



Sistemes Operatius

Capítol 2: El nucli d'un sistema operatiu

Tema 2. El nucli del Sistema Operatiu

1. Descripció
2. Esquema del nucli
3. Representació dels processos PCB (*Process Control Block*)
4. Controlador de la interrupció: FLIH (*First Level Interruption Handler*)
5. El Dispatcher
6. Mecanismes de comunicació, sincronització i exclusió mútua
7. Classificació dels kernels
8. Nucls de SO contemporanis

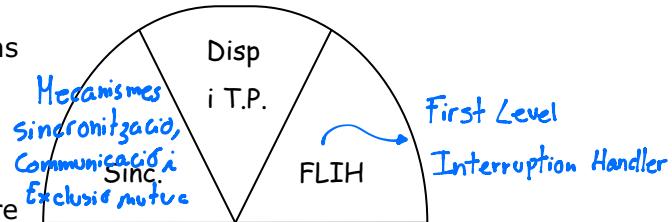
Tema 2: El nucli d'un Sistema Operatiu

1. Descripció

Nucli és la part del SO que interacciona directament amb el hardware.

2. Esquema del nucli

1. Controlador d'interrupció: gestiona les interrupcions del sistema
2. Dispatcher: s'encarrega de commutar tasques.
3. Mecanismes de sincronització / comunicació entre processos.



Exemple. Implementació dels semàfors amb les primitives wait() i signal().

3. Representació dels processos: PCB (Process Control Block)

- Nom del procés (PID del Unix equivalent)
- Estat
- Bloc de context: es guardarà amb el PCB
 - comptador de programa
 - registre de flags
 - registres de propòsit general
 - registres d'accés a memòria

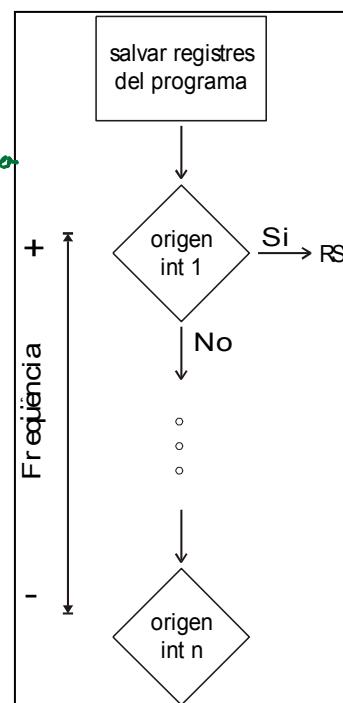
Altra informació que es podria guardar

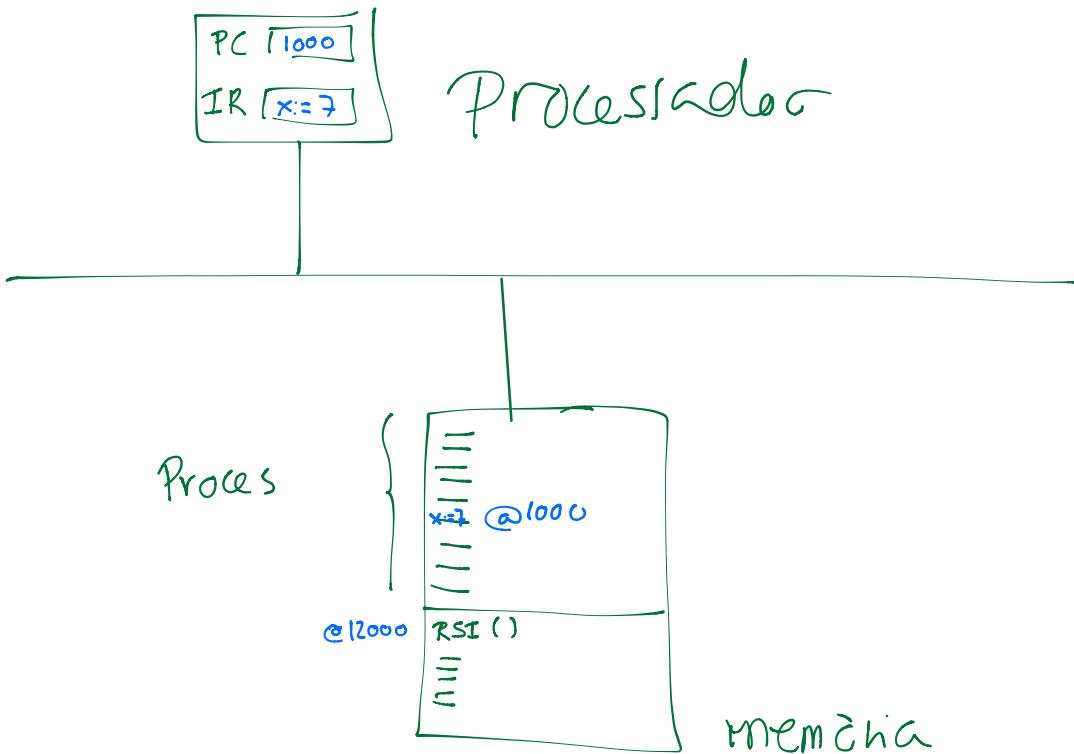
- info accounting (temps real execució, ...)
- info scheduling (per exemple prioritat del procés)
- info de l'estat E/S (per exemple la llista de dispositius associats, llista de fitxers oberts, etc.)

4. Controlador de la interrupció: FLIH (First Level Int Handler)

Proporciona la resposta adequada a interrupcions hardware i software. Funcions:

1. Salvar el context del procés actual: registres que anem a modificar a la RSI = context
2. Determinar l'origen de les interrupcions
 - preguntar a tots els dispositius un a un, segons una prioritat determinada, pels que més s'utilitzen o menys.
 - Tenir hardware que distingeix les int i salta automàticament a la RSI (apareix la prioritat de int)





↑ estem fent el proc, ens trobem a @1000 d'un proc
RSI és l'@ 12000 → Qui és el que diu que anem a @12000 per usoldre la interrupció

i després continuar? → FLIH ← organitza
Que fa?

↳ No anirà de l'@1000 a l'@12000 directe, el FLIH primer mira la taula per veure quina interrupció tenim i on la trobem per usoldre! → Aquesta taula està en disc dur: la carrega = la RAM per mirarla.

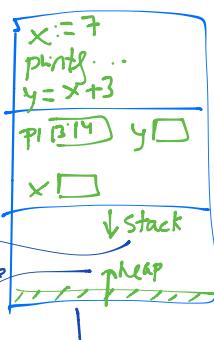
FLIH abans de carregar l'@ a la que hem d'aner a executar la interrupció guarda l'@ a la que estavem: el que estavem fent.

↳ guardam a una pila

Organització d'un programa/proc

stack → on es guarden var local i parametres per valor

heap → guardem memòria dinàmica o de l'usuari (màlles, etc)



codi

dades → var i const
globals

Si tinguessim corrdumped,
guardem info, veiem que és CD,
fem exit i eliminem el guardet
si fessim control C, marxarem,
però no eliminarem la info de la Pila



Quan tenim una interrupció, doncs,
el FCH guarda la info en la que
estava fent al stack^{n(pila)}, quan ha
acabat fa un POP i continua ...

✓ explicat a conti
nua a la pàg

interrupcions funcionen individualment per a cada procés

3. Executa RSI

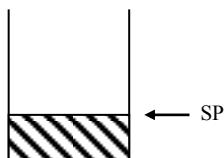
Com salvem i restaurem els contextos?

- Automàticament, el hardware ens ajuda i ho fa sol (en casos de CPUs pensades per multitasca).
- Manualment, el hardware no col·labora (per CPUs no pensades per multitasca per exemple 8086).

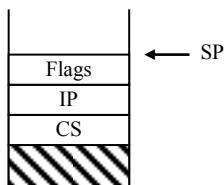
Anem a veure un canvi de tasca a càmera lenta. A fi d'entendre com funciona aquest canvi de tasca, cal veure que passa amb les piles del programes. Parlem de piles en plural perquè cada programa ha de tenir la seva pila, la seva zona de dades i la seva zona de codi independents.

Passos:

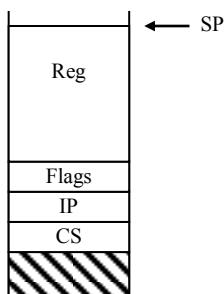
- 1) La tasca A està executant codi i té la pila carregada fins a un cert nivell.



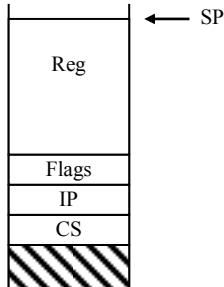
- 2) Es produeix una ITR (interrupció de temps real). Aquesta crida, com totes les interrupcions, provoca que el CS (code segment), l'IP (Instruction Pointer) i els flags es posin a la pila.



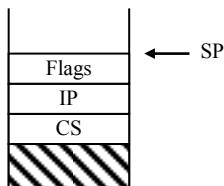
- 3) La RSI posa tots els registres a la pila, amb la intenció de restaurar-los al final de la RSI.



- 4) El dispatcher salva l'entorn volàtil a la zona d'entorn volàtil que hi ha a l'struct associat a la tasca de processos. Per aconseguir-ho, només cal que salvi SS (Stack Segment) i SP (Stack pointer)
- 5) Busca a la taula de processos el següent procés a ser executat, llegeix el seu entorn volàtil (SP' i SS') i el restaura. A partir d'aquest moment, la pila (SS i SP) està apuntant a la pila del procés que cal executar. Com que qui ha parat anteriorment aquest procés és el dispatcher, és d'assumir que la pila estarà disposada de la mateixa manera que la pila del procés A, però amb la informació del procés B, ja què és la pila de B. És a dir, la pila de B seria:



- 6) Ara s'executa la sortida de la RSI. En primer lloc, es restauren els registres (registres que tenia B abans que l'aturessin).



- 7) Abans de sortir de la RSI s'executa, com sempre, la instrucció reti. Aquesta provoca que el processador tregui els flags, IP i CS de la pila, és a dir, la CPU retornarà a executar el codi de la tasca B i la pila estarà tal i com l'havia trobat el dispatcher.

Reflexions:

- 1) Després del punt 7, estem executant el codi de B amb els valors dels registres, flags i pila que tenia B.
- 2) L'única cosa que hem fet és salvar i restaurar l'entorn, la resta ha vingut sol.
- 3) Per a què tot a això funcioni cal observar que les tasques d'usuari no poden disposar de EI/DI (Enable Interrupt/Dissable Interrupt), ja que si fan DI fins que no tornen a executar EI, la RSI de temps real queda inhibida, és a dir, el dispatcher no funciona.
- 4) Caldrà que la màquina disposi d'un rellotge capaç d'emetre interrupcions periòdiques (un timer)

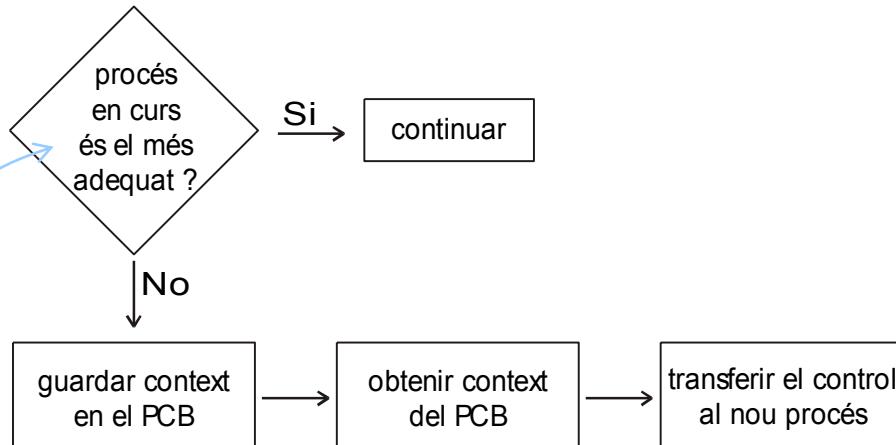
5. El Dispatcher (o Scheduler de Baix Nivell)

Quan haurem de cridar el dispatcher ?

- a) Després que una interrupció canviï l'estat d'algun procés
- b) Quan una interrupció interna faci que no pugui seguir executant-se
- c) Després que un senyal d'error provoqui la suspensió del procés actual.

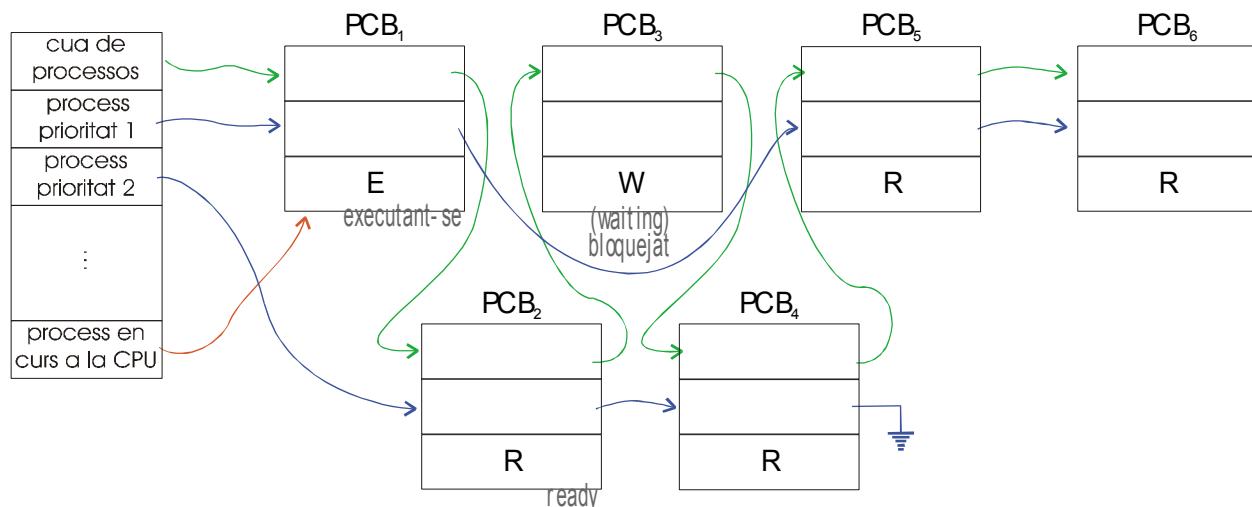
s'apunta tots els processos que
té en execució →
té apuntat la
info de cada
programa .

Funcionament



Com sabem això? Tindrem una estructura com la següent:

Estructura Cua Processos



Què hi guardem a cada struk de la we?

• @ execució

• context

• prioritat

• estat \rightarrow running \rightarrow executant

\downarrow ready \rightarrow execució

\hookrightarrow blocked \rightarrow per cosa que esperem \rightarrow està executant però s'ha de fer fins que el que volem

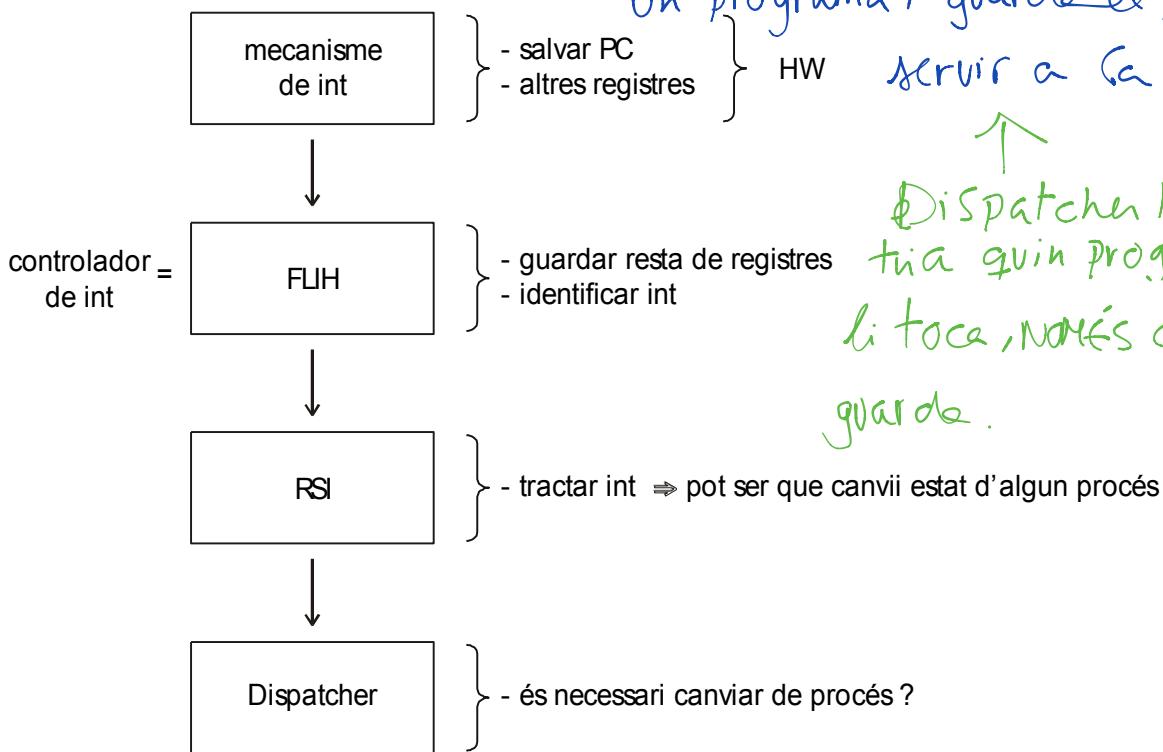
• mida \rightarrow saber quan ocupa el programa

• id \rightarrow id de procés \leftarrow PID.

\hookrightarrow és únic, per poder identificar TOTS els que tenen en execució

• recursos que estem fent servir \rightarrow ex: ⁶saber fitxers que estem fent servir d'aquest programa. Ex: si estem amb un programa i fem int c

Relació de FLIH i Dispatcher



Per tant → Dispatcher és el que fa el canvi i carrega a CPU un programa i guarda el que feiem servir a la CUD.

Dispatcher NO tria quin programa/procés li toca, NOMÉS carrega i guarda.

6. Mecanismes de Comunicació, sincronització i exclusió mútua

La majoria de mecanismes estan dins del nucli del sistema operatiu.

→ per a intercanviar info entre un procés & B. Tenim diferents mecanismes.

- **Comunicació:** mecanismes que permeten comunicar processos: intercanvi d'informació. Per exemple pipes, sockets, cues de missatges, memòria compartida.
 - **Sincronització:** mecanismes que permeten sincronitzar processos. Per exemple semàfors i monitors.
 - **Exclusió mútua:** mecanismes que permeten garantir la integritat de la informació compartida entre processos. Per exemple semàfors i monitors.
- P1(1) P2(1)*
llegir(x) escrit(x)
P1 P2
mem compartida *amb la sincronització el P2(1) s'evantara si el P1 ho ha fet, si no, perquè sino podríem escriure basura.*

7. Classificació dels kernels

Els kernels els podem classificar en diferents famílies: kernels monolítics, microkernels, kernels híbrids i exokernels. Aquesta classificació es fa segons com està organitzat el kernel internament i com interactuen entre si les diferents parts. Hi ha molta controvèrsia sobre qui és el millor model, i no s'ha arribat i probablement sigui difícil de que s'arribi a un consens sobre la millor opció, hi ha massa factors que hi influeixen. En aquest apartat veurem una petita introducció a cada model.

7.1 Espai d'usuari versus espai de sistema

Com hem anat veient fins ara, el kernel ha d'estar en memòria en qualsevol moment; també hem vist que les aplicacions han d'estar en memòria en el moment d'executar-se; tot el que s'executa ha de passar per memòria. També hem vist, però, que el kernel ha de poder garantir una certa estabilitat, fins i tot comptant que les aplicacions poden estar mal fetes i modificar regions de memòria que no els pertoquen.

Així doncs, com evitem que una aplicació que està en memòria pugui modificar el kernel que també hi és ? La solució ha estat tenir una separació entre espai de sistema i espai d'usuari. La separació entre l'espai de sistema i l'espai d'usuari s'aconsegueix gràcies a una protecció que proveeix el processador. Aquest pot estar executant codi en diferents "rings", de més o menys privilegis, d'aquesta manera podem posar el kernel en un "ring" de més privilegi que el de les aplicacions, així podrem protegir l'espai de memòria on corre el kernel, i també aconseguir que diferents aplicacions no es "vegin" entre si.

Per saltar d'espai d'usuari a espai de sistema es fa mitjançant interrupcions software (trap) que el kernel controlarà. Una aplicació demanarà realitzar una operació que cal que es faci en espai de sistema mitjançant una interrupció software. D'aquesta manera el kernel prendrà el control i realitzarà les comprovacions necessàries per veure si aquesta operació la pot realitzar o no.

L'ús d'aquestes interrupcions software per comunicar-se amb l'espai de sistema s'anomenen crides a sistema.

Per tal de poder distingir els diferents models de kernel és molt important saber quines parts d'aquest s'executen en espai de sistema i quines parts es deixen com a aplicació en espai d'usuari.

7.2. Kernel monolític

Un kernel monolític és aquell que inclou tots els serveis dins seu. Tot s'executa en espai de sistema. Això inclou els serveis del nucli i els drivers. D'aquesta manera el nombre de canvi de contextos entre espai de sistema i espai d'usuari s'intenta minimitzar. Aquest aspecte incrementa teòricament el rendiment d'aquest tipus de nucli.

Que els drivers hagin d'estar en espai de sistema no significa que tots els drivers hagin d'estar carregats en memòria, però si que quan aquest s'hagin d'usar s'ha de passar d'alguna manera a espai de sistema.

Un aspecte molt crític en els kernels monolítics és el tema de la portabilitat. Solen ser molt difícils de portar a arquitectures diferents a les que han estat inicialment programats. Això és degut a que són porcions molt grans de codi que estan totes lligades a la màquina. Si vols portar-lo a una arquitectura diferent ho has de reescriure tot tenint en compte que ha de funcionar en una altre tipus diferent de màquina. Tot i així, si veiem els kernels monolítics que hi ha fets veurem que han estat portats a moltes arquitectures. Això se sol aconseguir amb l'ús de directives de precompilació, on indiquen per quina arquitectura es compilà el kernel.

Un dels avantatges principals dels kernels monolítics és el fet que són els més simples de desenvolupar, pots tenir un gran nombre de funcionalitats sense haver de començar a fer ús de opcions de protecció bastant complicades del processador, com ja hem dit l'altre avantatge és la suposada millora de rendiment pel fet de reduir el nombre de canvis de contextes, això pot ser cert en X86 però no té perquè ser-ho en altres arquitectures.

La filosofia dels kernels monolítics és que tot el codi que va dins del kernel ha de ser correcte, una part de codi que no funcioni bé dins del kernel fa que tot el sistema tingui problemes, així s'ha de tenir molta cura en el que s'acaba admetent dins del sistema, aquest cas sol ser crític en els drivers realitzats per tercers, un problema en un driver pot fer que el sistema caigui, ja que la protecció que es pot donar és mínima.

Una bona referència és un article de Andrew S. Tanenbaum a la revista Computer de Maig de 2006 anomenat "Can We Make Operating System Reliable and Secure?" pàgines 44-51

7.3. Microkernel

L'enfoc del microkernel consisteix en definir una capa d'abstracció molt simple sobre del hardware que corre en espai de sistema, aquesta capa conté el conjunt mínim de funcionalitats per tal de poder usar tot el maquinari de forma multiplexada, ens proporciona així l'abstracció i la protecció necessàries per poder implementar qualsevol altra capa fora de l'espai de sistema.

Normalment en el microkernel s'inclouen els serveis bàsics maneig de memòria, l'scheduling i la comunicació interna. El temps de resposta dels microkernel es suposa millor ja que en qualsevol moment es pot interrompre qualsevol tasca que estigui corrent a nivell d'usuari, i la major part del temps se'l passa en aquest espai. Té beneficis també en sistemes distribuïts i multiprocessadors doncs és molt més fàcil separar els serveis, la contrapartida és el major nombre de canvi de contextes que es fa que pot suposar a tenir un cost sobre el rendiment bastant alt.

En contra als kernels monolítics els microkernels tendeixen a buscar una independència major amb l'arquitectura del sistema, això es deu a que només s'ha de reescriure la part que depèn directament del hardware, aquesta capa s'anomena HAL (Hardware Abstraction Layer), d'aquesta manera ens estalviem de reescriure tota la part de serveis que usa la HAL.

7.4 Kernels Híbrids

Els kernels híbrids són una barreja de kernel monolític i microkernel, intentant treure el millor de cada un, la idea és tenir part de codi no essencial dins de l'espai de sistema, sent aquest codi aquell que ens dona un millor augment de rendiment. La majoria de sistemes operatius cauen en major o menor mesura dins aquesta categoria, ja sigui perquè són microkernels que han afegit codi, o bé kernels monolítics que han tret fora de l'espai de sistema certes parts.

S'ha de veure clara la diferència entre un kernel híbrid i un monolític on carreguem parts dinàmicament, el primer té parts de serveis que trobaríem en espai de sistema que els han passat a espai d'usuari, mentre que el segon carrega parts a espai de sistema quan les necessita.

7.5. Exokernels

Aquest és un enfoc completament diferent al que hem vist fins ara. La idea és que el programador que treballa sobre un exokernel tingui control total sobre el hardware, sent el kernel realment petit i oferint només multiplexació de recursos i certa protecció. D'aquesta manera podem tenir corrent a sobre altres kernels de manera que l'exokernel els hi deixa usar els diferents recursos de forma que els hi "sembli" que tenen tot el maquinari a disposició.

Mentre els termes de microkernel i kernel monolític estan ben definits l'exokernel té diferents variants com són nanokernel, picokernel, caché kernel o virtualizing kernel.

8. Nucls de SO contemporanis

Nutt, G. Sistemas operativos, 3^a Edició, Addison Wesley, 2004, pàgs. 98-103

Nucli de UNIX

Desenvolupat a principis de la dècada de 1970, és un kernel monolític amb funcions bàsiques de gestió de processos, memòria, arxius i dispositius d'E/S.

Nucli de Linux

Linus Torvalds al 1991.
És un kernel monolític però amb mòduls instal·lables dinàmicament.

Nucli de Windows NT

Primerament sembla tenir una arquitectura estil microkernel però mirat més detalladament sembla una estructura basada en capes.