

Projecte de Sistemes Operatius

FIB 2011-2012

Professors: Yolanda Becerra, Julita Corbalán, Juan José Costa, Jordi Garcia, Marisa Gil,
Jordi Guitart, Gemma Reig, Amador Millán

INTRODUCCIÓ AL PROJECTE ZEOS

El que es pretén en aquest primer projecte és implementar part d'un S.O. anomenat ZeOS, inspirat en un sistema Linux 2.4. El sistema operatiu ZeOS fou desenvolupat inicialment per un grup d'estudiants de la F.I.B., amb el suport de professors del departament d'AC. Per tant, qualsevol de vosaltres pot anar enriquint aquest S.O. i anar fent les aportacions que cregui oportunes.

Es partirà d'una implementació bàsica, a la qual s'aniran afegint diverses funcionalitats. Aquesta implementació bàsica realitza el procés d'arrencada del S.O. (boot) i inicialitza totes les estructures de dades necessàries pel correcte funcionament. Per tal d'afegir noves funcionalitats al sistema, s'haurà de programar tant a alt nivell (llenguatge C) com a baix nivell (llenguatge ensamblador).

Primerament s'haurà de treballar una part imprescindible en qualsevol S.O., el procés d'arrencada o boot. A continuació es gestionaran algunes de les interrupcions bàsiques i captura d'excepcions. Tot seguit es treballarà la gestió de processos dins d'un S.O. (aquesta part serà la més extensa i interessant de l'assignatura). Finalment s'acabarà implementant algunes parts de l'entrada/sortida, així es podrà interactuar amb el sistema.

Durant aquest primer projecte heu de tenir clars conceptes de les següents assignatures:

- SO: Mecanismes d'entrada al sistema (excepcions/interrupcions/crides a sistema). Gestió de processos (estructures de dades/algorismes/polítiques de planificació/canvi de context/crides a sistema relacionades). Gestió de entrada/sortida (Dispositius/canals/crides a sistema relacionades).
- EC1: Subrutines i excepcions
- EC2: Gestió entrada/sortida i gestió de memòria.

1. Que hi ha en aquest document?

El que trobareu en aquest document en primer lloc és una sessió introductòria on l'objectiu és aprendre a utilitzar les eines que necessiteu per fer el projecte. L'emulador, les comandes del debugger per veure les adreces del codi, etc. És el que anomenem P0. A continuació teniu la descripció de les diferents parts del projecte ZeOS. En total, consta de tres parts, cadascuna d'elles inclou una entrega específica al Racó, una entrevista amb el vostre tutor i la realització d'un qüestionari. A l'entrega 1.1 es tracten els mètodes d'entrada al kernel (interrupcions, excepcions i crides a sistema). L'entrega 1.2 tracta principalment de la gestió de processos. I finalment l'entrega 1.3 correspon a la gestió de l'entrada de dades.

2. Que heu de fer al laboratori?

Abans d'anar als laboratoris heu de fer primer una lectura de tota la entrega per tenir una visió global del que es demana. Llavors feu un disseny de la feina a fer (noves dades, funcions, algorismes, etc). Finalment implementeu el vostre disseny i valideu el vostre codi amb els jocs de proves necessaris.

AGRAÏMENTS: *Aquest document ha estat preparat amb la participació dels següents professors que han impartit PROSO en altres cursos: Ruben González, Silvia Llorente i Pablo Chacín. I com a col·laboradors: A. Bartra, M. Muntanyà i O. Nieto.*

P0: SESSIÓ INTRODUCTÒRIA

Els objectius d'aquesta sessió són:

- Que us familiaritzeu amb l'entorn de treball.
- Que aprengueu les eines amb les quals treballarem.
- Que comenceu a analitzar i modificar el codi de ZeOS.
- Que apreneu les comandes bàsiques de Bochs.
- Que recordeu alguns conceptes que necessitareu per fer el projecte 1.

El que heu de fer és seguir l'enunciat d'aquesta sessió i anar responnent les qüestions que anireu trobant. Al final de l'enunciat hi ha algunes preguntes addicionals que també heu de saber respondre, i que es poden demanar en alguna de les entrevistes.

1. Entorn de treball

Quan estem desenvolupant un S.O. nou necessitem l'entorn de treball que ens proporciona un S.O. complet. En el nostre cas desenvoluparem ZeOS a les aules de S.O. que tenen la següent configuració:

- S.O. : Ubuntu 6.06
- Compilador: gcc 4.0.3
- Emulador: Bochs versió 2.3.

Podeu treballar des de casa, només heu d'anar amb compte amb les versions de S.O., compilador etc.

Un cop generat ZeOS podríem rebotar la màquina amb el nou sistema (després de copiar-lo en un disquet). No obstant, tenint en compte que haurem de recompilar i carregar moltes vegades el sistema el que farem perquè el mecanisme sigui més ràpid és executar-lo sobre un emulador de l'arquitectura (Bochs), de forma que si ens hem equivocat no haguem de rebotar la màquina sinó només arrencar de nou l'emulador. Aquí teniu una llista de detalls importants relacionats amb l'entorn de treball.

- L'entorn de desenvolupament de la pràctica és el *Ubuntu* i la màquina virtual *Bochs* (bochs.sourceforge.net), de lliure distribució, per si voleu fer la instal·lació a casa. **Amb qualsevol altra distribució de Linux heu de tenir en compte que potser haureu d'instal·lar el paquet de desenvolupament *dev86*, ja que habitualment no està instal·lat per defecte.**
- En iniciar els ordinadors us haureu d'identificar (username y password de la FIB). Un cop identificats us sortirà un menú per carregar el S.O. Escolliu Ubuntu 6.06 i premeu *Enter*.
- El sistema té dos usuaris: *alumne* (pass: sistemes) i *root* (pass:crsl2007). Pel projecte 1 us recomanem que utilitzeu el usuari alumne.
- Cada alumne disposa al servidor *albanta* d'un compte amb 100 Mb d'espai, amb el mateix username i password que teniu a la resta de màquines de la FIB. **És important que salveu el treball de cada sessió en el vostre compte del servidor per tal de poder-lo restaurar en la següent sessió, ja que no es manté còpia local al PC on trebal·leu.**

- Hi podeu pujar (comanda: put "nom_fitxer") i baixar (comanda: get "nom_fitxer") els vostres fitxers entrant per sftp (comanda: sftp username@albanta, on username és el vostre nom d'usuari dels sistemes de la FIB). Us recordem que també podeu accedir a aquest servidor des de casa (usant el SocksCap) <http://www.fib.upc.edu/ca/LCFIB/Serveis/Remota/acces-proxy.html>.
- Des del PC us podeu connectar a la pàgina web de l'assignatura per a consultar la documentació que necessiteu i per agafar el fitxers de treball: <http://docencia.ac.upc.edu/FIB/PROSO>.
- Des del PC també us podeu connectar al servidor *albanta* (amb ssh i sftp) per guardar i restaurar els vostres fitxers de treball. Recordeu que només heu d'utilitzar albanta per guardar els vostres codis, no es pot utilitzar per fer desenvolupament perquè no funcionarà.
- Us serà útil comprimir i descomprimir els fitxers per moure'ls dels PC's a albanta. Les comandes usades són:
 - ✓ Comprimir (Compress):
tar czvf nom_fitxer.tar.gz llista_fitxers_a_comprimir
 - ✓ Descomprimir (eXtract): tar xzvf nom_fitxer.tar.gz

Amb aquestes comandes fem el tar i el gzip a la vegada. No tots els shells suporten fer-ho en un sol pas. En aquest cas, les comandes serien:

```
Comprimir: tar cvf fitxer.tar llista_fitxers_a_unir i després gzip fitxer.tar
Descomprimir: gunzip fitxer.tar.gz i després tar xvf fitxer.tar
```

- Als PC's (Ubuntu) hi trobareu diversos editors per treballar (GVim, emacs, nedit ...).
- Comproveu que podeu accedir als diferents comptes de les diferents màquines
- Descarregueu al vostre PC de treball el fitxer zeos.tar.gz (http://docencia.ac.upc.edu/FIB/PROSO/index_files/zeos.tar.gz) .

1.1. Coneixements previs

Durant aquest primer projecte se suposa que teniu coneixements adquirits d'altres assignatures però també que sou capaços de treballar en determinats entorns. En concret se suposa que sou capaços de:

- Fer programes de certa complexitat en llenguatge C.
- Fer programes de certa complexitat en llenguatge ensamblador de Linux i386.
- Afegir codi en ensamblador a un fitxer en C.
- Modificar un Makefile per afegir noves regles.

Si no teniu algun d'aquests coneixements hauríeu de buscar informació addicional a la que es dona a l'assignatura. La informació que es dona en els apèndix (que podeu trobar a la pàgina web de l'assignatura) us pot ser d'utilitat.

2. Introducció a ZeOS

La construcció del sistema operatiu és similar a la construcció d'un executable convencional. A continuació veurem com es construeix el sistema operatiu a partir del codi font. La construcció és molt similar a la que es fa servir per construir el sistema operatiu Linux. De fet, aquesta documentació pot servir, excepte uns canvis menors,

per explicar com es construeix Linux. Per construir ràpidament el ZeOS només cal teclejar make al directori amb els fitxers i el fitxer de Makefile guiarà el procés. No hauria d'haver cap problema.

2.1. Contingut

El contingut dels fitxers es pot dividir en els següents grups segons la seva extensió:

1. Com veiem tenim els fitxers de codi font escrits en llenguatge C, extensió `.c`.
2. A més hi ha dos fitxers amb extensió `.S` que són escrits en llenguatge ensamblador del Intel 80386 amb **sentències de preprocessament**. Els fitxers `.c` i `.S` són els únics que incorporen codi al sistema operatiu.
3. També tenim el fitxer de Makefile que guiarà la construcció del sistema operatiu. És l'únic fitxer sense extensió.
4. Veiem, a més, uns fitxers amb extensió `.lds` que són guions d'enllaçat usats per l'enllaçador ld.
5. I finalment en un directori a part (igual que a Linux) tots els fitxers de capçalera (extensió `.h`).

Normalment per a cada fitxer de codi font en llenguatge C hi ha un fitxer de capçalera amb extensió `.h` que incorpora les capçaleres de totes les funcions incloses en aquell fitxer per si des d'altres fitxers `.c` també es volen cridar aquestes funcions.

Tots els fitxers que incorporen codi al ZeOS, és a dir els fitxers `.c` i `.S`, es poden dividir en diversos grups segons la disposició final de cada fitxer en la imatge del sistema operatiu.

A la Figura 1 es mostra l'aspecte final que ha de tenir la imatge del sistema operatiu una vegada construïda¹.

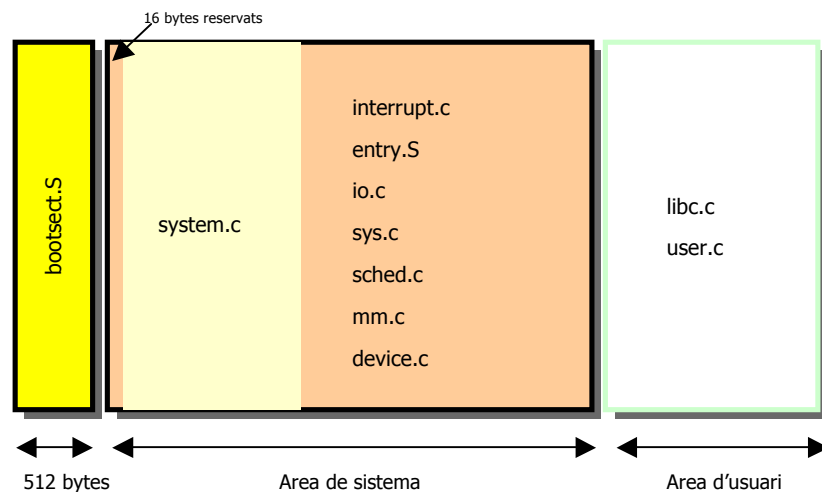


Figura 1. Esquema de ZeOS

¹ Degut als constants canvis que es fan, és possible que el nombre i nom dels fitxers que us donem no coincideixi exactament amb els del dibuix.

Si ens fixem en aquesta figura tenim 3 grans blocs:

El **bloc del sector d'arrencada**. Aquest bloc només està compost per un sol fitxer: bootsect.S i ha d'ocupar exactament 512 bytes ja que ha de cabre al sector 0 d'un disquet convencional.

El **bloc de l'àrea de sistema**. Aquest bloc és el més voluminós i conté els fitxers principals del sistema operatiu. Aquest bloc es col·loca en una part de memòria que garanteix que s'executa en nivell de privilegi màxim del processador.

El **bloc de l'àrea d'usuari**. Aquest bloc conté un programa d'usuari de prova per comprovar que les crides a sistema funcionen correctament. El codi que hi ha dins, el processador l'executa en nivell de privilegi mínim.

Un es pot preguntar per què el programa de prova d'usuari està junt al sistema operatiu en el mateix fitxer quan això mai passa a la realitat. Sempre tenim els programes d'usuari en un disc amb un sistema de fitxers i el programa està en un directori del sistema de fitxers. La qüestió és que per simplificació, no hi ha sistema de fitxers, ni tampoc carregador d'executables. Llavors per executar el programa de prova d'usuari només el posem en un àrea de memòria amb privilegis d'usuari i li transferim l'execució.

```
ENTRY(main)
SECTIONS
{
    . = 0x10000; /* Començem a las 0x10000 */
    .text.main : { BYTE(24); /* reservats per emmagatzemar les dades de gestió del codi d'usuari */
                 *(.text.main) }
    .text : { *(.text) }
    .rodata : { *(.rodata) }
    .data : { *(.data) } /* Dades inicialitzades */
    .bss : { *(.bss) } /* Dades no inicialitzades */
    . = ALIGN(4096); /* Vector de task_structs, alineats a pàgina */
    .data.task : { *(.data.task) }
}

```

Codi 1: Codi del fitxer system.lds

El Codi 1 ens mostra com a exemple el codi del fitxer system.lds que indica a l'enllaçador de C on ha de col·locar a memòria les seccions que té un fitxer executable. En aquest cas les més importants són el punt d'inici (adreça 0x10000), amb la directiva BYTE reservem 24 bytes al fitxer per guardar posteriorment la mida del codi d'usuari i altres dades necessàries per a gestionar el codi d'usuari. També és important destacar la última secció (.data.task) que podeu trobar al fitxer **sched.c**.

```
union task_union task[NR_TASKS] __attribute__((__section__(".data.task")));
```

Codi 2: Secció .data.task (sched.c)

2.2. Arrencada

Aquests tres blocs es concatenen un darrere de l'altre per formar un únic fitxer binari amb el contingut del sistema operatiu ZeOS. Aquest fitxer si el copiem de forma binària

en un disquet sense format començant pel sector 0 podem disposar de la imatge autoarrencable del sistema operatiu en disquet. El podem provar en qualsevol màquina que tingui disquetera i pugui arrencar des de disquet.

El funcionament és senzill, quan encenem l'ordinador el primer que s'executa, molt abans que el sistema operatiu és un programa que resideix a un àrea de memòria de només lectura (ROM) anomenada BIOS. Aquest programa comprova que l'estat del PC sigui correcte, tothom pot veure com es comprova la memòria, els discs, etc... A continuació localitza el dispositiu d'arrencada preconfigurat i intenta accedir-hi. El que fa és carregar el sector 0 d'aquell dispositiu a memòria, sigui un disquet o un disc dur.

Però és clar, en 512 bytes que ocupa el primer sector que carrega la BIOS no és suficient per emmagatzemar tot el sistema operatiu. Només podem emmagatzemar el carregador. Com ZeOS és molt més simple, el carregador ja cap en 512 bytes y és el BIOS qui el carrega complert a memòria.

Una vegada copiats els 512 bytes del sector d'arrencada per la BIOS a memòria, aquest li transfereix l'execució al primer byte d'aquest petit bloc. En aquest punt comença veritablement a executar-se el sistema operatiu. La tasca principal del codi del sector d'arrencada és acabar de carregar el que queda de la imatge del sistema operatiu que encara està al disquet i posar-la a memòria.

2.3. Muntatge de la imatge

Per explicar la construcció del sistema operatiu començarem pel final. Sabem que hem d'obtenir el mapa de la Figura 1 a partir dels tres blocs: sector d'arrencada, sistema i usuari. Considerarem que aquests tres blocs ja els tenim i veurem el pas d'ajuntar-los en un sol fitxer. Aquest procés es realitza fent servir el Makefile que us donem.

El programa que s'encarrega d'ajuntar-los és el fitxer `build.c`. Si el programa `make` detecta que la utilitat `build` no està construïda la compila:

```
$ gcc -Wall -Wstrict-prototypes -o build build.c
```

Codi 3: Construcció del fitxer `build`

Ara, per enganxar els tres blocs un a darrera de l'altre només és necessari executar la següent comanda (el `make` ens ho fa per nosaltres):

```
./build bootsect system.out user.out > zeos.bin  
Boot sector 512 bytes.  
System is 24 kB  
User is 1 kB  
Image is 25 kB
```

Codi 4: Construcció de la imatge de ZeOS (`zeos.bin`)

on `bootsect` és el contingut binari del sector d'arrencada, `system.out` és el contingut binari de l'àrea de sistema i `user.out` és el contingut binari de l'àrea d'usuari i `zeos.bin` és el binari del OS.

El que fa el `build` és mirar les mides dels blocs, sumar-les i escriure el valor en una posició específica del sector d'arrencada, concretament als bytes 500 i 501 marcats al `bootsect.S` com a `sysize`.

El codi del sistema operatiu fa una cosa semblant per carregar el codi d'usuari. Ha de saber, ja a memòria, on comença el codi d'usuari i on acaba. És per això que a la

Figura 1 veiem 24 bytes reservats a l'inici del bloc de sistema. Al principi estan buits però la utilitat build hi escriu, entre d'altres coses, on comença el codi d'usuari i quant ocupa.

2.4. Què us donem?

- Estructura de fitxers de ZeOS.
- Makefile.
- Macros en assemblador.
- Boot bàsic del sistema.
- Inicialització de la GDT, TSS i taula de pàgines.
- Funcions per inicialitzar la IDT.

3. Bochs

3.1. Introducció

Bochs és un emulador de PC Intel x86 programat en C++. Fou creat vora l'any 1994. Inicialment no era lliure, fins que Mandrake el va comprar i el va posar amb la llicència GNU LGPL. És un emulador una mica lent (tot i que a les pràctiques no es notarà) però, com a contrapartida, és molt segur. Aquest curs utilitzarem la **versió 2.3**.

3.2. Depuració de codi amb Bochs

Per a depurar la imatge és necessari controlar l'execució amb un debugger. Això és possible si, quan es compila Bochs, s'activen les opcions de depuració (això ja està fet a la instal·lació que teniu als laboratoris).

Bochs ens ofereix dues opcions per a la depuració de codi: utilitzar un debugger extern (com per exemple, el gdb) o utilitzar el debugger intern, que es suministra com a part de l'emulador. Totes dues opcions són exclusives (un mateix executable de Bochs no pot suportar els dos debuggers alhora). Al laboratori teniu instal·lades les dos versions, de manera que podeu escollir el debugger que us sigui més fàcil d'utilitzar.

L'opció d'utilitzar el debugger no està activa per defecte en el paquet binari estàndard del Bochs. Per activar la utilització del GDB cal recompilar el Bochs, després d'haver executat el configure amb l'argument `--enable-gdb-stub`:

```
$ ./configure --enable-gdb-stub
```

D'altra banda, si el que volem és activar la utilització del debugger intern cal configurar la compilació de Bochs usant:

```
$ ./configure --enable-debug --enable-disasm
```

3.3. Execució

L'executable de Bochs compilat per a utilitzar el debugger extern el trobareu a **/usr/local/bin/bochs**. I l'executable de Bochs compilat per a utilitzar el debugger intern està a **/usr/local/bin/bochs_nogdb**.

3.4. Configuració de l'execució de Bochs

El fitxer per a configurar l'execució Bochs és el `.bochsrc`. Ens concentrarem en dos aspectes:

- Localització de la imatge a carregar: A la línia 134 trobareu la línia de defineix que bochs botarà del vostre fitxer `zeos.bin`. El nom del fitxer ha de correspondre amb el path on tingueu el fitxer `zeos.bin`. Si treballeu a casa i modifiqueu el path l'haureu de modificar. Ara mateix està configurat per executar la imatge de ZeOS que teniu al directori on treballeu.

```
floppya: 1_44=./zeos.bin, status=inserted
```

- Configuració del debugger extern: Al final del fitxer hem afegit una línia per a configurar la utilització del debugger GDB. Si Bochs ha estat compilat per a utilitzar el debugger extern i no troba aquesta línia s'executarà sense cap suport per a debugging.

```
gdbstub: enabled=1, port=1234, text_base=0, data_base=0, bss_base=0
```

Cal remarcar que aquesta línia només té sentit per a la versió de bochs compilada per a utilitzar el debugger extern (al laboratori, **/usr/local/bin/bochs**). Per defecte aquesta línia està comentada, és a dir, el fitxer `.bochsrc` és vàlid per a les dues versions de bochs instal·lades al laboratori. Si voleu executar la vostra imatge utilitzant el `gdb` heu de descomentar aquesta línia. Un cop tingueu al `.bochsrc` la configuració que us interessa ja podeu executar Bochs per a carregar la vostra imatge. Al laboratori podeu executar la versió de Bochs compilada per a utilitzar el debugger extern utilitzant la comanda:

```
$ bochs -q
```

O per a executar la versió compilada per a utilitzar el debugger intern:

```
$ bochs_nogdb -q
```

3.5. Control d'execució amb debugger extern GDB

Per a utilitzar el GDB per a depurar la vostra imatge (el debugger GNU), cal seguir els següents passos:

- Compilar el codi de ZeOS amb símbols de debug, fent servir el flag "-g" en el CC. El Makefile que us proporcionem ja ho fa.
- Comprovar que al fitxer `.bochsrc` teniu activada la línia de configuració del `gdbstub`

- Executar la versió de Bochs amb el debugger extern activat:

```
$ bochs -q
```

La màquina virtual Bochs s'inicialitzarà i es quedarà esperant una connexió des del GDB (vegeu Figura 2)

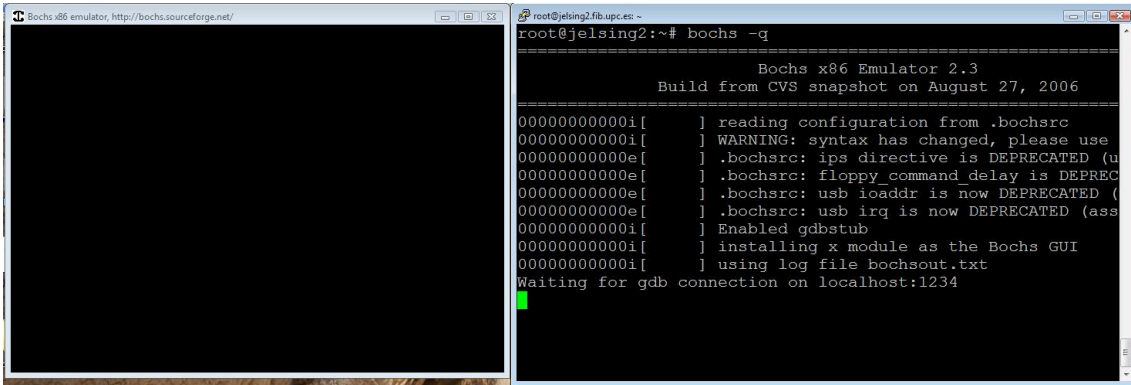


Figura 2. Finestra d'emulació (esquerra) i comandes (dreta) que apareixen al arrancar Bochs.

- Després d'executar el Bochs, en un nou terminal, executar el GDB o un front-end al GDB (com, per exemple, el ddd) amb l'objecte del programa que es voldepurar com a argument. En el nostre cas, usarem el binari 'system'

```
$ gdb system
```

- Connectar el GDB amb el Bochs executant aquesta comanda en el GDB (el port ha de ser el mateix que heu posat en el fitxer .bochsrc)

```
(gdb) target remote localhost:1234
```

- Afegir els símbols d'usuari

```
(gdb) add-symbol-file user
```

- A partir d'aquest punt ja es poden posar breakpoints, continuar l'execució, etc. A la següent URL podeu trobar una guia de consulta ràpida de les principals comandes del GDB: <http://refcards.com/docs/peschr/gdb/gdb-refcard-a4.pdf>

3.6. Control d'execució amb debugger intern

Per a utilitzar el debugger intern de Bochs el primer que heu de fer és comprovar que al fitxer .bochsrc no apareix la línia de configuració del gdb (gdbstub) i després executar la comanda del bochs sense gdb:

```
$ bochs _nogdb -q
```

Un cop executada aquesta versió de Bochs, us sortiran dues finestres (vegeu Figura 3. Al **arrancar Bochs veureu la finestra de emulació (dreta) i la de comandes(esquerra)**

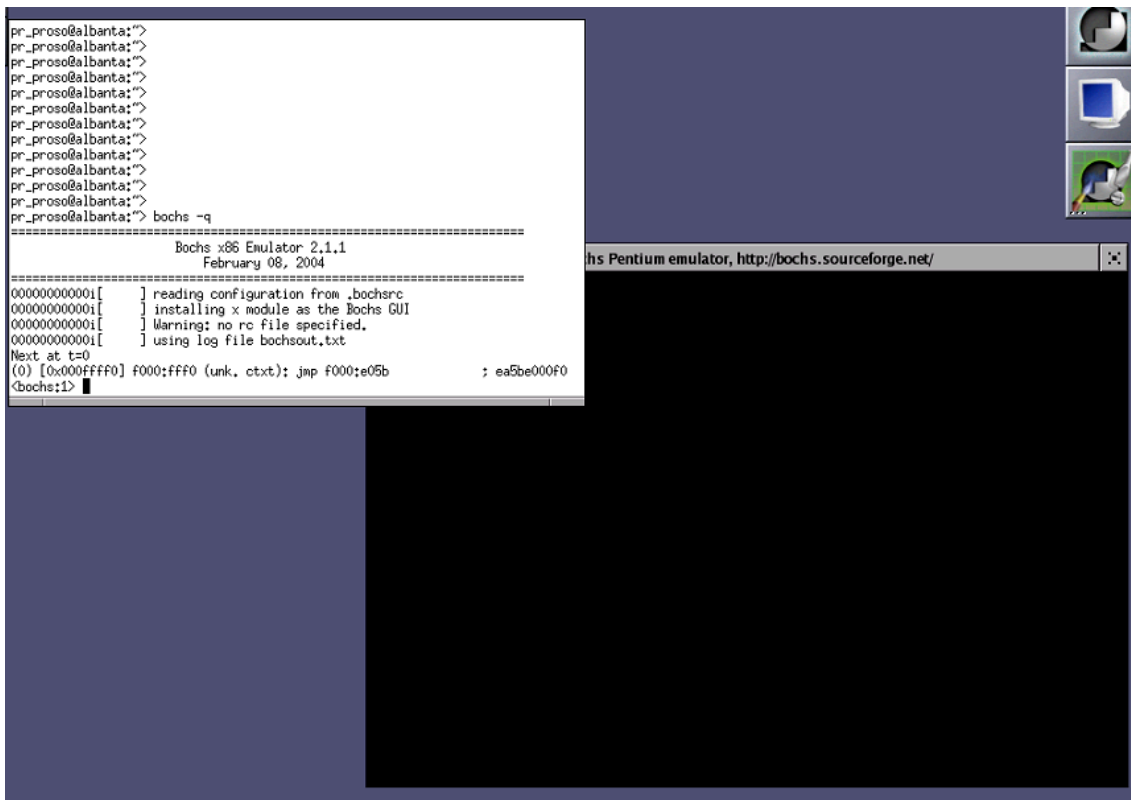


Figura 3. Al arrancar Bochs veureu la finestra de emulació (dreta) i la de comandes(esquerra)

A la finestra de comandes tindreu un prompt on podeu indicar comandes a executar. Algunes de les comandes que podeu executar són (consulteu la url <http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html> per a més informació):

- **c**: executar la imatge fins al final o fins al breakpoint més proper (si n'heu posat). En qualsevol moment es pot prémer Ctrl-C per a finalitzar l'execució de la imatge i tornar al prompt.
- **s [num]**: (step) executar num línies. En cas de no posar num, se n'executarà una.
- **n [num]**: (next) executar num línies. Si es troba una crida a funció, l'executa sencera i continua a la següent línia.
- **b address**: (breakpoint) posa un breakpoint a adreça de memòria indicada. L'adreça es pot posar en decimal (123459), octal (0123456) o, més comú, en hexadecimal (0x123abc). (Per exemple si poseu "b 0x100000" poseu un breakpoint a la primera línia de codi del procés d'usuari.
- **q**: sortir de bochs
- **info r o info registers**: mostra el contingut dels registres del mode en que estiguem (usuari o sistema).
- **print-stack [num_words]**: imprimeix num_words del top de la pila. En cas de no posar-ne, el valor per defecte és 16. Sols és fiable en cas que estiguem en mode sistema, quan l'adreça base del segment de pila és 0.
- **help**: ens mostra les diverses comandes que podem fer. Compilació, muntatge i generació d'imatges

Si voleu comprovar que la imatge de ZeOS que genereu és bootable podeu seguir aquests passos.

- Compilar els fitxers font bàsics que us passem i generar una imatge de ZeOS bootable des de disquet: **make disk**. Per fer el make disk primer heu de muntar la disquetera i inserir un disquet.
 - Observeu en el *Makefile* com es fa per generar una imatge bootable de disquet.
- Bootar des del disquet per provar la imatge generada.

Recordeu que cal salvar el treball realitzat a cada sessió en el vostre compte del servidor *albanta* per tal de poder-lo restaurar en la següent sessió.

3.7. Links relacionats

Teniu més informació sobre el debugger GDB a:

<http://www.cs.cmu.edu/~gilpin/tutorial/>

Teniu més informació sobre el debugger intern de Bochs a:

<http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html>

<http://www.cs.umd.edu/~hollings/cs412/s02/debugger.html>

4. Treball a realitzar

El codi del fitxer user.c serà específic per a cada entrega. Això vol dir que, per la resta d'entregues haureu d'actualitzar el contingut d'aquest exemple que us adjuntem per fer proves amb el debugger.

Seria bo mantenir sempre el fitxer user.c amb el codi estrictament necessari (o sigui, esborreu el codi que no s'utilitzi).

4.1. Modificació del codi d'*usuari*

- Modifiqueu el main() del fitxer user.c de la següent manera:

```
int __attribute__((__section__(".text.main")))
main()
{
    long count, acum;
    count = 75;
    acum = 0;
    acum = outer(count);
    while(1);
    return 0;
}
```

- Afegiu al fitxer user.c les funcions següents:

```

long inner(long n)
{
    int i;
    long suma;
    suma = 0;
    for (i=0; i<n; i++) suma = suma + i;
    return suma;
}

long outer(long n)
{
    int i;
    long acum;
    acum = 0;
    for (i=0; i<n; i++) acum = acum + inner(i);
    return acum;
}

int __attribute__((__section__(".text.main")))
main(void)
....

```

- Genereu una nova imatge del ZeOS (comproveu si surt algun warning al fer la compilació. Arregleu-los en cas de que apareguin).
- Per a interpretar el funcionament del vostre codi, us serà útil conèixer la situació a memòria de les diferents parts de l'imatge (codi, dades). **Mireu al man el funcionament de les comandes objdump i nm.**

```

00010000 <main-0x4>:
 10000:  10 00          adc  %al,(%eax)
  ...
00010004 <main>:
 10004:  55            push %ebp
 10005:  89 e5         mov  %esp,%ebp
 10007:  83 ec 08     sub  $0x8,%esp
 1000a:  83 e4 f0     and  $0xfffff0,%esp
 1000d:  6a 00       push $0x0
 1000f:  9d          popf
 10010:  ba 18 00 00 00  mov  $0x18,%edx
 10015:  b8 18 00 00 00  mov  $0x18,%eax
 1001a:  fc          cld
 1001b:  8e da       mov  %edx,%ds
 1001d:  8e c2       mov  %edx,%es
 1001f:  8e e2       mov  %edx,%fs
  ...

```

Codi 5: Exemple de informació que proporciona la comanda "objdump -d system"

```

0001042c T set_ss_pag
00010224 T set_task_reg
000102b8 T setTrapHandler
000100f8 T setTSS
00011000 D task
0001c000 D taula_pagusr
0001d020 B tss
00010538 D usr_main
0001d8a0 B x
00010544 D y

```

Codi 6: Exemple de sortida que genera la comanda "nm system"

En aquests exemples podem veure que amb la comanda **objdump** podem veure exactament el codi que ha generat el compilador i les adreces de les funcions, molt útil de cara a posar un *breakpoint* (Codi 5). Amb la comanda **nm** podem veure on estan ubicades les variables i funcions. En aquest cas podem destacar la **taula de processos** a l'adreça **0x11000** (Codi 6).

- Executeu la comanda "**objdump -d user.o**" per a veure el codi en llenguatge ensamblador del fitxer user.c. Observeu que les adreces dels símbols que surten són PIC (Position Independent Code) i comencen a partir de zero.
- Ara executeu "**objdump -d user**". Coincideixen les adreces? Per què?
- Executeu tot el programa (comanda `continue`). Per recuperar el control dins el debugger cal fer Ctrl + C. Després de fer-ho comproveu quina línia de codi estàveu executant.
- Torneu a executar el programa pas a pas. Per això, afegiu un BREAKPOINT a la primera línia de codi de la rutina `inner()` i continueu l'execució amb `continue`. Comproveu que en comptes d'executar el programa sencer, l'execució s'atura en la línia de codi que hem marcat amb un BREAKPOINT. Continueu avançant en l'execució fent una sola línia de codi cada cop (comandes `step` i/o `next`²).

Nota: afegir un BREAKPOINT és tot sovint la única manera pràctica d'arribar fins a una subrutina. En aquest cas, el nombre de comandes STEP que caldrien fins arribar a la rutina `inner()` és molt elevat. Durant el projecte amb ZeOS, un BREAKPOINT és la única manera de fer que l'execució arribi fins a l'interior d'una rutina de servei a la interrupció (RSI).

- Examineu el contingut dels registres.
- Quina comanda del debugger permet veure el contingut de memòria? Us serveix per veure el contingut de la variable `acum` del user.c?

4.2. Utilització de l'ensamblador "in-line"

Afegiu al user.c una funció amb la següent capçalera:

```
int add(int par1,int par2);
```

² Hem detectat que després de fer un Ctrl + C, Bochs no pot continuar l'execució amb les instruccions del debugger `step` o `next`, cal usar `continue`, `stepi` o `nexti`.

Aquesta funció haurà de fer (en ensamblador) l'equivalent a:

```
return par1+par2;
```

Consulteu l'annex per veure com s'afegeix codi en ensamblador en un programa C. Per fer la suma heu d'accedir als paràmetres (recordeu que estan a la pila), sumar-los, y retornar el resultat. Farem dues versions:

- Tot el codi de la funció estarà en ensamblador. Això vol dir que heu de recordar com es retorna un valor en ensamblador (en quin registre heu de deixar el resultat?). No podeu fer servir els paràmetres d'entrada i sortida de l'ensamblador in-line.
- Definirem una variable local y el resultat de la suma el deixarem en aquesta variable. Després farem un return de la variable. Consulteu a l'annex com es fa per modificar una variable declarada des de C en un codi ensamblador. En aquest cas, heu de fer servir els paràmetres d'entrada i sortida de l'ensamblador in-line.

Afegiu una crida a la funció `add()` al main del `user.c`, deixant el resultat de la suma en una variable local.

- Comproveu amb el debugger que les dues versions funcionen correctament.
- On està la variable resultat de la suma (pila, registre, memòria)?

4.3. Modificació del codi de *sistema*

En aquesta part es demana millorar la funció interna del kernel per escriure per pantalla (`printc`).

Modificar la funció per a escriure caràcters per la pantalla, `printc`. Cal:

- Controlar `\n`: Si a l'*string* d'entrada es troba el caràcter `\n` s'interpretarà com un salt de línia i per tant cal incrementar el número de fila i posar a zero el número de columna

Modifiqueu els espais dels missatges al codi de boot de ZeOS (fitxer `system.c`) per veure que funciona correctament.

4.4. Qüestions addicionals

Intenteu comprendre les parts rellevants del mecanisme de boot (`bootsect.S`) i de la inicialització del sistema (`system.c`). Podeu fer-vos un esquema de l'estructura de fitxers i del seu contingut. Hauríeu de ser capaços de respondre aquestes preguntes:

1. Diferències principals entre mode real i mode protegit.
2. En quin mode bota el sistema?
3. Què són els nivells de privilegi d'execució?
4. Què és el bootloader? A quina part del disc es troba?
5. Què és la BIOS? Per a què serveix?
6. Què és la IDT? Per a què serveix?
7. Quins passos té el Makefile de ZeOS? Per a què serveixen les diferents opcions de compilació i enllaçat?
8. Opcionalment, podeu fer que, si s'arriba a les coordenades màximes de la pantalla es faci un scroll

ENTREGA 1.1: MECANISMES D'ENTRADA AL SISTEMA

1. Introducció

Per executar codi del sistema operatiu s'ha de fer mitjançant el mecanisme d'interrupcions. Una interrupció és un event que trenca l'execució seqüencial del programa. Les interrupcions es classifiquen en síncrones i asíncrones. Síncrones són aquelles que provoca la CPU al final de l'execució d'una instrucció. Les asíncrones són generades per altres dispositius hardware.

A partir d'aquest moment anomenarem **excepcions** les interrupcions síncrones i **interrupcions** les interrupcions asíncrones, però ambdues segueixen el mateix mecanisme.

Cada interrupció s'identifica amb un nombre entre 0 i 255. Les interrupcions s'associen a una determinada funció mitjançant una taula de sistema anomenada IDT (Interrupt Descriptor Table). Cada entrada d'aquesta taula conté la informació necessària per descriure què s'ha de fer quan es produeix la interrupció. Entre altres dades hi ha l'adreça de la rutina a executar (que anomenarem **handler**) o el nivell de privilegi que ha de tenir el codi que la provoca.

Els codis d'interrupcions són els següents:

- 0-31, excepcions i interrupcions no emmascarables
- 32-47, interrupcions mascarables
- 48-255, interrupcions software. Dedicarem la 0x80 (128) per les crides a sistema tal com ho fa Linux.

Aquesta nomenclatura així com els codis d'interrupcions són els que apareixen als manuals de Intel. Per tenir una informació més completa podeu consultar el capítol 4 del "**Understanding the Linux kernel**".

Per programar qualsevol excepció o interrupció (excepte la de la crida a sistema) farem tres passos:

1. **Inicialitzar** l'entrada de la IDT corresponent a la excepció o interrupció
2. **Programar el handler** que s'executarà quan es produeixi la interrupció o excepció. El paper del handler serà d'intermediari per cridar a la rutina de servei.
3. **Programar la rutina de servei** a la interrupció o excepció que serà el codi que farà realment el servei associat a la interrupció o excepció.

En el cas de la crida a sistema hem de fer un pas **addicional**. Com que generar una crida a sistema directament és complicat farem, per cada crida a sistema, una funció que s'encarregarà de fer el pas de paràmetres específic a la crida i de provocar la interrupció. Aquestes funcions s'anomenen **wrappers**. A la secció 4 s'explicarà amb més detall.

1.1. Conceptes previs

- Interrupció, excepció i crida a sistema.
- Context d'un procés.
- Gestió hardware d'una interrupció.

- Comprovació i paràmetres i retorn de resultats.

1.2. Gestió hardware d'una interrupció

Un cop tenim el sistema inicialitzat quan la CPU detecta que s'ha produït una interrupció el hardware realitza els següents passos de forma automàtica: (és un resum, teniu més detalls al "Understanding the Linux kernel")

- Determina l'índex i del vector de la interrupció i accedeix a l'entrada corresponent de la IDT
- Comprova que la interrupció ha estat generada des de una font autoritzada. Per fer-ho, compara el nivell de privilegi actual amb el guardat a l'entrada corresponent de la IDT. En cas d'accés no autoritzat, es genera una excepció de protecció general.
- Comprova si el nivell de privilegi del handler és diferent del nivell de privilegi actual (serà el nostre cas). Això vol dir que hem de canviar de pila.
- Per fer el canvi de "pila usuari → sistema", el sistema agafa l'adreça de la pila de sistema de la TSS (veure Task State Segment (TSS) secció 3.4). Un cop tenim el ss i el sp de sistema, el hardware guarda a la nova pila els valors del ss i esp de la pila anterior (la d'usuari en el nostre cas). També es guarden el registre $eflags$ (paraula d'estat o psw), cs i eip (de l'adreça que va provocar la interrupció).
- Salta a executar l'adreça que es va guardar a la IDT a l'entrada i . **Fins a aquest pas, aquests passos els fa el hardware de forma atòmica.**

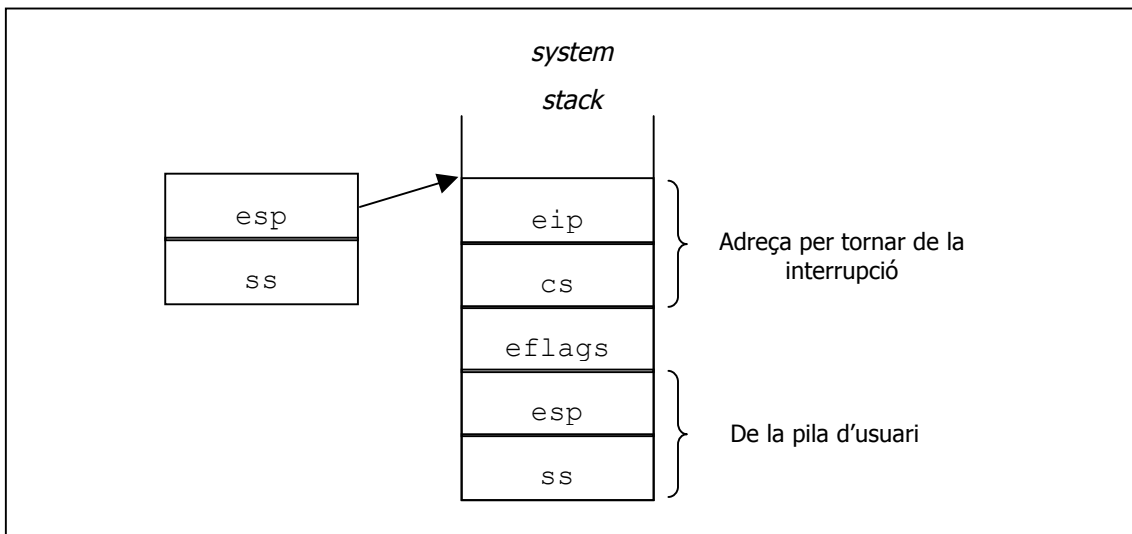


Figura 4. Estat de la pila de sistema quan comencem a executar el handler de una interrupció

- Executar la rutina de servei corresponent a la interrupció.
- Un cop executada, s'ha de tornar el control a la instrucció que va provocar la interrupció. Això es fa executant la instrucció **iret**. La instrucció **iret** agafa el valor dels registres eip i cs del top de la pila, carrega el registre $eflags$ amb el valor guardat a la pila i modifica els registres esp i ss per tornar a la pila corresponent al mode amb que es va produir la interrupció (mode usuari en el nostre cas). **Tots aquests passos es fan de forma atòmica amb la instrucció iret.**

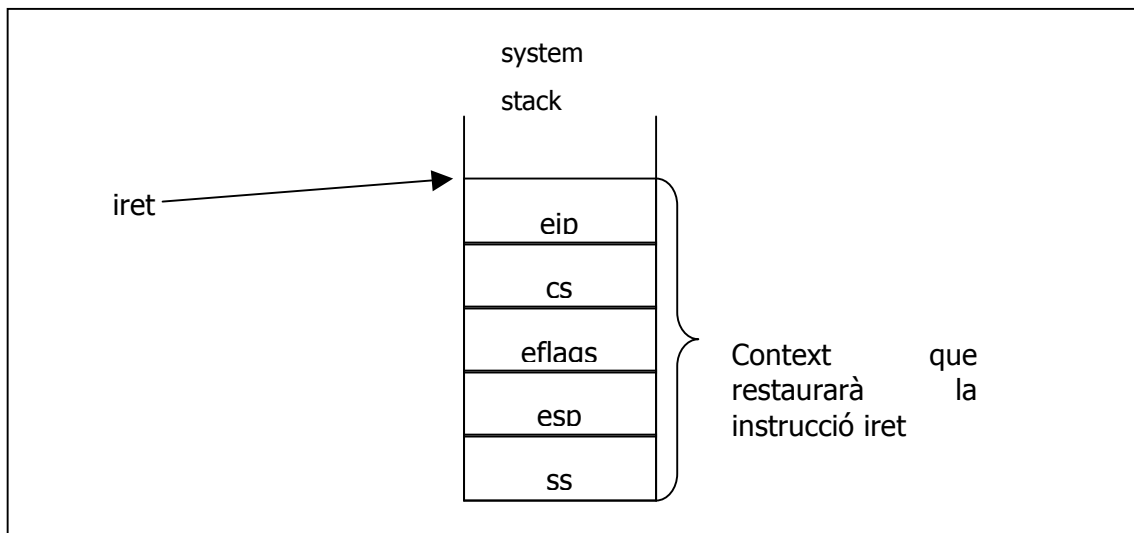


Figura 5. La pila de kernel ha de tenir aquest estat abans de executar iret

1.3. Treball a realitzar

El que farem en aquesta entrega serà:

- Incloure el codi necessari per poder rebre totes les excepcions
- Incloure el codi necessari per gestionar les interrupcions de rellotge i teclat
- Incloure el codi genèric necessari per poder fer crides a sistema
- Incloure el codi necessari per implementar la crida a sistema *write*

A les següents seccions us donem una guia de com afegir una excepció, una interrupció i una crida a sistema. **Heu de tenir clar que pràcticament és el mateix, ja que totes tres es gestionen mitjançant la IDT.** Tot i així, en aquest enunciat ho farem per separat per veure-ho amb detall però és important entendre que són molt similars.

1.4. Conveni en els noms de funcions

Per facilitar el seguiment del codi anomenarem les funcions de forma similar. El nom dels handlers serà el nom de la interrupció o excepció més "_handler" i el de la rutina de servei serà el nom de la interrupció o excepció més "_routine". Per exemple, per a la excepció "divide error" el nom del handler serà "divide_error_handler" i el de la rutina de servei "divide_error_routine".

En el cas concret de les crides a sistema, el handler s'anomenarà "system_call" i les rutines de servei es diran "sys_nomsyscall", per exemple "sys_write".

1.5. Fitxers

En general, els fitxers contenen funcions que comparteixen un objectiu o un tipus comú. En aquesta entrega els principals (potser no únics) fitxers que intervenen són:

- system.c: Inicialització del sistema.
- entry.S: Punts d'entrada al sistema (handlers).
- interrupt.c: Rutines de servei d'interrupcions i excepcions.
- sys.c: Crides a sistema.
- device.c: Part dependent de dispositiu de les crides a sistema
- libc.c: Wrappers de les crides a sistema.

- io.c: Gestió bàsica de l'entrada/sortida.

Amb aquesta guia **heu de decidir** on afegiu les modificacions al codi que us demanem.

NOTA: Afegiu en els fitxers d'include tots els prototipus de funcions que necessiteu perquè la compilació no doni warnings

Per a cada entrega, heu de **comprendre i ser capaços d'explicar i justificar les preguntes que es facin.**

2. Programació d'excepcions

En aquesta part s'hauran de programar totes les excepcions del sistema (exceptuant-ne la 15ena, reservada a Intel). Per capturar una excepció i executar la rutina que haguem creat per atendre-la, haurem d'entrar a sistema, ja que són operacions protegides. Normalment, els sistemes operatius intenten recuperar-se de l'excepció i restaurar el sistema. Les nostres rutines de servei seran molt simples:

- Mostrarem per pantalla que s'ha produït una excepció i direm quina és.
- El sistema quedarà parat en un bucle infinit i no tornarem al handler.

2.1. Paràmetres de excepcions

Depenent de la excepció es possible que el hardware ens hagi de proporcionar alguna informació per poder solucionar-ho (per exemple en el cas d'un page fault quin tipus d'accés l'ha provocat) Aquesta informació té una mida fixa (4 bytes) i la passa el hardware en forma de paràmetre a la pila de forma automàtica. Heu de tenir en compte que no es passa de la mateixa forma que es passen els paràmetres entre funcions en C, la pila no està igual, el paràmetre està al top de la pila quan es comença a executar el codi del handler . No totes les excepcions tenen aquest paràmetre, les que en tenen o no està indicat a la Taula 1. La Figura 6 ens mostra com estaria la pila de kernel i on apuntarien els registres `esp` i `ss` a l'inici d'una excepció de les que reben un paràmetre del kernel.

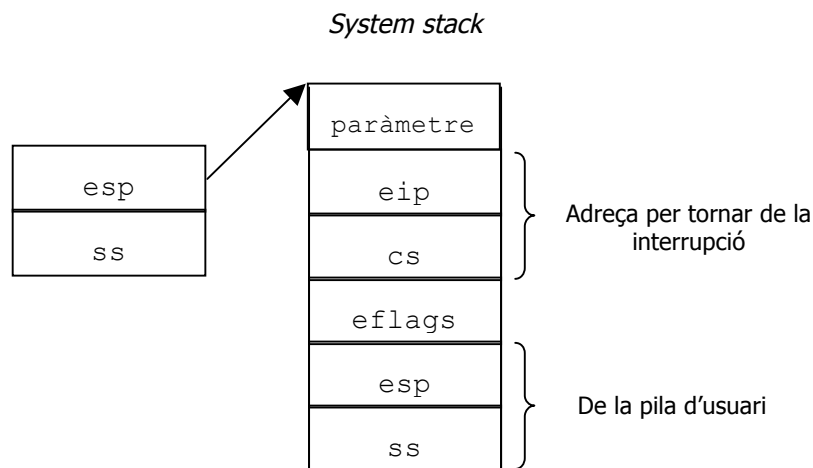


Figura 6. Pila de kernel a l'inici d'una excepció que rep un paràmetre del hardware

Per **inicialitzar la IDT** us proporcionem la següent funció:

```
setInterruptHandler(int posicio, void (*handler)(), int nivellPriv)
```

Us donem la implementació de la funció, heu de decidir on la utilitzeu. Els paràmetres que se li passen per programar l'excepció són:

- `int posicio`: entrada corresponent a la IDT.
- `void (*handler)()`: adreça del handler que atindrà l'excepció.³
- `int nivellPriv`: nivell de privilegi amb el que s'accedeix. Contemplarem dues possibilitat: 0 → Kernel, 3 → Usuari. **Heu de pensar i decidir** quin nivell de privilegi ha de tenir el codi que genera una excepció.

On heu afegit les crides a la funció `setInterruptHandler(...)`? Mireu al fitxer `system.c` que ja hi ha una funció d'inicialització global de la IDT.

La següent taula us mostra la posició de les excepcions a la IDT, el seu nom i el nombre de bytes que ocupa el seu paràmetre (o codi d'error⁴) (si en té):

Taula 1: Llista d'excepcions del sistema

# IDT	Excepció	Parametre
0	Divide error	No
1	Debug	No
2	NM1	No
3	Breakpoint	No
4	Overflow	No
5	Bounds check	No
6	Invalid opcode	No
7	Device not available	No
8	Double fault	4 bytes
9	Coprocessor segment overrun	No
10	Invalid TSS	4 bytes
11	Segment not present	4 bytes
12	Stack exception	4 bytes
13	General protection	4 bytes
14	Page fault	4 bytes
15	Intel reserved (<i>no fa falta fer-la</i>)	
16	Floatin point error	No
17	Alignment check	4 bytes

2.3. Programar el handler

Haurem de programar **un handler per cada excepció** que vulguem gestionar. Tots els handlers de les excepcions tenen un esquema comú. Han de fer els següents passos (en ensamblador):

1. Definir la capçalera d'ensamblador de la funció. Per fer-ho, podeu usar la macro⁵ **ENTRY**. Cridant-la a l'inici del codi de la funció i passant-li el nom de l'excepció com a paràmetre (`ENTRY(nom_excepció)`), s'encarrega de definir la capçalera.
 - a. A partir d'aquest punt sempre que vulguem definir una capçalera de funció en ensamblador utilitzarem la macro **ENTRY**.
2. Salvar el context (feu servir la macro **SAVE_ALL**)

³ Si no saps com es declaren i s'utilitzen els punters a funcions consulta el apèndix de programació en C

⁴ Per algunes excepcions, la unitat de control de la CPU també genera un codi d'error hardware i el posa a la pila de sistema abans de començar el handler de l'excepció.

⁵ Mireu a l'annex corresponent com funciona el mecanisme de macros de l'ensamblador.

3. Cridar la rutina de servei.
4. Restaurar el context. **Heu de programar una macro que anomenarem `RESTORE_ALL`** que restauri el context guardat al punt 2. A partir d'aquest punt utilitzarem aquesta macro per restaurar el context.
5. Eliminar el paràmetre (si l'excepció en tenia), consulteu la Taula 1.
6. Retornar de l'excepció. Com que no és un retorn "normal" sinó amb un canvi de mode inclòs retornarem amb un **`iret`** en comptes d'un `ret`⁶.

2.4. Programar les rutines de servei

En aquesta entrega la gestió d'excepcions serà molt senzilla. Heu de programar **una rutina de servei per cada excepció** que vulguem gestionar. En aquesta entrega tan sols haurà de:

- Mostrar un missatge per pantalla informant de l'excepció produïda.
- Com que no solucionarem el problema que ha generat l'excepció, afegirem a la rutina de servei un bucle infinit per bloquejar el sistema.

2.5. Provar que s'ha programat l'excepció correctament

Al codi del fitxer `user.c` us donem una línia de codi que podeu descomentar per forçar que es generi una excepció de protecció general.

```
__asm__ __volatile__ ("mov %0, %%cr3" : : "r" (0) );
```

Amb aquesta sentència es provoca una excepció de protecció general en intentar accedir a un registre del sistema des de mode usuari

Proveu alguna excepció més, per exemple intentar generar una excepció de divisió per 0 (segurament haureu de recompilar el `user.c` sense el flag d'optimització `-O2`). Feu les proves que calguin per provar el mecanisme de gestió d'excepcions. En cas de no funcionar intenteu veure amb el debugger quin és el problema. Comproveu si s'arriba a executar la excepció.

3. Programació d'una interrupció

En aquesta part s'hauran de programar la interrupció de rellotge del sistema i la interrupció de teclat. La Figura 7 ens mostra els passos que seguiria la interrupció de rellotge. La interrupció pot arribar en qualsevol punt del codi d'usuari. Si hem programat correctament la IDT, el primer que s'executarà és la funció `clock_handler` que haurem programat al fitxer `entry.S`. Dins del codi d'aquesta funció trobem la crida a la funció `clock_routine` que és la que realment fa la gestió de la interrupció.

⁶ Heu de tenir clar quines dades es guarden a la pila al passar a mode sistema i que fa la instrucció `iret`.

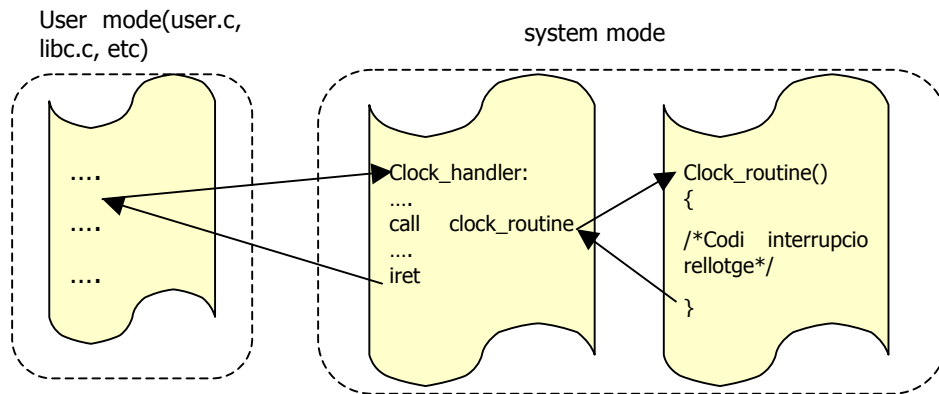
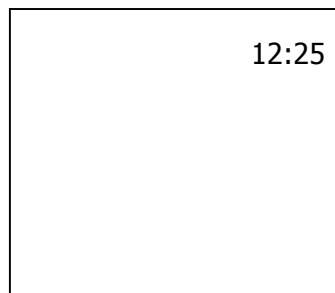


Figura 7. Passos de la interrupció de rellotge

3.1. Interrupció de rellotge

La interrupció de rellotge permetrà, un cop programada, mostrar per pantalla el temps que ha passat des que s'ha realitzat el boot, que és el que es farà en aquesta entrega. S'espera un resultat com el de la següent figura, és a dir, que el temps es vagi mostrant a una part prefixada de la pantalla.



Per programar una interrupció com la de rellotge (és a dir, de tipus mascarable) és necessari fer:

- Inicialitzar la IDT. Igual que en cas d'excepcions. La interrupció de rellotge es troba a l'entrada 32.
- Programar el handler.
- Programar la rutina de servei.
- Desinhibir la interrupció. La interrupció de rellotge és una interrupció "mascarable" que al codi que us donem està inhibida, això vol dir que no es produeixen. En aquesta entrega heu de modificar la màscara que desinhibeix interrupcions i que trobareu a la rutina **enable_int()** (fitxer **hardware.c**).

Heu de tenir en compte que un cop permeteu les interrupcions comencen a arribar. Recordeu que a la interrupció de rellotge farem principalment gestions de planificació del sistema. És important que des de el primer moment penseu en quin punt ja teniu estable el sistema i podeu permetre que us arribin interrupcions. La idea seria no haver de moure més endavant la funció **enable_int** per evitar errors. **On heu afegit la crida a la funció enable_int()?**

3.1.1. Programació del handler

La programació del handler d'una interrupció és molt similar al d'una excepció. Els passos són:

1. Definir la capçalera d'assemblador de la funció
2. Salvar el context
3. Cridar la rutina de servei
4. Fer l'**EOI (End Of Interruption)**. És indispensable per avisar al sistema que hem processat la interrupció i ja podem rebre un altre. Per fer l'EOI s'han de realitzar les següents instruccions **Feu una macro** que anomenarem EOI que inclogui la següent seqüència de instruccions:

```
movb $0x20, %al;
outb %al, $0x20;
```

5. Restaurar el context
6. Retornar de la interrupció (Vigileu que esteu tornant a mode usuari)

3.1.2. Programar la rutina de servei

Heu de programar la rutina de servei corresponent a la interrupció de rellotge. En aquesta entrega farem:

1. Escriure per pantalla el temps (minuts:segons) que ha passat des que s'ha fet el boot. Heu de tenir en compte que:
 - a. 18 interrupcions de rellotge (tics) = 1 segon
 - b. El temps transcorregut ha de mostrar-se sempre al mateix lloc de la pantalla. **Heu de programar** una nova funció per escriure a pantalla (`void printc_xy(int x,int y,char c);`) que rebrà com a paràmetre les coordenades on volem escriure. (El valor de les coordenades x e y a la sortida de la funció ha de quedar igual que abans d'executar-la.)
 - c. **Haureu de programar** una funció que passi d'enters a caràcters ASCII: `void itoa(int num, char *buffer);`

3.1.3. Provar que s'ha programat la interrupció correctament

S'haurà de veure per pantalla els minuts i segons que van passant des del moment de boot del S.O. En cas de no funcionar intenteu veure amb el debugger quin és el problema. Comproveu si s'executa la interrupció, si torneu a mode usuari, etc.

3.2. Programació de la interrupció de teclat

Com és sabut, el teclat és el principal dispositiu d'entrada de dades cap al sistema. Quan un usuari prem una tecla, el dispositiu genera una interrupció hardware. En aquest apartat es tracta d'encarregar-se de capturar aquesta interrupció i fer-ne el tractament per tal de que el S.O. pugui fer la lectura de la tecla premuda.

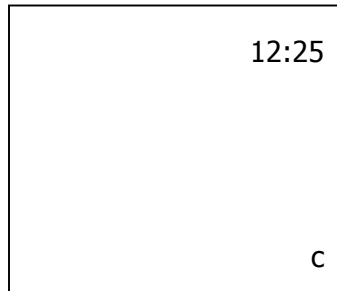
Caldrà implementar la rutina d'atenció a la interrupció de teclat (com s'ha fet anteriorment amb el rellotge). La interrupció del teclat es defineix a la posició 33 de la IDT. Els passos a fer són:

1. Permetre la interrupció de teclat: Modificar la màscara de interrupcions i inicialitzar la IDT

2. Programar el handler
3. Programar la rutina de servei

3.2.1. Rutina de servei de teclat

La rutina de servei de teclat estarà encarregada d'escriure per la pantalla el caràcter que correspon a la tecla que s'ha pitjat.



La rutina de servei de la interrupció de teclat ha de fer els següents passos:

- Llegir el port del registre de dades (0x60) amb la rutina del fitxer io.c

```
unsigned char inb(int port)
```

Amb el valor llegit d'aquest port s'ha de ser capaç de:

- Distingir si es tracta d'una make o un break. La Figura 8 ens mostra el contingut del registre. El bit 7 ens indica si és un make o un break. Els bits 0..6 ens retorna un codi de rastreig que cal traduir a caràcter.
- **Si es tracta d'un make:** Obtenir el caràcter que correspon al codi de

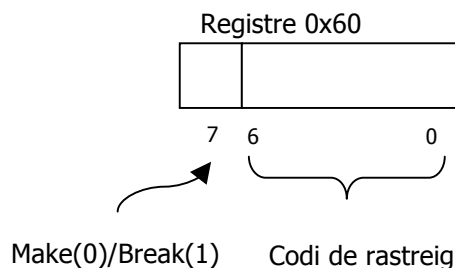


Figura 8. Contingut del registre 0x60 de teclat

rastreig premut a partir de la taula de traducció donada (char_map que teniu a interrupt.c). Aquest mapa de caràcters permet fer la traducció del que es llegirà del teclat cap a un caràcter ASCII.

- Escriure per pantalla el caràcter obtingut a la cantonada inferior dreta de la pantalla.
- Si es tracta d'una tecla que no correspon a un caràcter ASCII (Control, Enter, Backspace...) s'haurà d'imprimir la paraula "Control".

4. Programació d'una crida a sistema

En aquesta part s'haurà de programar la primera crida a sistema que es farà, que serà la **write**. Permetrà escriure per pantalla cadenes de caràcters a la pantalla des de mode usuari. Podeu consultar el capítol 8 del "Understanding the Linux Kernel"

Com ja sabeu d'altres assignatures, les crides a sistema tenen un punt d'entrada comú que es correspon amb la interrupció 0x80. Això vol dir que amb un punt d'entrada (1 handler) tindrem les N crides a sistema. **En aquesta entrega** programarem totes les funcions necessàries (comuns) a totes les crides a sistema (així com estructures de dades) i les específiques de la crida write.

La Figura 9 ens mostra els diferents passos que segueix una crida a sistema. El codi d'usuari crida el que ell creu que és la crida a sistema però que de fet és només un adaptador. El codi de la llibreria implementa aquest adaptador que anomenarem **wrapper** i que s'encarrega de fer el **pas de paràmetres** entre usuari i sistema, de generar el **trap** (int \$0x80) i de processar el **resultat**.

Un cop es genera la interrupció s'executa com una interrupció més: execució del handler i execució de la rutina de servei. En aquest cas no cal desinhibir la interrupció 0x80 ja que no és una interrupció mascarable. El que si haurem de fer és inicialitzar la IDT. Tal com hem dit, totes les crides a sistema tenen un handler comú, que anomenarem **system_call**. En aquest handler hem de veure quina és la crida a sistema concreta que volem fer i executar la rutina de servei corresponent (sys_write a l'exemple).

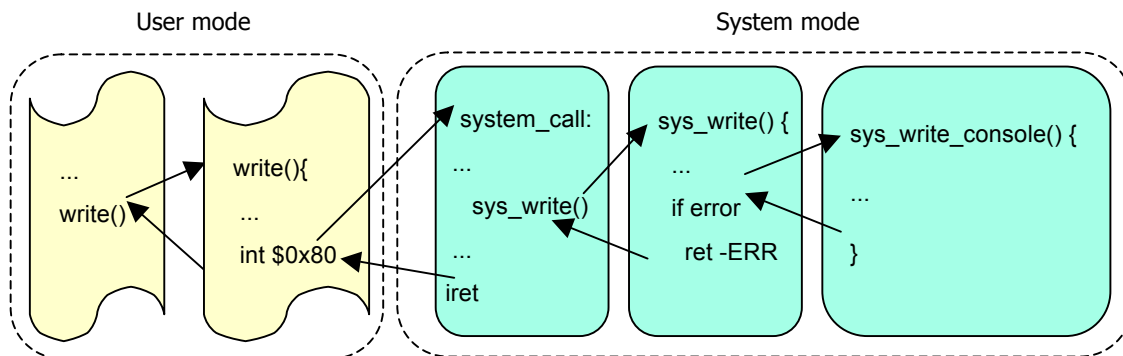


Figura 9. Passos que segueix una crida a sistema

IMPORTANT: Per totes les crides a sistema que us demanem haureu de comprovar al man de Ubuntu el seu funcionament: que ha de fer en el cas correcte i quins són els casos d'error.

4.1. Independència dels dispositius

Com ja heu vist a l'assignatura de SO, un dels principis bàsics a l'hora de dissenyar un sistema operatiu és la independència dels dispositius; és a dir, que la interfície de les crides al sistema d'entrada/sortida sigui la mateixa, independentment del dispositiu sobre el que estigui demanant el servei.

Tenint aquest principi en compte, es dissenyarà cada crida considerant una part independent i una altra dependent del dispositiu. D'aquesta manera, tindrem el sistema preparat per créixer en complexitat, o en cas d'afegir més dispositius. També ens aproparem més al disseny d'un sistema operatiu real.

Per exemple, la crida sys_write (part independent), cridarà després a sys_write_console (part dependent), en cas d'escriure per pantalla.

4.2. Retorn de resultats

En el cas de Linux, la convenció per retornar els resultats de les crides a sistema considera un retorn positiu o zero quan l'execució de la crida ha estat correcta i un valor negatiu quan s'ha produït un error. A més, en cas d'error, aquest valor negatiu indica quin ha estat aquest error.

No obstant, el valor retornat per la crida és processat pel wrapper de la crida a sistema que es troba en la llibreria de C. Així, si el valor de retorn era positiu o zero, aquest serà retornat tal qual al programa d'usuari. Però si el valor era negatiu, el wrapper guardarà el valor absolut d'aquest retorn en una variable d'error definida a la llibreria (el famós **errno**) i retornarà al programa d'usuari un -1 per indicar que la crida a sistema ha provocat un error. En aquest cas, si l'usuari desitja informació addicional de l'error produït ha de consultar la variable **errno**. Si la crida a sistema no retorna error, la variable **errno** no s'ha de modificar.

Volem incorporar a ZeOS aquesta convenció en la gestió de resultats. Així, **totes les crides a sistema que implemeteu en el projecte** hauran de retornar un nombre negatiu indicant l'error produït en cas d'execució incorrecta. Podeu trobar les constants que s'utilitzen per identificar aquests errors a Linux al fitxer **errno.h**. Afegiu-les a ZeOS. A més, els vostres wrappers de les crides a sistema hauran de processar els retorns negatius de les crides, actualitzant la variable **errno** amb el valor absolut d'aquests errors i retornant un -1.

També caldrà afegir una funció per tal que els usuaris puguin obtenir informació sobre l'error generat per una determinada crida a sistema. Així, afegirem a la llibreria la funció **void perror ()**, que escriurà per la sortida estàndard un missatge d'error en funció del valor de la variable **errno**.

↳ **Nota:** En la descripció de les crides a sistema que apareixen en aquest document, es consideren els valors de retorn des del punt de vista de l'usuari, per tant, en cas d'error sempre es retorna -1.

4.3. Programació del wrapper del write

Heu de programar el wrapper del write (els wrappers els programarem en ensamblador-inline). La capçalera del write és la següent:

```
int write (int fd, char * buffer, int size);
```

Un wrapper qualsevol ha de fer:

1. El pas de paràmetres de mode usuari a mode sistema a ZeOS es fa a través de registres com a Linux. Per tant hem de copiar els paràmetres de la pila als registres. Els paràmetres es posen als registres **ebx**, **ecx**, **edx**, **esi**, **edi**. L'ordre és: el primer paràmetre (més a l'esquerra) a **ebx**, el segon a **ecx**, etc.
2. Posar l'identificador de la crida a sistema al registre **eax**. En el cas del write, és el nombre 4.
3. Generar el trap: `int $0x80`
4. Procés del resultat (veure secció 4.2)
5. Retornar

IMPORTANT: Aquestes són les funcions bàsiques d'un wrapper, no hem inclòs accions genèriques a les funcions en ensamblador com poden ser salvar i restaurar els registres que utilitzem. Consulteu les convencions de C per a Linux en l'Annex

corresponent per recordar l'ordre del pas de paràmetres en C o els registres que cal salvar en cada cas.

4.4. Inicialització IDT

El punt d'entrada a una crida a sistema l'inicialitzarem utilitzant la crida

```
void setTrapHandler(int posició, void (*handler)(), int nivellPriv)
```

Els paràmetres que se li passaran són equivalents als del setInterruptHandler (Penseu quin nivell de privilegi heu de posar en aquest cas).

4.5. Programació del handler

Heu de programar el handler comú a totes les crides a sistema. Els passos a fer són:

- Salvar el context (macro SAVE_ALL)
- Comprovar que és un nombre de crida a sistema correcta (és a dir, que està en el rang de crides a sistema definides). Penseu com ho podem fer de forma senzilla. En cas de que no ho sigui, cal retornar l'error corresponent.
- Invocar la crida a sistema corresponent
- Actualitzar el context salvat al entrar al kernel perquè al restaurar-lo trobem a eax el resultat de la crida a sistema. Recordeu que les crides en C deixen el resultat de les funcions al registre eax. Si no modifiquem aquest context, al restaurar retornaríem el valor que tenia el eax al entrar en el kernel.
- Restaurar el context
- Retornar a mode usuari

Per invocar la crida a sistema corresponent, **necessitarem una taula** que relacioni l'identificador de cada crida amb la seva rutina. Anirem omplint la taula a mesura que anem afegint crides a sistema.

Es definirà la taula de crides en ensamblador anomenada **sys_call_table**. Aquesta taula es va omplint amb entrades de l'estil:

```
ENTRY(sys_call_table)
.long sys_ni_syscall //adreça de sys_ni_syscall
.long sys_nomrutina // adreça de sys_nomrutina
```

Cal tenir en compte que les crides no implementades però que es troben dins del rang vàlid han de contenir la crida a **sys_ni_syscall (que també heu de programar)**. Aquesta crida només retornarà un identificador negatiu de l'error produït. Cada línia és una entrada de la taula, no s'ha de posar ";" al final.

La forma d'executar la crida a sistema indexada per eax seria:

```
call *sys_call_table(, %eax, 0x04);
```

- Podeu consultar com declara al codi font de Linux la taula sys_call_table (la url <http://lxr.linux.no/> ens facilita la navegació pel codi de Linux).

4.6. Rutina de servei a la crida a sistema write

El següent pas a fer serà definir la rutina de servei `sys_write`, que és la que s'ocuparà de mostrar els caràcters per pantalla i fer les comprovacions que pertoquin.⁷

```
int sys_write(int fd, char * buffer, int size);
```

fd: canal on escriurem. En aquesta entrega ha de ser sempre l'1

buffer: punter als bytes que es volen escriure

size: nombre de bytes a escriure

retorna → Nombre negatiu si error (indicant el tipus d'error) i el nombre de bytes escrits si OK.

IMPORTANT: En aquest punt es molt important que us fixeu que la crida `sys_write` té paràmetres, que algú ens ha d'haver passat, sinó el codi que genera el compilador seria incorrecte. La crida a aquesta funció l'heu afegit vosaltres al handler, això vol dir que el pas de paràmetres també l'heu fet (o l'heu de fer) vosaltres. Penseu si cal afegir alguna cosa més al codi que ja teniu. Feu un dibuix de com està la pila just després de fer la crida al `sys_write` a veure si els paràmetres són o no al seu lloc. Si hi són, penseu com heu fet el pas de paràmetres.

Les crides a sistema (en general) han de fer els següents passos:

1. **Comprovar els paràmetres:** `fd`, `buffer` i `size` (penseu els casos d'error). Heu de tenir en compte que el sistema ha de ser robust i per defecte ha d'assumir que el els paràmetres que venen d'usuari venen d'un codi no segur (**la `libc.c` és codi d'usuari**). Feu una comprovació exhaustiva de tots els errors que poden tenir els paràmetres.
 - a. Per comprovar el `fd`, definiu una nova funció **`int comprova_fd (int fd, int operació)`** que comprova si el canal especificat i la operació que es vol fer. Les operacions poden ser `LECTURA` o `ESCRITURA` (definiu dues constants). La funcionalitat d'aquesta funció es comprovar que la operació indicada sobre el canal és correcta, retornant 0, o, en cas contrari, retorna un identificador negatiu de l'error produït. En aquesta entrega farem una versió mínima comprovant que el canal sigui l' 1 (que correspon amb `stdout`) i la operació `ESCRITURA`.
 - b. Per comprovar el `buffer`, **en aquesta entrega** us podeu limitar a verificar que el punter passat no sigui `NULL`.
2. **Copiar dades des de/a l'espai d'adreces d'usuari si és necessari.** Mireu el funcionament de les funcions `copy_to_user/copy_from_user` (secció 4.6.1).
3. **Cridar a la rutina dependent del dispositiu o implementar el servei demanat** si no és E/S. En aquest cas:

```
int sys_write_console (char *buffer, int size);
```

Aquesta funció escriu per pantalla les dades que rep a `buffer` i retorna el nombre de bytes escrits.

4. Retornar resultat

⁷ Recordeu que el `write` és una crida a sistema UNIX per escriure bytes, no caràcters, encara que el tipus d'entrada sigui un punter a `char`. Per això no s'ha de fer cap tractament especial amb les dades.

4.6.1. Còpia de dades des de/a espai d'adreces de usuari

Còpia de dades des del SO al usuari o viceversa es una operació crítica perquè pot provocar la vulnerabilitat del nucli del sistema. A aquesta entrega es presenta la primera part d'aquesta operació, on vosaltres haureu d'utilitzar dues funcions *copy_from_user* and *copy_to_user*. Durant la segona entrega haureu d'implementar la funció *access_ok* que es la funció que donarà robustesa al Sistema Operatiu.

El manual de "The Linux Kernel Module Programming Guide" dona la següent argumentació sobre la necessitat d'aquestes funcions:

"The reason for *copy_from_user* or *get_user* is that Linux memory (on Intel architecture, it may be different under some other processors) is segmented. This means that a pointer, by itself, does not reference a unique location in memory, only a location in a memory segment, and you need to know which memory segment it is to be able to use it. There is one memory segment for the kernel, and one for each of the processes.

The only memory segment accessible to a process is its own, so when writing regular programs to run as processes, there's no need to worry about segments. When you write a kernel module, normally you want to access the kernel memory segment, which is handled automatically by the system. However, when the content of a memory buffer needs to be passed between the currently running process and the kernel, the kernel function receives a pointer to the memory buffer which is in the process segment. The *copy_from_user* and *copy_to_user* functions allow you to access that memory"

Es a dir que aquestes dues funcions (*copy_from_user* i *copy_to_user*) son les responsables d'encapsular la complexitat deguda a les diferències arquitecturals dels processadors. En el nostre cas, aquestes funcions només seran útils per a realitzar còpies de dades entre els espais d'adreces d'usuari i sistema.

4.7. Provar que s'ha programat la crida a sistema correctament

Proveu que el codi afegit funciona correctament afegint crides a *write* dins el codi d'usuari. Feu diferents proves modificant els paràmetres per comprovar que tot és correcte: el pas de paràmetres, l'accés a les dades, el control d'errors, el retorn de resultat, etc. Proveu també amb diferents tipus de cadena de caràcters: variables locals, globals, inicialitzades, no inicialitzades, etc. En cas de no funcionar intenteu veure amb el debugger quin és el problema. **NOTA:** Afegiu en els fitxers d'*include* totes les capçaleres que necessiteu perquè la compilació no doni warnings. Per finalitzar les proves del vostre codi feu que el *user.c* cridi a la rutina *runjp* definida a la llibreria que us proporcionem (*libjp1.a*) i comproveu que totes les proves finalitzen correctament. Per fer-ho, necessitareu modificar el *Makefile* per a enllaçar l'executable del *user* amb la llibreria.

ENTREGA 1.2: GESTIÓ DE PROCESSOS

1. Introducció i descripció del treball a realitzar

En aquesta entrega afegirem tot el codi i les estructures de dades necessàries per definir i gestionar processos a ZeOS. També afegirem el codi per inicialitzar els processos, per crear processos nous, per destruir-los per planificar-los i per sincronitzar-los.

Heu de tenir present durant la realització de l'entrega que el codi relacionat amb processos és clau per tenir un bon rendiment al sistema. Això vol dir que l'espai de les estructures de dades ha de ser mínim i els algorismes de gestió el màxim d'eficients.

Com ja sabeu, un procés té un espai d'adreces propi. Un procés executant-se en mode usuari accedirà a la seva pila d'usuari, a les seves dades d'usuari i al codi d'usuari. Quan s'està executant en mode sistema accedeix a les dades i codi de kernel i utilitza la seva pròpia pila de sistema.

En primer lloc el que farem serà definir les estructures de dades que necessitem, bàsicament, el descriptor de procés.

Un cop definides les dades afegirem el codi per inicialitzar els processos. Això inclourà la taula de processos i la llista de processos *ready*.

Un cop afegit el codi d'inicialització afegirem les crides a sistema per crear, modificar la prioritat i destruir processos: `fork()`, `nice(int quantum)` i `exit()`.

Després, haurem de planificar els processos. Això vol dir modificar la interrupció de rellotge per decidir quan toca fer un canvi de procés (o canvi de context), implementar la rutina de canvi de context, i les funcions necessàries per gestionar la política de planificació.

Per últim, haurem de implementar les crides de sincronització entre processos amb semàfors i afegirem als processos una gestió d'informació estadística bàsica sobre la seva execució.

La gestió de processos implica coneixements de la gestió de memòria. Aquí intentem donar-vos uns conceptes bàsics. Molta feina se us dóna feta però **s'espera de vosaltres que entengueu el codi que se us dóna.**

La gestió de processos en general la podeu trobar al capítol 3 i 11 del *Understanding the Linux Kernel*. També podeu trobar informació molt útil sobre la organització de memòria al capítol 2.

Aquesta entrega és la més difícil i per tant és recomanable fer moltes proves. És molt important dividir aquesta feina i anar provant per parts. Feu un joc de proves exhaustiu que inclogui aquestes proves: provar que les estructures de dades estan bé (procés inicial, cola de run), provar a fer el fork amb només un procés nou, provar la planificació amb només dos processos, fer N processos i provar la planificació amb N processos.

1.1. Conceptes previs bàsics

- Procés. Dades relacionades amb la gestió de processos: PCB (`task_struct`), llistes, etc.
- Espai d'adreces d'un procés. Diferència entre adreces lògiques i físiques. Paginació.

- Canvi de context. Planificació. Polítiques de planificació.

1.2. Fitxers

Els fitxers a modificar principalment en aquesta part són:

- sched.c/sched.h. Codi de gestió/definició de processos

2. Definició d'estructures de dades

Els objectius d'aquesta entrega són crear les estructures de dades necessàries per poder gestionar els processos a ZeOS. S'haurà de definir l'estructura de dades del descriptor de procés (**task_struct**).

2.1. Definició de les estructures de dades pels processos

Es començarà **definint l'estructura de dades del descriptor de procés (task_struct)**. Es poden afegir tants camps com es creguin necessaris. També haureu de controlar quins processos estan creats/l·liures. Això es pot fer de diferents formes, decidiu vosaltres com ho fareu.

Per facilitar l'accés al task_struct, en comptes de tenir el task_struct i després una pila per procés, solaparem en memòria les dues estructures de dades. Per això utilitzarem una **union de C**, que és un tipus de dades especial. **En C, si declares una union de a,b i c, l'espai que el compilador reserva és el màxim(a,b,c), no la suma de l'espai de a, b i c.**

Al fitxer sched.h trobeu la definició buida del task_struct. **Heu de completar la seva definició.** La declaració de la task_union ja us la donem feta. La task_union és la unió entre el descriptor de procés (task_struct) i la pila de sistema del procés. **Heu d'entendre com funciona i quines implicacions té** que pila de sistema i task_struct comparteixin espai.

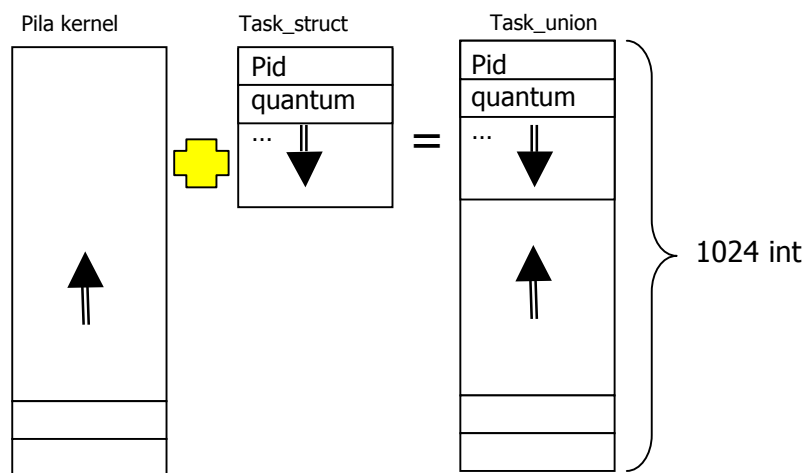


Figura 10. La pila de kernel i la task_struct de cada procés utilitzen la mateixa pàgina de memòria (4k)

task_union. En aquest vector, cada task_struct està protegida per una pàgina buida (task_protect) que no té permís d'escriptura (vegeu Figura 11). D'aquesta manera, podem detectar si s'intenta sobre escriure la task_struct d'un procés degut a un

overflow en l'accés a la pila de kernel d'un altre procés i, en aquest cas, generar una excepció de fallada de pàgina.

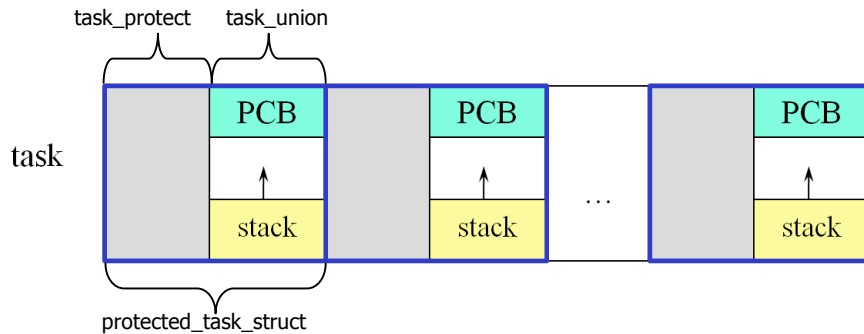


Figura 11. Vector task

El procés associat a l'entrada 0 de la taula serà el procés inicial, que anomenarem procés 0, procés inicial, o task 0. Aquest procés no podrà ser eliminat mai.

2.2. Identificació de processos

Un cop tenim la taula de processos, una qüestió important és com identificar un procés. Penseu que al entrar al sistema, per exemple en una crida a sistema o a la interrupció de rellotge, hem de saber qui procés som. **Com sabem quin procés som?: per la pila de kernel.**

¿Quina informació tenim quan entrem al codi de kernel? Sabem:

- Que les piles de sistema i la task_struct estan físicament al mateix espai i que ocupen 4KB.
- Que la task_struct està al principi de la pàgina que ocupa.
- Que el canvi dels registres que apunten a la pila (de kernel) és automàtic, per hardware (veure secció 3.4).

Per tant, **si al registre esp (adreça de la pila) li apliquem una màscara per posar a 0 els darrers X (calculeu quants) bits tenim l'adreça de l'inici del task_struct.**

Heu de fer una funció (o macro) que anomenarem **current** que ens retorni l'adreça del task_struct actual basant-vos en aquesta idea. Amb aquesta macro tindrem el punter a la task que s'està executant en cada moment. A partir d'aquí ja podem accedir a les seves dades.

```
struct task_struct * current();
```

2.3. Llistes de processos

El vector task conté tots els possibles processos del sistema. Un cop creats, els processos s'agrupen per estats (READY, BLOCKED, etc). Per agrupar-los, utilitzarem la estructura de llistes doblement enllaçada que teniu.

Per incloure un procés en una llista, cal modificar el task_struct. Al fitxer list.h trobareu el API per gestionar llistes. En aquest apartat necessiteu implementar la funció (o

macro) **list_head_to_task_struct** per traduir del punter a list_head a punter a task_struct. Podeu fer servir la macro list_entry.

```
struct task_struct * list_head_to_task_struct(struct list_head * l);
```

Un cop definides les llistes, s'haurà de crear la llista de processos en estat RUN (només 1 en el nostre cas) + READY. Serà una llista de tots els processos en aquests estats. Anomenarem a aquesta llista `runqueue`. **Heu de declarar i inicialitzar la llista de processos run+ready.**

3. Gestió de Memòria

En aquesta secció us expliquem alguns conceptes i estructures de dades relacionades amb la gestió de memòria perquè sapiguen com fer els canvis que us demanem relacionats amb processos.

A ZeOS el que us donem és tota la part de segmentació inicialitzada (i fixa) i la part de paginació inicialitzada pel procés inicial. En principi a ZeOS tots els processos accediran a les mateixes pàgines lògiques i el que farem serà canviar les pàgines físiques a les que es tradueixen.

Per poder explicar la creació de processos és necessari tenir alguns coneixements de organització de memòria a les arquitectures 80x86.

En el cas de Zeos heu de diferenciar entre adreces lògiques i físiques.

- Lògiques⁸. Un unsigned integer de 32 bits s'utilitza per representar fins a 4GB. El rang d'adreces és de 0x00000000 a 0xffffffff. **Totes les adreces que genera la CPU son lògiques i es tradueixen mitjançant la MMU.**
- Físiques. Utilitzades per adreçar memòria del chip. Estan representades per unsigned integers de 32 bits.

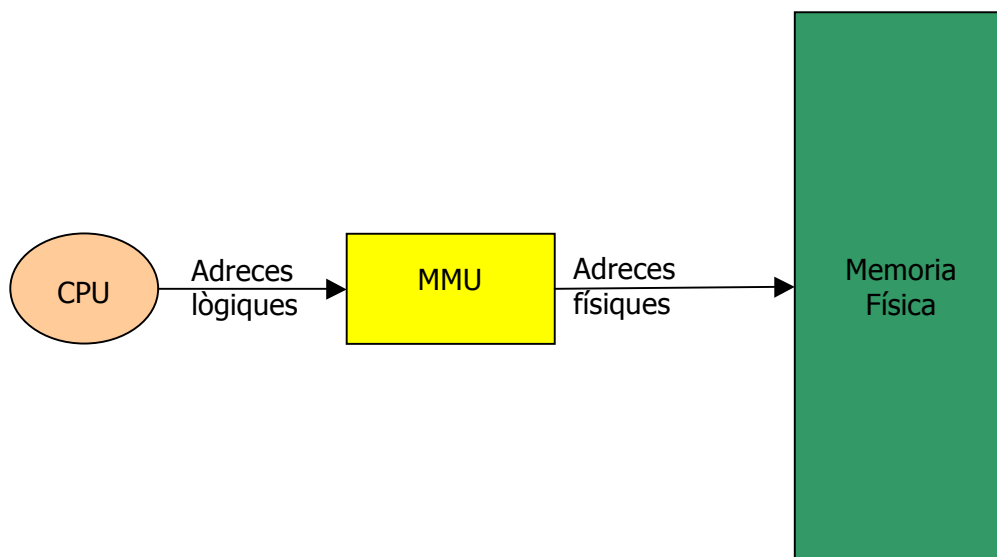


Figura 12. Traducció d'adreces lògiques a físiques

⁸ Al Understanding the Linux kernel diferencian entre lògiques, lineals i físiques. El que aquí anomenem lògiques al llibre son lineals.

La Figura 12 ens mostra la relació entre les diferents adreces. A continuació expliquem breument els conceptes necessaris de segmentació i paginació.

3.1. MMU: Mecanisme de Paginació

La paginació tradueix d'adreces lògiques a adreces físiques. La memòria està organitzada en marcs de pàgina, també anomenats **frames**, de la mateixa mida. La estructura de dades que tradueix de adreces lògiques a físiques és la taula de pàgines.

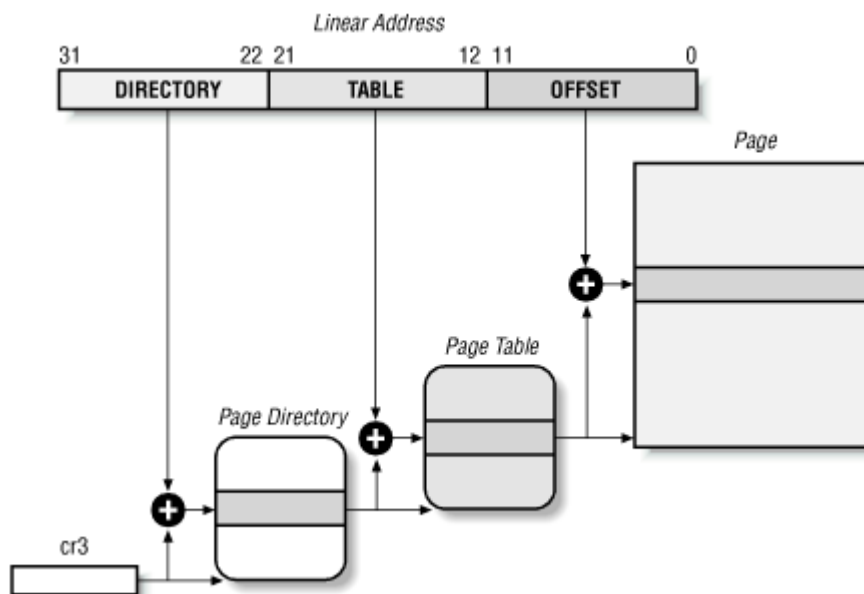


Figura 13. Paginació a les arquitectures 80x86

La Figura 13 ens mostra l'esquema bàsic de traducció d'adreces mitjançant paginació que tenim a ZeOS. A l'arquitectura Intel, donada una adreça de 32 bits, els 10 bits més alts defineixen la entrada del directori (bits 31-22). En el nostre cas sempre valdran 0 ja que només utilitzarem la entrada 0 del directori. Els 10 bits entremitjos (bits 21-12) defineixen la pàgina lògica. I els 12 bits baixos (bits 11-0) defineixen el desplaçament dintre la pàgina (fins 4 KB). Per tant, per obtenir el nombre de pàgina a partir d'una adreça lògica només hem de fer: $\text{num_pàgina} = (\text{adreça} \gg 12)$.

Per activar el mecanisme de paginació a les arquitectures 80x86 és necessari **posar a 1 el bit PG del registre cr0**. Si PG=0 vol dir que les adreces lineals són iguals a adreces físiques. Aquesta inicialització ja us la donem feta.

3.2. El directori i la taula de pàgines

L'esquema de adreces que s'utilitza a ZeOS consisteix a tenir dos nivells d'indexació. El primer nivell és una pàgina de directori que apunta a les taules de pàgines definides. Per conèixer l'adreça del directori tenim un registre especial, el **cr3**. **En el cas de ZeOS només utilitzarem 1 taula de pàgines, per tant, el nostre directori només tindrà una entrada vàlida.** La inicialització del directori ja us la donem feta, consulteu el fitxer mm.c.

Les entrades del directori i de la taula de pàgines tenen la mateixa estructura. Cada entrada te els següents flags (mireu el codi de ZeOS):

```

typedef union
{
    unsigned int entry;
    struct {
        unsigned int present : 1;
        unsigned int rw      : 1;
        unsigned int user    : 1;
        unsigned int write_t : 1;
        unsigned int cache_d : 1;
        unsigned int accessed : 1;
        unsigned int dirty   : 1;
        unsigned int ps_pat  : 1;
        unsigned int global  : 1;
        unsigned int avail   : 3;
        unsigned int pbase_addr : 20;
    } bits;
} page_table_entry;

```

En el cas de ZeOS hem definit cada entrada com una **union** per inicialitzar-la de forma més senzilla. D'aquesta forma si definim el camp entry=0 és com posar tota la entrada a 0. Definiu una nova funció per fer aquesta operació amb la següent capçalera:

```
void del_ss_pag(unsigned pagina_logica);
```

El camp pbase_addr conté el nombre de pàgina física (o frame) pel qual es traduirà la pàgina lògica corresponent. (traduirà de **nombre de pàgina lògica a nombre de pàgina física**). Recordeu que el contingut del camp pbase_addr és el nombre de la pàgina física, no l'adreça de la pàgina. Per canviar la traducció d'una pàgina concreta podeu utilitzar la funció (mm.c):

```
void set_ss_pag(unsigned num_pagina_logica, unsigned num_pagina_fisica);
```

3.3. Translation Lookahead Buffer (TLB)

El TLB és com una cache de la taula de pàgines utilitzada per optimitzar l'accés a memòria. En aquesta taula es guarden les entrades de la taula de pàgina del procés actual que s'han utilitzat. D'aquesta forma per traduir les adreces lineals a físiques ens estalviem el primer accés al directori. A més l'accés al TLB és molt ràpid. La única cosa que hem de tenir en compte és que les entrades de la TLB **no es sincronitzen automàticament amb la taula de pàgines**. Per tant, modificacions sobre la taula de pàgines poden deixar amb un valor incorrecte les entrades del TLB i per tant aquestes entrades s'han d'invalidar. En concret, sempre que esborrem una entrada de la taula de pàgines o en modifiquem el seu contingut, caldrà invalidar les entrades del TLB. Aquesta invalidació d'entrades també s'anomena fer flush del TLB. **Per fer un flush del TLB cal reescriure el registre cr3 (rutina set_cr3(void)).**

3.4. Task State Segment (TSS)

La arquitectura 80x86 defineix un segment (o regió de memòria) específic anomenat Task State Segment (TSS) per implementar canvis de context per hardware. Ni ZeOS, ni Linux, volen utilitzar la TSS però l'arquitectura els obliga a definir una TSS per cada cpu (1 en el nostre cas). **La TSS és principalment utilitzada per conèixer l'adreça de la pila de kernel quan fem un canvi de mode de usuari →**

sistema. Podeu consultar als fitxers de ZeOS els camps que té la TSS i com s'inicialitza en el cas del procés inicial que ja us donem fet. És important entendre com funciona la TSS perquè s'utilitza en el canvi de context. Torneu a llegir els passos que us explicàvem a la pàgina 17 sobre els passos que realitzar el hardware quan es produeix un canvi de mode d'execució (en una interrupció) per entendre la utilització de la TSS.

La Figura 14 ens mostra un exemple de com s'utilitzen els camps `ss0` i `esp0` de la TSS. Aquests camps sempre apunten a la base de la pila de kernel. Quan entrem al kernel el hardware els utilitza per saber on està la pila de kernel.

Recordeu que el canvi del contingut dels registres `ss` i `esp` es fa per hardware automàticament al canviar a mode kernel.

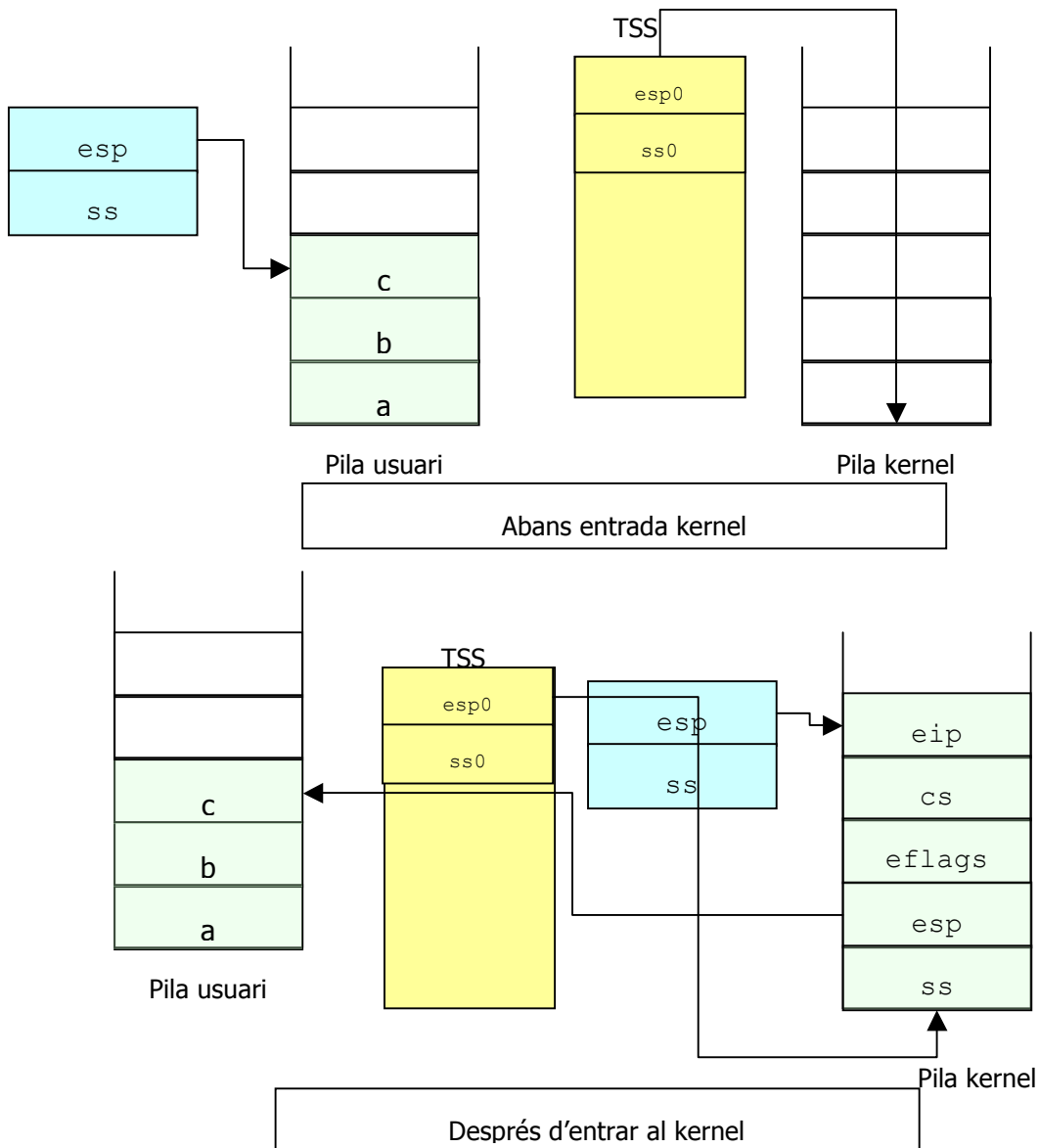


Figura 14. Els camps `esp0` i `ss0` de la TSS ens indiquen on està la pila de kernel per fer el canvi de mode usuari a mode kernel

3.5. Organització concreta de ZeOS

3.5.1. Espai lògic

L'espai lògic del procés és l'espai adreçable des dels programes d'usuari. L'espai lògic està compost per segments i pàgines. La segmentació a ZeOS s'ha reduït a la mínima expressió com és el cas de Linux. Si mireu el fitxer bootsect.S veureu com hem inicialitzat la segmentació. Veureu que els segments estan definits per començar a l'adreça 0 i amb una mida infinita. Això significa que una adreça que és segment:desplaçament es traduirà bàsicament per una adreça lògica equivalent al desplaçament.

Per que tot quadri, i atès que no tenim carregador, hem fet que el linkador generi les adreces a partir de la 0x100000 (L_USER_START a mm_address.h). La organització dels segments l'heu d'entendre a fi de tenir una visió global de la gestió de memòria però no heu de modificar res del que us donem ja fet.

3.5.2. Espai d'adreces lògic a ZeOS

L'espai lògic del procés està compost per N pàgines de codi i M pàgines de dades+pila, en ambdós casos situades de manera consecutiva.

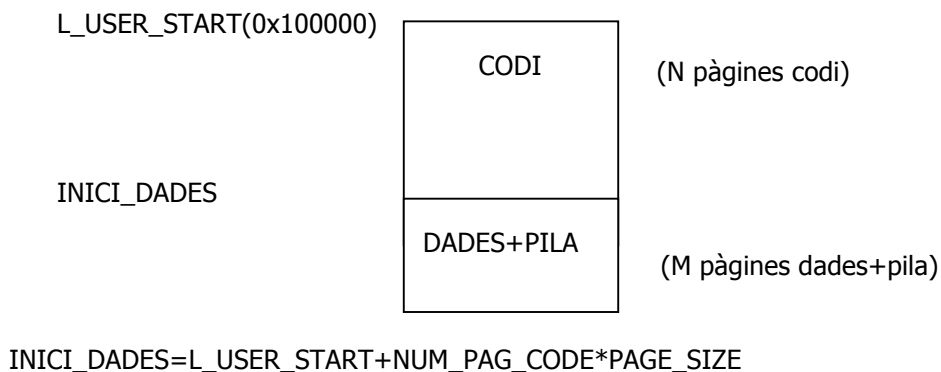


Figura 15. Espai d'adreces lògic

La Figura 15 ens mostra les adreces d'usuari d'un procés qualsevol. A l'esquerra de la figura teniu les adreces lògiques de qualsevol procés. L'espai lògic de tots els processos comença a l'adreça L_USER_START i te NUM_PAG_CODE pàgines de codi i NUM_PAG_DATA pàgines de dades.

Si traduïm les adreces a nombres de pàgina lògica, tenim que a partir de la pàgina lògica (L_USER_START>>12 == entrada 256 de la taula de pàgines) es situa l'espai d'adreces del procés, que comença pel codi. Tindrem 4KB per cada entrada.

En el cas de ZeOS les primeres 255 entrades de la taula de pàgines corresponen a espai de sistema (mireu la inicialització al fitxer mm.c). La pàgina (L_USER_START>>12) és la primera pàgina lògica de codi (definiu una constant per accedir-hi a mm_address.h). Tots els processos tenen, per tant, 255 + NUM_PAG_CODE+NUM_PAG_DATA entrades vàlides de la taula de pàgines: 0-255 sistema, només accessibles en mode kernel, NUM_PAG_CODE pàgines de codi usuari i NUM_PAG_DATA de pàgines de dades+pila usuari. A la configuració inicial de ZeOS, NUM_PAG_CODE=8 i NUM_PAG_DATA=20.

Les pàgines lògiques i físiques de sistema i de codi seran compartides, això vol dir que les entrades lògiques de la taula de pàgines de tots els processos apuntaran a les mateixes pàgines físiques.

Les pàgines de dades+pila d'usuari són privades, per tant, la traducció de pàgina lògic a pàgina física (o frame) serà diferent per cada procés en el cas de les pàgines de dades i pila.

Com que no utilitzarem moltes pàgines, utilitzarem només una entrada del directori de taules de pàgines (ENTRY_DIR_PAGES). Aquesta entrada conté l'adreça de la taula de pàgines. Com que només part de la taula de pàgines és diferent entre processos (les pàgines de dades+pila), hem decidit que tindrem una única taula de pàgines per tots els processos i que **modificarem les pàgines corresponents a dades+pila quan fem el canvi de context.**

En aquesta entrega heu d'implementar la funció access_ok, que donat un rang de memòria, amb una adreça de memòria inicial, una longitud, i un mode d'accés, comprova que les adreces dins aquest rang pertanyen a l'espai lògic del procés que invoca la funció.

```
int access_ok(int type, const void *addr, unsigned long size)
type READ o WRITE
addr adreça d'usuari
size tamany del block a validar
retorna 1 si el accés és vàlid i 0 si no ho és
```

Aquesta comprovació s'inclourà a totes les crides a sistema que tinguin com a paràmetre un punter a la memòria del procés.

3.5.3. Espai físic

El conjunt de totes les adreces físiques corresponents a les adreces lògiques del procés és l'anomenat **espai físic**. Tot i que les adreces de l'espai lògic siguin consecutives, les adreces de l'espai físic associat poden no ser-ho. A ZeOS l'organització de la memòria física serà la següent. Les primeres adreces físiques estan assignades a les pàgines lògiques del sistema. Les pàgines de codi d'usuari seran compartides per tots els processos i començaran a l'adreça física PH_USER_START (definida a segment.h). A partir de l'adreça PH_USER_START+NUM_PAG_CODE*PAGE_SIZE es situaran les dades i piles del processos en execució (veure Figura 16).

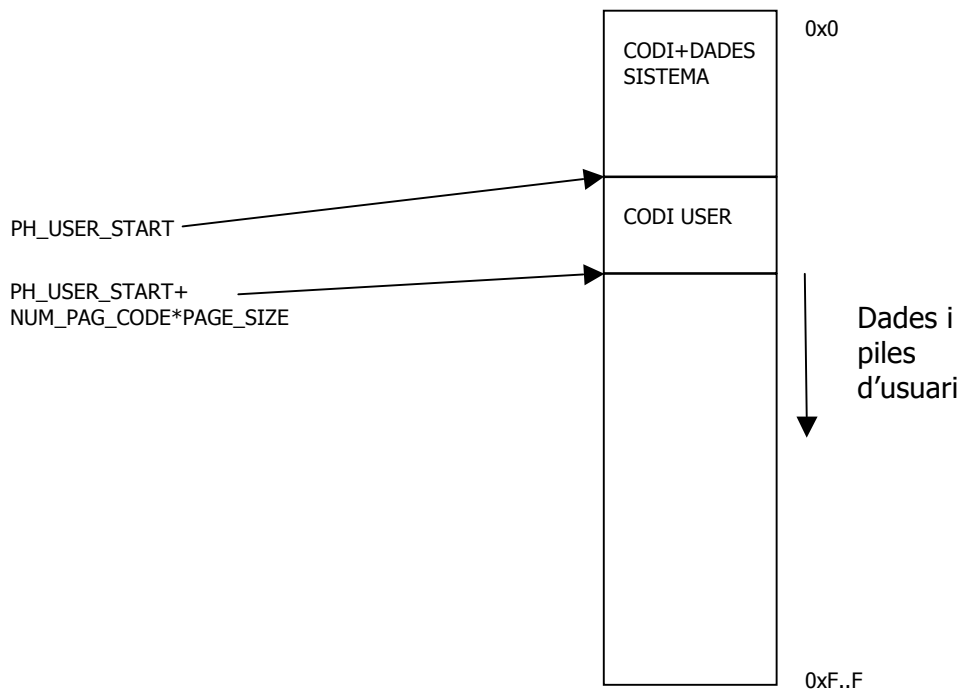


Figura 16. Memòria física a ZeOS

Fixeu-vos que la única diferència entre els diferents processos seran la traducció de les pàgines lògiques corresponents a dades+pila. **Heu d'afegir els camps que necessiteu a la task struct per saber quines pàgines físiques té assignades el procés, penseu quina informació necessiteu per poder modificar la taula de pàgines en el canvi de context.**

Pel que fa a les pàgines físiques, o frames, es calcularan a partir de les adreces físiques. Per exemple, les pàgines físiques del codi del P0 estaran a partir del frame ($PH_USER_START \gg 12$). **Definiu-vos una macro per calcular el nombre de pàgina a partir de una adreça de memòria.**

Cada cop que es creï un nou procés serà necessari buscar a la memòria física uns frames disponibles per a assignar a les dades i pila d'aquest nou procés. D'altra banda, cada cop que un procés finalitzi la seva execució caldrà marcar com a disponibles els frames de dades i pila que aquest procés tenia assignats.

Per tal de que el sistema pugui saber quins frames estan ocupats (assignats a alguna pàgina lògica) i quins frames estan lliures, hem definit la taula `phys_mem` (fitxer `mm.h`) que té una entrada per a cada frame i que ens dirà quin és l'estat del frame que representa. La inicialització d'aquesta estructura us la donem feta com a part de la rutina `init_mm` (veure el fitxer `mm.c`) i simplement marca com a ocupats els frames assignats al sistema, i els que contenen el codi d'usuari que compartiran tots els processos (veure Figura 17).

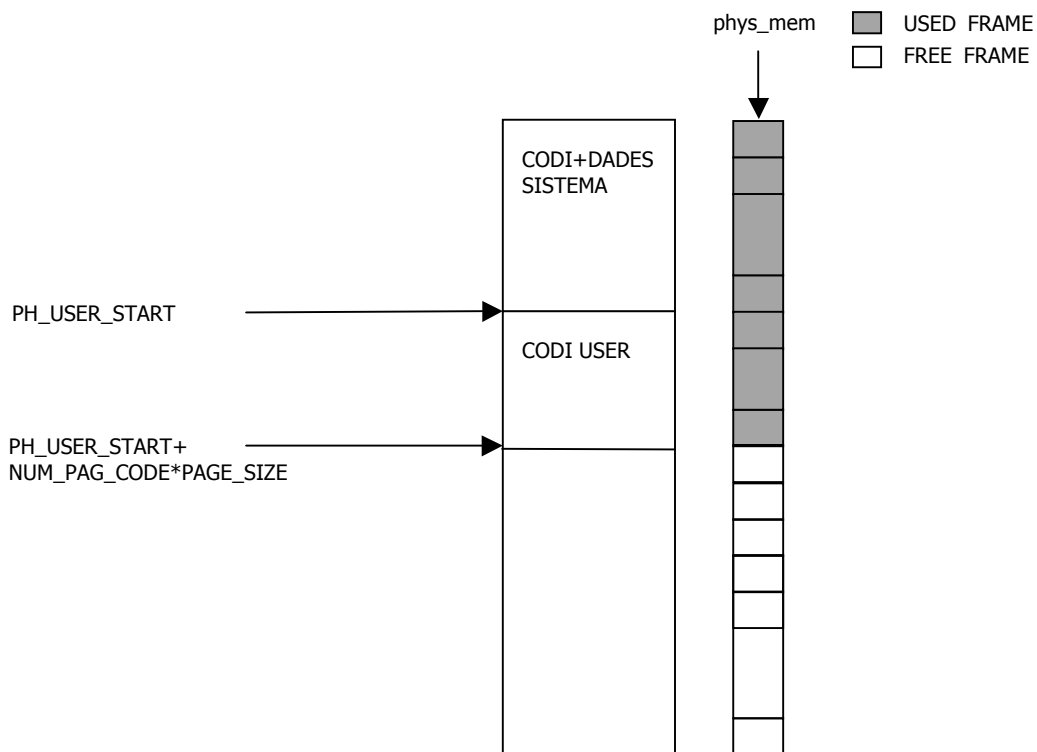


Figura 17 Inicialització de la memòria física i de la taula phys_mem

Per a completar la interfície de manipulació la taula phys_mem us donem feta la funció alloc_frame (per reservar pàgines). Comproveu el seu funcionament. **Heu d'implementar la funció free_frame. A continuació teniu la descripció de la interfície d'aquestes dues funcions.**

```
int alloc_voidframe(void)
```

La funció alloc_frame busca 1 frame disponible. Si el troba, el marca com a ocupat a la taula phys_mem i el retorna com a resultat. Si no hi ha cap cap frame lliure retorna -1.

```
void free_frame(unsigned int frame)
```

La funció free_frame marca com a lliure el frame que li passem com a paràmetre.

4. Inicialització del procés inicial

En aquesta part s'haurà d'inicialitzar el procés inicial. Consistirà en donar els valors adequats als seus camps del task_struct i inserir-lo a la **runqueue**. Aquest procés és especial, ja que s'executa des de l'inici del boot i no acaba mai. **Mai pot ser matat.**

Per la inicialització dels camps del task_struct del procés inicial, podeu consultar com estan inicialitzats els camps d'una task a Linux. A ZeOS no s'hauran d'inicialitzar tants camps, sols els que s'hagin definit al task_struct. Al fitxer ja us hem afegit la capçalera de la funció `init_task0()` i part de la implementació de la funció, **afegiu el que falti.** El procés 0 haurà de tenir, obligatòriament el PID=0.

Heu de tenir en compte que les estructures del hardware que s'havien d'inicialitzar perquè el sistema funcionés ja us les hem donat fetes:

- **Inicialització TSS (funció setTSS).** La TSS és bàsicament l'estructura que utilitza el hardware per saber on està la pila de kernel. Llegiu la secció 3.4 i després estudeu la funció setTSS per veure com s'inicialitza la pila de kernel.
- **Inicialització de la memòria (funció set_user_pages).** Llegiu detingudament la secció 3 i estudeu el codi del fitxer mm.c. Fixeu-vos que aquesta funció només inicialitza la taula de pàgines, no guarda cap informació sobre les pàgines al task_struct. **Haureu de modificar aquesta funció per a completar la inicialització de la memòria** del procés 0 guardant a la seva task_struct la informació que sigui necessària sobre les seves pàgines.

Un cop completada la inicialització, compilar la imatge i executar Bochs. Veure que el procés inicial s'inicialitza correctament i no dona cap error

Un cop completada la inicialització, compilar la imatge i executar Bochs. Veure que el procés inicial s'inicialitza correctament i no dona cap error

5. Crides a sistema per gestionar processos

En aquesta part implementarem dues crides a sistema a ZeOS relacionades amb la gestió de processos: **getpid** i **fork**. getpid retorna el pid del procés que l'invoca. fork crea un nou procés, còpia del que fa la crida (que anomenarem pare), inserir-lo a la llista de processos preparats per executar i retornar el seu pid.

5.1. Implementació de la crida "getpid"

La capçalera de la crida és:

```
int getpid(void).
```

Haurà de retornar el pid del procés que l'ha invocada.

S'ha **d'implementar la crida** seguint els passos de les crides a sistema explicades a l'entrega anterior. L'identificador de la crida getpid és el 20.

5.2. Implementació de la crida "fork"

La capçalera de la crida és:

```
int fork(void).
```

El seu valor de retorn és diferent segons si es retorna al pare o al fill. En el cas del pare, es retorna el pid del procés creat; en el cas del fill, es retorna un 0. En els dos casos, si es produís un error es retornaria un -1. Heu de recordar que el codi d'usuari serà compartit, per tant no cal modificar les entrades de la taula de pàgines que apunten a codi. Les dades+pila són privades de cada procés. S'hereten al fer el fork i a partir d'aquell moment són privades. Els passos per fer el fork són:

- 1 Implementar el wrapper de la crida a sistema. El seu identificador és el 2.
- 2 Implementar la rutina de servei sys_fork. És més complicada que les anteriors. Ha de:
 - a. Buscar una entrada lliure a la taula de processos. En cas de no haver-hi espai per un nou procés, es retornarà l'error corresponent.

- b. Herència dades de sistema: Copiar el `task_union` del pare al fill. Penseu si cal modificar la taula de pàgines per accedir a la pila de sistema del fill. Per copiar podeu usar la funció `copy_data`.
 - c. Herència dades d'usuari:
 - i. Buscar les pàgines físiques on es mapejaren les pàgines lògiques de dades+pila del procés fill (utilitzant la funció `alloc_frame` que us demanem que implementeu a l'apartat 3.5.3).
 - ii. Copiar les pàgines de dades+pila d'usuari de pare a fill. No podem accedir directament a les pàgines físiques del procés fill perquè no estan mapejades a la taula de pàgines del pare. Tampoc podem mapejar-les directament ja que corresponen a les mateixes pàgines lògiques que les del procés pare. Per tant, haurem de mapejar-les en unes noves entrades de la taula de pàgines de forma provisional (només pel fork), de forma que puguem accedir de forma simultània a les pàgines de pare i fill. Per tant, farem:
 - Utilitzar de manera temporal entrades lliures de la taula de pàgines (només per fer la còpia). Feu servir les funcions `set_ss_pag` i `del_ss_pag`.
 - Fer la còpia de les pàgines de dades+pila. Tingueu present que la vostra implementació ha de funcionar independentment del nombre de pàgines de codi i dades definides.
 - Alliberar les entrades temporals a la taula de pàgines. Consulteu la secció Translation Lookahead Buffer (TLB) 3.3 per determinar quines implicacions pot tenir això pel TLB.
 - Guardar la informació sobre els nous marcs de pàgines al `task_struct` del fill.
 - d. Assignar un nou PID al procés. Recordeu que no ha de coincidir amb l'índex del procés a la taula de processos.
 - e. Inicialitzar els camps del `task_struct` no comuns al fill.
 - Penseu quin o quins registres no seran comuns a la tornada del procés fill i modifiqueu el seu contingut a la pila de sistema perquè quan es restauri el context cada un rebi els seus valors.
 - f. Inserir el nou procés a la llista de preparats: `runqueue`.
 - g. Retornar el pid del procés fill.
- 3 Compilar la imatge i executar Bochs. Provar les crides a sistema i comprovar que les dues crides funcionen correctament.

6. Implementació del canvi de context entre processos

En aquesta entrega es tracta d'implementar una rutina que permeti efectuar el canvi de context entre dos processos. El canvi de context consisteix en canviar el procés que s'està executant per un altre, cosa que comporta, entre d'altres modificacions: canviar l'espai d'adreces d'usuari (canviar la taula de pàgines), canviar a la pila de kernel (per restaurar el nou context), i restaurar el context del nou procés. Aquesta rutina s'anomenarà **task_switch**.

La seva capçalera és:

```
void task_switch(union task_union *t)
t: punter al task_union del procés que es passarà a executar
```

Haurà de:

- a. **Actualitzar la TSS** perquè apunti a la pila de sistema de t.
- b. **Actualitzar la taula de pàgines** perquè les pàgines de dades+pila d'usuari de t siguin accessibles.
- c. **Canviar a la pila de sistema del nou procés**
- d. **Restaurar els registres.**
- e. **Cal fer EOI? (penseu si cal i on s'ha de fer)**
- f. IRET (penseu perquè fem iret i no ret)

El canvi de context s'utilitza quan el kernel planifica processos, quan la política ha decidit que un nou procés s'ha d'executar, però en aquest cas ho provarem abans per simplicitat. Compilar la imatge i executar Bochs. Provar el canvi de context. Feu la prova amb més de dos processos.

7. Destrucció de processos

L'objectiu d'aquesta part és implementar la crida a sistema **exit**. El que fa és destruir el procés que la crida. S'haurà, per tant, d'eliminar el procés de la taula de processos, alliberar els seus recursos, i fer un canvi de context.

S'ha de tenir en compte que hi ha un procés que aquesta (ni cap) crida pot eliminar, el procés inicial.

La seva capçalera ha de ser:

```
void exit(void)
```

1. Implementar el wrapper de la crida a sistema. El seu identificador és l'1.
2. Implementar la rutina de servei sys_exit.
 - a. Alliberar les estructures de dades del procés i tots els seus recursos (semàfors, ...).
 - b. Alliberar els frames de dades i pila que tenia assignat el procés (utilitzant la funció free_frame que us demanem que implementeu a l'apartat 3.5.3).
 - c. Aplicar la política de planificació i fer un canvi de context (usant la funció task_switch).
3. Provar la crida a sistema exit.
 - a. Comproveu que funciona correctament creant i destruint processos.
 - b. Comproveu el comportament del procés inicial si intenta fer un exit

8. Tractament d'una excepció: el page fault

Ara que ja podem crear i destruir processos dinàmicament, podem modificar el tractament de les excepcions de manera que afectin només al procés que l'ha provocat (i no a tot el sistema com vam implementar a la primera entrega del projecte). En concret modificarem el tractament de l'excepció de la fallada de pàgina que en el cas de ZeOS es produirà quan un procés faci un tipus d'accés no permès a memòria

(direcció no vàlida o tipus d'operació no permesa). Fins ara el tractament que tenim fa que el codi de sistema es quedi executant un bucle infinit. Ara el modificarem i emularem el tractament per defecte que executa Linux per a aquest tipus d'excepció i destruïrem els processos que generin fallades de pàgina. A més a més mostrarem per pantalla el missatge "Excepció de fallada de pàgina" indicant el pid del procés que ha generat l'excepció. En cas que sigui el procés 0 qui genera aquesta excepció, després de treure el missatge ens quedarem en un bucle infinit com fins ara.

Modifiqueu el joc de proves per a que un procés generi una fallada de pàgina i comproveu que el procés es destrueix i la resta de processos continuen la seva execució amb normalitat. Si necessiteu comprovar quina adreça intentava accedir el procés quan ha provocat la fallada de pàgina podeu consultar el valor del registre de control cr2 . Comproveu també el comportament de la gestió de fallada de pàgina per al procés inicial.

9. Planificació de processos

En aquesta part es vol afegir la política de planificació de processos. En aquest cas, implementarem una política Round Robin a ZeOS, però vosaltres heu de fer un disseny/implementació de forma que la política concreta quedi encapsulada i sigui fàcilment substituïble per un altre.

S'afegiran els camps necessaris al `task_struct` per dur-la a terme, s'afegirà la política de planificació a la interrupció de rellotge a més de la crida a sistema `nice`. També heu de tenir en compte que posteriorment, cada vegada que es bloquegi un procés, per exemple amb un semàfor, també haurem de cridar la política de planificació.

9.1. Implementació de la política Round Robin

Aquesta política de planificació de processos consisteix en executar cada procés un nombre determinat de tics (el seu "quantum"), fent un `task_switch` quan s'acabin els tics del procés en curs. Els tics s'actualitzen a la interrupció de rellotge, per tant, és aquí on invocarem la política per veure si hem de canviar de procés.

Afegir aquesta política implica:

- Controlar quants tics porta el procés actual amb la cpu. Feu servir una variable global que es decrementa a cada tic.
- Quan s'esgoti el *quantum*, actualitzar la runqueue (eliminar el current i insertar-lo de nou), i fer canvi de context al primer procés de la runqueue. Recordeu que quan un procés entra a executar després d'un canvi de context ho fa amb el seu quantum sencer.
- Provar que la política de Round Robin s'executa correctament. Això vol dir que es justa i que evita la inanició.

9.2. Funcionalitat associada a la política

Heu de pensar quina funcionalitat necessita una política de planificació i dissenyar i implementar el conjunt de funcions necessàries. El que volem és que el codi del scheduler i de la política quedin encapsulats i ben organitzats.

Per exemple, necessiteu funcionalitat per:

- Actualitzar les variables de control de la política, per exemple, en el cas de Round Robin actualitzar la variable 'tics' amb el temps d'execució del següent procés

- Consultar a la política si cal fer un canvi de context
- Escollir el següent procés a executar
- Bloquejar un procés en una cua concreta
- Desbloquejar un procés d'una cua concreta
- Etc

9.3. Implementació de la crida a sistema “nice”

La capçalera de la crida és:

```
int nice(int quantum).
```

Ha de modificar el quantum del procés que la crida (el nou quantum es tindrà en compte la propera vegada que el procés s'executi). Retorna el valor anterior del quantum si tot ha anat correctament i -1 en cas d'error.

1. **Implementar la crida a sistema.** L'identificador de la crida nice és el 34.
2. Proveu que la crida nice funciona correctament

10. Sincronització entre processos: Semàfors

En aquesta part, volem implementar crides de sincronització entre processos. Utilitzarem semàfors que ens serviran per a què diferents fragments de codi d'usuari es puguin executar en exclusió mútua. Un semàfor serà un recurs més d'un procés, veurem que quan un procés creï un semàfor, aquest procés serà el seu propietari, per tant, quan el procés mori (exit), tots els semàfors que tingui creats hauran de ser destruïts.

Haureu de declarar una taula estàtica de semàfors amb NR_SEM semàfors (heu de definir la constant NR_SEM amb valor 10), implementant per cada semàfor les estructures de dades que creieu oportunes.

Per gestionar els semàfors caldrà implementar una sèrie de crides a sistema, seguint els passos de les crides ja definides fins al moment.

10.1. Implementació de la crida a sistema sem_init

La capçalera de la crida és:

```
int sem_init (int n_sem, unsigned int value)
n_sem: identificador del semàfor a inicialitzar
value: valor inicial del comptador del semàfor
retorna: -1 si error, 0 si OK
```

L'identificador d'aquesta crida a sistema és el 21. Aquesta crida a sistema inicialitza el comptador del semàfor `n_sem` a `value`. Al mateix temps inicialitza la cua de processos bloquejats en aquest semàfor i les dades necessàries per a la seva correcta utilització. El procés que inicialitza un semàfor es converteix en el seu propietari.

El valor de retorn 0 indica una execució correcta. Si `n_sem` no és un identificador vàlid per a un semàfor o bé ja es troba inicialitzat, el valor retornat a usuari serà -1.

10.2. Implementació de la crida a sistema sem_wait

La capçalera de la crida és:

```
int sem_wait (int n_sem)
n_sem: identificador del semàfor
retorna: -1 si error, 0 si OK
```

L'identificador d'aquesta crida a sistema és el 22. Si el comptador del semàfor `n_sem` és més petit o igual que zero, aquesta crida bloqueja en aquest semàfor el procés que la ha invocat. En cas que el comptador sigui més gran que zero, aquesta crida decrementa el valor del semàfor. El procés 0 no es pot bloquejar en cap cas, per tant s'ha de considerar un error la utilització d'aquesta crida pel procés 0.

El valor de retorn 0 indica una execució correcta. Si `n_sem` no és un identificador vàlid per a un semàfor, el semàfor no està inicialitzat, o el semàfor es destruït mentre el procés que executa el `sem_wait` estava bloquejat, el valor retornat a usuari serà -1.

10.3. Implementació de la crida a sistema `sem_signal`

La capçalera de la crida és:

```
int sem_signal (int n_sem)
n_sem: identificador del semàfor
retorna: -1 si error, 0 si OK
```

L'identificador d'aquesta crida a sistema és el 23. Si no hi ha cap procés bloquejat en el semàfor `n_sem` aleshores aquesta crida incrementa el comptador del semàfor `n_sem`. En el cas que hi hagi un o més processos bloquejats sobre `n_sem`, aquesta crida desbloqueja el primer procés.

El valor de retorn 0 indica una execució correcta. Si `n_sem` no és un identificador vàlid per a un semàfor o bé el semàfor no està inicialitzat, el valor retornat serà -1.

10.4. Implementació de la crida a sistema `sem_destroy`

La capçalera de la crida és:

```
int sem_destroy (int n_sem)
n_sem: identificador del semàfor a destruir
retorna: -1 si error, 0 si OK
```

L'identificador d'aquesta crida a sistema és el 24. Aquesta crida destrueix el semàfor `sem i`, en cas que hi haguessin processos bloquejats, els desbloqueja fent que el `sem_wait` retorni -1. El valor de retorn 0 indica una execució correcta.

Si `n_sem` no és un identificador vàlid per a un semàfor, o bé el semàfor no està inicialitzat, o bé el procés que executa el `sem_destroy` no és el propietari del semàfor, el valor retornat serà -1.

Feu les proves necessàries per garantir l'ús correcte de les crides de sincronització, tant per implementar una exclusió mútua com per sincronització entre processos.

11. Informació estadística de processos

La finalitat d'aquest apartat consisteix en afegir als processos de ZeOS informació estadística. Interessa saber el temps que cada procés ha usat la cpu, el nombre de ràfegues que ha executat (és a dir, quants cops el procés ha passat de l'estat Ready a l'estat Run), i quants tics li manquen per acabar el seu quantum. Al fitxer `stats.h`

trobareu la definició d'una estructura per emmagatzemar les estadístiques de cada procés:

```
struct stats {
    int total_tics; /* Total tics executed by the process */
    int total_trans; /* Total transitions ready to run */
    int remaining_tics; /* Remaining tics to end the quantum */
};
```

Tingueu en compte que aquesta estructura serà necessària tant al codi d'usuari com al codi de sistema.

1. Afegir els camps necessaris al `task_struct` i el codi necessari per a actualitzar-los
2. Implementar la crida a sistema per a accedir a aquesta informació:

```
int get_stats(int pid, struct stat *st)
pid: identificador del procés del qual es vol consultar les estadístiques
st: adreça on l'usuari rebrà les estadístiques del procés
```

L'identificador de la crida és el 35. Retorna 0 si tot ha anat correctament i -1 en cas d'error. El paràmetre `pid` ha de ser l'identificador d'un procés **actiu** (pot ser el procés actual, un procés en estat de ready o un procés bloquejat). Les estadístiques del procés es retornen en el paràmetre `st`. Fixeu-vos que es tracta d'un punter de memòria a usuari. Penseu i controleu els casos d'error.

12. Joc de proves

Per finalitzar les proves del vostre codi feu que el `user.c` cridi a la rutina `runjp` definida a la llibreria que us proporcionem (`libjp2.a`) i comproveu que totes les proves finalitzen correctament. No oblideu que heu de modificar el `Makefile` per a enllaçar l'executable del `user` amb aquesta llibreria i assegureu-vos que no s'està intentant enllaçar al mateix temps a la `libjp1.a`.

ENTREGA 1.3: GESTIÓ D'E/S

En aquesta part del projecte continuarem ampliant el ZeOS amb una gestió bàsica d'E/S. En primer lloc us presentarem l'esquema global que introduïrem a Zeos per tal que tingueu una visió global del problema. A continuació, us demanem que **dissenyeu i implementeu** les estructures de dades i les operacions d'accés a aquestes estructures necessàries per a suportar aquest model. Tanmateix, heu de completar la implementació de l'accés a dos dispositius: la pantalla i el teclat. Finalment, completareu el **disseny i la implementació d'un sistema de fitxers** senzill afegint la gestió específica dels fitxers de dades (ZeosFAT).

Alguns criteris que heu de tenir en compte en aquesta part del projecte és que volem oferir un accés a dispositius, i més endavant també a fitxers de dades, el més homogeni possible. Per això seguirem l'aproximació de definir els tipus de dades necessaris per nivells, des de les modificacions a la `task_struct` fins a les dades de dispositius, i les operacions necessàries per accedir a elles. També heu de recordar que ZeOS està basat en Linux, podeu utilitzar aquesta referència a l'hora de dissenyar les estructures de dades.

1. Model d'E/S

El model d'entrada sortida que proposem, busca uniformitzar en la mesura del possible el tractament de tots els dispositius. Per això definirem tres tipus de dispositius: físics (el hardware), lògics (abstracció que amaga el hardware) i virtuals (interfície que ofereix el sistema operatiu al programador). Això ens permetrà dividir les operacions de gestió i accés en dos nivells diferents: operacions independents al dispositiu (o genèriques) i operacions dependents del dispositiu (o específiques). A més, tindrem un únic espai de noms per tots els dispositius lògics

1.1. Dispositius físics

Són els que es corresponen amb el hardware. Cadascun necessita funcions de gestió específiques. Per oferir un accés el més homogeni possible **definirem un tipus de dades** genèric amb els punters a les funcions d'accés específiques del dispositiu (li direm `file_operations`). Així doncs, per a cada dispositiu físic que suporti ZeOS **heu d'inicialitzar una estructura de `file_operations`, implementar les funcions específiques, i fer que des de les funcions independents es cridi a la funció específica corresponent.**

A ZeOS suportarem dos tipus de dispositius físics: pantalla i teclat.

- La pantalla. L'únic mode d'accés vàlid serà el d'escriptura. La crida dependent ja la coneixeu: `sys_write_console`
- El teclat. Mantindrà un buffer amb les tecles introduïdes. L'únic mode d'accés vàlid serà el de lectura. La gestió del teclat és una mica més complexa ja que és bloquejant i per si sola inclou definir nous tipus de dades. A la secció 2.2 us expliquem els detalls. La funció per llegir de teclat li direm `sys_read_keyboard`.

1.2. Dispositius lògics

Els dispositius lògics són una abstracció que independitza al programador i a gran part del sistema operatiu del hardware que hi ha instal·lat. Seguirem la filosofia Linux i

representarem tots els dispositius lògics com fitxers. Això vol dir que tots els dispositius compartiran l'espai de noms amb els fitxers de dades (**espai de noms únic**). Per a simplificar el disseny, farem que un dispositiu lògic no pugui tenir més d'un nom alhora (és a dir, no suportarem varis *links* simultanis com fa Linux).

A més del nom, els dispositius lògics podran tenir més característiques, unes d'elles estàtiques (no depenen de l'ús que s'està fent del dispositiu) i unes altres dinàmiques (descriuen l'ús del dispositiu). **Heu de decidir com emmagatzemar aquestes característiques i implementar les operacions d'accés necessàries.**

Com a **característiques estàtiques** tindran (com a mínim) el nom (màxim FILE_NAME_SIZE=10 caràcters per simplificar), el mode d'accés vàlid (lectura/escriptura/lectura&escriptura), i el punter al tipus *file_operations* que ens definirà les operacions específiques del dispositiu associat. Com que no tenim usuaris a ZeOS, no inclourem informació sobre permisos d'accés per diferents usuaris.

Com a **característiques dinàmiques** podran tenir, per exemple, la posició seqüencial dins el dispositiu (per implementar tipus d'accés seqüencial com els fitxers de dades), el mode d'accés amb el que s'ha iniciat l'ús del dispositiu.

A l'hora de definir les estructures de dades que emmagatzemaran aquestes característiques, haureu de tenir en compte:

- La possibilitat de que dos o més processos estiguin treballant alhora amb un mateix dispositiu. En aquest cas, estudeu les avantatges que es poden obtenir de la compartició de les estructures de dades de les característiques estàtiques.
- La possibilitat d'accedir a un mateix dispositiu des de diferents processos, compartint les dades d'accés (per exemple al duplicar un canal o heretar la taula de canals del pare) o sense compartir les dades d'accés, inclús des del mateix procés (per exemple, al invocar diferents vegades l'open del dispositiu). Estudieu en aquest cas, si s'han de mantenir diferents característiques dinàmiques o no i les implicacions que té.

1.2.1. Directori: organització de l'espai de noms a ZeOS

L'estructura de dades que permet associar el nom i les característiques (estàtiques i dinàmiques) d'un dispositiu és el directori. En un sistema més complet el directori es guardaria a disc per què no fos volàtil. En el nostre cas, per simplicitat, el mantindrem només en memòria. Per començar, definirem un esquema de directori molt senzill. Hem de **dissenyar un directori d'un únic nivell** (sense subdirectoris), amb una mida màxima prefixada (10 entrades). Els dispositius lògics que representen a la pantalla i al teclat estaran pre-creats (creeu-los al finalitzar la càrrega del kernel, utilitzeu els noms "DISPLAY" i "KEYBOARD" respectivament). **Haureu d'inicialitzar les entrades corresponents al directori i les estructures de dades amb les característiques dels dos dispositius.**

1.3. Dispositius virtuals (canals)

Els processos accediran als dispositius lògics mitjançant dispositius virtuals (ens referirem també com *fd*, *file descriptors* o canals). Quan el programador vol accedir a un dispositiu lògic, primer ha de crear un dispositiu virtual associat a aquell dispositiu lògic (a Linux es pot fer amb la crida a sistema *open*). Els dispositius virtuals són **locals als processos**. Cada procés ha de tenir una taula de canals per a mantenir l'associació dels dispositius virtuals amb els lògics (amb una mida que limitarem a 10 entrades). Un procés no pot modificar la taula de canals d'un altre procés. El cas estàndard és que cada procés tingui tres canals inicials: **stdin** (entrada estàndard al fd

0), **stdout** (sortida estàndard al fd 1), i **stderr** (sortida estàndard d'error al fd 2). Això, però, no és una situació fixa ja que després el procés pot modificar la taula de canals, per exemple, obrir un nou fitxer, tancar-lo o duplicar-lo (detalls a la secció 1.4).

La semàntica dels sistemes basats en UNIX també defineix que la **taula de canals s'hereta al fer un fork**. Heretar vol dir que **el nou procés es crearà amb una taula de canals idèntica a la taula de canals del procés pare. Per tant, l'ús dels dispositius que en el moment de la creació estan referenciats des de la taula de canals serà compartit.**

1.4. Operacions de gestió independents del dispositiu

En aquest apartat es descriu la interfície que farà servir el programador per associar un dispositiu virtual amb un dispositiu lògic (open i dup) i per a desfer aquesta associació (close). També es descriu l'operació de lectura independent (read) (l'operació independent d'escriptura (write) ja l'hem definit a l'entrega 1).

1.4.1. open

```
int open( const char *path, int flags)
    path: nom del fitxer a obrir
    flags: mode d'obertura del fitxer (O_RDONLY,O_WRONLY,O_RDWR)
    retorna: -1 si error, o descriptor de fitxer vàlid
```

La crida `open` permet associar un dispositiu lògic identificat per un nom amb un dispositiu virtual. Cada vegada que s'executa un `open` sobre un dispositiu aquest ens **retornarà un nou file descriptor (fd) local al procés**. Haurà de localitzar la informació del dispositiu al directori i obtenir les seves característiques estàtiques. A més, cada `open` representarà un nou ús del dispositiu i per tant inicialitzarà unes noves característiques dinàmiques. Per exemple, el mode en que s'ha obert i un punter de lectura/escriptura que ens definirà la posició actual en aquells dispositius que ens ofereixin accés seqüencial. El fd retornat serà el primer lliure que tingui el procés. Tindrà l'identificador 5 de crida a sistema.

1.4.2. read

```
int read (int fd, char *buffer, int size)
    fd: canal d'on es llegirà
    buffer: punter on es copiaran les dades
    size: mida del que es llegirà
    retorna: -1 si error
           nombre de bytes llegits si OK
```

Llegeix `size` bytes del dispositiu virtual identificat per `fd` i deixa els bytes llegits en `buffer`. El seu identificador de crida a sistema és el 3. Haurà de cridar a la funció depenent del dispositiu lògic associat. En cas que la crida hagi anat bé, es retorna el nombre de bytes llegits. Altrament, es retorna el codi d'error -1. Les dades es llegiran a partir de la posició actual de lectura/escriptura. A la secció 2.2 us donem detalls tècnics per implementar la crida `read` específica del teclat.

1.4.3. dup

Aquesta crida duplica el canal `fd`, que rep com a paràmetre, a la primera posició lliure de la taula de canals del procés. Al ser una dúplica, el descriptor de fitxer retornat i el d'entrada comparteixen les característiques dinàmiques relacionades amb l'`open` (punter de lectura/escriptura per exemple).

```
int dup (int fd)
    fd: canal a duplicar
retorna: -1 si error, nou canal si OK
```

El seu identificador és el 41.

1.4.4. close

```
int close (int fd)
fd: canal a tancar
retorna: -1 si error, 0 si OK
```

El seu identificador és el 6. Desfà l'associació entre el fd indicat i el dispositiu lògic. Heu d'actualitzar totes les estructures de dades implicades.

1.5. Què heu de fer....

- Definiu els nous tipus de dades que necessitem per gestionar els dispositius (físics, lògics i virtuals). Decidiu i definiu qualsevol tipus de dades addicional que creieu convenient. Modifiqueu les ja existents, com la `task_struct`, com convingui.
- Definiu les operacions per a manipular les estructures de dades.
- Definiu e inicialitzeu els dispositius lògics amb nom "DISPLAY" i "KEYBOARD", que representaran respectivament als dispositius físics pantalla i teclat. Inicialitzeu el directori amb aquests dos dispositius.
- Actualitzeu l'ús de la crida `sys_write_console` per a què usi el nou model.
- Modifiqueu les crides a sistema ja implementades que es vegin afectades per les noves estructures de dades.
- Definiu les constants que necessitareu per a indicar el mode d'obertura dels fitxers: `O_RDONLY` (0x1), `O_WRONLY` (0x2) i `O_RDWR` (0x3)
- Inicialitzeu l'`stdin`, `stdout` i `stderr` del procés inicial. Les estructures de dades han de quedar com si hagués executat: `open("KEYBOARD",O_RDONLY); open("DISPLAY",O_WRONLY); open("DISPLAY",O_WRONLY);`
- Implementeu les crides a sistema demanades.

Un cop fet això, podreu provar tot el model. Feu un joc de proves per veure que es compleixen totes les especificacions demanades.

2. Operacions de gestió dependents del dispositiu teclat

L'objectiu principal d'aquesta secció és descriure la implementació de la crida `read` (lectura). Es vol que el sistema pugui llegir del teclat, i d'aquesta manera se'l dotarà de certa interacció amb l'usuari.

Caldrà ampliar la funcionalitat de la interrupció de teclat que vam programar a la primera part del projecte per tal d'implementar un sistema de buffering intermedi (serà un buffer circular) de forma que els caràcters llegits es guardaran en aquest buffer a la interrupció de teclat. A la crida `read` mirarem si hi han caràcters suficients al buffer de

teclat (i cap altre procés esperant dades). En cas contrari bloquejarem el procés fins que hi hagin caràcters suficients.

Com a conseqüència de tota aquesta gestió, s'hauran d'implementar noves estructures de dades. Aquestes permetran gestionar els processos en execució, els que es dediquen a la lectura, etc...

Resumint, la feina a fer es pot dividir en varies parts molt diferenciades:

- Implementació de les noves estructures de dades
- Ampliació del tractament de les interrupcions del teclat

↳ **Important: Per evitar que es doni la situació que tots els processos es trobin bloquejats, farem que el procés 0 no es pugui bloquejar. Tingueu en compte aquesta simplificació en els vostres jocs de proves.**

2.1. Estructures de dades a definir

- Keyboardqueue: en aquesta cua s'aniran afegint els processos que es bloquegen pel teclat.
 - Tenint en compte que hi ha diferents punts als sistema on els processos es poden bloquejar/debloquejar, seria convenient tenir encapsulada aquesta funcionalitat en dues funcions.
- Buffer circular: s'implementarà sobre un vector de mida prefixada. Sobre aquest buffer s'hauran de definir les operacions necessàries per tal de que compleixi els requeriments d'un buffer circular tipus FIFO. Definiu el nou tipus de dades i les operacions per accedir-hi.

2.2. Implementació de l'operació específica de lectura pel teclat

Hem de garantir que les lectures es fan en ordre seqüencial, és a dir, si dos processos fan la crida read, fins que el primer que la faci no rebi els N caràcters demanats no es donaran caràcters al segon.

Els passos a fer són:

- Comprovar els paràmetres
- Accedir a les operacions específiques de lectura definida al file_operations. Aquesta funció haurà de:
 - Comprovar si hi ha **suficients caràcters al buffer** circular i **no hi ha cap procés bloquejat** esperant dades, copiar els N caràcters (size) a l'adreça apuntada per buffer i tornar el nombre de caràcters llegits. Cal actualitzar el buffer circular.
 - En altre cas:
 - Apuntem la informació necessària a la task_struct per gestionar la petició de caràcters: quants caràcters volem llegir, etc (**penseu quina informació cal guardar i on**)
 - Bloquegem el procés i apliquem la política de planificació per passar a executar el procés que toqui.

Penseu que un cop el procés està bloquejat, serà desbloquejat per la rutina d'atenció a la interrupció de teclat, que es qui detectarà que l'entrada de dades que esperava el procés ha finalitzat.

2.3. Rutina de servei de teclat

En aquest apartat cal ampliar la funcionalitat de la rutina de servei a la interrupció de teclat per tal que implementi un sistema de buffering. El que farem és que només copiarem els caràcters llegits a **espai d'usuari** en dos casos:

- El buffer està ple i hi ha processos esperant dades
- El buffer no està ple però el primer procés de la cua de bloquejats ja té tots els caràcters que demanava

La seqüència seria la següent:

- Comprovar si hi ha algun procés bloquejat esperant la lectura de caràcters.
- **Si no hi ha cap procés bloquejat:**
 - Es guarda el caràcter en el buffer circular. Si el buffer està ple, aquest caràcter es perd.
- **Si hi ha algun procés bloquejat:**
 - Mirarem **quants caràcters** li mancaven per llegir al primer procés bloquejat.
 - ◆ Si la lectura ha finalitzat, copiarem els caràcters a l'adreça corresponent. **MOLT IMPORTANT:** Heu de tenir en compte que la interrupció de teclat s'executa en el context d'un procés i li copiem dades a un altre procés. Haurem de canviar temporalment la taula de pàgines per poder accedir. A més, desbloquejarem el procés. Caldrà també actualitzar el retorn de la crida read.
 - ◆ Si no ha finalitzat però el buffer està ple igualment li copiarem els caràcters però actualitzarem les dades que teníem al `task_struct` que ens indicaven quants caràcters volíem llegir, etc. **Penseu que cal actualitzar.**

↳ **Nota:** Recordeu quin era el procediment per copiar informació entre diferents espais d'adreces

2.4. Que heu de fer....

- Afegiu els nous tipus de dades
- Modifiqueu la rutina d'atenció a la interrupció de teclat
- Modifiqueu la crida a sistema read
- Proveu que tots els casos del read funcionen correctament

3. Sistemes de fitxers: ZeOSFAT

En aquesta última part introduïrem un sistema de fitxers a ZeOS semblant al FAT: ZeOSFAT. A més de dissenyar e implementar aquest sistema de fitxers haureu de fer les modificacions necessàries a l'esquema de la primera part per tal que els fitxers de la ZeOSFAT formin part del mateix espai de noms que els fitxers que representen els dispositius. A més, us demanarem que implementeu alguna crida a sistema nova així com alguna modificació a les ja existents.

3.1. El sistema de fitxers

Un sistema de fitxers defineix la forma de gestionar l'espai de disc, tant per assignar espai de disc als fitxers quan aquests creixen, com per gestionar l'espai lliure, com per organitzar els blocs de dades dels fitxers per optimitzar el seu accés. Tot això es fa de forma transparent a l'usuari, que sempre veu les mateixes crides a sistema.

Per evitar tenir que tractar amb el disc físic, implementarem un disc a memòria on hi instal·larem el nou sistema de fitxers. El disc estarà format per un conjunt de MAX_BLOCKS blocs de mida fixa de 256 bytes.

Per gestionar un disc s'han de dissenyar noves estructures de dades en funció dels **criteri d'assignació d'espai** i de **gestió de blocs lliures** que vulgueu utilitzar.

3.1.1. Assignació d'espai

Nosaltres us proposem que utilitzeu com a criteri d'assignació d'espai una **assignació per blocs**: cada vegada que necessitem espai addicional escollirem un bloc. Aquest sistema no pateix de fragmentació externa, però implica que hem de gestionar els blocs que pertanyen a un mateix fitxer. Com a sistema de gestió utilitzarem un **sistema d'assignació encadenament en taula tipus FAT** (File Allocation Table). En aquest opció, els blocs de dades estan encadenats, cada bloc apunta al següent, però els apuntadors estan en una taula apart. Necessitarem, per cada fitxer, un apuntador al primer bloc de dades de cada fitxer.

3.1.2. Gestió de l'espai lliure

Podeu utilitzar la mateixa FAT per gestionar els blocs lliures.

3.2. Operacions dependents del dispositiu

Els fitxers de dades a ZeOSFAT seran una seqüència de bytes. La interpretació de les dades se la donarà el codi d'usuari, no el sistema de fitxers ni el kernel. Per accedir-hi s'utilitzaran les crides a sistema d'entrada/sortida. Per tant, no hi haurà cap marca ni caràcter especial de fi de fitxer.

El nostre sistema de fitxers ens permetrà guardar fitxers de dades en aquest disc. Un fitxer ocuparà un o més d'un bloc. Per simplicitat, mantindrem el mateix sistema de directori que haurem dissenyat a la primera part d'aquesta tercera entrega ampliat amb les dades que necessiteu per poder gestionar els fitxers de ZeOSFAT (per exemple la mida del fitxer).

Els fitxers de dades de ZeOSFAT es veuran com un nou dispositiu, això vol dir que per accedir-hi haurem de definir les operacions específiques que oferirem. Aquestes operacions seran:

- open: per començar a utilitzar el fitxer. Marca que hi ha un nou fd apuntant al fitxer.

- read/write: lectura/escriptura seqüencial de les dades. A més de l'adreça a l'espai d'usuari i la mida a copiar, rebrà com a paràmetre d'entrada/sortida la posició actual de lectura/escriptura i retornarà el nombre de bytes llegits/escrits.
- release: per marcar que s'està tancant un fd que apuntava al fitxer.
- unlink: per alliberar les dades d'un fitxer quan l'eliminem. No es podran eliminar les dades d'un fitxer mentre hi hagi algun procés amb un file descriptor que apunti a aquest fitxer.

Penseu quina ha de ser la nova definició del tipus `file_operations` per tenir en compte els requeriments dels fitxers de dades. I modifiqueu la inicialització dels dispositius que ja teníeu per tal que s'ajustin al nou tipus.

El sistema de fitxers s'haurà d'inicialitzar durant la inicialització de ZeOS i a partir d'aquest moment l'usuari ja el podrà fer servir normalment.

3.2.1. Inicialització de ZeOSFAT

```
int initZeOSFat(void)
retorna: -1 si error, 0 si OK
```

Inicialitza tot el sistema de fitxers amb un directori arrel '/' buit, sense cap fitxer. A partir d'aquest moment, l'usuari ja pot fer servir el sistema de fitxers normalment.

3.3. Crides a sistema

3.3.1. open

Ampliarem la crida a sistema ja implementada amb els flags `O_CREAT(0x4)` i `O_EXCL(0x8)` que es podrà afegir amb una OR lògica | a qualsevol de les opcions que ja existien. Quan el flag `O_CREAT` sigui present s'haurà de crear un nou fitxer de dades al sistema de fitxers ZeOSFAT. Si el fitxer ja existeix, s'obrirà. En canvi amb la combinació `O_CREAT|O_EXCL`, si el fitxer ja existeix, donarà error. En el cas de que estiguem creant el fitxer, el flag de mode especificat (`O_RDONLY`, `O_WRONLY`, `O_RDWR`) indicarà el mode d'accés vàlid pel fitxer mentre existeixi. Fixeu-vos que aquesta manera d'especificar els permisos dels fitxers és diferent que a Linux.

3.3.2. read/write/close/dup

Si a la primera part d'aquest entrega heu fet un bon disseny, possiblement no haureu de modificar res d'aquestes crides. Comproveu que realment és així.

3.3.3. Esborrar fitxers

```
int unlink(const char *path)
path: path al fitxer que es vol esborrar
retorna: -1 si no s'ha pogut esborrar o 0 si OK
```

Esborra el fitxer `path` del sistema de fitxers. El seu identificador és el 10. Si el fitxer està obert per algun procés, no es podrà esborrar. Per tant, heu de controlar quants processos tenen referències a cada fitxer.

4. Joc de proves

Per finalitzar les proves d'aquesta part del projecte us **proporcionem 6 llibreries**: `libjpopen.a`, `libjppwrite.a`, `libjppread.a`, `libjppclose.a`, `libjppdup.a`, `libjppunlink.a`. Cada llibreria

implementa la seva pròpia rutina runjp que s'encarrega de provar una de les crides a sistema. És a dir, **per generar l'executable del user només podeu enllaçar una** d'aquestes llibreries al mateix temps (tampoc es poden utilitzar simultàniament amb les llibreries libjp1.a i libjp2.a).