Apuntes de Sistemas Opertaivos I

Mauricio Elian Delgadillo Garcia

Índice general

Capítulo 1

Conceptos Introductorios

Hw = Hardware, Sw = Software

1.1. Que es un Sistema Operativo?

Un Sistema Operativo es un conjunto de programas, de un sistema informático, que administra los recursos del hardware.

1.1.1. Para quienes el SO administra?

Para los Procesos

1.1.2. Cuáles son los recursos del hardware?

Seria bastante largo enumerar todos los recursos del Hw. Sin embargo, podemos agruparlos en cinco rubros:

- La CPU: O también las CPU's, pues una computadora puede tener varios procesadores.
- La RAM.
- La Memoria Secundaria: Se refiere a los CD, DVD, Discos, Memoria 'Flash', etc.
- Los Dispositivos I/O (Entrada/ Salida): Impresora, pantalla, teclado, mouse, tarjeta de vídeo, etc.
- La Red: Se refiere a los datos que transitan por ella.

El SO "presta" un recurso del Hw por un tiempo a un proceso. Luego que proceso ya no lo necesita, el SO libera al recurso (es decir, le "quita" el recurso al proceso que lo estaba usando)

1.1.3. Que es un Proceso?

En realidad, en esta presentación no se podría dar la definición exacta de proceso, dado que el temario abordado aquí es demasiado superficial. Sin embargo, nos conformaremos con una definición informal:

Un **proceso**, a grandes rasgos, es un **programa** en ejecución (run).

A un programa, usualmente se le llama aplicación, aunque es mas conocido por su abreviatura: App, entonces, simplemente:

$$\text{Proceso} = \text{App corriendo}$$

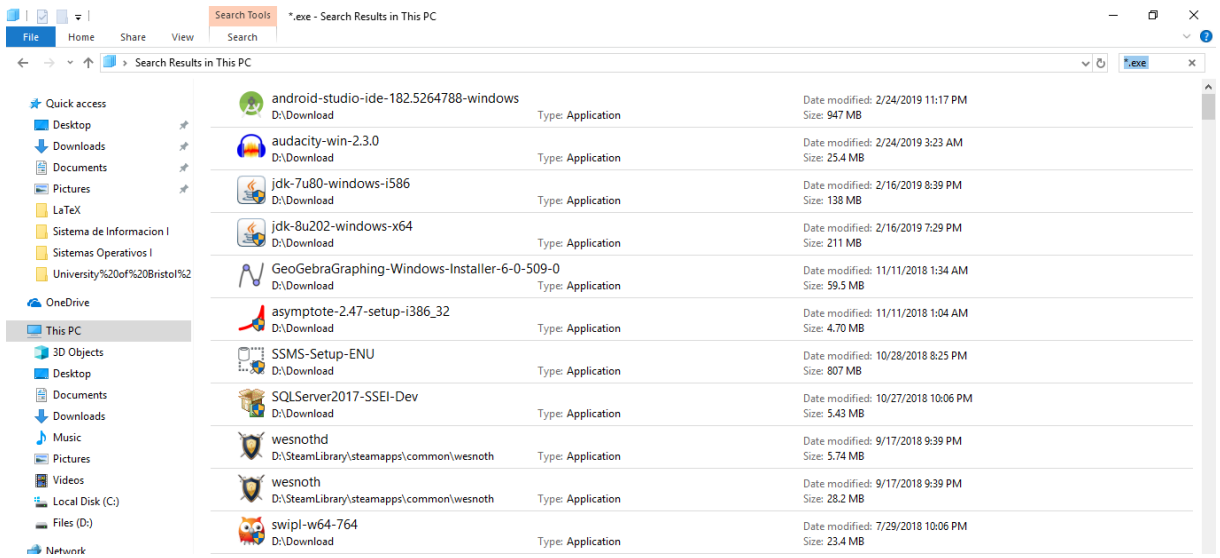
Al igual que una imagen o audio, una App es simplemente un archivo que esta almacenado en la memoria secundaria. A éstos archivos que son App's se les llama 'archivos ejecutables'.

En Windows, los 'archivos ejecutables' (App's) tienen la extension .exe

Dado que Linux no maneja formalmente el concepto de 'extensión', las App's, en este SO, se pueden almacenar en archivos sin extension.

Problema 1

En Windows se hizo una búsqueda de archivos ejecutables .exe (App's) y se obtuvieron muchos. *Estos archivos son procesos?*

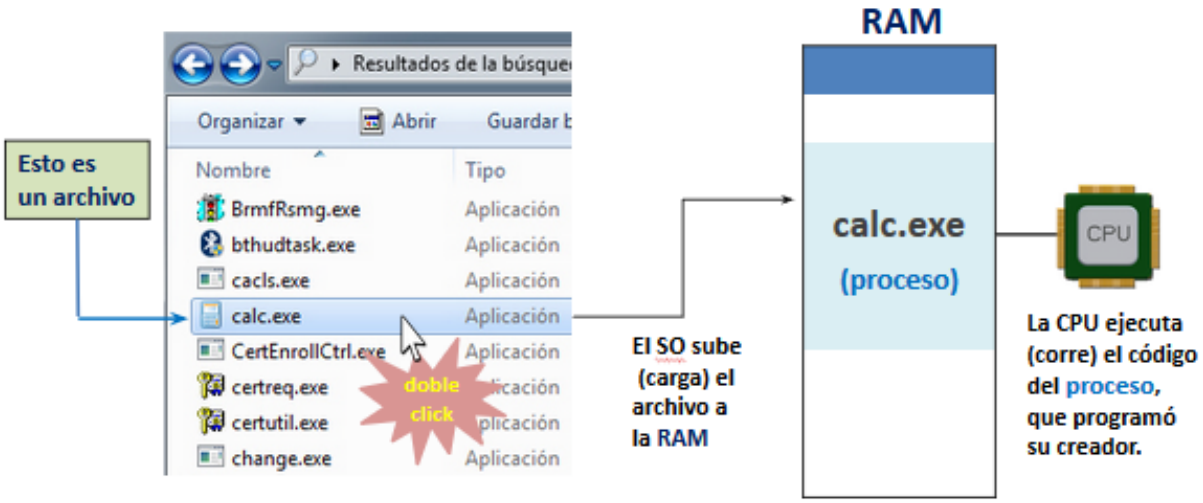


Respuesta: No. Los .exe encontrados son solamente Archivos.

Para que un archivo .exe (Una App) se convierte en proceso, se lo debe "correr" con algunas de estas ordenes:

- Hacer doble-click sobre el.
- Situar el mouse sobre el, pulsar botón derecho y escoger la opción 'Abrir'

Una vez recibida la orden de correr (run), el SO cargar[1] el archivo .exe a la RAM y entonces la CPU empieza a ejecutar su código.



Entonces, para que una **App** (archivo ejecutable, .exe) se convierte en proceso, la **App debe** estar cargando en la **R.A.M.** y ser ejecutada por la **C.P.U.** Así:

Proceso = **App** corriendo (run) = **App** cargada en la **R.A.M.** y siendo ejecutada por la **C.P.U.**

Recuerda: "Todo lo que se procesa debe estar cargado en la R.A.M."

1.2. Tipos de Sistemas Operativos

Según la gestión del procedimiento, los SO se dividen en dos tipos:

- SO Monoprocesos o Monotarea.- Son aquellos que son capaces de ejecutar a lo sumo UN proceso a la vez.

  El mas destacado de este tipo de SO, fue el MS-DOS® (Micro Soft-Disk Operating System), creado en 1981 por Microsoft.

[1] El termino "cargar" usualmente es sustituido por la palabra "subir", del ingles, upload.

- SO Multiprocesos o Multitarea.- Son aquellos que tienen la habilidad de mantener en estado de ejecución a uno o más procesos. Los SO mas conocidos son: Windows, Linux, Android,...etc.

  Actualmente, es inconcebible que un SO no sea multiproceso, pues la gente esta acostumbrada a correr mas de un proceso a la vez (e.g. escuchar música y navegar en Internet).

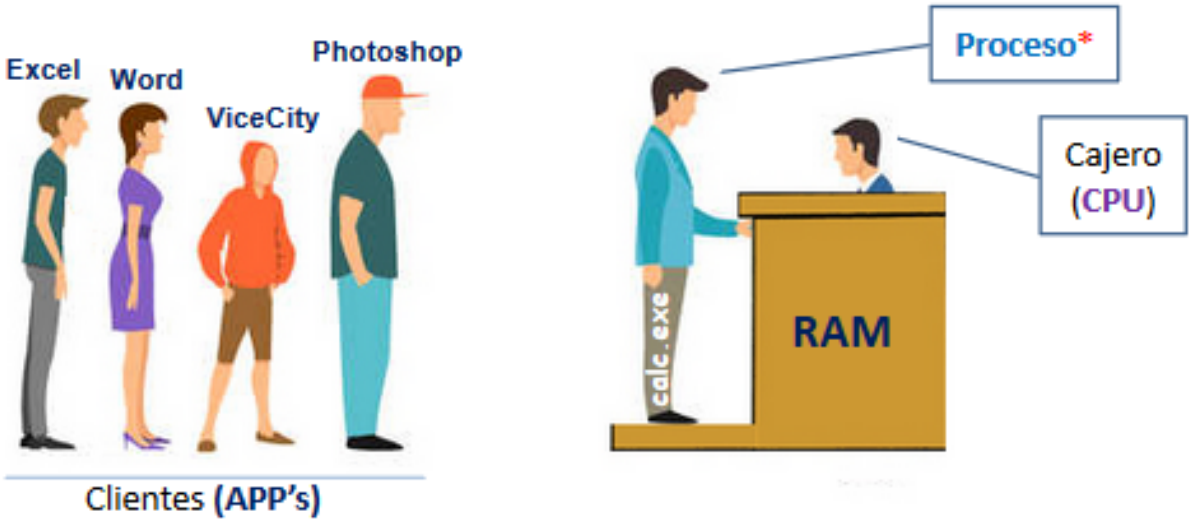
Para entender mejor, supóngase que una persona desea usar la computadora para escribir un documento y escuchar música. Para ello escoge las App's: 'Microsoft Word' (winword.exe) y el 'Reproductor de Windows Media' (wmplayer.exe).

- Si la computadora tiene un **SO Multiproceso** (e.g. Windows) esta persona, podrá correr ambas App's 'sumultáneamente': es decir, mientras escribe su documento en Word, podrá escuchar música con el Reproductor (el SO es capaz de ejecutar los procesos winword.exe y wmplayer.exe a la vez)

- Si la computadora tiene un **SO Monoproceso**, este individuo, solo podrá cargar una App a la vez. Es decir, si decide escribir su documento en Word, no podrá escuchar música con el Reproductor (y viceversa), pues el SO solo puede ejecutar un solo proceso a la vez (winword.exe o wmplayer.exe, pero no ambas)

Problema 2

En un Banco hay un solo cajero que atiende a todos los clientes. Si hacemos la analogía: cajero ≡ CPU, clientes ≡ App's. **Este Sistema de atención es Mono o Multiproceso?**



* Es un **Proceso**, porque ésta **App** está cargada en la **R.A.M.** y está siendo atendido por la **C.P.U.**

Respuesta: El Sistema de atención es Monoproceso, porque el cajero (CPU) atiende a un cliente (proceso) a la vez.

Es decir, cada vez que el cajero termina de atender a un cliente, recién podrá atender al próximo. En ningún momento, el cajero podrá atender a dos o mas clientes a la vez.

Pero, aunque el Sistema es Monoproceso, la cola muestra el orden de *atención de las APP's por parte de la CPU* (Ejecución de la APP (run)).

Así, observando el gráfico anterior podemos deducir que: luego que **calc.exe** finalice, la APP **Photoshop** correrá (run): luego que Photoshop termine su ejecución, se ejecutará (run) **ViceCity**: y así sucesivamente, hasta completar todas las APP's de la cola.

A esta forma de trabajo, se le llama "**Procesamiento por lotes**" o "**Batch**"

1.3. Batch

En los S.O. es posible realizar el Procesamiento por Lotes, mas conocido como "**batch**", simplemente creando un archivo texto, en la cual se anoten los nombres de los ejecutables (APP's) que queremos correr.

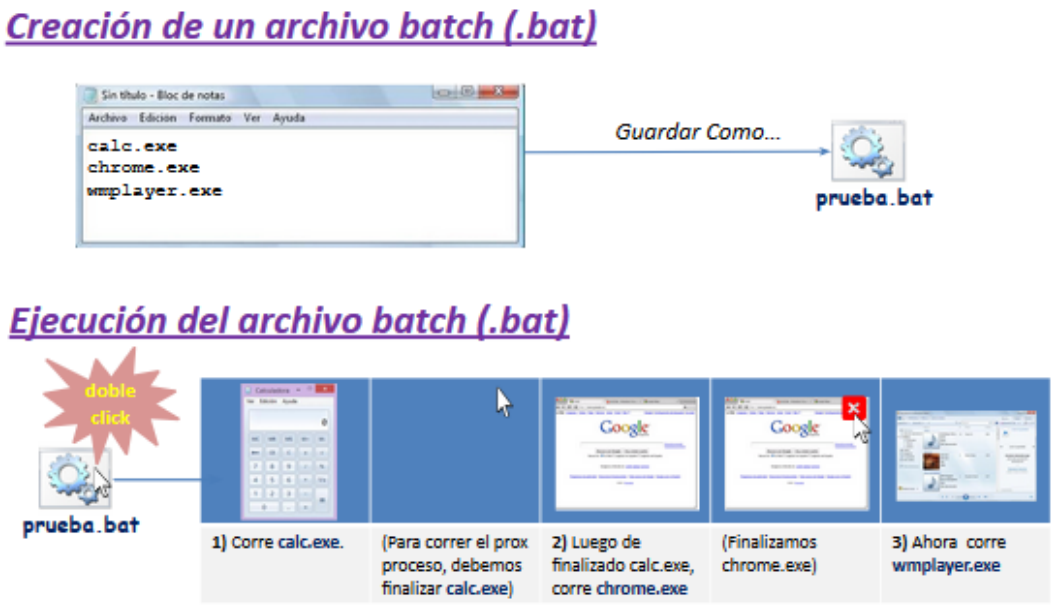
Por ejemplo en Windows para crear un Archivo de Procesamiento por Lotes:

1) Abrimos un Editor (e.g. el Block de Notas).
2) Escribimos los nombres de los archivos ejecutables, en el orden en que queremos correrlos (e.g. calc.exe, chrome.exe, wmplayer.exe)
3) Guardamos este texto con la extension **.bat** o **.cmd***

*Un archivo .bat o .cmd, en realidad es considerado un programa escrito en un lenguaje llamado *shell-script*. Por este motivo, es posible crear archivos .bat o .cmd, con un contenido mas complejo.

Luego, podemos ejecutar este archivo batch, simplemente haciendo doble-click sobre él.

Recuerde: El proceso **batch** es **monoproceso**.



1.3.1. Implementación del Multiproceso

Comó la CPU es capaz de ejecutar a más de un proceso "simultaneamente"?

Actualmente, la técnica usada para lograr esto, se llama Tiempo Compartido o "**Time- Sharing**". Esta Técnica, consiste en asignar reiteradamente la **C.P.U.** a c/u de los procesos, un intervalo de tiempo llamado "**quantum**" (**q**) o "**Time-Slice**"

Por ejemplo. Supongamos que tenemos tres procesos cargados en la R.A.M.: p1.exe, p2.exe y p3.exe : La C.P.U. atiende 1**q** de tiempo a p1; luego 1**q** a p2; después 1**q** a p3; luego 1**q** a p1; luego 1**q** a p2, etc.

La **C.P.U. cambia de un proceso a otro** tan rápidamente, que el usuario humano imagina que los tres procesos corren simultáneamente. (Al cambio de proceso por parte de la C.P.U., se le llama **Cambio de Contexto** o **switch-context**)

1.3.2. S.O. Mono y Multi-usuario

S.O. Mono-usuarios : Son aquellos que proveen servicio y procesamiento a **un solo** usuario a la vez.

S.O. Multi-usuarios : Se le llama multi-usuario a la característica de un S.O. que permite proveer servicio y procesamiento a uno o más usuarios **simultáneamente**

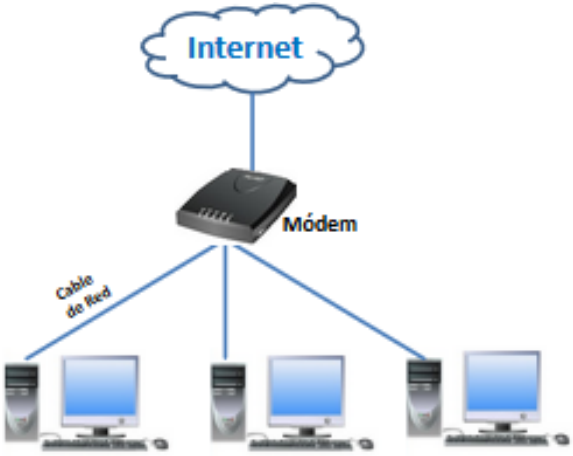
En general, si en una instalación informática vemos a 1 ó más *terminales** conectadas a **una** sola **Computadora**, decimos que estamos en presencia de un Ambiente Multi-usuario (es decir, el S.O. está en Modo Multi-usuario).

La **Computadora** a la que comúnmente se le llama **Servidor**, es la que provee los recursos del Hw (C.P.U., R.A.M., Tarjeta de Vídeo, Disco Duro, etc) y por tanto, el S.O. está instalado y corriendo en ésa **Computadora**.

*Un **terminal** o **consola** es un dispositivo electrónico que se utiliza para interactuar con un computador. Es una maquina que incluye teclado y pantalla (y mouse) y es usada para introducir (input) u obtener (output) datos, y mostrarlos (print)
Siempre se debe recordar que una terminal **no procesa nada**, solo recibe/envía datos desde/hacía la Computadora Servidor.

1.3.3. Problema 3

Es un pequeño Café-Internet o Ciber-Café se tomó una fotografía (mostrada a la izquierda) y se averiguó como estaban conectadas sus computadoras (diagrama de la derecha)



El ambiente es **Monousuario o Multiusuario?**

Respuesta: El ambiente es **Monousuario**, porque cada computadora atiende a un y solo un usuario. En cada una de las tres computadoras, corre un S.O. que atiende y procesa los requerimientos de sus respectivos usuarios (Sería Multiusuario si usando **una sola** computadora, más de un usuario se podría conectar a ella)

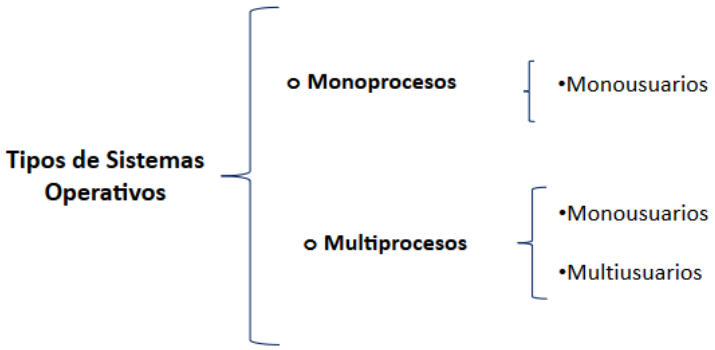
1.3.4. Problema 4

Tengo **una** sola **computadora**. En ella, la pantalla de Login me muestra que hay 4 usuarios que usan ésta máquina. Puede concluir que mi computadora es Multiusuario?

Respuesta: **NO**. Porque en mi computadora, **solo un** usuario puede trabajar a la vez. Lo que si puedo afirmar, es que el S.O. maneja "cuentas de usuario" o "user accounts". Así, cada usuario que usa mi **computadora**, tendrá su propio ambiente de trabajo ("escritorio"), permiso y acceso a ciertas APP's, etc.

*De Wikipedia: En el ámbito de la seguridad informática, **log-in o log-on** (**ingresar, entrar, "iniciar sesión"**) es el proceso mediante el cual se controla el acceso individual a un sistema informático a través de la identificación de los credenciales (Nombre Usuario + Contraseña) provistos por el usuario. En contraste, **log-out** (**salir, "cerrar sesión"**) es el término que se emplea, cuando el usuario deja de trabajar con su cuenta (account).

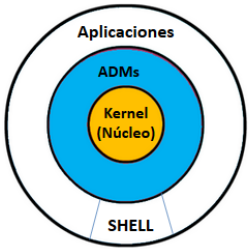
1.3.5. Resumen

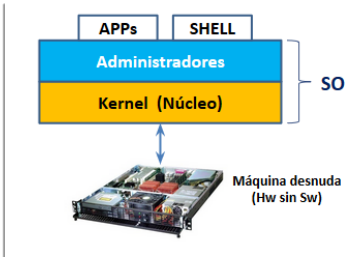


1.4. Arquitectura de un S.O.

Los siguientes dos diagramas, nos muestran que los S.O. se divide en dos grandes capas: El **Kernel** o núcleo y los **Administradores**.

Este diagrama (Derecha) es el más conocido de todos y es llamado **"Diagrama Cebolla o de Estratos"**. **"Diagrama Plataforma o Soporte"** (Izquierda). Muestra al SO como sosteniendo a las APPs y permitiendo que las mismas se comuniquen con el Hw. Así :

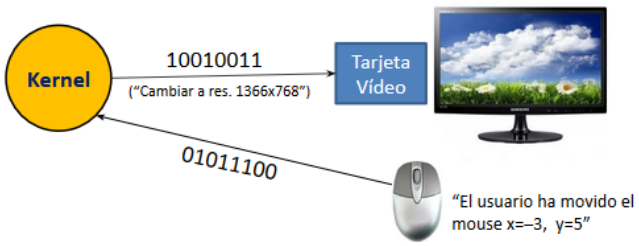




S.O. = Plataforma de Software

1.4.1. El Kernel

Es aquella parte del S.O. que interactúa directamente con los componentes del Hardware (Hw). Como se sabe, un Hw solo entiende instrucciones representadas por **números binarios** (lenguaje máquina de ése Hw) y es por éste motivo que el Kernel, en su gran mayoría, está escrito en Assembler.



En este pequeño ejemplo, el Kernel le dice al vídeo que cambie la resolución de pantalla a 1366x768. También, el mouse le comunica al Kernel que el usuario lo ha movido (así, un programa moverá el cursor).

Entonces, claramente, el Kernel (núcleo) es el módulo del S.O. más difícil y tedioso de implementar: **Para cada uno** de los componentes del Hardware (Hw), se debe escribir el código binario que recibe/envía datos desde/hacia ése Hw.

Pero, para facilitar su manipulación, el Kernel se presenta a los Administradores como una abstracción del Hw o como una serie de abstracciones de c/u de los componentes del Hw. Es decir, como un "Tipo de Abstracto de datos", que se manipula por procedimientos y funciones (métodos). Estos procedimientos y funciones que componen el Kernel, usualmente, están escritos en el lenguaje C.

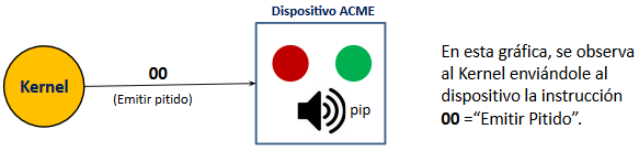
```
class Kernel{
    //... operaciones de la pantalla
  void setMode(byte modo){ //Cambia la resolución de la pantalla.
    mov ah, 13h
    mov al, modo
    int 10h
  }

  byte getMode(){ //Obtiene el nro que indica la resolución de pantalla.
    byte modo;
    mov ah, 13h
    int 10h
    mov modo, al
    return modo
  }

    //... operaciones del mouse. Rutina de interrupción
  int getMouseX(){ //Obtiene el desplazamiento horizontal del mouse
    ...
  }
}
```

Nota: Realmente el Kernel NO es una clase. Aquí, con fines didácticos, lo mostramos así.

1.4.2. El Kernel y los Drivers

Supóngase, absurdamente, que la IBM-PC "nació" con un dispositivo I/O, llamado "ACME", que tiene dos LED's (rojo y verde) y un sintetizador que produce un pitido.

Este dispositivo se conecta (enchufa) en el Puerto 225 y acepta las siguientes instrucciones de dos bits: **00**=Pitido, **01**=blink (parpadear) LED rojo, **10**=blink LED verde, **11**=blink ambos LED's.



Y, una vez más, el Kernel proveerá procedimientos para manipular al dispositivo ACME.

```
class Kernel{
    //...operaciones del dispositivo ACME
  void AcmeBeep(){  //Emitir pitido
    out 255, 0     //Enviar 00 al puerto 255
  }

  void AcmeRedBlink(){  //Hacer  parpadear al LED rojo.
    out 255, 1     //Enviar 01 al puerto 255
  }

  void AcmeGreenBlink(){  //Hacer  parpadear al LED verde.
    out 255, 2     //Enviar 10 al puerto 255
  }

  void AcmeBothBlink(){//Hacer  parpadear a ambos LED's.
    out 255, 3     //Enviar 11 al puerto 255
  }
  ...
}
```

Pero..., de pronto un fabricante construye un ACME2. Es decir, un dispositivo ACME mucho más mejorado. Aún más, este nuevo ACME2 trae instrucciones codificadas a tres bits. Estas son: **100**=Pitido, **101**=blink (parpadear) LED rojo, **110**=blink LED verde, **111**=blink ambos LED's.

Para que nuestro S.O. trabaje con este nuevo dispositivo, no tenemos que más remedio que modificar el Kernel (operaciones `AcmeBeep()`, `AcmeRedBlink()`, ...): Es decir, creemos una nueva versión de nuestro S.O.

Así, cada vez que se construya un nuevo dispositivo (e.g. ACME3), debemos modificar el Kernel, obteniendo así una nueva versión de nuestro S.O. (Todo esto se soluciona con el uso de **Drivers**).

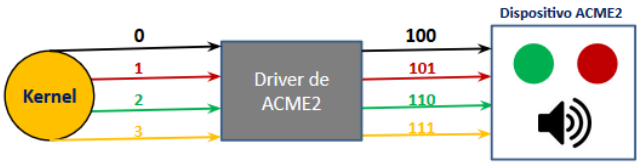
Wikipedia: Un **Device Driver** o simplemente **Driver** (en Windows: **controlador**), es un programa informático que permite al Kernel interaccionar con un periférico, haciendo una **abstracción del hardware** y proporcionando una interfaz (posiblemente estandarizada) para utilizar el dispositivo.

Para entender mejor, supongamos que nuestro Kernel ahora hace uso de un Driver para acceder al dispositivo ACME. Entonces, nuestro Kernel no se comunica directamente con el Hw ACME, sino con su Driver: El Kernel se abstrae, porque imagina que el Driver es el dispositivo ACME.

Si observamos ahora el código Kernel notaremos que las operaciones no envían (send) datos al puerto 255, sino que hacer uso del método "send" del Driver.

```
class Kernel{
    //...operaciones del dispositivo ACME
  void AcmeBeep(){  //Emitir pitido
    Driver.send(0)     //Enviar 00 a ACME
  }

  void AcmeRedBlink(){  //Hacer  parpadear al LED rojo.
    Driver.send(1)     //Enviar 01 a ACME
  }

  void AcmeGreenBlink(){  //Hacer  parpadear al LED verde.
    Driver.send(2)     //Enviar 10 a ACME
  }

  void AcmeBothBlink(){    //Hacer  parpadear a ambos LED's.
    Driver.send(3)     //Enviar 11 a ACME
  }
  ...
}
```

En la gráfica de la derecha se muestra al Kernel enviando (send) al Driver de ACME2 los códigos 0,1,2 y 3; y vemos que éste Driver **traduce** estos valores a los códigos que entiende el Hw ACME2 (0 lo reemplaza por 100, 1 por 101, 2 por 110 y 3 por 111).



En definitiva, si un fabricante decide crear una versión diferente de un dispositivo (e.g. un ACME3), este fabricante deberá crear un Driver para este nuevo dispositivo. Así, el Kernel enviará datos al Driver como si fuese el dispositivo de la primera version (o versión estándar), y el Driver **convertirá** estos datos en valores que entiende la nueva versión.

1.4.3. Los Administradores

Un Administrador (**ADM**) es un módulo que se encarga de gestionar un tipo de recursos del Hardware (Hw). Idealmente, se han propuesto cinco administradores:

- ☐ **ADM de Procesos o del Procesador:** Es el encargado de asignar la(s) C.P.U.(s) a los procesos.
- ☐ **ADM de Memoria:** Gestiona la memoria R.A.M.
- ☐ **ADM de Información:** Es el encargado de administrar los datos de la Memoria Secundaria.
- ☐ **ADM de Dispositivo I/O:** Permite a las APP's trabajar con los periféricos. Es precisamente este ADM, el que utiliza las operaciones del Kernel que interactúan con Drivers
- ☐ **ADM de Red:** Gestiona todo lo concerniente a los datos que transitan por la Red.

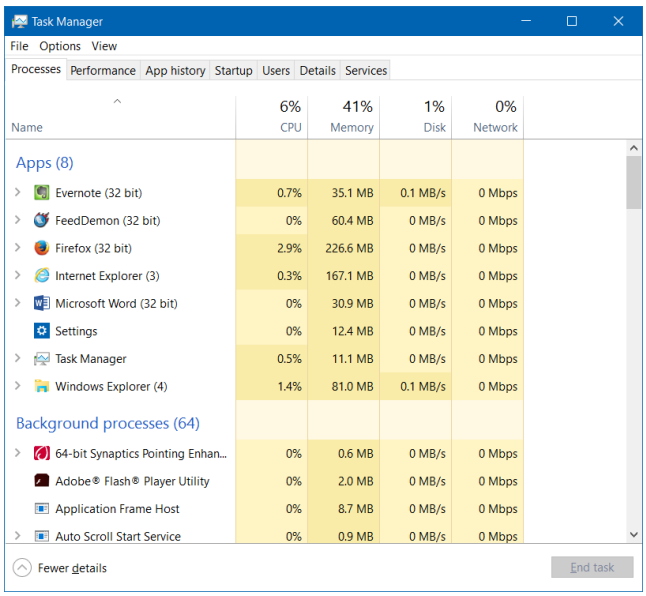
Al igual que el Kernel, un **Administrador** del S.O., se presenta como un conjunto de procedimientos o funciones que pueden ser **usados** por las **APP's**. Por ejemplo:

```
class DeviceAdm{    //Adm de Dispositivos. En la realidad, un ADM no es una class.
  …
  void setScreenResolution(byte modo){  //Cambia la resolucion de la pantalla.
    if (Kernel.getMode() != modo)
        Kernel.setMode(modo);
  }

  byte getScreenResolution(){  //Obtiene el nro que indica la resolución de pantalla.
    return Kernel.getMode();
  }
  …
}
```

Aunque este ejemplo es muy pequeño, en él se puede apreciar como el **ADM** implementa sus operaciones, comunicándose con el **Kernel**.

Problema 5

En general, un usuario de Windows corre **"Administradores"**, para gestionar algún periférico elemento del Sistema (e.g. "El ADM de Tareas", "ADM de Impresión", etc) **Dónde se localizan éstos Administradores?**



Respuesta: En realidad son **Aplicaciones (APP's)** y no **Administradores***, en el contexto de un S.O., a estas **APP's** le suelen llamar "**Manager**" (**MGR**), que al español lo traducen como "Administrador". Por ejemplo, "**El ADM de tareas**" es una APP, cuyo ejecutable se llama "**taskmgr.exe**"

*Siempre se debe recordar que los **Administradores** de un S.O. **no tienen ventanas**, y por lo tanto el usuario humano no puede verlos. Sin embargo, excepcionalmente, un **ADM** mostrará un mensaje cuando ocurra un error grave en el Sistema (a esto se le denomina **Panis**). El mensaje de error más temido en Windows es el emitido por el **ADM de Dispositivos:** "La pantalla azul"

1.4.4. El Shell

Wikipedia: El **shell** o **intérprete de órdenes** o **intérprete de comandos** es una **App** que provee una interfaz de usuario para acceder a los servicios de un Sistema Operativo.

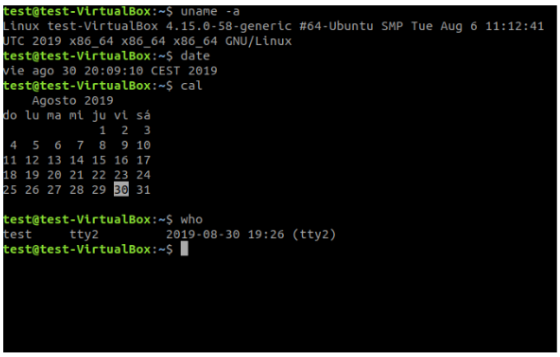
Dependiendo del tipo de interfaz que empleen, los shells pueden ser:

- **Shell-CLI:** Command-Line Interface (solo texto).
- **Shell-GUI:** Graphical-User Interface (con gráficos).

CLI

Command Line Interface

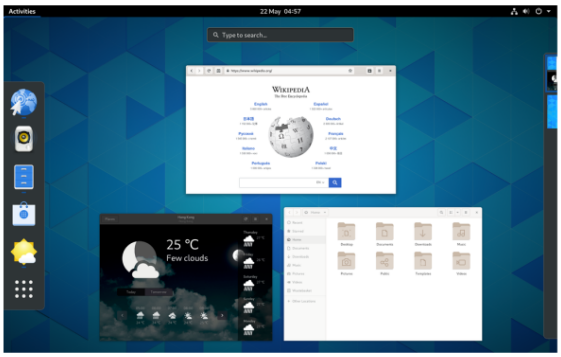
Interacción mediante texto.
Se basa en el uso de un lenguaje codificado.



GUI

Graphical User Interface

Interacción a través de elementos gráficos.
Posibilita una interacción amigable e intuitiva.



1.5. Implementación Monolítica

Wikipedia: Un Kernel **Monolítico*** es una arquitectura de S.O. donde éste en su totalidad trabaja en el espacio del núcleo. Difiere de otras arquitecturas en que solo define **una** interfaz virtual (capa) de alto nivel sobre el Hardware del ordenador.

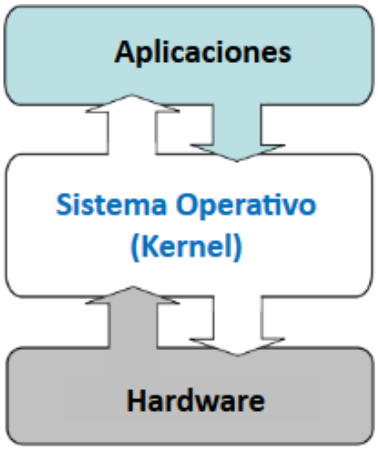
Es un Kernel Monolítico, un conjunto primitivo de llamadas al sistema implementa todos los servicios propios del S.O. tales como la Administración de procesos, Administración de memoria, etc.

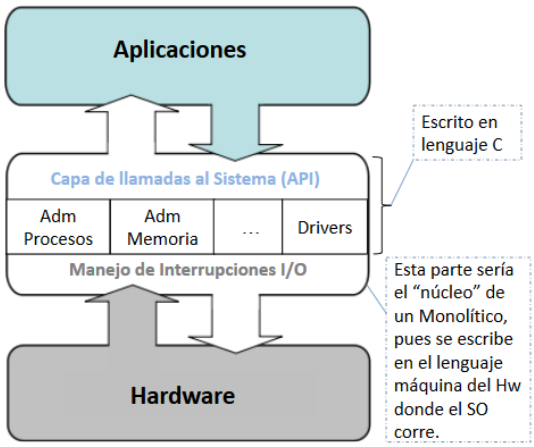
*La palabra "**Monolítico**", según la Real Academia Española, quiere decir "Que está hecho de una sola piedra". Entonces en informática:

Software Monolítico = Software implementado en un solo módulo

La implementación **monolítica** fue la primera en ser usada y aún se sigue usando (Unix, Linux, Windows 9x, Android). Por éste motivo, es muy común llamarle **Kernel** o **TODO** el **Sistema Operativo.**

Así, la visión básica de un S.O. se reduce al diagrama de la derecha, la misma que es comúnmente mostrada en asignaturas de Introducción a la Informática. Los **S.O. Monolíticos** son eficientes y rápidos en su ejecución y gestión, pero carecen de flexibilidad para soportar cambios.





1.6. Apéndice. Términos muy utilizados

- **Boot** = Es el proceso que **inicia** el S.O. cuando se enciende una computadora. Se encarga de la inicialización del sistema y de los dispositivos.
- **Shutdown** = Proceso inverso al boot. El S.O. guarda datos en el Disco y **deja de administrar** a la computadora.
- **Reboot, Restart o "Reiniciar"** = shutdown + boot.
- **Kill** = Terminar un proceso (en el ADM-Tareas, click a `Terminar proceso`)

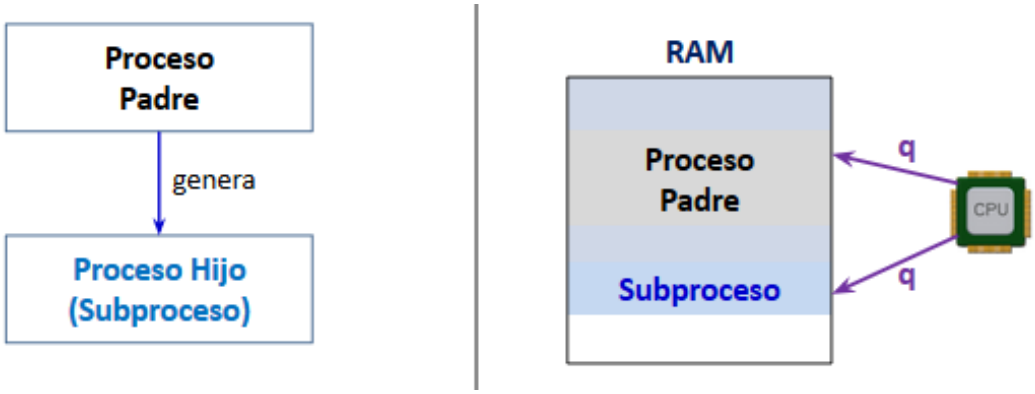
- **Foreground-Process** = Es un Proceso corriendo en **Primer Plano**. Es decir, **un proceso que tiene ventanas** y permite la interacción con el usuario humano.
- **Background-Process** = Es un Proceso corriendo en Segundo Plano o "detrás de escena". Estos procesos **No tienen ventanas** y corren con poca o ninguna interacción con el usuario humano.
- **Daemon o Demonio** = (**D**isk **A**nd **E**xecution **MON**itor) Es un Background-Process que corre continuamente (nunca finaliza por sus propios medios) y verifica la ocurrencia de un evento o condición.

Capítulo 2

Unidad 2. Subprocesos

2.1. Qué es un Subproceso?

Un **Subproceso** es un proceso **generado** por otro proceso.



(Izquierda) Al proceso generador se le llama "**Proceso Padre**" y al proceso generado se le llama "**Proceso Hijo**" o **Subproceso**. (Derecha) Es bueno que un **Subproceso** es también un proceso y por lo tanto tiene su propio espacio de **RAM** y recibe el quantum **q** como cualquier otro proceso.

2.1.1. Cómo se genera un Subproceso?

Usualmente un programador novel no es consciente que su APP está generando **subprocesos**: Algunas líneas de su programa invocan a una **operación** (función/procedimiento) de un ADM y éste último decide que el **código de ésa operación** se trate como un **subproceso**. *Generalmente las operaciones que interactúan con los dispositivos I/O, generan subprocesos.*

Espera (wait) en el Proceso Padre. A veces un subproceso pueden dejar en ***estado de espera*** (wait o bloqueado) al proceso padre, hasta que él complete su ejecución. Es decir, el proceso padre no podrá ejecutar la siguiente instrucción de su código, hasta que el subproceso finalice.

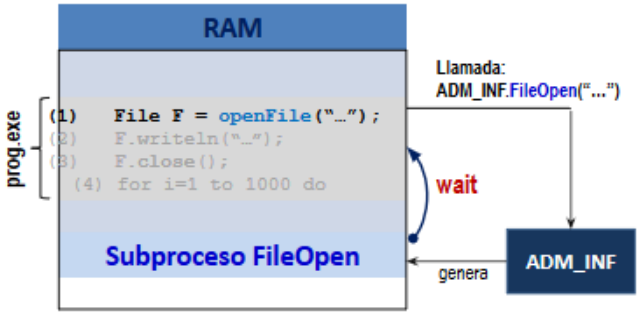
Recuerde: *No todos los subprocesos dejan en espera al proceso padre.*

Por ejemplo: Supongamos que tenemos el siguiente programa (al que llamaremos "**prog**"), el cual escribe un string a un archivo.

```
(1)  File F = openFile("notas.txt");          //Abrir el archivo.
(2)  F.writeln("Hoy aprendí subprocesos");    //Escribir en el archivo.
(3)  F.close();                               //Cerrar el archivo.
  (4)for i=1 to 1000 do                       //Mostrar 1, 2, 3, ..., 1000
     Begin
       showMessage( IntToStr(i) );
     End;
```

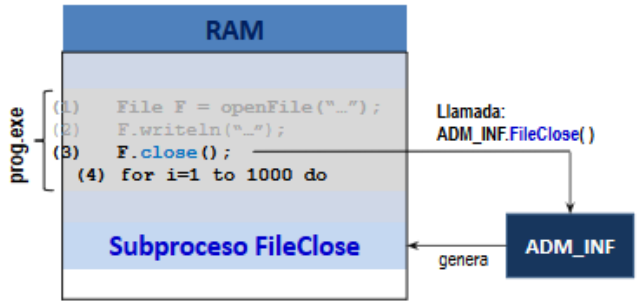
Asumimos que al compilar, generamos el archivo ejecutable (APP): **prog.exe**. Cuando corremos esta APP (proceso), las lineas (1) y (3) hacen sendas llamadas a funciones del **ADM de Información** (ADM-INF), las cuales generan subprocesos.

Al ejecutar la línea (1) se hace una llamada al ADM-INF, el cual genera el subproceso **FileOpen**. Este **subproceso**, dejará en espera (**wait**) al proceso padre **prog.exe**, porque no es posible escribir (writeLn) en un archivo si éste no ha sido abierto aún.



Luego que el subproceso ***FileOpen*** *finalice, entonces el proceso padre* ***prog.exe****, ejecutará la línea (2) y luego la linea (3).*

Similarmente, la línea (3), generará un subproceso **FileClose**, pero éste **no** dejará en espera al proceso padre. Así, mientras **FileClose** se encarga de cerrar el archivo, **prog.exe**, continuará con la línea (4) ...



2.1.2. Problema

Seleccione la respuesta correcta, a los asertos (a) y (b).

(a) Todo **archivo ejecutable*** genera un proceso al correr.

- ◯ Si
- ◯ No
- ◯ No necesariamente

(b) Todo proceso proviene de un **archivo ejecutable**.

- ◯ Si
- ◯ No
- ◯ No necesariamente

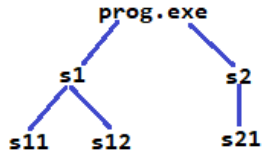
*Un **archivo ejecutable** es mas conocido como **App** (Application o programa). En Windows, éstos archivos usan la extensión **.exe**

Respuesta (a): **Si**. Todo archivo ejecutable (App, .exe) SIEMPRE genera un **proceso** al correr. Precisamente, ésa es la definición más elemental de un proceso: "*Un proceso es un programa (App) en ejecución*"

Respuesta (b): **No necesariamente.** Recordemos que los **subprocesos** son también **procesos** y ellos NO provienen de un archivo ejecutable, sino que son *generados* por otro proceso. (e.g. Se puede observar que los (sub)procesos "**FileOpen**" y "**FileClose**" NO provienen de un archivo ejecutable).

2.1.3. Árbol de Procesos

Puesto que un subproceso es también un proceso, éste a su vez puede generar subprocesos, generando así un árbol de (sub)procesos.



En la gráfica, por ejemplo, se observa el árbol de procesos generado por el proceso padre **prog.exe**. El proceso padre genera los subprocesos **s1** y **s2**. A su vez, **s1** genera los subprocesos **s11** y **s12**, mientras que **s2** genera el subproceso **s21**.

2.2. Definición de Proceso

Considerando las respuestas del Problema 1, nos damos cuenta que la definición de proceso, debe tomar en cuenta a los subprocesos, pues ellos también son procesos:

Wikipedia: "Un **proceso** es una unidad de actividad que se caracteriza por la ejecución de una **secuencia de instrucciones**, un conjunto asociado de recursos del sistema, y **un estado actual**".

Otra definición de proceso más escueta es: "*Un **proceso** es un código que corre (run) independiente de otro código*"

Desarrollando la definición de **proceso**:

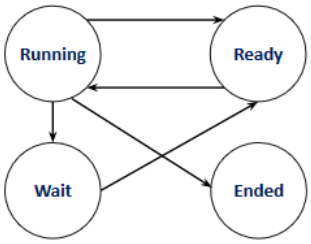
- Un **proceso es la ejecución de una secuencia de instrucciones**. Obviamente, porque un proceso es básicamente un algoritmo o programa, ejecutándose. (Recordemos que un algoritmo o programa es una secuencia de instrucciones o pasos.)
- Un **proceso tiene asociado un conjunto de recursos del sistema**. Todo proceso necesita recursos para correr. El primer recurso importante que se le asocia es el espacio de R.A.M. donde el proceso está alojado. El segundo recurso en importancia, es el quantum (ejecución de las instrucciones del proceso, por parte de la C.P.U.).
- Un **proceso puede estar en un estado**. Los estados de un proceso son 4 (Running, Ready, Wait y Ended)

2.3. Los Estados de un Proceso

1) **Running. Corriendo.** La C.P.U. está ejecutando el código del proceso. Es decir, el proceso está haciendo uso del quantum.
2) **Ready. Listo o Preparado.** El proceso no tiene impedimentos para correr, pero la C.P.U. está corriendo el código de otro proceso. En otras palabras, el proceso no esta haciendo uso del quantum, sino que está aguardando por él.
3) **Wait. Espera o bloqueado.** El proceso **tiene impedimentos** para correr. Esta esperando a que uno de sus subprocesos finalice.
4) **Ended. Finalizado.** El proceso ha terminado su ejecución. Cuando esto ocurre, el proceso es desalojado de la R.A.M. (FreeMem) y todos los demás recursos que él utilizó le son devueltos al S.O.

2.3.1. Grafo de Estados

El **Grafo de Estados** de **un** proceso, es una representación gráfica de las transiciones (cambio de estado) que un proceso experimenta o podrá experimentar durante su permanencia en el sistema. Los vértices representan a los Estados y las aristas (flechas), las transiciones.



Grafo de Estados General

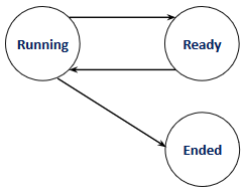
En la gráfica, se muestra el Grafo de Estados de cualquier proceso. **Nota.** En el Grafo de Estados no se toma en cuenta a la orden Kill (terminar proceso) dada por el usuario.

- La transición **Running→Ready**, se da cuando el proceso deja de ser atendido por la C.P.U. es decir, cuando el proceso acaba de agotar su quantum.
- La transición **Ready→Running**, es experimentada por el proceso cuando recupera el quantum (i.e. la C.P.U. vuelve a ejecutar su código).
- El cambio de estado **Running→Wait**, lo hace el proceso, cuando genera un subproceso y éste último lo deja en Espera (puede rever este concepto, trasladándose a la **Sección 2.1.1.**)
- Cuando está en Wait, el proceso es ignorado por la C.P.U. porque es depositado en otra Estructura de Datos (E.D.), diferente a la de los procesos que están en Ready. Cuando el subproceso que lo mantiene en Wait finaliza, el S.O. traslada el proceso a la E.D. donde están los procesos en Ready. Por éste motivo, observamos: **Wait→Ready**
- Cuando la C.P.U. ejecuta la "última linea" del proceso (o sea el proceso aún está en running), el S.O. finaliza al proceso: **Running→Ended**

2.3.2. Problemas

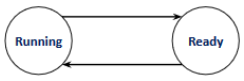
Problema 2 Un tipo de procesos, al que llamaremos ACME, son procesos que nunca generan subprocesos. Dibuje el Grafo de Estados de un proceso ACME.

Solución: Puesto que un proceso ACME no genera subprocesos, entonces nunca entrará en Espera (Wait). Por lo tanto, del Grafo de Estados General, sacamos este estado y tenemos la solución.



Problema 3 Dibuje un Grafo de Estados para un proceso P, el cual no genera subprocesos y ***nunca finaliza por sus propios medios.***

Solución: Quitamos el estado Wait, porque P no genera subprocesos. Decir que un proceso ***nunca finaliza por sus propios medios***, es afirmar que **corre indefinidamente** (i.e. el código del proceso, no tiene una instrucción o sentencia de finalización). Por tanto, la transición Running→Ended **No** se da. Dado que el estado Ended no tiene aristas, podemos quitarlo del Grafo, y así obtenemos la solución:



Problema 4 Dibuje un Grafo de Estados para un proceso P, tomando en cuenta que:

1) No genera subprocesos, pero el S.O. lo pone en Espera (Wait) cada 10 minutos.
2) Si P está en Espera, el S.O. lo saca de este estado luego de 20 minutos.

3) El proceso P no finaliza por sus propios medios, pero el S.O. finaliza a P, luego que éste ha estado corriendo por más de 24 horas.

Solución El Grafo de Estados de respuesta, lo obtendremos solucionando los puntos 1), 2) y 3) dados en el enunciado.

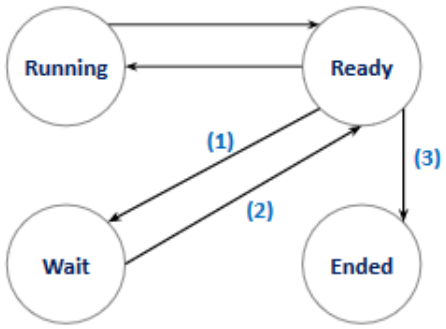
1) La transición de Running a Wait **No** puede darse, porque ninguna linea de código del proceso P genera subprocesos. Es decir, teniendo **el proceso** (el quantum de) **la C.P.U.***, no se cambia al estado Wait.

"El S.O. lo pone en Espera (Wait) cada 10 minutos":** Esto significa que unas **lineas de código del S.O. son ejecutadas por la C.P.U.** y éstas lineas ponen en Espera a P. Por tanto, obviamente, si **la C.P.U. está atendiendo al S.O.**, el proceso P no tiene a la C.P.U. Es decir, el proceso P está en Ready. Asi, obtenemos la transición **Ready→Wait**.

* **El proceso tiene la C.P.U. = el proceso esta in Running**
** Los tiempos que se mencionan en este tipo de ejercicios son solo nominales. Lo importante, para la confección del Grafo, es si el estado se da o se puede dar.

2) **"El S.O. lo saca de la Espera, luego de 20 minutos".** SIEMPRE es el S.O. el que saca de Espera a un proceso. Y como se vio antes, el proceso es trasladado a la E.D. de los Ready's, es decir, pasa de **Wait→Ready**.

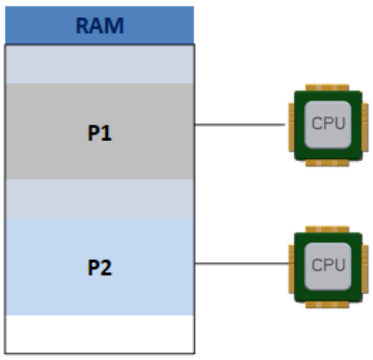
3 Como es el S.O. el que finaliza a P, el S.O. tiene a la C.P.U. cuando ocurre esto y, por ende, el proceso P esta en Ready. Así, obtenemos la transición: **Ready→Ended**.



2.4. Apéndice. Tipos de Procesamiento

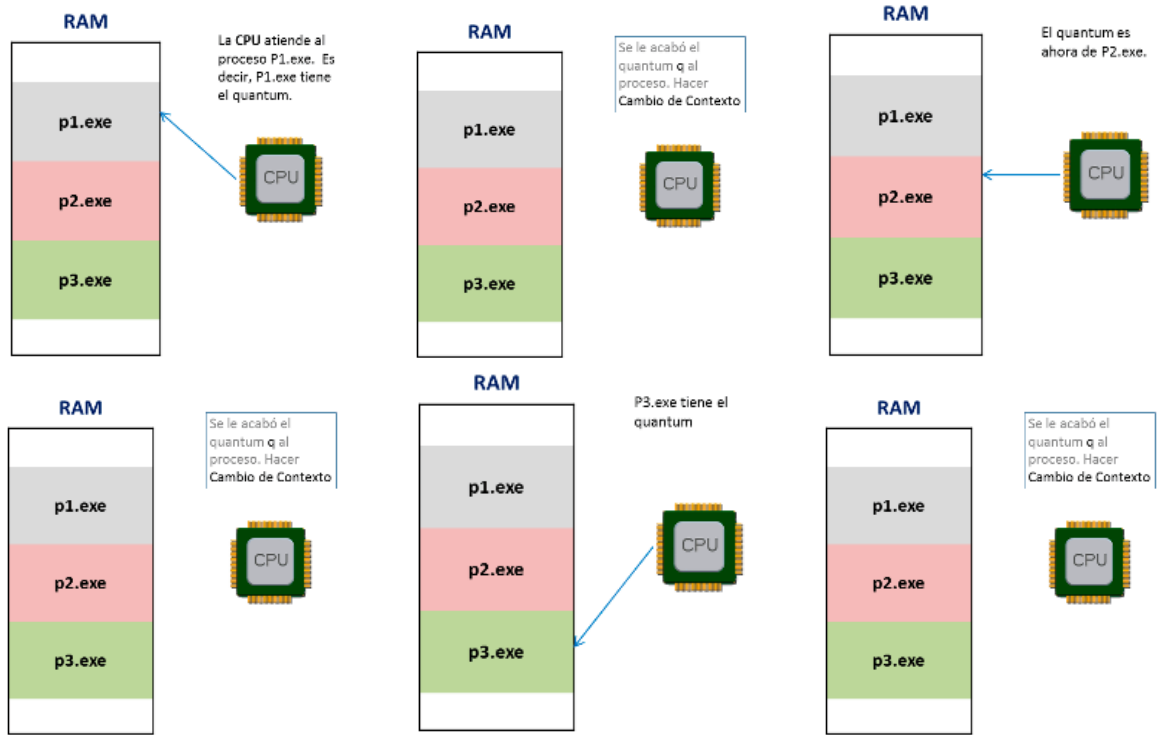
Procesamiento Paralelo Implica que dos o más procesos pueden ser ejecutados efectivamente en distintos procesadores **al mismo tiempo**. Así, si se tienen n procesos, se precisaran n C.P.U's que los ejecuten simultáneamente (una C.P.U. por proceso).

Por ejemplo, en la gráfica, se observa a los procesos P1 y P2 corriendo en paralelo. Para lograr esto, se precisan de dos C.P.U.'s: Una C.P.U. que ejecute el código de P1 y otra para el código de P2.



Procesamiento Concurrente. Consiste en que **una C.P.U.** alterna la ejecución de los procesos en porciones fijas de tiempo q (**quantum**). A esta técnica se le llama **Time-Sharing** o **Tiempo Compartido.**

Por ejemplo, se observa a los procesos **p1.exe**,**p2.exe** y **p3.exe** corriendo concurrentemente. **Nota:** En muchos S.O., el **quantum** es de un milisegundo. (Recuerda: Concurrente = "en el mismo periodo de tiempo")



Procesamiento en Tiempo Real. Se puede tener una noción rápida de éste concepto si se traduce la frase "**en Tiempo Real**", con el término televisivo "**al vivo**".

Wikipedia: *Procesamiento Real* es aquel que **interacciona con su entorno físico** y responde a los estímulos del entorno dentro de un **plazo de tiempo determinado**

Capítulo 3

Unidad 3. Aspectos Tecnicos del Hw

3.1. Como funciona una computadora?

Voy a tener una memoria (RAM) donde vamos a escribir instrucciones, y vamos a tener un cerebro (CPU) lee las instrucciones y las ejecuta.

3.1.1. Conoces una maquina de Von Nuemman?

Es la que tenemos día a día, una computadora actual.

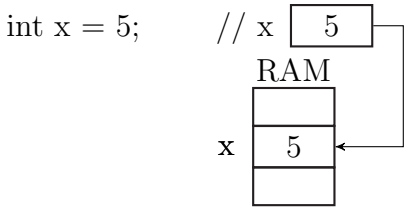
3.1.2. Diferencia entre digital y analógico?

Si es digital se representa por números (Binarios), un archivo Digital es: mp3, jpg, mp4, FLAC..., si es Analógico significaría sin lógica, se basa en marcas que solo entiende el hardware que lo grabo mediante símbolos aleatorios.

Una computadora es RAM y CPU, i.e. no pueden faltar ni la RAM ni la CPU.
i.e = ist est = es decir = esto es.

3.2. Registro de un Hw

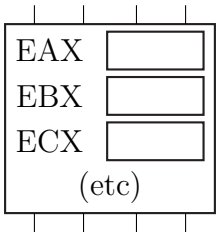
Una Variable de un programa, es un espacio de RAM, reservada para ella. **Por ejemplo.-**



Muchos Hw's traen su propia memoria.
Por ejemplo: La CPU, la impresora, la tarjeta de vídeo.

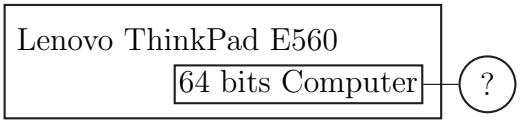
Un registro de un Hw es una 'variable' que ocupa la memoria de ese Hw.
Por ejemplo En un manual leímos:
'El registro PageCounter de la impresora al menos la...'
Como PageCounter es un registro, este se encuentra en la memoria de la impresora (no en la RAM)
Diferencia entre variable y registro: ambos almacenan datos, la diferencia es donde están, las variables en la RAM y los registro en el Hardware.

No todos los dispositivos tienen registros (i.e. no traen memoria), pero alguno de ellos son capaces de share ('compartir','usar') la RAM, por ejemplo: el mouse y el teclado, no traen memoria (no tienen registro), pero son capaces de usar áreas de la RAM, para guardar sus datos.
También la CPU tiene sus propios registros, que tienen nombres muy conocidos e inventados por Intel (EAX, EBX, ECX,...)



3.3. n-bit Computer

En la caja de una Computadora se lee:

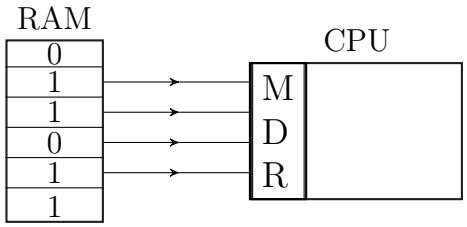


3.3.1. Que quiere decir '64-bit Computer'?

Quiere decir, que esta computadora tiene un bus de datos de 64-bit. Es decir, Es capaz de procesar 64-bit de una sola vez.

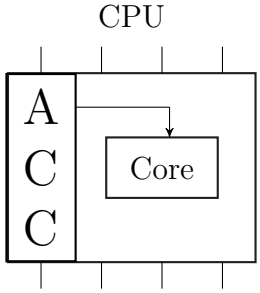
3.3.2. Que es El Bus de Datos?

Es un bus en paralelo que transporta bits desde la RAM hacia la CPU y viceversa. Define la potencia del RAM. Ejemplo: Bus de datos de 4 bits.



Bus paralelo de 4 bits = 4 hilos (cables).
En cada ráfaga (burst) de bus viajan los bits que son transportados desde/hacia la RAM. Cuando viajan hacia la CPU, los bits son almacenados en el registro (espacio de memoria) MDR o ACC (Acumulador). Obviamente el tamaño del MDR es igual al tamaño del bus de datos. En este ejemplo, el bus de datos es de 4-bit y por tanto el MDR sera de 4-bit.
Un Bus en Serie es mas lento que en Paralelo.

Una vez el MDR recoje la instrucción que le envió el bus de datos, toma esos datos y las redirije al Core (o núcleo de la microprogramación)



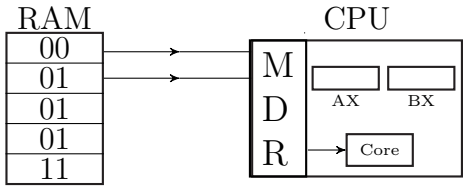
3.4. n-bit code

Al ver las propiedades de una aplicación, leemos que dice 'aplicación de 32-bits'.

3.4.1. Que quiere decir esto?

Quiere decir que se esta usando el juego de instrucciones de 32-bits del procesador. En otras palabras, el procesador puede ejecutar operaciones (aritméticas, lógicas, otros) hasta un máximo de n-bits, en otras palabras, cada instrucción de la CPU a lo sumo, ocupan n-bits.
Por ejemplo: Un 2 bit-code. La CPU maneja instrucciones de 2 bits

$$00 \rightarrow \text{Ax} = 3$$
$$01 \rightarrow \text{Ax++}$$
$$10 \rightarrow \text{Bx} = \text{Ax-Bx}$$
$$11 \rightarrow \text{Ax} = \text{Ax+Bx}$$

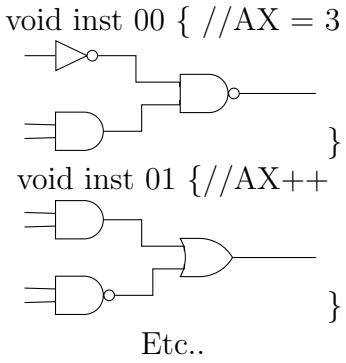


Ax y Bx son registros de este CPU.
Empieza 00 y entra al MDR se lo pasa al Core y hace la micro-programación
Luego 01 entra al MDR pasa al Core, y lo pone en AX la instrucción

El hardware no puede ejecutar los programas comparativos en este 2 bit-code, porque no existen dichas instrucciones.

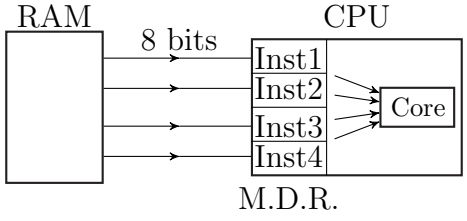
3.4.2. Que hay en el Core?

El Core de una CPU, puede verse como una CPU en si misma, pues implementa los registro ALU, etc. también el core implementa los microprogramas del procesador, Un micro programa es como un "procedimiento" escrito como un circuito de compuertas Están implementadas las instrucciones, como 'procedimientos' pero a nivel electrónico. Un 'procedimiento' del Core se llama micro-programas, en las siguientes figuras simulamos las instrucciones anteriores y su 'representación' a nivel electrónico



A este CPU imaginario de 2-bit Code (Sw) lo colocamos en una 8-bit Computer (Hw)

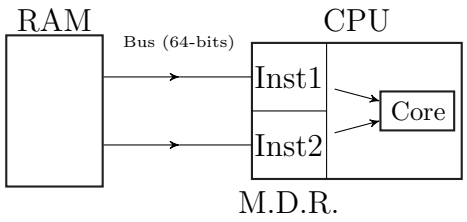
Como funciona todo esto?



En una ráfaga del bus esta computadora envía 4 instrucciones a la CPU. La CPU procesa una a una estas instrucciones.

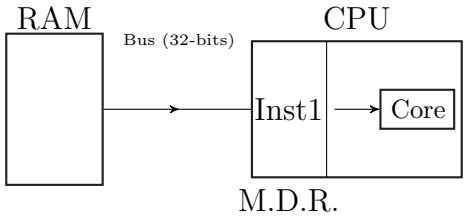
(*) Explique lo que pasa.

Se tiene una APP de 32-bit (code) por esta corriente en una computadora de 64-bit (64-bit computer).
Respuesta: En cada ráfaga del bus le llegan 2 instrucciones a la vez.



(*) Explique lo que pasa.

Se tiene una APP de 32-bit (code) por esta corriente en una computadora de 32-bit (32-bit computer).
Respuesta: En cada ráfaga del bus le llegan 1 instrucciones a la vez



En cual de estas computadoras corre mas rapido la app de 32-bits? Porque?

En la de 64-bit Computer. Porque en una sola ráfaga de bus le llega al MDR dos instrucciones. En cambio, en la computadora de 32-bits le llega una sola instrucción por cada ráfaga de bus.

(*) Observe

8-bits Computer y APP's de 16-bits (code).
Esta situación se dio con el procesador Intel 8088 (la que dio origen a la PC). La CPU necesitaba 2 ráfagas de bus para completar una instrucción en el MDR y luego pasarla al Core.

Un S.O. es un conjunto de Programas (Code) que administra...

Windows 32-bits = Cada uno de los programas de Windows esta codificado a 32-bits

3.4.3. La familia x86

8086 = 16-bit Computer (16-bit code), 80186, 80286, 80386, 80486
Pentium^R = Patentado para uso legal, Pentium II, ..., Pentium IV
Core x-Duo...
Core i3, i5, i7, i9

Sera posible tener una 8-bit computer usando 16-bit code?

Respuesta: Si, Realmente esto se dio con el Intel 8088.

Su MDR era de 16-bit y por lo tanto se necesitaba 2 ráfagas del bus de datos para llenarlo. Una vez lleno, pasaba la instrucción al Core (núcleo)

Los Sistemas Operativos para la linea x86 se desarrollaron hasta 32-bits (Code), por este motivo. x86 significa hasta 32-bits (Code)

los registros tienen que ser capaces de soportar el bit code

3.4.4. El Controlador de un Hardware

Es el chip 'jefe', encargado de controlar el dispositivo y de comunicarse con la CPU

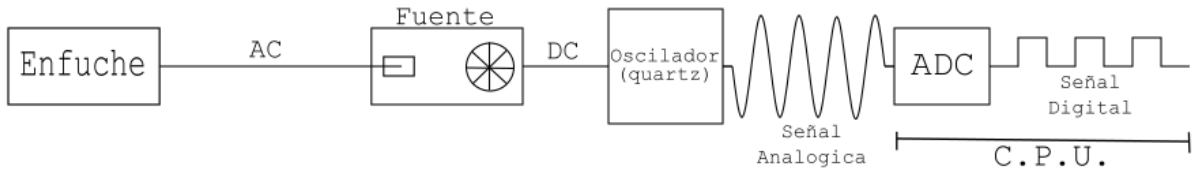
Usualmente el controlador tiene un bajo relieve o en pintura (troquelado) y esta la marca del fabricante del dispositivo

(*) Cuantos USB tienen la computadora?

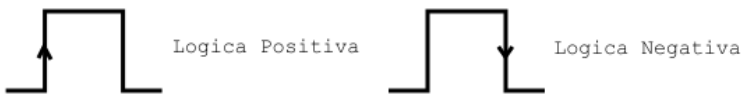
La respuesta es 1, y se llama Controlador-USB, en realidad el USB es el Bus administrado por un controlador.

El controlador-USB recibe la señal del dispositivo (bytes) y el controlador los redirige al puerto del dispositivo

3.5. La velocidad de la CPU



A cada '⎍' se le llama ciclo. Sin embargo, los circuitos digitales de la CPU solo funcionan cuando la señal sube o baja



1 Hertz = 1 Hz = 1 $\frac{ciclo}{seg}$ = 1 $\frac{\sqcap\!\sqcup}{seg}$

Sin importar si la lógica es positiva o negativa, deducimos que por cada siglo ($\sqcap\!\sqcup$) obtenemos corriente para los circuitos. Por cada $\sqcap\!\sqcup$ las maquinas trabajan. Los Hertz en el sistema M.K.S., también se agrupan en múltiplos de 10^3:

$$1KHz = 10^3Hz$$

$$1MHz = 10^6Hz$$

$$1GHz = 10^9Hz$$

$$1THz = 10^{12}Hz$$

Supongase que se tiene dos robots albañiles, los cuales son capaces de poner un ladrillo en la pared, cada vez que le damos corriente (osea, cada vez que el robot recibe un $\sqcap\!\sqcup$).

Ahora conectamos el robot A a un oscilador de 4Hz y al robot B a un oscilador de 3Hz. Quien ganara? el robot A, no es mas rápido que B sino es que recibe mas corriente o Hertz para realizar sus programas.

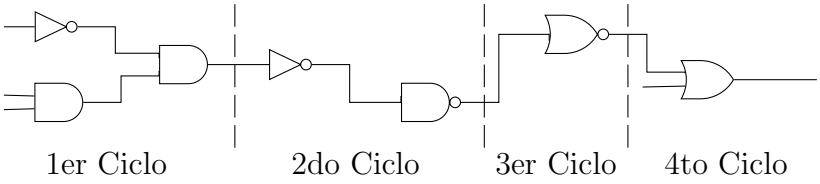
Como se vio en este ejemplo, puede ser que dos CPU's trabajen a la misma velocidad, cuando se le da un ciclo, pero la CPU que tenga un oscilador con mayor frecuencia (es decir mas Hertz), completara mas rápido los programas.

3.5.1. Ley ('Efecto') de Joule

A mayor oscilación (osea mas $\sqcap\!\sqcup$) mayor calor generamos

3.5.2. Ciclos de Máquinas

Es la cantidad de ciclos ($\sqcap\!\sqcup$) que necesita un micro-programa (instrucción que esta dentro Core) para llevar a cabo su operación, por ejemplo, la instrucción MUL necesita 4 ciclos de maquina.



- Se tiene el siguiente programa ASM

MOV AX, 3 // 1 $\sqcap\!\sqcup$
MUL BX // 4 $\sqcap\!\sqcup$
SUB AX,BX // 2 $\sqcap\!\sqcup$

(a) En que tiempo una CPU∼10Hz hace este programa? (Figura en el cuaderno)

El programa necesita 7 ciclos
La CPU genera $10\frac{ciclos}{seg}$
Cuanto tarda?

$$\frac{10\sqcap\!\!\llcorner}{1seg} = \frac{7\sqcap\!\!\llcorner}{xseg} \rightarrow x = 7 \times \frac{1}{10} seg \rightarrow x = 0{,}7seg$$

(b) Cuanto tarda el programa con una CPU∼15 Hz?

El programa necesita 7 ciclos.
Entonces, el programa tarda

$$\frac{15\sqcap\!\!\llcorner}{1seg} = \frac{7\sqcap\!\!\llcorner}{xseg} \rightarrow x = 7 \times \frac{1}{15} seg \rightarrow x = 0{,}47seg$$

3.6. Las rutinas POST

//POST = Power-On, Self-Test

Que reside en la ROM.
/*(rutinas) BIOS de Base Input/Output System. En la ROM-BIOS encontramos:

- La rutina POST.
- Interrupciones de Hw Basicas.
- SETUP.

*/

La rutina POST hace la sgte:

- Muestra el logo del que hizo la ROM o de la empresa que lo encargo
- Luego de un proceso al teclado de la RAM, procede a testear la RAM. De las pastillas validas (sin defectos), la RAM sabe cuanta memoria disponible tenemos
- Luego, envía un bit de 'saludo' a cada uno de los puertos físicos de la computadora, con la intensión de comunicarse con el controlador del dispositivo que esta conectado (si lo hay) Si recibe el ACK del dispositivo, la ROM sabe que hay en dispositivos en ese puerto

  /* Muchos dispositivos le intentan comunicar al humano, Stand on-line con la CPU.
  Por disco, el teclado hace un parpadeo de sus LED's. El lector de CD, enciende su LED al igual que los discos duros. La impresora hace un movimiento del carro o emite un pitido*/

- Cuando se copia a la RAM las rutinas de introducción de Hw básicas

- **Recordar siempre:** Finalmente, la rutina POST busca un almacenamiento de memoria secundaria (Disco duro, CD, Flash,...) en cuyo sector 0 este en una marca llamada "Marca de Boot".
  Si tal marca existe la rutina POST carga los siguientes 256 bytes y le cede el control a ese programa
  //Ceder el Control = Que ese programa haga lo que quiera con esa computadora

  /* Si la rutina POST no encuentra la "Marca de Boot" en la memoria secundaria emite un mensaje "Insert System Disk..."*/

  Si un disco, CD, flash tiene la "Marca de Boot", se dice que este dispositivo es Booteable.

Que es program-boot o boot-Program?
Es el programa de arranque de un S.O.

/* la palabra 'boot-strapping', *en un cuento norteamericano, el héroe de la historia podía elevarse desde el suelo tirando de las trenzas de sus botas*, boot-strapping es levantarse de las trenzas, boot = levantarse solo (Informática)*/

El programa de arranque, como es muy pequeño carga a la R.A.M. otro programa y le cede el control. Este programa a su vez carga otro y le cede el control... y así sucesivamente.(Figura en el celu), El sistema operativo se levanta solo, cuando se enciende pasa a la rutina POST cede el control y se carga, En el '88 se hizo el primer virus electronico (c) Brain

Capítulo 4

Unidad 4. Administrador de Procesos

Este administrador (al que abreviaremos ADM-PROC), es el encargado de dar la CPU a cada uno de los Programas que están cargados en la R.A.M., Recordemos que para la CPU toda la R.A.M. es un solo programa. Por este motivo se crea el concepto de proceso.(Figrua en el celu)
/* Un SO es monoproceso si corre un proceso a la vez */
Es el modulo del SO que se encarga de asignar la(s) CPU(s) a los procesos. En otras palabras, es el que implementa el Multiproceso.

4.1. Partes de un Administrador de Procesos

Teóricamente, un administrador de procesos (ADM-PROC) se divide en dos partes

4.1.1. El Planificador (scheduler)

Es el que decide, que proceso recibirá el próximo quantum (q).

4.1.2. El despachador (dispatcher)

Es el encargado de realizar un efectivo cambio de contexto (context-switch, Cambio de proceso) al proceso que escogió el planificador

Aunque en teoría, este administrador, se ve como 2, en realidad se implementan en 1

```
void Planificador(){
    //Planificador
          +
    //Despacho
}
```

PIC = Controlador Programable de Interrupciones
Cuando ocurre un evento o un suceso I/O, este emite un I.R.Q. (Interrupt ReQuest) Al PIC.

El PIC es como una secretaria de la C.P.U. que le asigna números a cada uno de los dispositivos (el # de I.R.Q.)
Así:

- El Teclado es el dispositivo 1 (i.e. El teclado emite el I.R.Q.)
- El Timer (Cronómetro del sistema) es el 0
- El Mouse es el 12

El P.I.C. es necesario porque simultaneamente porque dos o mas dispositivos pueden emitir un I.R.Q. También el P.I.C. asocia el I.R.Q. con el numero de interrupciones
Por Ejemplo
Teclado I.R.Q. → INT 9

4.2. El Timer o Cronómetro del sistema

El chip 8243 o Timer es capaz de ejecutar la INT 8 cada milisegundo pues el emite un I.R.Q. 0 cada milisegundo. Este tipo sin que nadie lo moleste, le molesta a la C.P.U cada milisegundo, la INT 8 es puesta por el BIOS para actualizar el reloj de software.
La INT 8, el sistema operativo llama al planificador. *Cada vez que se cumpla el quantum (q) se ejecuta el planificador*. Imagen en el teléfono(4/9/19),. El planificador es un "hackeador"porque no hace que vuelva a la CPU ya que puede controlar su IP.

4.3. El Bloque de Control del Proceso (P.C.B)

Un bloque, es un área de memoria (de la RAM o de la secundaria), donde se almacena alguna información. Usualmente, un block esta compuesto por uno o mas campos (bytes).
En particular, A cada proceso se le asigna un bloque de información llamado P.C.B. (Process Control Block).

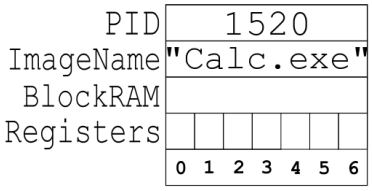
Recuerde: **Un Proceso tiene un y solo un P.C.B.**

Si un "proceso" no tiene P.C.B., la C.P.U. jamas ejecutara su código. Un P.C.B. almacena la información que necesita el proceso para correr. Muchos campos del P.C.B., son usado por el S.O., para dar información al User. Cada Sistema Operativo define un P.C.B. distinto (i.e. el P.C.B. de Windows difiere del P.C.B. de Linux) pero coinciden en al menos estos cuatro campos: P.I.D, ImageName, BlockRAM, Registers[ ]. Como programadores podemos ver al P.C.B. como:

- un RECORD (Delphi)
- una Struct (C, C++)
- una Class con todos los campos public y sin métodos (Java, C#)

```
Class P.C.B.{
    int P.I.D.:
    String ImageName;
    AlgunTipo BlockR.A.M.;
    int Registers[ ];
}
```

Gráficamente se vería así:

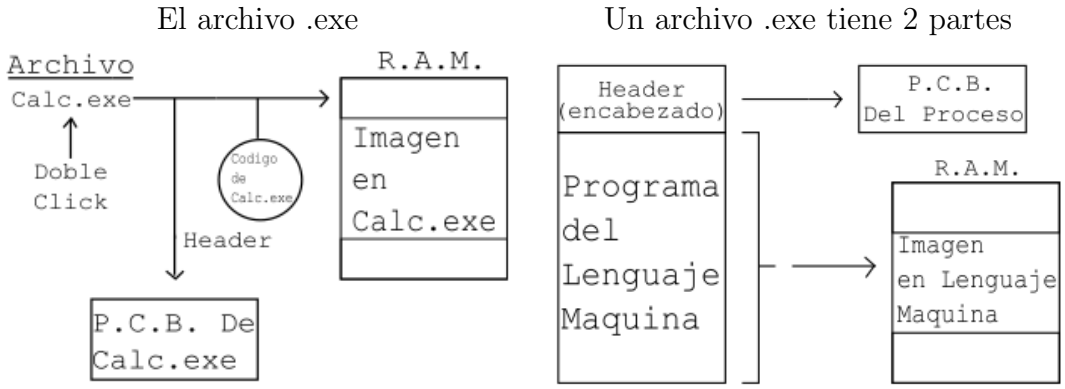


4.3.1. Identificador del Proceso (P.I.D.)

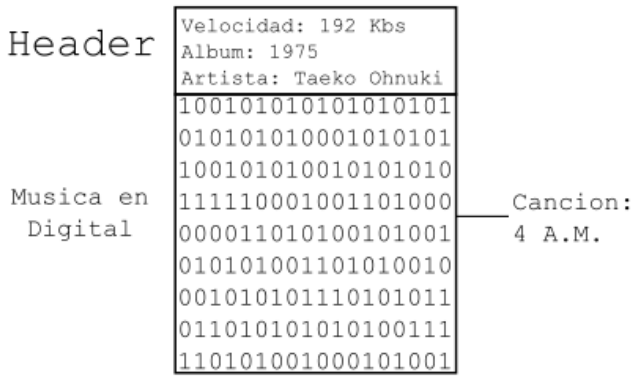
Es un numero que identifica en forma única al proceso. Es equivalente al numero de registro de un estudiante.

4.3.2. ImageName (Nombre de Imagen)

Básicamente, es el nombre que el Sistema Operativo al espacio de RAM que ocupa el código del proceso, le asigna al proceso cuando esta cargado en la RAM. Por razones didácticas el S.O. le da un nombre de imagen igual al nombre del archivo ejecutable (Pueden haber tres paginas word pero solo un proceso esta corriendo gracias a la imagen)



Los archivos .mp3 tienen un Header



4.3.3. El campo de Block-R.A.M.

En este campo se almacena el o los bloques de R.A.M. que de imagen ocupan el proceso, por ejemplo: Si se usa asignación continua

Block-R.A.M. = (Dir. inicio, Tamaño)

Subimos (upload) el programa (APP) calc.exe el archivo calc.exe se convierte en un proceso (Figura en el teléfono 6/9/19)

4.3.4. El campo Registers [ ]

Es un Vector donde cada casilla esta asociada a un registro de la C.P.U., es decir, si la C.P.U. tiene 'n' registros, este vector tendrá 'n' casillas. Este campo del P.C.B. es de uso exclusivo del ADM de procesos, mas específicamente por del void Planificar().

Este array (vector) mantiene una capa de los registros de la C.P.U., por ejemplo, supongamos que la C.P.U. tiene 3 registros (AX, BX, IP (asigna a la C.P.U. la siguiente instrucción)). Por lo tanto, el Campo Registers[] tendrá 3 casillas (Figura en el telefono 6/9/19)

4.3.5. El ciclo "fetch"de la C.P.U.

fetch = Buscar y traer.
El ciclo de la C.P.U., entre otras cosas básicamente hace esto:

- Trae la instrucción "apuntado" por el registro I.P.(Instrucción Pointer).
- Ejecutar esa instrucción en el Core.
- Incrementar el I.P. a la siguiente instrucción.

Puerto = Área de memoria (Figura en el celu)

Se crea una tecnología del proceso "tiene" (porque los procesos no lo tienen) registros."El IP del proceso P1.exe" se refiere a la copia del IP de la CPU, en el campo Registers de la CPU del proceso P1.exe (Imagen en el celu 9/9/19). Puede que el programa no sea sirva como debe, Porque? Porque se debe guardar todos los registros en el P.C.B antes de pasar el IP a otro P.C.B.

Como se genera el quantum (q)?

Multiprocesos tiempo compartido (time share), usando interrupciones en Hw. (Figura en el celu.)

El P.I.C. enumera los dispositivos por números: 0 = TIMER, 1 = Teclado, ..., 12 = Mouse.

Cuando ocurre un evento (suceso)?

I.R.Q. = Interrupt ReQuest (Pedido de Interrupción)
P.I.C. = Programmable Interrupt Controller (Controlador Programable de Interrupciones)

El dispositivos emite un I.R.Q. al P.I.C., el P.I.C. evalúa el pedido y luego interrumpe a la C.P.U. para que ejecute 1 rutina (procedures) asociada al nro de interrupciones. Luego de ejecutar esa rutina, la C.P.U. vuelve donde estaba.

Como se genera un Quantum?

En la tarjeta madre, se encuentra conectado el chip I.C.8253, mas conocido como Timer (cronometro)(del sistema), el cual emite un IRQ cada milisegundo.

La interrupción 8, es la asociada al Timer. En esta rutina (procedimiento), el S.O. hace algunas cosas y también ejecuta el planificador.

Entonces, cada milisegundo, el Timer interrumpe a la C.P.U. para que este ejecute el planificador, es decir, este es el quantum que se menciona en los libros técnicos.

Recuerde: Cada vez que se cumple el quantum (q) se ejecuta el planificador (rutina) (esto no se puede cambiar)

Registers: Se copia los registros de la C.P.U.
El campo Registers[ ] del P.C.B, guarda los valores de los registros de la C.P.U., cuando el proceso pierde el quantum; y copia estos valores, cuando el proceso recupera el quantum.

- Pierde el quantum = Pierde la C.P.U. = La C.P.U. deja de atenderlo.
- Tener el quantum = Tener la C.P.U. = La C.P.U. ejecuta el código del proceso.

(Figura en el Celu)

4.3.6. Pseudo - Codigo para el Cambio de Contexto

Podemos guardar los registros de la C.P.U. en el campo Register.
PCB.Registers ← CPU.Registers; //Save Context
Para asignar el campo Registro a los registros de la C.P.U.:
CPU.Registers ← PCB.Registers; se le llama "cambio de Contexto" o "dispatch P.C.B."

4.4. Esquema general de un Planificador

Un planificador, básicamente, tiene 2 variables (globales) que usa:

- Una estructura de Datos, donde almacena los PCB's que van a recibir la C.P.U. que están en Ready (aguardando el quantum)

- Una variable tipo PCB:

PCB PRUN;

en Delphi

VAR PRUN: PCB;

La cual guarda (apunta) al PCB del proceso que está corriendo (i.e.Que tiene la CPU)
//PRUN = Process in RUNning
//ED = Almacena los PCB's de los procesos que están en Ready.

Por ejemplo: (Figura en el celu 9/9/19)
El algoritmo de planificación tiene dos partes:

Que hacer con el Proceso (PRUN) que acaba de perder el q?

El proceso puede querer finalizar. En este caso lo desalojamos de la RAM y recuperamos los recursos que el proceso utilizo (Figura en el cuaderno)

```
void Planificador(){
//i) una banderita
if(Finalizo(PRUN))
        FreeMen(PRUN); //Sacar al Procesos de la RAM
else{ //Depositar PRUN a la E.D.
        PRUN.Registers ← CPU.Registers;
        ED.Meter(PRUN);
}
//ii) "Buscar" al nuevo PRUN
PRUN = ED.sacar();
CPU.Registers ← PRUN.Registers;
}
```

Recuerde: Antes de introducir el PCB a la ED, guarde los registros de a CPU en el PCB

PRUN.Registers ← CPU.Registers;
ED.Meter(PRUN);

Si el proceso no quiere finalizar, lo depositamos en la estructura de datos (Figura en el cuaderno)

Buscar al "nuevo"PRUN, desde la ED

(figura en el cuaderno)
/* En el S.O. Windows, Cuando un proceso quiere finalizar, utiliza la interrupción de Sw:

INT 20h

la cual pone en true una bandera que es la que lee la función

boolean Finalizo(PCB PRUN)

*/
(Figura en el cuaderno)

Estrategias de Planificación

Durante décadas se han propuesto diferentes estrategias de planificación.

- Planificador Round-Robin (RR)
- Planificador con colas de Prioridad
- Planificador por Envejecimiento (El mejor de todos)
- Planificación por sorteo (nueva)

4.5. Planificador RR (Round-Robin)

La E.D. (Estructura de Datos), que utiliza es una cola de P.B.C.'s (Figura 1. en el celu 11/09/19) cada vez que se ejecuta el quantum se deposita el PRUN en la Cola

Prácticamente este planificador replica el código del planificador general, dado como esquema (Figura 3. en el cuaderno)

```
void Planificador(){
        if(finalizo(PRUN))
                FreeMen(PRUN);
        else{
                PRUN.Reg = CPU.Reg;
                Q.meter (PRUN);
        }
        //"Buscar" al nuevo PRUN
        PRUN = Q.sacar();
        CPU.Reg = PRUN.Reg; //dar la CPU al PRUN
}
```

Note que hemos abreviado "Registers" por "Reg", Por implementaciones de algunos S.O.

Quantum's x Proceso

Es la cantidad de quantum's seguidos que , adrede, el planificador le da a un proceso

- Escribir un planificador RR, sabiendo que maneja procesos que nunca finalizan por sus propios medios.

  /* "Nunca finaliza por sus propios medios" = El proceso jamas le dice al S.0. que va a finalizar = si usamos

  if(finalizo(PRUN)) es false

  */

  Solución.- Si me dicen que los procesos nunca finalizan por sus propios medios, entonces ya no uso la función finalizo

- Escribir un planificador RR, que asigne 2 quantum's por proceso

Solución.- Usaremos un contador global (int cqp = 0;), el cual contara la cantidad de quantum's que ha recibido el actual PRUN.
Para decir que una variable es global, simplemente se la declara fuera del Planificador.

Pero, debe tomarse en cuenta que un proceso puede finalizar en el primer quantum dado.
Bosquejo para cuando me piden **n** quantum's por proceso

```
int cqp = 0; //var global
void Planificador(){
        cqp++;
        if(cqp == 2 || finalizo(PRUN))
                FreeMen(PRUN);
        else{
                PRUN.Reg = CPU.Reg;
                Q.meter (PRUN);
        }
        PRUN = Q.sacar();
        cqp = 0;
        CPU.Reg = PRUN.Reg;
}
```

4.6. Planificador con colas de prioriadad

En el "mundo real", no todos los procesos necesitan la C.P.U. en igual cantidad que los demás.

Por ejemplo, los procesos de tiempo real necesitan la C.P.U. mas veces que os demás pues están en una comunicación 'al vivo'

Cola Q[n];

Se dice que un proceso tiene mas prioridad que otro proceso, si el planificador le asigna mas quantum's en un mismo periodo de tiempo

Este planificador define 'n' colas, una por cada prioridad. Tecnicamente define un vector de colas
Por ejemplo, para n=3 colas, tenemos las colas Q[1],Q[2],Q[3].

En la programación, la 'Prioridad' se refiere al indice de la cola e.g. los PCB's de la cola Q[2] tienen prioridad = 2

/* Para sacar un elemento de Q[3]

PRUN = Q[3].sacar();

Para insertar a Q[1];

Q[1].meter(PRUN);

*/

4.6.1. Quantum x Cola

Es la cantidad de PCB's que deben salir de una cola, antes que el planificador cambie de cola Por ejemplo: Supongase que se tiene 3 colas de personas

Q[1] = Cola de 'Viejitos'
Q[2] = Cola de las Damas
Q[3] = Cola de Hombres

Para lograr la prioridad, podemos:

- Que salgan de la cola Q[1] (sean atendidos) 4 'Viejitos', luego...
- Que sean atendidos 2 Damas de Q[2] y luego...
- Que son atendidos 1 hombre de Q[3]

(Figura en el cuaderno)

4.6.2. Cola 'Alta' y cola 'Baja'

Para resumir, se llama cola alta a la cola de mayor prioridad y cola baja a la cola de menor prioridad. En el ejemplo anterior cual seria la cola Alta?

Cola Alta = Q[1]
Cola Baja = Q[3]

4.6.3. Nombres de Prioridad

Pero para el usuario del sistema operativo, no le basta el numero de prioridad porque le seria confuso. Por este motivo, el S.O. se inventa nombres a la prioridad.

Por Ejemplo: Usando las 3 colas entonces

- A la prioridad 1 (a los de Q[1]) le llamamos 'Alta'
- A la prioridad 2: 'Normal'
- A la prioridad 3: 'Baja'

4.6.4. El campo 'Prioridad' del P.C.B.

Siempre que se utiliza mas de una cola, en el P.C.B. se tiene un campo adicional 'Prioridad'

```
class PCB{
    int P.I.D.;
    ----------
    ----------
    ----------
    int Prioridad;
}
```

`int Prioridad` almacena el nro (indice) de la cola al cual pertenece el P.C.B.

```
/* PCB P;
   P = Q[2].sacar();
   Println(P.Prioridad);
*/
```

4.7. Planificación y velocidad de la C.P.U,

Se sabe que el quantum (q) es una interrupción a la C.P.U., para que deje un momento el código que esta ejecutando y ejecute el planificador.

Sin embargo, debe recordarse que un microprograma (procedure, void que esta en el Core de la C.P.U.) no puede interrumpirse, pues son instrucciones atómicas

//átomo = sin partes = que no puede dividirse

- En que tiempo se ejecutan estos procesos?

P0	P1
MOV AX,16 //3 ciclos	LEA BX,15 //1⎍⎍
MUL BX //4 ciclos	XOR AX,BX //1⎍⎍
ADD AX,BX //2 ciclos	DIV BX //4⎍⎍

sabiendo que:

- CPU $\sim$ 10 Hz
- El planificador ocupa 6 ciclos (Planificador RR)
- El quantum q = 2 ciclos
- El IP de la CPU esta en la 1era instrucción de P0

Solución: figura 1 en el cuaderno

El quantum dice que tiene que parar cada 2 ciclos, sin embargo cuando la instrucción vale mas que eso (> 2), el C.P.U. no le deja actuar al quantum hasta que dicha instrucción termine.

El tiempo en que tarda estos procesos es:
$39 * \frac{1}{10} = 3{,}9$ segundos

- El mismo enunciado del anterior ejercicio

P0	P1
SUB AX,10 //2 ⎍	MOV AX,BX //1⎍
AND AX,BX //1 ⎍	LEA BX,10 //2⎍
	SUB AX,BX //1⎍

  - T(planificador RR) = 0.5 seg
  - quantum cada 2 ⎍
  - C.P.U. $\sim$ 20 Hz

Solución:

Se aumenta la cantidad de ciclos hasta llegar al quantum

El tiempo en que tarda estos procesos es:
Total $= 8⎍ + 0{,}5 + 0{,}5 + 0{,}5 = 8 * \frac{1}{20} + 1{,}5 = 1{,}9$ segundos

4.8. Ejercicios Previos al 1er Parcial (P1)

- Un individuo ingreso un código a la R.A.M. (Figura 1 en el cuaderno)

  Que necesita este código para correr (RUN)?

  /* Correr = Run = Que recibe el quantum = Que recibe la CPU */
  Solución: Necesita de un PCB, el cual debe estar insertado en la ED del planificador.
  /* e.g. si el planificador es RR (Figura 2 en el cuaderno) */

  Recordemos que el planificador se ejecuta cada vez que se cumple el quantum y es el único que da la C.P.U. a los procesos cuyos P.C.B. están en su estructura de datos

- Dos computadoras A y B con las mismas características de Hw y Sw cargan un mismo proceso P y lo ejecuta pero la computadora A usa un S.O. multiproceso, la computadora B usa el mismo S.O. pero monoproceso.

En cual de ellas corre mas rápido P?

(Figura 3 en el cuaderno)
Solución: En la computadora B (la monoproceso)

Porque?

La computadora A debe ejecutar el planificador cada vez que se ejecuta el quantum por este motivo corre P mas lentamente, el monoproceso no tiene planificador, por ello no lo 'interrumpen' al realizar el proceso
Nota.- Se esta asumiendo que el proceso P necesita muchos quantum's para completar su código. Si solo necesitase en un quantum, ambas computadoras ejecutarían P al mismo tiempo.

- Una computadora es de 64-bit pero tiene un MDR de 32-bit.

Sera posible esto? Porque?

Solución: El ancho del Bus de Datos es de 64-bits (64 hilos). El MDR (ACC) es de 32-bits. Esto no seria posible, porque el MDR solo almacenaría 32-bits. Los otros 32-bits se perderían.(Figura 4 en el cuaderno)
(En cada ráfaga)

- Un planificador usa dos colas $Q_1[\,]$ y $Q_2[\,]$ y asigna un quantum por proceso y un quantum por cola (Figura 5 en el cuaderno)

  a) **Este planificador es de Prioridad?**
  Solución: No. Porque todos los procesos y todas las colas reciben la misma cantidad de quantum's. Es decir, este planificador trata equitativamente a los procesos y a las colas.

  b) Dado que el planificador no es de Prioridad: **Existe o no el campo prioridad?**
  Solución (Figura 6 en el cuaderno)
  Recordemos que el campo prioridad nos sirve para saber a que cola pertenece el proceso.
  Para recordar: Siempre que el planificador use más de una cola ($Q_1[\,]$,$Q_2[\,]$,...), el PCB Si tiene el campo entero Prioridad, si solo existe una no lo tiene.

  c) Implemente este planificador, tomando en cuenta que los PCB's que salen de una Cola se depositan en la otra Cola

- Completar los problemas planteados el día miércoles

- Implementar un planificador RR, sabiendo que los procesos nunca finalicen por sus propios medios y la E.D. donde se almacenan los P.C.B.'s es un vector.
Solución:

```
int i = 0; //PRUN = v[i]
void Planificador (){
    PRUN.Reg = CPU.Reg;
    -
    i = next(i);
    PRUN = v[i];
    CPU.Reg = PRUN.Reg;
}
```

- Escribir un planificador que trabaja con N colas: Q[N], Q[N-1], ..., Q[1] y:
  - Trabaja con procesos que nunca finalizan por sus propios medios.
  - Se asigna K quantum's x cola a la cola Q[K].

```
/*  1q x Cola a Q[1]
    2q x Cola a Q[2]
    3q x Cola a Q[3]
        (etc)
*/
```

  - El orden de atención es así: Q[N], luego Q[N-1], luego Q[N-2],...

Solución:

- Como es 1q x proceso, no necesitamos un "Contador de q's x proceso (cqp)" ni el 'if grandote'
- Como hay q x cola (mas de 1), usamos

```
int cqc = 0
```

y nextCola()

```
int nextCola(int k){
    if(k == 1)
      return N;
    else
      return k-1;
}
```

El general seria:

```
int cqc = 0; int k = N;
void planificador(){
     int i = PRUN.Prioridad();
     PRUN.Reg = CPU.Reg;
     Q[i].meter(PRUN);
     –
     PRUN  = Q[K].sacar();
     cqc++;
     if(cqc == K){
       K = nextCola(K);
       cqc = 0;
     }
     CPU.Reg = PRUN.Reg;
}
```

(a) A que se llama concurrencia? Es ejecutar o procesar en un mismo periodo de tiempo
Por ejemplo En una hora, un individuo:

- Trapeó su cuarto
- Acomodo su cama
- Acomodo sus cuadernos

En (el periodo de) una hora hizo estos 3 procesos
El proceso concurrente es implementado actualmente usando el quantum (figura 1 en el cuaderno)

(b) A que se llama Procesamiento Paralelo?
//Paralelo = al mismo tiempo = simultáneamente
Para que corran dos procesos en paralelo, se necesita una CPU por cada uno (figura 2 en el cuaderno)

(a) Dibuje el diagrama cebolla (o de estratos) del S.O.
//Estratos = capas
Solución: (Figura 3 en el cuaderno)

(b) Dibuje el diagrama "Plataforma"
Solución: (Figura 4 en el cuaderno)

Capítulo 5

Exclusión Mutua

Mutual Exclusión = Mutex //Sincrono, que va en orden.

5.1. El Problema

Supongase que en una cuenta bancaria hay 100 bs, la cual esta guardada en una variable global **saldo**

float saldo; //var global
⋮
//saldo = 100

también asuma que a esta cuenta pueden acceder 2 tarjetas. El algoritmo para actualizar la cuenta es el siguiente

Parámetro: monto (dinero a sacar de la cuenta)

```
(1) if(monto ≤ saldo)
      (2)saldo = saldo − monto; //Retiramos el dinero
    else
      (3)print(``Saldo insuficiente'');
```

Supongamos que ambas tarjetas quieren retirar los 100 bs al mismo tiempo

//ATM = Automatic Teller Machine = Cajero Automático

(Figura 1 en cuaderno)

Proceso 1	Proceso 2
(1) **if** (monto <= saldo) (2) saldo = saldo − monto; **else** (3) print(``Saldo insuficiente'');	(4) **if** (monto <= saldo) (5) saldo = saldo − monto; **else** (6) print(``Saldo insuficiente'');

Pero, como P1 y P2 corren en forma concurrente (i.e. por quantums) los procesos pueden producir errores.

5.1.1. Grafo de Planificación

Este grafo muestra la ejecución de las lineas de los procesos, siguiendo una planificación albitraria (Figura 2 en el cuaderno)

5.1.2. Otro grafo

(Figura 3 en el cuaderno) **Porqué hubo este error?**

5.2. Sección Critica

En ingles: Critical Section.

Se llama sección critica a un área de memoria de interés común para dos o mas procesos.

/* Área de memoria de interés común = Variable global usada por 2 o mas procesos e.g. (Figura 4 en el cuaderno) */

/* Acceder = Leer o escribir = R/W

```
x = 5; // write x
y = x - 1; //read x
print(x); //read x
```

*/

En el problema de los ATM's la sección critica es la variable global saldo (Figura 5 en el cuaderno) P1 y P2 accede a Saldo; esta variable es de sumo interés, porque con esta variable se decide si se paga o no;

5.3. Condición de Concurso o Carrera

Se llama condición de concurso cuando dos o mas concursos acceden a la sección critica

Una condición de concurso muy probablemente generara errores.

La exclusión Mutua, es la solución a la condición de concurso

A este problema se le han propuesto muchas estrategias de exclusión mutua, desde la década de 1960. Actualmente la solución viene en el Hardware y en los lenguajes de programación

1. Exclusión = Un proceso excluye al otro de la Sección Critica (S.C.) si él la está usando.
2. Mutua = La exclusión se hace, entre ellos sin la ayuda de nadie.

La primera 'solución' obvia seria usar una variable `boolean` que aquí se le llama candado o cerrojo la cual

candado = true ← Hay un proceso en la S.C.
candado = false ← No hay ningún proceso en la S.C.

5.4. Uso de Candados o Cerrojos

boolean candado = false; //var global

P1

```
(1) if(candado == false){
    (2)candado = true;
    (3)S.C.();
    (4)candado = false;
    }
```

P2

```
(5) if(candado == false)
    (6)candado = true;
    (7)S.C.();
    (8)Candado = false;
```

Un Grafo de Planificación
(Figura 1 en el cuaderno)

Un grafo de planificación no necesariamente se hace hasta que ambos procesos finalicen. Solo se dibuja hasta mostrar el problema. (Figura 2 en el cuaderno)

/* Sugerencia Se puede, en el grafo de planificación, usar Zig Zag, para mostrar un error en el esquema de E. Mutua (Figura 3 en el cuaderno) */

En general, los esquemas de Exclusión Mutua, basados en candados, NO FUNCIONAN

(○) Ejercicio

Un programador quiso evitar la condición de concurso, poniendo 2 candados C1 y C2.

candado = 0 (abierto)
candado = 1 (cerrado)

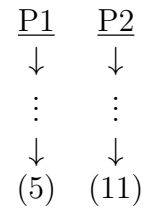
```
int c1 = 0; c2 = 0;
```

```
        P1
(1) if(c1 == 0){
    (2)c2 = 1;
    (3) if(c2 == 0){
        (4)c2 = 1;
        (5)S.C.();
        (6)c1 = c2 = 0;
    }
}
```

```
        P2
(7) if(c1 == 0){
    (8)c2 = 1;
    (9) if(c2 == 0){
        (10)c2 = 1;
        (11)S.C.();
        (12)c1 = c2 = 0;
    }
}
```

Este esquema funciona?

Para demostrar que el esquema no funciona usamos un grafo de planificación y ahí mostramos que ambos procesos están en la sección crítica.



(IMAGEN)

Otra forma

$$c1 = c2 = 0$$

(IMAGEN)

Definitivamente los candados no funcionan
//Dead lock

5.5. Espera Ocupada

Este esquema si funciona

Nota para recordar

La gran mayoría de los esquemas de (exclusión mutua) de mutex hacen uso de una espera ocupada, como el `syncronized` de JAVA.

La espera ocupada. hace mas lento, significativamente nuestro procedimiento o fragmento de código.

Espera Ocupada = Espera Ocupando la CPU //Espera Artificial

Este esquema de exclusión utiliza una variable global `Turno` la cual le dice al proceso que a el le toca entrar a la sección critica

Este esquema recibe el nombre completo de:

"Alternación Estricta con espera ocupada"

Si tenemos los procesos P0 y P1

`Turno = 0` ⇒ P0 entra a la S.C.
`Turno = 1` ⇒ P1 entra a la S.C.

Nota.- El **loop-Forever** que se escribe en los esquemas de exclusión mutua sirve para demostrar que el esquema funciona, aún cuando se lo ejecute una y otra vez.

```
While(true){  //loop-Forever
    if(...){
        ---
        ---  //<- Esquema
        ---
    }
}
```

```
int Turno = 0; //Var global, P0 entrara primero a la S.C.()
```

P0	P1
`While(true){`	`While(true){`
`(1) While(Turno != 0)`	`(4) While(Turno != 0)`
`;`	`;`
`(2) S.C.();`	`(5) S.C.();`
`(3) Turno = 1;`	`(6) Turno = 0;`
`}`	`}`

La espera ocupada se lleva a cabo en estas lineas

```
(4) While(Turno != 0) //o 1
;
```

La cual puede escribirse así:

```
(4) While(Turno != 0) {//o 1
}
```

Si el while es verdadero, el proceso no entra a la S.C.(). Haciendo un grafo de planificación.

(IMAGEN)

Se llama "Alternación estricta" porque se alternan estrictamente en orden, el ingreso a la sección crítica (S.C.)

PO, P1, P0, P1, P0,...
1ero 2do 3ero 4to 5to

/* No seria " estricta " la alternación, si
PO, P0, P1, P0, P1,P1
*/

5.5.1. Inconvenientes de este esquema

/* Este esquema evita la condición de concurso. Es decir es un esquema de exclusión mutua. Lo que se discute aquí es la eficiencia del Esquema */

Al ser de alternación estricta, un proceso no podrá ingresar a la S.C., si el otro proceso no lo ha hecho aún.

Por ejemplo, Supongamos que `Turno = 0` pero P0 no quiere entrar a la S.C., pero P1 si.

(IMAGEN)

5.6. Instrucción TSL (Assembler)

// TSL = Test 8 Set Lock = Testeo y cierre (el candado)
// Es una instrucción atómica.

A pedido de los desarolladores de software, toda CPU, incorpora una instrucción TSL. Como se sabe toda instrucción de la CPU es un micro programa alojado en el Core y que se ejecuta en forma atómica (La CPU no puede ser interrumpida hasta completar el micro programa)

$$\texttt{TSL Reg, Var} \rightarrow \begin{cases} Reg = Var; \\ Var = 1; \end{cases}$$

Por ejemplo:

$$\texttt{TSL EAX, Candado} \rightarrow \begin{cases} EAX = Candado; \\ Candado = 1; \end{cases}$$

```
        int Candado = 0  //0 = Abierto, 1 = Cerrado
ciclo:
        TSL EAX, Candado //EAX = candado; candado = 1
        CMP EAX, 0       //if(EAX != 0)
        JNE ciclo        //Goto ciclo;
        S.C.();
        Candado = 0;
```

(IMAGEN)

5.6.1. Probando el esquema

```
int Candado = 0

P0
 while(true){
  (1) ciclo: TSL EAX, Candado //EAX = candado; candado = 1
    (2) CMP EAX, 0       //if(EAX != 0)
    (2) JNE ciclo        //Goto ciclo;
  (3) S.C.();
  (4) Candado = 0;
 }

P1
 while(true){
  (5) ciclo: TSL EAX, Candado //EAX = candado; candado = 1
    (6) CMP EAX, 0       //if(EAX != 0)
    (6) JNE ciclo        //Goto ciclo;
  (7) S.C.();
  (8) Candado = 0;
 }
```

- Usando la instrucción TSL, el primer proceso, que ejecute esta instrucción, sera el que entre a la sección critica. El otro proceso queda en una espera ocupada, hasta que el `Candado = 0`

(IMAGEN)

- Cuando P1, ejecuta la linea (1) El TSL, abrió el candado. Por lo tanto P2 se queda en una espera ocupada.

5.6.2. Implementación en lenguajes de programación

Programación

- En Delphi, se implementa con 2 procedimientos.

```
Procedure enterCriticalSection(Candado){
 ciclo: TSL EAX, Candado
     CMP EAX, 0
 JNE ciclo
}

Procedure leaveCriticalSection(Candado){
 Candado = 0;
}

// Con estos procedimientos
```

```
P1                                  P2
 enterCriticalSection(Candado);      enterCriticalSection(Candado);
 SC();                               SC();
 leaveCriticalSection(Candado);      leaveCriticalSection(Candado);
```

- En Java utiliza la palabra reservada `synchronized`, la cual compila un esquema TSL en el bloque que hemos elegido.

```
synchronized(lista){
        SC() { x = lista.get(i);
               -
               -
               -
} // Candado = 0;
```

5.7. Dead Lock

//Dead lock = abrazo mortal = interbloqueo

Ocurre cuando los procesos esperan que la sección critica este disponible (creyendo que hay otro proceso en ella), pero la S.C. esta libre.

P1 cree que P2 esta en la S.C. y espera...

P2 cree que P1 esta en la S.C. y espera...

(IMAGEN)

Capítulo 6

Hilos

//Hilos = Thread.
Un hilo es como un subproceso que corre en el mismo espacio de direcciones del proceso.

/* Cuando un proceso está en RAM

(IMAGEN)
*/
Por este motivo a los Hilos se le llama soft - (sub) procesos.

//Soft process = procesos livianos o subprocesos.

6.1. Diferencia entre subprocesos o Hilo

La diferencia está en que: los subprocesos son procesos en si mismos y, por lo tanto reciben el quantum del planificador del sistema operativo. En cambio, un Hilo esta en el mismo espacio de dirección del proceso y hace uso del quantum que le llaga al proceso.

(IMAGEN)

El subproceso tiene su propio espacio de dirección y es tratado como un proceso más.

(IMAGEN)

Desde esta perspectiva, el proceso es considerado un Hilo denominado Hilo - principal o main - thread.

Entonces, si una aplicación (App) no usa Hilos, se lo considera una aplicación mono - hilo.

- No usa hilos: mono - threads;

- Usa hilos: multi - threads;

> Nota
>
> Dado que el proceso se lo considera un hilo el sistema operativo ve al proceso como un subproceso de él.

E.g. Hemos hecho una APP que suma 2 números: suma.exe

Si lo vemos con el ADM de Tareas:

Nombre de la Imagen	PID	Cant. de Subprocesos
suma.exe	900	1

> Nota
>
> También en el ADM de Tareas consideran a los hilos subprocesos porque Windows es consciente de los hilos que usa el proceso.

(∘) Supongamos que en el ADM de Tareas vemos esto

Imagen	Subprocesos
Prog.exe	6

y se sabe que Prog.exe solo esta usando hilos. Que significa el 6?

Solución:

Prog.exe está usando 5 hilos (como el proceso padre es considerado un hilo, vemos 6)

6.2. Implementación de Hilos

Existen 2 formas de implementar los Hilos:

i) Versión Vieja: El sistema operativo no es consciente (no sabe) que la App esta implementando Hilos por su cuenta. En este caso el compilador o el programador se encargan de la sub-planificación, del quantum que le llega al proceso.

ii) El S.O. es consciente (si sabe) de los hilos que corren en el proceso. Actualmente, los hilos se toman como subprocesos del proceso.

El S.O. le crea un sub-planificador a la App y | El proceso finaliza cuando <u>todos</u> los hilos finalizan |

- Si el main-thread finaliza en la i) (primera versión) mata a sus hilos aunque no hayan finalizado.
- Si el main-thread finaliza en la ii) (segunda versión) no finaliza hasta que sus hilos finalicen.

/* La implementación

i) se llama ULT = User Level Thread

ii) se llama KLT = Kernel Level Thread

*/

6.3. El "Ejecutar"

//En Java: void run()...

//En Delphi: Procedure Execute; Begin.... end;

Para simplificar la implementación de hilos los lenguajes de programación usan una `class`, de la cual se debe heredar y sobreescribir el ejecutar. El ejecutar es un procedimiento que es ejecutado por el hilo solo una vez.

Es decir, el hilo corre el ejecutar y muere (Finaliza)

(∘) Preg. Examen, Porque se tiene que volver a declarar el hilo?

Por ejemplo

```
class Hilo extends Thread { //extends = heredable
 @Override
  public void run(){
   system.out.println("Hola");
  }
}
```

En otro lado, usamos la class:

```
Hilo h1, h2;
h1 = new Hilo();
h2 = new Hilo();
h1.Start(); //h1 corre y empieza a recibir
            //quantums del planificador
            //h1 corre void run(){...}
```

Cuando el hilo termina de correr su procedimiento ejecutar (En hilos), los recursos que el utilizo son devueltos al S.O. y su PCB (TCB) es sacado de la estructura de datos del subPlanificador. Por este motivo, no es posible reiniciar el Hilo ya finalizado.

Si quiere llama de nuevo a `h1.start()`

`h1.start();` ← Error h1 ya no tiene PCB

Si quiero volver a correr h1

```
h1 = new Hilo(); ← Crear PCB para h2
            h1.start();
```

En el main thread puede finalizar pero los hilos pueden seguir corriendo.

Java usa KLT, no implementa los hilos, usa al S.O. para implementar hilos.

Otro ejemplo:

```
class Hilo extends Thread {
 @Override
  public void run(){
    while(true){
     System.out.print(".");
    }
  }
}
```

En el main thread:

```
Hilo h1 = new Hilo();
h1.Start(); //libera al hilo y empieza a
            //correr en paralelo por eso no
            //se pone directo run()
System.out.println("Chau");
```

(∘) Se tiene la sgte class

```
class Hilo extends Thread {
 @Override
  public void run(){
   for(int i=1; i<=3; i++){
    System.out.println(i);
   }
  }
}
```

En el main hacemos:

Version 1

```
void main(){
   Hilo h = new Hilo();
   [h.start();]
   for(int i = 1; i<=5; i++){
      System.out.println("bye");
   }
}
```

Version 2

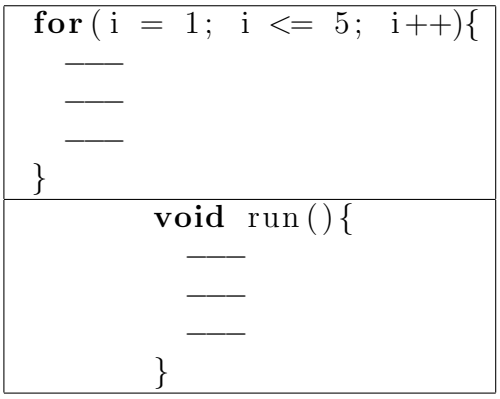
```
void main(){
   Hilo h = new Hilo();
   [h.run();]
   for(int i = 1; i<=5; i++){
      System.out.println("bye");
   }
}
```

El `h.start` hace que el hilo se haga al mismo tiempo que el Main Thread. El `h.run` hace que se llame al metodo, luego de hacer el metodo vuelve al programa principal (Main Thread)

Cual es la diferencia entre la version 1 y la version 2

En la version 1 luego del `h.start()`, tanto el main thread (hilo principal) (proceso) continua con su ejecución, mientras el hilo `h` ejecuta el run()

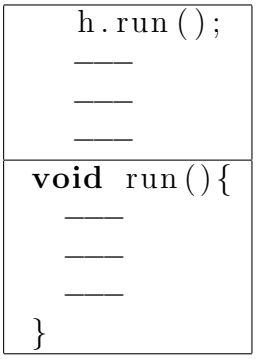
Proceso (Versión 1)



1) Main Thread, 2) Hilo h

En la versión 2 el main thread ejecuta el método run() como cualquier otro. Luego de ejecutar el run(); el main thread continua con el `For`. El hilo `h` jamas inicia su ejecución.

Proceso (Versión 2)



1) Hilo h (no iniciado, no corre), 2) Main Thread
El Hilo h viene al mundo a ejecutar su `ejecutar`

6.3.1. Alternación Estricta

Lanzar 2 Hilos h1 y h2 de una misma `class`, que accedan en alternación estricta a la sección crítica.

`class Hilo`

La SC() la vamos a simular con un mensaje (`System.out.println`)

Solución

```
Class Hilo extends Thread{
  Public static int Turno = 0;
  Public int ticket;

  @Override
    Public void run(){//Ejecutar
      while(true){
        while(Turno != ticket){
          System.out.println("SC");
          Turno = 1 - Turno; //Toggle 0,1
        }
      }
    }
}
```

```
Main Thread
  Hilo h0,h1;
  h0 = new Hilo();
  h1 = new Hilo();

  h0.ticket = 0;
  h1.ticket = 1;

  Hilo.turno = 1;

  h0.start();
  h1.start();
```

6.4. Uso de Synchronized

/* (Auto) Caja Mecánica = Sincrono
(Auto) Caja Automática = Asíncrono (Usando Hilos)
*/

Todo programador, que no conoce hilos realiza una programación. Es decir, las lineas de su código se ejecutan en forma secuencial

"Sincrona"

```
1)   x = 3;    //1er esta linea.
2)  y = x+1;    //luego esta.
```

Cuando se dice que la ejecución es Asincrona, queremos decir que ese código se ejecuta en forma independiente (en un hilo)

Recuerde
Sincrono = Sin Hilos; Asincrono = Un hilo ejecuta eso

(○) Determinar cual es linea es sincrono y asincrono

- `h.run` .- Sincrono
- `h.start` .- Asincrono

Tecnicamente, la compilación de la palabra reservada `synchronized` es un esquema de exclusión mutua, basado en el **TSL**.

Por Ejemplo:

Si tenemos

```
Synchronized void Algo(int x){
  System.out.println("x=" + x);
} // S.C.
```

En la `class Hilo` tenemos el siguiente `run`:

```
public void run(){
  1) Systen.out.println("hola");
  2) Algo(); //S.C.
  3) System.out.println("bye");
}
```

En el `main-thread` tenemos:

```
Hilo h1 = new Hilo();
Hilo h2 = new Hilo();

h1.start();
h2.start();
```

No es imagen en el hilo, van por el mismo camino, ejecutan el mismo **ejecutar** //Cuando un hilo entra al `synchronized` nadie lo molesta hasta que finalice

(IMAGEN)

6.5. Hilo "Planificador"

/* En realidad, este hilo, no planifica nada, solamente decide que Hilo puede entrar a la sección crítica */

/* Para este problema, no tiene sentido usar la palabra `Synchronized` porque no funciona */

Este problema se resuelve utilizando las bases de la alternación estricta, es decir, los hilos usan un `ticket` y una variable global `turno`. El `turno` es manipulado por el hilo planificador.

(IMAGEN)

Por ejemplo si buscamos un elemento en un vector y mandamos a dos Hilos a buscarlo del principio y el final del vector si no hubiera planificador se pasarían, pero, necesito que se detengan al encontrar el dato o al intersectarse.

Problema a resolver

En la `class Planificador extends Thread`

/* Al crear una class, en el constructor deba inicializar las variables y si hereda de una superclase, debe llamar a ese constructor como primera instrucción */

Podemos hacer esto:

```
class Planificador extends Thread{
  public static int Turno;
  int t;

  Public Planificador(){
    super(); // Importante
    t = 0;
    Turno = t;
  }
}
```

Ej:

```
@Override
  Public void Start(){
    t = 0; //1)
    Turno = t; //1)
    super.start(); //2)
  }
```

1) En caso que quiera hacer instrucciones previas (en este caso es un ejemplo ya que esta todo eso inicializado en el constructor)

2) Llama al método de la superclass y así no se pierde. Ejecutar el start() de la class Thread (Hecha por Java)

Si un procedimiento solo va a ejecutar 1 hilo necesito `synchronized`? No, ya que otro Hilo no invadirá la S.C.()

Capítulo 7

ADM. de Memoria Contigua

El Administrador de Memoria, al que abreviamos **ADM-MEM** es el encargado de administrar la RAM.

7.1. La RAM (Random Access Memory)

La RAM es volatil desde el punto de vista del Sw, la RAM puede verse como un Array (Verctor) de M casillas del tipo byte.

Definimos en C++:

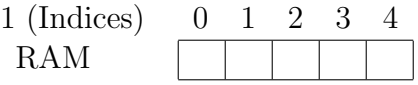
```
byte RAM[M];
```

en Delphi:

```
Var RAM: ARRAY[0...M-1] of byte;
```

Por ejemplo:

```
M = 5 bytes
```



RAM[2] = 80

Pero, usualmente la RAM se le escribe verticalmente y a sus indices se los llama direcciones (address). **No** debemos decir "casilla 2" de la RAM sino "byte 2" de la RAM (en este caso)

Áreas de la RAM

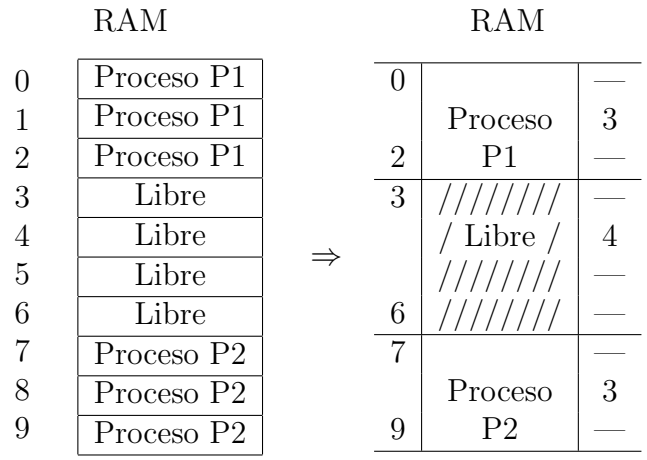
//Exhaustivo = cubre todos los casos y situaciones

En la mayor parte de los ejercicios se necesita dibujar la RAM. Si la dibujaramos en forma exhaustiva (con todas sus casillas) se gastaría mucho espacio y tiempo, por este motivo se la dibuja la RAM por áreas.

Un área de la RAM (Grupo de casillas contiguas) queda definida por:

```
(Dir de inicio, Tamaño)
```

Por ejemplo: con una RAM de M = 10 bytes (casillas)



Las áreas libres se ponen (//////)

7.2. El "Monitor"

Puesto que el S.O. es también un programa, algunos de ellos deben estar cargados en la RAM, para poder ser ejecutados. Por este motivo, el S.O. reserva un espacio en la RAM, para su uso personal. A esta área se le llama monitor.

//Monitor = Parte residente del S.O.

En la IBM-PC, el Monitor toma las primeras direcciones de la RAM (desde la dirección 0), porque en la Dirección 0 esta el vector de interrupciones. Entonces, se considera una RAM "vacía" cuando solamente está el Monitor.

RAM

Monitor (S.O.)
//////////////
//////////////
//////////////
//////////////
//////////////
//////////////
//////////////

RAM vacía

//RAM "vacía" = no tiene procesos cargados. Dato: En Windows 7, el Monitor es de 1MB

Haciendo un Zoom-in al área del Monitor.

RAM

0	Vector de Interrupciones
1	void Planificador()...
2	ED de los ADM's
3	Otros

7.3. Variables de la RAM

Para tomar decisiones más rápidos el ADM-MEM utiliza estas 4 variables globales (todas medidas en bytes)

- M = Cantidad de RAM que viene en el Hw.
- S = Tamaño del monitor
- MemAvail = Memory Available = Cantidad total de memoria disponible (//////)
- MaxAvail = Maximum Available = Tamaño del área libre (disponible) mas grande de la RAM

Por Ejemplo:

RAM

0	Monitor (S.O.)	— 100 —
	//////// //////// ////////	— 80 —
	P1	
	//////// //////// ////////	— 250 —
	P2	
	//////// //////// ////////	— 150 —
M-1	P3	

- M = 1500 bytes
- S = 100 bytes
- MemAvail = 80 + 250 + 150 = 480 bytes (Suma de todos los tamaños de áreas libres)
- MaxAvail = 250 bytes (tamaño del área mas grande libre)

7.4. Fragmentación

Formalmente:

Una RAM ésta Fragmentada si y solo si: $\text{MemAvail} \neq \text{MaxAvail}$

Informalmente una RAM esta fragmentada si tiene 2 o más áreas libres.

Se usa el término **Compacta** para decir **no Fragmentada**.

- RAM Compacta = RAM no-Fragmentada
- RAM Fragmentada = RAM no-Compacta

7.4.1. Grado de Fragmentación

El grado de fragmentación de una RAM anotado GF(RAM), utiliza una medida informal, que se basa en una cantidad de áreas libres.

$$\text{GF(RAM)} \begin{cases} \text{N/A si la RAM esta Compacta} \\ \text{Cantidad de áreas libres} > \text{Si la RAM está Fragementada} \end{cases}$$

//N/A = No aplica

Ejemplo:

RAM A

Monitor	
P1	
////////	—
////////	400
////////	—
P2	
P3	

- MemAvail = MaxAvail = 400 bytes
- RAM Compacta
- GF(RAM A) = N/A

RAM B

Monitor	
////////	—
////////	100
////////	—
P1	
////////	—
////////	200
////////	—
P2	
////////	—
////////	300
////////	—

- MemAvail = 600 bytes (≠)
- MaxAvail = 300 bytes (≠)
- RAM B Fragmentada
- GF(RAM B) = 3 (tiene 3 áreas libres)

RAM C

Monitor	
P1	
//////// //////// ////////	— 100 —
P2	
//////// //////// ////////	— 150 —

- MemAvail = 250 bytes (≠)
- MaxAvail = 150 bytes (≠)
- RAM C Fragmentada
- GF(RAM C) = 2 (tiene 2 áreas libres)

Cual es la RAM más fragmentada?

La RAM A no esta fragmentado, entonces analizamos la RAM B y la RAM C. Puesto que el GF(RAM B) es mayor al GF(RAM C) decimos que la RAM B esta más fragmentada que la RAM C

$$\text{GF(RAM B)} > \text{GF(RAM C)}$$

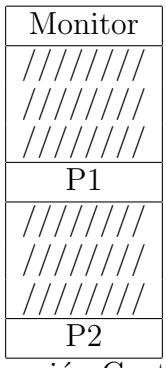
7.5. Asignación contigua y no contigua

Decimos que una asignación es contigua, si cada uno de los procesos son alojados en un solo bloque o área. Por otro lado, decimos que es no contigua, si el proceso es alojado en la RAM en 1 o más partes.

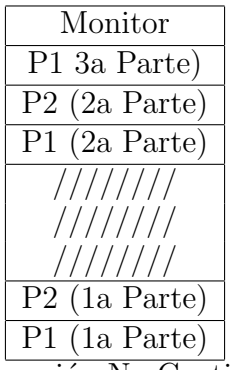
/* La historia de la casa gallega

(IMAGEN)

Por Ejemplo:



Asignación Contigua
(P1 y P2 están cargados totalmente en un solo bloque)



Asignación No-Contigua
(Los procesos ocupan mas de un área)

7.5.1. Estrategia de Asignación Contigua

Desde el primer diseño del S.O. se han propuesto muchas estrategias de asignación contigua, destacamos a:

- Asignación contigua por particiones variables (Multiples)
- Asignación contigua por particiones fijas (o por bloques) -Obsoleta-

7.6. Asignación contigua por Particiones Variables

Consiste en asignar la RAM a los procesos On-Demand (bajo demanda). Es decir según la necesidad de ellos.

Por ejemplo: Una RAM "vacía"se la cargan los procesos P1, P2 y P3

RAM "vacía"

Monitor
////////
////////
////////

RAM Cargada

Monitor	
P1	
P2	— 200 —
P3	
////////	—
////////	250
////////	—

MemAvail = MaxAvail = 250 (RAM Compactada)

Un ADM-MEM Contigua intenta evitar la Fragmentación a toda costa por este motivo, inserta los procesos uno al lado del otro (stacked). Pero la fragmentación se presenta cuando un proceso finaliza.
/*

if finalizo (PRUN)

| FreeMen (PRUN;) |

*/

Supongamos que P2 finaliza

Monitor	
P1	
////////	200
////////	
////////	
P2	
////////	250
////////	
////////	

$$\text{MemAvail} = 200 + 250 = 450\ (\neq)$$
$$\text{MaxAvail} = 250\ (\neq)$$
RAM Fragmentada

- Si un proceso P4 quiere entrar a la RAM con `n = 300 bytes`:
  Que le responde el ADM-MEN?
  Puesto que `n` es mayor que el MaxAvail el ADM-MEM le responde con un error clásico

> Un principio del ADM-MEM:
> "*Si el ADM-MEM es capaz de dar RAM a proceso entonces no le puede negar esta solicitud*"

- Diagrama de Flujo de esta estrategia:

  (IMAGEN)

7.6.1. ED de este ADM-MEM