Tema 1: Estrucutura dels Sistemes Operatius

Apunts de Sistemes Operatius

Jordi Mateo Francesc Solsona

Índex

Dualitat Kernel/Usuari **1**
- Kernel . . . 1
- Mode Usuari . . . 1
- Comunicació entre Kernel i Usuari . . . 2
  - Cas d'Estudi: Crida al Sistema `read()` . . . 2
  - Cas d'Estudi: Hooks . . . 2
- Canvis de Context . . . 3
  - Exemple de Canvi de Context . . . 3
- Interrupcions i Excepcions . . . 3
  - Interrupcions . . . 3
  - Excepcions . . . 4

Estructura dels Sistemes Operatius **4**
- Arquitectura Monolítica . . . 5
  - Característiques de l'arquitectura monolítica . . . 5
  - Implementació de l'arquitectura monolítica . . . 5
  - Modularització . . . 6
  - Exemples d'ús de l'arquitectura monolítica . . . 6
  - Resum de l'arquitectura monolítica . . . 6
- Artuitectura en Capes . . . 6
  - Característiques de l'Arquitectura en Capes . . . 7
  - Implementació de l'Arquitectura en Capes . . . 7
  - Exemples d'Ús d'Arquitectura en Capes . . . 8
  - Resum de l'Arquitectura en Capes . . . 8
- Arquitectura Microkernel . . . 8
  - Característiques de l'Arquitectura Microkernel . . . 8
  - Exemples d'Arquitectura Microkernel . . . 9
  - Resum de l'Arquitectura Microkernel . . . 9
- Altres Arquitectures . . . 10

Dualitat Kernel/Usuari

Els sistemes operatius moderns estan dissenyats per proporcionar una interfície entre les aplicacions d'usuari i el maquinari del sistema. Aquesta interfície es pot dividir en dues parts principals: el **kernel** i **l'espai d'usuari**. Aquesta separació és fonamental per garantir la seguretat, l'estabilitat i la gestió eficaç dels recursos del sistema.

Kernel

El kernel és la part central del sistema operatiu, que es carrega a la memòria quan s'inicia el sistema. És responsable de gestionar els recursos del sistema, com *la memòria, la CPU, els dispositius d'entrada/sortida i els processos*. Aquesta part del sistema operatiu s'executa en mode privilegiat, també conegut com a **mode kernel** o **mode supervisor**, que li permet accedir directament al maquinari i gestionar els recursos del sistema sense restriccions.

Mode Usuari

El mode usuari és l'entorn on s'executen les aplicacions i els programes d'usuari. Aquestes aplicacions s'executen en un entorn aïllat del kernel, amb restriccions d'accés als recursos del sistema. Per poder accedir als recursos del

sistema, com la memòria o els dispositius d'entrada/sortida, les aplicacions han de fer **crides al sistem** a, que són interceptades pel kernel i executades en mode privilegiat.

Comunicació entre Kernel i Usuari

La comunicació entre el kernel i l'usuari es realitza a través de crides al sistema, que permeten a les aplicacions sol · licitar serveis al sistema operatiu. Aquestes crides es realitzen a través d'una interfície estàndard que proporciona el sistema operatiu, com ara les **llibreries de sistema** o les **API (Application Programming Interface)**.

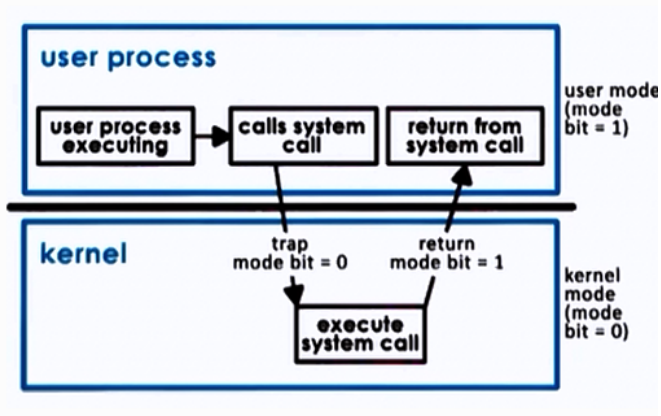

Figura 1: Dualitat Kernel/Usuari

Cas d'Estudi: Crida al Sistema `read()`

Imaginem un usuari que executa un programa en C que necessita llegir un fitxer del disc. Aquest programa s'executa en mode usuari i no té accés directe al disc. Per tant, ha de fer una crida al sistema per sol · licitar al kernel que llegeixi el fitxer del disc. El kernel, en rebre la crida, accedeix al disc, llegeix el fitxer i retorna les dades a l'aplicació.

Anem a veure-ho pas a pas:

1. L'usuari executa un programa en mode usuari que necessita llegir un fitxer del disc. `read(fd, buffer, size)`.
2. Els arguments fd, buffer i size es guarden als registres RDI, RSI i RDX, respectivament (en arquitectura x86_64). Els registres RD, RSI i RDX contenen els arguments de la crida al sistema.
3. El progrma fa una crida a la llibreria que conté la funció `read()`
4. El sistema operatiu afegeix el codi de `read()` al registre RAX (en arquitectura x86_64). El registre RAX conté el número de la crida al sistema.
5. El sistema operatiu executarà `syscall` per passar el control al kernel. Això s'anomena **TRAP**.
6. El kernel localitza el `handler` de la crida al sistema `read()` i executa el codi associat. Per fer-ho, busca l'apuntador a la taula de crides al sistema i executa la funció associada al valor de RAX.
7. El kernel utilitza els arguments dels registres RDI, RSI i RDX per llegir el fitxer del disc.
8. Col · loca les dades llegides al buffer (paràmetre de sortida).
9. Retorna el control al programa d'usuari. Sol · licita la interrupció `syscall` per tornar al mode usuari.
10. La llibreria de sistema que conté la crida al sistema `read()` retorna el control el resultat al programa d'usuari.
11. El programa d'usuari pot continuar amb l'execució amb les dades llegides del disc o bé tractar l'error si n'hi ha hagut, observant el valor de retorn de la crida al sistema o el contingut de la variable `errno`.

Aquesta comunicació entre el kernel i l'usuari a través de crides al sistema permet a les aplicacions interactuar amb el sistema operatiu i accedir als recursos del sistema de manera segura i controlada.

Cas d'Estudi: Hooks

Imagina que un atacant pot modificar l'adreça d'una funció de la taula de crides al sistema. Es a dir, un usuari malintencionat modifica l'apuntador a la funció `read()`. Això permetria a l'atacant executar un codi maliciós cada vegada que es cridi aquesta funció i retornar al usuari un resultat aparentment vàlid. Aquesta és una de les raons per les quals el mode kernel és tan important i ha de ser protegit a tota costa.

Canvis de Context

Els recursos físics de processament són limitats, i el sistema operatiu ha de canviar el context d'execució entre el mode usuari i el mode kernel. Això es coneix com a **canvi de context**.

Assumeix que tenim un únic processador. La dualitat requereix que el sistema operatiu pugui canviar el context d'execució entre el mode usuari i el mode kernel. Això implica canviar l'estat del processador i la memòria per passar de l'execució d'un procés d'usuari a l'execució del kernel i viceversa.

Els passos per realitzar un canvi de context són els següents:

1. **Guardar el context actual**: Abans de canviar el context, el sistema operatiu ha de guardar l'estat actual del processador, com els registres, el comptador d'instruccions i l'estat de la memòria. Aquesta informació es guarda en una estructura de dades anomenada **context de procés**.
2. **Canviar el context**: Un cop s'ha guardat el context actual, el sistema operatiu pot canviar el context per passar de l'execució d'un procés d'usuari al kernel o viceversa. Aquest canvi de context es realitza a través d'una instrucció específica del processador, com ara `syscall` o `int`.
3. **Restaurar el context**: Un cop s'ha canviat el context, el sistema operatiu ha de restaurar l'estat del processador amb les dades del nou procés que s'executarà. Aquesta informació es recupera del context de procés guardat en el pas anterior.

Exemple de Canvi de Context

Considerem dos processos, P1 i P2, que s'executen al mateix temps. Cada cop que el sistema operatiu vol canviar de context entre P1 i P2, ha de seguir els passos anteriors per:

- Desar l'estat de P1.
- Canviar el context per executar P2.
- Restaurar l'estat de P2.

Aquesta tasca la realitza el sistema operatiu a través del **planificador de processos**, que és responsable de gestionar l'execució dels processos i decidir quin procés s'executarà en cada moment. Per tant, cada cop que el planificador decideixi canviar de context, haurà de canviar de mode usuari a mode kernel i viceversa.

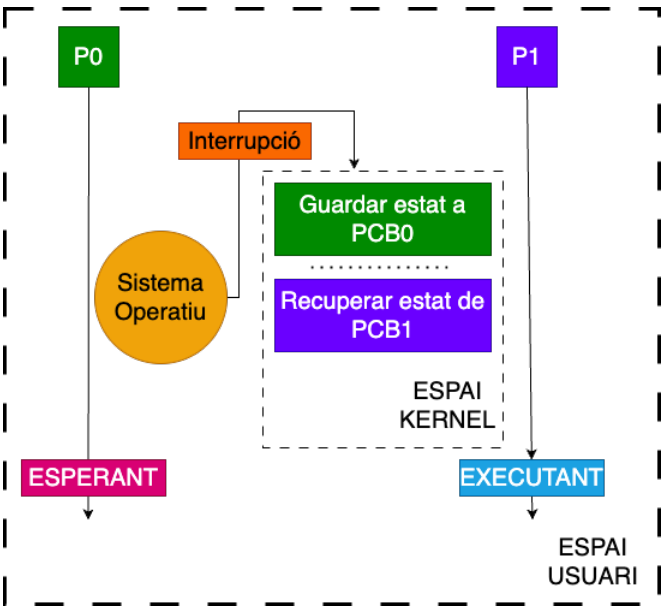

Figura 2: Canvi de Context

Interrupcions i Excepcions

A part de les **TRAP**, hi ha dos mecanismes que permeten canviar el control del sistema operatiu al kernel: les **interrupcions** i les **excepcions**.

Interrupcions

Les interrupcions són senyals generades per dispositius externs o per altres parts del sistema que requereixen atenció immediata del sistema operatiu. Poden ser de software o hardware i inclouen interrupcions de rellotge,

d'entrada/sortida o de hardware.

Exemple d'Interrupció Quan un dispositiu d'entrada/sortida ha completat una operació, genera una interrupció per notificar al sistema operatiu que les dades estan disponibles. El sistema operatiu respon a aquesta interrupció canviant el control al kernel per gestionar-la. El seu tractament és **prioritari** i no es poden desactivar.

Passos per gestionar una interrupció

1. El dispositiu genera una interrupció.
2. El processador interromp l'execució del programa actual i salta a la rutina d'interrupció associada a l'interrupció.
    1. Mode usuari -> Mode kernel.
    2. Desar el context del procés actual.
    3. Buscar la rutina d'interrupció associada a l'interrupció.
3. S'executa la rutina d'interrupció.
4. Es restaura el context del procés i es continua amb l'execució del programa.

Excepcions

Les excepcions són situacions excepcionals que es produeixen durant l'execució d'un programa, com ara errors de divisió per zero, accés a memòria no vàlida o instruccions no vàlides. Les excepcions poden ser provocades per errors en el codi del programa o per condicions inesperades en l'execució. El seu tractament és **no prioritari** i es poden desactivar.

Tipus d'Excepcions

- **Fault**: Aquesta excepció informa abans de l'execució de la instrucció que ha provocat l'excepció. Per exemple, una excepció de falta de pàgina.
- **Trap**: Aquesta excepció informa després de l'execució de la instrucció que ha provocat l'excepció. Per exemple, una crida al sistema.
- **Abort**: Aquesta excepció no informa de l'instrucció que ha provocat l'excepció. Per exemple, una excepció de protecció.

Exemple d'Excepció Imagineu que un programa intenta accedir a una adreça de memòria que no té permís d'accés. Aquesta acció provocarà una excepció de protecció, que el sistema operatiu haurà de gestionar per evitar que el programa es bloquegi o corrompi la memòria del sistema.

Exemples:

Nom	Descripció	Tipus
Divisió per zero	Intent d'executar una divisió per zero.	**Fault**
Falta de pàgina	Intent d'accedir a una pàgina de memòria que no està carregada a la memòria principal.	**Fault**
Protecció de memòria	Intent d'accedir a una regió de memòria protegida.	**Abort**
Instrucció no vàlida	Intent d'executar una instrucció no vàlida.	**Abort**
Interrupció de rellotge	Interrupció generada pel rellotge del sistema.	**Trap**
Overflow aritmètic	Intent d'emmagatzemar un valor més gran que el permès en un registre.	**Trap**
Excepció de punt flotant	Excepció generada per operacions aritmètiques amb nombres de punt flotant.	**Trap**

Estructura dels Sistemes Operatius

Els sistemes operatius poden dissenyar-se seguint diferents models arquitectònics, depenent dels objectius de rendiment, seguretat i mantenibilitat que es persegueixin. A continuació, es descriuen alguns dels dissenys més habituals.

Arquitectura Monolítica

Un sistema operatiu **monolític** es defineix per estar format per un únic conjunt de funcions que operen íntegrament en mode kernel, és a dir, amb accés il · limitat a tot el maquinari del sistema. Per tant, tenim un **únic programa** que opera íntegrament en mode **kernel**, és a dir, amb *accés total als recursos de maquinari*. Això vol dir que totes les funcionalitats del sistema (gestió de processos, memòria, dispositius d'entrada/sortida, etc.) estan integrades dins d'un únic espai de memòria compartit. Tot i això, es possible estructurar els kernels monolítics de manera que utilitzin la dualitat **kernel/usuari**.

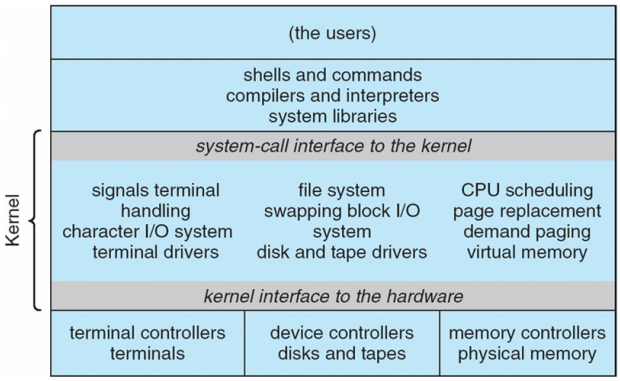

Figura 3: Arquitectura Monolítica

Característiques de l'arquitectura monolítica

1. **Execució en mode kernel**: Tot el codi del sistema operatiu, incloent-hi controladors de dispositius, el planificador de processos, el sistema de fitxers i el gestor de memòria, s'executa amb privilegis de mode *kernel*. Això implica que totes les operacions poden accedir directament al maquinari i a la memòria sense restriccions.
2. **Crides al sistema**: Els processos d'usuari poden accedir a les funcionalitats del sistema operatiu a través de crides al sistema, que permeten sol · licitar serveis. Aquestes crides es realitzen a través d'una interfície comuna que permet als programes d'usuari interactuar amb el sistema operatiu.
3. **Gestió centralitzada**: Totes les funcionalitats del sistema operatiu estan integrades en un únic programa, fet que facilita la seva gestió i manteniment. Això permet una interacció ràpida i directa entre les parts del sistema.
4. **Alta eficiència i baix aïllament**: No hi ha barreres entre els diferents components del sistema. Això permet una interacció ràpida i directa entre les parts del sistema. No obstant això, aquesta falta d'aïllament també introdueix riscos importants: un error en qualsevol component del sistema pot corrompre o danyar el funcionament global.
5. **Dificultat de manteniment**: A mesura que el codi del sistema creix, la seva complexitat també ho fa, fet que dificulta la seva comprensió, depuració i manteniment. Cada vegada que es vol afegir una nova funcionalitat o corregir un error, cal modificar el codi del kernel complet, la qual cosa pot ser perillós i costós en termes de temps i esforços.

Implementació de l'arquitectura monolítica

Un sistema operatiu monolític es pot implementar utilitzant llenguatges de programació com *C*, que permet un accés directe a les funcionalitats de baix nivell del maquinari. El procés de compilació implica agrupar tot el codi font del sistema operatiu en un únic executable a través d'un linker, que enllaça les diverses funcions en un sol fitxer binari. Aquest executable conté totes les funcions del sistema operatiu i es carrega a la memòria quan s'inicia el sistema.

A més del kernel, que es carrega sempre durant l'arrancada d'un ordinador, molts sistemes operatius monolitics suporten extensions carregables sota demanda, com són les **shared libraries** o les **dynamic link libraries**. Aquestes extensions permeten ampliar les funcionalitats del sistema sense necessitat de reiniciar-lo, ja que es poden carregar i descarregar dinàmicament en temps d'execució.

Modularització

Els sistemes monolítics clàssic tendeix a ser rígids i dificils de mantenir i actualitzar. Per aquest motiu, molts d'ells han evolucionat cap a una arquitectura més modular. Aquest enfocament modular permet carregar i descarregar components del sistema operatiu de manera dinàmica, millorant així la seva flexibilitat i mantenibilitat. Un exemple clar d'aquesta evolució és el sistema operatiu Linux.

Encara que el kernel bàsic es manté monolític, molts dels seus components es poden carregar i descarregar sota demanda, com si fossin mòduls. Aquests mòduls són peces de codi que ofereixen funcionalitats específiques, com ara controladors de dispositius o extensions de seguretat, que es poden afegir o treure sense necessitat de reiniciar el sistema o recompilar tot el kernel.

Els avantatges d'aquest enfocament són:

- **Carregament dinàmic de mòduls**: Els mòduls es poden carregar dinàmicament mitjançant modprobe o descarregar amb rmmod, permetent ampliar les funcionalitats del sistema operatiu sense interrupcions.
- **Optimització de la memòria**: Només es carrega a la memòria el codi necessari, millorant l'eficiència i l'ús dels recursos.
- **Seguretat i estabilitat**: Si un mòdul té un error, es pot descarregar o substituir sense afectar la resta del sistema, millorant la resiliència del sistema en comparació amb els sistemes monolítics tradicionals.

Exemples d'ús de l'arquitectura monolítica

El kernel de Linux és un exemple clàssic d'arquitectura monolítica modular. En la figura es mostra un esquema simplificat on es pot observar com tot el codi del sistema operatiu s'executa en mode kernel.

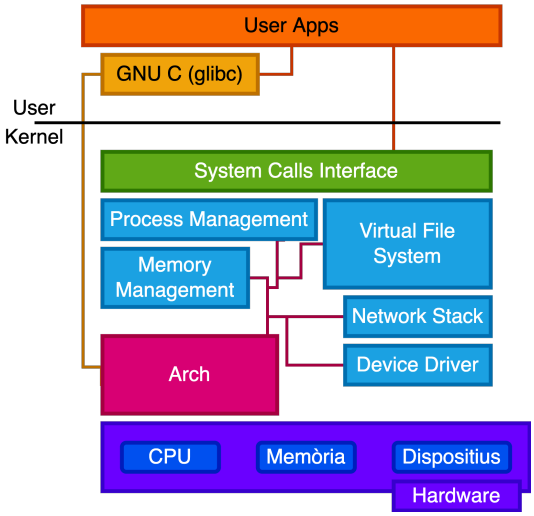

Figura 4: Arquitectura Línux

Resum de l'arquitectura monolítica

Avantatges	Inconvenients
Rendiment elevat	Manca d'aïllament
Accés directe al maquinari	Manteniment complex
Modularitat en kernels com Linux	

Artuitectura en Capes

L'arquitectura en capes és un model de disseny utilitzat en la construcció de sistemes operatius que divideix el sistema en nivells d'abstracció, cadascuna d'elles amb un conjunt específic de responsabilitats i funcionalitats. Aquesta divisió té com a objectiu millorar l'organització del codi, facilitar la seva mantenibilitat i permetre un desenvolupament modular, basat en el principi de **separació de responsabilitats**. Les capes interaccionen de manera jeràrquica, on les capes superiors depenen de les capes inferiors per accedir a serveis de baix nivell.

En la figura es mostra un esquema simplificat d'un sistema operatiu basat en arquitectura en capes, on es pot observar com les diferents capes del sistema interactuen entre elles.

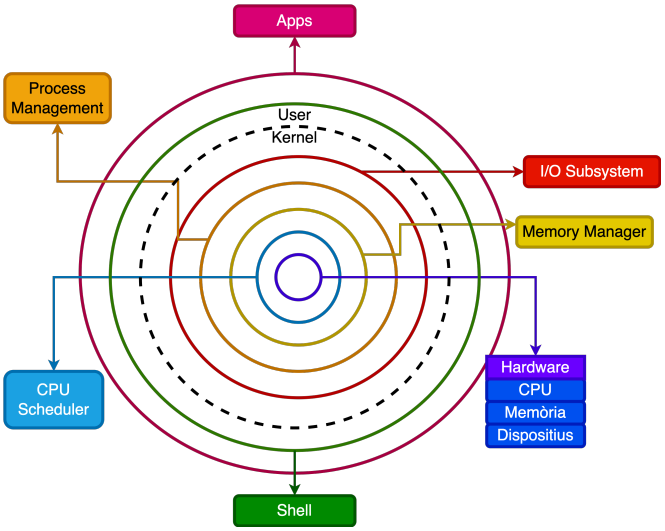

Figura 5: Arquitectura en Capes

Característiques de l'Arquitectura en Capes

1. **Separació de responsabilitats**: Cada capa del sistema operatiu té una responsabilitat ben definida i ofereix serveis a la capa immediatament superior. Això permet que el sistema estigui estructurat de manera ordenada i coherent, facilitant-ne la comprensió i manteniment.
2. **Interdependència entre capes**: Les capes superiors no poden accedir directament als recursos del sistema (com ara el maquinari). Han de fer-ho a través de serveis proporcionats per les capes inferiors. Així es garanteix que cada nivell estigui aïllat dels detalls de les implementacions més baixes, evitant l'accés indiscriminat als recursos i augmentant la seguretat i la robustesa del sistema.
3. **Mantenibilitat i extensibilitat**: La modularitat d'aquesta arquitectura facilita les tasques de desenvolupament, prova i actualització de cadascuna de les capes. Qualsevol canvi o millora es pot aplicar en una capa concreta sense afectar necessàriament les altres, sempre que es mantinguin les interfícies establertes.
4. **Abstracció**: Les capes superiors no necessiten conèixer els detalls de com es realitzen les operacions a les capes inferiors. Per exemple, una capa de gestió de processos pot sol · licitar a la capa de gestió de memòria que alliberi recursos, sense saber exactament com es fa aquest alliberament.

Implementació de l'Arquitectura en Capes

Aquesta arquitectura sovint es visualitza com un conjunt d'anells concèntrics, on cada anell representa una capa del sistema operatiu amb diferents nivells de privilegi i accés a recursos.

Els anells més interns (capes inferiors) tenen privilegis més alts, ja que estan més a prop del maquinari i gestionen recursos essencials, mentre que els anells més externs (capes superiors) tenen menys privilegis i estan més aïllats del maquinari.

Els anells es defineixen de la següent manera:

- **Anell 0 (mode kernel)**: Aquesta és la capa més interna, i per tant, la que té més privilegis. Aquest anell té accés complet a tots els recursos del sistema i pot realitzar operacions amb el maquinari de forma directa.
- **Anells 1 i 2**: Són anells de privilegis intermedis, on es poden situar serveis o funcions del sistema operatiu que necessiten cert accés als recursos, però amb menys llibertat que l'anell 0. Aquests anells solen utilitzar-se en funcions com gestors de controladors de dispositius o altres serveis del sistema que requereixen menys privilegis que el nucli. Per exemple, el controlador de disc pot operar en l'anell 1, mentre que els serveis de xarxa poden fer-ho en l'anell 2.
- **Anell 3 (mode usuari)**: Aquesta és la capa més externa i la que té menys privilegis. És l'anell on s'executen els programes d'usuari. Els processos que s'executen en aquest nivell no tenen accés directe als recursos del sistema, sinó que han d'utilitzar crides al sistema per sol · licitar serveis de les capes inferiors (anells 0-2). D'aquesta manera, es protegeix el sistema operatiu de possibles errors o vulnerabilitats dels programes d'usuari.

Quan un procés d'una capa superior (per exemple, un programa d'usuari en l'anell 3) requereix accés a un servei proporcionat per una capa inferior, ha de realitzar una crida al sistema.

El principal avantatge dels anells es que el podem extendre al subsistema de l'usuari. Per exemple, un professor pot escriure un programa per avaluar els estudiants i executar-lo en l'anell **n**, mentre els programes d'estudiants s'executen en l'anell **n+1**. D'aquesta manera, els estudiants no poden modificar les seves notes o consultar les notes dels altres estudiants, ja que treballen en diferents anells.

Exemples d'Ús d'Arquitectura en Capes

- *THE (Technische Hogeschool Eindhoven) System*: Un dels primers exemples d'implementació de l'arquitectura en capes és el sistema operatiu THE, dissenyat per Dijkstra als anys 60. Aquest sistema estava organitzat en cinc capes, on cada capa tenia un conjunt de funcions específiques, com la gestió de processos o l'entrada/sortida.
- *Multics*: Un altre sistema notable que utilitza aquesta arquitectura és Multics (Multiplexed Information and Computing Service), que va ser pioner en l'ús d'anells concèntrics per a la protecció i el control d'accés. Multics va influenciar profundament els sistemes operatius posteriors, com Unix, tot i que Unix simplificà moltes de les idees introduïdes per Multics.
- *UNIX*: Aquest sistema operatiu, creat per Dennis Ritchie i Ken Thompson als anys 70, va adoptar una arquitectura en capes més simple que Multics. Aquestes capes són: Hardware, Kernel, Shell i Aplicacions. Un exemple és: NetBSD.

Resum de l'Arquitectura en Capes

Avantatges	Inconvenients
Millora de la seguretat i l'estabilitat	Rendiment inferior
Facilitat de desenvolupament i manteniment	Rigidesa
Aïllament funcional	

Arquitectura Microkernel

La idea de **l'aquitectura microkernel** és traslladar la major part dels serveis del sistema operatiu fora del nucli, de manera que aquest sigui el més petit i simple possible.

Aquesta arquitectura es basa en el principi de **minimitzar el codi en mode kernel**, traslladant la major part de les funcionalitats a **serveis externs** que s'executen en *mode usuari*. L'objectiu principal és minimitzar el codi en mode kernel per millorar la *seguretat, estabilitat i manteniment*.

La figura mostra un esquema simplificat d'un sistema operatiu basat en arquitectura microkernel, on es pot observar com els serveis del sistema operatiu s'executen en mode usuari, mentre que només les funcionalitats bàsiques es mantenen en el nucli.

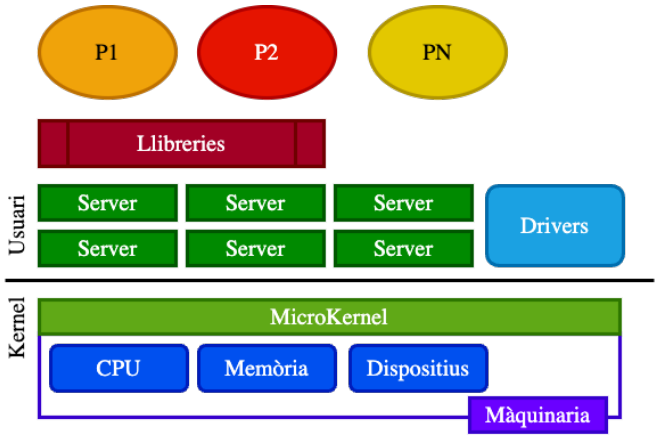

Figura 6: Arquitectura Microkernel

Característiques de l'Arquitectura Microkernel

1. **Fiabiltiat i Modularitat**: Permet que els serveis s'executin com a processos independents en espai d'usuari. Aquesta compartimentació garanteix que un error en un servei no afecti la resta del sistema. Aquest enfocament millora la **fiabilitat del sistema** en comparació amb l'arquitectura monolítica, on tot el sistema operatiu s'executa en mode kernel.

Per exemple, un sistema operatiu monolític amb 5 milions de línies de codi pot contenir entre 10.000 i 50.000 errors (bugs), segons un estudi que estima que hi ha entre 2 i 10 errors per cada 1.000 línies de codi.

En canvi, en un sistema microkernel, gran part del codi que podria contenir errors es trasllada fora del nucli, a l'espai d'usuari, on els errors no comprometen la integritat del sistema en conjunt.

Imagineu un driver d'àudio que falla en un sistema microkernel: l'única conseqüència serà la pèrdua del so, però el sistema operatiu seguirà funcionant amb normalitat. En canvi, en un sistema monolític, un error en aquest mateix driver podria provocar una fallada total del sistema a causa de la seva execució en mode kernel.

2. POLA (Principle of Least Authority): Segons aquest principi, cada procés ha de tenir només els permisos estrictament necessaris per realitzar les seves tasques. Això millora la seguretat, ja que limita l'abast d'acció de cada procés, evitant que pugui accedir a recursos o funcionalitats no necessàries que podrien ser compromeses.

   Per exemple, un driver d'un dispositiu d'entrada/sortida només tindrà accés als ports específics associats al seu dispositiu, i no podrà accedir a altres recursos del sistema ni interactuar directament amb el nucli.

3. **Separació de Mecanismes i Polítiques** El microkernel implementa una separació clara entre els mecanismes i les polítiques.

   Per exemple, el nucli pot tenir el mecanisme per assignar una prioritat a un procés, però la política de com s'assignen les prioritats s'implementa fora del nucli, a l'espai d'usuari. Això permet flexibilitat en la configuració del sistema.

Exemples d'Arquitectura Microkernel

Els sistemes basats en microkernel són menys comuns que els monolítics en sistems d'escriptori, però són més freqüents en sistemes encastats, temps real i en entorns crítics on la seguretat i la fiabilitat són essencials.

- **Mach**: Un dels primers exemples d'implementació de l'arquitectura microkernel és el sistema Mach, desenvolupat a la Universitat Carnegie Mellon als anys 80. Mach va ser dissenyat per ser un nucli molt petit i simple, que proporcionava només les funcionalitats bàsiques del sistema operatiu, com la gestió de processos i la comunicació entre processos. La resta de serveis, com la gestió de memòria o els controladors de dispositius, es van traslladar a **servidors externs** que s'executaven en mode usuari.
- **MacOS**: El sistema operatiu MacOS, desenvolupat per Apple, va aprofitar l'estabilitat i seguretat del nucli **Mach** com a base per al seu sistema operatiu. En aquest sistema, serveis com la gestió de memòria, la gestió de fitxers i la xarxa es van traslladar fora del nucli, a **servidors externs**, millorant la estabilitat, fiabilitat i modularitat del sistema.
- **Symphony OS**: Un sistema operatiu basat en Linux que utilitza una arquitectura microkernel per proporcionar una interfície d'usuari innovadora i una experiència d'ús diferent.
- **QNX**: Sistema operatiu en temps real (RTOS) que utilitza un microkernel per oferir alta fiabilitat i rendiment en entorns crítics, com l'automoció o la indústria aeroespacial. QNX permet la recuperació automàtica de serveis fallits sense interrompre el funcionament del sistema.
- **MINIX 3**: Un sistema operatiu basat en microkernel, dissenyat per ser segur i robust. MINIX 3 utilitza un microkernel d'unes 15.000 línies de codi en C i unes 1.400 en assemblador. El nucli gestiona la planificació de processos, la comunicació entre processos i les interrupcions. La majoria de serveis, com els controladors de dispositius, s'executen en mode usuari, i estan aïllats del nucli. Un aspecte interessant de MINIX 3 és el servei de reencarnació, que monitoritza altres serveis i drivers. Si detecta una fallada en algun component, pot reiniciar-lo automàticament sense interrompre el sistema, augmentant encara més la seva robustesa.

Resum de l'Arquitectura Microkernel

Avantatges	Inconvenients
Seguretat i fiabilitat: Els serveis en mode usuari estan aïllats del nucli, minimitzant l'impacte d'errors.	**Rendiment inferior**: Rendiment inferior: La comunicació entre el mode usuari i el kernel pot ser més lenta a causa del canvi de context.
Modularitat i flexibilitat: Els serveis es poden actualitzar o reiniciar sense interrompre el sistema.	**Complexitat**: La gestió de serveis externs pot ser més complexa que en un sistema monolític.
Política de seguretat granular: POLA permet limitar els privilegis de cada procés.	**Cost d'implementació**: El disseny i la implementació d'un microkernel pot ser més complexa que en un sistema monolític.

Avantatges	Inconvenients
	Latència: La comunicació entre serveis pot introduir latències addicionals.

Altres Arquitectures

A part de les arquitectures monolítica, en capes i microkernel, hi ha altres models arquitectònics que poden ser utilitzats en el disseny de sistemes operatius. Alguns d'aquests models són:

1. **Arquitectura híbrida**: L'arquitectura híbrida combina característiques dels models monolític i microkernel. En aquest tipus de sistemes, es busca un equilibri entre el rendiment d'un nucli monolític i la modularitat d'un microkernel. Per exemple, es poden tenir parts del nucli funcionant en mode kernel amb alta integració (com en els sistemes monolítics) i altres components, com els drivers o serveis de xarxa, que s'executen en mode usuari o com a serveis modulars, com en els microkernels.

   Exemple: *Windows NT*, yé un nucli monolític per a operacions crítiques, però alguns serveis, com els subsistemes d'entrada/sortida o els sistemes de seguretat, s'executen en un entorn més modular, oferint així una barreja de rendiment i seguretat.

2. **Arquitectura de Màquines Virtuals**: el sistema operatiu s'executa en una capa de virtualització que simula el maquinari real. Aquesta capa d'abstracció permet **executar múltiples sistemes operatius simultàniament** en una única màquina física, cada un funcionant com si tingués accés complet al maquinari. Aquest model és ideal per a l'aïllament d'entorns, la portabilitat i la utilització eficient de recursos.

   Exemple: - Hipervisors: *VMware*, *VirtualBox*, *KVM*. - Java Virtual Machine (JVM).

3. **Contenidors**: Aquest model utilitza la virtualització a nivell d'aplicació per aïllar i executar aplicacions en entorns virtuals. En lloc de simular el maquinari complet, els contenidors comparteixen el mateix sistema operatiu i només aïllen les aplicacions i les seves dependències. Cada contenidor funciona de manera independent, però comparteix el nucli amb altres contenidors, cosa que redueix la sobrecàrrega en comparació amb les màquines virtuals tradicionals.

   Exemple: *Docker*, *LXC*.

4. **Exokernels**: Arquitectura extremadament minimalista que intenta proporcionar als programes d'aplicació un control directe sobre el maquinari. A diferència dels microkernels o dels kernels monolítics, els exokernels no intenten abstraure gairebé res del maquinari, sinó que ofereixen primitives de baix nivell per accedir directament als recursos de la màquina. La gestió de recursos, com la memòria i la CPU, queda delegada als programes d'usuari o a les seves llibreries.

   Exemple: Imaginem que particionem una màquina en diferents màquines virutals amb un subconjunt de recursos. Cada màquina virtual té un exokernel que li permet accedir directament als recursos de la màquina física. La idea es reduir el *overhead* del sistema separant la multiprogramació de les funcions del sistema operatiu en l'espai usuari.

5. **UniKernel**: Aquest model es basa en la idea de crear un sistema operatiu que només conté les funcionalitats necessàries per executar una aplicació específica. Els UniKernels són sistemes operatius molt petits i optimitzats per a una sola aplicació, i són populars en entorns de computació en el núvol.

   Exemple: *MirageOS* o *LibOS*.