dit UPM

Monoalgorítmez

1. Características
2. Modelo funcional
   - Interfaz del usuario
   - Interfaz del programador:
     llamadas al sistema
3. Modelo estructural
   - Estructura del núcleo
   - Proceso de arranque
4. Modelo procesal e implementación

dit UPM

Monoalgorítmez: características

- **UCP:** Algorítmez; **MP:** 64 KB
- **Periféricos:** pantalla, teclado, impresora y dos disquetes de 1.440 KB
- **Sistema operativo de monoprogramación**
- **Sistema de ficheros:**
  - fichero = secuencia de bytes
  - nombre: 1–14 caracteres (más «`A:`» o «`B:`»)
  - permisos de acceso: bits RWX
  - cada fichero *abierto* tiene un *«descriptor»* ($0 \leq d < 24$) y una variable *«posición»*
  - ficheros especiales: `tty`, `lp`, `fd0`, `fd1`
  - directorio de un solo nivel

Monoalgorítmez: modelo funcional

- **Interfaz del usuario:**
  El **intérprete de órdenes** *(shell)* reconoce órdenes *(commands)* como nombres de ficheros que contienen programas ejecutables:
  `cp fich1 fich2; mv fich1 fich2; lpr fich; ls; rm fich; chmod 5 fich...`
- **Interfaz del programador:**
  Las **llamadas al sistema** *(system calls)* se implementan con «macros» predefinidas en el ensamblador:

(0) `CREATE(#N,P)`	(1) `DELETE(#N)`	(2) `OPEN(#N,M)`
(3) `CLOSE(DF)`	(4) `WRITE_A(DF,#Z,#AV)`	(5) `READ_A(DF,#Z,#AV)`
(6) `WAIT_A(#AV)`	(7) `WRITE(DF,#Z,NB)`	(8) `READ(DF,#Z,NB)`
(9) `POS(DF,D,K)`	(10) `CHMOD(#N,P)`	(11) `STAT(#N,#INF)`
(12) `LD_EX(#N,#ARG)`	(13) `LD_OV(#N)`	(14) `EXIT`

Llamadas al sistema mediante macros

- Van acompañadas de parámetros. Por ejemplo:

  `CREATE(#DNOM, #6)`

  crea un fichero cuyo nombre está en una sucesión de bytes a partir del que tiene la etiqueta `DNOM`, le pone RWX = 110 y lo abre en modo escritura
- El ensamblador *expande* cada macro en una secuencia de instrucciones que pasan los parámetros al S.O. Para la anterior:

```
PUSH .0   ; se salvan los     BRK ; llamada mediante interrupción
PUSH .2   ; registros         POP  .6 ; se restauran
PUSH .6   ; R0, R2 y R6       POP  .2 ;  los registros para el
CLR  .0   ; "0" = CREATE      POP  .0 ;  programa de usuario
LD   .2,#DNOM; dirección del nombre
LD.B .6,#6 ; permisos = 6 (lectura y escritura)
```

- El S.O. puede devolver por R13 un resultado (p. ej., el descriptor, en el caso de `CREATE` u `OPEN`) o un código de error (número negativo)

Llamadas para lectura y escritura: `READ_A` y `WRITE_A`

- Fichero abierto previamente con `CREATE` o con `OPEN`, que habrán devuelto el descriptor en R13 (o un error...).
  `tty` abierto siempre: descriptor = 0
- `READ_A(DF, #ZONA, #AV)`: lee n bytes
  `WRITE_A(DF, #ZONA, #AV)`: escribe n bytes
  - `DF`: el descriptor
  - `#ZONA`: dirección de una zona de n bytes en el programa de usuario
  - `#AV`: dirección de un byte de «aviso» en el programa de usuario: el S.O. lo inicializa a 0 y pone 1 al terminar la transferencia
- Tamaño de las zonas:
  - $n = 512$ bytes para los ficheros ordinarios (y para `fd0` y `fd1`)
  - variable para periféricos de caracteres (`tty` y `lp`): cadena termina con H'0D o H'00

`READ_A/WRITE_A/WAIT_A`: ejemplo 1

Envío de mensajes a la pantalla

```
      ---
MENS1 DATA.B "Estoy con cálculo 1",H'0D
MENS2 DATA.B "Estoy con cálculo 2",H'0D
AV    RES.B 1
      WRITE_A(#0,#MENS1,#AV)
      ---   ;  cálculo 1, concurrente con
      ---   ;  la escritura de MENS1
      ---   ;
      WAIT_A(#AV)
      WRITE_A(#0,#MENS2,#AV)
      ---   ;  cálculo 2, concurrente con
      ---   ;  la escritura de MENS2
      WAIT_A(#AV)
      EXIT
```

READ_A/WRITE_A/WAIT_A: ejemplo 2

Lectura (y procesamiento) de diez bloques

```
MODULE  LEE10BL
FROM    PR  IMPORT  PROCESA
EXPORT  ZONA
   DIEZ   EQU       10
   NFICH  DATA.B    "B:abc.txt",0
   ZONA   RES       256
   AVISO  RES.B     1
   DFICH  RES.B     1
   PROG   LD        .0,#DIEZ
          OPEN(#NFICH,#0)
          ST.B      .13,DFICH
   BUCLE  READ_A(DFICH,#ZONA,#AVISO)
          WAIT_A(#AVISO)
          CALL      /PROCESA
          SUB       .0,#1
          BZ        FIN
          BR        BUCLE
   FIN    CLOSE(DFICH)
          EXIT
         END       PROG
```

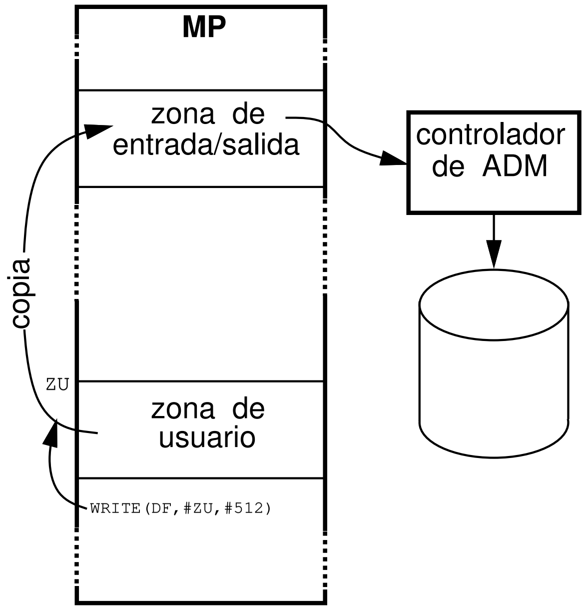

READ_A/WRITE_A/WAIT_A: ejemplo 3

Lectura (y procesamiento solapado) de diez bloques

```
MODULE  LEE10BL
FROM    PR  IMPORT  PROCESA
 DIEZ   EQU       10
 NFICH  DATA.B    "B:FICH",0
 ZONA0  RES.B     512
 ZONA1  RES.B     512
 AV     RES.B     1
 DF     RES.B     1
 PROG   LD.B      .0,#DIEZ
        CLR       .1
        OPEN(#NFICH,#0)
        ST.B      .13,DF
        READ_A(DF,#ZONA0,#AV)
 BUCLE  WAIT_A(#AV)
        SUB.B     .0,#1
        BZ        FIN
        CMP.B     .1,#0
        BZ        LECZ1
        READ_A(DF,#ZONA0,#AV)
        CLR       .1
        LD        .2,#ZONA1
        CALL      /PROCESA
        BR        BUCLE
 LECZ1  READ_A(DF,#ZONA1,#AV)
        LD.B      .1,#1
        LD        .2,#ZONA0
        CALL      /PROCESA
        BR        BUCLE
 FIN    LD        .2,#ZONA1
        CALL      /PROCESA
        CLOSE(DF)
        EXIT
       END       PROG
```

Escritura a través de «buffer»

`WRITE(DF, #ZU, NB)`



Lectura a través de «buffer»

`READ(DF, #ZU, NB)`

- **anticipación** (cuando la lectura termina al final de un bloque)
- Programa para lectura y procesamiento de 10 bloques:

```
MODULE  LEE10BLCB
EXPORT  ZU
FROM    PR  IMPORT  PROCESA        BUCLE  READ(DF,#ZU, #512)
  DIEZ   EQU       10                     CALL      /PROCESA
  NFICH  DATA.B    "B:abc.txt",0          SUB       .0,#1
  ZU     RES       256                    BZ        FIN
  DF     RES.B     1                      BR        BUCLE
  PROG   LD        .0,#DIEZ        FIN    CLOSE(DF)
         OPEN(#NFICH,#0)                  EXIT
         ST.B      .13,DF            END        PROG
```

- con «tty»: **tecleo anticipado** (*type ahead*)

Otras llamadas al sistema

- `POS(DF, DESPL, K)`: cambia el valor de *posición*
  - Si `K` = 0, nuevo valor = `DESPL`
  - Si `K` = 1, nuevo valor = antiguo valor + `DESPL`
  - Si `K` = otro, nuevo valor = longitud fichero + `DESPL`
  - Devuelve nuevo valor en [R13, R12]
- `CHMOD(#DNOM, P)`: pone permisos (0 ≤ `P` ≤ 7)
- `STAT(#DNOM, #INFO)`: devuelve en `INFO` (10 bytes) las informaciones del «inodo»

Carga y ejecución de programas

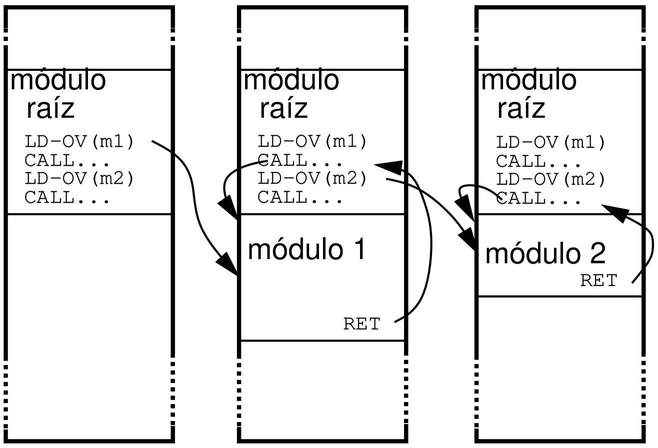
`LD_EX(#DIRPROG, #DIRARG)`

- A continuación (en la MP) del programa *(padre)* se carga otro *(hijo)*: el almacenado en el fichero cuyo nombre está en `DIRPROG`
- A partir de la dirección `DIRARG` puede haber argumentos para el hijo
- Pasa a ejecutarse el hijo
- El padre seguirá su ejecución cuando en el hijo aparezca `EXIT`

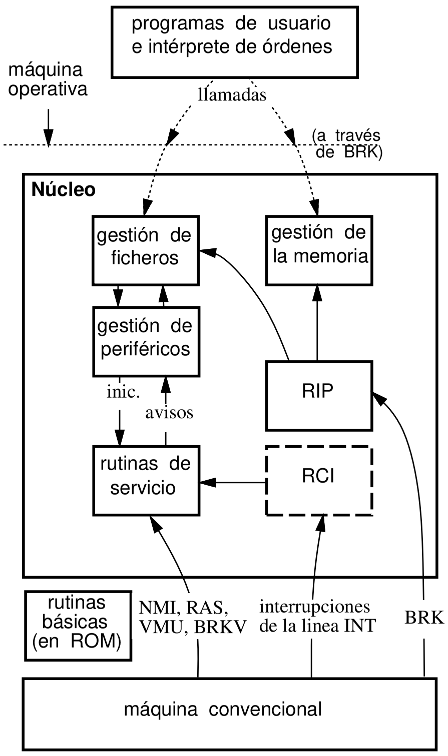
Módulos solapables *(overlays)*

`LD_OV(#DIRPROG)`

Operación obsoleta
para máquinas con poca MP y S.O. sin memoria virtual
(Monoalgorítez, y MS-DOS en sus orígenes)



Modelo estructural de Monoalgorítmez

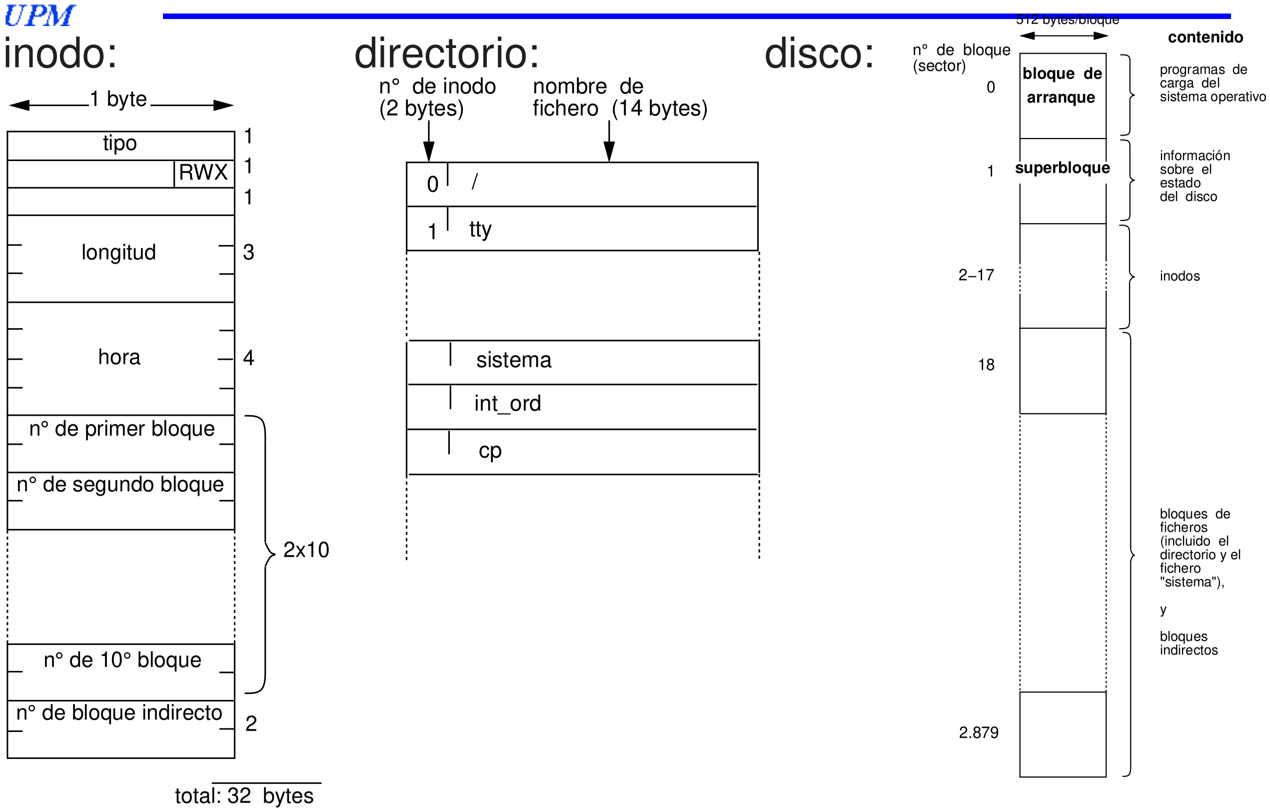


Módulo principal del núcleo:

```
MODULE MP_NUCLEO
    FROM INICIA IMPORT INI_ENT
    FROM RUTINAS IMPORT MENS_ERROR
    EXPORT CARGAINT
         ORG    0
         RES.B  268; reservado para los
             ; vectores de interrupción
PRINC    BR     /INI_ENT
CARGAINT LD_EX(#NOM_INT,#ARG)
         CALL   /MENS_ERROR
         BR     PRINC
NOM_INT  DATA.B "A:int_ord",0
ARG      DATA.B 0 ; int_ord no necesita
                  ; argumentos
         END    PRINC
```

Ensamblado y montado con los demás módulos del núcleo, se almacena en un fichero del disco `A:` *de nombre* «`sistema`»

inodos, directorio y disco



Proceso de arranque

Tres pasos:

1. Ejecución de una rutina en ROM:
   Comprueba el hardware y lee del disco el primer sector (programa «boot») y lo carga en la MP
2. Ejecución del programa «boot»:
   Lee del disco el fichero «`sistema`» y lo carga en la MP
3. Ejecución del progama «`INICIA`» (contenido en «`sistema`»):
   Pone vectores de interrupción, tabla de periféricos, etc., y termina cargando en la MP el intérprete de órdenes

Proceso de arranque (1): la rutina en ROM

- H’0400 → RE; H’FC00 → CP; H’FC00 → PP
- empieza ciclo de instrucción
- rutina de arranque (en ROM, a partir de H’FC00):
  - comprueba recursos de hardware
  - lee primer sector del disco 0 y lo carga en la MP, entre las direcciones H’F600 a H’F7FF
  - `BR /H'F600`
- sigue el proceso de arranque con la ejecución del programa que se acaba de cargar, ya en RAM...

Proceso de arranque (2): el programa «boot»

El programa cargado (procedente del sector 0 del disco)

- lee el superbloque (contiene el formato), el primer inodo (corresponde al directorio), busca en el directorio el fichero «`sistema`» y carga su inodo en la MP
- carga los bloques del fichero `sistema` a partir de la dirección 0 de la MP
- dirección de comienzo de ejecución (en el último registro del módulo de carga; corresponde a `PRINC` ≡ D’268 del módulo principal del núcleo)
- `BR /268`: con esto empieza la ejecución del módulo principal, que llama a `INICIA`

Proceso de arranque (3): el programa `INICIA`

- pone vectores de interrupción para los periféricos conectados y les da permiso de interrupción
- rellena la tabla de periféricos
- carga en la MP el inodo de `tty`
- carga en la MP el inodo del directorio
- carga en la MP los superbloques de los discos
- `LD.E #H'0100`: modo usuario, permiso de interrupciones
- `BR /CARGAINT`: `CARGAINT` está importada del módulo principal
- el módulo principal carga el intérprete de órdenes con `LD_EX(#NOM_INT,#ARG)`

Monoalgorítmez: el intérprete de órdenes

```
MODULE INT_ORD
FROM RUTINAS
  IMPORT ANALIZA, COMPLETA,
         MENS_ERROR
EXPORT ORDEN, NOMBRE, DISCO
 ORDEN    RES.B   80
 NOMBRE   RES.B   16
 INVIT    DATA    "Mono>", 0
 DISCO    DATA    "A:"; (inicial)
```

```
INT_ORD WRITE(#0,#INVIT,#80)
        READ(#0,#ORDEN,#80)
        CALL      /ANALIZA
        CMP.B     .13,#0
        BN        INT_ORD
        CALL      /COMPLETA
        CMP.B     .13,#2
        BNN       INT_ORD
        LD_EX(#NOMBRE,#ORDEN)
        CMP.B     .13,#0
        BNN       INT_ORD
        CALL      /MENS_ERROR
        BR        INT_ORD
       END     INT_ORD
```