Apunts finals de SO2 (i SOA)

Aquests apunts inclouen tot el temari del QP2020 (el del coronavirus) d'SO2/SOA

Index =====

---

1. Introducció

Sistema operatiu: Capa entre el software i el hardware.

Gestiona:

- Processos
- Memòria
- I/O i sistema de fitxers
- Multiprocessadors

1.1 Boot

Proces de boot: Power –> BIOS –> Bootloader -> OS

Power: Reset de tots els dispositius i carrega de la BIOS que està harcodejada a la placa. Cal carregar-la en memòria.

BIOS(`Basic Input Output System`): Detecta i inicia els dispositius hardware. Des d'un dispositiu bootable que s'escull, carrega la partició marcada amb el flag de boot. Aquesta té 512 bytes i conté el bootloader.

Bootloader: Carrega la imatge del kernel a memòria. Un cop carregat l'executa.

OS kernel: Inicia el SO. Estructures internes, hardware necessari etc.

---

2. Mecanismes d'entrada al sistema

Cal separar el codi provilegiat del codi d'usuari. Calen mecanismes per comunicar aquests dos codis. El sistema monopolitza tot l'accés a dispositius i prevé que els usuaris hi accedeixin de manera no controlada.

Instruccions privilegiades: Aquelles que només pot usar l'SO. Quan una d'elles és executada, el HW comprova que estigui sent executada per codi de sistema. Si no es genera una excepció.

`La quantitat de nivells d'execució que pot haver-hi és definit per l'aquitectura.`

Crida a sistema: Mecanisme de l'usuari per demanar recursos al sistema.

2.1 Interrupcions / Excepcions

Aquestes permeten canviar el nivell de privilegi de l'execució mentre duri el seu codi.

N'hi ha de tres típus:

- ***Excepcions***: Síncrones i produides per la CPU després de l'execució d'una instrucció.
- ***Interrupcions HW***: Asíncrones i produïdes pel HW.
- ***Traps***: Interrupcions SW. Són síncrones. Les cridem amb la instrucció `int`.

2.2 IDT (Instruction Descriptor Table)

Taula que gestiona les interrupcions. Té 256 entrades:

# entrada	Contingut
0	@handler_excepcio_0
...	...
31	@handler_excepcio_31
32	@handler_int_hw_0
...	...
47	@handler_int_hw_14
48	@handler_trap_0
...	...
255	@handler_trap_206

El (trap) syscall_handler està a la posició 0x80.

```
mov %eax, 4
int 0x80
```

Cuan cridem a la *trap* 0x80 al registre `%eax` hi haurà el numero de la syscall per poder consultar a la ***syscall_table***.

syscall_table: Per cada entrada de la taula té l'adressa a una rutina a executar.

2.3 TSS (Task State Segment)

Estructura de dades que guarda informació sobre l'estat del sistema. Conté informació com per exemple la ***base de la pila*** de sistema.

En principi cada procés n'hauria de tenir una però linux només en té una per mantenir distància amb l'arquitectura. Això és una pijada d'intel.

2.4 Gestio de les interrupcions

Registres bàsics del sistema i que cal sempre tenir en compte al canviar de context:

```
eip: instruction pointer
cs: code segment. Són conté el segment de memòria de codi que s'està executant.
flags: Estat del processador
esp: stack pointer
ss: stack segment. Conté el segment de memòria de dades on es troba la pila.
```

Gestió en dues fases. El codi de control + la rutina:

Instrucció `int`: Accepta com a paràmetre el index a la taula IDT. Compara permissos i guarda els registres a dalt mencionats en el mateix ordre i carrega el valor d'aquests de la TSS per poder accedir al codi de sistema. Aquesta crida és l'entrada al codi de sistema i l'elevació de privilegis.

```
Interrupt -> IDT(numero int) -> handler -> routine()
\________/  \_______________________________________/
(user/hw/CPU)              (sistema)
```

Handler: Escrit en assabler. Depèn de l'aquitectura. És el punt bàsic d'entrada al sistema, fa la gestió del hw necessaria i crida a la rutina de la interrupció. Té els seguents passos:

- **Save All**: Guardar els registres del sistema (tots ells).
- **EOI (Només per les interrupcions HW)**: Notfica al PIC contoller que s'acabo la interrupt. Es fa aquí per si hi ha un canvi de tasca en plena interrupció (no es podria tractar cap altre int hw fins que es reprengues l'execucio de la tasca altre cop).
- **Crida** a la rutina.
- **Restore All**: Es restaura els registres que s'han guardat amb el *save all*
- **Retorn**. Es fa per mitja de la comanda `iret`.

Exemple de handler d'una interrupció HW:

```
ENTRY(clock_handler)
  SAVE_ALL;
  EOI;
  call clock_routine;
```

```
    RESTORE_ALL;
    iret;
```

Exemple d'un syscall handler:

```
  ENTRY(system_call_handler)
    SAVE_ALL
    cmpl $0, %eax
    jl err
    cmpl $MAX_SYSCALL, %eax
    jg err
    call *sys_call_table(, %eax, 0x04)  //taula de syscalls
    jmp fin
err:
    movl $-ENOSYS, %eax
fin:
    movl %eax, 0x18(%esp)
    RESTORE_ALL
    iret
```

2.5 Interrupcions HW

- No tenen codi d'error
- Necessiten el EOI per notificar al ***PIC*** (Programmable Interrupt Controller) de s'ha acabat la interrupció que estava gestionant.

2.6 Excepcions

Cal tenir en compte que algunes d'elles guarden un codi d'error sobre de la pila. Sobre de l'eip quan empilen. després de restaurar el context hw (restore all) el borren.

2.7 Syscalls

S'usa la instrucció `int 0x80` (a windows s'usa 0x2e) amb el codi de la syscall guardat en `%eax` per cridar-les.

> **Paràmetres:** S'usa en aquest ordre `ebx`, `ecx`, `esi`, `edi`, `ebp`, i la pila a partis d'aquí. (en windows `ebx` ens apunta a la regió de memòria on estan).
>
> **Retorn:** `eax` contindrà un valor positiu o un codi d'error en negatiu.

Cal una forma fàcil perquè l'usuari pugui interactuar amb elles -> ús de ***wrappers***.

> **Wrapper:** Funció responsable del pas de paràmetres, identificar la crida a sistema demanada consultant el valor de `eax` i cridar al trap 0x80 (`syscall_handler`). Un cop retorni al crida a sistema,

retornarà el resultat. **Si hi ha hagut un error, posarà el codi d'error a la variable `ERRNO` i retornarà -1**.

Es complica doncs una mica més el fluxe:

```
syscall -> wrapper -> IDT(numero int) -> syscallHandler(eax) -> sysCallTable -> rutina
\_________________/   \_____________________________________________________________/
       (user)                                (sistema)
```

2.7.1 Fast Syscalls

Per mitja de la crida a `sysenter` i `sysexit` en comptes de `int`.

Ens permet estalviar-nos les comprovacions de permissos de la IDT i fer més senzill el fluxe de dalt accedint directament a la *syscall_table*. Això ho podem fer perque sempre que executem una syscall sóm usuari que demana recursos al sistema, això és dona per controlat.

SYSENTER_EIP_MSR: Registre que guarda l'adreça de la `syscall_table`. Necessari per fer el canvi al codi de sistema.

SYSENTER_ESP:MSR: Apunta a la base de la TSS. S'usa per poder carregar el punter `esp0` que apunta a la base de la pila de sistema.

Canvis en el wapper: Com que no fem cap crida a `int`, coses que feia el HW les haurem de fer a manija. Haurem de fer el `save_all` i el `restore_all` i passar correctament els valors eip i esp al MSR.

3. Espai de direccions d'un procés

3.1 Generació d'executables

- **Fase 1:**
  - **Compilació:** D'alt nivell a codi objecte
  - **Montatge:** Creació d'executables a partir de codi objecte i llibreries. La diferència entre l'obkecte i l'executable són les adreces. Les del objecte són relatives a l'inici de l'objecte. L'executable te unes capçaleres amb els segments definits.
- **Fase 2:**
  - **Carregador:** Carrega l'executable en memòria allà on cal.

3.2 Espais d'adreces

Espai de direccions lògic del processador: Rang de direccions a les que pot accedir un processador. Depèn del bus. **Espai de direccions lògic del procés:** Espai d'un procés en execució. Són les adreces que usa el processador. **Espai de direccions físic del procés:** Espai d'adreces de memòria física associades a les adreces lògiques del procés.

Cal doncs una traducció -> **MMU** Memory Management Unit. Unitat de HW per fer la traducció.

3.3 MMU (Memory Management Unit)

Unitat encarregada de la traducció d'adreces.

```
CPU -> unitat de segmentació -> unitat de paginació -> memòria física
  @lògica                @lineal                 @física
       \_____________________________________________/
                              MMU
```

Un procés està dividit en segments i un segment està dividit en pàgines. Els segments normalment són *codi*, *dades*, *pila* i *heap*.

- **Unitat de segmentació:** Usa la taula de segments que apunta a la base de cada segment. Aquesta taula és única per proces. Aquesta taula s'anomena *GDT*.
- **Unitat de paginació:** És l'encarregada de fer la paginació. *Com ho fem?*

  · · · Usa la **taula de pàgines** -> una entrada per pàgina i una taula per procés. -> 4 MB de taula per cada procés, és molt gran, tenint en compte que s'ha de guardar en memòria. -> Per alleuregir pes estaa organitzada en un directori

  Directori: És una taula de taules de pàgines. Ens permet multinivell. D'aquesta manera ens assegurem que només tenim les taules de pàgines que ens calen. EL registre `cr3` indicarà en tot moment on es troba aquest directori. Cada entrada del directori és una taula de pàgines. L'entrada al directori serà indexada pels bits de major pes de l'adreça lògica. D'aquesta manera usem bits que abans no usavem (una adreça física té 20 bits i una de lògica en té 32 així que usarem aquests 12 per indicar l'entrada del directori, és a dir, per indicar la taula de pàgines a usar).

  TLB: Translation Lookaside Buffer. Ens permet tenir una caché de les adreces traduides. Per invalidar-la n'hi ha prou amb canviar el valor del registre `cr3`.

4. Gestió de processos.

procés: Executable carregat en memòria i en execució. Caracteritzat per:

- L'execució d'una seqüència d'instruccions
- Un estat actual (registres).
- Conjunt associat de recursos (disc, pantalla, memòria)

4.1 Estructures de dades

PCB: Process control block. Estructura del sistema que guarda el context d'execució. Entre d'altres conté:

- Identificador del procés (PID).
- Estat del procés
- Recursos del procés (pàgines de memòria, fitxers oberts...)
- Estadístiques del procés. (les stats de zeos)
- Informació pel planificador, com el quantum o la prioritat.
- Context d'execució.
- Altres que no es mencionen en aquesta assignatura Exemple zeos, la realitat és molt més complexa: `C struct task_struct { //PCB int PID; /* Process ID. This MUST be the first field of the struct. */ page_table_entry * dir_pages_baseAddr; struct list_head list; unsigned long *kernel_esp; unsigned long quantum; //per la planificacio dels processos struct stats estat; int nice; // 0 => prioritat ; 1 => no pL_HEAP_STARTrioritat void *brk; //per la gestio de la mem dinamiq };`

Pila de sistema: Cal una pila de sistema per a cada procés. El cap esp0 de la TSS apunta a la base d'aquetsa pila.

EL PCB i la pila es guarden en una Union anomenada `task_union`

Task Union: Union del PCB + la pila del sistema. Es fa amb un union!! No s'usa struct simplificar les qüestions d'adreces (permet l'ús de la crida current(). Veure punt 4.2.1). D'aquesta manera sabem que el top de la pila és el principi del PCB. `C union task_union { struct task_struct task; unsigned long stack[KERNEL_STACK_SIZE]; /* pila de sistema, per procés */ };` **Les `task_union` s'agrupen en un vector. En linux normal està en memòria dinàmica. En zeos té 10 posicions i s'anmena `task`**

Com s'organitzen els PCBs?

Els PCBs s'agrupen en llistes doblement encadenades. En zeos la llista és el camp `list` del PCB:

- **Ready:** Conté els processos que esperen a que el planificador els dongui pas a la cpu .
- **Free:** Conté aquells PCBs lliures preparats per quan un nou procés els necessiti.
- **Run:** Procés/os en execució. En zeos no existeix. Estar *run* és igual a no tenir

4.2 Operacions

Operacions que es poden fer ambn els processos

4.2.1 Identificació `int getpid`: Crida a sistema que ens retorna l'identificador del procés.

Com se sap quin procés s'està executant? - Windows: Hi ha una cua de RUN que ens indica quin procés s'executa per cada processador. - Linux: Es calcula amb el punter a la pila. Les `task_union` estan alineades a pàgina (els útlims 8 bits estan a 0). Per tant només cal fer una màscara amb l'`esp` de la pila de sistema:

```
int sys_getpid()
{
return current()->PID;  //La crida a current fa aquesta mascara mencionada.
}
```

4.2.2 Canvi de context Es basa en guardar el context d'un procés per poder-lo executar més tard i passar a executar-ne un d'altre.

Es guarda el context en la pila de sistema i al PCB es guarda la posició de la pila que permet recuperar aquest context.

Cal guardar:

- Context HW (registres necessaris): Els recursos HW són compartits.

No cal guardar:

- Espai de direccions del mode usuari: tenim la info necessaria al PCB.
- Espai de direccions del kernel: Pila. Perquè la TSS és única i la la pila la podrem calcular de tornada amb l'adreça del PCB.

Restaurar un procés - TSS: Ha d'apuntar a la base de la pila del nou procés - Context HW: Cambiar l'`esp` perquè apunti al context del procés i restaurar-lo. - Execució: Carregar al PC la direcció del noy codi a executar.

Com ho implementem? -> `task_switch`

Ens caldran dues funcions:

`task_switch`: Aquesta funció és un wrapper que guarda i restaura els registres `esi`, `edi` i `ebx` perquè els perdríem. A més, ens cal aquesta funció perquè cal guardar una adreça de retorn controlada a la pila.

```
ENTRY(task_switch)
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    pushl %ebx
```

```
    pushl 8(%ebp)
    call inner_task_switch
    addl $4, %esp
    popl %ebx
    popl %edi
    popl %esi
    popl %ebp
    ret
```

inner_task_switch: Fa el switch en si. Té un funcionament complexe:
1. Carreguem el camp `esp0` de la TSS perquè apunti a la base de la pila de sistema. 2. Cal canviar l'MSR per mantenir al coherència amb `sysenter`. 3. Canviem el registre `cr3` perquè apunti a la base del directori. 4. Guardem `ebp` al PCB. Cal tenir en compte que `ebp` ens apunta a la pila del nostre sistema del procés actual (és l'enllaç dinàmic)! A més cal tenir en compte que en aquest moment `esp` = `ebp` 5. Obtenim l'antic `ebp` del PCB del procés que volem restaurar i el carreguem al registre `esp`. D'aquesta manera tindrem el registre `esp` apuntant just al camp `ebp` de la pila. És a dir, a l'anic enllaç dinàmic que s'havia guardat. De manera que al fer `pop ebp` i `ret` haurem tornat al `task_switch` tal i com s'havia deixat abans. Canvi fet.

```
void inner_task_switch(union task_union*t){   //Punter a la task union del nou procés
    tss.esp0 =  KERNEL_ESP(t);   //#define KERNEL_ESP(t)  (DWord)  &(t)->stack[KERNEL_STACK_
    writeMsr(0x175, (int) KERNEL_ESP(t));

    if(current() -> dir_pages_baseAddr == t-> task.dir_pages_baseAddr)
        set_cr3(t -> task.dir_pages_baseAddr);

    current() -> kernel_esp = (unsigned long *) getEbp();

    setEsp(t -> task.kernel_esp);

    return;
}
```

4.2.3 Creació de processos És pot fer per mitja de duplicar el procés (copiar codi i dades) per mitjà de la crida `fork` o duplicar el fluxe (compartint l'espai de direccions amb el pare, threads en OpenMP) per mitjà de la crida `clone`. Aquí es tractarà només el `fork`. La creació de threads es tracta en un apartat posterior. `int fork()`: Crida a sistema que ens permet generat un procés. La seva implementació es descriu a continuació: 0. Estat inicial: Tenim un procés amb codi i dades d'usuari. 1. Obtenir PCB lliure: A la cua de `free`. 2. Inicialitzar PCB: Bàsicament copiar el del pare al fill. 3. Inicialitzar l'espai d'adreces: Carregar un executable (no fet a zeos) o heredar del pare dades i codi:

`3.1 Cerquem pàgines físiques lliures.`

3.2 Mapejem al nou procés el codi de sistema i el d'usuari. Aquest serà compartit.

3.4 Mapegem les adreces físiques que hem obtingut al punt 3.1 a l'espai d'adreces lògic del

3.5 Ampliem l'espai d'adreces del pare amb aquestes noves pàgines físiques del fill i copiem

3.6 Desmapejem aquestes adreces de l'espai d'adreces del pare i fem _flush_ de la TLB. (Toca

4. Actualitzar el `task_union` del fill amb els nous valors pel PCB assignant un nou PCB.
5. Peparem la pila del fill per al task_switch (com si se n'hagués fet un perquè així sigui restaurable). L'adressa de retorn del fill serà una funció anomenada `return_from_fork` que farà que a la que el procés nou agafi el control, la crida a `fork()` feta retornarà 0. 6 Insertar el procés nou a la llista READY 7 Retornar el pid del nou procés creat.

```
int sys_fork()
{
  int PID=-1;

  // a) creates the child process
    if(list_empty( &freequeue )) return -EAGAIN;
    struct list_head * free_head = list_first( &freequeue );
    list_del(free_head);
    struct task_struct * child_task = list_head_to_task_struct(free_head);

    // b) copiem pcb del pare al fill
    copy_data(current(), child_task, sizeof(union task_union));

    // c) inicialitzem el directori
    allocate_DIR(child_task);

    // d) cerquem pagines fisiques per mapejar amb les logiques
    int brk = (int) current()->brk;
    int NUM_PAG_DATA_HEAP = NUM_PAG_DATA + (brk & 0x0fff)? PAG_HEAP(brk) : PAG_HEAP(brk) - 1
    int frames[NUM_PAG_DATA_HEAP];   //cal afegir les pags del heap
    int i;
    for (i=0; i<NUM_PAG_DATA_HEAP; i++)
    {
        frames[i] = alloc_frame();
        if(frames[i] < 0)
        {
            for(int j = 0; j < i; j++) free_frame(frames[j]); // alliberem els frames reser
            list_add(&child_task->list, &freequeue);                // tornem a encuar la
            return -ENOMEM;
        }
```

```
}
// e) heredem les dades del pare
page_table_entry *parent_PT = get_PT(current());
page_table_entry *child_PT = get_PT(child_task);

//  > i) creacio de l'espai d'adresses del process fill:


//  >> A) compartim codi de sistema i usuari
for(i=1; i < NUM_PAG_KERNEL; i++)  // suposem que el codi de sistema esta mapejat a les
{
    child_PT[1+i].entry = parent_PT[1+i].entry;
}

for(i=0; i < NUM_PAG_CODE; i++)
{
    set_ss_pag(child_PT, i+PAG_LOG_INIT_CODE, get_frame(parent_PT, i+PAG_LOG_INIT_CODE))

}

//  >> B) assignem les noves pagines fisiques a les adresses logiques del proces fill
for(i=0; i < NUM_PAG_DATA_HEAP; i++)
{
    set_ss_pag(child_PT, i+PAG_LOG_INIT_DATA, frames[i]);

}

// > ii) ampliem lespai d'adresses del pare per poder accedir a les pagines del fill pe
for(i=0; i < NUM_PAG_DATA_HEAP; i++)
{
    unsigned int page = i+PAG_LOG_INIT_DATA;
    set_ss_pag(parent_PT, page+NUM_PAG_DATA_HEAP, frames[i]);
    copy_data( (unsigned long *)(page*PAGE_SIZE), (unsigned long *)((page+NUM_PAG_DATA_H
    del_ss_pag(parent_PT, page+NUM_PAG_DATA_HEAP);


}
set_cr3(get_DIR(current())); // flush de la TLB


// f) assignem nou PID
PID = nextPID++;
child_task->PID = PID;

// g) modifiquem els camps que han de canviar en el proces fill
init_stat(child_task);
```

```
    // h) preparem la pila del fill per al task_switch
    union task_union * child_union = (union task_union *) child_task;

    ((unsigned long *)KERNEL_ESP(child_union))[-0x13] = (unsigned long) 0;
    ((unsigned long *)KERNEL_ESP(child_union))[-0x12] = (unsigned long) ret_from_fork; // 5
    child_task->kernel_esp =    &((unsigned long *)KERNEL_ESP(child_union))[-0x13];

    // i) empilem a la readyqueue el fill
    if (child_task->nice)list_add_tail(&child_task->list, &readyqueue);
    else list_add_tail(&child_task->list, &priorityqueue);

    // j) retornem el PID del fill


    return PID;
}
```

//recordar parlar d'init i idle quan parli del planificador.