ITBI Tutoriatele 2-3

Albert Simon, Spiridon Mihnea-Andrei

October 29, 2025

1 Organizarea sistemului de operare

Discutam putin despre modul in care este organizat sistemul de operare. Cele mai importante concepte sunt cel de **fisier** si de **proces**.

Sa ne amintim ca fisierele si procesele nu sunt nimic mai mult decat **abstractii**, constructii oferite de sistemul de operare ca noi sa putem rationa mai bine si ca sa utilizam calculatorul mai usor. In realitate, un fisier "este" o secventa de biti pe hard disk, iar un proces "este" o secventa de instructiuni pe care le executa un CPU.

Totusi, noi nu controlam direct bitii si instructiunile, ci vorbim cu sistemul de operare (de exemplu, prin shell) si controlam in schimb fisiere si procese prin anumite operatii specializate (*cp*, *mv*, *mkdir*, *touch*, *kill*, etc.)

1.1 Fisierele

1.1.1 Ce sunt fisierele?

In Unix, toate datele din memoria secundara (de pe hard) si chiar si unele informatii despre activitatile care se petrec in memoria principala sunt reprezentate prin notiunea de fisier. Cand ne gandim la un fisier ne vin prima oara in minte acele lucruri pe care le vedem si pe care putem sa dam dublu-clic, *main.cpp* sau $file.txt$. Intr-adevar, acelea sunt fisiere, insa in Unix mai sunt si alte lucruri care sunt fisiere desi poate initial nu ne-am astepta:

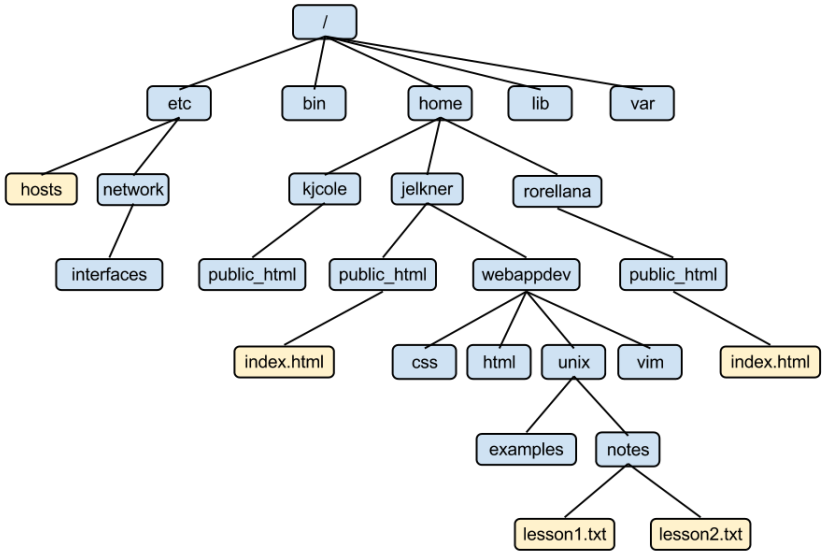
- Directoarele sunt fisiere. Este posibil sa fie putin contraintuitiva ideea deoarece suntem obisnuiti sa grupam fisierele obisnuite in foldere, pe care le vedem ca fiind distincte, insa sistemul de operare le reprezinta la fel. Un director e doar un fisier cu proprietatea ca are in el alte fisiere.
- Unele canale de comunicatie intre procese sunt reprezentate ca fisiere. Ne putem imagina o coada de mesaje. Un proces scrie in coada, iar altul citeste din coada. Canalul de comunicatie se cheama un **pipe** sau un **FIFO** (dupa principiul de functionare al cozii, first in, first out). Acest pipe poate sa poarte un nume si sa fie un fisier. Creeati un pipe cu numele *pipe*! Va puteti folosi de comanda *mknod pipe p*.
- Exista de asemenea notiunea de "link" in Unix. Daca va e familiar, in Windows echivalentul de "shortcut", si este pur si simplu un fisier care face trimitere catre un alt fisier, aflat la alta locatie. Scurtatura in sine ocupa spatiu si este considerata un fisier.

Toate acestea sunt fisiere. Fiecare are un comportament special, ele pot sa fie "fisier care tine alte fisiere", "fisier care functioneaza ca o coada" sau "fisier care trimite catre alt fisier", insa pentru sistemul de operare ele sunt **reprezentate intern la fel**.

1.1.2 Cum sunt organizate fisierele?

Structura lor este de arbore, avand intotdeauna radacina /. Directoarele sunt noduri de arbore aflate oriunde, iar fisierele obisnuite pot sa fie numai frunze.

Observati ca nu ne aflam in situatia sa avem *padure* precum in alte sisteme de operare (spre exemplu, in Windows avem C:\, D:\). Daca introducem, de exemplu, un USB sau un CD in calculator el va aparea in acest arbore unic, nu in alt arbore. Sistemul de operare se va ocupa, in general, automat, si va face continuturile dispozitivului accesibile in */mnt* sau */media* (de obicei).



1.1.3 Cum accesam fisierele?

Exista trei operatii fundamentale pe care le putem face cu un fisier:

- Sa citim date din el
- Sa scriem date in el
- Sa il lansam in executie, daca este un program

Aceste operatii sunt codificate cu *rwx*, pentru **R**ead, **W**rite si e**X**ecute.

Se pune insa problema daca avem voie sa facem aceste operatii. Trebuie sa nu uitam ca lucram intr-un context multi-utilizator, poate nu acasa, pe calculatorul personal, dar in cadrul unei echipe, organizatii, companii etc. ne vom confrunta cu urmatoarea problema: cine ce permisiuni are?

Pentru a ne asigura ca nu avem probleme cauzate de accesul haotic la fisiere s-a introdus notiunea de **utilizator**. Un utilizator este tot o **abstractie** oferita de sistemul de operare. In realitate un utilizator e un **proprietar** (detinator, stapan) **de procese si fisiere**. Il putem privi ca pe un set de drepturi de acces la anumite fisiere, alaturi de diverse date (nume de utilizator, parola, directorul $HOME, etc.)

In fiecare fisier permisiunile *rwx* sunt trecute de trei ori: odata pentru **U**ser-ul care detine fisierul (de obicei creatorul sau), odata pentru **G**roup-ul de care apartine fisierul si inca odata pentru **O**ther, toti ceilalti.

Fiecare dintre aceste permisiuni are asociata o valoare numerica (4 pentru read, 2 pentru write, 1 pentru execute). Suma acestor valori e o cifra in octal (baza 8). Permisiunile in sine sunt fiecare in parte unul dintre cei biti (in baza 2) care alcatuiesc cifra octala.

Comenzi de control al utilizatorilor si ai accesului lor la fisiere:

- *chmod* schimba bitii de permisiune de mai sus, setandu-i la o valoare in octal primita ca argument direct, sau actualizandu-i daca se ofera un string de tipul "$o + rw$ = pentru Other, acorda permisiunile de Read si Write sau "$g - x$" = pentru Group, revoca permisiunea de eXecute.
- *chown* schimba ce utilizator si ce grup detin un anumit fisier.
- *useradd*, *usermod*, *userdel* controleaza utilizatorii.
- *passwd* schimba parola utilizatorilor.

Sfat: Nu va chinuiti sa retineti sintaxa la absolut fiecare comanda, ci important e sa stiti ca exista, principiul de functionare din spate, si unde sa o gasiti cand aveti nevoie.

1.2 Procesele

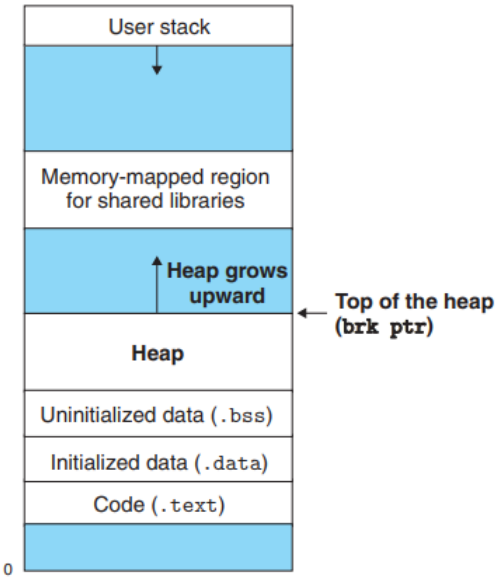
1.2.1 Ce e un proces?

Trebuie lamurita foarte clar distinctia dintre *program* si *proces*. Un program e un fisier care contine instructiuni pe care CPU-ul le poate intelege (apropo: il identificam dupa prezenta lui x in permisiuni). Un program nu "ruleaza" sau "are memorie". Un program e doar o lista de instructiuni Assembly, programul = lista in sine.

Un proces este ceea ce obtinem cand luam acea lista si o rulam efectiv. Despre proces spunem ca e "imaginea unui program in memoria principala". Un proces este **abstractia** care reprezinta totalitatea starii executiei unui program.

Cand trecem de la program la proces, procesul primeste:

- Un PID (process ID) unic
- O portiune din memoria RAM, pentru variabile si stiva de apel (amininiti-va de ASC!)
- File descriptori, prin care interactioneaza cu datele de pe hard
- Niste timp in care sa execute efectiv (nu uitati ca timpul e o resursa, pe care SO trebuie sa o gestioneze!)
- Un UID si GID, preluate de la persoana care a lansat in executie programul ca sa creeze procesul

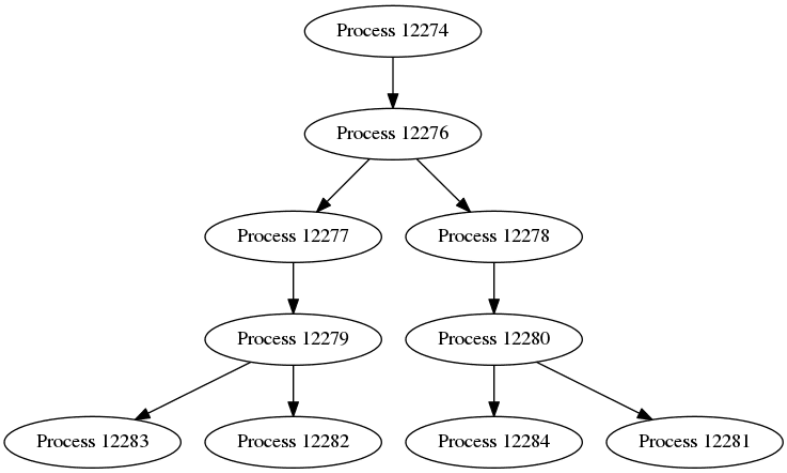


Exemplu: Un program, de exemplu *ls*, este unul singur. Insa, acelasi program poate sa fie rulat simultan de mai multi utilizatori, in mai multe procese, si fiecare proces poate sa se afle in stari diferite. Poate utilizatorul *Mihnea* a cerut sa faca *ls* intr-un director cu 10 fisiere si poate in acelasi timp utilizatorul *Simon* a cerut *ls* intr-un director cu 20 de fisiere si a specificat si flagurile $-la$. Programul este acelasi, executia si rezultatul difera, avem doua procese diferite, fiecare cu PID-ul, variabilele, etc. proprii.

1.2.2 Cum sunt organizate procesele?

La fel ca directoarele, toate procesele au parinte! Asta inseamna ca ne aflam din nou intr-un arbore, numai ca de data aceasta este un arbore al proceselor, nu al fisierelor.

Daca radacina arborelui de directoare era /, atunci mentionam ca radacina arborelui de procese este acel proces cu $PID = 1$, anume acel *init* care se pornea primul cand deschideam calculatorul. Toate procesele il au, daca nu ca parinte direct, macar ca stramos comun, pe *init*. Atentie, nici aici nu avem padure, doar arbore.

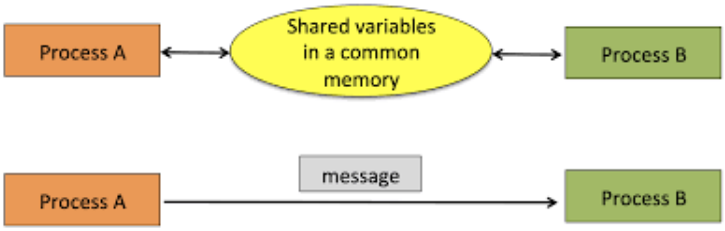


1.2.3 Comunicarea interproces

Procesele sunt utile pentru ca prin intermediul lor ne putem sa ne punem calculatorul "la treaba" si sa obtinem ce rezultat ne dorim. Totusi trebuie sa stim despre cum comunica procesele intre ele, si cum putem noi sa comunicam cu ele.

Intre procese exista doua paradigme de comunicare. Acestea sunt:

- memory sharing
- message passing



In primul caz procesele, amandoua, cunosc o zona de memorie partajata in care pot sa scrie si sa citeasca ambele. Evident, trebuie sa o faca intr-un mod organizat, daca cele doua nu sunt programate corespunzator in acea zona comuna ar fi haos. A doua metoda este mai structurata. In loc sa aiba o zona de memorie comuna, de data aceasta se implica kernel-ul sistemului de operare, care e responsabil de a trimite mesaje dintr-un proces in alt proces (gestionand, intern, coada de metical bar latex tsaje).

Exemplu: Sa zicem ca procesul 1848 vrea sa-i trimita procesului 1625 un mesaj in paradigma message-passing. Undeva in cod, 1848 va face un apel de tipul send(mesaj, 1625). In codul sau, 1625 poate sa faca receive(mesaj, 1848) ca sa primeasca mesajul lui 1848.

Exemplu: Procesul 1848 vrea sa transmita un mesaj catre 1625 printr-un pipe numit *mypipe*. Atunci, 1848 face send(mesaj, *mypipe*) si 1625 face receive(mesaj, *mypipe*). In acest caz procesele nu isi cunosc identitatea unul altuia, dar au o casuta postala prin care sa comunice. Paradigma este tot message-passing, numai ca este anonima.

Incercati: Semnul de | din shell este un pipe, creeaza un pipe anonim intre doua procese. Primul isi scrie output-ul in pipe, iar al doilea isi ia input-ul din pipe. Procesele ruleaza simultan, numai ca cel de-al doilea mai e nevoit ocazional sa-l astepte pe primul. Rulati:

- echo "Hello, world!" | grep "o"

  Output-ul lui echo va fi transmis prin pipe ca input la grep, care va gasi toate literele "o" din sirul "Hello, world!".

- mknod pipe p
  echo "Hello, world!" > pipe &
  cat pipe

  Folosind pipe-ul numit *pipe*, echo lasa mesajul "Hello, world!" in pipe. Ulterior, cat vine si citeste acel mesaj, afisandu-l pe ecran.

1.2.4 Semnale

Ati primit vreodata o notificare pe telefon care v-a determinat sa faceti ceva? Un mesaj de la seful de grupa legat de un deadline, de exemplu, va determina sa va apucati de tema.

Procesele din Unix pot si ele sa primeasca "notificari", si, in mod similar pot fi programate sa faca ceva ca urmare a primirii acelei notificari. Aceste notificari poarta numele de "semnal".

Exemplu: Ati mai intalnit semnale pana acum. Daca ati avut o eroare pe pbInfo pana acum, foarte probabil sa fi primit mesajul de eroare "killed by signal 11". Comanda *kill* ne ajuta sa trimitem semnale, in general (nu doar semnalul SIGKILL). Rulati comanda $kill\ -l$. Cine este semnalul 11? Stiind ca SIGSEGV inseamna "eroare de segmentare", adica ca probabil ati accesat memorie invalida (e.g. depasit vectorul in C++), va puteti explica eroarea de pe pbInfo?

Ce trebuie sa retinem despre semnale?

- Comanda *kill* pentru a le trimite
- Sunt notificari pe care le primesc procesele
- Procesele pot sa aiba un raspuns la aceste notificari (e.g. ruleaza o functie, se inchid, scriu ceva pe ecran) sau pot sa le ignore

2 Probleme

- *Recapitulativ.* Am stabilit ca shell-ul da programelor un context folosind variabile de mediu. Putem sa facem un program, noi, care le listeaza pe toate? (Solutia in sol2-1.cpp)
- *Recapitulativ.* Ce erau flag-urile? Enumerati cateva flag-uri des intalnite ale comenzilor cunoscute de voi. Cum aflam toate flag-urile unei comenzi? (*man* sau cu flag-ul lung $--help$)
- Ce sunt fisierele in Unix, ce tipuri speciale exista?
- Ce are un proces si nu are un program?
- Ce paradigme de comunicare interproces cunoasteti?
- Alte exercitii/intrebari de la laborator, din afara laboratorului, sau curiozitati.