Lezione 3

Struttura di un sistema operativo

Sistemi operativi

20 marzo 2012

Marco Cesati

System Programming Research Group
Università degli Studi di Roma Tor Vergata

Di cosa parliamo in questa lezione?

È ancora una lezione introduttiva sui sistemi operativi: quali servizi devono offrire e come possono essere strutturati

1. Servizi del SO
2. Interfacce con l'utente
3. Chiamate di sistema
4. Controllo dei processi
5. Gestione dei file
6. Gestione dei dispositivi di I/O
7. Gestione delle informazioni
8. Comunicazioni
9. Progettazione e struttura del SO

Servizi di un SO

Un sistema operativo offre un ambiente in cui eseguire i programmi applicativi e in cui offrire servizi

Le categorie di servizi di un SO:

- Interfaccia con l'utente
- Esecuzione dei programmi
- Operazioni di I/O (Input/Output)
- Gestione del file system
- Comunicazioni
- Gestione degli errori
- Assegnazione delle risorse
- Contabilizzazione delle risorse
- Protezione e sicurezza

Tipologie di interfacce utente

- Interfaccia a lotti
  - comandi codificati insieme ai file da eseguire
  - tipica dei SO dei mainframe
- Riga di comando (CLI, Command Line Interface)
  - shell di comandi
  - tipica dei minicalcolatori e dei microcalcolatori
- Interfaccia grafica (GUI, Graphical User Interface)
  - gestore delle finestre
  - tipica dei sistemi operativi “general-purpose” moderni

Interfaccia a lotti

- L'interfaccia a lotti (*batch*) è una interfaccia utente non interattiva
- L'utente descrive in anticipo tutto il lavoro che deve essere svolto
- Nei sistemi operativi dei mainframe veniva utilizzato un linguaggio apposito detto Job Control Language
- Nei sistemi operativi moderni gli utenti possono descrivere un insieme di operazioni (job) da svolgere in sequenza od in parallelo
  - Le shell di comandi possono leggere i comandi da eseguire da file su disco chiamati script
  - Esistono linguaggi interpretati progettati appositamente: Perl, Python, ...

Interprete dei comandi

- L'interprete dei comandi è una interfaccia utente basata sull'idea di linea comando
  - CLI, Command Line Interface
- Quando l'interfaccia è pronta ad accettare un nuovo comando visualizza sulla console o terminale una stringa di caratteri detta *prompt* (ad esempio, "`$ `")
- In sistemi operativi molto semplici l'interprete dei comandi è integrato nel nucleo del sistema operativo
- Nella maggior parte dei casi l'interprete dei comandi è realizzato da un programma di sistema detto *shell*
  - In MS Windows: *Command shell* (`CMD.EXE`), *PowerShell*
  - In UNIX: *Bourne shell* (`sh`), *Bourne Again shell* (**`bash`**), *Korn shell* (`ksh`), *C shell* (`csh`, `tcsh`), *Z shell* (`zsh`), ...

Esempio:

```
$ date -R
Mon, 19 Mar 2012 14:39:01 +0100
$ _
```

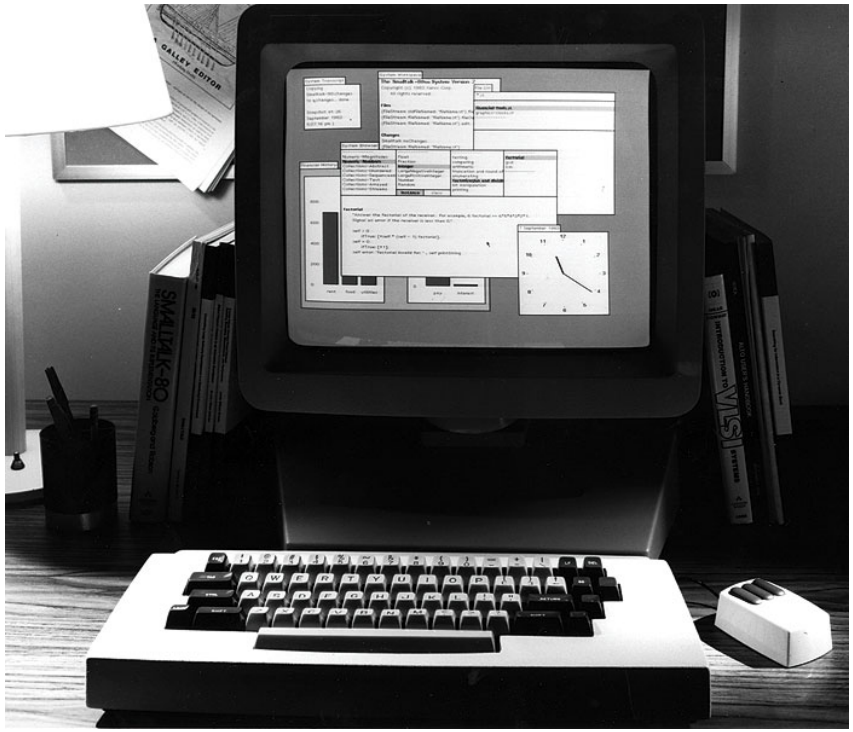

Comandi interni ed esterni

Due diversi tipi di comandi:

- Comandi esterni: file eseguibili su disco contenenti il programma che realizza le operazioni richieste
  - La shell crea un processo per eseguire il comando esterno
  - I file eseguibili sono memorizzati in directory predefinite
  - In UNIX e MS Windows la variabile d'ambiente `PATH` memorizza una lista di directory in cui la shell cerca il comando esterno
  - Esempi in Linux: `ls`, `rm`, `cp`
- Comandi interni: le operazioni richieste sono eseguite direttamente dalla shell senza creare un nuovo processo
  - Esempi in Bash: `cd`, `alias`, `jobs`

Interfaccia grafica

- L'interfaccia grafica consente all'utente di interagire con il sistema tramite pulsanti, icone e menù attivabili con mouse e/o tastiera
- Acronimo inglese: GUI (Graphical User Interface)
- La prima GUI usata nello Xerox Alto (1973)



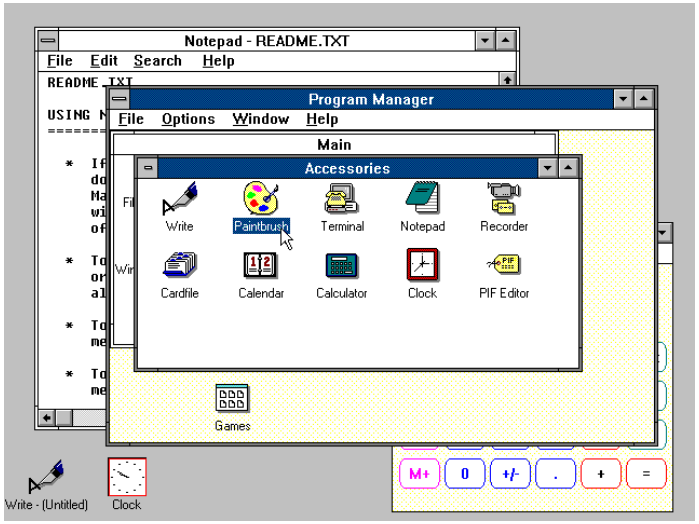
Source: DigiBarn computer museum (cc)

Interfaccia grafica (2)

- 1978–1984: Apple Lisa e Macintosh

Source: http://www.mac-history.net

- 1985–1995: MS Windows v1, v2, e v3



Source: http://www.guidebookgallery.org

Interfaccia grafica (3)

- Nei sistemi MS Windows attuali la GUI è integrata nel SO
- Nei sistemi Apple Mac OS X più recenti la GUI è implementata da
  - Un *window server* chiamato Quartz Compositor
  - Un *window manager* chiamato Aqua
- Nei sistemi Linux la GUI è in genere implementata da
  - il *window server* X Window System "X11" (MIT, 1984)
  - un *window manager* come Gnome o KDE

Tutte le GUI tentano di facilitare l'interazione con l'utente ricreando sullo schermo una scrivania (desktop) virtuale con cartelle (directory), fogli (documenti), cestino, orologio, ...

Chiamate di sistema

Le chiamate di sistema sono la principale interfaccia tra il nucleo del SO ed i programmi applicativi

Le chiamate di sistema costituiscono:

- *Punti di ingresso* (*entry point*) del nucleo del SO
- Un meccanismo per richiedere i servizi del nucleo del SO
- Un meccanismo per assegnare temporaneamente maggiori privilegi ad un processo
  - Nei SO con meccanismi di protezione dei processi, il nucleo esegue in una modalità più privilegiata delle applicazioni:

    Kernel Mode vs User Mode
  - La CPU deve offrire adeguato supporto, ad esempio eccezioni del processore che modificano il livello di privilegio, oppure una istruzione macchina dedicata

Esempio: nelle attuali CPU Intel l'istruzione macchina `sysenter` consente di invocare una procedura ad un indirizzo prefissato in memoria cambiando automaticamente il ring level

Application Programming Interface

Le chiamate di sistema non devono essere confuse con le interfacce per la programmazione di applicazioni (API)

Una API è una procedura che consente ad una applicazione di accedere ad un determinato servizio

- L'API spesso è rappresentata da un prototipo di una funzione di libreria
- Per l'applicazione non importa se il servizio è realizzato
  - da una libreria esterna, oppure
  - dal nucleo, per mezzo di una o più chiamate di sistema
- Spesso le chiamate di sistema possono essere invocate direttamente da una applicazione
  - una libreria di sistema conterrà una piccola funzione chiamata *wrapper* per eseguire la chiamata di sistema gestendo gli argomenti ed i valori restituiti
  - la funzione wrapper della chiamata di sistema costituisce la API del servizio offerto

Esempio d'esecuzione di una API di sistema

- Un programma applicativo invoca l'API `printf()`
- Il codice di `printf()` entro la libreria standard C:
  - prepara la stringa da stampare
  - invoca la funzione *wrapper* della chiamata di sistema `write()` passando come argomento il descrittore di file dello standard output e la stringa da stampare
- La funzione *wrapper* della chiamata di sistema `write()`:
  - salva gli argomenti in registri del processore
  - salva in un registro il numero identificativo della `write()`
  - esegue l'istruzione macchina `sysenter`
- La CPU passa in Kernel Mode e comincia ad eseguire il codice della chiamata di sistema `write()`
- A lavoro finito, terminano nell'ordine
  - la chiamata di sistema `write()` (CPU → User Mode)
  - la funzione *wrapper* di `write()`
  - la funzione di libreria `printf()`

Sistema di supporto all'esecuzione

L'insieme delle funzioni wrapper delle chiamate di sistema fa parte del sistema di supporto all'esecuzione dei programmi

Ad esempio, in un sistema Linux il sistema di supporto all'esecuzione è contenuto in alcune librerie dinamiche:

- `linux-vdso.so`: libreria "virtuale" associata ad ogni processo attivo: contiene le istruzioni che invocano la chiamata di sistema (es., `sysenter`)
- `ld-linux-x86-64.so`: libreria contenente codice utilizzato durante la creazione del processo e per risolvere i simboli definiti entro librerie dinamiche
- `libc.so`: libreria standard C, contenente le funzioni wrapper della maggior parte delle chiamate di sistema

Differenza tra libreria statica e dinamica

- Statica: codice copiato nel file eseguibile in fase di linking
- Dinamica: codice inserito nella memoria del processo in fase di esecuzione

Categorie delle chiamate di sistema

1. Controllo dei processi
2. Gestione dei file
3. Gestione dei dispositivi
4. Gestione delle informazioni
5. Comunicazioni

Controllo dei processi

Nei sistemi multiprogrammati devono esistere chiamate di sistema che consentono di gestire e manipolare i processi

Alcuni esempi:

- Terminare un processo attivo
  - POSIX: `exit()`
  - MS Windows: `TerminateProcess`
- Creare un nuovo processo
  - POSIX: `fork()`
  - MS Windows: `CreateProcess()`
- Coordinarsi con altri processi
  - POSIX: `wait()`
  - MS Windows: `WaitForSingleObject()`
- Eseguire un nuovo programma
  - POSIX: `execve()`
  - MS Windows: `CreateProcess()`

Livello d'errore

Il livello d'errore rappresenta un valore numerico associato ad un processo appena terminato e che viene propagato al processo "genitore"

- È un concetto che nasce con i sistemi a lotti: più era grave l'errore, più era grande il valore numerico
- La terminazione normale (successo) è rappresentata dal valore zero (POSIX: macro `EXIT_SUCCESS`)
- Valori maggiori di zero sono spesso associati a codici d'errore specifici
- Per ottenere il livello d'errore
  - di un comando lanciato dalla shell Bash: variabile `$?`
  - di un comando lanciato dalla shell di comandi Windows: costrutto `ERRORLEVEL` o variabile `%ERRORLEVEL%`

Gestione dei file

Tutti i SO offrono chiamate di sistema per la gestione dei file

Alcuni esempi:

- Creazione e cancellazione di file e directory
- Apertura, lettura, scrittura, chiusura di file
- Modifica dei meta-dati associati ad un file
  - Nome del file
  - Utente e gruppo proprietari del file
  - Permessi d'accesso
  - Data e ora di ultimo accesso o modifica

I sistemi di tipo UNIX possiedono diversi tipi di file:

- File regolari e directory
- *FIFO* o *named pipe* (comunicazione tra processi)
- *Socket* (comunicazione tra processi eventualmente remoti)
- *Link simbolici*
- *Device file* a caratteri e a blocchi
- *File virtuali* (interazione con il nucleo)

Gestione dei dispositivi di I/O

Tutti i SO devono offrire chiamate di sistema che consentano ai processi di interagire con i dispositivi di I/O

In genere si consente ai programmi applicativi di utilizzare le stesse API per la gestione dei file

- Nei SO di tipo UNIX l'integrazione tra dispositivi di I/O e file è effettuata a livello di chiamate di sistema, ossia direttamente dal nucleo
- In altri SO (es., MS Windows) l'integrazione è effettuata a livello di libreria di sistema oppure di interfaccia utente
- In SO molto semplici (es., MS-DOS) non c'è alcuna integrazione: l'applicazione deve utilizzare meccanismi specifici di ciascun dispositivo di I/O

Gestione delle informazioni

I SO devono offrire chiamate di sistema che consentano di ottenere informazioni dal sistema

Alcuni esempi:

- Ottenere data e ora di sistema
- Ottenere informazioni sullo stato d'esecuzione dei processi (ad es., per il *debugging* dei programmi)
- Ottenere informazioni sulle prestazioni dei processi e del sistema (*profiling*)
- Ottenere informazioni sullo stato dei file system (spazio disco occupato, stato di frammentazione dei file, data dell'ultimo controllo del file system effettuato)

Comunicazioni

I processi debbono in generale comunicare per cooperare

Due modelli principali di comunicazione:

- A scambio di messaggi: i processi spediscono e ricevono, con l'aiuto dei servizi del nucleo, pacchetti di dati
  - Si può applicare anche a processi su calcolatori interconnessi
  - È facile avere meccanismi di protezione tra i processi
  - I messaggi sono un meccanismo di sincronizzazione
- A memoria codivisa: i processi scrivono e leggono dati da una zona di memoria fisica condivisa tra loro
  - È più efficiente (non c'è intervento diretto del nucleo)
  - La sincronizzazione tra i processi non è banale

Programmazione multi-thread

Caso particolare del modello a memoria condivisa: i processi possono condividere gran parte del proprio spazio di memoria, compreso il codice e le strutture di dati globali

Progettazione di un sistema operativo

Quali sono i criteri di progetto di un buon sistema operativo?

- *. . . deve essere sicuro e facile da usare. . .* (un utente)
- *. . . deve essere flessibile, affidabile ed efficiente. . .* (un sistemista)
- *. . . deve consentire di scrivere applicazioni in modo rapido e senza errori. . .* (un programmatore)

Progettare un SO è un compito difficile perché i suoi requisiti sono spesso soggettivi e non quantificabili

Non esiste una unica soluzione, e non esiste il SO perfetto!

Meccanismi e criteri

Un buon sistema operativo distingue tra meccanismi e criteri

- I meccanismi determinano *come* compiere qualcosa
- I criteri (policy) determinano *cosa* si deve compiere

La mancata distinzione tra meccanismi e criteri porta alla mancanza di flessibilità: l'utente non può decidere *cosa* fare

Se un SO implementa meccanismi ma non criteri, l'utente ha la possibilità di determinare da sé *cosa* deve essere fatto

Ad esempio:

- Linux consente all'utente di scegliere il window manager preferito, quindi di cambiare lo stile della GUI
  - Linux implementa i meccanismi della GUI, non i suoi criteri
- Windows XP non consente all'utente di modificare lo stile della GUI
  - Windows XP implementa meccanismi **e** criteri della GUI

Realizzazione del sistema operativo

La realizzazione di un sistema operativo è analoga a quella di ogni altro programma software: deve essere scritto il relativo programma

Quali linguaggi di programmazione si utilizzano?

- I primi SO erano scritti in codice macchina e/o in linguaggio assembly
- 1961: MCP (Master Control Program) per i calcolatori Burroughs: scritto in ESPOL, derivato dall'ALGOL
- 1964: MULTICS, scritto in PL/1
- 1969: UNIX, scritto in linguaggio C
- Oggi: Linux, MS Windows, e molti altri SO sono scritti per la maggior parte in linguaggio C

Struttura dei sistemi operativi

Il nucleo di un sistema operativo è un programma lungo e complesso

Per tentare di semplificare il processo di sviluppo dei SO si è cercato di rendere la loro struttura ordinata e gerarchica

Con il metodo stratificato il codice di un SO viene suddiviso in strati, dal più basso vicino all'hardware fino all'interfaccia utente

Ciascuno strato utilizza funzioni e servizi dello strato immediatamente sottostante, al fine di offrire servizi allo strato sovrastante

Il numero di strati, in pratica, non può essere troppo elevato, perché

- i meccanismi di comunicazione tra uno strato e l'altro introducono overhead non trascurabili
- può essere difficile od impossibile determinare la posizione esatta nella gerarchia di una certa funzionalità

Microkernel

I sistemi operativi tradizionali sono detti *monolitici*, in quanto il loro nucleo implementa tutti i servizi offerti

Diversi SO più recenti sono basati invece su un modello detto a *microkernel*

Il microkernel è un piccolo programma che realizza pochi servizi essenziali:

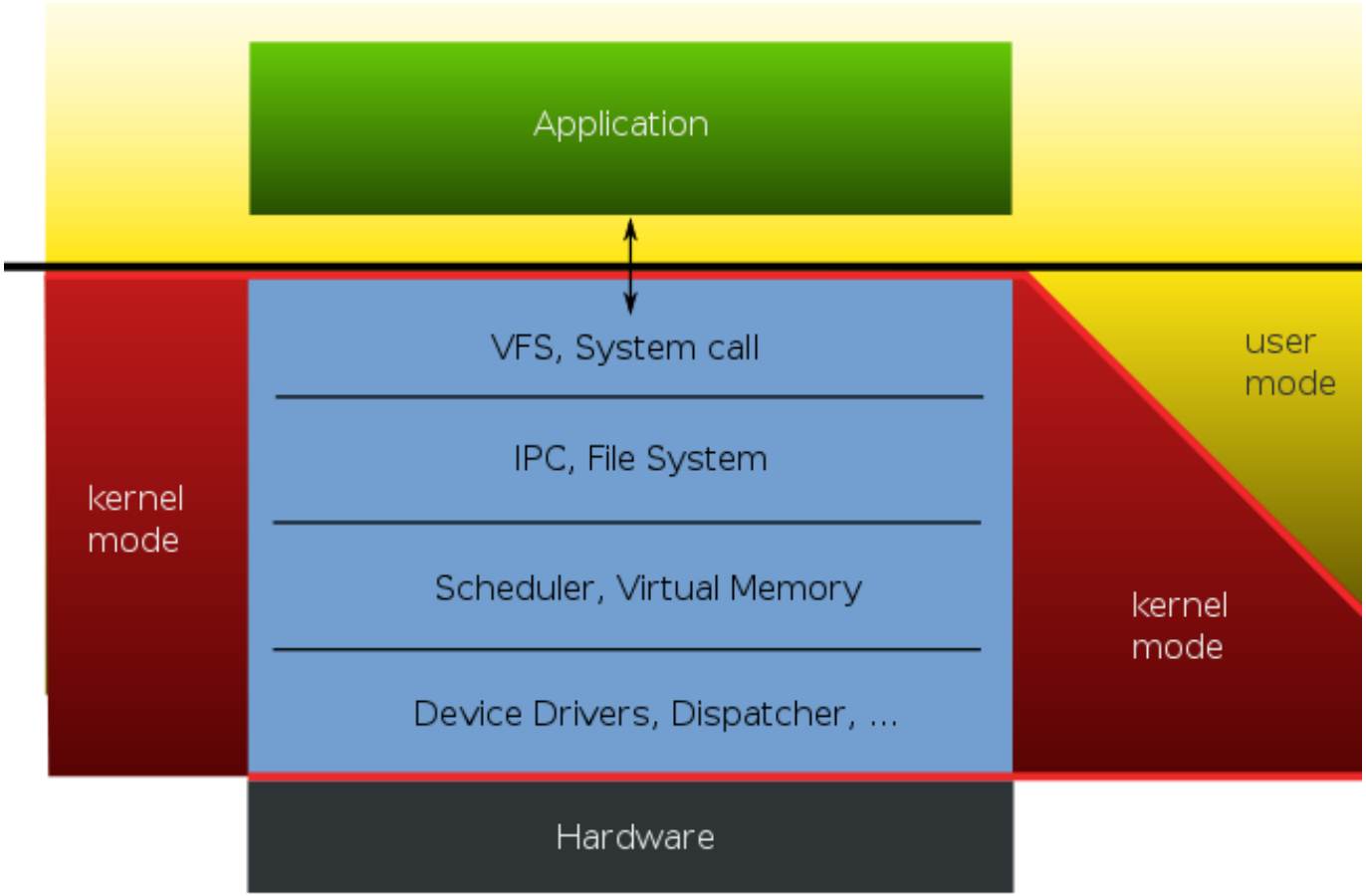
- driver dei circuiti H/W di base
- schedulazione dei processi
- comunicazione di base tra processi

Tutti gli altri servizi offerti alle applicazioni (driver delle periferiche, stack di rete, file system, . . .) sono realizzati da applicazioni di sistema

Quali vantaggi e svantaggi offre l'approccio microkernel?

- Nucleo di ridotte dimensioni, verificabile e mantenibile
- Le interfacce tra i vari componenti sono ben definite
- Lo scambio di messaggi introduce overhead significativi

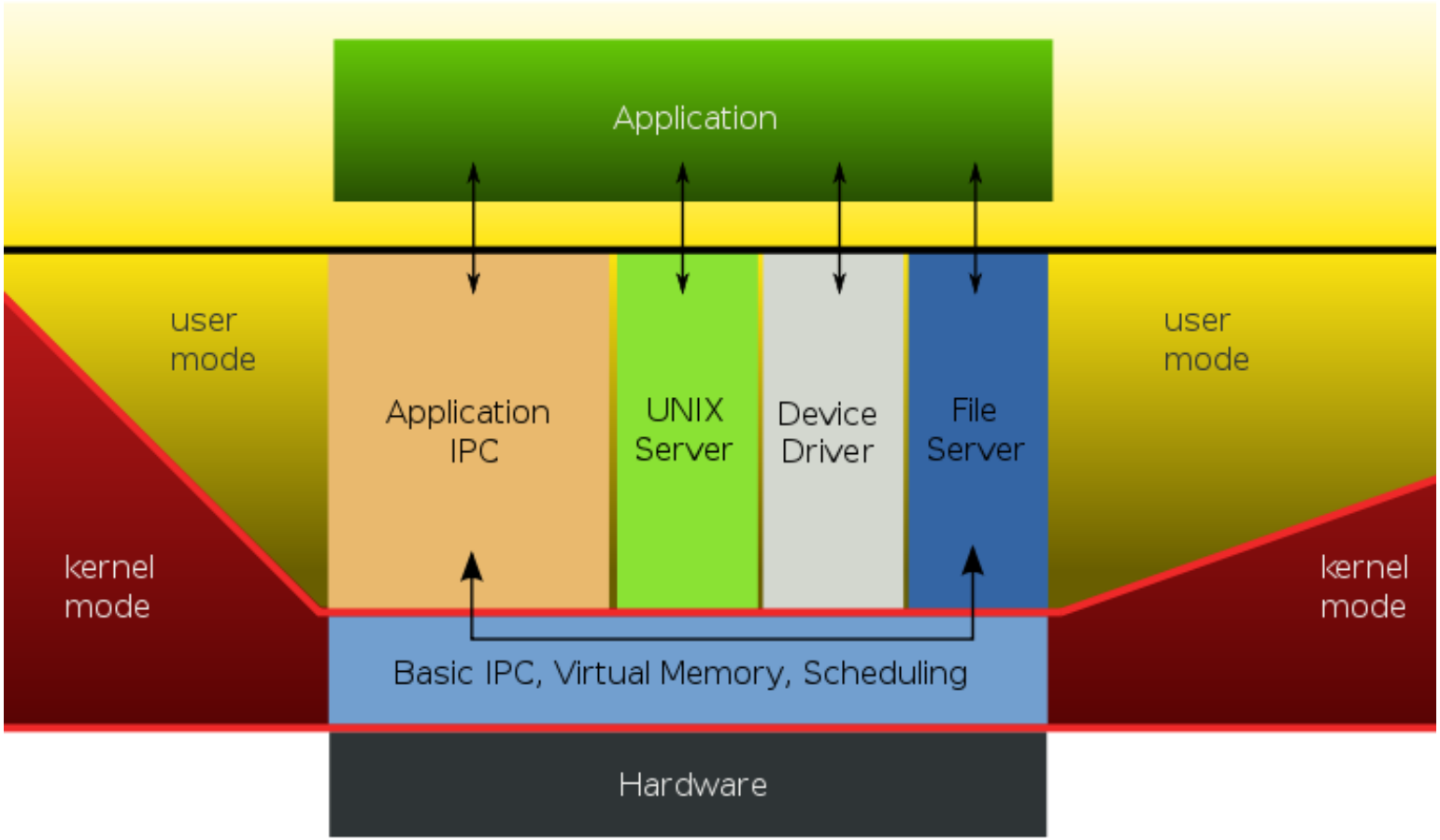
Struttura di un SO monolitico



Fonte: Golftheman – public domain

Esempi: Linux, FreeBSD, SunOS Solaris, . . .

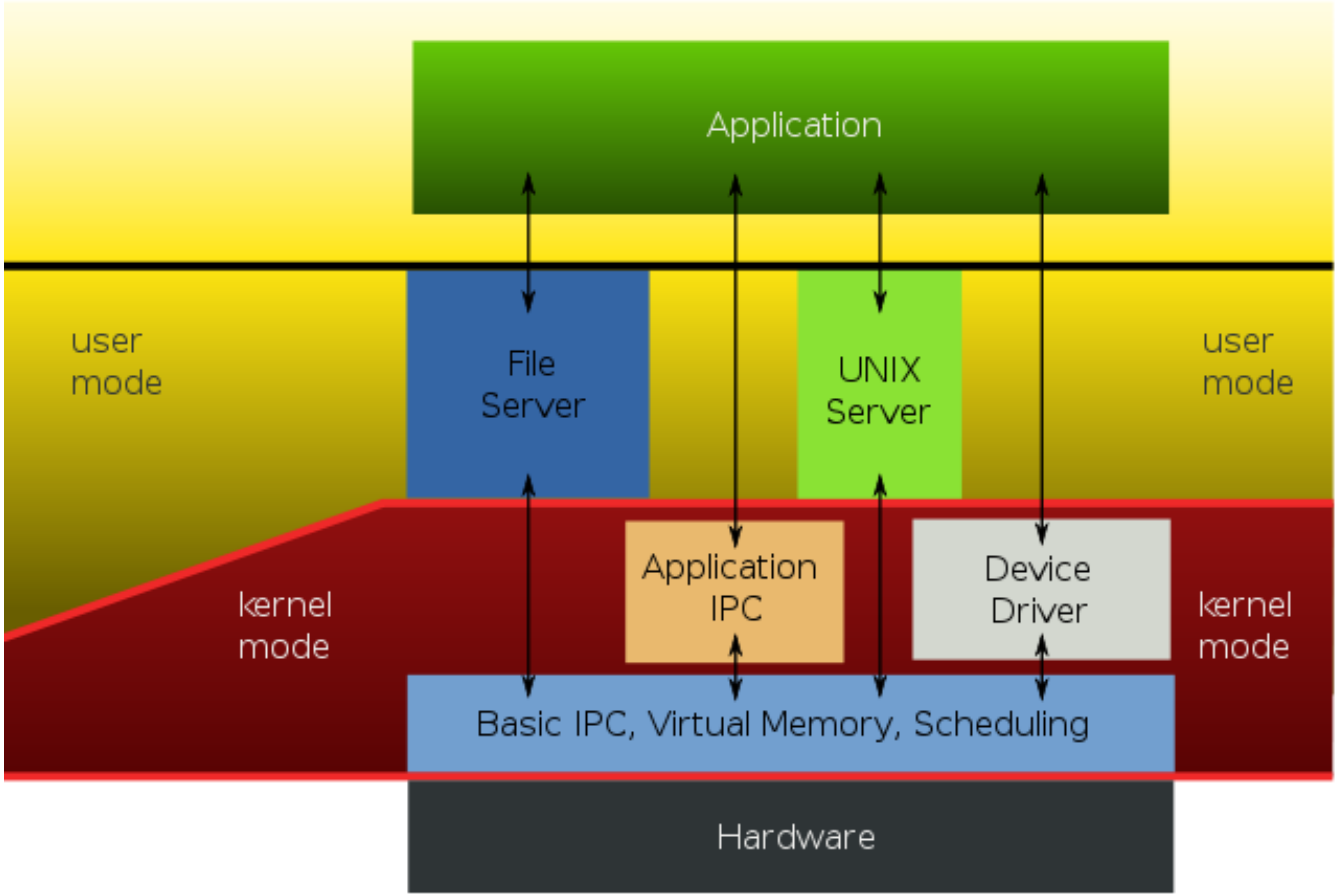
Struttura di un SO a microkernel



Fonte: Golftheman – public domain

Esempi: QNX, GNU Hurd (Mach & Ell Four), BeOS, . . .

Struttura di un SO ibrido



Fonte: Golftheman – public domain

Esempi: Windows NT, Mac OS X (XNU), . . .

Moduli

La struttura interna di Linux, Solaris o Mac OS X è modulare

- Un ridotto numero di componenti del nucleo offre un insieme di servizi di base
- Ulteriori servizi (driver di periferiche, file system, ecc.) sono implementati da codice memorizzato in moduli che possono essere caricati in memoria secondo necessità
- La modularità dei componenti del nucleo facilita la definizione di interfacce chiare e stabili
- A differenza del metodo stratificato, ciascun modulo può utilizzare servizi di qualunque altro modulo
- La comunicazione tra i vari moduli avviene direttamente, quindi non si devono pagare i costi dello scambio di messaggi