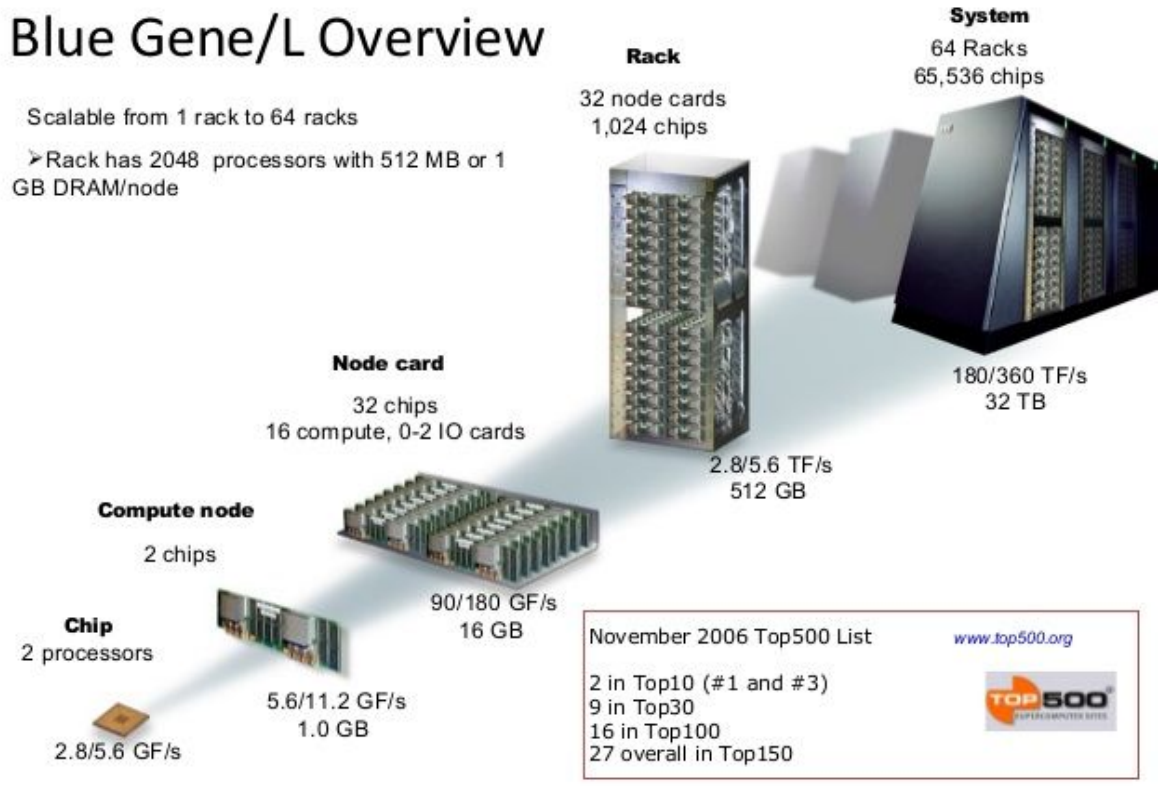
CONCEPTES AVANÇATS DE SISTEMES OPERATIUS (CASO)
Facultat d'Informàtica de Barcelona,
Dept. d'Arquitectura de Computadors,
curs 2019/2020 – 2Q

3r Control Parcial **19 de maig de 2020**

- L'examen és individual
- Totes les respostes han de ser raonades.
- Lliureu al Racó un document pdf, docx, odt o txt amb les respostes.
- L'examen és amb llibres i apunts.
- Temps d'examen 1h i 40 minuts

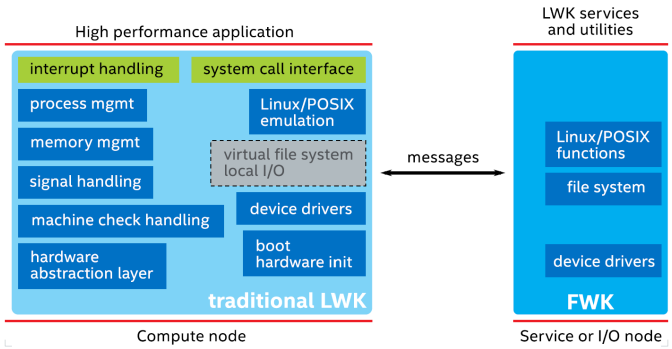
1. **Què és un Sistema Operatiu?** Per a la familia de supercomputadors BlueGene d'IBM es va implementar un lightweight kernel anomenat oficialment CNK (i Blrts extraoficialment). Suportava una gran quantitat de crides al sistema de Linux, tot i que només es dedicava a computació (Compute node a la figura); totes les crides d'entrada/sortida eren redireccionades cap a un únic node Linux (al Node Card de la figura).



El CNK no permetia operacions fork/exec, en canvi, un compute node estava orientat a executar un procés MPI multithread, amb rigorosa afinitat. Tenint present la definició de Sistema Operatiu que has estudiat:

(a) Creus que el CNK és un sistema operatiu?

El CNK (Compute Node Kernel) és un sistema operatiu (protegeix al sistema i el gestiona per a un ús òptim) si fitem el hardware de la màquina al node sobre el que s'executa. De la resta del supercomputador, el CNK no sap res i per tant no pot fer d'interfície entre el programador i el supercomputador. El CNK és un sistema operatiu per al *compute node*, però necessita del FWK (Full Weight Kernel) per a comunicar-se amb l'exterior. En aquest sentit, CNK és un co-kernel.



(b) En quina de les funcionalitats atribuïdes a un SO, el CNK té una mancança important?

Tot i això, encara que només ens fixem en el *compute node*, el CNK té mancances com a sistema operatiu. En no poder fer fork/exec, no pot tenir un shell, no pot tenir un intèrpret de comandes, ni tampoc pot suportar llenguatges d'scripting. Per tant, la funcionalitat de proveir una interfície a l'usuari no la pot satisfer. Aquesta funcionalitat la dona el FWK, en el cas de BG/L, un node Linux.

2. Mach tasks vs Unix processes

A les figures estan representats els models de memòria d'una task de Mach i d'un procés Unix. Aquesta representació data de mitjans dels anys 80.

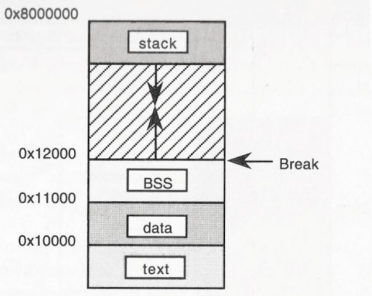


Figure 4.7: Memory Model of a UNIX Program

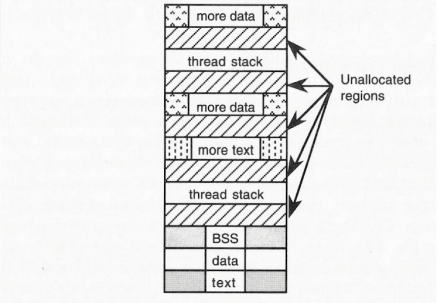


Figure 4.8: Memory Model of a Mach Program

(a) Quina era (és) la syscall (o les syscalls) per crear aquest dibuix a Mach? I a Unix?

A Mach cal crear una task i dos threads per tenir el dibuix de la figura. Amb aquestes crides:

```
kern_return_t   task_create
                (task_t                                 parent_task,
                 ledger_port_array_t                        ledgers,
                 int                                   ledger_count,
                 boolean_t                            inherit_memory,
                 task_t                                   child_task);
kern_return_t   thread_create
                (task_t                                 parent_task,
                 thread_act_t                          child_thread);
```

A Unix, cal crear un procés per a tenir el dibuix de la figura. Amb aquesta crida:

```
pid_t fork(void);
```

Centrant-nos en el model de memòria, respón a les següents preguntes:

(b) Quina és la diferència més significativa entre els dos dibuixos? Quina era (és) la syscall per gestionar la zona ratllada a Unix (Linux)? I a Mach?

El que més sobta és l'organització de la memòria dinàmica. Mentre que en el cas de Unix, la zona ratllada és continua, en el cas de Mach, la zona ratllada està trocejada.

A Unix, les syscall per gestionar la zona ratllada eren `int brk(void *addr);` i `void *sbrk(intptr_t increment);` que movien el final del segment de dades (*program break*) del procés.

A Mach, des del seu inici, un programa podia assignar, desassignar, protegir i compartir zones de memòria arbitràries entre **tasks**. Aquestes zones no han de ser contínues, sinó que poden estar disperses per tot l'espai lògic del programa. A laboratori hem treballat la crida

```
kern_return_t   vm_allocate
                (vm_task_t                            target_task,
                 vm_address_t                             address,
                 vm_size_t                                   size,
                 boolean_t                                anywhere);
```

(c) A POSIX (i per tant, s'implementa tant a Linux com a BSD), es defineix una system call per gestionar la memòria de manera semblant a Mach. Quina és? Quina funció de més alt nivel coneixes que la faci servir? En quines circumstàncies?

A Linux existeix la crida:

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

que, com la crida de Mach, treballa en unitats de pàgina. La funció de LibC `void *malloc(size_t size)` que fa servir `brk()` o `mmap` depenent del valor de `size`.

(d) Explica quins advantatges veus al model actual de Linux. Creus que necessita d'un suport hardware?

Al Unix original, la gestió de memòria buscava tractar amb espais continus. La memòria dinàmica s'assignava movent el *program break* amb la crida `int brk(void *addr);`. Quan apareixen màquines amb mides grans d'espai d'adreces aquest model orientat a fer crèixer el segment `bss` començar a trontollar. Amb l'inclusió de la crida al systema `mmap()` es poden administrar moltes zones de memòria separades per forats, disperses per tot l'espai d'adreces del programa.

No necessita més suport hardware que l'inherent al model de memòria de paginació sota demanda.

3. Què li va passar al Pathfinder?

   Al juliol de 1997 es va iniciar la missió de la NASA "Mars Pathfinder" que va estar a punt de fracassar. Pocs dies després d"aterrar" el sistema va començar a patir resets. Explica en què va consistir el problema i com es va resoldre. Com a guia, mira de respondre a aquestes preguntes:

   (a) Quin sistema operatiu es va fer servir a la missió? Encara té suport actualment?

   VxWorks i sí, encara té suport actualment.

   (b) Quin va ser el problema? Com el van poder diagnosticar?

   Inversió de prioritats.

   VxWorks es pot executar en un mode on enregistra una traça de tots esdeveniments interessants del sistema, incloent canvis de context, usos d'objectes de sincronització i interrupcions. Després de la fallada, enginyers del JPL van passar hores i hores fent servir una rèplica exacta de la nau espacial amb les traces activades, intentant replicar les condicions exactes en què creien que s'havia produït el reset. De matinada, quan només un enginyer encara no s'havia anat a casa, finalment va reproduir el reset a la rèplica. L'anàlisi de la traça revelava la inversió de prioritats.

   (c) Quants threads estaven involucrats en el problema? Què feia cadascun?

   Hi va haver tres threads involucrats.

   El Pathfinder tenia una zona de memòria compartida (de fet, un bus de dades). Un thread d'alta prioritat gestionava aquesta memòria sovint, per posar i treure dades. L'accés a aquest bus estava sincronitzat amb mutex. Hi havia un altre thread de baixa prioritat dedicat a les dades meteorològiques que de tant en tant accedia al bus per deixar les seves dades, és a dir, adquiria el mutex, escrivia al bus i alliberava el mutex. També hi havia un thread de prioritat intermedia, dedicat a les comunicacions que no feia servir el bus.

   La cosa funcionava bé gairebé sempre. Si el thread d'alta prioritat es trobava el mutex agafat, es bloquejava fins que el thread de baixa prioritat alliberava el mutex. Però, en rares ocasions succeia que el thread de prioritat intermèdia accedia durant el breu període de temps que el thread d'alta prioritat estava bloquejat al mutex. El thread de baixa perdia el processador, sense alliberar el mutex, en favor del d'intermedia. Llavors es produïa un *timeout* quan el bus de dades feia massa temps que no era

gestionat i el sistema feia un reset.

(d) Com es va resoldre? Quins dels mecanismes disponibles a la llibreria de threads van fer servir?

Quan VxWorks s'executa en mode debug, conté un intèrpret de C que permet als programadors executar funcions i expressions *on the fly*. Els enginyers del JPL, sortossament, van decidir enviar el Pathfinder amb el mode debug activat. Els atributs del mutex estaven enmagatzemats amb les variables globals, les adreces de les quals quedaven a la taula de símbols i accessibles des de l'intèrpret de C. Els enginyers van carregar, des de la Terra, un petit program en C al Pathfinder, que un cop executat va canviar els atributs del mutex.

El mecanisme de la llibreria de threads emprat va ser l'herència de prioritats. És a dir, quan un thread està bloquejant threads de més prioritat que ell, aquest s'executa a la prioritat més alta d'aquells bloquejats per aquest mutex. Es va aprofitar un dels atributs del mutex, canviant el seu valor a `PTHREAD_PRIO_INHERIT`.

4. *Jitter* i les interrupcions En el nostre sistema, mirem l'assignació que tenen les interrupcions a processadors (CPUs), usant el fitxer d'informació /proc/interrupts:

```
$ cat /proc/interrupts # mostra els comptadors d'interrupcions arribades a cada CPU
           CPU0       CPU1       CPU2       CPU3
 0:        115          0          0          0   IO-APIC    timer               # rellotge
 1:      66487          0          0          0   IO-APIC    i8042               # teclat
 8:          0         65          0          0   IO-APIC    rtc0         # real time clock
 9:          0     286280          0          0   IO-APIC    acpi    # interprocessor interrupt
18:          0          0          0    1609132   IO-APIC    i801_smbus # system management bus
40:          0          0          0     387057   PCI-MSI    ahci                # USB 1
41:          0          0     712702          0   PCI-MSI  xhci_hcd              # USB 2
42:          0          0          0     102754   PCI-MSI    eth0                # xarxa
45:        926          0          0          0   PCI-MSI    snd_hda_intel:card1 # soundcard
46:          0    3626057          0          0   PCI-MSI  iwlwifi               # wifi
47:          0          0    2897210          0   PCI-MSI    i915            # tarjeta gràfica
48:          0          0          0        150   PCI-MSI    snd_hda_intel:card0 # soundcard
NMI:         0          0          0          0   Non-maskable interrupts
LOC:  52380920   48263461   51959641   48009742   Local timer interrupts # rellotge per CPU
RES:   3338614    2688932    2320548    1848684   Rescheduling interrupts(*)
CAL:   4492766    4433077    4450137    4561569   Function call interrupts(*)
TLB:   4490349    4430039    4448187    4558658   TLB shootdowns(*)

Fem algunes estadístiques sobre la recepció i distribució de les interrupcions:

- Totals:
       CPU0       CPU1       CPU2       CPU3
   64770177   63727911   66788425   61077746
- En percentatge sobre el total:
       CPU0       CPU1       CPU2       CPU3
```

```
      25.26%     24.85%    26.05%     23.82%

(*) Aquestes interrupcions s'anomenen Inter-Processor Interrupts (IPIs), i les utilitza l'S
```

Per donar una referència de quant de temps ha passat mentre es rebien totes aquestes interrupcions, aquest és l'uptime de la màquina:

```
$ uptime
16:38:31 up 2 days,  9:20, 16 users,  load average: 0.07, 0.06, 0.01
```

I aquesta és la distribució d'interrupcions per segon

```
                                   CPU0       CPU1       CPU2       CPU3
interrupcions per segon  313.8     308.8      323.6      295.9
total: 1242 interrupcions per segon
```

També mirem la configuració del kernel, pel que fa a la interrupció de rellotge:

```
# CONFIG_HZ_100 is not set
# CONFIG_HZ_250 is not set
# CONFIG_HZ_300 is not set
CONFIG_HZ_1000=y
CONFIG_HZ=1000
```

(a) Expliqueu el tema del jitter en el sistema operatiu, com afecta als processos i també a les aplicacions paral·leles, i relacioneu-lo amb l'assignació de les interrupcions que veieu en aquesta informació prèvia que hem vist.

> El jitter consisteix en l'entrada en execució de serveis, dimonis, interrupcions, que son necessaris pel funcionament normal del sistema, però que interfereixen en l'execució d'una aplicació. Si l'aplicació és paral.lela, l'afectació és habitualment major, perquè es propaga del procés/ flux afectat a la resta de processos/fluxos.
>
> Respecte al sistema donat, veiem que és un sistema de propòsit general, en el que l'arribada de les interrupcions d'ha distribuït entre els 4 cores que té l'ordinador. D'aquesta manera, qualsevol procés corrent en el sistema patirà el jitter. Podriem millorar la situació concentrant més les interrupcions a un sol core, per exemple el 0 i deixant la resta lliures d'interrupcions.

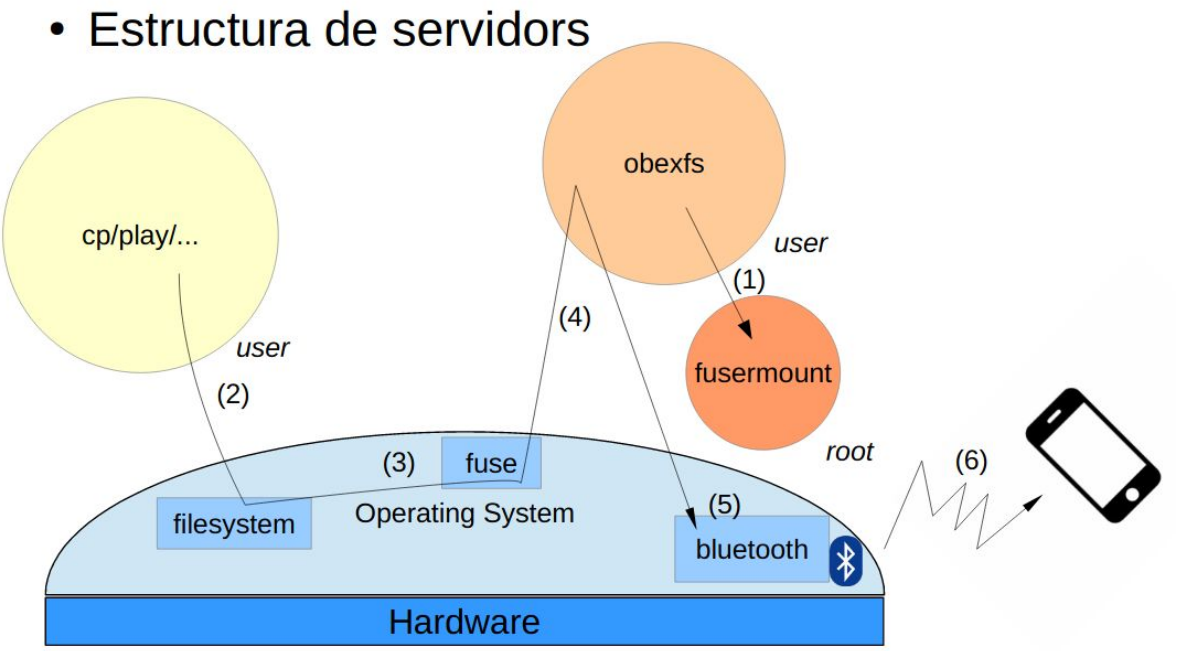
(b) Doneu la vostra opinió sobre si un sistema amb aquestes característiques podria servir per donar algun tipus de servei de temps real en mode hard.

> Aquesta opció d'aïllar les interrupcions a un sol core també seria recomanable per usar el sistema per suportar temps real en mode hard, perquè tal i com està no es pot garantir que no tinguis una ràfaga de 2 o més tipus d'interrupció en el core que té un deadline.

5. OBEX i accés a sistemes de fitxers remots

   Explica com OBEX - Object Exchange - ens permet accedir a dades exportades per altres dispositius (ordinadors, telèfons mòbils...). Usa el següent dibuix per millorar l'explicació.

   Per no haver d'entregar un dibuix, en l'explicació pots fer referència als punts numerats amb (1)



   .. (6).

(1) la comanda que usa Obex, utilitza el fusermount (setuid) per fer el muntatge del sistema de fitxers, usant File System in User Space - FUSE, a través de bluetooth. (2) llavors ja es poden usar comandes com cp/play... que faran les seves crides a sistema (read/write) per anar al punt de muntatge, (3) de forma que el sistema detectarà que ha de parlar amb el procés inicial d'Obex, a través del FUSE (4) i enviarà les peticions de read/write... al procés obexfs, que les traduirà a missatges en el protocol bluetooth (5), per arribar fins el dispositiu mòbil (6).