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.

torna a l'index

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 context 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.

torna a l'index

---

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'objecte 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 tamany del bus.

Espai de direccions lògic del procés: Espai d'un procés en execució. Són les adreces (logiques) que pot accedir el proces. Relatives a una direccio 0, igual per a tots els procesos.

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`.

torna a l'index

---

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:

```
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 prioritat
     void *brk; //per la gestio de la mem dinamica
};
```

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!!. D'aquesta manera sabem que el top de la pila és el principi del PCB.

```
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`**

4.1.1 Llistes Per organitzar les tasques, 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 estar a cap llista.

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 compatibilitat amb `sysenter`. 3. Canviem el registre `cr3` perquè apunti a la base del directori; canviem el espai d'adreces del proces.
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 un tasca lliure: A la cua de `free`.
2. Inicialitzar PCB: Bàsicament copiar el del pare al fill.
3. Inicialitzar l'espai d'adreces: 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 procés nou.

   3.5 Ampliem l'espai d'adreces del pare amb aquestes noves pàgines físiques del fill i copiem totes les dades juntament amb la pila del pare a aquestes noves pàgines.

   3.6 Desmapejem aquestes adreces de l'espai d'adreces del pare i fem *flush* de la TLB. (Tocant el valor del registre `cr3`)
4. Actualitzar el `task_union` del fill amb els nous valors pel PCB assignant un nou PID.
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_ESint 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;
}P(child_union))[-0x12] = (unsigned long) ret_from_fork; // 5 registres + SAVE_ALL (11 regs)
    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;
}
```

4.2.4 Destrucció de processos Es fa per mitjà de la crida `int exit()`. Ha de: 1. Alliberar recursos assignats. - PCB - Espai de direccions. - Altres coses

que hagi pogut demanar com semàfors o memòria dinàmica. 2. Posar la tasca a la llista de tasques lliures 3. Executar al planificador perquè esculli un nou procés.

Nosaltres a zeos fem el `exit` sense paràmetres ni sicronització però en un linux normal, el procés pare es pot sincronitzar amb els fills per mitjà de la crida a `waitpid`. Això fa que el fill ha de mantenir informació sobre la seva mort en el PCB i doncs aquest no s'allibera fins que no es consulta aquesta informació (estat zombie). Si el pare mor sense fer el `waitpid` és el procés init (initial) qui *adopta* els fills i fa el `waitpid`.

```
void sys_exit()
{
    struct task_struct *task = current();
    page_table_entry *task_PT = get_PT(task);
    int brk = (int) current()->brk;
    int NUM_PAG_DATA_HEAP = NUM_PAG_DATA + (brk & 0x0fff)? PAG_HEAP(brk) : PAG_HEAP(brk) - 1
    for(int i =0; i < NUM_PAG_DATA_HEAP ; ++i)
    {
        free_frame(get_frame(task_PT,i+PAG_LOG_INIT_DATA));
        del_ss_pag(task_PT, i+PAG_LOG_INIT_DATA);
    }

    task->PID=-1;
    update_process_state_rr(task, &freequeue);
    sched_next_rr();

}
```

4.2.5 Planificador de processos **Procés idle:** Procés del sistema que ocupa la CPU quan no hi ha ningú per fer-ho. És un procés que corre en mode sistema i no forma part de cap cua.

Procés init: Procés incial del sistema. Tots els processos són fills d'aquest.

> **Cal un sistema que gestioni els canvis de procés perquè tots s'executin** -> Planificador de processos.

Planificador de processos: S'encarrega de decidir quan els processos passen de `ready` a `run` i vicersesa. En un sistema linux normal n'hi ha 3:

- A llarg termini (batch): S'executa quan comença o acaba un procés.
- A mitjà termini: Encarregat del swap. S'executa quan hi ha escasetat de memòria.
- A curt termini: Selecciona el següent procés a executar i s'executa cada X temps decidit per la política.

Ràfega de CPU: Temps consecutiu que un procés està en execucio.

Ràfega d'E/S: Temps consecutiu que un procés està en una operació d'E/S.

Aquestes fases es poden representar en un diagrama de Gantt

Estat de bloqueig: Que no esta en execucio, normalment vol dir que esta a una cua de *blocked* fins que algu el mou a `ready` (***apropiació diferida***) o a `run` (***apropiació directa***).

Algoritme del planificador:

```
void schedule (void)
{
    update_sched_data_rr();   //ACTUALITZA DADES DEL SCHEDULER (QUANTUM QUE QUEDA)
    if (needs_sched_rr())     //SI LI TOCA CANVIAR
    {
        struct task_struct *curr = current();
        update_process_state_rr(curr, &priorityqueue);   //CANVIA L'ESTAT DEL PROCÉS
        sched_next_rr();                 //EXECUTA AL QUE LI TOQUI
    }
}
```

4.2.5.1 Polítiques de planificació de processos

No apropiatives: El sistema usa la CPU fins que es bloqueja

Apropiatives: EL planificadort pot expulsar un procés de la CPU.

- Diferides: Al cap d'un temps es fa la planificació.
- Immediates: Es fa sempre que hi ha un canvi.

Per a dur a terme aquestes polítiques, s'usen cues. Cada cua pot tenir una política diferent.

4.2.5.2 Algoritmes de planificació

FCFS: First come, first served. No apropiatiu. Algoritme de *tonto el último*.

Prioritats: cada procés té assignada una prioritat. Entre processos de igual prioritat és fifo. Pot provocar inanició.

Round Robin: A *pito-pito*. Es canvia cada cop que a un procés se li acaba el quantum. Es poden aplicar priotitats. Cal pensar que podem tenir moltes cues amb algoritmes diferents i polítiques diferents.

Els algoritmes ens han de garantir:

- Justícia: Tots els processos han de mamar de la CPU
- Eficiència: La CPU ha d'anar A Puto Gas, no la podem tenir fent el vago.
- Productivitat: Cal maximitzar el nombre de treball per unitat de temps.
- Temps d'espera: Cal minimtzar-lo.
- Temps de resposta: cal minimitzar el temps que triga un procés en obtenir el seu primer resultat.
- Temps de retorn: minimitzar el temps que triga en executar-se *un procés*.

Les diferents prioritats s'implementen tenint diferents cues de `ready`.

4.2.6 Fluxes (threads) Múltiples processos executant el mateix codi amb dades compartides -> Generem paral · lelisme

Concurrencia: Quan un processador és compartit en el temps per varis fluxes. Es genera paral · lelisme quan hi ha fluxes executant-se en paral · lel.

Els processos mai compartiran la pila.

Processos Multifluxe: Permetre diferents seqüències d'execució simultànies d'un mateix procés.

Fluxe: Cada una d'aquestes seqüències. És la unitat mínima de treball de la CPU. Cada fluxe d'un procés comparteix tots els recursos i totes les característiques. Cada fluxe està associat a (és a dir, cada thread té un/a únic/a) **una pila, un program counter i un banc de registres**. La resta de recursos és compratit entre els fluxes.

Què guanyem usant fluxes en ves de processos?

- Perdem overhead de gestió: creació destrucció i canvi de context ens els evitem.
- Aprofitem recursos
- Com que compartim dades la comunicació entre threads és més senzilla -> OJU! problemes de condició de carrera que resoldrem més tard.
- Ens permet explotar el paral · lelisme i la concurrència.
- Millorem la modularitat de les aplicacions perquè podem encapsular les feines.
- Podem crear fluxes destinats a E/S per fer-ho de forma asíncrona.
- Podem atendre a varies peticions en un servidor.

Com és el cas específic de Linux?

Linux usa la crida a sistema `int clone()` per generar threads.

- Linux no fa distinció entre threads i processos a l'hora de planificar. Tot són tasques que poden compartir o no els recursos amb altres tasques. En un thread, el `task_struct` conté punters enlloc de dades.

`int clone ( int (*fn) (void *), void *child_stack, int flags, void *arg ):` Retorna el PID del *procés* creat (recordar que hem dit que el planificador no distingeix entre processos i threads). Rep com a paràmetre el punter a una funció, una zona de memòria per usar com a pila, les flags i el argument de la funció.

POSIX ens proporciona una interfície per la gestió dels threads -> la llibreria `pthreads`. Permet:

- Creació de threads amb `pthread_create`.
- Identificació del thread amb `pthread_self()`.

- Finalització del thread amb `pthread_exit`.
- Sicronització de final de flux amb `pthread_join`.

4.2.7 Sincronització entre processos. Cal evitar condicions de carrera entre recursos compartits entre els processos concurrents -> Gestió de regions crítiques.

Solució: Exclusió mutua en les regions crítiques on poden haver condicions de carrera.

S'ha de garantir unes condicions per un correcte accés a una regió crítica: - Només hi pot haver un fluxe a una regió crítica. - Un fluxe no pot esperar indefinidament per entrar a una regió crítica. - Un fluxe que s'està executant fora d'una regió crítica no pot evitar que d'altres hi entrin. - No es pot fer cap hipòtesi sobre el nombre de processadors ni la seva velocitat d'execució.

Exclusió mutua: Només permetre un fluxe en una regió. Es garanteix un accés seqüencial. Un procés mantindrà la regió encara que hi hagi un canvi de context. - Caldrà que el programador identifiqui i marqui les regions crítiques degudament -> El sistema ens ofereix crides a sistema per marcar aquestes regions i **l'arquitectura ens facilita operacions atòmiques** és a dir, que només executarà un thread alhora: un exemple és el `test_and_set` de PAR. Aquestes operacions venen definides pel processador.

Com poden implementar-se aquestes crides a sistema?

Espera activa o busy waiting: Per mitja d'una instrucció atòmica de l'arquitectura com un `test_and_set`. Anem consultant tota l'estona el valor de la variable. - Grans inconvenients: Ocupem la CPU amb càlcul tremendo inútil amb tot el que això comporta i saturem el bus de memòria anant sempre a buscar la mateixa instrucció(recordar PAR). - Grans solucions: Bloqueix del procés

Espera de bloqueig: Permet reduir el gast de la CPU i els accessos a memòria. Bloquejarem els processos que no puguin entrar a la regió crítica per mitjà de semàfors.

Semàfor: Estructura de dades que ens permetrà controlar els accessos a una regió crítica per mitjà de crides a sistema implementades amb les operacions atòmiques de l'arquitectura (consultes i modificacions del contador del semàfor). **Cada semàfor té associat un contador i una cua de processos bloquejats.** Aquest contador ens indica el nombre de processos que poden accedir simultaniament al recurs. n=1 permet l'exclusió mutua. Les seguents crides a sistema ens permeten interactuar amb ells: - **`sem_init(sem,n):`** Crea un semàfor. *sem* és una estructura de dades de la que no hem parlat. `C sem -> count = n; ini_queue(sem->queue);` - **`sem_wait(sem):`** Demanar acces a una regió critica protegida per el semàfor `sem` (lock). En cas de que no s'hi pugui accedir, es **bloqueja el procés**. És a dir, l'inclou a la cua de `blocked` del semàfor. `C sem -> count--; if(sem->count<0) bloquejar(sem->queue);` - **`sem_signal(sem):`** Sortida de la zona d'exclusió (unlock). Si hi ha processos

esperant (`sem->count <= 0`) cal **despertar un procés**. És a dir, agafar un procés de la cua de `blocked` i cardar-lo a `ready` o a `run`, depenent de la política del semàfor. `C sem -> count++; if(sem->count <= 0) //vol dir que hi ha processos esperant despertar(sem->queue);` > **Quan s'ha de bloquejar un procés normalment es fa primer espera activa i després semàfors perquè aquests últims tenen un overhead més alt.**

Com podem usar semàfors?

`sem_init(sem,1):` Anomenat *Mutex*. Només entra un procés.

`sem_init(sem,0):` Sincronització dels processos. Per controlar els threads. Els atura a tots.

`sem_init(sem,N):` Restricció de recursos. Com a la cua del super. Hi ha només 3 caixes i molta gent.

Deadlocks: Abraçades mortals. Poden donar-se si hi ha processos bloquejats esperant un event d'un altre procés que també està bloquejat esperant un event d'un altre i així en cicle. A necessita B que necessita C que necessita A i la hem liat. Tot bloquejat. Com ho solucionem?

Hi ha 4 condicions que s'han de complir perque hi hagi un *deadlock*. **Hem d'evitar com a mínim una d'elles.** - `Hi ha d'haver exclusió mútua: Mínim de 2 recursos no compartibles`: És dificil de solucionar. Només podem fer que vetllar perque els recursos siguin compartibles. - `Un fluxe aconsegueix un recurs i espera per un altre`: Evitar que això es pugui fer. - `No preempció, és a dir, que no hi hagi prioritat en els recursos i que quan un fluxe pilli un recurs no el deixi anar`: Permetre treure recursos a processos. - `Hi ha d'haver un cicle de dos o més processos on cadascun necessita un recurs bloquejat per un d'altre`: Ordenar les peticions i fer que hagin d'aconseguir els recursos en el mateix ordre.

torna a l'index

---

5. Gestió de l'entrada i sortida

Entrada i sortida: Transferencia de dades desde o fins a un procés. -> Entre processos o procés - dispositiu.

L'SO ha de gestionar l'accés a dispositius Aquest accés és molt variat i depèn del dispositiu. - Cal garantir una capa d'uniformitat i fer que l'accés sigui igual per a tots els dispositius. El codi d'usuari ha de ser independent al típus de dispositiu accedit. - Cal garantir que només l'SO hi tindrà accés: instruccions privilegiades. - Cal optimitzar el rendiment general del sistema. - Cal permetre que apareixin nous dispositius.

Linux usarà 3 típus de dispositius que actuaran a mode de capes per permetre aquesta separació entre l'usuari i el dispositiu: - **Dispositiu físic:** No són visibles per l'usuari i només són accessibles pel sistema. Són els *Device Drivers*. Codi de baix nivell depenent del dispositiu que implementa la interfície definida pel SO (implementa les crides `open`, `close`, `read`, `write`... se'n parla més endavant). - **Dispositius lògics:** Són un fitxer. Una abstracció lògica creada pel sistema. Poden tenir diferents dispositius associats. Per exemple la tty té teclat i pantalla associats. Són visibles des de l'usuari. - **Dispositus virtuals:** L'usuari tracta sempre amb el dispositiu virtual. Cada procés te els seus. Són els descriptors de la taula de canals.

S'usen diferents estructures de dades per la seva gestió: - **Taula de canals:** Única per procés, relaciona un numero de la taula de canals amb el dispositiu lògic associat. - **Taula de fitxers oberts:** Taula que conté aquells fitxers que tenen dispositius virtuals. Ens indica per cada fixter quants dispositius virtuals té associats, amb quins permissos i enllaça amb la taula d'inodes. - **Taula d'inodes:** Indica per cada inode al que s'ha accedit quantes referències té a la taula de canals.

> **S'usen unes crides a sistema genèriques que s'implementen en els device drivers.** - **`int open(char *fitxer, int mode, int permissos):`** Crea un dispositiu virtual pel dispositiu passat per paràmetre (retorna el nombre del canal). Ens crea entrades a la taula de canals i la de fitxers oberts. - **`int close(int canal):`** Tanca el disp. virtual - **`int read(int canal, char *buff, int nbytes):`** Espera llegir *nbytes* del canal amb fd *canal*. - **`int write(int canal, char *buff, int nbytes):`** Anàlog al read. - **`dup i dup2:`** Modifiquen les entrades de la taula de canals. Permeten duplicar i modificar les referències. - **`ioctl i fcntl:`** Són custom. El dispositu les pot implementar per afegir funcionalitats. Per exemple: SI tinc una estació meteorològica i no ho vull llegir tot, només l'anemòmetre -> podria implementar una d'aquestes.

Dispositu: Objecte amb el que el procés es vol comunicar. Es controla amb dos típus d'estructures de dades:

- **Característiques dinàmiques (dictats a la taula de fitxers oberts):** Són diferents a cada accés, com ara el mode d'acces, la posició etc...
- **Característiques estàtiques (presents a l'inode):** Es guarden en una estructura anomenada **descriptor de dispositiu (DD)**. Conté les caràcterístiques de del dispositiu com el nom, el propietari, el mode, etc i els les adreces a les funcions de llegir, escriure, obrir, etc.

Això dona suport a la concurrència perquè dos dispositius vituals poden accedir a les mateixes característiques dinàmiques o, poden haver-hi diferents característiques dinàmiques pel mateux dispositiu i així permetre que dos dispositius virtuals accedeixin de forma concurrent al dispositiu amb característiques dinàmiques diferents (Veure transparències 19 - 29 del tema per il · lustrar

l'explicació).

5.1 Mecanismes d'accés a un dispositiu.

Descriptor de Dispositiu: Conté les característiques estàtiques i els punters a les funcions perquè es pugin usar. Defineix l'interfície d'acces (que és general per tots els dispositius).

Device Driver: Implementa les funcions específiques del dispositiu. Controla el hw del dispositiu.

A través d'aquest descriptor de dispositiu, el dispositiu lògic pot comunicar-se amb el driver que implementa les funcions que l'usuari demanarà al dispositiu lògic.

La comunicació entre el driver i el dispositiui es pot fer de dues maneres: - **Per enquesta (polling):** Preguntar cada interval de temps. És molt poc eficient. - **Per interrupció:** El procés reb una interrupció quna el dispositiu acaba. El procés pot bloquejar-se fins a rebre la interrupció.

I pot ser: - **Síncrona:** No es continua l'execució del procés d'usuari fins que no finalitzi l'operació d'E/S. - **Asíncrona:** Es continua executant codi del procés mentre es resol l'E/S.

Com s'implementen?

Per mitjà de **gestors**: És un procés del sistema encarregat d'atendre i resoldre peticions d'E/S. Simplifica l'accés a les estructures de dades. **Pot haver-hi un o més gestors per dispositiu.** Pseudocodi:

```
for (;;){
    esperar petició
    recollir paràmetres
    fer e/s
    entregar resultats
    notificar final d'E/S
}
```

Com es sincronitza el procés amb el gestor? - A través de semàfors (veure a sota a l'explicació completa de l'algoritme). - **IORB:** Input Output Request Block. Estructura de dades que permet el pas de paràmetres. El seu contingut depèn del dispositiu. Cada gestor/dispositiu té una cua d'IORBs amb les peticions pendents. Les rutines E/S omplen els IORBs. Conté: - Buffer d'usuari on cal deixar les dades (o on es troben). - longitut. - id d'operació - típus d'operació - **io_fin:** Estructura que ens permet el retorn de paràmetres. Conté l'indentificador de l'operació d'ES i el seu resultat. Hi ha una cua d'io_fin per dispositiu.

Overview general i exemples del que s'ha anat dient:

Operació d'E/S Síncrona:

1. Es fa un read. El trap arriba a la taula de syscalls i es crida a la rutina del read.
2. el read llegeix la taula de canals i veu el disp. lògic associat a aquest canal. Crida al read que hi ha al device descriptor de l'inode d'aquest fitxer.
3. Aquest read apuntat pel DD realitza l'operació pertinent i retorna.

Operació d'E/S Sícrona amb gestor: 1. Es fa un read. El trap arriba a la taula de syscalls i es crida a la rutina del read. 2. el read llegeix la taula de canals i veu el disp. lògic associat a aquest canal. Crida al read que hi ha al device descriptor de l'inode d'aquest fitxer. 3. Aquest read de dispositiu crea un IORB i l'encua a la cua d'iorb del gestor. 4. Aquest read de dispositiu fa un `sem_signal(sem)` per alliberar el gestor del `sem_wait(sem)`. A continuacio, el read fa un `sem_wait(io_id)`. 5. El gestor que ha despertat, selecciona un IORB de la cua i efectua l'operació pertinent amb el dispositiu. 6. El gestor efectua un `sem_signal(io_id)` perquè el read del disporitiu es desbloquegi. 7. El read recull el io_fin de la cua d'io_fin del gestor i retorna.

Operació d'ES asíncrona amb gestor 1. Es fa un read. El trap arriba a la taula de syscalls i es crida a la rutina del read. 2. el read llegeix la taula de canals i veu el disp. lògic associat a aquest canal. Crida al read que hi ha al device descriptor de l'inode d'aquest fitxer. 3. Aquest read de dispositiu crea un IORB i l'encua a la cua d'iorb del gestor. 4. Aquest read de dispositiu fa un `sem_signal(sem)` per despertar al gestor si estava bloquejat i retorna el io_id. Hi haurà un procés fora del read de dispositiu (és a dir, un codi que no depèn del dispositiu), que farà el `sem_wait(id_io)` per recollir el resultat. 5. El gestor que ha despertat (o ja estava despert, qui sap), selecciona un IORB de la cua i efectua l'operació pertinent amb el dispositiu. Crea el `io_fin` i l'encua a la cua d'io_fins 6. El gestor efectua un `sem_signal(io_id)`. 7. El codi que hagi fet el `sem_wait(io_id)` és desbloqueja i recull el resultat de la cua d'io_fins i retorna el resultat.

Possibles optimitzacions: - Tenir buffers on volcar les dades a escriure/llegir perquè puguin anar fent mentre els processos treballen. - Tenir dispositius intermitjos que gestionin els accessos com una cua d'impressió. Només s'imprimeix d'un en un però jo puc enviar mil coses a l'impressora i no se'm bloquejen els processos. - Usar millors algoritmés d'accés al dispositiu - Usar RAIDs i altres millors organitzacions del HW.

5.2 Comunicació entre processos Hi ha diferents maneres de comunicar processos: - Memòria compartida: Entre fluxes. - Pas de missatges: Sockets i pipes. - Signals.

5.2.1 Sockets **Socket:** Dispositiu lògic de comunicació **bidireccional** que permet comunicar processos en la mateixa màquina o en màquines remotes. És caracteritzat per: - Tipus de connexió: Orientat o no orientat a connexió - Espai de noms: Els processos han de tenir un nom. Es designa amb una IP i un PORT. - Protocol de comunicació: TCP, UDP, ICMP

Sockets orientats a connexió (mantenen una "sessió"):

Servidor: 1. `socket()` Crea i associa el socket a un canal. 2. `bind()` Associa ip i port al socket i el publica. 3. `listen()` Configura el nombre de conexions que es poden rebre. 4. `accept()`: Espera solicitud. Al rebre-la, duplica el canal i torna a obrir el port per rebre més peticions 5. S'efecuten reads i writes a través del canal de comunicació.

Client: 1. `socket()` Crea i associa el socket a un canal. 2. `connect()` Solicita connexió a un servidor. 3. S'efecuten reads i writes a través del canal de comunicació.

Sockets no orientats a connexió

Servidor i client: 1. `socket()` Crea i associa el socket a un canal. 2. `bind()` Associa ip i port al socket i el publica. 3. S'efectuen crides a `sendto()` especificant destí i `recvfrom()` per rebre del destí.

sockaddr: Estructura de dades que conté tota la informació del socket.

Endianisme Problema de la transmissió. Cal per conveni sempre transmetre en Big Endian. Només passa amb nombres, no amb els caràcters!! Tindrem crides a sistema que ens permetran fer aquest canvi d'informació (`htons`, `htonl`...)

Linux té un sistema de capes per gestionar els sockets:

- Capa socket: Lídia amb l'interficie d'usuari.
- Capa protocol: Implementa els nivells de sessió, transport, xarxa i enllaç.
- Capa interfície. A nivell físic (codificació elèctrica del missatge).

La comunicacnió entre elles es fa per mitja de l'estructura `sk_buff` que codifica el missatge i te els camps necessaris poder transmetre la info per totes les capes.

Enviar: Al fer-se un `write` es crea un `sk_buff` a la capa de socket. S'envia un trap a la capa de protocol que codificarà el missatge segons el que dictin les capes de sessió, transport, xarxa i enllaç i la info del socket i l'enviarà a la tarja de xarxa que el codificara en forma d'impulsos elèctrics per que pugui ser enviar.

Rebre: Al rebre els impulsos elèctrics, la capa d'interfície crea un `sk_buff` i emet una interrupció hw per la capa de protocol. Aquesta recull l'`sk_buff`, el processa i el transmet a través d'un trap a la capa de socket que s'encarrega de transmetre el missatge a l'usuari.

Per cada connexió es necessita un `sk_buff`. Per tant, el que passa en un atac DoS de *syn floading* era que s'enviaven molts *syn* i mai cap *ack* de manera que la memòria es petava a base de `sk_buff`. Ara això ja no passa perquè el `listen` elimina l'`sk_buff` al rebre'l.

5.2.2 Pipes Intercanvi de dades entre processos a la mateixa màquina. Són una regió de memòria amb un inode associat. Aquest inode conté els buffers de memòria i les operacions sobre els buffers, les operacions d'accés a la pipe i el semàfor que implementa els bloquejos. De dos típus: - Sense nom: No tenen nom que les representi i per tant són només una regió de memòria. - Amb nom:

Necessitaran un dispositiu perquè hi puguem fer referència a aquesta regió de memòria

Són unidireccionals i a mesura que es llegeix es borra el contingut escrit. Comportament FIFO.

> **LA SEGUENT PART D'AQUEST TEMA QUE ÉS SISTEMA DE FITXERS NO HA ENTRAT A EXÀMEN AQUEST ANY I NO S'HA DONAT A CLASSE, NO ESTÀ PRESENT ALS APUNTS**

torna a l'index

---

6. Gestió de la memòria.

D'aquest tema només s'ha tractat memòria dinàmica en aquest curs.

Memòria dinàmica: Mecanisme que permet demanar i alliberar memòria a demanda.

6.1 Memòria dinàmica del sistema

Buddy System: Gestió de la memòria a base de reservar blocs en potències de dos. A base d'aquestes potències, ens permet gestionar un bloc a través d'un arbre binari. Si divideixo un bloc obtinc dos fills en l'arbre. Les fulles son els blocs de memòria actuals. Permet dues operacions: - **Splitting:** Separar un bloc en dos (generar dos fulles). - **Coalescing:** Juntar dos blocs consecutius. Han de ser germans en l'arbre (juntar dos fulles i per tant eliminar-les en pos d'un bloc més gran). D'aquesta manera, s'organitzen les regions de memòria en forma d'arbre (il · lustració a les transpas 14-25).

Té grans inconvenients: - Fragmentació externa: Perquè queden blocs lliures petits que no es poden aprofitar. - Fragmentació interna: Perquè només puc reservar en potències de 2.

Slab allocator: Intenta resoldre els problemes del buddy system. Consisteix fer una reserva anticipada d'objectes del sistema (anomenada *slab*). Aquests *slabs* es guarden en cachés. Cada caché conté objectes del mateix tamany i la informació de si està en us o no. D'aquesta manera el sistema no demana regions individuals de PCBs o regions individuals de smàfors sinó que els demana d'slab en slab. Així sempre demana molta més informació, els blocs són més grans i reduïm el risc de segmentació.

6.2. Memòria dinàmica per l'usuari.

Tindrem una zona especial de memòria dedicada: el HEAP.

`int *sbrk(int incr);` Crida a sistema dedicada. Si l'increment és positiu dona memòria, si és negatiu la treu. És poc pràctica perque no ens permet alliberar

zones concretes de memòria.

Com que el sbrk és molt complicat d'usar, hi ha moltes llibreries que han implementat els seus *allocatadors* per fer aquest procés més senzill. Nosaltres estudiarem el `malloc()` i en concret el *Doug Lea Malloc*

6.2.1 Doug Lea Malloc Basats en chunks alineats a 8 bytes amb una capçalera i una zona de memòria usable per l'usuari.

Cada cop que s'allibera un chunk s'afegeix a una llista doblement encadenada que conté els chunks organitzats per taamanys. Aquesta llista, a cada posició conté una llista de chunks d'un tamany concret doblement encadenats. Hi ha una una llistya per cada mida diferent dins als 512 bytes. A partir d'aquí ja s'ordenen de manera diferent.

Capçaleres: Anomenats **bounary tags**. Es troben al principi i al final. Indiquen el tamany de la regió i si aquesta està plena o buida.

Camp de dades: Camp del tamany demanat per l'usuari. EL punter retornat per malloc apunta a l'inici d'aquesta regió. En el cas de que estigui lliure, aquest camp contindrà un punter al chunk anterior i un al posterior de la llista de chuncks lliures.

Aquesta gestió permet fer *splitting* al reservar i *coalescing* de blocs consecutius a l'alliberar.

torna a l'index