Curs 02: Interfața sistemului de fișiere

Sumar curs anterior
sistem de operare: roluri, definiții
de ce să știm sisteme de operare, low-level, de ce să fim full stack: performanță, securitate, depanare/troubleshooting
nevoia de sistem de operare: portabilitate, izolare/securitate, planificarea accesului la resurse
probleme cu sistemul de operare: risc de securitate (TCB crescut), overhead
user/application space vs kernel/supervisor space, context privilegiat/context neprivilegiat
apel de sistem (pentru accesul la codul sistemului de operare): protejare/securitate vs. overhead
microkernel vs kernel monolitic: securitate/izolare vs performanță
Date
preluate date de la intrare și stocare
prelucrare date stocate
generare date și stocare

= sistem de fișiere

datele sunt stocate ca fișiere
sf este organizarea datelor ca fișiere, de obicei pe disc (suport persistent)
persistență, organizare/separare, eficiență spațială/temporală, partajare
+ diagramă: aplicații (utilizare) ---> sistem de fișiere (organizare) ---> date (suport persistent)
Fișiere
fișierul este o secvență de octeți (byte stream) stocați pe disc
fișiere binare; text cele pentru care octeții sunt citibili (human readable)
fișierele sunt formate din date și metadate (atribute ale fișierelor), ambele stocate pe suport persistent
metadatele unui fișier se găsesc într-o structură numită FCB (File Control Block), (ex. inode pe Unix)
+ diagramă: date + metadate (FCB), atribute în metadate
metadate: nume (șir de caractere), identificator (de ex. inode number), dimensiune, user (owner), group, permisiuni, timpi de acces
comanda stat în Unix (și apelul stat(2)/fstat(2)) afișează metadatele unui fișier
+ demo cu stat pe fișiere
Fișier deschis
pentru ca un fișier să fie folosit trebuie să fie deschis
apeluri precum open(2) (Linux)/fopen(3) (standard C) deschid un fișier
un fișier este deschis dintr-un proces
deschiderea unui fișier duce la crearea unei structuri de fișier deschis; structura este în memorie în kernel/supervisor space în urma apelului de sistem de creare a unui fișier (ex open(2) pe Linux, CreateFile pe Windows)
+ diagramă: proces ---> apel de sistem de deschidere ---> structură de fișier deschis ---> structură de fișier pe disc (FCB + date)
structura de fișier deschis conține: referință către FCB-ul fișierului, permisiuni de deschidere, cursor/pointer de fișier (file pointer/cursor)
+ diagramă continuată: adăugare permisiuni de deschidere și cursor/pointer de fișier în structura de fișier deschis
Deschiderea unui fișier
permisiunile de deschidere sunt un subset al permisiunilor FCB-ului: dacă un fișier are permisiuni de FCB de citire (pentru utilizatorul procesului care face operația) atunci permisiunile de deschidere pot fi doar de citire, nu de scriere
cursorul de fișier este plasat pe valoarea 0 sau pe dimensiunea fișierul (din FCB) depinzând de opțiunile de deschidere
+ demo cu apeluri open:
open(..., O_RDWR); // cursorul pus pe 0
open(..., O_RDWR | O_APPEND); // cursorul pus pe dimensiunea fișierului (din FCB)
open(..., O_RDWR | O_TRUNC); // cursorul pus pe 0, dimensiunea pusă pe 0, datele fișierului "șterse" (marcate ca fiind blocuri libere)
+ demo: folosim strace pentru a vedea ce efect au apelurile ANSI fopen(..., "r"); fopen(..., "w"); fopen(..., "rw"); fopen(..., "r+"); fopen(..., "w+"); fopen(..., "a"); fopen(..., "a+");
apelurile de deschidere a unui fișier întorc un handle de gestiune a fișierului, folosit de operații de citire și scriere
la nivelul cel mai de jos, handle-ul este un număr, numit descriptorul de fișier, asta întoarce apelul open(2)
Crearea unui fișier
atunci când deschidem un fișier și acesta nu există, poate fi creat
pentru a fi creat trebuie să:
* avem permisiuni de scriere în directorul în care va fi creat
* avem opțiunea O_CREAT sau echivalentă la apelul de deschidere
când este creat un fișier, nu are date și se inițializează FCB-ul acestuia:
* numele este dat ca parametru
* user/group este al procesului care a efectuat apelul de sistem
* timpii de acces sunt inițializați la timpul curent
* dimensiunea este 0
* permisiunile (numite și permisiuni de creare) sunt date ca al treilea parametru: atenție, e o greșeală frecventă omiterea permisiunilor
+ demo cu open(2) pentru creare fișier cu permisiunile de creare bine transmise, în octal
Descriptorul de fișier, tabela descriptorilor de fișiere
descriptorul este un index într-o tabelă numită tabela descriptorilor de fișiere (file descriptor table), reținută în kernel/supervisor space
există câte o tabelă de descriptori de fișiere per proces
o intrare în tabela descriptorilor de fișiere este un pointer către structura de fișier deschis
+ diagramă completă cu descriptor de fișier (user/application space) ---> tabelă de descriptori de fișiere ---> structură de fișier deschis ---> structură de fișier pe disc (ultimele trei în kernel space)
o intrare în tabelă este validată/inițializată în momentul deschiderii unui fișier (open(2)/fopen(3)); este invalidată în momentul închiderii unui fișier (close(2)/fclose(3))
+ demo: scris la descriptorul 3 folosind write(2), descriptorul 3 fiind nevalid, afișat eroare (errno + perror)
intrările în tabelă pot referi și altceva în afară de fișiere și pot fi create și altfel: socketi (creare intrare cu socket(2), închidere folosind close(2)), pipe-uri (creare cu pipe(2), închidere cu close(2)), terminale; în tabelă se găsesc pointeri către structuri specifice de socket sau pipe sau terminal (nu structuri de fișier deschis)
dimensiunea tabelei este limitată pentru a preveni abuzuri de prea multe fișiere deschise
+ demo cu getfdtablesize(2)
descriptorii 0, 1 și 2 sunt descriptorii standard și referă respectiv, standard input, standard output și standard error (uzual referă structuri de terminal)
+ demo cu un proces sleep și afișarea descriptorilor standard cu lsof, "numele" fișierului este numele dispozitivului de terminal (/dev/pty/0 sau similar)
+ demo cu afișarea descriptorilor standard pentru shell-ul curent
primul fișier deschis va avea descriptorul 3
+ demo cu descriptorul primului fișier deschis
și funcțiile C standard obțin tot un descriptor de fișier: fopen(3) apelează open(2), inițializează o intrare în tabelă și întoarce indexul corespunzător ca descriptor; acesta este "îmbrăcat" în structura FILE
+ demo cu structura _IO_FILE pe Linux care conține pe lângă buffere "int fd;" sau ceva similar
Operații cu date și cursorul de fișier
o dată obținut un descriptor de fișier (sau un handle precum FILE) putem face operații cu fișierele
operațiile uzuale sunt read/write
read: citim date *din* fișier *într-un* buffer (din user/application space)
write: scriem date *dintr-un* buffer (din user/application space) *în* fișier
operațiile de citire/scriere avansează cursorul de fișier
+ diagramă cu fluxul datelor la citire, fluxul datelor la scriere
citirea nu avansează dincolo de dimensiunea fișierului; scrierea poate trece peste și modifica/crește dimensiunea fișierului
+ demo cu afișarea cursorului fișierului după operații de citire și scriere
operațiile de poziționare (lseek/fseek/SetFilePointer) pot plasa oriunde cursorul; îl pot da înapoi
+ demo cu afișarea cursorului după operații de poziționare
Operații mai interesante/ciudate
putem configura dimensiunea fișierului la o valoare dată, fără a avea date de acea dimensiune: truncate(2)/ftruncate(2) și comanda truncate
use case: pentru a da impresia unor aplicații că au fișier suficient de mari (de exemplu discuri de mașini virtuale)
+ demo cu comanda truncate și funcțiile truncate(2)/ftruncate(2)
sparse file: fișiere cu "găuri"
poziționare cursor de fișier dincolo de limita fișierului și scris informații
use case: Bittorrent, se “alocă” fictiv tot fișierul și apoi se scriu date unde sunt primite chunk-urile prin protocolul Bittorrent, în rest rămân găuri
+ demo de creare a unui sparse file
+ tabel cu ce modifică apelurile open, read, write, seek, truncate: cursorul de fișier și dimensiunea fișierului
Operații cu date: system I/O vs buffered I/O
putem efectua operații cu date citire/scriere folosind interfața low-level (wrappere peste apeluri de sistem: read(2)/write(2) sau WriteFile(2)/ReadFile(2)) sau interfața C standard (fread(3)/fwrite(3))
interfața C standard este portabilă între platforme și folosește buffere, se mai numește buffered I/O
operațiile trec prin buffere, bufferele sunt parte din libc, în user/application space
+ diagramă cu aplicație ---> libc (buffered I/O) cu buffere ---> system I/O / apeluri de sistem ---> kernel (fișiere)
avantaj buffered I/O: evitare apel de sistem, dezavantaj: nu sunt "sincrone" (nu "ajung" direct la dispozitiv) și consumă memorie pe buffere
se face "flush" la commanda fflush sau când se ajunge la Enter (pentru standard output) sau la buffer plin (pentru fișiere) sau mereu (unbuffered, pentru standard error)
+ demo cu apeluri de sistem efectuate atunci când facem buffered I/O față de când face system I/O
Internele redirectărilor
se întâmplă să dorim ca informațiile afișate la ieșirea standard să ajungă într-un fișier: comanda > file
ca să se întâmple acest lucru trebuie ca intrare cu indexul/descriptorul 1 (corespunzător ieșirii standard) din tabela descriptorilor de fișiere a procesului să refere o structură de fișier deschis pentru fișierul file
+ diagramă cu
acest lucru îl obținem prin close(1) și apoi open("file", ...)
+ demo cu close și open
uzual, însă, pentru mai multă flexibilitate, de exemplu refolosirea unui fișier deja deschis, folosim duplicarea descriptorilor

= duplicarea descriptorilor

facem ca o intrare din tabela de descriptori de fișiere să refere aceeași structură de fișier deschis ca altă intrare
apelul dup(fd) face ca prima intrare liberă găsită în tabela de descriptori de fișiere să refere structura de fișier deschis referită de fd
apelul dup2(olfd, newfd) face ca intrarea cu indexul newfd să refere structură de fișier deschis referită de oldfd; newfd este închis dacă este cazul să fie deschis
+ diagramă cu tabelă de descriptori de fișier cu mai mulți descriptori referind aceeași structură de fișier deschis
+ (opțional): discuție despre nevoia de dup2(2) și condiții de cursă pentru dup(2) + close(2)
dup/dup2 sunt folosite la redirectare
+ demo: close(1), dup(fd)
+ demo: dup2(fd, 1)
+ actualizare diagramă cu pașii pentru duplicarea standard output
similar se întâmplă la operatorul | (pipe) din shell de redirectare a comenzilor; îl vom discuta la cursul 3: Procese
Închiderea și ștergerea fișierelor
fișierele se închid folosind close(2)/fclose(3)
închiderea presupune invalidarea intrării corespunzătoare descriptorului din tabela de descriptori de fișiere
mai există un câmp în structura de fișier deschis: contor de utilizare: operațiile de duplicare cresc acel contor
atunci când toate toate referințele dispar, structura de fișier este eliberată din memorie (spunem că fișierul este închis)
+ diagramă cu referințe multiple din tabela de descriptori de fișier către structura de fișier și operații close(2)
similar pot exista mai multe referințe de la fișier deschis la același FCB: dacă facem mai multe apeluri open(2)
diferența dintre dup și mai multe apeluri open(): nu se partajează cursorul de fișiere
+ demo cu dup și open și modificarea cursorului de fișiere folosind fseek, urmărire cu lsof
Despre cursul 12: implementarea sistemului de fișiere
operațiile prezentate nu diferă între sisteme de fișiere diferite
diferențele de implementare a sistemului de fișiere le vom prezenta în cursul 12
cursul va prezenta internele FCB pe disc
structura unui sistem de fișiere pe disc
Sumar
stocăm datele în fișiere, fișierele sunt date structurate, metadatele fișierelor sunt agregate în FCB (file control block)
fișierele sunt deschide pentru a fi folosite, se întoarce la nivel low-level un descriptor de fișier
un descriptor este o intrare într-o tabelă de descriptori de fișiere care conține un pointer către o structură de fișier deschis
mai multe structuri de fișier deschis pot referi același FCB (mai multe apeluri open(2) pe același fișier)
mai multe intrări în tabela de descriptori de fișiere pot referi aceeași structură de fișier deschis (folosind dup(2) sau dup2(2))
operațiile read(2), write(2) și lseek(2) modifică cursorul de fișier (prezent în structura de fișier deschis)
operația ftruncate(2) modifică dimensiunea fișierului (prezentă în FCB)

Curs 03: Procese

Sumar curs anterior

stocăm datele în fișiere, fișierele sunt date agregate/compartimentate, metadatele fișierelor sunt agregate în FCB (file control block)

Fișierele sunt deschise pentru a fi folosite, se întoarce la nivel low-level un descriptor de fișier
un descriptor este o intrare într-o tabelă de descriptori de fișiere care conține un pointer către o structură de fișier deschis
mai multe structuri de fișier deschis pot referi același FCB (mai multe apeluri open(2) pe același fișier)
mai multe intrări în tabela de descriptori de fișiere pot referi aceeași structură de fișier deschis (folosind dup(2) sau dup2(2))
operațiile read(2), write(2) și lseek(2) modifică cursorul de fișier (prezent în structura de fișier deschis)
operația ftruncate(2) modifică dimensiunea fișierului (prezentă în FCB)

Acțiuni în sistemul de calcul

utilizatorul dorește execuția de acțiuni în sistem: folosirea procesorului/procesoarelor
datele sunt în memorie (aduse acolo de la I/O, eventual suport persistent) și apoi aduse pe procesor
în memorie avem date și cod (instrucțiuni)
+ diagramă cu procesor, memorie, date
dorim să executăm mai multe acțiuni diferite pe un sistem: folosim procese

Procese

încapsularea unei acțiuni în sistemul de calcul: date și cod în memorie, rulare instrucțiuni pe procesor, interacțiune cu I/O
+ diagramă cu proces care abstractizează: memorie, procesor, I/O
permite multi-programare: mai multe acțiuni pe sistem
procesele sunt izolate între ele: memoria este separată, rulează separat pe procesor, folosesc secvențial I/O
sistemul de operare se ocupă de izolarea și planificarea proceselor
nevoia de procese: mai multe acțiuni, procesoare multiple
provocări legate de procese: izolare, planificarea accesului la resurse, comunicarea inter-procese, accesul mai multor procese la resurse limitate

Atributele unui proces

identificator (PID)
resurse: spațiu virtual de adrese (memorie), timp de lucru pe procesor, fișiere deschise (în tabela de descriptori de fișiere)
user/group
starea unui proces (mai târziu în curs)
+ demo cu /proc/pid/status
+ ps -eF (afișează atribute)
structura de proces se numește PCB (process control block), reținută în kernel space
+ demo: afișarea structurii task_struct din Linux: https://elixir.bootlin.com/linux/v4.20.13/source/include/linux/sched.h#L590

Crearea unui proces

dintr-un proces existent (procesul părinte), uzual un shell
se creează o ierarhie de procese: un proces are un singur proces părinte dar oricâte procese copil
+ demo cu vizualizarea ierarhiei de procese
PID-ul procesului părinte este determinat cu apelul getppid(); PID-ul proceselor copil e întors de apelul de creare
procesul părinte invocă loader-ul care încarcă datele și codul dintr-un executabil în memoria noului proces: loading, load-time
executabilul/programul este imaginea procesului: datele (variabile globale inițializate .data, neinițializate .bss și read-only .rodata) și codul (.code sau .text)
executabilul are un punct de intrare (entry point) de unde începe execuția noului proces (prin acela se ajunge la funcția main())
+ diagramă cu shell-ul, comanda ls, executabilul /bin/ls și crearea unui nou proces
procesele sunt identificate prin PID, nu prin nume, mai multe procese pot avea aceeași imagine de executabil

fork() și exec()

în Linux, crearea unui proces nou se face cu două apeluri: fork() și exec()
apelul fork() creează un proces copil ca fiind o copie a procesului părinte; partajează informații precum tabela de descriptori de fișiere, pornesc de la același cod
+ demo cu partajarea cursorului de fișier
apelul fork() se apelează o dată și se întoarce de două ori: o dată în procesul copil și o dată în procesul părinte
+ exemplu apel fork()
apoi apelul exec() invocă loaderul și încarcă o nouă imagine de executabil
apelul exec() modifică spațiul virtual de adrese al procesului, fără a schimba PID-ul acestuia

Încheierea unui proces

normală și anormală
normală: se ajunge la sfârșitul codului sau apelează exit()
anormală: procesul execută o acțiune nevalidă și este omorât de sistemul de operare (i se trimite un semnal) sau procesul este omorât de un alt proces (tot printr-un semnal)
rolul procesului părinte este de a se îngriji de colectarea de informații legate de încheierea procesului copil
spunem că procesul părinte așteaptă (wait) după procesul copil; așteptarea duce la furnizarea de informații despre modul în care și-a încheiat execuția
atunci când shell-ul creează un proces de obicei așteaptă încheierea sa (apelează wait())
dacă rulăm o comandă cu & la sfârșit (pentru rulare în background), shell-ul nu așteaptă încheierea comenzii
un proces orfan este un proces al cărui proces părinte s-a încheiat; procesele orfane sunt înfiate în general de procesul init
un proces zombie este un proces care și-a încheiat execuția dar nu a fost așteptat de părintele său
dacă un proces părinte moare și nu a așteptat un proces copil zombie, acesta este zombie orfan; este înfiat de procesul init și apoi este "curățat" din sistem
+ demo cu procese orfane și zombie

Procese daemon

sunt procese detașate de terminal, stdin, stdout, stderr nu referă terminale, de obicei /dev/null
procesul părinte este init
nu sunt interactive, realizează acțiuni de mentenanță sau oferă servicii

Rularea proceselor

procesele sunt planificate să ruleze pe procesoare
în mod normal fiecare proces are asociată o cuantă de rulare, când expiră, sistemul de operare plasează alt proces pe procesor
schimbarea unui proces cu un alt proces poartă numele de schimbare de context
mai multe detalii în cursul 4: Planificarea execuției
unele procese sunt CPU-intensive dacă petrec mult timp pe procesor sau I/O intensive dacă operează adesea pe dispozitive I/O
+ demo cu CPU intensive și I/O intensive

Starea proceselor

un proces care rulează pe procesor este în starea running
un proces care execută o operație I/O se blochează în așteptarea încheierii operației, intră în starea blocking/waiting
atunci când un proces poate rula, dar nu are alocat un procesor, este în starea ready
stările running, blocking și ready sunt cele trei stări principale ale unui proces
+ diagramă cu stările proceselor
tranziția din running în ready se întâmplă când unui proces îi expiră cuanta
tranziția din running în blocking este când un proces realizează o operație I/O blocantă
tranziția din blocking în ready are loc când operația I/O blocantă s-a definitivat
tranziția din ready în running este când se eliberează un procesor
mai multe în cursul 4: Planificarea execuției

Comunicarea între procese. Pipe-uri

comunicarea inter-proces (IPC: Inter-Process Communication)
transfer de informații, notificări
apelul wait() e o forma de IPC de notificare
socketii sunt o formă de transfer de informații
pipe-uri: cu nume (FIFO, intrare în sistemul de fișiere) și anonime (doar în memorie)
+ demo cu pipe-uri cu nume
pipe-uri anonime (numite simplu pipe-uri): folosite în shell la comanda cmd1 | cmd2
un pipe este un buffer în kernel cu două capate: unul de citire și unul de scriere
pot fi folosite doar între procese înrudite
se redirectează stdout-ul unui proces la pipefd[1] (capătul de scriere)
se redirectează stdin-ul celuilalt proces la pipefd[0] (capătul de citire)
+ demo cu pipe-uri anonime cu proces părinte, proces copil

Sumar

procesele sunt încapsularea execuției într-un sistem
procesele abstractizează procesorul, memoria și I/O
un proces este creat de un alt proces, dintr-un executabil, prin loading
atributele unui proces sunt păstrate în PCB (Process Control Block)
pe Unix există apelurile fork() și exec(): fork() duplică spațiul de adrese al unui proces, exec() invocă loader-ul
procesele formează o ierarhie, procesul părinte așteaptă încheierea proceselor copil
procesele orfane sunt adoptate de init
procesele zombie sunt cele care nu au fost încă așteptate de procesul părinte
procesele comunică pentru transfer de date și pentru notificări
pipe-urile anonime (folosite de shell la comanda cmd1 | cmd2) sunt folosite doar între procese înrudite

Curs 04: Planificarea execuției

Sumar curs anterior

procesele sunt încapsularea execuției într-un sistem
procesele abstractizează procesorul, memoria și I/O
un proces este creat de un alt proces, dintr-un executabil, prin loading
atributele unui proces sunt păstrate în PCB (Process Control Block)
pe Unix există apelurile fork() și exec(): fork() duplică spațiul de adrese al unui proces, exec() invocă loader-ul
procesele formează o ierarhie, procesul părinte așteaptă încheierea proceselor copil
procesele orfane sunt adoptate de init
procesele zombie sunt cele care nu au fost încă așteptate de procesul părinte
procesele comunică pentru transfer de date și pentru notificări
pipe-urile anonime (folosite de shell la comanda cmd1 | cmd2) sunt folosite doar între procese înrudite

Multitasking și scheduling

pe un sistem există mai multe procese
sistemul dispune de un număr limitat de procesoare
sistemul de operare trebuie să asigure accesul tuturor proceselor la procesoare
un proces rulează pe un procesor (sau mai multe) și apoi lasă locul altui proces: multi-tasking
sistemul de operare planifică accesul unui proces la procesoare: scheduling
planificatorul de procese este o componentă din sistemul de operare care decide când un proces părăsește procesorul și ce proces îi ia locul

Schimbarea de context

un proces este un program căruia i se atașează un context de execuție
când planificatorul decide că un proces părăsește procesorul, are loc o schimbare de context
schimbarea de context înseamnă salvarea informațiilor procesului anterior într-o zonă din sistemul de operare și restaurarea informațiilor noului proces
schimbarea de context înseamnă overhead
schimbarea de context se poate produce:
* voluntar: un proces decide sau cauzează schimbarea de context
** un proces cedează de bună voie procesorul: yielding
** un proces execută o operație blocantă (de exemplu citire de la un dispozitiv care nu are încă date)
** un proces își încheie execuția
* nevoluntar: planificatorul sistemului de operare decide forțarea unui proces de pe procesor
** un proces a stat prea mult timp pe procesor
** apare în sistem un proces mai important
+ demo cu schimbări voluntare și nevoluntare la rularea comezii find

Reminder: Stările proceselor

un proces care rulează pe procesor este în starea running
un proces care execută o operație I/O se blochează în așteptarea încheierii operației, intră în starea blocking/waiting
atunci când un proces poate rula, dar nu are alocat un procesor, este în starea ready
stările running, blocking și ready sunt cele trei stări principale ale unui proces
+ diagramă cu stările proceselor
tranziția din running în ready se întâmplă când unui proces îi expiră cuanta
tranziția din running în blocking este când un proces realizează o operație I/O blocantă
tranziția din blocking în ready are loc când operația I/O blocantă s-a definitivat
tranziția din ready în running este când se eliberează un procesor

Stările proceselor și schimbările de context

tranzițiile din starea running sunt tranziții de schimbare de context: procesul cedează procesorul în fața altui proces
când un proces cedează procesorul, se alege un proces din starea READY
ce se întâmplă când nu există nici un proces în starea READY? procesul idle
ce tranziții au loc la diferitele tipuri de schimbări de context:
RUNNING -> READY: yielding, prea mult timp pe procesor, proces prioritar
RUNNING -> WAITING: un proces execută o operație blocantă
RUNNING -> TERMINATED: un proces și-a încheiat execuția
noul proces execută tranziția READY -> RUNNING

Planificatoare cooperative și preemptive

planificatorul este o componentă a sistemului de operare; este o funcție care este apelată pentru:
* a selecta un proces aflat în READY
* a înlocui procesul curent (din RUNNING) cu noul proces selectat (din READY)
planificatorul este apelat pentru a efectua schimbarea de context
planificatoarele pot fi cooperative și preemptive
planificatoarele cooperative permit doar schimbări de context voluntare
cele preemptive permit și schimbări de context nevoluntare: a preempta = a forța părăsirea procesorului
planificarea cooperativă are dezavantajul starvation: un proces poate acapara complet procesorul și să nu lase alte procesoare să ruleze
în general planificarea preemptivă presupune existența unei cuante de timp asociate unui proces; expirarea cuantei de timp forțează înlocuirea procesului (context switch)

Criterii de evaluare a unui planificator

sunt esențiale două metrici: productivitate (throughput) și echitate (fairness)
un planificator este cu atât mai productiv cu cât se consumă cât mai puțin timp schimbând contextul și mai mult timp rulând procese; procesele să își termine cât mai repede treaba
un planificator este cu atât mai echitabil cu cât fiecare proces are acces la procesor; adică procesele stau cât mai puțin timp în coada READY până să intre în RUNNING
un sistem este inechitabil când un proces stă foarte mult în coada READY și nu este planificat pe procesor: starvation
un sistem este neproductiv când sunt schimbări de context foarte dese și procesele fac foarte puțină treabă
turnaround time: timpul din care un proces intră în sistem până când își încheie execuția
average turnaround time: media turnaround time pentru toate procesele din sistem
waiting time: suma timpilor în care un proces așteaptă în starea READY
average waiting time: media waiting time pentru toate procesele din sistem
un sistem productiv are average turnaround time mic
un sistem echitabil are average waiting time mic
Atenție: waiting time se referă la timpul petrecut în starea READY (nu în starea WAITING)
+ demo cu timpul de așteptare (în coada READY)

Cuanta de timp. Planificatorul round robin

cuanta de timp se asociază unui proces în planificatoare preemptive
când expiră cuanta unui proces acesta este scos de pe procesor, i se calculează o nouă cuantă și este trecut în starea READY
timer-ul procesorului (uzual o dată 1 ms sau 10ms) generează întrerupere de ceas; în rutina de tratare a întreruperii de ceas se verifică dacă un proces în RUNNING are cuanta expirată
cuanta este dinamică, se actualizează la fiecare expirare
productivate vs echitate: cuantă mare vs cuantă mică
cuantă mare -> schimbări de context mai rare, productivitate sporită
cuantă mică -> schimbări de context mai dese, echitate sporită
cuanta variază între procese; procesele sunt CPU-intensive vs I/O-intensive
procesele I/O-intensive primesc în general o cuantă de timp mai mare pentru că se vor bloca și vor trece din RUNNING în WAITING și vor lăsa alt proces în loc
procesele CPU-intensive primesc în general o cuantă de timp mai mică; este posibil să ruleze pe procesor până la expirarea cuantei; cu o cuantă mare ar dura mai mult până ar lăsa alt proces în locul său pe procesor

Coadă/cozi de procese READY. Prioritățile proceselor

când are loc o schimbare de context, planificatorul alege un proces aflat în starea READY și îl trece în RUNNING; ce procese alege
planificatorul round-robin are o coadă de procese READY; ia pe rând procesele READY și le trece în RUNNIG; când un proces ajunge în READY e adăugat la sfârșitul cozii
unele procese sunt mai importante; au prioritate mai mare
planificatoarele cu priorități au mai multe cozi pentru procesele READY: câte o coadă pe o prioritate; când se planifică un proces se ia primul proces din coada cu prioritatea cea mai mare
prioritate statică este o prioritate care nu poate fi modificată pe parcursul execuției procesului; un sistem care folosește doar priorități statice poate duce la starvation pentru procesele cu prioritate mai mică; vor rula mereu procesele cu prioritate mai mare dacă dintre acestea sunt mereu în starea READY
prioritățile statice în Linux sunt numite "nice"
+ demo cu procese cu valoare nice diferită
pentru a preveni starvation, prioritățile sunt dinamice, se modifică pe parcursul execuției; un proces care stă destul de mult într-o coadă READY cu prioritate mai mică va fi promovat într-o coadă READY cu prioritate mai mare
modificarea priorității ține cont de natura procesului, similar alegerii cuantei: procesele CPU-intensive primesc, în general, o prioritate mai mică decât procesele I/O intensive

Comunicarea inter-proces

procesele comunică pentru a transfera informații, pentru notificare sau pentru a asigura integritatea datelor (sincronizare)
transferul de informații se face prin
* comunicare de tip țeavă, sau transfer de mesaje: pipe-uri (anonime), pipe-uri cu nume (FIFO), socketi locali (UNIX), socketi de rețea (Berkeley), cozi de mesaje
** există API-uri și biblioteci de message passing: MPI, ZeroMQ, RabbitMQ
* comunicare cu date partajate: memorie partajată (shared memory)
** API de date partajate: OpenMP
la comunicarea prin transfer de mesaje există un sender, un receiver, un canal virtual de comunicare și un protocol înțeles de parteneri; în general nu e nevoie de sincronizarea accesului la date pentru că fiecare partener are o instanță
la comunicarea prin memorie partajată partenerii pot fi cititori sau scriitori; datele fiind comune, este nevoie de sincronizare pentru a asigura integritatea acestora

notificarea se face prin semnale (numite și excepții pe Windows); un proces își definește o rutină de tratare a semnalului/excepției, rutină apelată la apariția semnalului
semnalele pot fi livrate de
* sistemul de operare ca urmare a întâmpinării unei situații neprevăzute în execuția procesului: de exemplu acces nevalid la memorie (segmentation fault)
* de un alt proces pentru a notifica procesul curent de o condiție sau eveniment
sincronizarea se face prin mutex-uri, semafoare, variabile condiție; mai multe au fost prezentate la APD, vom insista la cursul 9: Sincronizare

Sumar

mai multe procese concurează pe procesoarele sistemului; planificatorul (parte a sistemului de operare) gestionează accesul proceselor la procesoare (trecerea în starea RUNNING)
planificarea unui proces, adică înlocuirea unui proces cu altul se numește schimbare de context
schimbarea de context se poate produce:
* voluntar: un proces decide sau cauzează schimbarea de context
** un proces cedează de bună voie procesorul: yielding
** un proces execută o operație blocantă (de exemplu citire de la un dispozitiv care nu are încă date)
** un proces își încheie execuția
* nevoluntar: planificatorul sistemului de operare decide forțarea unui proces de pe procesor
** un proces a stat prea mult timp pe procesor
** apare în sistem un proces mai important
planificatoarele pot fi cooperative și preemptive; cele cooperative nu pot preveni apariția situației de starvation
planificatorul urmărește două obiective conflictuale: productivitate și echitate
planificatoarele preemptive folosesc cuantă de timp și prioritate pentru fiecare proces
comunicarea intreprocese presupune transfer de mesaje, memorie partajată, notificare sau sincronizare

Curs 05: Gestiunea memoriei

Sumar curs anterior

mai multe procese concurează pe procesoarele sistemului; planificatorul (parte a sistemului de operare) gestionează accesul proceselor la procesoare (trecerea în starea RUNNING);
planificarea unui proces, adică înlocuirea unui proces cu altul se numește schimbare de context
schimbarea de context se poate produce:
* voluntar: un proces decide sau cauzează schimbarea de context
** un proces cedează de bună voie procesorul: yielding
** un proces execută o operație blocantă (de exemplu citire de la un dispozitiv care nu are încă date)
** un proces își încheie execuția
* nevoluntar: planificatorul sistemului de operare decide forțarea unui proces de pe procesor
** un proces a stat prea mult timp pe procesor
** apare în sistem un proces mai important
planificatoarele pot fi cooperative și preemptive; cele cooperative nu pot preveni apariția situației de starvation
planificatorul urmărește două obiective conflictuale: productivitate și echitate
planificatoarele preemptive folosesc cuantă de timp și prioritate pentru fiecare proces
comunicarea intreprocese presupune transfer de mesaje, memorie partajată, notificare sau sincronizare

Procese și memorie

într-un sistem de calcul modern, acțiunile sunt realizate în cadrul unui proces care abstractizează procesor, memorie, I/O
executabile încărcate în memorie: date statice, cod; apoi date dinamice în memorie
memoria proceselor trebuie să fie izolată
ideal procesele au câtă memorie își doresc

Funcționarea unui sistem de calcul. Rolul memoriei

la nivel scăzut acțiunile sunt realizate de procesor prin interacțiunea cu memoria și I/O-ul, conform modelului von Neumann
+ diagramă cu modelul von Neumann
procesorul preia date in memorie sau din I/O, le prelucrează, le stochează înapoi în memorie sau I/O
procesorul execută instrucțiuni stocate tot în memorie
ciclul de execuție al procesorului: instruction fetch, instruction decode, data fetch, execution, data writeback
+ actualizare diagramă von Neumann cu ciclul de execuție al procesorului
instrucțiunile sau codul (ce e de făcut) și datele (cu ce faci) sunt reținute în memorie
memoria este legată de procesor/procesoare prin magistrală (bus, uzual FSB: Front Side Bus): magistrala de date și magistrala de adrese
memoria poate fi privită ca un vector de octeți adresat prin magistrala de adrese
citirea din memorie: TODO
scrierea în memorie TODO
mai multe procesoare pot concura la accesarea memorie și genera probleme de concurență (race conditions): mai multe la cursul 9 și la ASC

Internele memoriei

Ulrich Drepper - What every programmer should know about memory
tipuri de memorie
static vs dynamic
RAM
ROM (PROM, EPROM, EEPROM)
rânduri, coloane
detalii la PM
refresh
latență, bandwidth, frecvență
diferența de viteză memorie procesor
instruction prefetch
branch prediction
speculative execution
nevoia de memorie cache
+ demo cu informații despre memoria sistemului (meminfo, /proc/cpuinfo, getconf)

Memoria cache

reminder de la CN
viteză memorie cache
cache hit, cache miss
There are only two hard things in Computer Science: cache invalidation and naming things. -- Phil Karlton
There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors.
There are only two hard problems in computer science:
0) Cache invalidation
1) Naming things
5) Asynchronous callbacks
2) Off-by-one errors
3) Scope creep
6) Bounds checking

Memoria în sistemele multitasking

nevoia de izolare
suficient de multă memorie pentru un proces
fiecărui proces i se asociază un spațiu virtual de adrese
în spate, sistemul de operare asociază adresele virtuale cu adrese efective
memoria virtuală e o convenție, memoria fizică reține datele efective
avantaje memorie virtuală: separație între procese, impresia că fiecare proces are tot spațiul disponibil pentru sine, nu e nevoie să știi ce adrese fizice vei folosi
dezavantaje memorie virtuală: overhead de calcul a adresei fizice din adresa virtuală la fiecare acces

Spațiul virtual de adrese al unui proces

unic fiecărui proces
asigură separație între procese
sistemul de operare face asocierea între memoria virtuală (a fiecărui proces) și memoria fizică: există o tabelă de asociere pentru fiecare procese
+ diagramă cu adresă virtuală -> tabelă de asociere -> adresă fizică
spațiul virtual de adrese ocupa 2^32 octeți (4GB) pe un sistem pe 32 de biți
cuprinde zone: text (cod), rodata, data (date inițializate), bss (date neinițializate), heap, biblioteci (cu subzone: text, rodata, data, bss), stivă
în mod uzual partea superioadă din spațiul de adrese al fiecărui proces este rezervată sistemului de operare
+ diagramă cu spațiul virtual de adrese
memoria sistemului de operare este mapată în partea finală a spațiului virtual de adrese al tuturor proceselor
pe Linux spațiul (3GB, 4GB) este rezervat sistemului de operare (0xc0000000-0xffffffff): este inaccesibil din user mode (spunem că avem split 3/1); pe Windows 32bit avem split 2/2
+ demo cu urmărirea spațiului de adrese al unui proces (pmap)
Ce se întâmplă dacă avem mai mult spațiu virtual pentru un proces decât spațiu fizic? Nu vom putea folosi spațiul virtual complet, sau vom folosi spațiul de swap, sau vom partaja memorie fizică?
Ce se întâmplă dacă avem mai mult spațiu fizic decât spațiu virtual (de exemplu rularea de procese legacy 32 de biți pe un sistem pe 64 de biți)? Mai multe procese pot ocupa întreg spațiul fizic.
x86: 32 biți pentru adrese virtuale, 32 de biți pentru adrese fizice (maxim RAM accesibil 2^32 = 4GB)
x86_64 (curent, 2019): 48 de biți folosiți pentru adrese virtuale (plan pentru 57 de biți), 46 sau 52 de biți pentru adrese fizice (maxim RAM accesibil: 2^46 = 64 TB)
https://simonis.github.io/Memory/, https://en.wikipedia.org/wiki/X86-64#Virtual_address_space_details

Compartimentarea spațiului virtual de adrese. Segmentare

pentru asocierea spațiului virtual cu spațiul fizic, prima opțiune este să asociem tot spațiul virtual (4GB) la o parte fizică, alt spațiu virtual (4GB) la altă parte fizică
probleme: avem nevoie de 8GB RAM pentru doar două procese; multe zone din spațiul virtual nu sunt folosite
alternativa: segmentare: compartimentarea spațiului virtual de adrese în zone (precum text, heap, stivă) și asocierea fiecărei zone la spațiul fizic
+ diagramă cu segmentare
implementarea segmentării: tabela de segmentare, selector de segment
adresa virtuală este compusă din selectorul de segment și offset-ul în cadrul segmentului (în notația CS:EIP, CS e selectorul de segment, code segment; EIP e adresa/offset-ul în cadrul segmentului); selectorul extrage adresa fizică asociată segmentului și adună offsetul, rezultă adresa fizică efectivă
+ diagramă pentru calculul adresei fizice
segmentarea este un concept mai vechi care este acum depășit de paginare; are sens sa vorbim de segmentare pentru a vedea evoluția compartimentării memorie și pentru cazurile în care, pe arhitectura x86, veți întâlni precizări legate de segmentare
segmentation fault (SIGSEGV - Signal Segment Violation): acces dincolo de limita unui segment al procesului

avantaj: asociem doar ce este nevoie
dezavantaj: fragmentare externă: este dificil să găsim loc (potrivit) pentru un nou segment fizic
segmentarea e o soluție legacy, mai are unele relicve pe sistemele x86
soluție: folosirea paginării

Paginare

soluția folosită curent în sisteme moderne ce folosesc memorie virtuală
compartimentarea spațiului virtual al fiecărui proces și al spațiului fizic al sistemului în componente de dimensiune fixă, numite pagini: pagini virtuale (pages) și pagini fizice (frames)
în mod uzual, paginile au 4KB
pe un sistem pe 32 de biți, un proces are spațiu virtual de 4GB și are deci, 4GB/4KB = 2^20 pagini virtuale
o tabelă de asociere, tabela de pagini (page table), face asocierea între pagini virtuale (pages) și pagini fizice (frames)
asocierea se mai numește mapare/mapping
+ diagramă de sus cu proces, spațiu virtual de adrese, pagini virtuale, tabelă de pagini, pagini fizice
translatarea de adrese este folosirea tabelei de pagini pentru a calcula o adresă fizică dintr-o adresă virtuală
avantaj folosire paginare: nu mai avem fragmentare externă; găsim un loc pentru o pagină și o alocăm acolo; paginile fizice corespunzătoare unui proces vor fi împrăștiate în memoria fizică
dezavantaj: poate apărea fragmentare internă: alocăm o pagină (4KB) și folosim doar câțiva octeți; spațiu ocupat de tabela de pagini
+ demo cu granularitatea alocării (multiplu de pagină prin apel de sistem)

Tabela de pagini

conține 2^20 intrări pe un sistem pe 32 de biți
este indexată cu adresa pagini virtuale și conține adresa paginii fizice
o adresă virtuală este cuprinsă din: adresă de pagină virtuală (20 de biți) + offset în pagină (12 biți): 2^12 = 4KB, cât cuprinde o pagină
o adresă virtuală este cuprinsă din: adresă de pagină fizică (20 de biți) + offset în pagină (12 biți)
+ diagramă cu translatarea de adrese din adresa virtuală în adresa fizică folosind tabela de pagini
responsabilitatea translatării aparține unei componente hardware de pe procesor: MMU (Memory Management Unit)
fiecare intrare în tabela de pagini se numește page table entry (PTE)
bitul valid/invalid din PTE precizează dacă o pagină este validă; dacă nu, rezultă în excepție de acces la memorie (page fault)
în PTE apar precizări despre tipul de operații permise (read-only, read-write)
există o tabelă de pagini per proces, reținută în memoria fizică
un registru dedicat reține adresa tabelei de pagini în memorie pentru procesul curent: PTBR (Page Table Base Register), cr3 pe x86, TTBR pe ARM
neajunsuri ale tabelei de pagini:
* este stocată în memorie, ocupă spațiu: soluție: tabelă de pagini ierarhică
* este nevoie de accesarea tabelei de pagini pentru fiecare operație cu memoria: soluție: TLB

Tabelă de pagini ierarhică

sau paginare multi-nivel
în forma implicită a adresei virtuale: 20 de biți pentru adresa paginii (PTE index) și 12 biți pentru offset: 2^20 intrări
tabelă de pagini ierarhică: 10 biți (page directory index), 10 biți (PTE index), 12 biți offset
o intrare în page directory referă o structură page table (cu 2^10 intrări), o intrare în tpage table are 2^10 intrări și conține adrese de pagini fizice
mai multe niveluri (4-5) pentru sistemele pe 64 de biți
+ diagramă cu tabela de pagini ierarhică
dacă o zonă lipsește, intrarea în page directory este nevalidă și nu referă page table
avantaj: spațiu redus
dezavantaj: mai mult overhead de translatare

TLB

Translation Lookaside Buffers
o operație cu memoria înseamnă un acces la tabela de pagini (în memoria fizică) pentru extragerea mapării și apoi un acces la memoria efectivă pentru extragerea informației (2 accese)
pentru a reduce overhead-ul, TLB reține cele mai recent accesate intrări în tabela de pagini; este un cache
are 128-256 intrări cu cele mai recente mapări
cu un "hit rate" bun se reduce timpul de translatare
+ diagramă cu folosirea TLB pentru translatarea de adrese
+ demo cu informații despre TLB
când are loc TLB flush? la address space switch (context switch)
după address space switch TLB este repopulat pe măsură ce sunt accesate noi pagini
de ce se mapează sistemul de operare în spațiul virtual de adrese al fiecărui proces și nu i se asociează un spațiu virtual de adrese separat? la apel de sistem și întoarcerea din apel de sistem nu este nevoie de schimbarea spațiului de adrese și deci, de TLB flush
TLB flush e soluția cea mai simplă
procesoarele moderne oferă precizarea unui identificator de spațiul de adresă (Address Space ID, ASID, pe ARM sau Process-Context ID, PCID, pe x86) pentru flush selectiv la nivelul TLB-ului când are loc un address space switch

continuăm la cursul următor cu detalii despre memoria virtuală: demand paging, swapping, shared memory, file mapping, page replacement, thrashing

Sumar

memoria este folosită pentru a reține cod/instrucțiuni și date folosite de procesor
memoria fizică (RAM) este mai înceată decât procesorului: folosim memorie cache
pentru izolare și utilizare facilă folosim memorie virtuală
fiecare proces are asociat un spațiu virtual de adrese compus din zone

sistemul de operare asociază/mapează spațiul virtual de adrese al fiecărui proces la spațiul fizic (memorie RAM)
o formă veche de translatare era segmentarea
forma curentă este paginarea: împărțirea spațiului virtual și a spațiului fizic în pagini
tabela de pagini face asocierea între pagini virtuale (pages) și pagini fizice (frames)
translatarea este realizată de o componentă hardware (de pe procesor) numită MMU (Memory Management Unit)
tabela de pagini este reținută în memorie și este: mare (ocupă spațiu) și înceată (accesarea tabelei de pagini înseamnă acces la memorie)
PTBR (Page Table Base Register) reține adresa în memorie a tabelei de pagini pentru procesul curent
tabela de pagini ierarhică combate dezavantajul spațiului ocupat în memorie de tabela de pagini clasică (neierarhică)
TLB (Translation Lookaside Buffers) combate dezavantajul overhead-ului de translatare (nevoie de acces la memorie)
este nevoie de TLB flush la address space switch, când se schimbă tabelele de pagini

Curs 06: Memoria virtuală

Sumar curs anterior

memoria este folosită pentru a reține cod/instrucțiuni și date folosite de procesor
memoria fizică (RAM) este mai înceată decât procesorului: folosim memorie cache
pentru izolare și utilizare facilă folosim memorie virtuală
fiecare proces are asociat un spațiu virtual de adrese compus din zone
sistemul de operare asociază/mapează spațiul virtual de adrese al fiecărui proces la spațiul fizic (memorie RAM)
o formă veche de translatare era segmentarea
forma curentă este paginarea: împărțirea spațiului virtual și a spațiului fizic în pagini
tabela de pagini face asocierea între pagini virtuale (pages) și pagini fizice (frames)
translatarea este realizată de o componentă hardware (de pe procesor) numită MMU (Memory Management Unit)
tabela de pagini este reținută în memorie și este: mare (ocupă spațiu) și înceată (accesarea tabelei de pagini înseamnă acces la memorie)
PTBR (Page Table Base Register) reține adresa în memorie a tabelei de pagini pentru procesul curent
tabela de pagini ierarhică combate dezavantajul spațiului ocupat în memorie de tabela de pagini clasică (neierarhică)
TLB (Translation Lookaside Buffers) combate dezavantajul overhead-ului de translatare (nevoie de acces la memorie)
este nevoie de TLB flush la address space switch, când se schimbă tabelele de pagini

Spațiul virtual de adrese al unui proces

un proces are un spațiu virtual de adrese propriu
mecanismul de memorie virtuală asociază adrese virtuale cu adrese fizice, la nivel de pagini
zone de memorie virtuală:
* alocate static (la load time): cod/text, rodata, data, bss, biblioteci
* alocate dinamic (la run time): biblioteci, stivă, heap
load time: momentul în care este pornit un proces, lansarea în execuție, când folosim ./a.out în linia de comandă, când se apelează exec()
run time: în momentul în care procesul rulează, se află în execuție; bibliotecile se încarcă dinamică cu apeluri de tipul dlopen() (POSIX) sau LoadLibrary (Windows)
alocarea comandată din program (de exemplu malloc()) în cadrul unei zone înseamnă alocare de memorie virtuală

Alocarea la cerere (demand paging)

atunci când alocă o pagină virtuală nouă (folosind un apel malloc() sau mmap() sau alocând pe stivă) nu e obligatoriu să alocăm și pagină fizică
spunem că "rezervăm" o pagină de memorie virtuală
amânăm alocarea efectivă (a paginii fizice) până la primul acces (lazy allocation)
numim acest proces de tip "lazy" on-demand paging (paginare/alocare la cerere) sau, mai simplu, demand paging
demand paging permite amânarea folosirii memoriei fizice până la prima utilizare, relaxează consumul de memorie fizică

Pași în alocarea memoriei: rezervare, alocare, mapare

spunem că rezervăm pagini virtuale, fără a avea neapărat corespondent într-o pagină fizică
rezervarea înseamnă marcarea paginii virtuale ca fiind folosite, în spațiul virtual de adrese al procesului
atunci când avem un prim acces, alocăm pagina fizică
o dată cu alocarea paginii fizice realizăm asocierea între pagina virtuală și pagina fizică, adică maparea
asocierea se face prin completarea intrării în tabela de pagini: în locul aferent paginii virtuale completăm adresa paginii fizice

Internele alocării la cerere (demand paging)

tabela de pagini e indexată după pagini virtuale: conține adresa pagini fizice corespunzătoare, un bit de validitate, biți de permisiuni
tabela de pagini de interpretată la nivel hardware de MMU (Memory Management Unit)
sistemul de operare păstrează informații suplimentare despre tabelele de pagini
atunci când se rezervă primă oară o pagină virtuală (la load time sau la run time), se marchează pagina ca nevalidă în tabela de pagini, și ca "validă-nemapată" în informațiile sistemului de operare

la primul acces, MMU generează excepție (intrarea în tabela de pagini este nevalidă), excepție e capturată de sistemul de operare, sistemul de operare alocă o pagină fizică, o completează în tabela de pagini și marchează intrarea validă (și în informațiile sale interne)
excepția este numită page fault (excepție de acces la pagină)

Page fault (excepție de acces la pagină)

atunci când se face acces la o pagină marcată "nevalidă" în tabela de pagini, MMU generează "page fault" către procesor
procesorul execută un page fault handler, o rutină de tratare înregistrată de sistemul de operare
sistemul de operare implementează în page fault handler mecanismul de demand paging și alte mecanisme precum swapping sau copy-on-write
dacă o pagină este "nevalidă" în tabela de pagini și este marcată ca "nevalidă/nealocată" în informațiile sistemului de operare se generează către procesul în cauză excepție de memorie de tipul "segmentation fault"
+demo cu page fault-uri pentru demand paging

Memorie partajată

două sau mai multe procese pot partaja pagini de memorie
intrările din fiecare tabelă de pagini (a fiecărui proces) referă aceeași pagină fizică
paginile virtuale pot diferi
+ diagramă memorie partajată
este folositoare pentru partajarea codului de executabil sau pentru codul pentru biblioteci partajate: mai multe procese create din același executabil vor partaja zona de cod (zona code/text) și zona read-only (zona rodata); mai multe procese care folosesc aceeași bibliotecă partajată (shared library) vor partaja codul acelei biblioteci (zone code/text) și zona read-only (zona rodata)
nu se partajează datele bibliotecilor, sunt modificate de fiecare proces în parte și sunt proprii fiecărui proces
+ diagramă pentru partajare zone + biblioteci
util atunci când este creat un proces copil folosind fork(); inițial se partajează *tot* spațiul fizic: fiecare dintre procesul părinte și procesul copil are un spațiu virtual propriu, adică o tabelă de pagini proprie; dar spațiile virtuale referă același spațiu fizic, adică tabelele de pagini au același conținut

copy-on-write

tradițional, la fork() se crea un spațiu fizic nou, pentru noul proces, nu doar unul virtual; adică se face un duplicat al spațiului fizic și se populează în tabela de pagini a procesului copil nou creat
devine problematic pentru că de obicei după fork() se apelează exec() și se înlocuiește tot spațiul virtual și fizic, ceea ce înseamnă că duplicarea de la fork() a fost degeaba
soluția este folosirea mecanismului copy-on-write atunci când facem fork()
când mecanismul copy-on-write, imediat după fork() fiecare spațiu virtual (al procesului părinte și al procesului copil) referă același spațiu fizic; nu se alocă spațiu fizic suplimentar (în afară de cel pentru noua tabelă de pagini)
procesul părinte și copil partajează întreg spațiul de adrese; e OK pentru zone non-writable (code/text și rodata) dar nu pentru zone writable
atunci când unul dintre procese realizează o scriere, pagina corespunzătoare este duplicată, se actualizează intrarea în tabela de pagini a procesului care a făcut scrierea: referă pagina nouă; modificarea are loc doar în pagina nouă
este o formă de operație "lazy" (similar demand paging): se amână duplicarea paginii până la primul acces de scriere
duplicarea nu are loc la apel de citire
în detaliu:
* imediat după fork() paginile sunt marcate valide/read-only în tabela de pagini
* la o acțiune de scriere MMU generează page fault și se apelează page fault handler-ul
* în informațiile internele ale sistemului de operare pagina respectivă este marcată "validă/copy-on-write"
* sistemul alocă o pagină fizică nouă, duplică în ea conținutul paginii inițiale și actualizează intrarea în tabela de pagini a procesului care a realizat scrierea, pagina este marcată read-write
* este reapelată instrucțiunea de scriere care a cauzat page fault-ul inițial și acum se face modificarea efectivă în pagina fizică nou alocată
dacă se fac mai multe apeluri fork() (mai multe procese copil sau nepot sau strănepot) acestea partajează toate spațiul fizic și au paginile marcate read-only; duplicarea paginii se întâmplă atunci când un proces realizează o acțiune de scriere ducând la actualizarea intrării în tabela de pagini a procesului; intrările în tabelele de pagini ale celorlalte procese nu se modifică
principiul copy-on-write (acronimul COW) este întâlnit și în virtualizare (snapshot-restore, migrarea mașinilor virtuale) și în sisteme de fișiere (reținerea versiunilor anterioare)

Swapping

memoria fizică fiind limitată există șansa să fie nevoie să alocăm o pagină fizică dar să nu fie disponibilă
o situație este să fie terminat un proces pentru a elibera spațiul fizic folosind de acesta (out-of-memory handler, numit și OOM)
soluția folosită în sistemele de operare moderne este folosirea spațiului de swap
spațiul de swap este o zonă din memoria secundară (disc, zonă persistentă) folosită ca suport de stocare a paginilor
atunci când nu există o pagină fizică disponibilă, se alege o pagină fizică și se evacuează în spațiul de swap: swap out
pagina eliberată e marcată nevalidă în tabela de pagini și "validă-swapped out" în informațiile procesului
pagina fizică este acum mapată în spațiul virtual de adrese al procesului care a avut nevoie de ea; dacă este o pagină nouă, ideal se umple cu zero-uri, pentru a preveni information leak din primul proces
+ diagramă cu swap out
atunci când se accesează o pagină swapată, MMU generează page fault (este marcată nevalidă în tabela de pagini)
se găsește o pagină liberă (potențial se face swap out la o pagină) și se face swap in (se aduce pagina necesară din swap)
+ diagramă cu swap in
spațiul de swap e o partiție dedicată (pe Linux) sau o zonă dintr-o partiție dată (pe Windows)
dacă există o utilizare intensă a memoriei fizice apar operații de swap out/swap in dese
atunci când apar des operații de swap out/swap in, paginile sunt înlocuite foarte des și se consumă timp în schimbarea lor

numim acest fenomen thrashing, este prezent și la folosirea memoriei cache

Algoritmi de înlocuire de pagini

atunci când o pagină trebuie evacuată pe spațiul de swap (swap out), trebuie găsită o pagina potrivită
alegerea paginii de evacuat ține, în general, cont de cât de recent a fost modificată și cât de recent a fost folosită
paginile au în general un bit (dirty) care spune că a fost modificată
se preferă paginile care au fost cel mai puțin recent utilizate (least recently used)
paginile care nu au fost modificate și sunt deja pe swap nu trebuie să fie swapped out la înlocuire; conținutul lor este deja acolo
thrashing

Maparea fișierelor

pentru a simplifica lucrul cu fișierele, acestea pot fi mapate în spațiul de adresă al unui proces
adică scrierea într-o pagină virtuală conduce la scrierea în blocul corespunzător de pe disc al procesului
are loc uzual pentru fișiere executabile și biblioteci partajate: sunt mapate în spațiul virtual al proceselor
+ demo cu pmap care arată numele fișierelor
+ demo cu lsof și vizualizarea zonelor de tip "txt"
operațiile cu fișierele sunt acum operații cu memoria, nu mai sunt apeluri de sistem read/write
avantaje: overhead scăzut temporal (nu se fac apeluri de sistem) și spațial (nu se alocă buffere în user space pentru apelurile read/write)
dezavataj: fișierele trebuie să aibă dimensiunea știută pentru mapare, nu se poate crește dimensiunea (cum se întâmplă atunci când folosim write() pentru a scrie dincolo de dimensiunea fișierului)

Sumar

spațiul virtual de adrese al procesului e compus din zone statice și dinamice, zone read-only și read-write
în general, paginile fizice nu se alocă la comanda de alocare a utilizatorului ci la primul access: demand paging
tabela de pagini are marcată intrarea nevalidă, urmând ca informațiile sistemului de operare să știe că e vorba de acces nevalid sau demand paging
se fac în ordine: rezervarea unei pagini virtuale, alocarea unei pagini fizice, maparea paginii fizice la pagina virtuală
un acces la o pagină marcată nevalidă face ca MMU să transmită o excepție de acces la pagină numită page fault, care apelează un page fault handler înregistrat de sistemul de operare
zonele read-only sunt partajate între procese
inițial la fork() procesul părinte și procesul copil partajează toate paginile fizice
imediat după fork(), paginile fizice sunt marcate read-only; la primul acces are loc copy-on-write: duplicarea paginii și marcarea acesteia read-write în procesul ce a generat acțiunea de scriere
pentru a preveni epuizarea spațiului fizic de memorie, se extinde prin folosire spațiului de swap
swap out: evacuarea unei pagini fizice pe swap
swap in: readucerea unei pagini din swap în memoria fizică
dacă au loc operații de swap in și swap out pe aceeași pagină, sistemul devine ineficient, apare fenomenul de thrashing
fișierele pot fi mapate în spațiul virtual de adrese al procesului pentru eficiență temporală și spațială: este cazul fișierelor executabile și al bibliotecilor partajate

Curs 07: Securitatea memoriei

Sumar curs anterior

spațiul virtual de adrese al procesului e compus din zone statice și dinamice, zone read-only și read-write
în general, paginile fizice nu se alocă la comanda de alocare a utilizatorului ci la primul access: demand paging
tabela de pagini are marcată intrarea nevalidă, urmând ca informațiile sistemului de operare să știe că e vorba de acces nevalid sau demand paging
se fac în ordine: rezervarea unei pagini virtuale, alocarea unei pagini fizice, maparea paginii fizice la pagina virtuală
un acces la o pagină marcată nevalidă face ca MMU să transmită o excepție de acces la pagină numită page fault, care apelează un page fault handler înregistrat de sistemul de operare
zonele read-only sunt partajate între procese
inițial la fork() procesul părinte și procesul copil partajează toate paginile fizice
imediat după fork(), paginile fizice sunt marcate read-only; la primul acces are loc copy-on-write: duplicarea paginii și marcarea acesteia read-write în procesul ce a generat acțiunea de scriere
pentru a preveni epuizarea spațiului fizic de memorie, se extinde prin folosire spațiului de swap
swap out: evacuarea unei pagini fizice pe swap
swap in: readucerea unei pagini din swap în memoria fizică
dacă au loc operații de swap in și swap out pe aceeași pagină, sistemul devine ineficient, apare fenomenul de thrashing
fișierele pot fi mapate în spațiul virtual de adrese al procesului pentru eficiență temporală și spațială: este cazul fișierelor executabile și al bibliotecilor partajate

Securitatea sistemului. Securitatea memoriei

un sistem sigur funcționează conform specificațiilor
un defect duce la o funcționare necorespunzătoare: abuz/atac (răuvoitor) sau eroare (greșeală)

atacatorul urmărește: furt de informație (leak, steal, disclose), îngreunare (cripple, denial-of-service) sau obținerea controlului; obținerea controlului pentru a fura informații sau îngreunare sau pentru a genera alt atac sau pentru a folosi resurse
un defect/bug este o funcționare necorespunzătoare; dacă acest defect poate fi folosit în beneficiul atacatorului (steal, cripple, control) atunci este o vulnerabilitate; spunem că o vulnerabilitate este un defect exploatabil
un exploit este o metodă de atac a unei vulnerabilități
un vector de atac este o secvență de pași, uzual compusă din mai multe exploit-uri care duce la obținerea de beneficii concrete atacatorului
un subset al securității sistemului este securitatea aplicațiilor
securitatea aplicațiilor înseamnă securitatea acestora înainte de rulare (verificare, analiză statică, dinamică) și în timpul rulării (runtime application security)
securitatea la rulare presupune atac/apărare în folosirea resurselor
un atac la rulare presupune, în general, denaturarea fluxului normal de execuție al aplicație (altering the control flow graph: CGF)
securitatea memoriei aplicației se referă la atacuri și mijloace preventive legate de scrierea și citirea datelor și citirea și executarea codului

Fluxul de execuție

pașii pe care îi urmează aplicația în momentul rulării
fluxul de execuție (execution flow) este definit în momentul implementării într-un limbaj de programare uzual compilat în cod mașină într-un fișier executabil care apoi este rulat din memorie într-un proces
fluxul de execuție este descris prin graful fluxului de control al aplicației (control flow graph, CFG)
un nod în graf este o secvență liniară de instrucțiuni (basic block)
un arc în graf este un salt (jump, branch)
în mod obișnuit, o aplicație este sigură dacă CFG-ul este urmat corect la rulare
este, desigur, posibil, să existe un cod de tipul backdoor, plasat de dezvoltator, parte validă din CFG care să fie declanșat într-o anumită situație
mai posibil, însă, un atac presupune adăugarea unor noi arce sau noduri în CFG, adică modificarea fluxul de execuție în beneficiul atacatorului (steal, cripple, control): control flow hijack
această modificare a fluxului de execuție se realizează prin atacuri la adresa memoriei aplicației când aceasta rulează
de obicei atacurile la adresa memoriei pornesc de la o vulnerabilitate de tipul buffer overflow/index-out-of-bounds

Tipuri de zone de memorie după permisiuni

zonele de memorie pot fi: read-write, read-only, read-executable
read-executable: zone de cod/text
read-only: zonele .rodata
read-write: .data (date inițializate), .bss (date neinițializate), heap, stivă
un executabil (mapat în memoria procesului la load-time) conține .text, .rodata, .data, .bss
o bibliotecă partajată (mapată la load-time sau la run-time) conține aceleași zone .text, .rodata, .data, .bss
heap și stiva (stack) sunt create la pornirea procesului
un atacator este interesat în special de zonele read-write: ce poate să suprascrie pentru a afecta fluxul de execuție și a obține beneficii (steal, cripple, control)
un atacator se poate folosi de codul existent în zonele read-executable: code reuse

Reminder de la IOCLA: Buffer overflow. Index out of bounds

bufferele sunt zone continue de memorie care pot fi scrise și citite: sunt plasate în zone read-write
un buffer: adresă de start, dimensiune
un buffer este definit în C ca un array: un șir de caractere e un caz particular de array
în special în lucrul cu șiruri, folosirea necorespunzătoare a bufferelor poate duce la suprascrierea de date
buffer overflow: parcurgerea buffer-ului element cu element și trecerea de limita superioară (apeluri de tipul memcpy, strcpy, fgets pot face asta)
index out of bounds: accesarea unui index din afara spațiului buffer-ului (index negativ sau dincolo de dimensiunea buffer-ului)
putem spune că buffer overflow e un subcaz de index out of bounds
cauzate de erori de programare: char v[32]; fgets(v, 64, stdin);
se pot suprascrie date care afectează fluxul de execuție și oferi beneficii atacatorului
în general nu se primește segmentation fault sau alt tip de excepții pentru că paginile accesate prin overflow/index out of bounds sunt valide în spațiul virtual de adrese al procesului
+ demo cu buffer overflow care nu generează crash
în Java sau alte limbaje se primește excepție pentru că mașina virtuală face verificările, cu dezavantajul de overhead temporal

Reminder de la IOCLA: Funcționarea stivei

bufferele din funcții sunt alocate pe stivă (mai puțin cazul când sunt declarate statice); sunt dese atacuri ce folosesc vulnerabilități de tipul buffer overflow și index out of bounds pe stivă
stiva conține cadre de stivă (stack frames) pentru fiecare funcție apelată
structura unui cadru de stivă depinde de convenția de apel (calling convention)
pe x86 un cadru de stivă conține:
* parametrii funcției
* adresa de retur
* frame pointer (ebp)
* variabile locale
+ diagramă cu set de cadre de stivă
este interesant să realizăm atacuri pe stivă pentru că sunt informații critice ce pot fi suprascrise, în special adresa de retur
dacă atacatorul exploatează o vulnerabilitate de tipul buffer overflow sau index out of bounds, atunci poate suprascrie adresa de retur și poate redirecta/deraia (hijack) fluxul de control al programului în beneficiul său

buffer overflow-urile pe stiva se numesc stack buffer overflow; există și heap buffer overflow care suprascriu informații din heap

Atacuri simple de memorie. Code pointeri

pentru a deraia fluxul de execuție al unei aplicații, un atacator urmărește suprascrierea unor pointeri la zone de cod, precum adresa de retur
suprascrierea adresei de retur cu o valoare convenabilă va redirecta instruction pointer-ul la acea adresă de unde atacatorul va folosi codul dorit
code pointers sunt zone de memorie ce conțin adrese de zone de cod: adresa de retur, pointeri de funcție
din perspectiva apărătorului, acești pointeri trebuie protejați să nu fie suprascriși; atacatorul dorește să îi suprascrie
din suprascrierea unui code pointer rezultă două tipuri de atacuri:
* code reuse: suprascrierea cu o adresă deja existentă de cod (.text sau .text dintr-o bibliotecă)
* code injection: suprascrierea cu un cod scris într-o zonă read-write și executarea codului scris în acea zonă
avantaj code reuse: mai simplu, folosește resurse existente
avantaj code injection: flexibil, se poate injecta ce cod se dorește
+ demo cu suprascriere pointer de funcție
+ demo cu Stack buffer overflow pentru suprascriere pointer de funcție
+ demo cu Stack buffer overflow pentru suprascriere adresă de retur

Shellcode

o secvență de cod mașină injectată pentru a fi executată: code injection
uzual este combinată cu exploatarea unui buffer overflow și suprascrierea unui code pointer pentru a ajunge la acea zonă
pentru a fi executată zona trebuie să fie read-write (să poată fi scris shellcode-ul) și executabilă (să poată fi executat)
convențional un shellcode deschide un shell (exec("/bin/sh")), dar poate fi folosit la orice: deschis un socket, schimbat permisiuni, citit un fișier
un shellcode conține cod mașină și folosește apeluri de sistem, nu apeluri de funcții de bibliotecă; nu ar ști unde este plasat în memorie și unde se găsesc adresele funcțiilor
+ exemplu de shellcode (în limbaj de asamblare)
+ demo cu Stack buffer overflow cu shellcode
+ demo cu Stack buffer overflow cu shellcode pe stivă

Mecanisme defensive

pentru ca un shellcode să ruleze este nevoie de:
* o vulnerabilitate (precum buffer overflow) care să permită suprascrierea unui code pointer
* o intrare în program care să ducă la citirea shellcode-ului într-o zonă read-write
* executarea shellcode-ului din zona în care a fost suprascris
metodele dinainte de deploy pot duce la eliminarea vulnerabilităților
ne referim în particular la metode din momentul rulării aplicației (runtime application security)
în general se urmează un pattern: atac, soluție defensivă, bypass
input validation: valiarea intrării pentru a nu permite date binare acolo unde ar fi nevoie de text; bypass: shellcode alfanumeric
stack guard, stack canary, stack smashing protection: plasarea unei valori între buffer și adresa de retur
safe stack: plasarea code pointerilor și a datelor critice pe o stivă dedicată
DEP (data execution prevention): o zonă read-write nu poate fi și executabilă
ASLR (address space layout randomization): zonele de memorie sunt plasate la adrese aleatoare și nu se poate ușor găsi adresa cu care să se fie suprascrie un code pointer
CFI (control flow integrity): asigurat că este respectat fluxul de execuție al programului și nu se adaugă noi noduri sau arce în CFG

Stack Guard

Stack Smashing Protection (SSP) sau stack canary
se plasează o valoare între buffer și adresa de retur
suprascrierea adresei de retur prin buffer overflow va însemna suprascrierea valorii canar, lucru ce va fi detectat la părăsirea funcției
canarul este plasat într-o zonă dedicată
se pot suprascrie în continuare variabile locale
+ demo cu program compilat cu SSP și fără SSP, văzut codul în limbaj de asamblare
mic cost de performanță, se poate aplica SSP selectiv pe funcțiile ce conțin pointeri
bypass: se suprascrie canarul cu el însuși; se plasează uzual 0x00 și 0x0a în canar pentru a "opri" funcții de lucru cu șiruri din suprascriere
bypass: se suprascrie handle-ul de tratare a suprascrierii canarului
+ demo cu stack canary bypass

AddressSanitizer

https://github.com/google/sanitizers/wiki/AddressSanitizer
soluție integrată de securizare a memoriei
overhead semnificativ, bun în faza de dezvoltare
mai multe tipuri de "sanitizers"
integrat în compilator

Safe Stack

code pointerii sunt plasați într-o zonă dedicată: safe stack
buffer overflow-ul nu suprascrie code pointeri
se modifică stack frame-ul unei funcții
e nevoie de modificarea modulul în care compilatorul generează codul de tip prologue și epilogue al funcțiilor

Data Execution Prevention

atacurile cu shellcodes (de code injection) sunt posibile dacă zonele read-write pot fi și executate
există suport în hardware pentru marcarea paginilor ca neexecutabile
stiva, heap-ul, data, bss sunt marcate ca neexecutabile: se poate injecta cod, nu se poate executa
bypass: se folosește alt tip de atac (code reuse) care să remapeze o zonă read-write ca executabilă (folosind un apel de tipul mprotect()/VirtualProtect())

Code Reuse. Return-to-libc

refolosirea codului existent în memorie în zonele read-execute: .text sau .text din biblioteci
se pot folosi funcții întregi sau părți din funcții
return-to-libc înseamnă apelul unei funcții din biblioteca standard C; un apel uzual este system("/bin/sh"); pentru deschiderea unui shell
variații de code reuse sunt return-oriented programming (înlănțuirea de secvențe mici care se încheie în ret) sau jump-oriented programming (înlănțuirea de secvențe mici care se încheie în instrucțiuni precum jmp *eax)
trebuie știute adresele de cod unde se face saltul

ASLR

măsură defensivă care face dificilă descoperirea de adrese: adrese de cod, adrese în stivă
se plasează aleator zone din spațiul de adrese al procesului: heap, stivă, biblioteci
+ demo cu ASLR activat și dezactivat
dacă un executabil este compilat cu suport de PIE (Position Independent Executable) se plasează aleator și zonele executabilului (.text, .rodata, .data, .bss)
+ demo cu executabil cu PIE și fără PIE
bypass: pe sistemele pe 32 de biți se poate face brute force până când se "nimerește" adresa
bypass: memory disclosure: se obține (leak) o adresă care ajută la calculul altor adrese

CFI

validarea fluxului de execuție și întreruperea programului dacă apar arce sau noduri noi în CFG
overhead semnificativ, util în anumite situații sau în etapa de testare
bypass: data-oriented attacks: atacuri care folosesc abuziv fluxuri existente în CFG dar care nu ar trebui permise
ideal este ca în faza de dezvoltare/testare să se acopere cât mai mult din CFG-ul programului (CFG coverage)

Mai multe detalii

Security Summer School: http://security.cs.pub.ro/summer-school/wiki
Compilatoare (C3, anul 4 semestrul 1): https://ocw.cs.pub.ro/courses/cpl
Computer and Network Security: http://ocw.cs.pub.ro/cns
Wargame/CTF sites: http://captf.com/practice-ctf/

Sumar

un sistem este sigur dacă funcționează conform specificațiilor
o componentă este securitatea aplicațiilor la rulare (runtime application security)
o parte importantă este securitatea memoriei
spațiul virtual de adrese al procesului este compus din zone de memorie cu permisiuni diferite: read-write, read-only, read-executable
o aplicație are un flux de execuție descris de control flow graph (CFG)
există două tipuri de atacuri: atacuri ce adaugă noi noduri în graf (code injection attacks) sau care adaugă arce și refolosesc graful în moduri benefice atacatorului (code reuse attacks)
vulnerabilitățile "standard" sunt buffer overflow și out of bounds errors
un atacator urmărește suprascrierea de informații pentru a modifica CFG-ul programului
interesant este de realizat stack buffer overflow attacks, pentru că se suprascrie adresa de retur
adresa de retur este un code pointer, un pointer la o zonă de cod; suprascrierea unui code pointer oferă atacatorului posibilitatea controlului fluxului de execuție (control flow hijack)
partea de code injection presupune injectarea unei secvențe de cod în memoria procesului: shellcode
mecanisme de protecție sunt: stack guard, data execution prevention, address space layout randomization, control flow integrity
în general se urmărește schema: vulnerabilitate/problemă, atac, metodă preventivă, bypass la metodă preventivă

Curs 08: Fire de execuție

Sumar curs anterior

un sistem este sigur dacă funcționează conform specificațiilor
o componentă este securitatea aplicațiilor la rulare (runtime application security)
o parte importantă este securitatea memoriei
spațiul virtual de adrese al procesului este compus din zone de memorie cu permisiuni diferite: read-write, read-only, read-executable
o aplicație are un flux de execuție descris de control flow graph (CFG)
există două tipuri de atacuri: atacuri ce adaugă noi noduri în graf (code injection attacks) sau care adaugă arce și refolosesc graful în moduri benefice atacatorului (code reuse attacks)
vulnerabilitățile "standard" sunt buffer overflow și out of bounds errors
un atacator urmărește suprascrierea de informații pentru a modifica CFG-ul programului
interesant este de realizat stack buffer overflow attacks, pentru că se suprascrie adresa de retur
adresa de retur este un code pointer, un pointer la o zonă de cod; suprascrierea unui code pointer oferă atacatorului posibilitatea controlului fluxului de execuție (control flow hijack)
partea de code injection presupune injectarea unei secvențe de cod în memoria procesului: shellcode
mecanisme de protecție sunt: stack guard, data execution prevention, address space layout randomization, control flow integrity
în general se urmărește schema: vulnerabilitate/problemă, atac, metodă preventivă, bypass la metodă preventivă
+ demo de atac/bypass de Stack Smashing Protection

Tipuri de acțiuni în sisteme de calcul

sistemele de calcul oferă sprijin pentru realizarea de acțiuni ale utilizatorilor
o acțiune: obținerea unor date de intrare, prelucrarea lor, furnizarea unor date de ieșire (sau părți din acestea)
exemple:
* un server web obține date de pe disc (fișier, bază de date) și la transferă la rețea (I/O intensive); un server web cu procesare (de exemplu modul PHP sau Ruby sau Python) are și o fază de procesare (preponderent I/O intensive)
* un editor în general obține date de la tastatură și le scrie pe disc într-un fișier; face prelucrare atunci când face căutări sau înlocuiri (preponderent I/O intensive cu burst-uri de CPU intensive)
* utilitarul grep obține date din fișier, caută prin ele expresii regulate și scrie rezultatul în alt fișier (CPU intensive)
* un procesor video (parte dintr-un player sau editor video) face encoding/decoding la datele dintr-un fișier (CPU intensive)
acțiunile (sau sub-acțiunile) sunt I/O intensive sau CPU intensive
I/O intensive: operează des cu discul, placa de rețea sau alte forme de I/O
CPU intensive: folosesc des procesorul

Sisteme de calcul multi-core / multi-procesor

mai multe procesoare ne permit să:
* rulăm mai multe acțiuni diferite, câte una pe procesor
* să rulăm aceeași acțiune în paralel pe mai multe procesoare
în general o acțiune este abstractizată de un proces, care este planificat pe un procesor
pe sistemele de operare multi-procesor putem rula mai multe procese diferite, sporind astfel productivitatea sistemului (throughput); alternativa ar fi să rulăm pe rând fiecare proces (multitasking, scheduling): un procesor rulează un editor, altul rulează un server web, altul un procesor video, altul un browser
pentru a rula aceeași acțiune în paralel pe mai multe procesoare, avem opțiunea să creăm mai multe procese de același tip (prin fork): mai multe procese de tip server web, fiecare servind câte o conexiune; mai multe procese de tip procesor video, fiecare lucrând pe o parte din datele de intrare
dezavantaje folosire procese
* pentru aceeași acțiune ai nevoie de date comune, mai dificil de partajat (se poate cu memorie partajată, dar e nevoie de un efort)
* overhead de memorie pentru fiecare proces creat: tabelă de pagini nouă, structuri interne în sistemul de operare
* overhead-ul de creare nu e atât de mare: se creează o tabelă de pagini nouă și structuri interne (se face relativ repede)
* overhead de schimbare de context: schimbare de tabele de pagini, flush la TLB

Thread-uri (fire de execuție)

thread-urile sunt o variantă simplă a proceselor pentru executarea de acțiuni: se mai numesc lightweight processes (LWP)
un proces are unul sau mai multe thread-uri
thread-urile sunt o abstractizare doar pentru procesor (acțiune); spațiul virtual de adrese, descriptori de I/O aparțin procesului și sunt informații partajate între thread-uri
thread-urile compensează dezavantajele folosirii proceselor:
* sunt ușor de partajat datele: tot spațiul virtual de adrese al unui proces este partajat între thread-urile aceluiași proces
* thread-urile se creează mai ușor (se creează o structură internă) și au overhead de memorie redus (nu se creează o un nou spațiu de adrese, deci nu se creează o nouă tabelă de pagini)
* schimbarea de context are penalizare mai mică atunci când se schimbă thread-uri ale aceluiași proces
+ demo timp de creare thread-uri și procese
în general thread-urile măresc productivitatea/throughput-ul sistemului: mai multă procesare și mai multe rezultate pe unitatea de timp

thread-urile simplifică programarea atunci când e nevoie de date partajate: nu e nevoie de un API dedicat pentru memorie partajată
folosirea thread-urilor are dezavantaje:
* izolare: partajează spațiul virtual de adrese și o problemă a unui thread corupe întreg spațiul de adresă; un defect (posibilă vulnerabilitate) afectează toate thread-urile
* sincronizare: partajarea spațiului de adrese poate duce la corupere de date dacă un thread folosește datele altui thread
* depanare dificilă: o problemă de programare va fi mai greu identificată într-un program multithreaded: poate fi o problemă cauzată de un thread în memoria altui thread; cu atât mai dificil este când se folosesc biblioteci externe (cod neimplementat de programator)
în cazul unui procesor de video, este recomandat să avem o implementare multi-threaded, care va partaja datele prelucrate și va folosi mai multe core-uri ale sistemului
un server web poate fi multi-proces sau multi-threaded după cum e mai mult accentul pus pe eficiență/throughput sau pe securitate/izolare

Threads-uri vs. procese. Atribute ale unui thread

spunem că un proces abstractizează memoria prin spațiul virtual de adrese, I/O prin descriptori de fișiere și procesorul prin thread-uri; un procesor poate avea unul sau mai multe thread-uri
un thread abstractizează doar procesorul
un thread este pornit în același spațiu de adrese indicându-i-se o funcție de la care începe execuția
un proces este pornit dintr-un executabil (exec) sau din procesul curent din punctul curent (fork)
un thread este abstractizarea cea mai simplă a contexului de execuție; un thread definit simplist printr-un instruction pointer (ce are de executat) și un stack pointer (o stivă, ce a executat până acum)
schimbarea de context presupune schimbarea unui thread (instruction pointer, stack pointer, celelalte registre) cu alt thread
dacă thread-urile aparțin unor procese diferite e nevoie de schimbarea spațiului de adrese (schimbare de tabelă de pagini, flush la TLB)
un thread este definit de un TCB (Thread Control Block) conținând în general informații de execuție: thread id (TID), stare, stivă, pointer la proces / spațiul de adrese din care face parte, timp de rulare, prioritate
un proces are un thread principal (main); exprimarea "procesul execută o acțiune/instrucțiune" este improprie; spunem că "un thread al procesului (sau thread-ul principal) execută o acțiune/instrucțiune"

Crearea și încheierea unui thread

când un thread este creat i se indică funcția ce o va rula
la crearea unui thread i se alocă o stivă și se începe rularea acelei funcții
spunem că un thread se activează; e vorba de activarea unui thread
+ demo cu creare de thread-uri și spațiu de adresă
thread-ul devine o entitate planificabilă (context de execuție); se poate bloca, îi expiră cuanta
un thread își încheie execuția:
* când se încheie funcția
* când se încheie procesul curent (un thread al său apelează exit() sau apare o eroare/excepție)
* când se apelează o funcție specifică de închidere a thread-ului (pthread_exit())
similar proceselor există un apel care așteaptă încheierea unui thread pentru a recupera informațiile de ieșire; operația se numește join
un thread poate întoarce o valoare la încheierea sa, recuperată cu join; așa cum procesul întoarce un cod de ieșire, recuperat cu wait

Partajarea spațiului de adrese între thread-uri

thread-urile partajează spațiul de adrese al procesului
spunem că thread-urile au memoria partajată
când un proces nou este creat cu fork() acesta are o copie a tabelei de pagini care referă același spațiu de adrese, dar se aplică copy-on-write la acces de scriere
când un thread nou este creat, tot spațiul de adrese este partajat: o modificare făcută de un thread e vizibilă în alt thread
+ demo cu partajare informație între procese și thread-uri
spunem că thread-urile nu partajează: stiva, o zonă dedicată numită TLS (thread local storage) sau thread specific data
afirmația de mai sus este improprie: un thread are acces la stiva altui thread dacă folosește construcții care permit asta; sistemul de operare nu (poate) face enforce ca un thread să nu modifice stiva altui thread; e vorba de același spațiu de adrese; similar și pentru TLS
+ demo cu stiva unui thread în spațiu de adrese
un thread are stivă proprie, deci variabilele locale sunt proprii fiecărui thread
dacă are nevoie de variabile globale proprii (ca să nu transmită parametri între funcții) un thread folosește TLS; TLS este o zonă dedicată unde fiecare thread are o referință a acelei variabile
cel mai simplu mod de a aloca o variabilă în TLS este folosirea atributului __thread în API-ul POSIX
+ demo cu TLS/__thread
pentru accesul coerent la date partajate este nevoie de folosirea de primitive de sincronizare (mai multe în cursul 9: Sincronizare)

Implementarea thread-urilor

API-ul de lucru cu thread-uri (politica) include:
* funcție de crearea unui thread
* funcție de așteptare/join a unui thread
* funcție de identificare a unui thread (aflare TID)
* funcție de închidere a unui thread
* primitive de sincronizare
implementarea din spate și partea de planificare a thread-urilor (mecanismul) este independentă de API
implementarea poate fi: kernel-level threads și user-level threads (sau green threads, fibers)

kernel-level threads înseamnă suport la nivelul sistemului de operare; thread-urile sunt entitățile planificabile la nivelul sistemului de operare; sunt folosite de planificatorul sistemului de operare, au cuantă de execuție, stare de execuție, sunt parte din coada READY
kernel-level threads pot folosi procesoarele sistemului: mai multe thread-uri ale aceluiași proces pot rula simultan pe un sistem multi-procesor
schimbarea de context necesită intervenția nucleului; e nevoie de suport în kernel
user-level threads sunt o implementare completă în user-space; nu este nevoie de suport în kernel
planificatorul este implementat în user-space
o operație blocantă a unui user-level thread blochează tot procesul; soluția este să se folosească operații I/O asincrone
se mai numesc green thread în implementările de mașini virtuale (de exemplu JVM)
fibrele sunt thread-uri user space folosite cu planificare cooperativă (yield)
avantaje kernel-level threads: suport complet multi-procesor, bune pentru acțiuni CPU intensive
avantaje user-level threads: nu necesită suport în kernel, timp de activare scurt, performanță ridicată pentru acțiuni I/O intensive
pentru a reduce timpul de activare se folosesc modele de tipul thread-pool (precum boss-workers sau worker-threads de la APD)

Internele implementării thread-urilor

un thread este implementat ca o structură TCB (thread control block)
o posibilă implementare este o listă de TCB-uri referită de PCB (structura procesului); așa este în Windows: EPROCESS are o listă de ETHREADS
o altă implementare este o structură pentru address space și thread-uri care pot fi atașate de adress space; nu există structură de proces efectivă; se întâmplă la microkernel-ul L4
în Linux un thread sau un proces sunt reprezentate de structura task_struct; dacă două structuri partajează spațiul de adrese spunem că sunt thread-uri ale aceluiași proces
în user-space implementarea este specifică bibliotecii de thread-uri sau mașinii virtuale

Sumar

acțiunile în sistemul de calcul sunt de lucru cu I/O și prelucrare: I/O intensive și CPU intensive
în mod tradițional, acțiunile sunt încapsulate în procese
pe un sistem multi-core putem executa mai multe acțiuni (procese) diferite sau un proces clonat și cu memorie partajată între el și procesele copil
dezavantajele proceselor sunt overhead de creare, planificare și consum de memorie (heavyweight)
thread-urile sunt lightweight processes cu timp de creare și rulare (activare) rapid, partajează spațiul de adresă
thread-urile au probleme din cauza lipsei de izolare între ele: corupere de memorie, coruperea întregului proces
un thread abstractizează procesorul: stivă/stack pointer și instruction pointer
un thread e definit de un TCB (thread control block)
activarea unui thread se face prin rularea unei funcții specifice thread-ului
întreg spațiul de adresă este partajat între thread-uri ale aceluiași proces; TLS (Thread Local Storage) și stiva sunt specifice fiecărui thread, dar pot fi în continuare accesate de un alt thread
thread-urile sunt implementate în forma kernel-level threads sau user-level threads
kernel-level threads au suport complet la nivelul nucleului, pot folosi suportul multiprocesor
user-level threads nu au suport în kernel (sunt implementate în user space), dar au timp de activare mai rapid

Curs 09: Sincronizare

Sumar curs anterior

acțiunile în sistemul de calcul sunt de lucru cu I/O și prelucrare: I/O intensive și CPU intensive
în mod tradițional, acțiunile sunt încapsulate în procese
pe un sistem multi-core putem executa mai multe acțiuni (procese) diferite sau un proces clonat și cu memorie partajată între el și procesele copil
dezavantajele proceselor sunt overhead de creare, planificare și consum de memorie (heavyweight)
thread-urile sunt lightweight processes cu timp de creare și rulare (activare) rapid, partajează spațiul de adresă
thread-urile au probleme din cauza lipsei de izolare între ele: corupere de memorie, coruperea întregului proces
un thread abstractizează procesorul: stivă/stack pointer și instruction pointer
un thread e definit de un TCB (thread control block)
activarea unui thread se face prin rularea unei funcții specifice thread-ului
întreg spațiul de adresă este partajat între thread-uri ale aceluiași proces; TLS (Thread Local Storage) și stiva sunt specifice fiecărui thread, dar pot fi în continuare accesate de un alt thread
thread-urile sunt implementate în forma kernel-level threads sau user-level threads
kernel-level threads au suport complet la nivelul nucleului, pot folosi suportul multiprocesor
user-level threads nu au suport în kernel (sunt implementate în user space), dar au timp de activare mai rapid

Resurse comune

facilitează interacțiunea între procese și thread-uri
un thread trebuie să aibă în vedere că nu este singurul care accesează; în caz contrar apar inconsevențe sau date corupte
dacă toate thread-urile citesc, nu e nici o problemă; când cel puțin unul scrie apar probleme:
* care este informatia corectă: cea de dinainte de scriere sau de citire
când scriu două thread-uri, e posibil ca operațiile de scriere să nu fie atomice, să se intercaleze rezultând în date inconsecvente:
* dacă un tread face a++, și altul a++ e posibil ca rezultatul final să fie a_initial+1 în loc de a_initial+2
problemele sunt mai mari la structuri de date mai mari
+ demo cu folosirea listelor fără acces exclusiv
problemele sunt greu de depanat, nu sunt deterministe
probleme ce pot apărea
* acces concurent de scriere la date comune
* acces de citire când datele nu au fost încă actualizate, sau acces după ce datele au fost curățate/refolosite: race conditions
* pierderea notificărilor și așteptare nedefinită
+ demo cu TOCTOU

nevoie de mecanisme de acces exclusiv la date, de serializare/atomizare a unor acțiuni și de ordonare a acțiunilor (notificare și așteptare)
+ de ce close(1) și open() sau close(1) și dup(3) sunt problematice dar dup2(3, 1) nu
trebuie ca toate thread-urile să respecte aceeași politică: dacă folosim biblioteci care nu sunt conștiente de asta, vor fi probleme
+ demo cu reentranță din cursul 08
dacă nu folosim biblioteci thread safe, nu folosim thread-uri; în Python există un GIL (Global Intepreter Lock) are serializează tot accesul

Primitive de sincronizare

reminder de la APD
pentru mecanisme de acces exclusiv și de atomizare de acțiuni: variabile atomice, mutex (lock / unlock), spinlock (lock / unlock)
pentru ordonarea acțiunilor: variabile de condiție (wait, notify), semafoare (up, down), monitoare (enter, leave, wait, notify), cozi de așteptare (wait, wake_up)
asigură funcționarea corectă a programului, dar:
* e dificil de avut în vedere toate cazurile și folosit cum trebuie toate mecanismele
* folosirea necorespunzătoare duce la probleme și mai greu de depanat
* codul serial încetinește programul (legea lui Amdahl)
* funcțiile de sincronizare au overhead
ideal este de micșorat folosirea primitivelor de sincronizare prin partiționarea datelor
regiunile critice sunt cele în care sunt date la comun sau operații care trebuie atomizate:
* regiuni de granularitate mică înseamnă că au cod serial puțin dar overhead-ul de lock()/unlock() este semnificativ
* regiuni de granularitate mare însemnă că overhead-ul de locking e mic, dar o parte mare e serializată (și se aplică legea lui Amdahl)
+ demo cu granularitate mică sau mare
ne conentrăm în continuare pe interne, partea de utilizare și bune practici o știți de la APD, o veți aprofunda la ASC și APP (anul 4 C1)

= Internele primitivelor de sincronizare

pentru ca primitivele de sincronizare să funcționeze, implementarea lock() și unlock() trebuie să fie atomică; oul și găina: cum implementăm lock() și unlock() atomic fără să folosim primitive de sincronizare
întreruperile pot întrerupe fluxul de execuție al unui program în orice moment; pe sistemele multicore, diferitele core-uri pot "intercala" comunicarea datelor pe magistrala partajată
instrucțiunile de procesor pot să nu fie atomice
instrucțiunea a += 5 se traduce în add [ebp-20], 5: atomică pe single core (read-update-write), dar neatomică pe multi-core (poate fi întreruptă de alt core, magistrala este partajată)
instructiunea a += 5 se traduce pe ARM în load r1, r2; add r1, r1, 5; store r2, r1; neatomică

atomizarea operațiilor se realizează la nivel hardware:
* x86 single core: nu e nevoie
* x86 multi core: lock pe magistrală: prefixul lock în fața unei operații
* ARM: operații tranzacționale: ldrex, strex; dacă nu reușește eșuează și încerci iar
în GCC se folosește __sync_fetch_and_add() ca wrapper peste cazurile de mai sus
+ demo cu operații atomice (directorul sum-threads)
+ demo cu sum-threads-arm; toolchain-ul de ARM se ia de aici:
https://releases.linaro.org/components/toolchain/binaries/latest-7/aarch64-elf/gcc-linaro-7.4.1-2019.02-x86_64_aarch64-elf.tar.xz

Implementarea lock (spinlock)

implementare naivă:

```
lock = 0; /* init */

while (lock == 1)
; /* do nothing */
lock = 1; /* get lock */
```
---
poate apărea un TOCTOU; este nevoie de atomizarea comparației și inițializării
procesoarele oferația de tipul compare-and-swap / compare-and-exchange: CAS(lock, 0, 1):

```
if value == to_compare
  value = to_update
return initial_value
```

```
while (CAS(lock, 0, 1) == 1)
  ;
```

+ demo cu operații atomice și spinlock (directorul lock)
reparcurs demo cu sum-threads

Spinlock vs mutex

spinlock-ul este primitiva de bază pentru asigurarea atomicității
face busy waiting în așteptarea eliberării, se bazează pe compare-and-swap

mutex-ul este o primitivă mai complexă, cu o coadă de thread-uri în așteptare; dacă mutex-ul este luat, thread-ul intră în sleep
pentru protejarea structurilor interne, un mutex are un spinlock
spinlock-ul este util pentru regiuni critice de mici dimensiuni
mutex-ul este util pentru regiuni critice mai mari sau unele în care thread-ul de blochează
+ demo cu overhead de sincronizare spinlock vs mutex (directorul spinlock-mutex)
de avut în grijă la cache thrashing pe multicore pentru spinlock-uri: atunci când un spinlock este folosit pentru thread-uri de pe mai multe core-uri aceste pot "muta" variabila spinlock-ului de pe un core pe altul afectând performanța cache-ului

Performanța transferului. Producător-consumator

componente hardware sau software comunica și își transferă informații: placă de rețea - procesor, server - client, dispozitiv de I/O - memorie
una (unele) produce (produc) informație, alta (altele) consumă
adesea una este mai rapidă decât cealaltă; există riscul ca cea mai înceată să dicteze comunicația
pentru aceasta se folosește buffering între cele două: cea care produce pune elemente în buffer, cealaltă le consumă; dacă e lent consumatorul, producătorul are spațiu de depunere; dacă e lent producătorul, consumatorul poate consuma din ce s-a strâns până atunci
în mod uzual buffer-ul folosit este un buffer circular: un buffer cu două capete: unul de citire și unul de scriere
cele două capete cresc independent: capătul de citire crește atunci când acționează consumatorul, capătul de scriere când acționează producătorul
când capătul de scriere a ajuns la capătul de citire, buffer-ul este plin
când capătul de citire a ajuns la capătul de scriere, buffer-ul este gol
accesul la buffer trebuie să fie protejat
+ demo cu buffer circular

Sumar

datele comune permit comunicare rapidă dar pot genera probleme
problemele apar când avem cel puțin un thread care scrie
pot să apară și probleme de race-condition când două acțiuni nu sunt atomizate și se interpune un alt thread care șterge / modifică o informație critică
zona care trebuie să fie atomizată este regiunea critică
primitivele de sincronizare urmăresc atomizarea accesului la o dată sau a unor operații (mutex, spinlock) sau ordonarea operațiilor (variabile condiție, semafor)
granularitatea regiunii critice afectează performanța sistemului: overhead de metode de sincronizare și zonă serială în cod
operațiile se atomizează la nivel hardware
diferă comportamentul pe single core și multi core și între arhitecturi
pe sistemele multi core o acțiune atomică single core este neatomică din cauza magistralei partajate

implementarea operației de lock() (spinlock) este realizată cu ajutorul operații compare-and-swap (CAS) prezentă pe toate arhitecturile
pe sistemele multi core operația CAS este prevăzută de lock pe magistrală
pe ARM accesul exclusiv la magistrală se realizează cu ajutorul unor instrucțiuni tranzacționale (de tipul totul sau nimic)
spinlock-urile (mai simple) sunt folosite pentru regiuni critice mici, mutex-urile (mai complexe) pentru regiuni critice mari sau acolo unde thread-ul se va bloca
pentru comunicarea între componente (hardware sau software) se folosește modelul consumator producător
diferența de viteză este compensată uzual de un buffer circular

Curs 10: Dispozitive de intrare/ieșire

Sumar curs anterior

datele comune permit comunicare rapidă dar pot genera probleme
problemele apar când avem cel puțin un thread care scrie
pot să apară și probleme de race-condition când două acțiuni nu sunt atomizate și se interpune un alt thread care șterge / modifică o informație critică
zona care trebuie să fie atomizată este regiunea critică
primitivele de sincronizare urmăresc atomizarea accesului la o dată sau a unor operații (mutex, spinlock) sau ordonarea operațiilor (variabile condiție, semafor)
granularitatea regiunii critice afectează performanța sistemului: overhead de metode de sincronizare și zonă serială în cod
operațiile se atomizează la nivel hardware
diferă comportamentul pe single core și multi core și între arhitecturi
pe sistemele multi core o acțiune atomică single core este neatomică din cauza magistralei partajate
implementarea operației de lock() (spinlock) este realizată cu ajutorul operații compare-and-swap (CAS) prezentă pe toate arhitecturile
pe sistemele multi core operația CAS este prevăzută de lock pe magistrală
pe ARM accesul exclusiv la magistrală se realizează cu ajutorul unor instrucțiuni tranzacționale (de tipul totul sau nimic)
spinlock-urile (mai simple) sunt folosite pentru regiuni critice mici, mutex-urile (mai complexe) pentru regiuni critice mari sau acolo unde thread-ul se va bloca
pentru comunicarea între componente (hardware sau software) se folosește modelul consumator producător
diferența de viteză este compensată uzual de un buffer circular

Dispozitive și interfețe de I/O

cele trei resurse ale sistemului de calcul sunt: procesorul, memoria și I/O
procesorul este folosit pentru prelucrarea datelor; memoria este folosită pentru a reține datele și codul proceselor (și nucleului), I/O este folosit pentru comunicarea cu exteriorul
un proces care nu are mod de comunicare cu exteriorul este inutil (nu citește nimic, nu generează nimic)
procesele comunică prin interfețe de I/O cu:
* utilizatorul: tastatură, mouse, monitor, imprimantă, cameră video
* alte procese
 * de pe același sistem: folosind fișiere, pipe-uri/fifo-uri, socketi UNIX
 * sau pe alte sisteme: folosind rețeaua), dispozitive unde sunt
* dispozitive dedicate:
 * de informare/investigare: senzori (elemente care culeg informație din exterior): de temperatură, umiditate
 * de control: actuatori (elemente mecanice sau electrice care execută o acțiune): de obicei în sistemele robotice sau industriale
 * de stocare: HDD, USB drive, CD-ROM
comunicarea cu dispozitivele I/O este critică și este mediată de sistemul de operare; sistemul de operare are grijă ca datele transferate către/de la proces să nu fie interpelate de alt proces; comunicarea cu I/O necesită, de obicei, tranziția în modul privilegiat (kernel space)
pe lângă multiplexarea/demultiplexarea datelor, nucleul oferă mecanisme de "scheduling" a transmiterii datelor pentru îmbunătățirea performanței
pentru I/O vorbim în general, de niveluri: nivelul hardware, nivelul sistemului de operare / kernel, nivelul aplicație / interfața user space; le vom parcurge bottom up

I/O în hardware

recapitulare de la PM
la nivel hardware, comunicarea cu dispozitivele de I/O are loc pe axa: procesor -> magistrală -> controller -> dispozitiv
magistrala (de I/O) reprezintă un număr de linii electrice pe care se transmit datele/comenzile/informațiile de la/către dispozitiv
datele ajung la un controller, un chip dedicat care are informații despre specificul fizic/electric al dispozitivului și îl comandă corespunzător
controller-ul dispune de zone de memorie (registre) care reții datele sau comenzile sau stările; aceste registre sunt adresate de procesor printr-o schemă dedicată, similar memoriei
registrele sunt de 4 tipuri:
* citire
* scriere
* comandă
* stare
procesorul folosește schema de adresare pentru a scrie sau a citi informații din registrul dedicat
sunt folosite două scheme de adresare a registrelor controller-ului:
* port-mapped I/O (isolated I/O): un spațiu dedicat de adresare este folosit
* memory-mapped I/O: un spațiu din adresarea *fizică* a procesorului este folosit pentru I/O; adică o parte din adresele fizice ajung în memoria principală, altele ajung în I/O
+ diagramă cu port-mapped I/O și memory-mapped I/O
memory-mapped I/O este folosit cel mai mult, simplifică circuitele interne pentru că folosește o singură schemă de adresare; de avut în vedere că datele ajung în memoria cache și este posibil să fie reordonate
memory-mapped I/O a crescut în prezență o dată cu trecerea la arhitecturi pe 32 de biți și 64 de biți când spațiul fizic de adresare a devenit mai generos
pe x86 există și port-mapped I/O (instrucțiunile IN și OUT din x86 assembly) și memory-mapped I/O
+ de văzut /proc/ioports și /proc/iomem (sudo cat ...); de făcut suma zonelor unde apare "system RAM" la /proc/iomem

Comunicarea hardware cu dispozitivele I/O

comunicarea cu dispozitivele de I/O trece prin controller: procesorul scrie în registrele controller-ului, apoi controller-ul le transferă dispozitivului; sau procesorul citește din registrele controller-ului datele primite de la dispozitiv
procesorul trebuie să știe când poate să citească sau să scrie; riscă să citească date nevalide sau să suprascrie datele care nu au fost trimise; procesorul trebuie să se sincronizeze cu controller-ul
sincronizarea se realizează în două moduri:
* polling: procesorul investighează constant starea controller-ului și când are de citit/scris date, operează
* întreruperi: procesorul este notificat de controller printr-un semnal electric pe o linie dedicată de disponibilitatea de a primi sau transmite date
în general preferăm folosirea întreruperilor, pentru că nu țin procesorul ocupat
întreruperile sunt însă dezavantajoase la un trafic mare de date (de exemplu plăcile de rețea 10Gbit, 40Gbit) și atunci se preferă polling; DPDK, un framework de prelucrare rapidă de pachete, folosește polling
se poate tranzita din polling în întrerupere în funcție de situație (la trafic mic întreruperi, la trafic mare polling)
întreruperile sunt livrate de controller către controller-ul de întreruperi al procesorului (APIC pe x86, GIC pe ARM) care apoi livrează întreruperea procesorului; întreruperile sunt livrate pe o "linie" de întrerupere care identifică astfel întreruperea (IRQ); primirea unei întreruperi duce la rularea unei rutine de tratare a întreruperii (ISR: Interrupt Service Routing) aflată în memoria sistemului de operare în tabela vectorilor de întrerupere (IDT: Interrupt Descriptor Table)
+ sudo cat /proc/interrupts
pentru a minimiza interacțiunea cu procesorul, anumite dispozitive folosesc DMA (Direct Memory Access): un chip dedicat realizează transferul unor blocuri mari de date din controller în memoria principală la comanda procesorului fără ca procesorul să se ocupe de realizarea transferului
procesoarele I/O sunt componente specializate care au și rol de comandă (au instrucțiuni proprii) pentru realizarea transferului; mai mult decât DMA au secvențe de instrucțiuni pe care le execută; de exemplu PLX IOP 480 încorporează un core PowerPC
+ diagramă cu procesor I/O

Device drivere

rutinele de tratare ale întreruperilor și părțile ce realizează transferul de la / către controller sunt parte din sistemul de operare (kernel), mai precis din drivere
un driver este o componentă ce rulează uzual în kernel space care se ocupă de comunicarea sistemului de operare cu dispozitivul; este în mod tipic un modul de kernel
+ /proc/devices arată asocierea între anumite dispozitive (identificate prin major) și driverul corespunzător (identificat prin nume)
în general există o interfață generică de comunicare cu device driverele, iar device driverele traduc această interfață în comunicarea cu controller-ul dispozitivului
la device driver ajung întreruperile de la hardware și cererile de comunicare declanșate prin apel de sistem din user space (de exemplu apeluri de forma read() sau write())
device driver-ul este astfel o colecție de funcții apelate la primirea întreruperilor sau a unor cereri de la aplicații
mai multe despre drivere la cursul de Sisteme de operare 2 (anul 4 C3, semestrul 2)

Niveluri intermediare pentru prelucrarea I/O

comenzile și datele transferate de la / către user space pot ajunge la device driver direct de la handler-ul de apel de sistem sau printr-un nivel intermediar la nivelul nucleului, nivel responsabil cu prelucrarea datelor din considerente de performanță sau pentru a reduce complexitatea driver-ului
acest nivel intermediar se numește, generic, I/O manager
este întâlnit des la discuri (spații de stocare) și la plăcile de rețea
la plăcile de rețea de la un apel de sistem până la driver, se trece prin stiva de networking; stiva se ocupă cu implementarea protocoalelor IP, TCP, în vreme ce driverul comunică cu placa de rețea care implementează protocolul Ethernet; vom mai detalia la cursul 11: Networking în sistemul de operare
la discuri, există un nivel intermediar (disk layer, block layer) și un nivel de implementare a sistemului de fișier deasupra sa, independente de drivere
implementarea sistemului de fișiere este generică și se ocupă de maparea unui fișier (văzut ca secvență de octeți) pe blocuri pe disc; driverul se ocupă de transferul efectiv către controller
block I/O layer se ocupă de operații de ordonarea și unificarea cererilor (sorting and merging) pentru a face cât mai puține cereri cu dispozitivul de stocare; ordonarea este utilă la discuri mecanice (hard disks) dar nu la discuri flash (solid-state drive)
block I/O layer gestionează buffer cache-ul: cache de memorie ale datelor de pe disc pentru a scrie / citi mai repede; astfel operațiile de scriere din user space vor ajunge în buffer cache și ulterior vor fi sincronizate pe disc; la fel, citirea de date se face tipic din buffer cache, doar când datele nu sunt prezente se aduc de pe disc
+ demo cu disk cache
tipic la citiri de pe disc se face read-ahead pentru a aduce date mai multe pentru accese viitoare
din acest motiv spunem că operațiile de tip read/write nu sunt, în general, blocante

Interfața user space

peste drivere și nivelurile intermerdiare nucleul expune interfața de apel de sistem aplicațiilor (system (call) API)
un proces abstractizează procesorul (thread-uri), memoria (spațiu virtual de adrese) și I/O (tabela de descriptori de fișiere)
operațiile expuse de kernel se realizează uzual peste un descriptor de fișier: fișier, director, char/block device, socket UNIX, socket de rețea, pipe, FIFO
interfața de file descriptori a fost extinsă în Linux la timere, semnale (timerfd, signalfd): universal I/O
în Windows, majoritatea componentelor sunt gestionate de un HANDLE, incluzând și evenimente, procese, thread-uri
operațiile uzuale pentru I/O expuse în system (call) API sunt de forma: open, close, read, write, seek (pentru block device-uri), ioctl
operația seek plasează un cursor/pointer; este specifică dispozitivelor cu acces aleator (block devices) nu la cele cu acces secvențial (char device-uri)
pentru a obține un descriptor de fișier, dispozitivele trebuie deschise (folosind open); se folosește o cale (o intrare în sistemul de fișiere)
acea cale poate să nu fie un fișier: poate fi un socket UNIX, un FIFO, un block device sau un char device
block/char device-urile se găsesc uzual în /dev/; sunt identificate de un major și un minor (acesta este modul în care un device driver este înregistrat, calea este doar ceva pentru înțelesul utilizatorului; putem folosi orice nume cât timp avem majorul și minorul corespunzător: https://www.quora.com/What-is-a-major-and-minor-number/answer/Jyoti-Singh-277?ch=99&share=56a118ea&srid=oLqI)
+ vizualizare intrări în /dev
+ demo cu ioctl pe CD-ROM

Creșterea performanței lucrului cu I/O

dispozitivele de I/O sunt lente; operațiile de I/O sunt adesea blocante și împiedică performanțe ridicate
de aceea sunt folosite soluții pentru a eficientiza lucrul cu I/O
operațiile read / write sunt numite operații sincrone și blocante
* sincrone: primesc atunci rezultatul operației
* blocante: dacă nu sunt date disponibile (pentru citire) sau nu este loc să se scrie (pentru scriere), thread-ul curent se blochează, se trece în starea WAITING, se schimbă contextul de execuție
alternativ, se pot folosi operații non-blocante (sincrone): se primește cât era disponibl sau se scrie cât era disponibil și se întoarce; dacă nu era disponibil se întoarce cu o semnalizare și se încercă mai târziu
o altă variantă este folosirea de operații asincrone (implicit non-blocante): se livrează o cerere către sistemul de operare și apoi operația se întoarce; ulterior se primește o notificare de la sistemul de operare sau se verifică încheierea operației
operațiile asincrone au avantajul eficienței: nu se blochează thread-ul, se execută altceva în paralel cu operația executată de kernel
au dezavantajul unui mod de programare complicat: este nevoie de un automat de stări care să fie actualizat pe măsură ce operația evoluează
modelul de programare asincronă este alternativă la modelul de thread-uri pentru prelucrarea de cereri multiple; de fapt, în biblioteca standard de Linux, API-ul POSIX de operații asincrone e implementat cu thread-uri; programarea cu thread-uri e mai simplă, dar are un overhead de schimbare de context și probleme de sincronizare; programarea cu operații asincrone este mai complexă dar are un overhead mai redus
în aplicațiile cu multe operații I/O este nevoie de un API scalabil pentru multe conexiuni/cereri care să multiplexeze între file descriptori / handle-uri: epoll pe Linux, CompletionPorts pe Windows, kqueue pe *BSD
o altă abordare este reducerea numărului de apeluri de sistem efectuate pentru transfer, realizată în două moduri:
* scatter / gather I/O: se colectează un vector de buffere și se face un singur apel pentru scrierea (scatter) sau citirea (gather) acestora
* zero-copy: se evită transferul de date din kernel în user space și invers preferându-se transferul direct în kernel space de la un dispozitiv la altul fără intermedierea user space (dar sub comanda user space); se face acest lucru prin memory mapped files sau funcții precum sendfile() pe Linux/Unix sau TransmitFile() pe Windows

Sumar

dispozitivele de I/O permit comunicarea proceselor cu exteriorul
există mai multe niveluri de prelucrarea I/O, de la hardware către aplicații
la nivel hardware, comunicarea dintre procesor și dispozitiv este intermediată de un controller
controller-ul conține registre care sunt citite/scrise de procesor

adresarea registrelor controller-ului se face prin memory-mapped I/O sau port-mapped I/O
pentru a ști când să scrie sau să citească date la sau de la controller, procesorul primește întreruperi sau folosește polling
întreruperile vin ca un semnal electric pe o linie de întrerupere ducând la rularea unei rutine (ISR)
rutina este o componentă de cod de obicei parte dintr-un device driver, parte din kernel space care gestionează hardware-ul
driver-ul primește date de la hardware prin controller și din user space prin apel de sistem
nucleul sistemului de operare are niveluri software intermediare care implementează componente generice sau eficientizează accesul: stiva de rețea, filesystem, block I/O layer
block I/O layer este responsabilă de buffer cache, caching în memorie a datelor de pe disc pentru operații cât mai rapide
la nivelul user space, se expune o interfață de tip descriptor de fișier și operațiile open, read, write, close, seek, ioctl
operațiile clasice sunt sincrone și blocante; pentru peformantă ridicată se folosesc operații neblocante sau asincrone
tot pentru performanță ridicată se folosesc soluții de tipul scatter-gather și zero-copy

Comunicatia procesor - controller se poate face:

- Port-mapped - dispozitivele de I/O au un spatiu separat de adrese (numit porturi) si necesita instructiuni speciale de procesor pentru a citi / scrie (instructiunile IN/OUT la x86).
- Memory-mapped - dispozitivele de I/O sunt mapate in memoria fizica, iar comunicatia se face prin scrierea/citirea la adrese prestabilite; orice instructiune de procesor poate fi folosita.

Dispozitivele de I/O functioneaza la viteza diferita de procesor; dupa ce o operatie de I/O este initiata, procesorul trebuie sa astepte completarea acesteia (e.g. citire de pe disk, e.g. un pachet nou vine pe placa de retea). Pentru a implementa aceasta sincronizare intre procesor si dispozitiv, se poate folosi:

- Polling: procesorul sta in bucla si verifica daca exista date noi. Avantaj: latenta scazuta, dezavantaj: cicli irositi atunci cand dispozitivul este lent sau datele vin cu frecventa mica.
- Intreruperi: procesorul isi continua alte taskuri (daca are, daca nu doarme) si este intrerupt atunci cand datele sunt disponibile. Avantaj: salveaza cicli; Dezavantaj: cand vin foarte multe intreruperi (e.g. una per pachet, un pachet la fiecare 60ns pe NIC de 10Gbps) overhead-ul intreruperilor este foarte mare - se trece in polling.

Pentru transferul datelor de la dispozitiv - memorie (sau invers), se poate folosi procesorul (costisitor) sau se poate folosi hardware de DMA (care permite dispozitivelor sa scrie in memorie direct).

Interactiune hardware - OS

O placa de retea (Network Interface Card, sau NIC pe scurt) expune mai multe perechi de ring-buffers (o pereche tx/rx) prin care SO trimite si primeste pachete. Comunicarea SO cu NIC se face folosind memory-mapped I/O, intreruperi sau polling (la incarcare mare) si DMA.

La initializarea placii, SO alloca o zona de memorie pentru pachete, si una pentru ring-uri, si le transmite placii scriind adresele memoriei nou alocate in zona de memorie monitorizata de placa.

RX: placa primeste un pachet si il salveaza intr-un buffer mic intern. Vede daca exista sloturi libere in RX; daca nu arunca pachetul. Ia primul slot liber din ringul de RX, si in el afla adresa zonei de memorie unde poate scrie pachetul. Folosind DMA, copiaza pachetul in memorie; pe unele arhitecturi pachetul este copiat direct in cache-ul L3 al procesorului (se numeste DDIO). Cand pachetul a fost copiat, avanseaza pointerul din ringul RX pentru a notifica SO ca exista un pachet disponibil. Daca intreruperile sunt activate, genereaza intrerupere.

TX: NIC-ul verifica sloturile din ringul de TX pe care le poate transmite. Daca exista un astfel de slot, se initiaza DMA din memorie catre bufferul intern; atunci cand pachetul este copiat, se incrementeaza pointerul pentru a anunta SO ca poate refolosi bufferul; se genereaza intrerupere daca ele sunt configurate.

Odata copiat in bufferul intern, pachetul este transmis atunci cand mediul devine liber; inainte de transmisie pachetul poate fi modificat, in functie de capabilitatile placii de retea:

- Se calculeaza checksum si se adauga in packet.

Curs 11: Networking in SO

Sumar curs anterior

Pentru a controla dispozitivele de Intrare/Iesire procesorul comanda controller-ul dispozitivului via magistrala (PCI, PCIe, etc).

- Se poate sparge pachetul in mai multe pachete mai mici, cand TSO este folosit (vezi discutie despre imbunatatire performanta mai jos).
- Se calculeaza FCS, se adauga la trailerul pachetului, si acesta este gata de a fi pus pe fir.

Multiplexare

- O placa de retea moderna poate expunea zeci de perechi de cozi RX/TX catre SO; fiecare din acestea se comporta ca o placa de retea separata pentru a comunica cu SO, e.g. genereaza propriile intreruperi.
- La transmisie, placa va lua pachete din cozile de TX folosind round-robin.
- La receptie, in mod normal placa va aplica o functie de hash la pachet pentru a decide in ce coada RX trebuie pus pachetul. Exista insa posibilitatea de a demultiplexa bazat pe adresa destinatie, i.e. fiecare coada RX sa primeasca pachetele pentru o anumita adresa IP.

Cozile multiple sunt in general folosite pentru a balansa pachetele catre toate core-urile din sistem - intreruperile diferitelor cozi pot fi directionate catre core-uri diferite. Cozile multiple pot fi folosite si pentru a simula mai multe placi de retea, fiecare cu adresa proprie de IP (vezi utiltarul ethtool).

Procesare SO - bottom half

RX: Atunci cand SO primeste pachete noi (anuntat via intrerupere), se va rula un cod care analizeaza fiecare pachet si hotaraste procesarea urmatoare. Se analizeaza adresa IP destinatie din pachet: daca aceasta este adresa uneia din adresele interfetelor active ale SO, packetul va fi procesat de partea “host” a stivei. Altfel, pachetul va fi procesat si eventual trimis mai departe (daca exista o intrare in tabela de rutare care se potriveste si SO este configurat astfel) sau va fi aruncat (drop).

In partea host, SO trebuie sa gaseasca socketul care va procesa pachetul si sa il livreze acestui socket. Selectia socketului se face urmand pasii urmatori:

- Se verifica daca exista o intrare in tabela de conexiuni deschise (folosind pe post de cheie un hash al 5-tuple din pachet - ip sursa, port sursa, ip destinatie, port destinatie, protocol). Daca da, se foloseste socketul destinatie din tabela de conexiuni deschise, se livreaza pachetul acestui socket si procesarea bottom half se in
- cheie.
- In caz contrar, se verifica in tabela de porturi asignate (apelul bind) - daca se gaseste un socket, pachetul va fi livrat stivei pentru acest socket.
- Altfel, pachetul este aruncat si eventual un pachet de raspuns generat si trimis catre sursa (ICMP).

Procesare stiva UDP

Fiecare socket UDP (creat ca urmare a apelului socket(,,PF_UDP) are o coada de pachete primite care nu au fost inca livrate. Atunci cand aplicatia executa recvfrom, va primi primul pachet din aceasta lista; daca nu exista nici un pachet, procesul va fi blocat pana cand un pachet vine; atunci procesul va fi deblocat si apelul recvfrom va intoarce noul pachet.

Procesare stiva TCP

Exista doua tipuri de sockets TCP: de ascultat (“listening”) si per conexiune. Dupa secventa socket/bind/listen avem un socket de ascultare. Apelurile accept si connect intorc un socket conectat. send / recv se pot executa doar pe un socket de conexiune, atunci cand conexiunea este in starea “CONNECTED”.

Atunci cand un segment TCP este receptionat, daca el are flag-ul SYN, se va folosi portul destinatie pentru a gasi singurul socket de ascultare; daca acesta exista, se va trimite SYN/ACK si va fi creata structura de date cu starea “half-open” care va fi asociata socketului de ascultare.

Atunci cand este primit ACK-ul pentru SYN/ACK, el va fi procesat de socketul de ascultare, conexiunea half-open este acum conectata, si este adaugata la coada de conexiuni disponibile corespunzatoare socketului de ascultare. Se adauga o intrare de asemenea in lista de conexiuni deschise (5-tuple) pentru noua conexiune. Aceasta intrare va fi stearsa atunci cand conexiunea este inchisa.

Atunci cand se executa accept, se ia prima conexiune disponibilia asociata socketului de ascultare, se creeaza un socket pentru ea, si se intoarce descriptorul asociat aplicatiei; daca nu exista conexiune, apelul se blocheaza.

Cand se primeste un pachet pentru o conexiune stabilita, se indentifica socketul si se vor salva datele intr-o zona de memorie per socket numita: ”receive buffer”. Apelurile recv vor lua date in ordine din aceasta structura; numarul de octeti primiti depinde de cantitatea datelor disponibile in receive buffer.

Daca exista “gauri” in receive buffer (de exemplu atunci cand un pachet se pierde iar pachetele urmatoare sunt receptionate), datele nu sunt livrate catre aplicatie decat atunci cand “gaura” este acoperita de retransmisia pachetului.

Receptie/Transmisie date din aplicatie

La TCP, pentru o conexiune stabilita, apelul send copiaza datele din spatiul utilizator intr-un buffer din nucleu, per conexiune, care se numeste **send buffer**. Apelul intoarce numarul de octeti copiati cu succes - acestia vor fi transmisi catre receptor atat timp cat reteaua functioneaza; insa nu este garantat ca vor fi primiti, de exemplu daca reteaua sau procesul la receptor pica.

Spre deosebire de UDP, unde un apel sendto creeaza un pachet IP, si un apel recvfrom intoarce continutul unui pachet IP, la TCP un apel send doar copiaza datele in buffer iar recv copiaza datele din buffer in userspace. La TCP pachetizarea este facuta automat de stiva, care creeaza un balans intre transmisia rapida (e.g. dupa primul octet primit de la utilizator) si overhead (transmiterea unui packet per fiecare octet primit ar irosi 64B per 1B de payload). Algoritmul lui Nagle este folosit in acest scop.

Transmisie date

Atunci cand un segment este gata de transmisie (exista date suficiente), se verifica daca fereastra de congestie (cwnd) si fereastra de receptie (receive window) permit transmiterea segmentului. Daca da, pachetul va fi creat, headerele adaugate si pus in coada ip_output a interfetei de iesire. De aici va fi copiat in ring-ul TX al placii de retea atunci cand exista slot-uri disponibile.

Considerente de performanta

Pentru UDP, se executa un apel de sistem pentru fiecare pachet; costul de tranzitie in/din kernel space este destul de mare, astfel incat viteza maxima este de 1Mbpps.

Pentru TCP, se poate amortiza costul apelului de sistem trimitand si primind mai mult de 1500 octeti / syscall (e.g. 10000 sau 100000).

Demo TCP versus UDP.

Totusi pachetizarea se face la dimensiunea pachetului suportat de interfata (1500 in mod uzual) - iar munca stivei este proportionala cu numarul de pachete. De aceea a aparut conceptul de segmentation offload, prin care stiva pachetizeaza la pachete mari (64KB) care sunt sparte fie de placa de retea (hardware offload), fie de partea de jos a stivei (generic segmentation offload) inainte de transmisie. Aceasta creste performanta semnificativ.

Demo cu TSO/GSO/LRO.

Discutia pana acum a avut in vedere o singura conexiune. Atunci cand un server trebuie sa trateze un numar mare de conexiuni simultan, exista trei variante de implementare:

1. 1 proces per conexiune
2. 1 thread pe conexiune
3. 1 proces/thread pentru multiple conexiuni, eventual cu mai multe threaduri pentru a folosi toate core-urile.

Exista demo la curs pentru toate variantele de mai sus. Varianta 1 are overhead mare de creare proces per conexiune (~1ms); numarul total de conexiuni este limitat de nr. maxim de procese din sistem. La varianta 2, overhead-ul de creare este mai mic (fara copierea tabelei de pagini, fara page faults, fara TLB flush la context switch), insa schimbarea de context este de asemenea costisitoare si creeaza latenta.

Varianta 3 foloseste epoll pentru a detecta cand se poate citi dintr-un socket fara blocare, insa este necesara mentinerea unei structuri de stare per client, ceea ce face programarea mai dificila.

Curs 12: Implementarea sistemelor de fișiere

Sumar curs anterior

placa de rețea este folosită pentru transferul datelor de la mai multe aplicații (de la mai mulți socketi)
placa de rețea și fiecare socket au buffere de transmitere (TX) și recepție (RX)
primirea pachetelor generează o întrerupere, procesorul apoi preia pachetele și le pune în memorie, apoi sunt transmise socket-ului
fiecare socket TCP are asociat un 5-tuplu: adresă sursă, port sursă, adresă destinație, port destinație, protocol; pe baza pachetului sursă al pachetului primit se demultiplexează socketul care trebuie să primească pachetul
în cazul UDP un pachet transferat din user space duce la transferul unui pachet (datagrame)
în cazul TCP se copiază într-un buffer al nucleului de unde este transferat la momentul potrivit
un apel de transmitere (TX, send) se blochează dacă bufferul nucleului este plin
un apel de primire (RX, receive) se blochează dacă bufferul nucleului este gol

pachetele sunt transmise la dimensiunea frame-ului plăcii de rețea (MTU: Maximum Transmission Unit), 1500 octeți
pentru a mări viteza există suport pentru segmentation offload în care un segnement mai mare este compartimentat în frame-uri
pentru prelucrarea datelor pentru mai multe conexiuni avem varianta asincronă sau varianta multithread/multiproces, varianta asincronă este eficientă dar e dificil de programat, este nevoie de menținerea unei stări a fiecărei conexiuni

Stocarea datelor și sistemul de fișiere

aplicațiile și sistemul de operare folosesc memoria pentru reținerea codului și datelor, procesorul pentru prelucrare și I/O pentru comunicare cu exteriorul
pornirea unei aplicații sau a sistemului de operare necesită un suport persistent (pentru reținerea executabilelor, bibliotecilor, imagini de kernel)
aplicațiile folosesc date (configurare, multimedia, baze de date) care necesită un suport persistent (non-volatile)
dispozitivele de stocare (storage) sunt dispozitive I/O care asigură suport persistent, în general discuri (CD/DVD/Blu-ray, HDD, SSD, USB flash drive), NVRAM; cartelele perforate (punch cards) sau bandă magnetică erau forme de stocare
datele sunt organizate în fișiere pe dispozitivele de stocare pentru a fi accesate de aplicații
fișierele sunt forme de compartimentare a datelor; uzual fișierele sunt structurate ierarhic
modul de organizare a datelor pe un suport persistent reprezintă sistemul de fișiere
sistemul de fișiere expune o interfață aplicațiilor
interfața este în linii mari generică, aceleași tipuri de operații sunt expuse de toate sistemele de fișiere, indiferent de implementarea acestora
este responsabilitatea sistemului de operare să ofere o interfață comună care să ascundă complexitatea și diversitatea sistemelor de fișiere
+ diagramă cu discuri, sisteme de fișiere, sistem de operare, interfață comună
un sistem de fișiere urmărește să asigure o structură ierarhică (pentru organizarea informațiilor în directoare, subdirectoare, fișiere), performanță (să fie rapid) și scalabilitate (să putem crea multe fișiere, fișiere mari)

Reminder: Interfața sistemului de fișiere

interfața sistemului de fișiere definește operațiile și structurile pentru lucrul cu fișiere
interfața cuprinde descriptori de fișiere și operațiile: deschidere, citire, scriere, căutare, închidere, creare, ștergere, mapare
fișierele sunt identificate de nume, un fișier deschis e identificat de un descriptor de fișier
deschiderea unui fișier oferă un descriptor de fișier procesului apelat
un descriptor de fișier este un index în tabela de descriptori de fișiere a unui proces care referă o structură de fișier: permisiuni de deschidere, cursor de fișier, pointer la structura de fișier de pe disc
structura de fișier de disc (numită și FCB, File Control Block, inode pe Unix) este copiată în memorie de pe disc
+ diagramă cu FCB pe disc, FCB în memorie, structură de fișier deschis, tabelă de descriptori de fișiere, descriptor de fișier
structura de fișier deschis dispare din memorie când dispar toate referințele către aceasta (când se închide un fișier)
FCB dispare din memorie când nu mai sunt fișiere deschise care o referă
FCB dispare de pe disc la operații de ștergere (rm, del)

File Control Block (FCB)

FCB este metadata unui fișier: informații despre fișier
este o structură persistentă care rezidă pe suportul persistent; este copiată în memorie atunci când este deschis un fișier și când se creează o structură de fișier deschis care o referă
se numește inode sau i-node (index node) pe sistemele de fișiere specifice sistemelor de operare Unix (ext4, HFS, AppleFS)
FCB conține:
* identificator (inode number, ino)
* controlul accesului (ownership, permissions)
* timpi de acces (timestamps)
* tip de fișiere
* dimensiunea datelor
* pointeri către date
+ folosire comandă stat pentru a afișa metadatele unui fișier
pointerii sunt către blocuri de date
există pointeri către blocuri de date care conțin pointeri către blocuri de date (indirectare)
există pointeri către blocuri de date care conțin pointeri către blocuri de date care conțin pointeri către blocuri de date (indirectare dublă)
există pointeri către ... (indirectare triplă)

Numele unui fișier. Dentry-uri. (Hard) link-uri

în general, un FCB nu conține numele unui fișier
numele unui fișier este reținut într-o structură separată numită dentry (directory entry)
un dentry conține referință către inode (inode number) și numele fișierului
un dentry se mai numește un link (sau un hard link)
pot exista mai multe dentry-uri care referă același inode, adică mai multe hard link-uri, util pentru a plasa un fișier în mai multe locuri
un FCB conține și numărul de link-uri
+ demo cu folosirea ln pentru crearea de link-uri hard și ls -i pentru afișarea de informații
un fișier este șters când nu mai are link-uri
+ demo cu recuperare de fișier șters din /proc
comanda rm, apelul remove() nu șterg un fișier ci șterg un link (un dentry)
apelul de sistem aferent este unlink()/unlinkat()
+ demo cu strace -e unlinkat rm a.txt

Directoare

dentry-urile sunt reținute în blocurile de date ale directoarelor
un director este un fișier ale cărui blocuri de date conțin dentry-uri; pot fi dentry-uri de fișiere sau de subdirectoare
tipul unui fișier poate fi: obișnuit (regular) sau director (directory) sau alte tipuri
comanda ln creează un dentry nou într-un bloc de date al unui director, dentry ce referă un inode/FCB existent
comanda rm/unlink șterge/invalidează un dentry într-un bloc de date al unui director
dacă un FCB nu mai are un dentry care îl referă este șters împreună cu blocurile de date ale FCB-ului
există un director rădăcină, acesta este marcat corespunzător în sistemul de fișiere
ierarhia sistemului de fișiere este construită din parcurgerea dentry-urilor din directorul rădăcină, dentry-uri referă alte FCB-uri de tip director, care conțin dentry-uri către FCB-uri de tip fișier și director și așa mai departe
un director trebuie să permită și urcatul în ierarhia, cu ajutorul comenzii cd ..
un director conține un dentry cu numele .., dentry ce referă FCB-ul directorul părinte
de asemenea, un director conține un dentry cu numele ., auto-referențiere, de unde ./a.out sau cp path/to/file .
de aceea, un director fără subdirectoare are două link-uri: dentry-ul din directorul părinte cu numele directorului și . (auto-referința)
pe cazul general, un director cu N subdirectoare are N+2 link-uri: dentry-ul din directorul părinte cu numele directorului și . (auto-referința) și intrările .. din fiecare subdirector
+ demo cu directoare și număr de link-uri de directoare
dimensiunea unui director (numărul de blocuri ocupate) este proporțional cu numărul de intrări din director (număr de dentry-uri): cu cât mai multe fișiere/directoare conține, cu atât dimensiunea unui director (numărul de blocuri ocupate) este mai mare
+ demo cu dimensiunea unui director

Alte tipuri de intrări

fișierele (indicate de FCB) pot fi fișiere obișnuite (regular files) sau directoare sau alte tipuri de fișiere
symbolic link-urile (symlink) sunt FCB-uri al căror conținut este o cale, un șir; șirul este interpretat și rezultă un dentry și FCB-ul corespunzător
un symlink poate fi "dangling" dacă șirul nu este o cale validă (nu se rezolvă la dentry și FCB)
comanda readlink rezolvă șirul de tip cale a unui FCB de tipul symlink
un symlink este un inode, un hard link este un dentry
putem crea symlink-uri la intrări din alte partiții (sisteme de fișiere montate), dar nu hard link-uri; două sisteme de fișiere diferite au fișiere diferite cu același inode, nu ar funcționa un hard link între sisteme de fișiere, nu ar face diferența
+ demo cu stat pe un symbolic link
alte intrări în sistemul de fișiere sunt socketi UNIX, named pipes (FIFO), char device, block device
aceste intrări au FCB dar nu au date; sunt doar interfețe de comunicare inter-proces (socketi UNIX, named pipes) sau de interacțiune cu resurse (hardware) expuse de sistemul de operare (char device, block device)

Gestiunea spațiului

când creăm un hard link (dentry) se ocupă un slot dintr-un bloc de date al unui director
ocuparea înseamnă validarea/activarea acelei intrări; putem spune că se alocă un bloc nou atunci când nu mai sunt slot-uri în blocurile curente
ștergerea unui link înseamnă invalidarea/dezactivarea acelei intrări (uzual se pune -1 sau 0 în câmpul ino al inode-ului)
în mod similar există o zonă dedicată pentru inode-uri; în acea zonă se validează/activează sau invalidează/dezactivează inode-uri în momentul creării sau ștergerii unui inode
un bloc de date este valid dacă este referit de un inode (și unul singur)
atunci când se scrie dincolo de limita curentă a fișierului se alocă un bloc de date nou, adică se referă acel bloc în cadrul inode-ului; când se trunchiază un fișier se "dezalocă" acel block, dispare referința din inode

Dispunerea informațiilor pe spațiul de stocare

un sistem de fișiere ocupă spațiu pe un disc în urma formatării: secțiuni din partiție conțin informațiile de stocare
sistemul de fișiere cuprinde un superbloc: acesta conține informații despre celelalte secțiuni (metadate despre metadate)
în mod uzual există o zonă care reține informații despre inode-urile valide/activate/ocupate (inode map) și o zonă care reține informații despre blocurile valide/activate/ocupate (data map); aceste zone sunt uzual bitmap-uri: bit 0 înseamnă inode sau bloc liber, bit 1 înseamnă inode sau bloc ocupat
o zonă dedicată reține inode-urile și altă zonă reține blocurile
inode-urile au pointeri (numere de blocuri) către blocurile corespunzătoare
inode-urile/blocurile valide sunt marcate astfel în bitmap-ul corespunzător
+ diagramă cu dispunerea informațiilor pe spațiul de stocare
când se creează un fișier, se găsește un loc liber (bitul 0 în inode map) se marchează în inode bitmap bitul 1 și se completează inode-ul corespunzător în zona de inode-uri
când se eliberează un fișier, se marchează bitul 0 în inode map
de obicei la eliberare nu se completează cu zero inode-ul eliberat sau blocurile eliberate; ceea ce înseamnă că se pot recupera date la nevoie sau se pot exfiltra date mai vechi de un atacator care are acces la disc

Operații cu sistemul de fișiere

un sistem de fișiere se alfă pe o partiție, în urma operației de formatare
formatarea creează superblocul și celelalte zone, creează inode-ul rădăcină
formatarea "raw" se asigură de zeroizarea partiției cu zero
un sistem de fișiere este montat pentru a fi folosit; montarea înseamnă legarea inode-ului rădăcină la o cale din sistemul de fișiere
sistemul de fișiere rădăcină este montat în /, alte sisteme de fișiere sunt montate în căi din sistemul de fișiere rădăcină
demontarea înseamnă dezactivarea căii unde a fost montat, sistemul de fișiere devine inaccesibil
în cazul unei întreruperi bruște este posibil să existe date inconsecvente: marcaje în data map, dar pointerii încă prezenți în blocul de date al fișierului, inode-uri marcate ca șterse dar intrări în valide de tip dentry; o astfel de soluție este rezolvată prin filesystem check (fsck)

pentru acces mai rapid la informații, este util ca blocurile de date să fie într-o zonă apropiată; operația se cheamă defragmentare

Sumar

aplicațiile au nevoie de persistență pentru executabile și pentru datele folosite
spațiul de stocare persistent (discuri, NVRAM) este formatat cu un sistem de fișiere
sistemul de operare asigură interfață generică peste mai multe sisteme de fișiere, organizare ierarhică cu fișiere și directoare
interfața de sistem de fișiere cuprinde operații și descriptori de fișiere
un descriptor de fișier este un index în tabela de descriptori de fișiere a procesului, intrare care referă o structură de fișier deschis, care referă FCB în memorie
FCB (file control block) este metadata unui fișier și conține pointeri către datele fișierului
numele fișierului nu se găsește în FCB ci într-o structură numită dentry; pot exista mai multe dentry-uri (numite hard link-uri) la un fișier
un dentry conține numele fișierului și indexul inode-ului
un director este un fișier ale cărui blocuri de date conțin dentry-uri, adică intrările în acel director
un director are o referință .. către directorul părinte, și o referință . către sine
un director are N+2 hard link-uri/nume: numele său (în dentry-ul părintelui), referința către sine și referințele .. ale celor N subdirectoare
alte tipuri de intrări în sistemul de fișiere sunt symbolic link-urile, socketii UNIX, FIFO-uri, char devices, block devices
symbolic link-urile conțin o cale, fiecare symbolic link este un inode; fiecare hard link este un dentry
pe disc datele sunt ținute în blocuri de date într-o zonă de blocuri, inode-urile într-o zonă de inode-uri
o zonă numită inode map și alta numită data map reține ce blocuri și ce inode-uri sunt ocupate
superblocul este o zonă care definește unde se găsesc zonele unui sistem de fișiere formatat
pentru folosirea unui sistem de fișiere, se formatează o partiție și apoi se montează într-un punct de montare (un director existent)
sistemul de fișiere rădăcină e montat înaintea altor sisteme de fișiere în /