PROGRAMACIÓN AVANZADA

M. en C. Miguel Alejandro Martínez Rosales

Laboratorio de cómputo móvil
UPIITA - IPN

mrosales81@gmail.com

http://www.labcomputomovil.upiita.ipn.mx/mrosales/

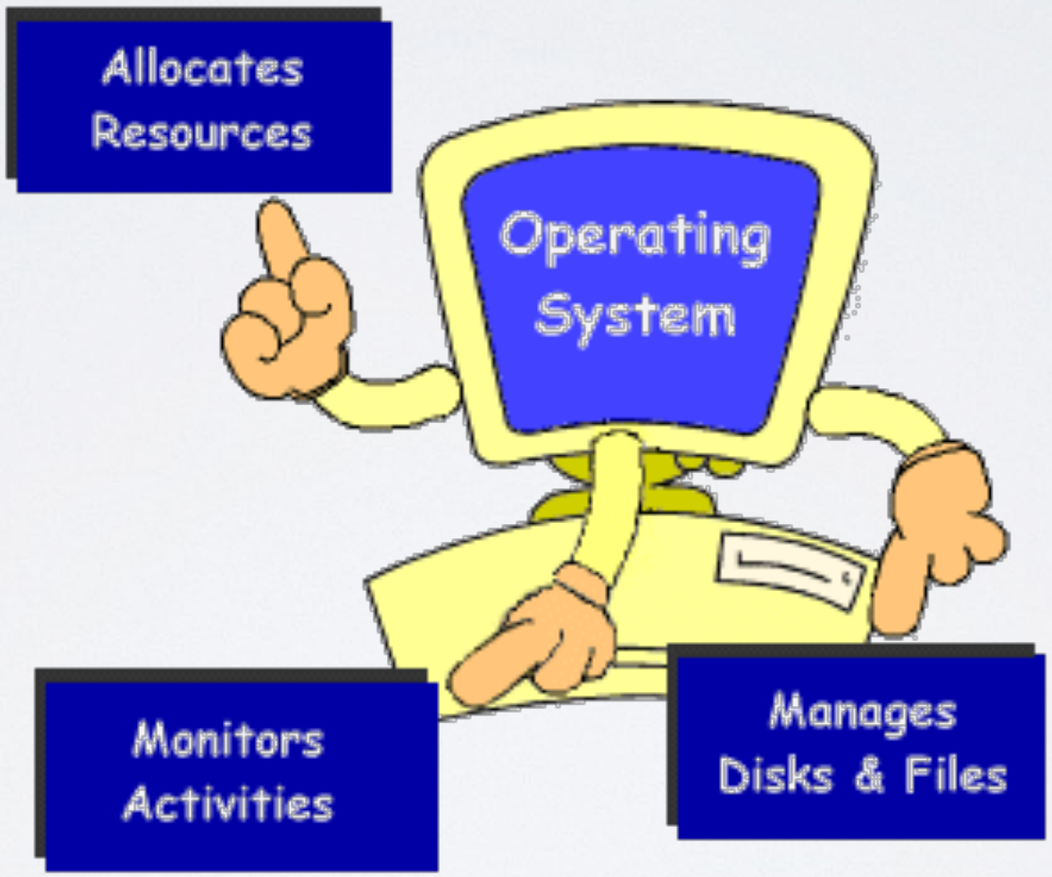
Agosto 2014

CONTENIDO

1. Introducción
2. Interrupciones
3. Procesos
4. Exclusión mutua
5. Sincronización y comunicación entre procesos

ANTECEDENTES HISTÓRICOS

Administrador de recursos



Máquina extendida

ADMINISTRADOR DE RECURSOS

- Proporciona asignación ordenada y controlada de los recursos de hardware entre los distintos programas que compiten por éstos.

- La administración de recursos incluye *multipexaje* de recursos.

- Perspectiva de *bottom-up*.

MÁQUINA EXTENDIDA

- Proporciona un conjunto abstracto de recursos simples.
- Administra recursos de hardware.
- Oculta la parte *fea* del hardware.
- Perspectiva de *Top-down*.

Los sistemas operativos han estado estrechamente relacionados con la historia de las computadoras en donde se ejecutan.

PRIMERA GENERACIÓN

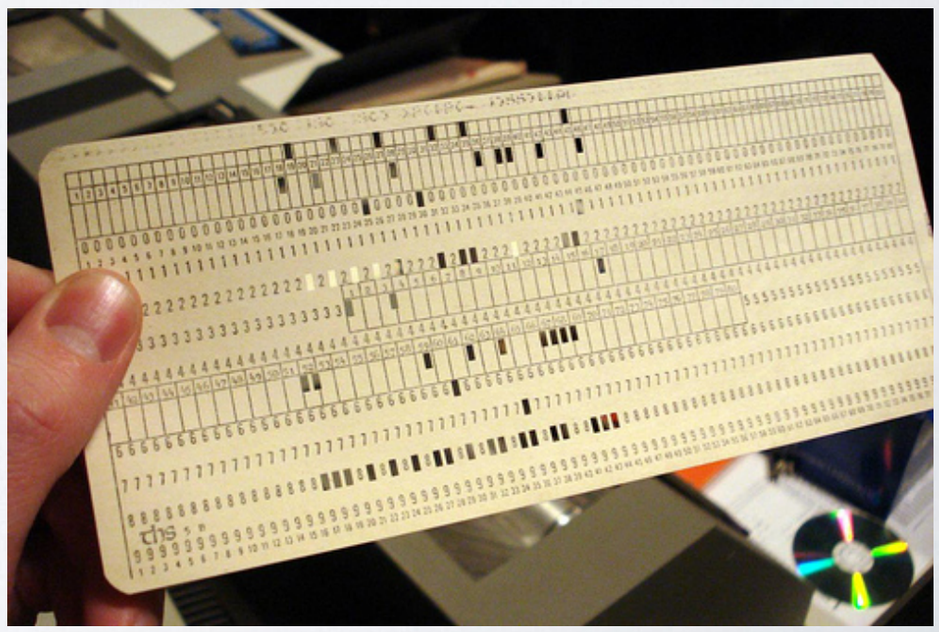
- Programación en lenguaje máquina o *plugboard*.

- No existían los lenguajes de programación.

- …y mucho menos los sistemas operativos.

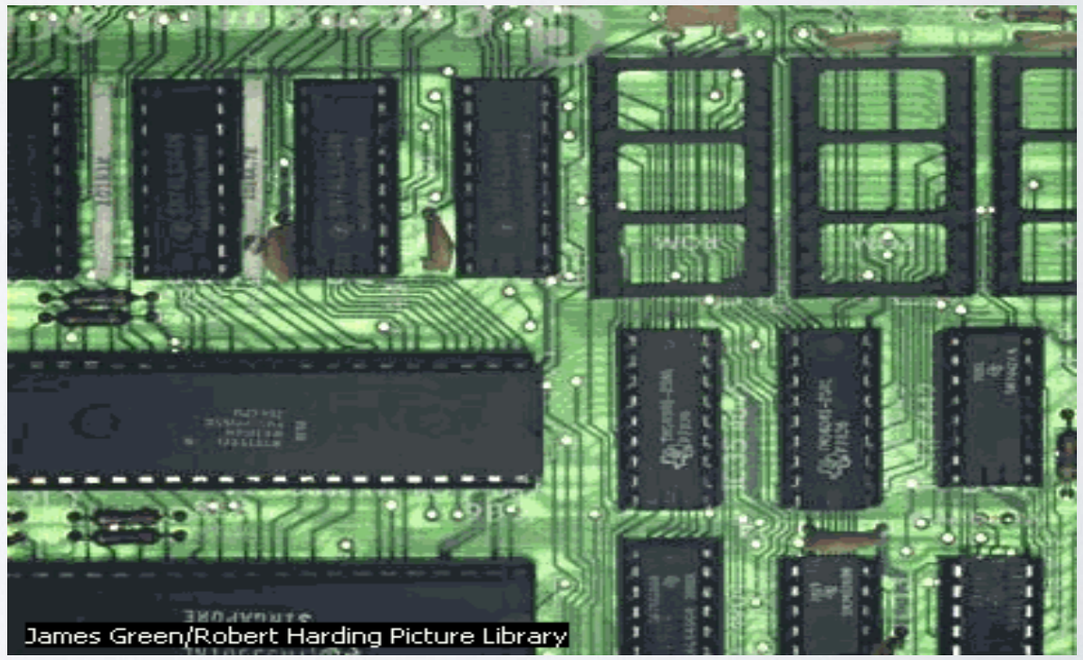
SEGUNDA GENERACIÓN

- Las tarjetas perforadas se procesaban por lotes.
- Se cargaba un programa especial, el cual leía y ejecutaba cada programa en la cinta.
- …este fue el antecesor del sistema operativo actual
  - FMS (Fortran Monitor System)
  - IBSYS (Sistema operativo de IBM para la 7094)



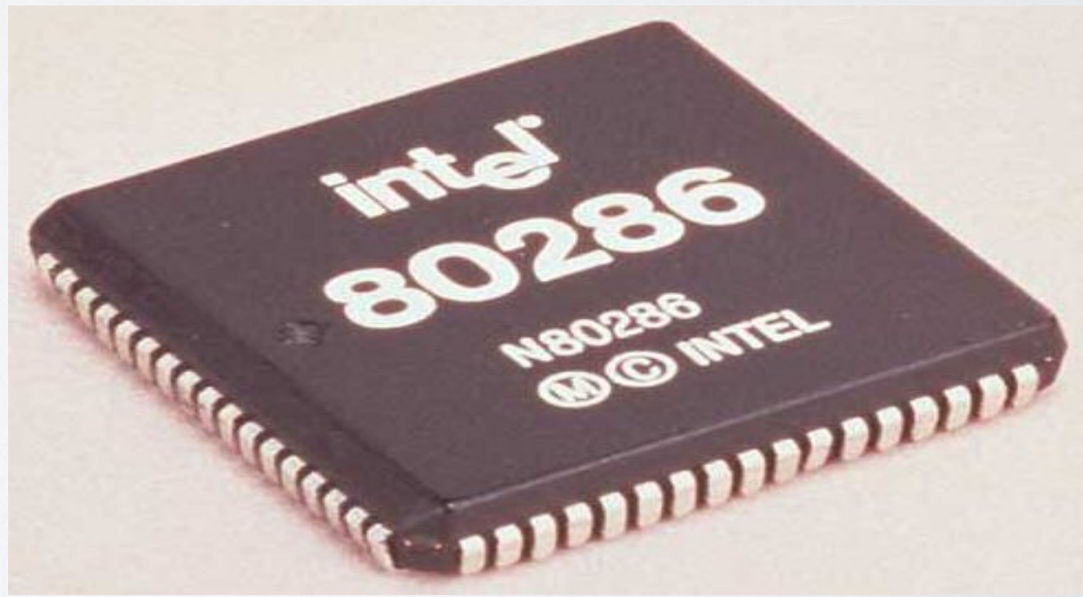
TERCERA GENERACIÓN

- IBM introdujo el OS/360 para la serie System/360.
- Multiprogramación (seguía siendo por lotes),
- Spooling
- Timesharing
- Apareció el sistema operativo MULTICS.
- Surge UNIX y sus variantes (de una modificación de MULTICS).



CUARTA GENERACIÓN

- Surgue CP/M (Control Program for Microcomputers) para 8080.
- Multiprogramación (seguía siendo por lotes).
- Surgen DOS, DOS/Basic, MS-DOS para la familia x86.
- GUI (Graphics User Interface).
- FreeBSD (Mac usan versiones modificadas).



ESTRATEGIAS DE DISEÑO

- Clara idea de lo que se quiere.
- ¿Qué se desea del sistema sistema operativo?
- Para sistemas operativos de propósito general hay que tener en cuenta.

1. Definir las abstracciones.
2. Proveer operaciones primitivas.
3. Asegurar el aislamiento.
4. Administrar el hardware.

ESTRATEGIAS DE DISEÑO

- ¿Por donde podrías comenzar?

PROCESOS

¿QUÉ ES UN PROCESO?

- Un **proceso** es una instancia de un programa, los registros y las variables.

- La **CPU** conmuta entre un proceso y otro para simular la ejecución simultanea de éstos.

- A esta conmutación rápida de un proceso a otro se conoce como **multiprogramación**.

CREACIÓN DE PROCESOS

Hay **cuatro** eventos que provocan la **creación** de procesos.

1. El arranque del sistema.
2. La ejecución desde un proceso, de una llamada al sistema para creación de procesos (**fork**).
3. Una petición de usuario para crear un proceso.
4. El inicio de un trabajo por lotes.

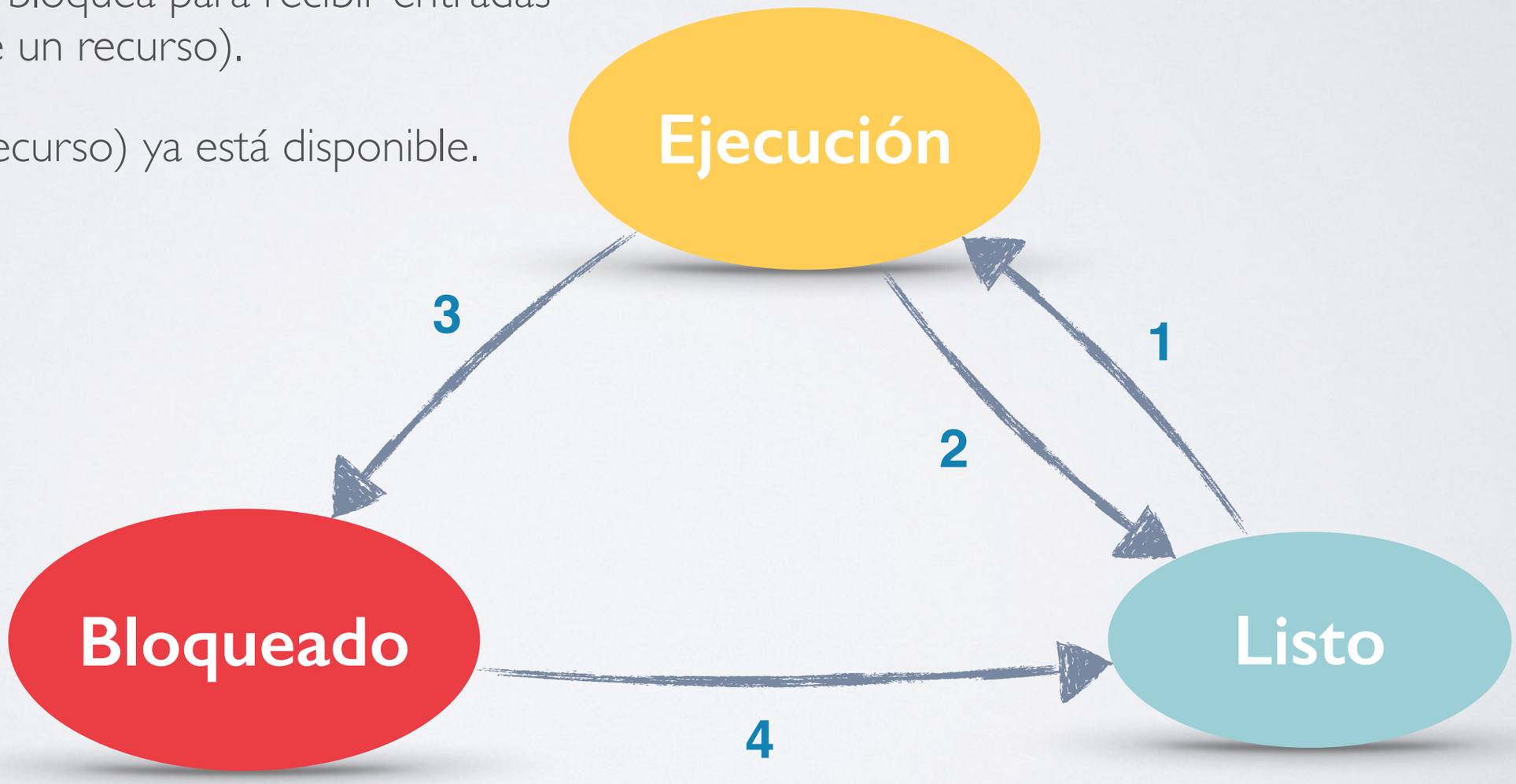
TERMINACIÓN DE PROCESOS

Hay **cuatro** eventos que provocan la **terminación** de procesos.

1. Salida normal (voluntaria).
2. Salida por error (voluntaria).
3. Error fatal (involuntaria).
4. Eliminado por otro proceso (involuntaria).

DIAGRAMA DE ESTADOS DE PROCESOS

1. El planificador selecciona otro proceso.
2. El planificador selecciona entre los procesos.
3. El proceso se bloquea para recibir entradas (en espera de un recurso).
4. La entrada (recurso) ya está disponible.



IMPLEMENTACIÓN DE PROCESOS

Para implementar el modelo de procesos, el sistema operativo mantiene una **tabla de procesos** (arreglo de estructuras), con solo una entrada por cada proceso.

Administración de procesos	**Administración de memoria**	**Administración de archivos**
Registros Contador del programa Palabra de estado del programa Apuntador de la pila Estado del proceso Prioridad Parámetros de planificación ID de proceso Proceso padre Grupo de procesos Señales Tiempo de inicio del proceso Tiempo utilizado de la CPU Tiempo de la CPU utilizada por el hijo Hora de la siguiente alarma	Apuntador a la infomración del segmento de texto Apuntador a la información del segmento de datos Apuntador a la información del segmento de pila	Directorio raíz Directorio de trabajo Descripciones de archivos ID de usuario ID de grupo

IMPLEMENTACIÓN DE PROCESOS

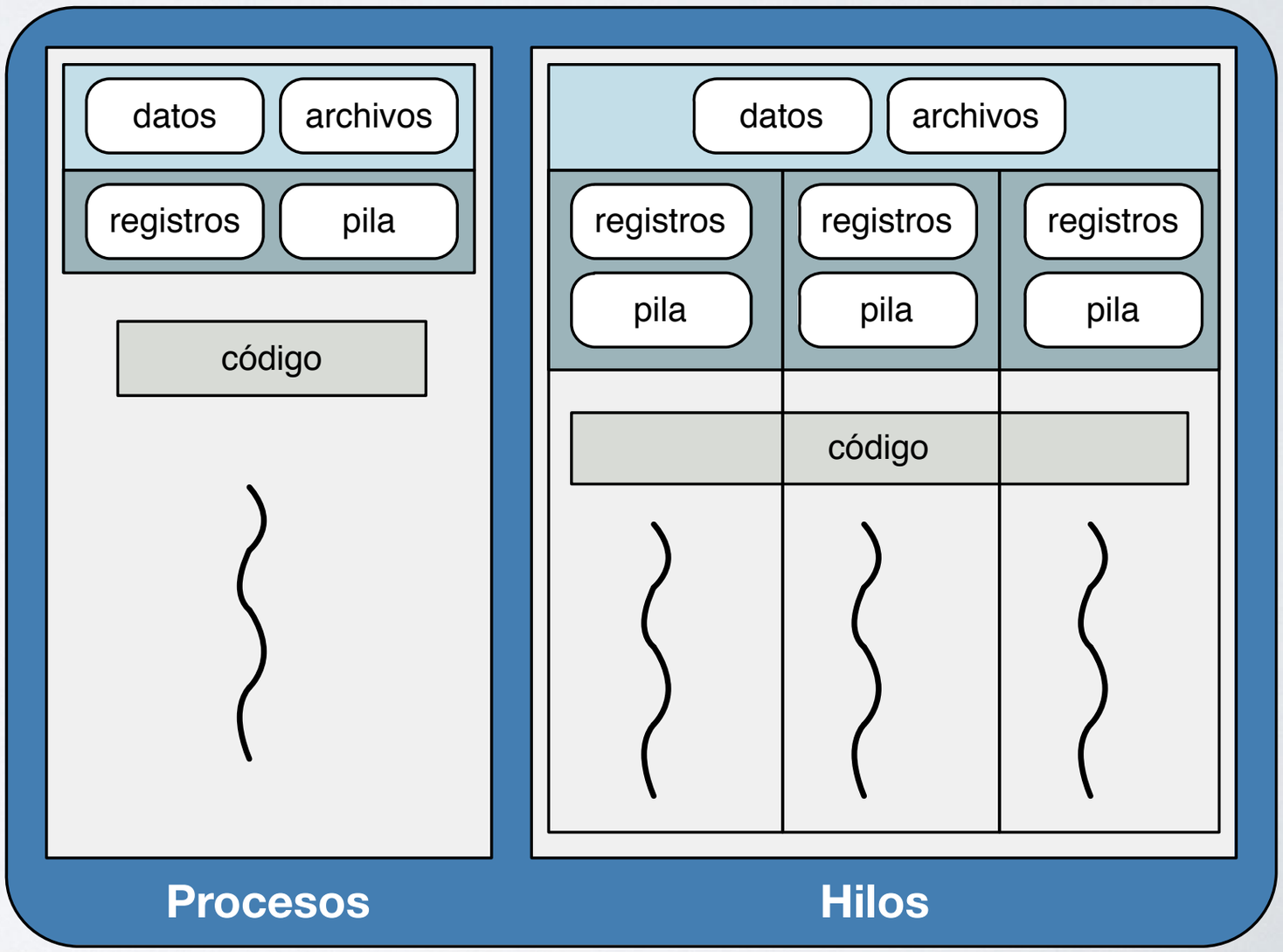
Detalles que darían dependiendo del sistema operativo. Esqueleto de lo que hace el nivel más bajo del sistema operativo cuando ocurre una interrupción.

1. El hardware mete el contador de programa a la pila.
2. El hardware carga el nuevo contador de programa del vector de interrupción.
3. Procedimiento en lenguaje ensamblador guarda los registros.
4. Procedimiento en el lenguaje ensamblador establece una nueva pila.
5. El servicio de interrupciones de C se ejecuta (por lo general lee y guarda la entrada en el búfer).
6. El planificador decide qué proceso se va a ejecutar a continuación.
7. Procedimiento en C regresa al código de ensamblador.
8. Procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

HILOS - THREADS

En los sistemas operativos tradicionales, cada procesos tiene un **espacio de direcciones** y un solo **hilo de control**.

Sin embargo, con frecuencia hay situaciones en las que es conveniente tener vais hilos de control en el mismo espacio de direcciones que se ejecuta en cuasi-paralelo, como si fueran procesos (casi) separados (excepto por el **espacio de direcciones compartido**).



THREADS

Elementos compartidos y privados entre hilos.

Elementos compartidos por los hilos de un proceso.

- Espacio de direcciones
- Variables globales
- Archivos abiertos
- Procesos hijos
- Alarmas pendientes
- Señales y maneadores de señales
- Información contable

Elementos privados para cada hilo.

- Contador de programa
- Registros
- Pila
- Estado

HILOS POSIX

Llamada de hilo	Descripción
pthread_create()	Crea un nuevo hilo.
pthread_exit()	Termina el hilo llamador.
pthread_join()	Espera a que un hilo específico termine.
pthread_yield()	Libera la CPU para dejar que otro hilo se ejecute.
pthread_attr_init()	Crea e inicializa la estructura de atributos de un hilo.
pthread_attr_destroy()	Elimina la estructura de atributos de un hilo.

CONDICIONES DE CARRERA

Cuando dos o más procesos están leyendo o escribiendo algunos datos compartidos y el resultado final depende de quién se ejecuta y exactamente cuando lo hace, se conoce como **condiciones de carrera.**

REGIONES CRÍTICAS

La parte del programa que accede a la memoria compartida se le conoce como **región crítica** o **sección crítica**.

Si pudiéramos ordenar las cosas de manera que dos procesos nunca estuvieran en sus regiones críticas al mismo tiempo, podríamos evitar las **condiciones de carrera**.

EVITAR CONDICIONES DE CARRERA

Hay cuatro condiciones para las condiciones de carrera.

1. No puede haber dos procesos de manera simultánea dentro de sus regiones críticas.

2. No pueden hacerse suposiciones acerca de las velocidades o el número de CPU.

3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos.

4. Ningún proceso tiene que esperar para siempre para entrar a sus región crítica.

EXCLUSIÓN MUTUA

Para evitar que dos o más procesos lean o escriban datos en la memoria compartida, archivos compartidos y todo lo demás compartido, es necesario llevar a cabo **exclusión mutua**.

Esto significa que de alguna forma se debe de asegurar que si un proceso está utilizando una variable o archivos compartidos, los demás procesos se excluirán de hacer lo mismo.

TÉCNICAS PARA EXCLUSIÓN MUTUA

Hay tres posibilidades para asegurar la exclusión mutua entre procesos.

- Deshabilitando interrupciones

  No pueden interrupciones de reloj.

- Variables de candado

  Considerando tener una sola variable compartida (1/0).

- Alternancia estricta

  Se hace uso de una variable **turno** que lleva la cuenta acerca de a que proceso le toca entrar a su región crítica. Evaluar constantemente esta variable se le conoce como **espera ocupada**.

  Al candado que utiliza la espera ocupada se le conoce como **candado de giro**.

ESPERA OCUPADA

```
while(TRUE){
  while(turno!=0);
  region_critica();
  turno = 1;
  region_no_critica();
}
```

```
while(TRUE){
  while(turno!=1);
  region_critica();
  turno = 0;
  region_no_critica();
}
```

SOLUCIÓN DE PETERSON

```
#define TRUE 1
#define FALSE 0
#define N 2
int turno;
int interesado[N];
void entrar_region(int proceso){
   int otro;
   otro = 1 - proceso;
   interesado[proceso] = TRUE;
   turno = proceso;
   while(turno==proceso && interesado[otro]==TRUE);
}
void salir_region(int proceso){
   interesado[proceso] = FALSE;
}
```