

- [Curs 1 - Introducere](#)
- [Curs 2 - Sistemul de fisiere - interfata de user space](#)
- [Curs 3 - Procese](#)
- [Curs 4 - Planificarea executiei. IPC](#)
- [Curs 5 - Gestiunea memoriei](#)
- [Curd 6 - Memoria virtuala](#)
- [Curs 7 - Securitatea memoriei](#)
- [Curs 8 - Fire de executie](#)
- [Curs 9 - Sincronizare](#)
- [Curs 10 - Dispozitive de intrare/iesire](#)

Curs 1 - Introducere

Cuprins

- De ce (un curs) de SO?
- Despre cursul de SO
- Despre sisteme de operare
- Sisteme de calcul. Hardware
- Concepte importante în SO
- Structura unui SO

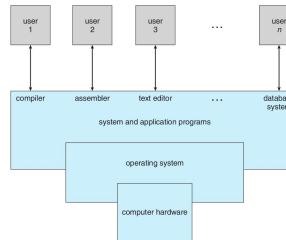
Sisteme de operare

- Gestionează resursele sistemului (hardware)
- Interfață dintre aplicații și hardware
- Mediarea accesului la hardware
- Securitatea și integritatea sistemului

De ce sisteme de operare?

- Independent de job / limbaj de programare/ tip de aplicație: Depanare, Performanță, Securitate
- SO oferă primitive care izolează aplicațiile de detaliile hardware
 - POSIX
 - Procese, fisiere, thread-uri, memorie virtuală.

Ce este un SO?



- un set de programe
- vedere top-down: extensie a mașinii fizice
- vedere bokom-up: gestionar al resurselor fizice
- scris în general în C
- relativ transparent utilizatorului ("trebuie să meargă")

Legătura SO - hardware

- SO este primul nivel software peste hardware
- Un SO performant folosește facilitățile hardware și expune facilitățile hardware aplicațiilor
- Hardware-ul are nevoie de SO pentru a putea fi folosit (pentru a construi aplicații/software)
- SO evoluează pe măsură ce evoluează hardware-ul

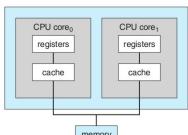
Sistem de calcul

- Sistemul fizic (hardware) + sisteme de operare și aplicații
- Baza pentru a dezvolta aplicații, a folosi aplicații, a construi sisteme
- În general interconectat cu alte sisteme
- De la servere cu mii de core-uri la dispozitive mici din zona IoT (Internet of Things)

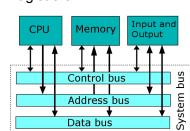
Hardware-ul unui sistem de calcul

- Procesor (CPU) – rulează codul (instrucțiunile) proceselor
- Memorie de lucru (RAM) – stochează datele și codul proceselor
- Magistrale – leagă CPU, module de memorie, dispozitive de I/O
- Dispozitive periferice (de intrare/iesire, I/E, I/O) – comunicarea cu exteriorul: utilizator, alte sisteme de calcul, alte dispozitive
- Spațiu de stocare (disc-uri, flash, ROM, NVRAM)
 - Programe (din care vor lua naștere procese)
 - Date pentru procese
 - Informații pentru utilizator (fisiere)

Procesor și memorie



Magistrale



Memorie

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 64 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS SRAM	on-chip or off-chip, CMOS DRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000,000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

CONCEPTE IMPORTANTE ÎN SO

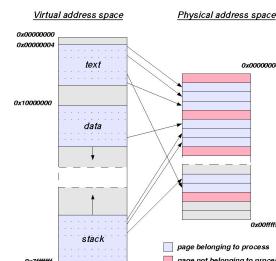
Shell

- Interfață utilizator-sistem de operare
- CLI sau GUI
- Bash vs. Windows Explorer / GNOME / KDE
- Un proces care permite pornirea de alte procente/aplicații – "shell spawns a process"

Procese

- Iau naștere dintr-un program executabil
- Program în execuție, entitate dinamică – Noțiunea de "runtime" (timpul rulării) se referă la proces
- Date și cod în memorie, se execută pe CPU
- Are asociate resurse: spațiu de adrese de memorie, fisiere deschise, socketi
- SO oferă protecție și comunicare inter-proces
- Ierarhie de procese la nivelul SO

Memoria unui proces



Memorie virtuală

- Un proces are un spațiu virtual (inexistent de fapt) de memorie – Procesul are impresia că toată memoria îl aparține
- Memoria virtuală decuplează vederea procesului de memoria sistemului
- Procesele lucrează cu adrese virtuale (din spațiu fizic de adrese) – Adresele fizice sunt adrese din memoria fizică
- SO mapează/ascociază spațiul virtual al proceselor cu memoria fizică

Thread-uri

- Un proces poate avea mai multe thread-uri
- Threadurile partajează spațiul virtual de adrese al procesului
- Utilă să faci mai multe lucruri cu aceeași date (din spațiu virtual de adrese al procesului)
- Permite folosirea facilităților hardware multiprocesor

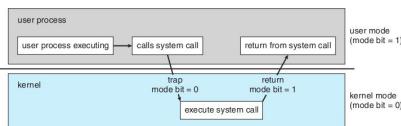
Concurență și sincronizare

- Threadurile sau procesele pot concura la achiziția de resurse (date în memorie)
- Accesele concurente pot duce la date inconsecvențe sau blocări
- Sincronizarea garantează accesul consecvent și ordonat la date

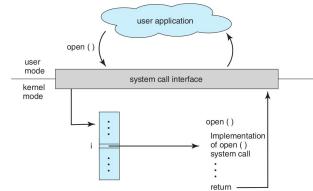
Moduri de execuție

- Procesorul are un mod privilegiat: supervisor mode, system mode, kernel mode
 - Pentru acțiuni privilegiate
 - Aici rulează sistemul de operare
- Aplicațiile rulează în modul neprivilegiat: user mode
 - Nu pot comunica cu hardware-ul sau cu alte procese
 - Pot doar acționa asupra spațiului de memorie propriu
- Nucleul intermediază accesul proceselor la hardware sau la resursele altor procese
- Separarea modurilor asigură securitatea sistemului
- Tranzitia user mode -> kernel mode = apel de sistem

Tranzitia user mode – kernel mode



Apeluri de sistem

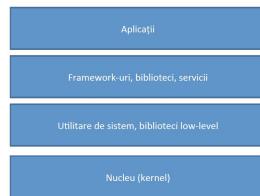


Fisiere

- Unități de stocare
- Deschise și folosite de procese
- Fișier pe disc (static) și fișier deschis (dinamic, în cadrul unui proces)
- Fișier pe disc: nume, dimensiune, permisiuni, timestamp-uri
- Fișier deschis: handle de fișier, cursor de fișier, drepturi de deschidere, operații pe fișier

STRUCTURA UNUI SO

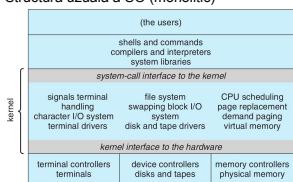
Stiva software pentru un sistem de calcul



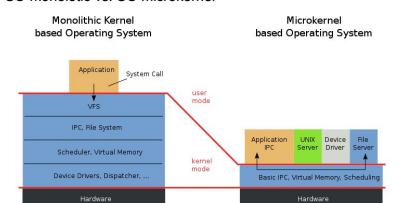
Nucleul (kernel-ului)

- Strict tehnic, nucleul este sistemul de operare
 - Windows are ca nucleu WindowsNT din imaginea ntoskrnl.exe
 - Linux este nucleul unei distribuții GNU/Linux și Android
 - Mac OS X și iOS au ca nucleu XNU
- Încărca la bootare apoi pomenesc primele aplicații și gestionează hardware-ul
- Răspunde apelurilor de sistem ale proceselor
- Gestionează resursele hardware
- Garantează integritatea sistemului

Structura uzuială a SO (monolitic)

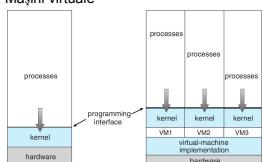


SO monolitic vs. SO microkernel



Monolithic	Microkernel
<ul style="list-style-type: none"> • Eficient • Coeziunea codului/datelor • Mai puțin flexibil • TCB (Trusted Computing Base) mai mare (design mai puțin sigur) 	<ul style="list-style-type: none"> • Mai lent (comunicare între servicii) • Componentizabil, flexibil, modular • TCB redus (design mai sigur)

Mașini virtuale



Tendințe curente în SO

- Securitate
- Dispozitive de mici dimensiuni (tinification)
- Scalare (CPU, memorie, disc), mașini virtuale
- Performanță, suport hardware pentru operații specifice

Curs 2 - Sistemul de fisiere - interfața de user space

Cuprins:

- Sisteme de fisiere
- Fisiere. Atribute ale fisierelor
- Operatiuni cu fisiere
- API pentru lucru cu fisiere
- Directoare

SISTEMUL DE FISIERE

Ce este un sistem de fisiere?

- Mod de organizare persistenta a informatiei
- Suport pe disc
- Utilizator: ierarhie de fisiere si directoare
- SO: algoritmi si structuri de date
- Performanta, scalabilitate, usoritate in utilizare

De ce sisteme de fisiere?

- Memoria este volatila, de dimensiune mica
- Suport persistent
- Structurat, organizat

Exemple de sisteme de fisiere: NTFS, ext4, FAT32, ISO9660, HFS+

Operatiuni cu SF

- Formatare
- Verificarea consistentei
- Montare
- Demontare
- Tuning (optiuni specifice)

FISIERE. ATRIBUTE ALE FISIERELOR

Ce este un fisier?

- Unitatea de baza in sistemul de fisiere
- Abstractizeaza informatica, datele

- Informatia structurata
- Identificat de utilizator prin nume
- Identificat in SF prin un numar (in Unix - inode number)

Tipuri de fisiere

- Fisiere obisnuite (regular files)
- Directoare
- Link-uri
- Dispozitive de tip caracter
- Dispozitive de tip bloc
- FIFO-uri
- Socketi Unix

Pentru a determina tipul unui fisier in Unix:

- Ls - I
- File
- Stat

Fisiere obisnuite

- De obicei se numesc doar fisiere
- Byte stream - flux de octeti (se citeste si se scrie octet cu octet)
- Organizarea datelor tine de tipul fisierului si de procesul care il foloseste

Directoare

- Colectii de alte fisiere
- Organizate (un vector de alte intrari)

Atribute ale fisierelor (comanda stat)

- Nume, identificator pe disc, dimensiune, drepturi de acces, owner, timpi de acces

Fisiere si fisiere deschise

- Fisierile sunt folosite de procese, iar in momentul folosirii, un fisier este "deschis"
- Un fisier pe disc este static si referit prin nume
- Un fisier deschis este dinamic si referit prin handle

Procese si fisiere

- Fisierile sunt deschise de procese pentru a fi folosite
- Fisierile deschise sunt gestionate prin handle
- Un handle este o referinta la un fisier deschis
- Un proces poate deschide mai multe fisiere
- Un fisier poate fi deschis de mai multe procese (e necesara sincronizarea)
- Un proces are o tabela de handle-uri: sunt retinute referintele la toate fisierele deschise

Cursorul de fisier: file cursor, file pointer, file offset

- Pozitia curenta pentru citire/scriere de un proces
- Incrementata la citire si scriere
- Poate fi repositionat

Procese, fisiere, handle-uri

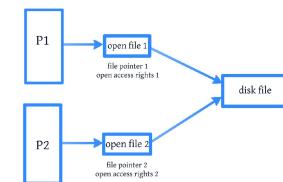
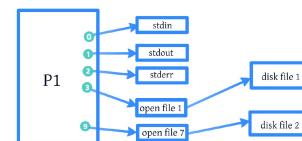


Tabela de descriptori (file descriptor table)

- File descriptor (Unix) = handle



Deschiderea unui fisier (open & close)

- Realizata de un proces
- Intrare: nume de fisier, drepturi de deschidere
- Iesire: handle de fisier
- Se aloca o structura de fisier deschis
- La inchiderea fisierului, se invalideaza handle-ul, se dezaloca structura de fisier deschis

Citire si scriere (read & write)

- Pe un fisier deschis, pe un handle
- La intrare: handle, cat se scrie/citeste, unde/de unde
- La iesire: cat s-a scris/citit
- Dupa citire, scriere, se avanseaza cursorul de fisier

Posizionare (seek)

- Se comanda plasarea cursorului de fisier
- Poate fi dat inapoi (rewind)
- Input: handle, cu cat se muta, in raport cu ce
- Output: noua pozitie

API DE LUCRU CU FISIERE

Shell

- Touch file.txt
- Cat file.txt
- Echo "abc" > file.txt
- Rm file.txt

ANSI C

- FILE *f = fopen("file.txt", "r+")
- N = fread(f, sizeof(char), 100, buffer)
- M = fwrite(f, sizeof(char), 200, buffer2)
- fseek(f, 100, SEEK_SET)
- fclose(f)

Unix(POSIX)

- Int fd = open("file.txt", O_RDWR)
- N = read(fd, buffer, 100)
- M = write(fd, buffer2, 200)
- lseek(fd, 100, SEEK_SET)
- close(fd)

Windows(Win32)

- HANDLE f = CreateFile("file.txt", GENERIC_WRITE, ..., OPEN_EXISTING, ...)
- Ret = ReadFile(f, buffer, 100, &bytesRead, NULL)
- SetFilePointer(f, 100, NULL, FILE_BEGIN)
- CloseHandle(f)

Java

- FileReader fr = new FileReader("file.txt")
- fr.read(buffer)
- fr.close()
- FileWriter fw = new FileWriter("file.txt")
- fw.write(buffer)
- fw.close()
- FileChannel fc = FileChannel.open("file.txt", READ, WRITE)
- fc.position(100)
- fc.close()

Python

- F = open("file.txt", "r+")
- Str = f.read(10)
- f.write(buffer)
- f.seek(0,0) #place at beginning
- f.close()

REDIRECTARI

Ce inseamna redirectare?

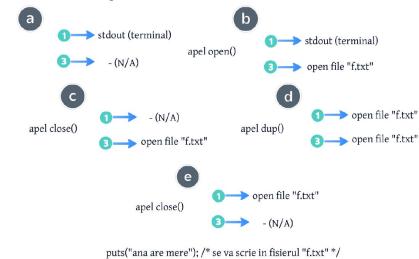
- Iesirea standard este transmisa intr-un fisier: echo "ana" > file.txt
- intrarea standard este transmisa dintr-un fisier: grep "ana" < /etc/passwd
- Este transmisa intr-un/dintr-un pipe: grep "ana" /etc/passwd | wc -l

De ce redirectare?

- Pentru a retine iesirea unei comenzi
- Pentru a transmite intrarea unei comenzi dintr-un fisier
- Pentru comunicare inter-proces(pipe, filtre
- Pentru a anula iesirea unei comenzi (/dev/null)

Implementare redirectare (echo "ana" > f.txt)

- Fd = open("f.txt", O_WRONLY)
- Close(STDOUT_FILENO)
- dup(fd)
- close(fd)
- Dup dupica un descriptor in primul descriptor liber, iar primul descriptor liber de mai sus e STDOUT_FILENO



BUFFERED I/O vs SYSTEM I/O

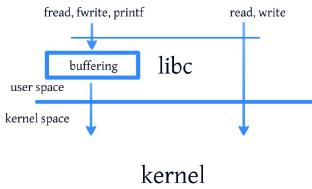
Buffered I/O (ANSI C I/O)

- Biblioteca standard C face buffering la informatii
- Nu sunt transmise instant catre sistemul de operare
- printf, fprintf, puts, fputs, fwrite
- Sunt transmise sistemului de operare: la fflush(), la fclose(), line buffered (standard input), fully buffered(fisiere): cand s-a umplut bufferul

System I/O (low-level I/O)

- Datele sunt transmise de la/catre SO
- Nu exista buffering
- Apelul se intorce cand o parte din date au fost transmise
- Read, write

Buffered I/O & System I/O



Buffered I/O	System I/O
<ul style="list-style-type: none"> - Buffered - Latenta in propagare - Memorie ocupata - Mai putine apeluri de sistem 	<ul style="list-style-type: none"> - Sincron - Transmisie directa - Doar bufferele programului - Overhead de apel de sistem

Curs 3 - Procese

Cuprins

- Rolul proceselor
- Atributele unui proces
- Planificarea proceselor
- Crearea unui proces
- Alte operatii cu procese
- API pentru lucrul cu procese

Ce este un proces?

- Modul în care se execută acțiuni în sistemul de operare
- Un program aflat în execuție
- Încapsularea/abstractizarea execuției în SO
- Abstractizare peste procesor

De ce procese?

- Ca să putem efectua acțiuni în SO
- Ca să avem o separare a acțiunilor: un proces face ceva, altul face altceva
- Ca să avem izolare: un proces nu încurcă alt proces
- Pentru modularizare: când ceva pică sau nu merge, să putem determina rapid problema

ROLUL PROCESELOR

Ce face un proces?

- Rulează instrucțiuni pe procesor – Instrucțiunile sunt în memoria procesului (cod)
- Lucrează cu informații de la input/output – Fișiere, retea, tastatură, monitor
- Interacționează cu alte procese – Comunicare, partajare resurse

Ce conține un proces?

- Memorie: cod, date
- Tabelă de descriptori de fișier
- Legături către alte procese (proces părinte, procese copil)

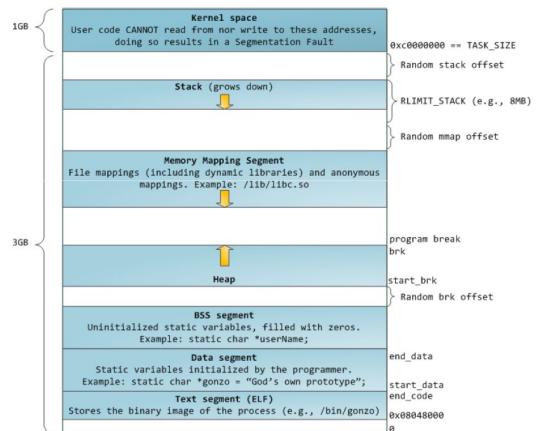
Proces și procesor

- Un proces rulează instrucțiuni pe procesor
- Procesele sistemului au nevoie de procesoare
- De obicei procesele sistemului sunt mai multe decât procesoarele sistemului – E nevoie de planificarea proceselor pe procesoare

Proces și memorie

- Un proces are o memorie proprie (izolată de alte procese)
- Cod/instrucțiuni și date

Spațiul (virtual) de adrese



- Fiecare proces are un spațiu (virtual) de adrese
- Asigură izolare față de alte procese
- Procesul are impresia folosirii exclusive a memoriei
- Toate resursele de memorie (cod, date, biblioteci, stivă) sunt referite prin zone în spațiu virtual de adrese

Tabela de descriptori

- Proprie fiecărui proces
- Interfață pentru I/O a unui proces
- Vectori de pointeri către structuri de fișiere deschise
- Structurile pot referi fișiere, socketi, dispozitive speciale (terminale)

Atribute ale unui proces

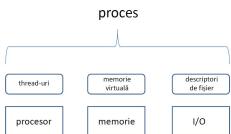
- PID
- parent PID
- pointeri către resurse
- stare (de rulare, așteptare)

- Instrucțiunile sunt aduse din memoria RAM în procesor și executate – Spunem că procesul se execută pe procesor

Proces și I/O

- Un proces comunică cu exteriorul: disc, retea, tastatură, monitor
- Comunicarea se face, de regulă, prin descriptori de fișier
- Un proces are o tabelă de descriptori de fișier
- Un descriptor de fișier referă un fișier, socket, terminal, dispozitiv etc.
- Operațiile cu I/O blochează procesul – I/O este mai lent decât procesorul – procesul așteaptă încheierea operației

Procesor, memorie, I/O



Tipuri de procese

- CPU bound (CPU intensive) – rulează des pe procesor
- I/O bound (I/O intensive)
 - rulează rar pe procesor
 - fac operații de I/O -> se blochează
- Interactive (foreground) – interacționează cu utilizatorul
- Noninteractive (batch, background) – servicii, daemoni

ATRIBUTELE UNUI PROCES

Cum arată un proces la nivelul SO?

- O structură de date
 - PCB (Process Control Block)
 - Descrie un proces la nivelul SO
- Informații legate de resursele folosite
- Un identificator (PID: process identifier)
- Legături cu celelalte structuri
- Informații de securitate, monitorizare, contabilizare

Resursele unui proces

- Timp de rulare pe procesor
- Memorie (cod și date)
- Tabelă de descriptori de fișier
- Unele resurse pot fi partajate cu alte procese

- cantă de timp de rulare
- contabilizare resurse consumate
- utilizator, grup

PLANIFICAREA PROCESELOR

Procese și procesoare

- Pentru a rula un proces are nevoie de un procesor
- Procesorul rulează instrucțiunile procesului
- Procesele într-un sistem sunt mai multe ca procesoarele
- Sistemul de operare asigură accesul echilibrat al proceselor la procesoare

Multitasking

• SO schimbă rapid procesele pe procesoare

- După un timp (cantă, time slice) un proces este scos de pe procesor și pus altul în loc – se spune că "expiră cantă" – acțiunea este numită "schimbare de context" (context switch)
- Cantă de timp de ordinul milisecundelor – se schimbă foarte rapid procesele – impresia de rulare simultană
- Un proces poate fi întrerupt în momentul executării unei secvențe

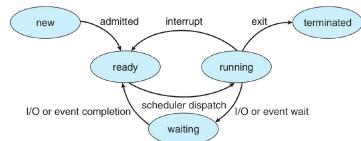
Starea unui proces

- rulare (RUNNING) – procesul este pe un procesor și rulează
- așteptare (WAITING) – procesul a executat o acțiune blocantă (de exemplu citire I/O) și așteaptă sosirea datelor; nu poate rula
- gata de execuție (READY) – procesul poate să ruleze pe procesor – pe moment nu are loc pe procesor, e alt proces acolo

Planificarea unui proces (scheduling)

- Un proces este adus din starea READY în starea RUNNING
- Este adus dacă există un procesor liber
- Un procesor este liber dacă procesul de pe procesor l-a eliberat
- Se spune că are loc o schimbare de context (context switch)

Tranzitii între stările unui proces



Ierarhia de procese

- Un proces poate crea unul sau mai multe procese copil
- Un proces poate avea un singur proces părinte
- În Unix, procesul init este în vîrful ierarhiei de procese
 - init este creat de sistemul de operare la boot
 - init creează apoi procesele de startup
 - procesele de startup creează alte procese etc

Procese și executabile

- Unul sau mai multe procese îau naștere dintr-un fișier executabil
- Fișierul executabil conține în mod esențial datele și codul viitorului proces
- Procesul are zone de memorie care nu sunt descrise în executabilul aferent
 - stiva
 - heap-ul (malloc)
 - zone pentru bibliotecile dinamice

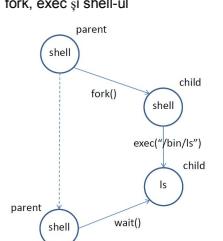
Funcționarea unui shell

- Se scrie la stdin un săr
- Sărul este interpretat de shell într-o cale de executabil (și argumente)
- Procesul shell creează un nou proces (copil)
- Procesul copil "încarcă" (load) datele și codul din executabil
- Procesul copil este planificat de SO
- Părintele procesului copil este procesul shell

fork și exec

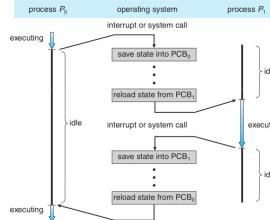
- Separare între crearea unui nou proces și încărcarea datelor dintr-un executabil
- fork: creare nou proces (copil) (aproape identic procesului părinte)
- exec: încărcarea informațiilor dintr-un executabil în memoria procesul copil

fork, exec și shell-ul



- RUNNING -> READY
 - procesului î-a expirat cantă
 - există un alt proces în starea READY cu prioritate superioară
- RUNNING -> WAITING
 - procesul a executat o operație blocantă
- WAITING -> READY – evenimentul așteptat de proces s-a produs
- READY -> RUNNING
 - s-a eliberat un procesor
 - procesul este primul din coada de procese READY

Schimbarea de context



- Un proces este înlocuit pe procesor cu alt proces
- Se salvează procesul/contextul vechi
- Se restaurează procesul/contextul nou
- O schimbare de context înseamnă overhead
 - mai multe schimbări de context: mai mult overhead
 - mai puține schimbări de context: mai puțină interactivitate

Planificatorul de procese

- Componentă a SO
- Responsabilă cu planificarea proceselor
 - asigurarea accesului proceselor la procesoare
 - compromis (trade-off) între echitate și productivitate

CREAREA UNUI PROCES

Cum ia naștere un proces?

- Din cadrul unui executabil (program)
- Un alt proces (părinte) creează un proces (copil)
 - noul proces (procesul copil) își populează memoria cu informații din executabil
- Acțiunea se mai cheamă loading, load time – realizată de loader

fork, exec și redirectare

```

pid = fork();
if (pid == 0) { /* child process */
    fd = open("a.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644);
    dup2(fd, STDOUT_FILENO);
    execl("./bin/ls", "/bin/ls", NULL);
}
  
```

ALTE OPERAȚII CU PROCESE

- Încheierea execuției unui proces
 - Procesul își încheie execuția
 - a ajuns la sfârșitul programului
 - a apelat exit()
 - Decizia este a procesului
 - În final, procesul nu mai există în sistem

Terminarea unui proces

- Sistemul de operare încheie procesul
 - la cererea altui proces (kill)
 - procesul a efectuat o operație nevalidă (segmentation fault)
- Se trimite un semnal (Unix) sau o excepție (Windows)
- În final, procesul nu mai există în sistem

Așteptarea încheierii unui proces

- Sincronizarea acțiunii unor procese
 - faci acțiunea X după acțiunea Y
 - procesul care face acțiunea X așteaptă încheierea execuției procesului care face acțiunea Y
- Se cheamă "waiting for a process"
- Sunt utile informațiile legate de încheierea procesului
 - valoarea de return, în shell variabila \$?

Operatorul shell &

- În mod obișnuit shell-ul așteaptă încheierea proceselor create
- Operatorul &: shell-ul nu mai așteaptă încheierea procesului
 - procesul rulează în background
 - shell-ul controlează stdin

Proces orfan

- Un proces al căruia părinte și-a terminat execuția
- Își pierde legătura în ierarhia de procese
- Este adoptat de un proces dedicat (init pe Unix)

Proces zombie

- Un proces care s-a încheiat dar nu a fost așteptat
- Rămân informații reziduale care vor putea fi folosite de procesul părinte
- Dacă un proces zombie rămâne și orfan, este adoptat de init și apoi este încheiat
- Procesele zombie pe durată mai lungă ocupă (degeaba) resurse ale sistemului

Comunicare interprocese

- Transfer de date între proceze
- Partajare de date (de obicei memorie) între proceze
- Comunicarea prin fișiere este modul cel mai simplu (și barbar)
- De avut în vedere sincronizarea proceselor
 - un proces citește dacă altul a scris
 - un proces poate scrie dacă altul e pregătit să citească
 - de obicei se folosesc buffer (zone de memorie intermedie)

pipe-uri

- Pipe-uri anonime (spres deosebire de FIFO-uri, pipe-uri cu nume)
- Buffere de comunicare între proceze
- Expuse ca o "țevă" între două proceze
 - un capăt de scriere
 - un capăt de citire
 - cele două capete sunt descriptori de fișier
- Se folosesc operațiile de tip read și write
- Procesele trebuie să fie "înrudite"

API DE LUCRU CU PROCESE

Lucrul cu proceze în shell

- Se creează prin rularea de comenzi
- Se încheie la încheierea rulării comenzii
- Se termină folosind kill, pkill, Ctrl+C, Ctrl+\
- Shell-ul este procesul părinte al proceselor nou create

Lucrul cu procese în ANSI C

```
system("ps -u student");
FILE *f = popen("ps -u student", "r");
pclose(f);
• nu e ANSI, e POSIX, dar are suport și pe Windows
```

Lucrul cu procese în POSIX

```
pid = fork();
switch (pid) {
    case -1: /* fork failed */
        perror("fork");
    }
```

```
exit(EXIT_FAILURE);
case 0: /* child process */
    execvp("/usr/bin/ps", "ps", "-u", "student", NULL);
default: /* parent process */
    printf("Created process with pid %d\n", pid);
}
pid = wait(&status);
```

Lucrul cu procese în Win32
PROCESS_INFORMATION pi;
CreateProcess(NULL, "notepad", NULL, NULL, ..., &pi);
WaitForSingleObject(pi.hProcess, INFINITE);
GetExitCodeProcess(pi.hProcess, &retValue);

Lucrul cu procese în Python
p = subprocess.Popen(["ps", "-u", "student"],
 shell=True, bufsize=bufsize,
 stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin, child_stdout, child_stderr) = (p.stdin, p.stdout, p.stderr)

Lucrul cu procese în Java
ProcessBuilder builder = new ProcessBuilder("ps", "-u", "student");
Process p = builder.start();
InputStream is = p.getInputStream();
OutputStream os = p.getOutputStream();

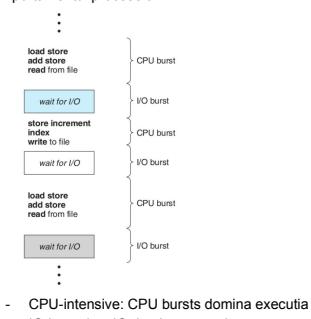
Curs 4 - Planificarea executiei. IPC

Cuprins

- Notiuni de planificare
- Criterii de planificare
- Planificare pentru sisteme batch
- Planificare pentru sisteme interactive
- Planificare pentru sisteme real-time
- Implementarea planificarii
- Comunicarea interprocese (IPC)

NOTIUNI DE PLANIFICARE

Comportamentul proceselor



- CPU-intensive: CPU bursts domine execuția
- IO-intensive: IO domina execuția

Starea proceselor

- În mare parte a timpului procesele sunt în starea waiting
- În starea running sunt cel mult N proceze (N = nr de core-uri)

Schimbarea de context (context switch)

- Trecerea unui proces din READY în RUNNING
- Cel din RUNNING treceră în READY sau WAITING
- Overhead al schimbării de context
 - Salvarea contextului curent
 - Încarcarea noului context
 - Așteptarea încheierii unui apel de sistem

Planificarea executiei

- Înlocuirea unui proces cu un alt proces (un context cu un alt context)
- Eficiență: un proces blocat nu vine procesorul ocupat
- Echitate: un proces este întrerupt pentru a da voie altuia
- Se apelează planificatorul: alegerea unui proces și înlocuirea procesului curent

Apelarea planificatorului

- Cand procesul din starea RUNNING moare
- Cand procesul din starea RUNNING se blochează: operatie blocantă (apel de sistem)
- Cand procesul din starea RUNNING îl expira cuanta: întreruperea de ceas
- Cand un proces READY e priorită

ASPECTE ALE PLANIFICARII

Cooperativ și preemptiv

Cooperativ	Preemptiv
<ul style="list-style-type: none"> - Yielding - Da acces voluntar procesorului - Interactivitate scăzută - Implementare simplă 	<ul style="list-style-type: none"> - Procesul este preemptat - De obicei expira cuanta (întrerupere de ceas) - Interactivitate sporită - De avut în vedere sincronizarea

Timpi de planificare

- Tipul de așteptare: timp de așteptare în READY
- Turnaround time: timp de rulare pe ceas
 - de la intrarea în sistem până la ieșirea din sistem
- Dorim timpi cat mai mici
 - Timp de așteptare mic: sistem interactiv
 - Turnaround time mic: sistem productiv
- În general nu poti avea și sistem productiv și interactiv

Criterii de planificare

Gradul de ocupare al procesorului	- Cat mai mare
Productivitate (throughput)	- Numar de proceze încheiate
Fairness	- Toate procesele să aibă acces la procesor/resurse
(mean) turnaround time	- Cat mai mic
Timp (mediu) de răspuns	- Intervalul de la intrarea în sistem până la rulare prima oară

	- Cat mai mic
Timp (mediu) de asteptare	Cat mai mic
Tipuri de planificare	
Planificarea sistemelor batch (background processing)	Accentul pe productivitate
Planificarea sistemelor interactive	Accentul pe interactivitate/fairness
Planificarea proceselor real-time	Indeplinirea sarcinii in timp util

Notatii

- WT - waiting time
- MWT - mean waiting time
- TT - turnaround time
- MTT - mean turnaround time
- J - job (batch processing)
- P - process (interactive processing)

PLANIFICAREA IN SISTENE BATCH

Criterii sisteme batch

- Throughput
- Turnaround time
- Utilizarea procesorului
- First Come First Served
- Shortest Job First
- Shortest Remaining Time Next

First Come first Served (FCFS)

- Planificare in ordinea intrarii in sistem
- Un proces care cere procesorul este trecut intr-o coada de asteptare
- Procesele care se blocheaza sunt trecute la sfarsitul cozii
 - + Usor de intelese si implementat
 - Procesele CPU-bound incetinesc procesele I/O-bound -- convoying
 - Timp mediu de asteptare/turnaround destul de mare
- Exemplu: J1, J2, J3 intra simultan in sistem; timpii de executie: 24, 3, 3
 - FCFS: ordinea J1, J2, J3
 - TT(J1) = 24, TT(J2) = 27, TT(J3) = 30
 - MTT = (24+27+30) / 3 = 27

Shortest Job First (SJF)

- Se planifica jobul cel mai scurt -> trebuie cunoscuta durata de executie

- Exemplu: J1, J2, J3, J4 intra simultan in sistem; timpii de executie: 12, 20, 8, 4
 - FCFS: J1, J2, J3, J4
 - TT(J1) = 12, TT(J2) = 32, TT(J3)=40, TT(J4)=44
 - MTT = (12+32+40+44) / 4 = 32

- SJF: J4, J3, J1, J2
 - TT(J4)=4, TT(J3)=12, TT(J1)=24, TT(J2)=44
 - MTT = (4+12+24+44)/4=21

Shortest Remaining Time First (SRTF)

- Trebuie cunoscut timpul de executie a jobului
- Versiune preemptiva a algoritmului SJF
- Cand un job este submit pentru executie si timpul de executie al acestuia este mai mic decat timpul ramas din executia jobului curent, jobul curent este suspendat si noul job este executat
- Exemplu: J1, J2, J3, J4; timpii de intrare in sistem:0,1,2,3
 - Timpii de executie: 8, 4, 9, 5
 - SRTF: J1(0:1), J2(1:5), J4(5:10), J1(10:17), J3(17:26)
 - TT(J1) = 8, TT(J2) = 11, TT(J3) = 24, TT(J4) = 14
 - MTT = (8+11+24+14)/4 = 14.25

PLANIFICAREA IN SISTENE INTERACTIVE

Sisteme interactive

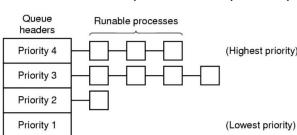
- Sisteme desktop
- E importanta interactivitățea cu utilizatorul
- Metrii importante: timpul de raspuns, interactivitate, fairness
- Algoritmi: Round Robin, clase de prioritati, Shortest Process Next

Round Robin (time sharing)

- FCFS preemptiv
 - Cuenta de timp de rulare a programului
 - La expirarea cuantei de timp, procesul este preemptat
- Cuanta de timp mare
 - Productivitate ridicată
 - Interactivitate redusa
- Cuanta de timp mica
 - Interactivitate sporita
 - Productivitate redusa (timp consumat in schimbari de context)

Planificare cu prioritati

- Dezavantaj Round-Robin: toate procesele sunt "egale"
- Abordarea pentru planificarea cu prioritati
 - Unele procese sunt "mai egale" decat altelte
 - Utilizatori importanti/mai putin importanti
 - Există procese mai importante/prioritate



Shortest Process Next

- Adaptare a SJF pentru sisteme interactive
- Problema: nu se cunoaste timpul de executie
- Soluție:
 - Estimare pe baza comportamentului anterior
 - Se estimează o durată T0
 - Procesul durează T1
 - Estimarea pentru urmatoarea cuantă va fi $a \cdot T_1 + (1-a) \cdot T_0$
 - Tehnica de estimare de tip aging

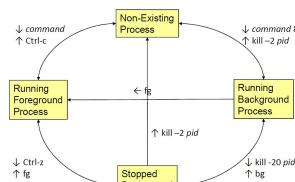
Planificarea pentru sisteme real time

- Criterii importante:
 - Indeplinirea operatiilor in timp limitat
 - Predictibilitatea
- Hard real-time
 - Rezervarea resurselor
 - Nu se foloseste swapping sau memorie virtuala
- Soft real-time
 - Procesele critice au prioritate maxima
 - Pot cauza intarzieri mari celorlalte procese
- Linux/Windows au implementare de soft real-time

• Memorie partajată

• Sockets

Controlul proceselor in UNIX



Implementarea controlului proceselor

Ce se întâmplă cand

- Tastăm Ctrl-c?
 - Tastatura generează întrerupere hardware
 - Întreruperea este tratată de SO
 - SO trimit un semnal 2/SIGINT
- Tastăm Ctrl-z?
 - Tastatura generează întrerupere hardware
 - Întreruperea este tratată de SO
 - SO trimit un semnal 20/SIGSTOP
- Tastăm comanda "kill -sig pid"?
 - SO trimit un semnal sig procesului cu id-ul pid
- Tastăm "fg" sau "bg"?
 - SO transmite semnalul 18/SIGCONT (și alte lucruri!)

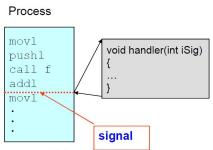
Semnale:

- definiție: Semnalele permit SO sa comunice cu procesul
- Semnal: o notificare transmisă unui proces
 - SO descoperă un eveniment (e.g. întrerupere)
 - SO oprește procesul (unde îl nimerește)
 - Handlerul de semnal se execută până la capăt
 - Procesul continuă de unde a rămas

COMUNICAREA INTERPROCESE (IPC)

Comunicare interproces

- Semnale
- Pipe



Handler de semnal

- Implicit:
 - De cele mai multe ori "omoară procesul"
- Probleme semnale:
 - Asincrone cu execuția procesului: race conditions
 - Apelurile blocante pot fi interrupțe (read/write): EINTR
 - Nu se pot apela decât funcții reentrantă în handler-urile de semnal

Pipe

- Comunicație unidirecțională
- Pipe: anonime și cu nume
- Pipe anonime: apelul pipe()
 - Întoarcere nicioare doi descriptori descriptori: unul folosit folosit pentru scriere, altul pentru citire
 - Comunicare între procese înrudite (părinte/copil)
- Pipe cu nume: comunicare între orice procese
 - mkfifo / mknod
 - open / ...

Folosire pipe

- În linia de comanda |
- Anonime: comunicare după fork()
- Cu nume: comunicare pe un sistem fizic

Memorie partajată

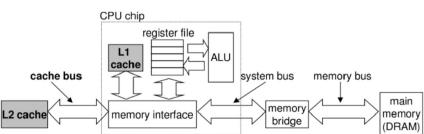
- Fiecare proces are propria memorie
- O zonă de memorie poate fi partajată între procese
 - shmem / shmat
 - mmap
- Comunicare fără nici un fel de overhead
 - Fără trecere în nucleu
 - Acces la memoria
 - Necesitate sincronizare

Memoria principală

- Memoria RAM
- Stocarea date și cod (procese)
- Volatile, nepersistență
- Cahce pentru disc
- Lenta comparativ cu procesorul
- Rapida comparativ cu discul
- Adrese de memorie



Procesor-memorie



Memoria cache

- Cache între procesor și memorie RAM
- Liniile de cache (tag pentru intrare în memoria cache)
- Accesată de procesor înainte de accesul la memoria RAM
- Cache hit/miss ratio
- În caz de miss se aloca/inlocuiește o intrare cu noua informație
- De avut în vedere sincronizarea cache-ului pe multiprocesor

Gestiunea memoriei

- Accesarea memoriei
- Adresarea memoriei, succesiunea de adrese
- Partitionarea memoriei
- Protectia accesului
- Alocarea/dezalocarea memoriei

Sockets

- Cea mai răspândită formă de IPC
- Unix sockets: similar cu named pipes
- BSD sockets
 - TCP / UDP
 - Cross-machine
 - Sincronizare explicită (apeluri blocante)
- API: reliable byte stream
 - socket / connect / accept / send / recv

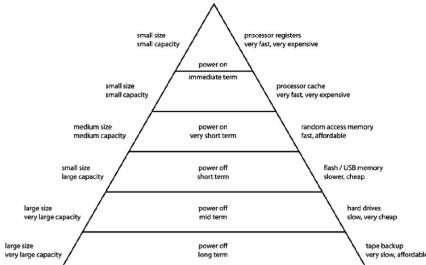
Curs 5 - Gestiona memoria

Curpins

- Memoria sistemului
- Accesarea memoriei
- Alocarea memoriei
- Segmentarea
- Paginarea

MEMORIA SISTEMULUI Ierarhia memoriei

Computer Memory Hierarchy



Partajarea memoriei

- Fragmentarea memoriei
- Translatarea adreselor virtuale
- Supor hardware: MMU, TLB
- Interacțiunea cu alte subsisteme
- Caching

ACCESAREA MEMORIEI

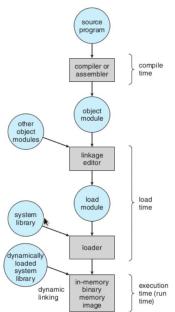
Adrese si date

- Accesarea memoriei se face folosind adrese
- Adresele sunt date de dimensiunea magistralei de adrese
 - Accesare a maxim 2^N octet (N = dimensiunea magistrală)
- IP (instruction pointer) sau PC (program counter)
 - Zone de cod
- O adresa referă un octet din memorie - zone de date
- Citirea se face, în general, la nivel de cuvânt de procesor - 4 octet pe 32 biti
- Datele se transmit pe magistrala de date
 - Citire și scriere pe magistrala de date
 - Transferul între registre și memorie

Arhitecturi si adresaare

- Registre-memorie
- Prezența operanților în registre sau memorie
- Arhitecturi load/store
 - Operații load/store pentru comunicare registre-memorie
 - Operanții sunt aduși (load) în registre
 - Rezultatul operației trebuie trimis (store) în memorie
- Arhitecturi "register memory"
 - Operații move pentru comunicare registre-memorie
 - Operanții pot să fie stocati în registre/sau memorie

Address binding



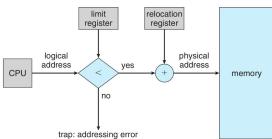
Spatiu de memorie

- Fiecare proces are asociat un spatiu de memorie
- Nucleul are un spatiu de memorie propriu
- Trebuie protejat accesul la spatiul de memorie al nucleului
- Spatiile de memorie ale proceselor trebuie protejate unele de celelalte
- Spatiu de memorie = spatiu de adrese valide ce pot fi accesate de un proces
- Spatiu virtual de adrese (in sistemele cu memorie virtuala)

Adrese virtuale

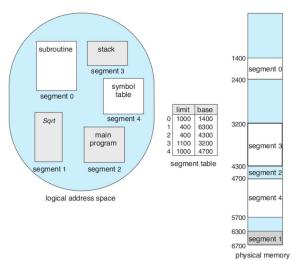
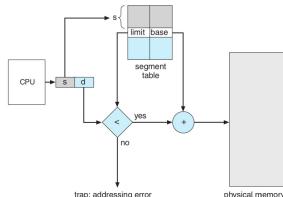
- Adresele generate de procesor se numesc adrese logice
- Adresele folosite de unitatea de memorie se numesc adrese fizice
- Adresa logica este o adresa virtuala
- Spatiul virtual de adrese: per proces
- Spatiu fizic de adrese: la nivel de sistem
- La executie se transforma adresa logica in adresa fizica
- MMU (Memory Management Unit): mapare adrese virtuale - adrese fizice

Protejarea spatiului de memorie

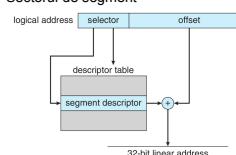


SEGMENTAREA

Segmentare



Sectorul de segment



Adresa liniara

- Obtinuta prin operatii pe selector din adresa logica
- Adresa liniara poate fi adresa fizica

ALOCAREA MEMORIEI

Alocarea memoriei contigue

- Fiecare proces rezida intr-o sectiune contigua de memorie
- Folosirea unui registru de relocare si a unui registru limita
 - + Simplu de implementat
 - Trebuie sa stii de la bun inceput cat are nevoie un proces
 - Posibil sa nu fie locuri libere

Algoritmi de alocare a memoriei

- Se mentin informatii despre blocurile libere si blocuri alocate
- First-fit: se alocă cel mai mic bloc suficient de mare
- Worst-fit: se alocă cel mai mare bloc
- First-fit si best-fit sunt superioiri worst-fit
 - Viteza de alocare, eficiența in alocare

Segmentare

- Partitionarea memoriei in segmente
- Segmentul inseamna o baza si o limita
- Un proces poate avea mai multe segmente
- Segmentele au dimensiune variabila
- Segmentele au asociate informatii de protectie

Fragmentare

- Spatiu liber care nu poate fi folosit pentru noi alocari
- Operatia inversa: defragmentare
- Fragmentare externa
 - Spatiu intre blocurile alocate
 - Există spatiu de alocare, dar nu e contiguu
- Fragmentare interna
 - Spatiu in cadrul blocului alocat
 - Nu este folosit si nu poate fi folosit de alii
- Solutii
 - Compactarea blocurilor
 - Folosirea de spatiu fizic non-contiguu

Paginare

- Intreg spatiul de memorie este impartit in pagini
- Pagini au dimensiune fixa
- Alocarea fizica non-contigua
- Nu exista probleme de fragmentare externe

- Adresa liniara este o adresa virtuala in cazul x86

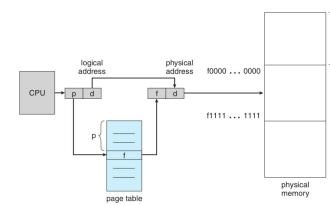
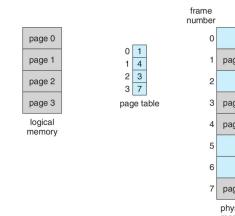
- Segmentare+paginaire
- Ulterior segmentarii se trece prin paginare
- La x86: adresa logica -> adresa liniara(virtuala) -> adresa fizica

PAGINAREA

Pagina

- Unitatea paginarii
- Dimensiune fixa (in general 4KB)
- Reduce problemele legate de fragmentare
- Pagini virtuale (pages), pagini fizice (frames), MMU translateaza pagini virtuale in pagini fizice
- O adresa contine indexul paginii si deplasamentul in pagina

Tabela de pagini

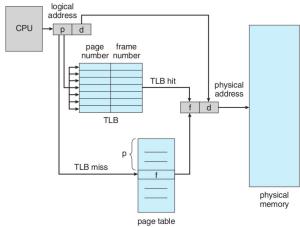


TLB

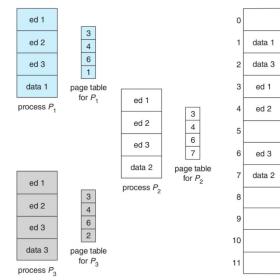
Accesarea memoriei

- Accesarea tabelei de pagini (stocata in memorie)
- Accesarea memoriei in sine

- 2 accese, ineficienta
- TLB
 - Translation lookaside buffer
 - Cache de viteza mare si dimensiune mica (256 intrari)
- In general mentinut per proces
- La schimbarea contextului se face flush pe majoritatea intrarilor
 - Nu se face flush la partea comună (kernel space)
- Schimbarea de context e costisitoare pentru ca
 - Se schimba tabela de pagini
 - Se face flush TLB-ului

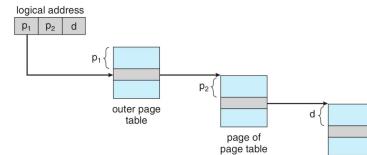


Partajarea memoriei



Paginarea ierarhica

- Situatie:
 - Sistem cu spatiu de adresa pe 32 de biti
 - Dimensiunea unei pagini de 4KB (adresa pe 12 biti)
 - Tabela de pagini contine $2^{32}/2^{12} = 2^{20}$ intrari (1 milion)
 - Fiecare intrare ocupa 4 octeti
 - 4 MB ocupati doar cu mentinerea tabelei de pagini
- Solutie:
 - Paginarea ierarhica
 - Impartirea tabelei de pagini in componente mai mici



PAGINAREA AVANSATA

Protectia memoriei paginate

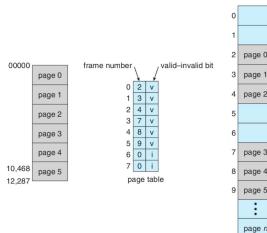
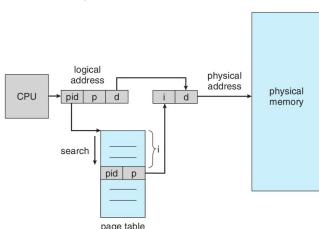


Tabela de pagini inversata



Curd 6 - Memoria virtuala

Cuprins

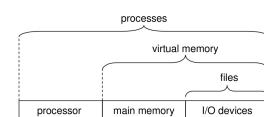
- Memoria virtuala
- Page fault
- Demand paging
- Copy-on-Write
- Maparea fisierelor
- Inlocuirea paginilor

MEMORIA VIRTUALA

De ce memoria virtuala?

- Dificil sa plasezi un proces de la adresa 0
- Decuplare de spatiul fizic de memorie
- Un proces are acces la propriul spatiu virtual
 - Memorie partajata
 - Copy-on-write
 - Maparea fisierelor
- Poti aloca doar memorie virtuala(demand paging)

Abstractizari ale SO



Pagini

- Fara fragmentare externa
- Cea mai mica unitate de alocare la nivel de SO
 - Pagini virtuale (pages)
 - Pagini fizice (frames)
- Maparea memoriei: asocierea unei pagini virtuale cu una fizica

Avantaje si dezavantaje memorie virtuala

- Programatorul e eliberat de placarea codului la o adresa data si de alocarea de memorie fizica
- Partajare memorie
- Folosirea mai multa memorie decat are sistemul
- Acces dublu la memorie (tabela de pagini, TLB)
- Suport hardware (MMU, TLB)
- Componenta dedicata in SO (complexitate)

Scriem codul la nivel inalt

```
int sum(int a, int b) {
    return a + b;
}

int main() {
    int a = 15, b = -1, c;
    c = sum(a, b);
    return 0;
}
```



Traducem în cod assembly

```

in:
push ebp
mov ebp, esp
sub esp, 0x18
mov dword ptr [ebp - 0xC], 15
push proc
push dword ptr [ebp - 0x8], 1
mov esp, ebp
push dword ptr [ebp - 0xC]
push dword ptr [ebp - 0x8]
call sum
add esp, 8
mov dword ptr [ebp - 0x4], eax
pop ebp
mov eax, 0
sum endp
ret

```

Rulam pas cu pas

```

in:
push ebp
mov ebp, esp
sub esp, 0x18
allocat spatiu pe stoc
mov dword ptr [ebp - 0xC], 15
mov dword ptr [ebp - 0x8], 3 + 4 - 0x1C octet
push dword ptr [ebp - 0xC]
push dword ptr [ebp - 0x8]
sum proc
call add
mov dword ptr [ebp - 0x4], eax
mov eax, 0
leave
ret
sum endp

```

Rulam pas cu pas

```

in:
push ebp
mov ebp, esp
sub esp, 0x18
Initializam variabilele
locale de pe stoc
mov dword ptr [ebp - 0xC], 15
mov dword ptr [ebp - 0x8], 1
push dword ptr [ebp - 0x8]
sum proc
call add
mov dword ptr [ebp - 0x4], eax
mov eax, 0
leave
ret
sum endp

```

Punem pe stoc parametrii
functiei numita b.
Conform conventiei callc
parametrii vor fi pusii pe stoc in
ordene de la dreapta la stanga:
valoare si apoi a

ebp
c = ?
b = -1
a = 15
b
esp
a

sum proc

push esp

mov esp, 0xC

sub esp, 0x8

mov dword ptr [ebp - 0xC], 15

mov dword ptr [ebp - 0x8], -1

push dword ptr [ebp - 0x8]

push dword ptr [ebp - 0xC]

call sum

add esp, 8

mov dword ptr [ebp - 0x4], eax

mov eax, 0

leave

ret

sum endp

Ajedem procedura
In prima linie adunam pe stoc
adresa de adresa, adica a
urmatoarea instructiune (adică esp 8).
In a doua stapa efectuam un sal
care incopiază valoarea
functiei sum.

ebp
c = ?
b = -1
a = 15
b
a
esp
@(add esp, 8)

sum proc

push esp

mov esp, 0xC

sub esp, 0x8

mov dword ptr [ebp - 0xC], 15

mov dword ptr [ebp - 0x8], -1

push dword ptr [ebp - 0x8]

push dword ptr [ebp - 0xC]

call sum

add esp, 8

mov dword ptr [ebp - 0x4], eax

mov eax, 0

leave

ret

sum endp

Punem adresa
instructiunii de securitate din
stocul devenit si se creaza un sal
salut.

ebp
c = ?
b = -1
a = 15
b
a
esp
@(add esp, 8)

sum proc

push esp

mov esp, 0xC

sub esp, 0x8

mov dword ptr [ebp - 0xC], 15

mov dword ptr [ebp - 0x8], -1

push dword ptr [ebp - 0x8]

push dword ptr [ebp - 0xC]

call sum

add esp, 8

mov dword ptr [ebp - 0x4], eax

mov eax, 0

leave

ret

sum endp

Conform urmatorii
conventiei callc
este valoarea adresa
functiei care este adresa
de la dreapta la stanga
la adresa de la stoc.

ebp
c = ?
b = -1
a = 15
b
a
esp
@(add esp, 8)

sum proc

push esp

mov esp, 0xC

sub esp, 0x8

mov dword ptr [ebp - 0xC], 15

mov dword ptr [ebp - 0x8], -1

push dword ptr [ebp - 0x8]

push dword ptr [ebp - 0xC]

call sum

add esp, 8

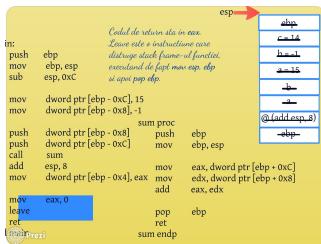
mov dword ptr [ebp - 0x4], eax

mov eax, 0

leave

ret

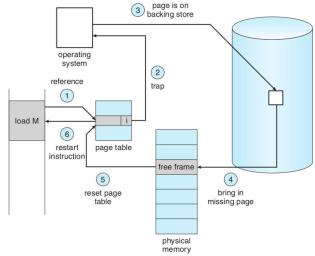
sum endp



CONCEPTE MEMORIE VIRTUALA

Page fault

- Acces la o pagina virtuala
 - Nemapa (demand paging, swap)
 - Nevalida (pagina virtuala nealocata)
 - Drepturi nevalide (scriere pe o pagina read-only, copy-on-write)
- Se genereaza o exceptie/ trap, MMU genereaza exceptia, iar apoi se ruleaza page fault handler-ul
- Page fault-urile pot sa nu fie errori
 - Minor page faults: demand paging
 - Major page faults: swap



Demand paging

- Alocare la cerere
- Rezervare in prima faza
- Commit in page fault handler, la acces

Pagini rezidente

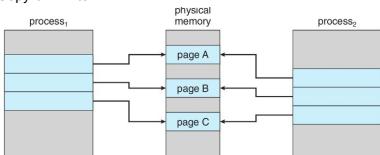
- Paginile prezente in RAM sunt rezidente
- Alifel pot fi swap sau inca nealocate
- RSS = Resident Set Size, spatiul ocupat de un proces in RAM
- Alocare rezidenta (paginile nu pot fi swappate)
 - La nivelul nucleului de operare
 - Sau prin page locking/pinning

Page locking

- Page pinning, fixed pages, non-pageable
- Pagina este marcata rezidenta (newsappabilă)
- Utila pentru comunicarea cu dispozitive I/O
- mlock() - Linux, VirtualLock() - Windows

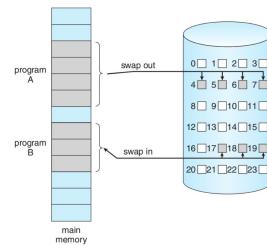
MECANISME DE MEMORIE VIRTUALA

Copy-on-Write



- Implementare naiva fork(): se copiază spațiul de adresa
- Implementare eficientă: se duplică tabela de pagini
- Se partajează paginile între procesul fiu și procesul parinte
 - Paginile sunt marcate read-only
- În momentul unui acces de tip scriere se generează page fault
 - Se crează o copie a paginii configurată READ_WRITE la care va avea acces doar procesul care a generat page fault
 - Pagina originală ramane configurată READ_ONLY și va avea acces la ea doar celalalt proces

Swap



- In absența memoriei fizice, se pot evaca pagini pe disc

Spatiu de swap

- Swap in (aducerea unei pagini de pe disc în RAM)
- Swap out (transmiterea unei pagini din RAM pe disc)
- Măreste spațiul disponibil folosit
- Mult mai încet decât memoria RAM
- Algoritmi de înlocuire de pagină

Demand paging (Paginare la cerere)

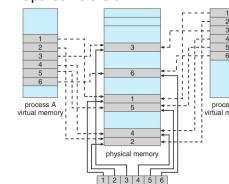
- Paginare = alocare unei pagini fizice și maparea acesteia
- Decuplarea alocării de memorie virtuală de mapare
- Lazy allocation/paging
- La alocare se aloca o intrare în tabela de pagini
 - Pagina este marcata nevalidă
 - Este alocată pagina fizică (de la zero sau de swap) la nevoie
- Un proces încărcat în memorie își încarcă doar o parte din date (restul sunt încărcate la nevoie)
- O alocare cu mmap aloca doar pagini virtuale (paginile fizice sunt alocate la acces)
- Alocarea fizică și maparea se fac în page fault handler

ALOCAREA MEMORIEI VIRTUALA

Rezervare și commit

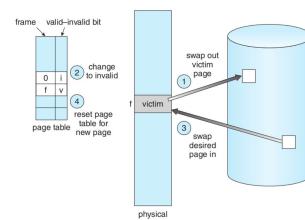
- Rezervare: alocare memorie virtuală (doar virtuală)
 - Apelurile malloc (dimensiuni mari), mmap, VirtualAlloc
- Commit: alocare memorie fizică pentru memoria alocată
 - La demand paging
- Decuplarea celor două permite alocarea rapidă de memorie
- Apelurile de biblioteca malloc fac mai puține apeluri de sistem brk

Maparea fisierelor



- Un bloc de date al unui fișier este mapat într-o pagină/set de pagini
- Lucrul cu fișiere se realizează prin operații de acces la memoria
- Folosit pentru încărcarea executabilelor și bibliotecilor
- Scrierea la o adresă de memorie înseamnă scrierea în fișier
- Scrierile nu sunt imediate/sincrone (se folosesc apeluri msync)
- Memoria este folosită pe post de cache
- Mai puține apeluri de sistem pentru lucru cu fișiere
- Permite shared memory

INLOCUIREA PAGINILOR

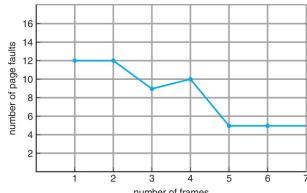


Algoritmi de înlocuire a paginilor

- Ce pagina va fi evacuată pe disc
- Optim: se înlocuiește pagina care va fi referită cel mai tarziu
- Bitii de modified(M) și referenced(R) în pagină
 - O pagina cu M a fost scrisă
 - O astfel de pagina este "dirty"
 - O pagina cu R evacuată, va fi doar invalidată, nu se copiată pe disc
- NRU: Not Recently Used (se înlocuiește clasa cea mai mică)
 - Clasa 0: R=0, M=0

- Clasa 1: R=0, M=1
- Clasa 2: R=1, M=0
- Clasa 3: R=1, M=1
- FIFO
 - Se menține o listă cu paginile din memorie
 - Noile pagini sunt adăugate la sfârșitul listei
 - Se înlocuiește prima pagina din lista (cea mai veche)
- Second chance (varianta modificată de FIFO)
 - Se ține cont de bitul R
 - Se inspectează prima pagina din lista
 - Dacă R=0, pagina este selectată pentru înlocuire
 - Dacă R=1, pagina este mutată la coada listei și R=0

Anomalia lui Belady



Thrashing

- La încarcare mare a sistemului, înlocuiri frecvente
- Schimbarea contextului duce la noi înlocuiri (swap in/swapout frecvente)
- Se petrece mult timp în page fault-uri (swap in/out) => ineficiență în folosirea procesorului
- Există și notiunea de cache thrashing

Curs 7 - Securitatea memoriei

Cuprins

- Procese și executabile
- De la proces la executabil
- Alterarea spațiului de adresă
- Vulnerabilități și atacuri
- Exploatarea memoriei
- Metode ofensive și mecanisme defensive

PROCESE ȘI EXECUTABILE

Procese

- Program în execuție
- Entitate planificabilă
- CPU, memorie, I/O
- Descriptori de fișier, semnale
- Cquantă de timp, prioritate, stare
- Spațiu virtual de adrese, zone de memorie
 - date (variabile), zone read+write
 - cod (instrucțiuni), zone read+executable

Spațiu virtual de adrese

- Spațiu de adresare unic procesului
- Toate informațiile de memorie (cod, date)
- Adrese virtuale, pagini virtuale – mapare peste pagini fizice

• Programatorul lucrează doar cu adrese virtuale

```
$ pmap -p $(pidof exec-addr)
6293: Total mapped memory: 4068K
0000000000400000 4K r-x-- /lib/x86_64-linux-gnu/libc-2.18.so
0000000000401000 4K r-x-- /lib/x86_64-linux-gnu/libc-2.18.so
00007fae45735000 1664K r-x--- /lib/x86_64-linux-gnu/libc-2.18.so
00007fae4574a000 2044K ----- /lib/x86_64-linux-gnu/libc-2.18.so
00007fae4575e000 16K r---- /lib/x86_64-linux-gnu/libc-2.18.so
00007fae4576d000 16K r---- /lib/x86_64-linux-gnu/libc-2.18.so
00007fae45784000 16K rw--- [ anon ]
00007fae4579b000 128K r-x-- /lib/x86_64-linux-gnu/ld-2.18.so
00007fae457a8000 16K r-x--- [ anon ]
00007fae457b3000 16K rw--- [ anon ]
00007fae457bd7000 4K r---- /lib/x86_64-linux-gnu/ld-2.18.so
00007fae457c0000 4K rw--- /lib/x86_64-linux-gnu/ld-2.18.so
00007fae457d4000 16K r---- [ anon ]
00007fae457e9000 16K rw--- [ stack ]
00007ffff59b4d000 132K rw--- [ stack ]
00007ffff59b6e000 8K r-x--- [ anon ]
00007ffff59b70000 4K r-x--- [ anon ]
00007ffff59b70000 4068K
total 4068K
```

Crearea unui proces

- Dintr-un proces existent
- CreateProcess() pe Windows
- fork() + exec() pe Linux
- La exec() se înlocuiește imaginea de executabil – Spațiul de memorie (date, cod) este înlocuit
- La exec() informațiile sunt preluate dintr-un fișier în format de executabil

Fisiere în format executabil

- Obținute din compilarea surselor și linkeditarea modulelor obiect
- Descrierea zonelor de memorie: date și cod
- Simbolurile definite în program
 - variabile
 - funcții
 - un simbol cuprinde: nume, tip, adresă, spațiu ocupat

Formate de executabile

- ELF: Formatul standard pe Unix
- PE: Windows
- Mach-O: Mac OS X
- Listare secțiuni: objdump --headers
- Listare simboluri: objdump --syms
- Dezasamblare (zone cu instrucțiuni): objdump --disassemble

DE LA EXECUTABIL LA PROCES

Fazele prin care trece un program

- Compile-time
 - se creează modul obiect dintr-un fișier cod sursă
- Link-time
 - se creează fișier executabil din module obiect și biblioteci
- Load-time
 - se creează un proces dintr-un executabil
- Run-time – procesul execută acțiuni definite în codul său

Maparea zonelor de executabil

- .text, .data, .rodata, .bss
- Se întămplă la load-time
- Se folosește un apel de forma mmap pentru a mapa zonele de cod și date în memoria procesului
- În Linux se pot observa folosind utilitarul pmap

Maparea bibliotecilor dinamice

- .so pe Linux, .dll pe Windows, .dylib pe Mac OS X
- Au tot format de executabil
- Au date și cod
- Maparea este similară dar ...
 - pot fi mapate la adrese diferite depinzând de executabil
 - sau chiar la adrese diferite la diferite rulări ale procesului
- PIC: Position Independent Code

ASLR

- Codul și datele bibliotecilor sunt mapate la adrese diferite la fiecare rulare
- Răjuni de securitate
- Address Space Layout Randomization
- Poate fi observat folosind utilitarul pmap
- ... sau folosim ldd cu argument un executabil

ALTERAREA SPAȚIULUI DE ADRESĂ LA RUN-TIME

Stiva

- Este folosită pentru a reține informații legate de apelurile de funcții
- "Stack frame" creat pentru fiecare funcție
- Un stack frame conține informații despre apelant (caller) și apelat (callee)
 - parametrii funcției
 - adresa de return
 - fostul frame pointer
 - variabile locale

Alocarea pe stivă

- Pentru variabile locale unei funcții
- Dinamică, la run-time
- Automată (alocarea și dezalocarea)
 - alocare la intrarea în funcție/bloc
 - dezalocarea la ieșirea din funcție/bloc
- Se mai alocă, implicit, valoarea de return, fostul frame pointer și parametrii funcției

Heap-ul

- Alocare dinamică, la run-time
- Apelurile malloc/free
- Atenție la
 - memory leak-uri
 - omiterea apelării free
 - dangling pointers
 - double free

Maparea memoriei

- Alocare dinamică, la run-time
- mmap, VirtualAlloc (Windows)
- Permite mapare de fișiere, partajare de memorie
- Control al alocării/dezalocării
- Permisii de acces
- Granularitate la nivel de pagină

VULNERABILITĂȚI ȘI ATACURI LA MEMORIA UNUI PROCES

Execuția codului într-un proces

- Zone de cod
- Instruction pointer
 - poziția curentă
 - incrementată cu dimensiunea unei instrucțiuni
- Fluxul se schimbă la branch-uri sau apeluri de funcții
- Poințieri de funcții

Bug-uri și vulnerabilități

- Ce este un bug?
- Când este un bug o vulnerabilitate?
- Ce obiective sunt în exploatarea unei vulnerabilități?

Tipuri de vulnerabilități la nivelul memoriei

- Suprascrierea datelor cu date de atac
 - suprascriere variabile (verificate în if)
 - suprascriere poințieri de funcții
- Alterarea fluxului de execuție
- Executarea de apeluri arbitrar
- Executarea de cod arbitrar (code injection)

exec("./bin/bash")

- Sau echivalentul system("./bin/bash")
- Obținerea unui shell într-o aplicație existentă
- Obiectivul inițial al unui atac
- Atacatorul se va folosi de vulnerabilități ale memoriei pentru a porni un shell
- Ulterior:
 - obținere informații confidențiale
 - denial of service
 - privilege escalation

STACK BUFFER OVERFLOW

Injectare de cod

- Punere de cod într-o zonă writable și executarea sa
- Să putem scrie cod
 - Folosim funcții de citire input (fgets, scanf)
 - să putem executa cod de acolo
 - zonă executabilită
 - jump la aceea adresă
- E nevoie de o zonă simultan writable + executable

Shellcode

- Instrucțiuni în cod binar
- Se scriu într-o zonă / buffer
- Se "sare" aici pentru execuție
- De obicei realizează exec("./bin/bash")
- Se încearcă injectarea acestuia într-o zonă accesibilă

Structura stivei (reminder)

- Adresa de return
- Fostul frame pointer
- Variabile locale (bufere incluse)
- Dacă un buffer este neîncăpător (overflow) putem suprascrie date, eventual chiar adresa de return

Stack buffer overflow

- Scrierea în buffer peste dimensiunea alocată
- De ce merge?
 - avem spațiu alocat pe stivă
- Ce suprascriem?
 - alte variabile
 - adresa de return
- Ce obținem
 - alterarea fluxului de execuție
 - rulare de cod arbitrar

Suprascrierea adresei de return

- Adresa pe care o va folosi apelatul la încheierea funcției
 - este adresa instrucțiunii următoare făjă de cea folosită de apelant în momentul apelului
- Se găsește pe stivă
- Stack buffer overflow poate conduce la suprascrierea adresei de return
- Se pune adresa shellcode-ului
- Shellcode-ul poate sta pe stivă

Funcții de lucru cu șiruri

- str*
- Împreună cu funcțiile de citire de intrare (fgets) sunt principalele funcții exploataabile
- NBTS (NUL -terminated byte strings)
 - dacă un șir nu e NUL-terminalat, putem suprascrie dincolo de dimensiunea sa
- Lungimea unui șir
 - trebuie sătă tot timpul
 - presupune legătura de lungimea șirului pot conduce la vulnerabilități de securitate

METODE OFENSIVE ȘI MECANISME DEFENSIVE

Injectare de cod

- Zone writable + executable
- Se poate plasa cod acolo și executa
- Acest cod se numește "shellcode"
- Clasic: pe stivă + stack buffer overflow
 - citire buffer de la standard input
 - variabilă de mediu

DEP

- Data execution prevention
- O zonă writable nu este executable
- W^X
- NX flag pe arhitecturile moderne
- Bypass: apel mprotect, VirtualProtect

return-to-libc

- Executarea unei funcții existente
- Clasic, system("./bin/bash")
- Nu este nevoie de injectare de cod
- Trebuie sătă adresa funcției

Canary value

- Plasată pe stivă între variabile locale și adresa de return
- Se verifică coerența informației la părăsirea funcției
- Previne efectul stack buffer overflows pentru a suprascrie adresa de return

Suprascriere de alte date

- Nu suprascriem adresa de return
 - Variabile locale alterează fluxul de lucru
 - Poințieri de funcții
 - Suprascriere poințieri de gestiune a heap -ului
- Randomizează plasarea bibliotecilor
 - dificil de găsit adresa funcțiilor de bibliotecă
 - dificil de realizat return-to-libc
- Randomizează plasarea stivei
 - adresa de return se suprascrie cu o adresă efectivă, nu cu o funcție cu jump relativ
 - adresa buffer-ului de pe stivă variază la fiecare rulare
 - dificil de sătă unde să sari pe stivă

CONCLUZII

Safe coding

- Atenție la funcții de lucru pe șiruri
- Folosii suport DEP, ASLR, canary value
- Analiză statică pentru verificarea de bug-uri sau vulnerabilități

Curs 8 - Fire de execuție

Cuprins

- Server de web
- Thread-uri
- API
- Implementare
- Sincronizare

Cum implementam un server de web?

- Cerinte:
 - Servesc un număr arbitrar de clienti
 - Fiecare client poate cere oricâte pagini (HTTP 1.1)
 - Menține statistică: nr total de pagini vizionate, nr total de clienti, octeti cititi etc.

Implementare securității

```
while(1){  
    int s = accept(ls);  
    fname = read_request(s);  
    while(fname){  
        read_and_send_file(fname);  
        update_stats();  
        fname = read_request(s);  
    }  
}
```

Probleme

- Un singur client simultan
- Ineficient chiar și cu un singur procesor

ALTERNATIVE

Folosind procese

```
while(1){
    int s = accept(ls);
    if(s==0){
        fname = read_request(s);
        while(fname){
            read_and_send_file(fname);
            update_stats();
            fname = read_request(s);
        }
    } else {...}
}
```

Probleme

- Cum actualizam statisticile?
- Cost mare pentru pornire proces

Implementare asincrona

```
while(1){
    int s = accept(ls);
    add_client(s);
    select(...);
    for(c:clients){
        if(FD_ISSET(c.s)){
            fname = read_request(s);
            c.d = open(fname, ...);
            c.status = read_file;
            //...
        }else if (FD_ISSET(c.d)){
            read(c.d, buf, 1000);
            send(c.s, buf, 1000);
            ...
        }
    }
}
```

Probleme

- Trebuie sa tinem stare pentru fiecare client
- Greu de implementat

Am dori o primitiva SO care:

- Executa secentual un set de instructiuni
- Este usor de pornit/oprit
- Partajeaza date cu usurinta

Folosind thread-uri

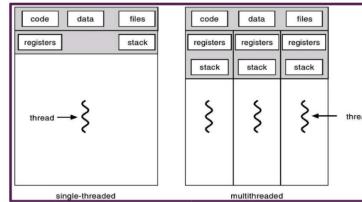
```
while(1){
    int s = accept(ls);
    pthread_create(&t, NULL, (void*)&cnt, (void*)&s);
}
```

...

```
void* cnt(void* p){
    int s = *(int*)p;
    Char fname = read_request(s);
    while(fname){
        read_and_send_file(fname);
        update_stats();
        fname = read_request(s);
    }
}
```

Fir de executie(thread)

- O secenta de control in cadrul unui proces
- Executa secentual un set de instructiuni
- Un proces are unul sau mai multe thread-uri care partajeaza resursele sale



Ce partajeaza threadurile?

- Variabile globale (.data, .bss)
- Fisiere deschise
- Spatiu de adresa
- Modul de tratare a semnalelor

Ce nu partajeaza threadurile?

- Registre
- Stiva
- Program counter/ instruction pointer
- Stare
- Masca de semnale
- TLS (Thread Local Storage)

Procese vs thread-uri

Procese	Thread-uri
Grupeaza resurse	Abstrakteaza executia
Fisiere, lucru retea	Stiva
Spatiu adresa	Registri
Fir de executie	Program counter

Avantajele thread-urilor

- Timp de creare mai mic decat al proceselor
- Timp mai mic de schimbare de context
- Partajare facila de informatie
- Utile chiar si pe uniprosesori

Dezavantajele thread-urilor

- Daca moare un thread, moare tot procesul
- Nu exista protectie la partajarea datelor
- Probleme de sincronizare
- Prea multe thread-uri afecteaza performanta!

Operatiuni cu thread-uri

- Lansarea in executie
- Incetarea executiei
- Terminarea fortaata (cancel)
- Asteptare (join)
- Planificare

Posix Threads

- Folosit pe sistemele Unix
- API pentru crearea si sincronizarea thread-urilor
- Folosire

- Inclus header-ul (#include <pthread.h>)
- Legarea bibliotecii (-lpthread)
- Max 5 pthreads

API PThreads

```
pthread_t tid;
pthread_create(&tid, NULL, threadfunc, (void*)arg);
pthread_exit(void* ret);
pthread_join(pthread_t tid, void** ret);
pthread_cancel(pthread_t tid);
```

Thread-uri in Linux

- Suport kernel pentru task-uri (struct task_struct)
- Procesele si thread-urile sunt task-uri
- Planificabile independent
- NPTL (New Posix Thread Library)
 - Implementare pthreads(1:1)
 - Foloseste apelul de sistem clone
 - Thread-urile sunt grupate in acelasi grup
 - Getpid intoarce thread group ID

Clone

- Specific Linux
- Folosit de fork si NPTL
- Diferite flag-uri specifice resursele partajate
 - CLONE_NEWNS
 - CLONE_FS, CLONE_VM, CLONE_FILES
 - CLONE_SIGHAND, CLONE_THREAD

Thread-uri in Windows

- Model hibrid: suport in kernel
- Fibre: fir de executie in user-mode
 - Planificate cooperativ
 - Blocarea unei fibre blocheaza firul de executie

API Windows

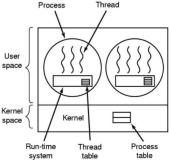
```
HANDLE CreateThread(...)
ExitThread
WaitForSingleObject/ MultipleObjects
GetExitCodeThread
TerminateThread
TlsAlloc
TlsGetValue/TlsSetValue
```

IMPLEMENTARE THREAD-URI

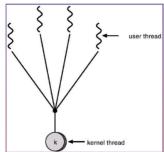
User-level

- O biblioteca de thread-uri ofera suport pentru crearea, planificarea si terminarea thread-urilor
- Mentine o tabela cu fire de executie: PC, registre, stare pentru fiecare fir
- Nucleul "vede" doar procese, nu si thread-uri
- Mai multe fire de executie sunt planificate cu un singur proces

Thread-uri implementate user-level



Mai multe fire de executie sunt mapate pe acelasi fir de executie din kernel



Avantaje

- Usor de integrat in SO:nu sunt necesare modificarile
- Pot oferi suport multithreaded pe un SO fara suport multithreaded
- Schimbare de context rapida: nu se executa apeluri de sistem in nucleu
- Aplicatiile pot implementa planificatoare in functie de necesitatii

Dezavantaje

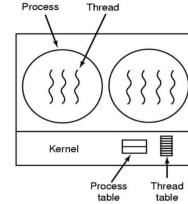
- Un apel de sistem blocant blocheaza intreg procesul
- Un page-fault blocheaza tot procesul
- Planificare cooperativa
- Multe aplicatii folosesc apeluri de sistem oricum

Kernel level threads

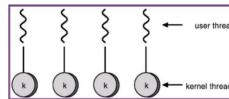
- Suport in kernel pentru creare, terminare si planificare
- Model unu-la-unu

Avantaje	Dezavantaje
Fara probleme la apeluri blocante sau page faults	Crearea si schimbarea de context e mai lenta
Pot fi planificate pe sisteme multiprocesor	

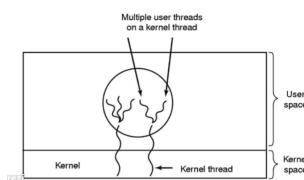
Implementare thread-uri in kernel



Un kernel thread pentru fiecare thread utilizator



Hibrid



Sincronizare

- Threadurile ofera acces la date comune
- Accesul trebuie mediat pentru a implementa programe corecte
- Chiar si programele cu un singur thread pot crea probleme
- Dar cele cu mai multe thread-uri?

Exemplu

```
int total_bytes;
void updateStatics(int j){
    total_bytes+=j;
}
Void signal_handler(){
    update_statistics(1);
}
```

Reentrantă

- O functie este reentrantă dacă poate fi executată simultan de mai multe ori, fără a afecta rezultatul
- Condiții necesare:
 - Nu lucrează cu variabile globale/statice
 - Apelează doar funcții reentrantă
- Reentrantă este importantă (mai ales) în programe cu un singur thread din cauza semnalelor

Reentrantă în practică

- Multe funcții din biblioteca setează variabila errno
- Depinde de implementare dacă acestea sunt reentrantă:
 - Anumite apeluri au veriuni reentrantă: gethostbyname_r
 - Activare cu macroul _REENTRANT

Thread safety

- O funcție este thread-safe dacă poate fi apelată din mai multe thread-uri în același timp
- Strategii implementare
 - Acces exclusiv
 - Semafoare
 - Monitorare
 - Thread-local storage
 - Reentrantă
 - Operări atomice

Curs 9 - Sincronizare

Cuprins

- Recapitulare thread-uri
- Necesa de sincronizare
- Sincronizarea la procese și thread-uri
- Moduri de sincronizare
- Implementarea sincronizării
- Problematica sincronizării

NEVOIA DE SINCRONIZARE

Necesa de sincronizare
unsigned long sum = 0;
void thread_func(size_t i){
 sum += i * i * i;
 printf("sum(%zu): %lu\n", i, sum);
}
int main(void){
 size_t i;
 for (i = 0; i < NUM_THREADS; i++)
 create_thread(thread_func, i);
 [...]
}

- Acces exclusiv / serializare / atomizare
 - Regiune critică
- Secvențiere / ordonare
 - Read after write, write after write, use after create
- Nedeterminism
 - Poate să meargă adesea deși sunt probleme

Interacțiune în sistemele de calcul

- Între componente hardware
- Între procese/thread-uri
- Între user space și kernel space
- Între aplicații și hardware
- Metode
 - Notificări
 - Transfer de date (e.g. message passing)
 - Date partajate (e.g. shared memory)

Exemple

1. Placa de rețea primește pachete pe care să le transfere procesorul sau DMA-ul în memorie
2. Două procesare au nevoie de acces la aceleși date din memoria RAM.
3. Un proces solicită sistemului de operare aducerea de date de pe disc.
4. Un proces așteaptă încheierea unei acțiuni a altui proces.
5. Mai multe thread-uri execută operații pe o structură de date comună (partajată).

Probleme

1. Placa de rețea primește mai multe informații decât poate procesorul prelucra; se pierd informații.
2. Ambii procesoare încearcă să modifice aceeași dată. Datele sunt incoerente/corupte.
3. Bufferul de memorie unde sistemul de operare stochează datele este suprascris cu alte date (cerute de alt proces).
4. Procesul " pierde" notificarea din partea celuilalt proces și așteaptă nedefinit.
5. Thread-urile modifică în mod concurrent structurile de date (vectori, liste, matrice, întregi, pointeri) și acum sunt incoerente/corupte.

Producător-consumator cu probleme

producer	consumer
<pre>wait(buffer_not_full); produce_item(); consume_item(); signal(buffer_not_empty);</pre>	<pre>wait(buffer_not_empty); consume_item(); signal(buffer_not_full);</pre>

Producător-consumator corect

producer	consumer
<pre>lock(mutex); if (is_buffer_full()) wait(buffer_not_full, mutex); produce_item(); consume_item(); signal(buffer_not_empty); unlock(mutex);</pre>	<pre>lock(mutex); if (is_buffer_empty()) wait(buffer_not_empty, mutex); consume_item(); signal(buffer_not_full); unlock(mutex);</pre>

Cod pe un server web

```
def add_quest_points(quest_level):
    points = get_current_player_points()
    points += points_for_level(quest_level)
    save_player_points(points)
    increase_player_quest_level()
def add_challenge_points(points):
    ... /* something similar to above */
```

Scenarii la codul de pe serverul web

- Se apelează simultan funcția `add_quest_points` și funcția `add_challenge_points`
 - Jucătorul rezolvă quest-ul și atunci generează apelul pentru challenge
 - Poate există quest-uri pe echipă și simultan doi jucători rezolvă părți din același quest (sau quest-uri diferite)
 - Jucătorul apăsa (conștient sau nu) de mai multe ori în browser pe butonul "Submit quest answer" (ajung două cereri identice la server). Care va fi noul nivel în quest al jucătorului? Ce punctaj va avea?
 - Serverul web poate fi multi-proces sau multi-thread. Sincronizarea se face diferit în acele cazuri.

Sincronizare

- Pot apărea probleme din interacțiune
 - Pierdere notificări, pierdere date
 - Date corupte, date incoerente
- Sincronizarea asigură
 - Comportament determinist
 - Date coerente
- Moduri
 - Secvențiere/ordonare operații (A după B)
 - Acces exclusiv la resurse (doar A sau doar B)

Sincronizarea prin secvențiere

- O acțiune înaintea altă acțiuni
- Exemplu: operația `wait/waitpid`
- Este nevoie ca un proces/thread să execute o acțiune și să producă un efect care să declanșeze acțiunea unui alt proces
- Exemplu: Thread-ul boss trebuie să producă "work item-ul" și să îl adauge în coadă înainte ca un thread worker să îl preia
- În absența sincronizării, se vor prelua date nevalide și programul va avea comportament arbitrar

Sincronizarea prin acces exclusiv

- Două sau mai multe procese/thread-uri au nevoie la aceleși date
- Exemplu: Mai multe thread-uri lucrează pe o matrice partajată
- Accesul exclusiv permite unui singur accesul
 - Serializare
 - Regiune critică
- Dacă au nevoie toate să citească nu e nevoie de sincronizare
- Dacă unul scrie și nevoie de sincronizare prin acces exclusiv
 - E posibil ca atunci când unul scrie altul să citească/parcurgă (de exemplu, într-o listă înăncuiată)
- În absența accesului exclusiv, cititorul va folosi structuri în stare incoerentă sau coruptă

SINCRONIZAREA LA PROCESE și THREAD-URI

Când folosim thread-uri

- Multi-core programming
 - CPU intensive: libx264 (encoding), comprimare, criptare
- În medii ce indică folosirea thread-urilor
 - Java
 - kernel development
 - OpenMP
- Pentru HPC (High Performance Computing)

Când nu folosim thread-uri

- Pe sisteme single core (dar există cazuri de utilizare și acolo)
- Atunci când nu suntem confortabili cu API-ul
- Atunci când avem programe seriale (nu paraleлизăm ce nu este nevoie)
- Atunci când însemnă foarte multă robustețea programului

Sincronizarea proceselor

- În general implicită
 - socket
 - message passing, message queues
- Explicită
 - mai rar primitive de sincronizare "clasice" (mutexuri, semafoare, file locks)
 - memorie partajată (acces exclusiv)
 - semnale (secvențiere)
 - API de lucru cu procese: `wait`

Sincronizarea thread-urilor

- În general explicită
- Date partajate, concurență: acces exclusiv
 - operații atomice
 - mutex-uri
 - spinlock-uri
- Operații secvențiale: ordonare operații
 - semafoare
 - variabile condiție
 - barriere

MODURI DE SINCRONIZARE

Date coerente și date corupte

- Datele coerente "au sens", determină comportamentul determinist pentru aplicație
- Interacțiunea necorespunzătoare (nesincronizată) produce date incoerente
- Exemplu: `read before write`

- Dacă un pointer a fost citit dar neinitializat ...

• Nevoie de sincronizare

Sincronizare implicită

- Primitivele folosite nu au ca obiectiv sincronizarea
 - Sincronizarea este un efect secundar (implicită)
- Transfer de date/mesaje: socketi, pipe-uri, cozi de mesaje, MPI
- Date separate/partiționate
- Avantajos: ușor de folosit de programator

Transferul datelor

- Metode: `read/write`
 - `read`: în general blocant, așteaptă date scrise
 - `write`: poate fi blocant dacă este buffer plin

• Sincronizare implicită

- nu există date comune, datele sunt copiate/duplicate
- secvențiere, ordonare: `read-after-write`
- E nevoie de sincronizare explicită pentru mai mulți transmițători sau receptori
- Avantaje: sincronizare implicită, ușor de programat, sisteme distribuite
- Dezavantaje: overhead de comunicare, blocarea operațiilor

Partiționarea datelor

- Se împart datele de prelucrat
- Fiecare entitate lucrează pe date proprii
- În mod ideal nu există comunicare între entități
 - realist, e nevoie de comunicare (cel puțin agregarea datelor finale)
- Avantaje: reducerea zonelor de sincronizare, programare simplă
- Dezavantaje: date duplicate, posibil overhead de partiționare și agregare, aplicabilitate redusă

Sincronizare explicită

- Referită doar ca "sincronizare"
- Metode/mecanisme specifice de sincronizare
- Două forme de sincronizare
 - pentru date partajate: primitive de acces exclusiv
 - pentru operații secvențiale: primitive de secvențiere, ordonare
- Folosită atunci când este nevoie
- Menține coerența datelor

Acces exclusiv

- În cazul datelor partajate
- Thread-urile/procesele concurează la accesul la date
 - pentru operații secvențiale: primitive de secvențiere, ordonare
- Folosită atunci când este nevoie
- Menține coerența datelor

- Acces exclusiv
 - delimitarea unei zone în care un singur thread are acces
 - regiune critică (critical section)
 - e vorba de date, nu cod
- Metode: lock și unlock, acquire și release
- Primitive: operații atomice, mutex-uri, spinlock-uri
- Avantaje: date coerente
- Dezavantaj: overhead de așteptare, cod serializat

Atomicitate

- Accesul la date să fie realizat de un singur thread
- Accesul exclusiv se referă la metoda
- Atomicitatea este o condiție a zonei/datei protejate
- Datele/zonile neutronice partajate sunt susceptibile la condiții de cursă
- Atomicitate: variabile atomice, primitive de acces exclusiv

Secvențierea operațiilor

- În cazul în care operațiile trebuie ordonate
- Un thread scrie, un alt thread citește
 - e nevoie ca un thread să aștepte după altul
- Un thread produce, alt thread consumă
- Secvențiere
 - un thread B se blochează în așteptarea unui eveniment produs de un thread A
 - thread-ul B execută acțiunea după thread-ul A
- Metode: signal/notify și wait
- Primitive: semafoare, variabile condiție, bariere
- Avantaje: comportament determinist
- Dezavantaj: overhead de așteptare

Folosire acces exclusiv

- Oricând avem date partajate
- Garantăm că un singur thread poate avea acces la acele date
- Codul este serializat (probleme de paralelism)
- Între lock și unlock definim o regiune critică (serială)
- Pe cât posibil folosim operații atomice (rapide)
- Dacă regiunea critică este mică folosim spinlock-uri
- Dacă regiunea critică este mai mare folosim mutexuri

Folosire secvențiere

- În general în probleme care se reduc la producător-consumator
 - Un thread "produce" informații/date/structuri/pachete, iar altul "consumă"
- De obicei există un obiect de sincronizare și o variabilă (flag) care indică producerea informației

- Producătorul produce, pune flag-ul pe activ și notifică obiectul de sincronizare
- Consumatorul verifică flag-ul, dacă este inactiv așteaptă la obiectul de sincronizare până când este notificat și apoi consumă informația

IMPLEMENTAREA SINCRONIZĂRII

Cadrul pentru sincronizare

- Mai multe entități active (procese/thread-uri)
- Puncte de interacționare (date comune, evenimente)

Preemptivitate

– O entitate activă poate fi întreruptă oricând de o altă entitate activă

- oricând = atunci când vine o întrerupere de ceas, se invocă planificatorul care poate decide trecerea entității active curente din RUNNING în READY și planificarea altăia

Implementare simplă

- Un singur proces/thread (doar pentru medii specializate)
- Date complet partionate (izolare)
- Dezactivarea preemptivității
 - configurație planificator
 - dezactivarea întreruperilor (întreruperii de ceas)
 - doar pentru sisteme uniprocesor
- Folosirea de operații atomice

Implementare mutex

- Un boolean pentru starea internă (locked/unlocked)
- O coadă cu thread-urile care așteaptă eliberarea mutexului

```
struct mutex {
    bool state;
    queue_t *queue;
};
```

```
void lock(struct mutex *m)
{
    while (m->state == LOCKED) {
        add_to_queue(m->queue);
        wait();
    }
    m->state = LOCKED;
}

void unlock(struct mutex *m)
{
    m->state = UNLOCKED;
    t = remove_from_queue(m->queue);
    wake_up(t);
}
```

42

Nevoia de spinlock

- Funcțiile lock și unlock pe mutex trebuie să fie atomice
- Dezactivarea preemptivității
- Folosirea unui spinlock
- Spinlock-ul este o variabilă simplă cu acces atomic
- Operațiile folosesc busy waiting

– așteptare nedefinită (pierderea notificării)

- Implementare inefficientă
 - granularitate regiune critică
 - cod serial
 - thundering herd
 - lock contention
- Probleme implicite
 - overhead de așteptare
 - overhead de apel

Deadlock

- Deadly embrace
- Două sau mai multe thread-uri așteaptă simultan unul după altul
 - A obține lock-ul L1 și așteaptă după lock-ul L2
 - B obține lock-ul L2 și așteaptă după lock-ul L1
- Lock-urile trebuie luate în ordine ca să prevină deadlock
- Forma de deadlock cu spinlock-uri: livelock

Așteptare nedefinită

- Așteaptă un eveniment
- Nu produce evenimentul (sau A pierde notificarea)
- Așteaptă nedefinit (evenimentul nu se produce)

Granularitatea regiunii critice

- Regiune critică mare
 - mult cod serial, paralelism redus
 - legea lui Amdahl
- Regiune critică mică
 - overhead semnificativ de lock și unlock față de lucru efectiv în zonă
 - lock contention (încărcare pe lock)

Thundering herd

- În momentul apelului unlock() sau notify() sunt trezite toate thread-urile
- Toate thread-urile încearcă achiziționarea lock-ului
- Doar unul reușește, restul se blochează
- Operația este reluată la următorul apel unlock() sau notify()

Overhead de apel

- Apelurile de lock, unlock, wait, notify sunt costisitoare
- În general înseamnă apel de sistem
- De multe ori invocă planificatorul
- Mai multe apeluri, mai mult overhead
- Lock contention generează mai mult overhead

• Pentru atomicitate: suport hardware

- TSL (test and set lock)
- cmpxchg (compare and exchange)

Implementare spinlock

- Un boolean/intreg cu acces atomic
- Atomic compare and exchange pentru lock

```
int atomic_cmpxchq(int val, int compare, int replace)
{
    /* This is an equivalent implementation. */
    /* The entire operation is atomic. */
    int init = val;
    if (val == compare)
        val = replace;
    return init;
}

void lock(struct spinlock *s)
{
    while (atomic_cmpxchq(s->val, 1, 0) == 0)
        ; /* do nothing */
}

void unlock(struct spinlock *s)
{
    atomic_set(s->val, 1);
}
```

0 - locked
1 - unlocked

Spinlock vs. mutex

Spinlock	Mutex
<ul style="list-style-type: none"> • Busy-waiting • Simplu • Pentru regiuni critice scurte 	<ul style="list-style-type: none"> • Blocant • Coadă de așteptare • Pentru regiuni critice scurte • Pentru regiuni critice mari sau în care thread-urile se blochează

PROBLEMATICA SINCRONIZĂRII

Dacă nu folosim sincronizare ...

- Race conditions
 - read-before-write
 - write-after-write
 - Time Of Check To Time Of Use
- Date corupte, incoerente
- Comportament nedeterminist, arbitrar
- Procesul/thread-ul poate provoca un crash (acces nevalid la memorie)

Dacă folosim sincronizare ...

- Implementare incorectă
 - deadlock

- De preferat, unde se poate, operații atomice

Curs 10 - Dispozitive de intrare/iesire

CONCLUZII

- Când folosim sincronizare explicită
 - Când e neapărat nevoie
 - În cazul în care avem date comune (partajate)
 - În cazul în care avem nevoie de secvențiere de operații
 - Nu am putut separa/partaja datele • Nu am putut folosi transfer de date/mesaje (sincronizare implicită)
- Sincronizarea este un rău necesar
 - rău: overhead, regiuni seriale, deadlock la folosire necorespunzătoare
 - necesar: fără sincronizare obținem date corupte, comportament arbitrar, crash-uri, vulnerabilități de securitate

Dacă folosim sincronizare explicită ...

- Ne referim la date (nu la cod)
- Gădim bine soluția
- Aveam grija la dimensiunea regiunilor critice
- Folosim variabile atomice unde se poate
- Pentru regiuni critice mici folosim spinlock-uri
- Pentru regiuni critice mari folosim mutexuri
- Nu folosim mai multe thread-uri decât avem nevoie
 - pool de thread-uri
 - modelul boss/workers
 - modelul worker threads

Dispozitive de intrare/iesire

```
Int fd = open/schet/pipe...;
read(fd, buf, 10);
write(fd2, buf, 10);
close(fd);
```

Dispozitivele de I/O sunt foarte diverse

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek rate transfer rate delay between operations	
I/O direction	read-only write-only read-write	CD-ROM graphics controller disk

Ce vor utilizatori?

- Usurința în utilizare:
 - Cod care rulează pe oricâte dispozitive de intrare-iesire
 - Să fie ușor să numească dispozitivele
 - Tratarea erorilor să se întâmple automat
- Performanță
- Operări dorite:
 - Sincrone (majoritatea)
 - asincrone (unele aplicații)

CUM IMPLEMENTAM?

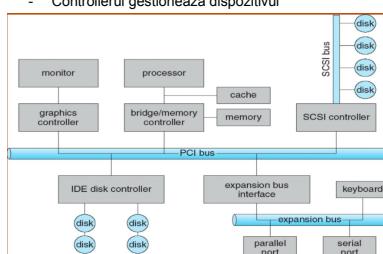
Hardware

- Două categorii de dispozitive: caracter și bloc
- Fiecare dispozitiv are
 - Un controller
 - Unul sau mai multe porturi
 - Conectat la o magistrală
- Procesorul da comenzi I/O controllerului
- Controllerul gestionează dispozitivul
- 2 categorii de dispozitive: caracter și bloc
- Fiecare dispozitiv are:

- Un controller
- Unul sau mai multe porturi
- Conectat la o magistrală

Procesorul da comenzi de I/O controllerului

Controllerul gestionează dispozitivul



Comunicația cu dispozitivele de I/E

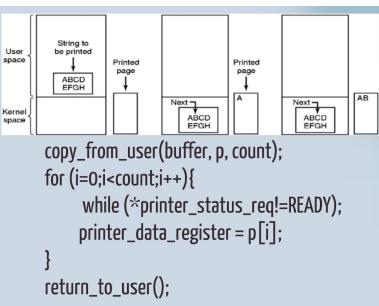
- Fiecare controller are:
 - Registre de stare, control, intrare, ieșire
 - (optional) buffer pentru date

Port mapped I/O

- Spatiu de adresă separat pentru dispozitivele de I/O
- O adresa de I/O se numește port
- Un port adresează un registru al controllerului
- Instrucțiuni specializate:
 - IN REG, 0x2F8
 - OUT REG, 0x2FA

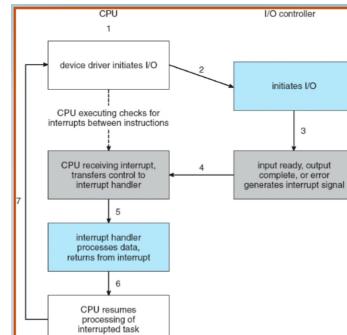
Memory mapped I/O

- Registrele I/O sunt mapate în spațiul de memorie
- Avantaje:
 - Nu necesita instrucțiuni specializate
 - Protecția de la memorie virtuală
 - Instrucțiunile pot referi memorie sau registre
- Dezavantaje:
 - Trebue inhibat cache-ul la nivel de pagina
 - Bridge-uri între magistrale -> mai lent decât port-mapped I/O



Intreruperi

- Anunță procesorul că s-a întâmplat un eveniment care trebuie tratat
- Pot fi mascabile sau nemascabile
- Sunt generate de software sau hardware (controller)
- Folosesc liniile de interrupție (IRQ line)
- Există o tabel cu rutine de tratare a interrupțiilor
- Procesorul rulează rutina de tratare a interrupției (ISR)

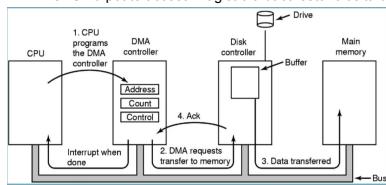


Implementare cu interruperi

```
copy_from_user(buffer, p, count); if (count==0) {
enable_interrupts();
while (*printer_status_reg!=READY);
*pinter_data_register = p[0];
scheduler();
}
*printer_data_register = p[i];
count -= i++;
}
acknowledge_interrupt();
return_from_interrupt();
```

Direct Memory Access

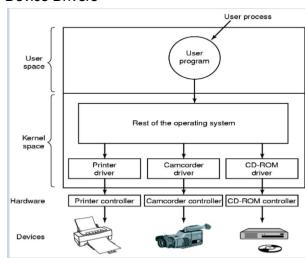
- DMA: dispozitiv separat specializat pentru copieri din memorie in memorie
- CPU comanda DMA sa efectueze transferuri date ca (sursa, destinatie, octeti)
- CPU nu poate accesa magistrala daca este folosita de DMA



Implementare cu DMA

```
copy_from_user(buffer,p,count); acknowledge_interrupt();
set_up_DMA_controller();
scheduler();
unlock_user();
return_from_interrupt();
```

Device Drivers

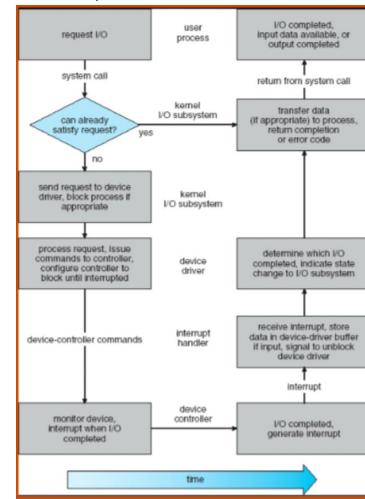


- Controleaza un dispozitiv sau clasa de dispozitive
- Ruleaza in kernel
- SO ofera interfata comună pentru drivere

Operatii independente de dispozitiv

- Planificare pentru eficiență și echitate
 - Sortare și comasare de cereri de I/O
- Buffering
 - Compensarea vitezei de transfer dintre dispozitive
 - Reasamblarea datelor
 - Semantica de copiere pentru utilizatori
- Caching: memoria ține copii ale datelor
- Spooling: se menține un buffer cu datele transmise unui dispozitiv
- Rezervarea dispozitivului

Fluxul unei operatii de I/O



API Utilizator

Char Devices	Block Devices	Network
Ex: mouse, tastatura	Ex: discuri, CDROM	Ex: 802.11, 802.5
Acces secvențial	Acces aleator	Separare protocol de interfata
Operatii: put, get	Operatii: read, write, seek	sockets
Transfer la nivel de caracter	Transfer la nivel de bloc	Operatii: connect, send, recv, close
Viteza redusa	Viteza ridicata	Viteza ridicata

Asynchronous I/O

- Unix: aio_read, aio_write, aio_suspend
- Windows: ReadFile, WriteFile, OVERLAPPED I/O

Initializare

- Dispozitive block/caracter: open, close
- Dispozitive retea: socket, connect, shutdown

Control

- Controleaza dispozitivul de I/O
- Depinde de dispozitiv!
 - ioctl/ DeviceIOControl
 - Setsocket, getsocket

Tipuri de operatii I/O

Blocante	Ne-blocante	Asincrone
<ul style="list-style-type: none"> - Procesul este suspendat pana la incheierea operatiei - Simplu de folosit 	<ul style="list-style-type: none"> - Operatia se intoarce imediat - Procesul primeste datele disponibile 	<ul style="list-style-type: none"> - Procesul ruleaza in paralel cu operatia - Este notificat atunci cand operatia este finalizata - Event-driven programming

Tipuri de operatii I/O

	blocante	ne-blocante
syncrone	read/write ReadFile, WriteFile	read/write O_NONBLOCK
asincrone	select	AIO aio_*, overlapped I/O