO QUE ESTÁ ANTES DE LA REGIÓN CRÍTICA ES LA PRIMITIVA DE ENTRADA.

Primitiva de entrada: Código que no permite dos a la vez en la región crítica

LA FALLA DEL ALGORITMO DE DEKKER ES QUE DECIMOS CUAL PROCESO ENTRA PRIMERO A LA REGIÓN CRÍTICA

El proceso debe de decidir cuando entrar.

- La segunda falla es que los procesos entran a la región crítica de manera alterna.

- Tercer error: ~~Cada proceso solo puede entrar una vez a la región crítica.~~

El segundo proceso en entrar solo puede entrar una vez a la región crítica

//clase_10

C: Administración de procesos.

Algoritmo de Dekker VZ Versión 2

```
int proc1_entro,
    proc2_entro;

void *proc_uno()
{
  while(1)
  {
    tareas_iniciales_uno();
P.E. while(proc2_entro==1);
    proc1_entro=1;
    region_critica_uno();
P.S. proc1_entro=0;
    otras_tareas_uno();
  }
}
```

```
void *proc_dos()
{
  while(1)
  {
    tareas_iniciales_dos();
P.E. while(proc1_entro==1);
    proc2_entro=1;
    region_critica_dos();
P.S. proc2_entro=0;
    otras_tareas_dos();
  }
}
```

```
main()
{
  <crea_proc_uno>
  <crea_proc_dos>
  <espera_proc_uno>
  <espera_proc_dos>
}
```

Falla 1: Se puede dar que los dos procesos entren al mismo tiempo a la región crítica. Cuando se le agota el tiempo en la P.E.

En C una variable se ~~inicializa~~ inicializa en 0

//El cerebro se alimenta de glucosa (carbohidratos)
↳ & Amamanta, alegrías

2.- Administración de procesos

Algoritmo de Dekker V3.

```
int proc1_entro, solicita,
    proc2_entro, solicita;

void *proc_uno()
{
  while(1)
  {
    tareas_iniciales_uno();
P.E. proc1_entro = 1;
     while(proc2_entro == 1);
    region_critica_uno();
P.S. proc1_entro = 0;
    otras_tareas_uno();
  }
}
```

```
void *proc_dos()
{
  while(1)
  {
    tareas_iniciales_dos();
P.E. proc2_entro = 1;
     while(proc1_entro == 1);
    region_critica_dos();
P.S. proc2_entro = 0;
    otras_tareas_dos();
  }
}
```

```
main()
{
  <crea_proc_uno>
  <crea_proc_dos>
  <espera_proc_uno>
  <espera_proc_dos>
}
```

Falla 1.- Se bloquean cuando se les expira tiempo antes del while al mismo tiempo (quedan en espera infinita).

7: Administración de procesos

Algoritmo de Dekker Versión 5 (Nos saltamos la 4 porque es muy parecida a la 5)

```
int proc1_solicita,    ---> Son banderas (0, 1)
    proc2_solicita,
    proc_favorecido;   ---> Dice el número de proceso a entrar

void *proc_uno()
{
   while(1)
   {
     tareas_iniciales_uno();
     proc1_solicita = 1;
     while(proc2_solicita==1)
        if(proc_favorecido==2)
        {
          proc1_solicita=0;
          while(proc_favorecido==2);
          proc1_solicita=1;
        }

     region critica_uno();
     proc_favorecido = 2;
     proc1_solicita = 0;
     otras_tareas_uno();
   } /* fin while(1) */
} /* fin proc_uno() */

main()
{ proc_favorecido=2 ; // No es que entre primero sino solo
                      cuando está en conflicto.

   <crea_proc_uno7
   <crea_proc_dos7
   <espera_proc_uno7
   <espera_proc_dos7

}
```

Primitivas de entrada

Primitivas de salida

2.- Administración de procesos

A. Tarea: Con mis propias palabras, decir en dónde y porqué es más eficiente el algoritmo de Peterson que la última versión de Dekker.

Algoritmo de Peterson

- Primitiva de entrada

```
proc1_solicita = 1;
proc_favorecido = 2;
while (proc2_solicita == 1 &&
       (proc_favorecido == 2); // Espera activa
```

- Primitiva de salida

```
proc1_solicita == 0;
```

Otra técnica para evitar la exclusión mutua:

Semáforos

Nuevamente Dijkstra creó el concepto de semáforo. Un semáforo es una variable protegida cuyo valor solo puede ser leído y alterado mediante las operaciones P y V y una operación de asignación de valores iniciales que la llamaremos inicia_semaforo.

Los semáforos se clasifican en dos tipos:

- Binarios: Son aquellos que solo reciben como valores: 0 y 1.
- Contadores: Son aquellos que pueden ser 0 y cualquier número natural.

Operacion P (proberen te verlagen) -> (Intentar decrementar)

Si el semáforo tiene un valor mayor que cero lo decrementa.

tambien se puede encontrar como wait, aequive, down o lock()

```
if (S>0)
   S = S-1;
else
   proceso en cola
   de espera de S;
```

} P(S)

2- Administración de procesos

Operación V (Verhoog) → (incrementar) // Avisa que ya puede continuar otro proceso

```
if(hay un proceso en espera de S)
  dejaq ue prosiga el proceso de la cola;
else
  S = S + 1
```

} V(S)

Aplicación:

Proc Uno

```
while(1)
{
  tareas_iniciales_uno();
  P(S);          ← Primitiva de entrada
  region_critica_uno
  V(S);          ← Primitiva de salida
  otras_tareas_uno();
}
```

Proc Dos

```
while(1)
{
  tareas_iniciales_dos();
  P(S);          ← P.E.
  region_critica_dos();
  V(S);          ← P.S.
  otras_tareas_dos();
}
```

Función principal

```
inicia_semaforo(S, 1)
```

La cantidad de procesos que entran a la región crítica, debe de ser 1.

→ No puede ser 0, porque sucederá un bloqueo mutuo.

* La ventaja con el uso de semáforo es que se pueden usar más procesos con facilidad, ya que en el de Dekker y Peterson aumenta el número de variables.

2. Administración de procesos.

~~Ejemplo típico:~~

Semáforos en el protocolo bloquear - despertar.

Este protocolo se refiere a que un proceso desea que se le avise si ocurre un evento específico y que algún otro proceso sea capaz de detectar la ocurrencia de ese evento para que le avise en el proceso en espera. (No maneja región crítica)

```
semaforo evento;

void *espera_evento() /// * porque es un hilo
{
    tareas_iniciales_espera();
    P(evento)
    otras_tareas_espera();
}

void *detecta_evento()
{
    tareas_iniciales_detecta();
    V(evento);
    otras_tareas_detecta();
}

main()
{
    inicia_semaforo(evento, 0);   → Solo funciona con cero
    <crea espera_evento>;
    <crea detecta_evento>;        & con 1 falla
}
```

Si evento = 0 → todavía no ocurre el evento
evento = 1 → Puede entrar.

---

de Espera infinita: Bloqueos mutuos, estados de i y los quedan en candado

2- Administración de procesos

El mismo pero para tres procesos:

semaforo evento

```
void *espera_evento1()
{
    tareas_iniciales_espera1();
    P(evento);
    otras_tareas_espera1();
}
```

Para solucionarlo poner un V(evento).

```
void *espera_evento2()
{
    tareas_iniciales_espera2();
    P(evento);
    otras_tareas3_espera2();
}
```

Para solucionarlo poner un V(evento).

```
void *detecta_evento()
{
    tareas_iniciales_detecta();
    V(evento);
    otras_tareas_detecta();
}

main()
{
    inicia_semaforo(evento, 0);
    < crea espera_evento1 >;
    < crea espera_evento2 >;
    < crea espera_evento detecta_evento >;
}
```

2.- Administración de procesos

Semáforos contadores

Se pueden controlar un conjunto de recursos.

Los semáforos contadores son útiles cuando hay que asignar un recurso a partir de un banco de recursos idénticos. Su funcionamiento es como sigue:

1.- El semáforo tiene como valor inicial el número de recursos contenidos en el banco de recursos idénticos.

2.- Cada operación P decrementa en 1 el semáforo, indicando que se ha retirado un recurso del banco y que lo esta utilizando algun proceso.

3.- Cada operación V incrementa en 1 el semáforo, lo que indica la devolución de un recurso al banco y que el recurso está disponible para ser asignado a otro proceso.

4.- Si se intenta una operación P cuando el semaforo tiene el valor de Ø, el proceso deberá esperar hasta que se devuelva un recurso al banco mediante una operación V.

Ejercicio

Cierto sistema cuenta con un banco de tres recursos idénticos, los cuales son compartidos por varios procesos. De acuerdo con las solicitudes (P) y liberaciones (V) de recursos que van realizando los procesos llenen la siguiente tabla.

inicia_semaforo (R, 3)

2- Administración de procesos

Proceso	Operación	Valor de R	Situación de la cola
A	P(R)	2	—
D	P(R)	1	—
X	P(R)	Ø	—
D	V(R)	1	—
F	P(R)	Ø	—
G	P(R)	Ø	G
B	P(R)	Ø	BG →
A	V(R)	Ø	B

Ejercicio 2.

El sistema cuenta con cuatro recursos idénticos, llena la siguiente tabla.

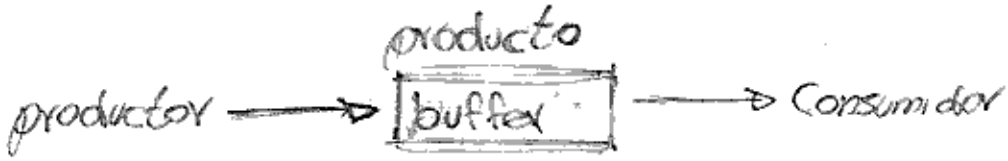
inicia_semaforo (R, 4)

Proceso	Operación	Valor de R	Situación en la cola.
M	P(R)	3	—
N	P(R)	2	—
P	P(R)	1	—
N	V(R)	2	—
X	P(R)	1	—
M	P(R)	Ø	—
P	V(R)	1	—
	V(R)	2	—

2- Administración de procesos

Semaforos Binarios : Productor - Consumidor

Esquemáticamente:



- El productor solo puede poner un producto a la vez.
- El consumidor se debe esperar hasta que haya un producto.
- El productor no puede colocar otro producto si el consumidor no ha tomado el producto.
- El consumidor no puede tomar dos veces el mismo producto, tiene que avisarle al productor que ya lo tomó.

Este algoritmo se necesitan dos semáforos.

```
int buffer;
semaforo num_depositado, num_recuperado;

void *productor()
{
    int datoAdepositar;
    while(1)
    {
        Calcula_dato_depositar();
        P(num_recuperado);      -> Primitiva de entrada
        buffer = datoAdepositar; -> Region critica
        V(num_depositado);      -> Primitiva de salida
    }
}
```

2: Administración de procesos.

```
void *consumidor()
{
    int datoRecuperado;
    while(1)
    {
PE ← P(dato num_depositado);
RC ← datoRecuperado = buffer;
PS ← V(num_recuperado)
        escribe(datoRecuperado);
    }
}

main()
{
    inicia_semaforo(num_depositado, 0);
    inicia_semaforo(num_recuperado, 1);
    <crea productor>
    <crea consumidor>
}
```

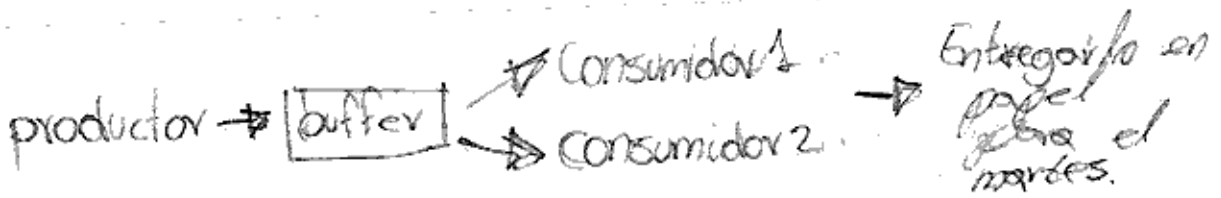
Hay una cola por cada semáforo

Prueba de escritorio.

```
inicia_semaforo(num_depositado, 0);
inicia_semaforo(num_recuperado, 1);
```

Proceso	Operacion	Valor de Pq num_depositado	Valor de num_recuperado
consumidor	P(num_depositado)	Ø/consumidor	1/—
productor	P(num_recuperado)	Ø/consumidor	Ø/—
productor	V(num_depositado)	Ø/—	Ø/—

2.- Administración de procesos



Bloqueos mutuos

Los bloqueos mutuos ocurren porque suceden las siguientes condiciones de Coffman.

En 1971, Edward Coffman describe cuatro condiciones que deben ocurrir simultáneamente para que ocurra un bloqueo mutuo.

1.- Exclusión mutua.	Los procesos reclaman acceso exclusivo de los recursos.
2.- Retención y espera	Los procesos mantienen los recursos que ya les habían sido asignados mientras esperan recursos adicionales.
3.- No espacio apropiatividad.	Los recursos no pueden ser arrebatados de los procesos que los tiene hasta su completa utilización.
4.- Espera circular.	Existe una cadena circular de procesos en que cada uno mantiene a uno o más recursos que son requeridos por el siguiente en la cadena.

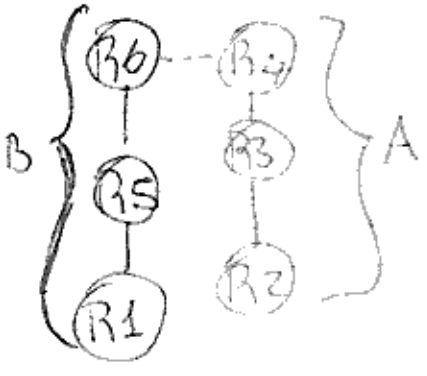
Tercera estrategia de Havender

Se impondrá un ordenamiento lineal a los recursos. Los procesos irán solicitando recursos cuyo ordenamiento sea creciente. Esto se realiza de la siguiente forma.

1.- Todos los recursos se enumeran globalmente, siguiendo un orden creciente del 1 al número de recursos compartidos.

2.- Los procesos pueden solicitar los recursos en cualquier momento, según con un cierto orden numérico (Creciente de recursos); debido a lo cual la gráfica de asignación de recursos no tendrá ciclos.

Proceso	Operación
B	P(R1)
A	P(R2)
A	P(R3)
B	P(R5)
A	P(R4)
B	P(R6)
A	P(R6)



Bloqueos mutuos

- Prevención
- Evitar
- Detección
- Recuperación

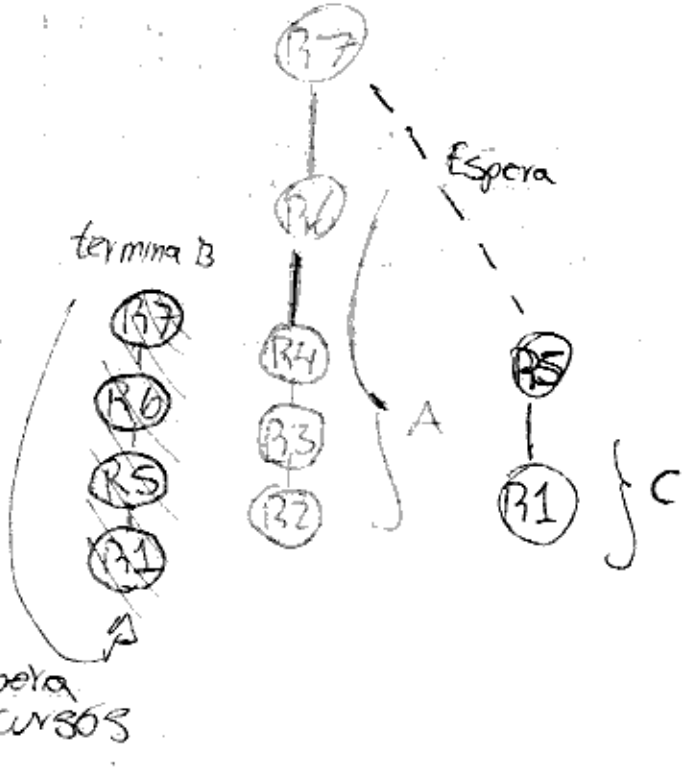
P(R) → Operación de un semáforo que solicita recursos

3. En cada instante uno de los recursos asignados tendrá el número más grande, y así el proceso que lo posea no pedirá un recurso asignado, es decir, si pide un recurso con número menor es muy probable que ya esté asignado y quede en espera infinita.

4. El proceso terminará o solicitará recursos con números mayores que estarán disponibles; por lo que:

- Al concluir liberará sus recursos.
- Otro proceso tendrá el recurso con el número mayor y también podrá terminar.
- Todos los procesos podrán terminar y no habrá bloqueo.

Proceso	Operación
B	P(R1)
A	P(R2)
A	P(R3)
B	P(R5)
A	P(R4)
B	P(R6)
B	P(R6)
C	P(R7)
A	termina
C	P(R1)
C	P(R7)
	R(R5)
	R(7)

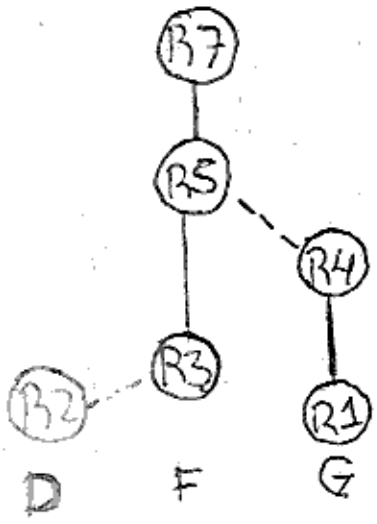


Desventaja
Se ocupan mucho los recursos de número menor y poco los de número mayor.

2. Administración de procesos

Ejercicio (3ra estrategia de Havender)

Proceso	Operación
D	P(R2)
	P(R3)
	P(R1)
	P(R4)
E	P(R3)
	P(R5)
	P(R7)
	P(R5)



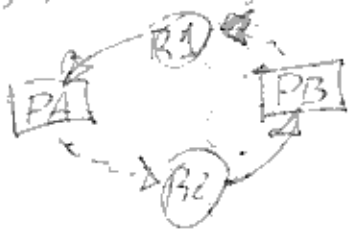
Desventajas de esta tercera estrategia de Havender.

a) Es un ordenamiento demasiado estricto para muchas situaciones del mundo real. Esto no se puede aplicar para problemas de concurrencia.

b) Lleva a los procesos a acaparar recursos de baja numeración.

c) ¿Conduce a inanición?

Bloqueo mutuo simple



Bloqueo mutuo: Mínimo dos procesos no pueden continuar

2.- Administración de procesos

Solución de tarea.


```
void *productor()
{
```

```
main()
{
    inicia_semaforo(dato_depositado2, 0);
    inicia_semaforo(dato_recuperado 2, 1);
    inicia_semaforo(dato_depositado1, 0);
    inicia_semaforo(dato_recuperado1, 1);
```

Usuario = proceso

2: Administración de procesos.

Evitar bloqueos mutuos.

Algoritmo del Banquero (Dikjstra).

n: recursos
p: procesos

Estados seguros.

- Cada proceso especifica por adelantado el número máximo de recursos que necesitarán durante su ejecución.
- El sistema operativo aceptará la petición de un usuario si la necesidad máxima de ese proceso no es mayor que n.
- Un proceso puede obtener o liberar recursos uno a uno.
- Algunas veces un usuario puede verse obligado a esperar para obtener un recurso adicional, pero el sistema operativo garantiza una espera finita.

2- Administración de procesos.

- El número real de recursos asignados a un proceso nunca será superior a la necesidad máxima declarada por ese proceso.
- Si el sistema operativo es capaz de satisfacer la necesidad máxima del proceso, entonces este proceso debe garantizar al sistema operativo que los recursos serán utilizados y liberados en un tiempo finito.

El algoritmo del banquero se basa en permanecer en estados seguros, por lo que, un estado seguro es aquel en el que está el sistema donde aseguran que los procesos podrán contar con sus recursos y así terminar.

Proceso	Préstamo actual	Necesidad Máxima
A	1	4
B	4	6
C	5	8

Para los siguientes casos indican si son estados seguros o inseguros.

→ D

Recursos totales: 12
Recursos disponibles: Préstamo actual R. totales − Préstamo actual = 2

Esto es un estado seguro

2: Administracion de procesos

Proceso	Préstamo actual	Necesidad máxima
A	8	10
B	2	5
C	1	3

Recursos totales: 12
Recursos disponibles: 1

Estado inseguro.

Proceso	Préstamo actual	Necesidad máxima
A	2	9
B	4	6
C	2	9

Recursos totales: 10
Recursos disponibles: 2

Estado inseguro

Proceso	Préstamo actual	Necesidad maxima
A	~~2~~ 3	5
B	2 3	4
C ← Lemma	4 ~~5~~ 6	~~6~~ 5
D	2 3	5

Recursos totales: 12
Recursos disponibles: 2, 4, 0 ~~1~~, ~~5~~ 4 3

Estado seguro

2- Administración de procesos

Ahora con semáforo contador (Con el ejemplo anterior)

Proceso	Operación	Valor (B)	Cola
A	P(B)	2	A
B D	P(B)	2	D, A
C	P(B)	1	D, A
C	P(B)	1	B, D, A
C	P(B)	0	B, D, A
C	Termina	Va 6	B, D, A
—	—	5	B, D
—	—	4	B
—	—	3	—
D	P(B)	2	—
D	P(B)	1	—
A	P(B)	1	A
B	P(B)	0	A

Desventaja:

- Solo para bancos de recursos idénticos
- Los procesos que van llegando quedan en espera.

Ventaja

- No hay inanición.

Alma con dos tipos de recursos

2: Administradores de procesos

	R1		R2	
Proceso	Préstamo actual	Necesidad máxima	Préstamo actual	Necesidad máxima
A	1	4	3	5
B	3	5	2	5
C	1	3	4	8

Recursos totales R1: 7
Disponibles R1: 2

Recursos totales R2: 11
Disponibles R2: 2

No estable

	R1		R2	
Proceso	Préstamo actual	Necesidad máxima	Préstamo actual	Necesidad máxima
A	1	4	3	5
B	3	5	2	4
C	1	3	4	6

R. totales R1: 7
Dis. R1: 2 Estable

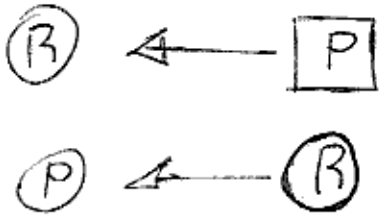
R. totales R2: 11
D. R2: 2

2.- Administradores de procesos

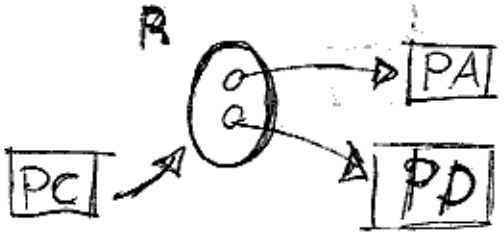
Detección de

Detección de bloqueos mutuos.

Gráficas de asignación de recursos y su reducción.



- El proceso P está asignado al recurso R
- El recurso R está asignado al proceso P



- Cuando un proceso solicita agunta al ciclo mas externo.

Para reducir una gráfica de asignación de recursos se deben seguir los siguientes pasos.

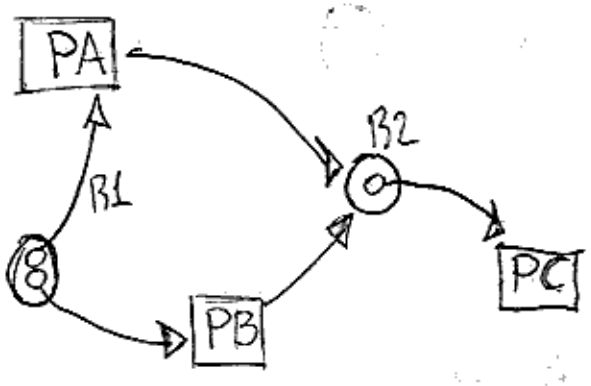
- Cuando a un proceso tiene asignados todos los recursos y no solicita ninguno entonces se reduce la gráfica por ese proceso, eliminando todas las flechas de recursos asignados.

2: Administración de procesos.

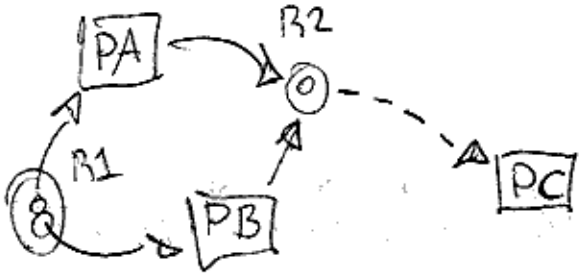
- Si a un proceso que solicita recursos se les puede asignar todos.

Si al final se puede reducir la gráfica por todos los procesos, significa que no hay bloqueo mutuo, en caso contrario, tanto los procesos como los recursos asignados, son los que están en bloqueo mutuo.

Reducir esta gráfica



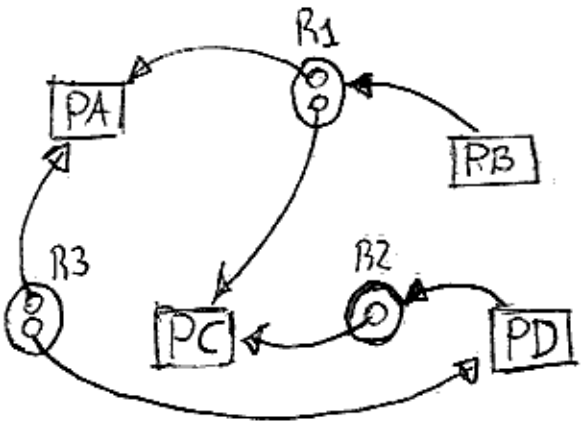
Reducción por PC, libera un R2



Reducción por PA, libera un R1 y un R2
Reducción por PB libera un R1 y un R2

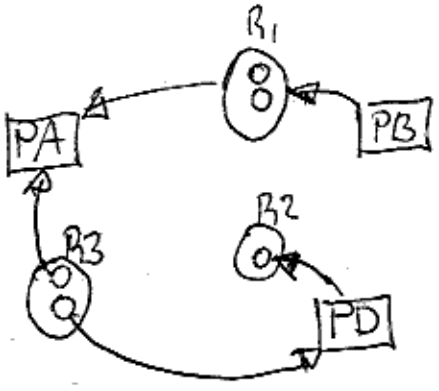
2.- Administración de procesos

Ejercicio. Aplicando el algoritmo de reducción de gráfica, determinar si hay o no bloqueo mutuo en la siguiente situación

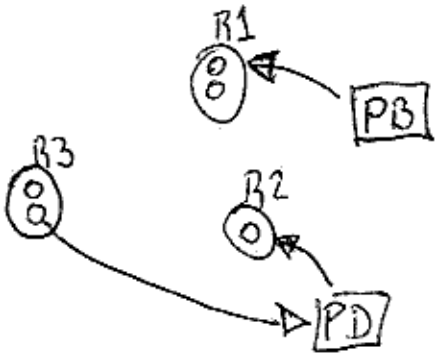


// PA —Solicita→ R1
PA ~~Asigna~~ R1

- Reducción por PC y libera un R1 y un R1

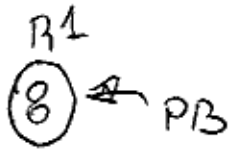


- Reducción por PA y libera un R1 y un R3



2: Administración de procesos

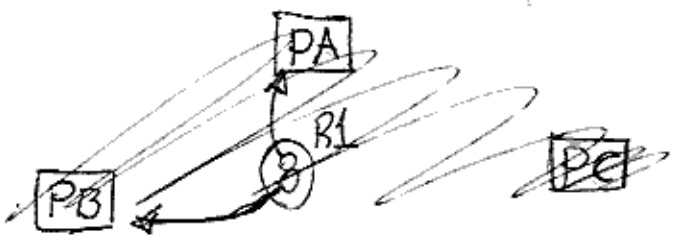
- Se asigna un R2 a PD
- Reducción por PD y libera un R2 y un R3

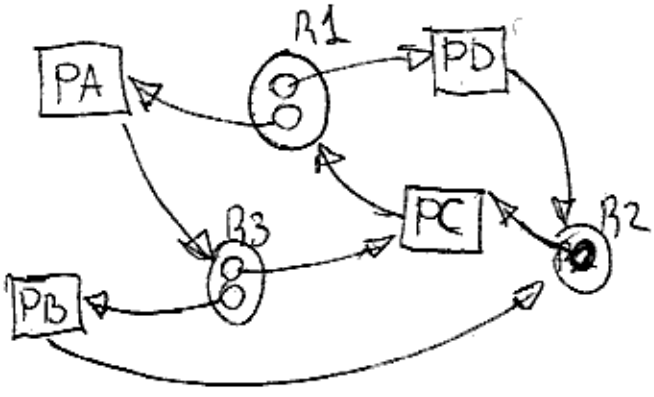


- Se asigna un R1 a PB
- Reducción por PB y libera un R1

∴ NO HAY BLOQUEO MUTUO

2: Administración de procesos



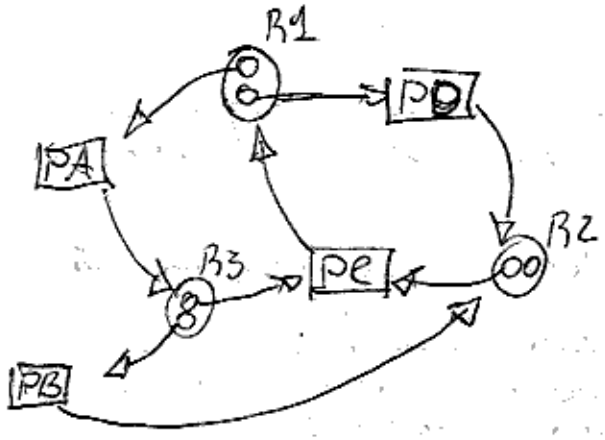


- PA tiene un R1 pero solicita un R3 y no se le puede dar.
- PB tiene un R3 pero solicita un R2 y no se le puede dar.
- PC tiene un R2 y R3 pero solicita un R1 y no se le puede dar.
- PD tiene un R1 pero solicita un R2 y no se le puede dar.

∴ Hay bloqueo mutuo.

- No se puede hacer ninguna reducción.

2: Administración de procesos



- Se asigna un R2 a PB
- Reducción por PB y libera un R2 y un R3
- Se asigna un R3 a PA
- Reducción por PA y libera un R1 y un R2
- Se asigna un R2 a PD
- Reducción por PD y libera un R1 y un R2
- Se le asigna un R1 a PC
- Reducción por PC y se libera un R1, un R2 y un R3

∴ NO HAY BLOQUEO MUTUO

2.- Administración de procesos

Recuperación ante un bloqueo mutuo

- Va de la mano con la detección porque se debe de saber que proceso y recurso están en bloqueo mutuo.

Basicamente hay dos técnicas de recuperación

- Eliminar procesos para liberar recursos: Eliminando el que tiene más recursos o menor prioridad (Elimitar el más importante)
- Usar una técnica efectiva de suspención y reanudación de procesos:
  - Suspender el que tenga más recursos.
  - Suspender el de mayor prioridad.

Aplicar el algoritmo de la avestruz. (No pertenece a ninguna de las 4 técnicas)

No hacer nada cuando tienes miedo como una avestruz.

- Mecanismos de implementación de sincronización.

Manejo de:

- Señales.
- Semáforos.
- Pipes.
- Variables mutex.

- Ejercicio de aplicación de mutex en hilos ~~Posix~~ POSIX
- Obtener la suma de los N elementos enteros de un vector usando N_H hilos.

Sin variables mutex

```
#include <stio.h>
#include <pthread.h>

#define N_H 4
#define N 1000

int A[N], Gsum;

void *codigo_hilo(void *id)
{
   int i, j;
   j = *(int *)id;
   for(i=j; i<j+N/N_H; i++
   {
      Gsum += A[i];
   }
   pthread_exit(id);
}

main()
{
   int i,h, e[N_H];
   pthread_t hilo[N_H];
   int error;
   int *salida;
   for(i=0; i<N; i++)
      A[i] = 1;
   for(h=0; h<N_H; h++)
   {
      e[h] = N/N_H * h;
      error = pthread_create(&hilo[h], NULL, codigo_hilo, &e[h]);
   }
```

Castear el apuntador para tomar su contenido

Región crítica

Al ejecutar varias veces no dará 1000, porque todos los hilos entran y hay intercalación

2: Administracion de procesos

```
for(h=0; h<N_H; h++)
{
   error = pthread_join(hilo[h], (void **) &salida);
   printf("\n Hilo terminado que suma de %d a %d",
          *salida, *salida+N/N_H-1);

   printf("\n Gsum= %d \n", Gsum);
}
```

Ahora con variable mutex.

```
:

pthread_mutex_t gMutex; → Cuando se crea tiene el valor de 1.

void *codigo_hilo(void *id)
{
   int i, j;
   j = *(int *) id;
   for(i=j; i<j+N/N_H; i++)
   {
      pthread_mutex = lock(&gMutex); → Primitiva de entrada
      Gsum += A[i];                  → Región crítica
      pthread_mutex_unlock(&gMutex); → Primitiva de salida
   }
   pthread_exit(id);
}
```

2: Administración de procesos

```
main()
{
  ...
  for(i=0 ; i<N ; i++)
  {
    A[i] = 1
  }
  pthread_mutex_init(&logMutex, NULL);
  ...
```

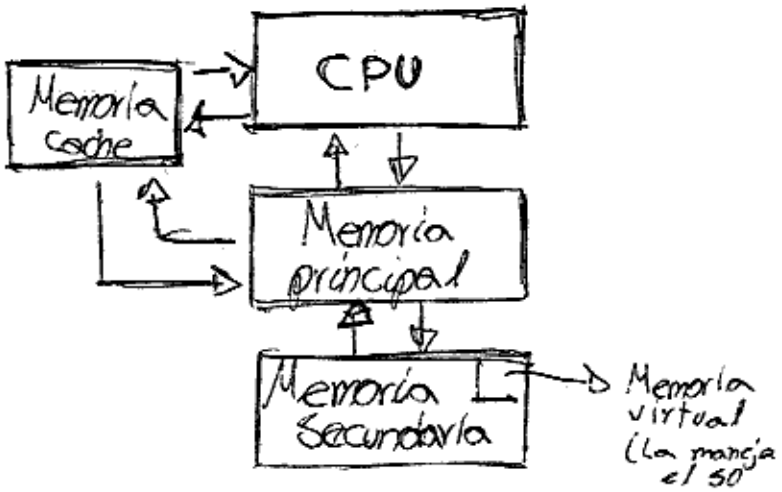
3.- Administración de Memoria

Objetivo: El alumno exp

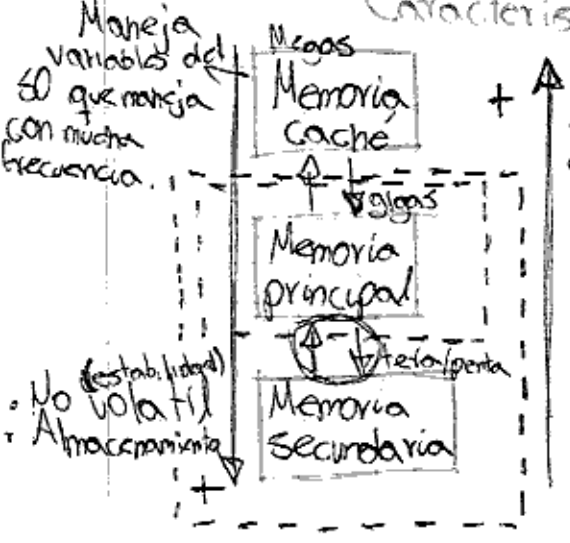
Funciones y operaciones de la administración de memoria.

- Asignar memoria ~~al~~ principal a los procesos (código y datos de los procesos).
- Liberar espacio en memoria para asignarcla a otros procesos de mayor prioridad o importancia.

Jerarquía de memoria.



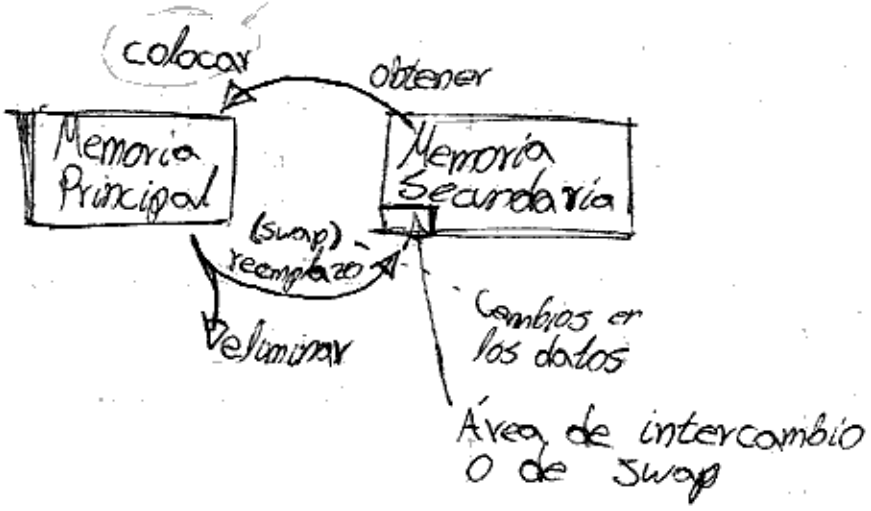
Características comparativas entre los tres tipos de memoria



- Velocidad de acceso a la información.
- Es volatil
- Costo (Un mega vale más arriba)

3= Administración de memoria

Operaciones entre Asignación de memoria



Asignación de memoria

Asignación contigua

- Todo el proceso esta en Memoria.
- Códigos y datos están de manera consecutiva en memoria.

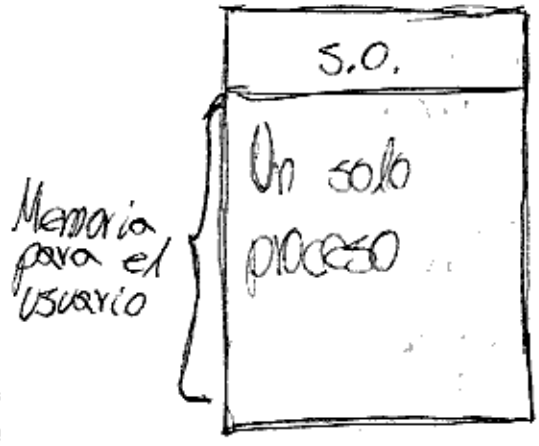
Asignación no contigua

- El proceso se divide en partes y no todas las partes están en memoria.
- Cada parte se almacena de manera consecutiva pero no necesariamente las partes que están en memoria están contiguas.

3. Administración de memoria

Técnicas de asignación contigua de memoria

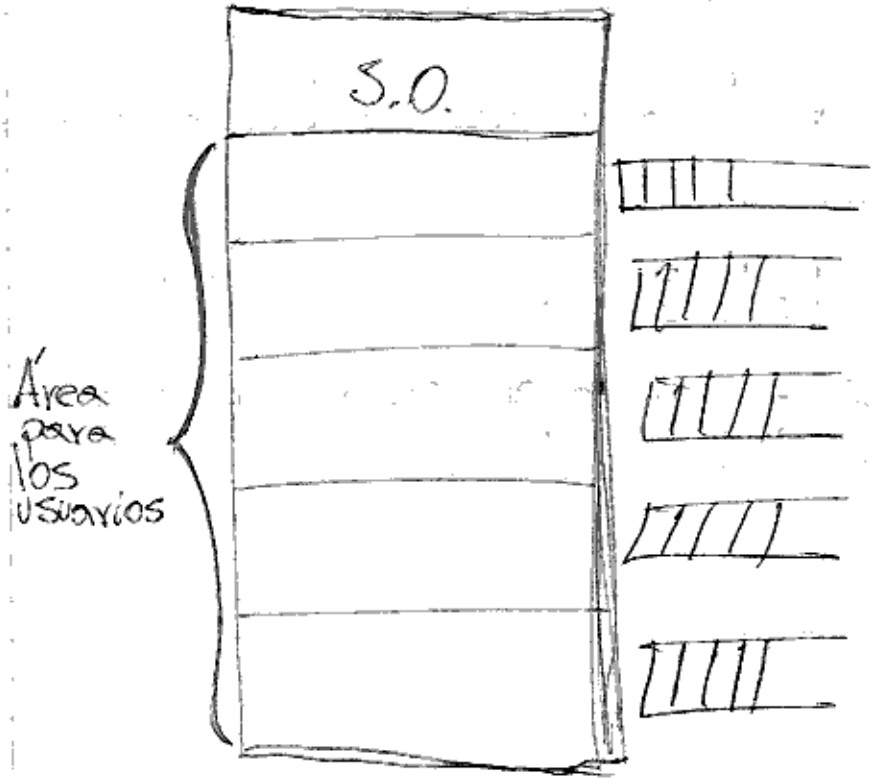
- Asignación Contigua simple



- El SO está compuesto de programas así como de datos.
- Ejemplo clásico MSDOS

Se divide el proceso y cada parte se llama OVERLAL.

- Multiprogramación con particiones fijas.



- Pueden ser de diferente tamaño pero la memoria es fija y no se puede cambiar.
- Debe entra el proceso completo (tanto código como los datos).
- Cada partición maneja una cola de procesos.

3- Administración de memoria

Traductores y compiladores pueden ser → Intérpretes (Al momento)

Se tienen dos técnicas de asignación de memoria en multiprogramación con particiones fijas.

- Traducción y carga absolutas.

  - Se utilizan compiladores y ensambladores que generen código binario con direcciones absolutas. De esta manera el programa ejecutable se colocará en la misma partición de memoria para su ejecución, si está ocupada la partición se va a la cola.

    - Puede suceder inanición, porque un proceso no termina y le siguen llegando más procesos.
    - Puede pasar que varios procesos se ejecuten en la misma partición.

- Traducción y carga con reubicación

  - El manejo de las direcciones que tienen los programas ejecutables son relativas. Como son relativas tienen que pasar por un proceso de carga con traducción, absolutas en una partición de memoria que esté vacía o la cola esté pequeña.

  - Fragmentación en la multiprogramación, con particiones fijas.

    Fragmentación: Espacio que no se ocupa dentro de la partición donde se ubica un procesos.

    Cada vez que termina un proceso se libera una fragmentación para que pueda entrar otro proceso.