Sisteme de operare

Cursul 5 - Procese și thread-uri

Drăgulici Dumitru Daniel

Facultatea de matematică și informatică,
Universitatea București

2008

Cuprins

Procese - generalități

SO cu multitasking sunt construite în jurul conceptului de proces.

Proces = execuție (instanță) a unui program.

Fiecărui proces îi sunt asociate un set propriu de date (cu care este rulat programul) și un set de informații de control (cu ajutorul cărora este gestionat de SO). Mai multe procese pot rula un același program, dar ele vor opera pe seturi de date distincte și vor avea seturi de informații de control distincte.

Procese - generalități

SO multitasking pot rula mai multe procese în paralel; paralelismul poate fi:
- **paralelism real**: fiecare proces pe câte un procesor distinct;
- **paralelism intercalat** (**time-sharing**, **pseudoparalelism**): mai multe procese pe același procesor, executându-se intercalat porțiuni din fiecare.

Modelul conceptual de proces (proces secvențial) este acela de execuție secvențială a instrucțiunilor unui program (conform logicii programului) pe un procesor virtual, mapat continuu (în cazul paralelismului real) sau discontinuu (în cazul time-sharing) pe un procesor fizic.

Ca terminologie (cf. Wikipedia):
- rularea fiecărui proces pe câte un procesor distinct s.n. **simultaneitate**;
- rularea mai multor procese pe același procesor ce comuta rapid de la un proces la altul, facându-le să comute între a fi în execuție și a fi în așteptare, s.n. **concurență** sau **multiprogramare**.

Procese - generalități

Comportamentul unui proces poate fi descris prin **urma** sa (**trace**), care este secvența instrucțiunilor executate de el. Comportamentul mai multor procese concurente poate fi descris prin evidențierea modului în care urmele lor se intercalează.

Procese - generalități

În cazul time-sharing, executarea în paralel a instrucțiunilor în cadrul unor procese diferite nu aduce economie de timp, deoarece pe procesorul fizic ele se excută tot secvențial. Câștigul apare atunci când unele procese sunt blocate (la cerere sau automat, în așteptarea întrunirii anumitor condiții), deoarece procesorul poate fi alocat între timp celorlalte.

Totodată, deoarece procesorul comută mereu între procese, rata cu care un proces efectuează operațiile nu este uniformă, nici măcar reproductibilă, dacă același proces este executat din nou - de aceea, programele executate în time-sharing nu trebuie scrise cu presupuneri intrinseci referitoare la duratele de timp.

De exemplu, presupunem că un program trebuie sa pornească o bandă, să aștepte ca ea să ajungă la turația necesară, apoi să citească pe rând înregistrările; așteptarea se poate programa sub forma unui ciclu ce execută de 10000 ori instrucțiunea vidă; atunci însă, dacă programul este rulat ca proces în time-sharing cu alte procese, SO îl poate întrerupe suficient de des/mult în timpul ciclului a.î. să aștepte prea mult iar pe banda primele înregistrări să treacă deja de capul de citire.

Procese - generalități

SO trebuie să conțină mecanisme pentru:
- creare, distrugere, întrerupere, repornire a proceselor;
- alocarea de resurse pentru procese;
- planificarea la execuție a proceselor (scheduling);
- arbitrarea accesului concurent al proceselor la resurse și evitarea interblocajelor;
- comunicarea între procese.

Unele dintre aceste mecanisme sunt oferite sub forma unor apeluri sistem apelabile din procese existente.

Cuprins

Crearea proceselor

Crearea proceselor este acțiunea prin care SO adaugă un nou proces la cele curent gestionate, alocându-i un spațiu de adrese și creând structurile de date necesare gestionării lui.

Fapte uzuale care determină crearea unor procese:
- inițializarea sistemului;
- la cererea explicită a unui alt proces existent (**process spawning**);
- o cerere a utilizatorului, într-un mediu interactiv;
- inițializarea unei lucrări în fundal, la întâlnirea pe fluxul de intrare a unui nou lot, în sistemele de prelucrare pe loturi (**batch systems**).

Din punct de vedere tehnic, toate procesele noi sunt create prin executarea de catre un proces existent a unui apel sistem de creare de proces (ex. "fork()" în UNIX/Linux, "CreateProcess()" în Windows).

Când un proces produce un proces nou, primul s.n. **proces părinte** (tată) iar celalalt **proces copil** (fiu) - între ele sistemul crează anumite legături logice.

Terminarea proceselor

Fapte uzuale care determină terminarea unui proces:
- ieșire normală (voluntară): procesul execută un apel sistem de încheierea execuției (ex. "exit()" în UNIX/Linux, "ExitProcess()" în Windows);
- ieșire cu eroare (voluntară): programul detectează (prin propriile instrucțiuni) o anumită eroare (ex: date incorecte) și cere terminarea;
- eroare fatală (involuntară): procesul cauzează o eroare (ex: încearcă o operație ilegală) și este terminat de sistem; în unele sisteme (ex. UNIX) procesul poate informa sistemul că dorește să se ocupe el însuși de anumite erori, caz în care sistemul nu termină procesul ci doar îi semnalează apariția uneia dintre aceste erori;
- terminare de către un alt proces (involuntară) - în acest scop procesul ucigaș execută un apel sistem (ex. "kill()" în UNIX/Linux, "TerminateProcess()" în Windows).

Cuprins

Ierarhii de procese

Când un proces crează alt proces (ex. cu "fork()" în UNIX/Linux), sistemul crează anumite legături logice între părinte și copil.

Procesele, legate în acest fel, formează o **ierarhie** - în general o arborescență.

Ierarhii de procese

În UNIX/Linux procesele se organizează de asemenea în sesiuni și grupuri; o sesiune poate avea mai multe grupuri; fiecare sesiune/grup are un proces lider (de sesiune/grup).

În principiu, orice nou proces se află inițial în aceeași sesiune și grup cu părintele său, dar ulterior se poate muta într-un alt grup sau poate iniția o sesiune nouă sau un grup nou unde să fie lider (executând anumite apeluri sistem).

De aceea, împărțirea proceselor în sesiuni și grupuri nu are o legătură strictă cu ierarhia dată de relația părinte-copil.

Ierarhii de procese

Windows nu conține conceptul de ierarhie de procese (toate procesele sunt egale).
Singurul aspect asemănător ierarhiei este legat de momentul creării de către un proces a unui proces nou - atunci părintele primește un handler cu care ulterior își poate controla copilul; părintele poate însă să transmită handler-ul altui proces, invalidând astfel ierarhia.

În UNIX procesele nu pot să-și "dezmoștenească" copii.

Ierarhii de procese

În general, când un proces se termină, furnizează un cod de retur (**exit status**) către procesul părinte - acesta îl poate colecta folosind apeluri sistem specifice (ex. "wait()" sau "waitpid()" în UNIX/Linux).

În cazul programelor scrise în C, codul de retur furnizat la terminare se poate specifica prin valoarea returnată de "main()" sau ca parametru al funcției "exit()".

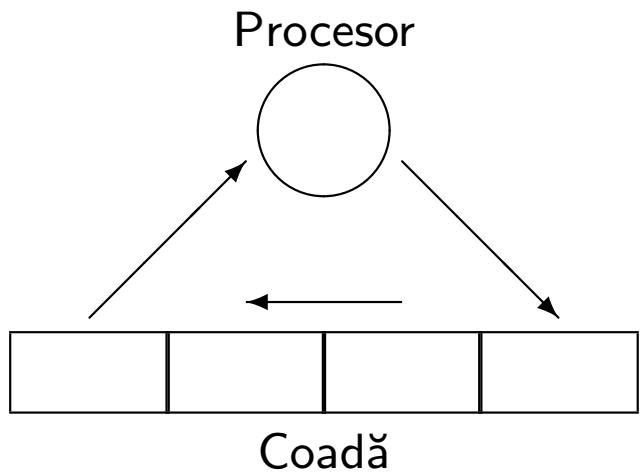
Cuprins

Stările unui proces

Pentru gestiunea proceselor, SO folosește o **tabelă de procese** (**process table**) având câte o intrare pentru fiecare proces, numită uneori **bloc de control** al procesului (**PCB**).

Stările unui proces

Pentru executarea concurentă a proceselor pe un procesor, acestea sunt inserate într-o structură de așteptare, organizată de exemplu ca o coadă cu priorități (conținând pointeri către procesele respective) iar o componentă a nucleului numită **planificator** (**scheduler**) ia periodic câte unul, îi alocă procesorul pentru a (mai) executa o parte din el, după un timp îl întrerupe, îl inserează înapoi în structura de așteptare, apoi ia alt proces, etc. - această activitate s.n. **planificare la execuție** (**scheduling**) și este una din activitățile nucleului. Ea se desfășoară pe baza unui **algoritm de planificare**.



Stările unui proces

Comutarea de la un proces la altul presupune salvarea/restaurarea **contextului procesului** - un set minimal de date (incluzând valorile regiștrilor utilizați, în special program counter-ul PC) ce definește starea în care s-a ajuns cu execuția lui și permite reluarea ei ulterioară. Această comutare s.n. **context switch**.

Mecanismul de planificare folosește ceasul intern al mașinii (hardware) - el produce la intervale regulate de timp o întrerupere hardware, iar acesteia i se poate atașa o rutină care întrerupe procesul curent și redă controlul nucleului pentru altă activitate de planificare.

Stările unui proces

În general, fiecare tip de întrerupere are asociată o locație de memorie (numită vector de întrerupere), unde trebuie plasată adresa unei rutine (handler-ul sau rutina de tratare a întreruperii). Când apare o întrerupere, sunt parcurși următorii pași:

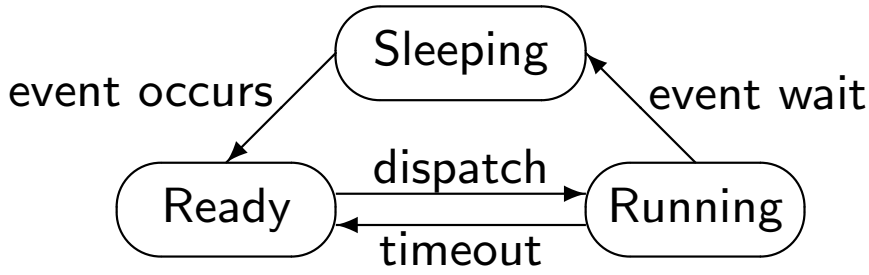
1. Hardware-ul memorează PC, alți regiștri, etc. pe stiva curentă.
2. Hardware-ul încarcă PC din vectorul de întrerupere, executându-se astfel salt la o rutină assembler de tratare a întreruperii.
3. Rutina assembler salvează regiștrii, etc. (de obicei în PCB procesului) și șterge informația încărcată pe stivă.
4. Rutina assembler inițializează o stivă temporară (setând registrul SP).
5. Se execută serviciul de tratare a întreruperii, o rutină scrisă de obicei în C.
6. Se apelează planificatorul, care decide ce proces se va executa în continuare.
7. Rutina C revine la rutina assembler.
8. Rutina assembler încarcă regiștrii și harta memoriei pentru procesul devenit acum curent și începe execuția acestuia.

Detaliile acestui mecansim pot varia de la un SO la altul.

Pentru a nu perturba mecanismul planificării, pe perioada executării rutinelor de mai sus se poate comanda dezactivarea întreruperilor.

Stările unui proces

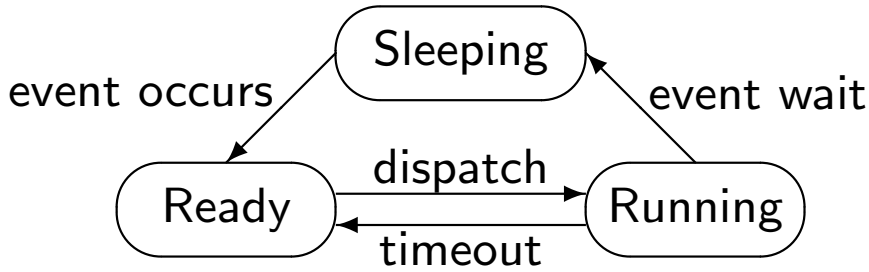
În timpul executării (intercalate), un proces (aflat în tabela de procese) se poate afla în mai multe stări:



- **în execuție** (**running**): folosește procesorul în acel moment; aici există două substări, procesul putând fi:
    * în **user mode** - când execută instrucțiuni scrise de utilizator (în programul executat);
    * în **kernel mode** - când execută un apel sistem sau întrerupere;
- **gata de execuție** (**ready**): nu folosește procesorul, dar poate fi pornit/continuat;
- **adormit** (**sleeping**): nu folosește procesorul și nu poate fi pornit/continuat pâna nu sunt întrunite anumite condiții externe;

Stările unui proces

În timpul executării (intercalate), un proces (aflat în tabela de procese) se poate afla în mai multe stări:



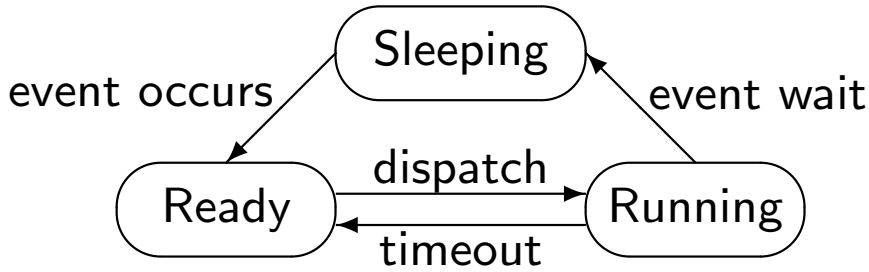
Tranzițiile între aceste stări se efectuează astfel:

user mode $\longrightarrow$ **kernel mode**: la apelarea unui apel sistem sau generarea unei întreruperi;

kernel mode $\longrightarrow$ **user mode**: la revenirea dintr-un apel sistem sau handler al unei întreruperi;

Stările unui proces

În timpul executării (intercalate), un proces (aflat în tabela de procese) se poate afla în mai multe stări:

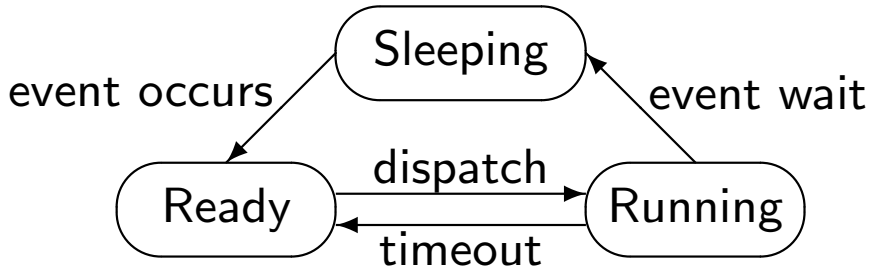


Tranzițiile între aceste stări se efectuează astfel:

running $\rightleftharpoons$ **ready**: prin decizii de planificare/pauzare (**dispatch/timeout**) ale planificatorului;

Stările unui proces

În timpul executării (intercalate), un proces (aflat în tabela de procese) se poate afla în mai multe stări:

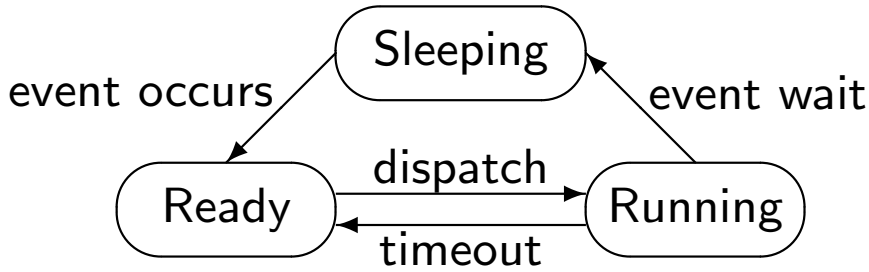


Tranzițiile între aceste stări se efectuează astfel:

running → **sleeping**: când procesul este în kernel mode și este penalizat, de exemplu pentru că încă nu sunt disponibile anumite resurse; exemple:
* procesul execută "scanf()", care apelează "read()" (apel sistem), și încă nu sunt disponibile date la intrare (utilizatorul încă n-a tastat nimic);
* procesul apelează "read()" pentru a citi dintr-un tub, dar tubul e vid și are scriitori (a se vedea cursul 6);
în ambele cazuri, procesul este trecut în sleeping;

Stările unui proces

În timpul executării (intercalate), un proces (aflat în tabela de procese) se poate afla în mai multe stări:

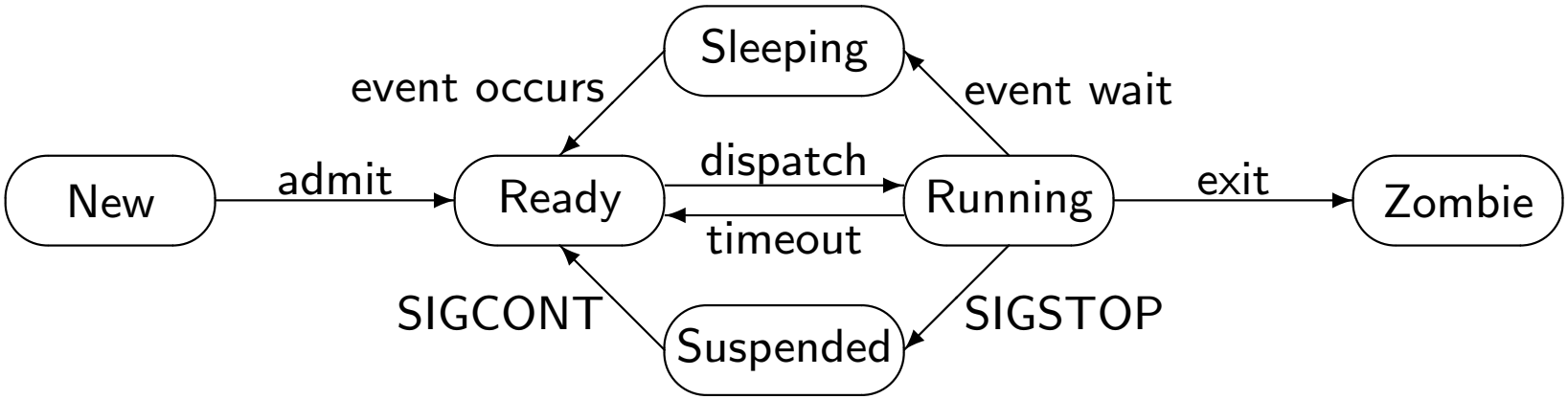


Tranzițiile între aceste stări se efectuează astfel:

sleeping $\rightarrow$ **ready**: când penalizarea încetează (de ex. resursa așteptată devine disponibilă - apar date disponibile în exemplul de mai sus); din acest moment, planificatorul îl poate alege oricând pentru running.

Stările unui proces

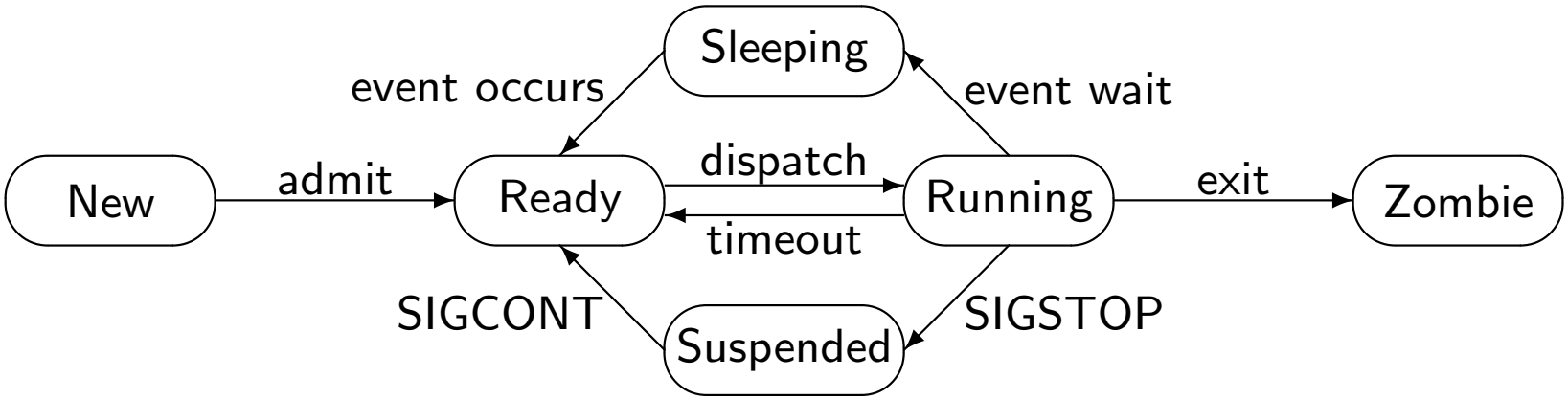
În timpul executării (intercalate), un proces (aflat în tabela de procese) se poate afla în mai multe stări:



Alte stări posibile (ultimile două sunt specifice UNIX/Linux):
- **new**: proces nou creat (sistemul a creat inițializările necesare), dar încă nu a fost admis pentru execuție (de exemplu încă nu există suficientă memorie);
- **suspendat** (**suspended**, **stopped**): nu folosește procesorul și nu poate fi pornit/continuat până nu se emite o comandă explicită în acest sens;
- **zombie**: nu folosește procesorul și este considerat terminat; PCB său încă este menținut în tabela de procese și conține informații minimale (codul de retur); în momentul când părintele său îi colectează codul de retur (cu "wait()" sau "waitpid()"), procesul zombie este eliminat (dispare complet).

Stările unui proces

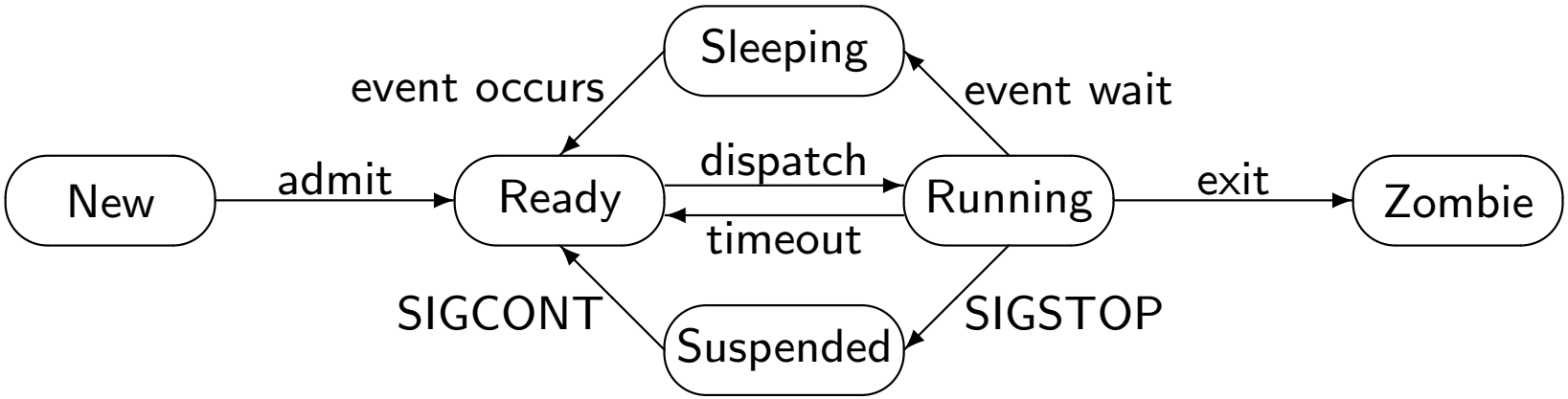
În timpul executării (intercalate), un proces (aflat în tabela de procese) se poate afla în mai multe stări:



Notăm că dacă părintele unui proces se termină (devine zombie sau dispare complet) înaintea lui, copilul devine automat fiul procesului "init";
"init" face periodic apeluri de tip "wait()" sau "waitpid()" și își elimina astfel copii proveniți între timp atunci când devin (sau dacă erau deja) zombie.

Stările unui proces

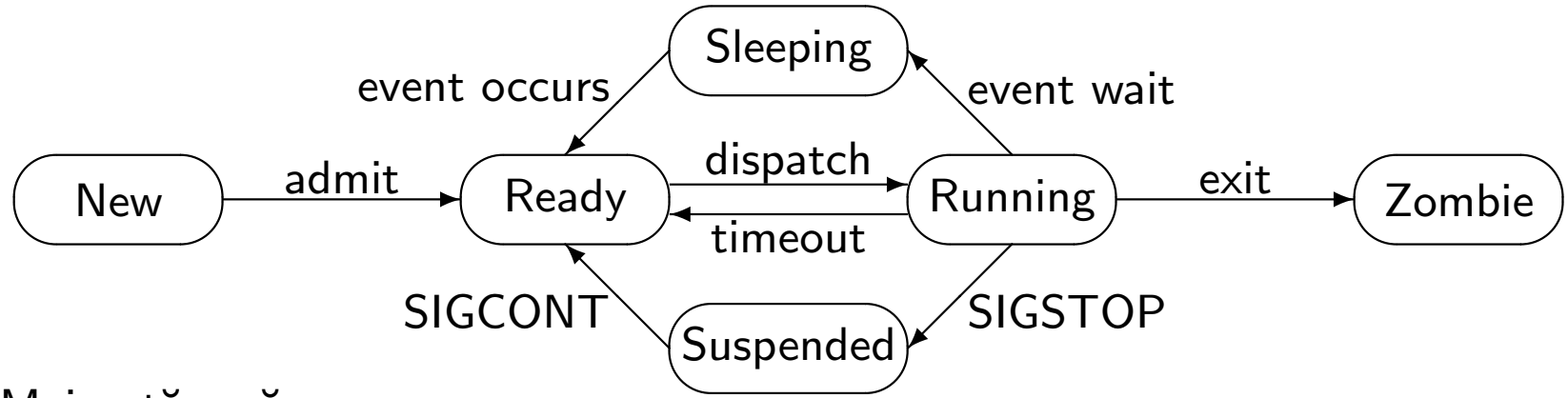
În timpul executării (intercalate), un proces (aflat în tabela de procese) se poate afla în mai multe stări:



Legat de noile stări, apar următoarele tranziții:
new → **ready**: când procesul este admis pentru execuție;
running → **suspended**: la primirea semnalului SIGSTOP;
suspended → **ready**: la primirea semnalului SIGCONT;
running → **zombie**: la terminare (normală sau anormală, de ex: return din "main()", apel "exit()", primirea unui semnal ucigaș precum SIGKILL).

Stările unui proces

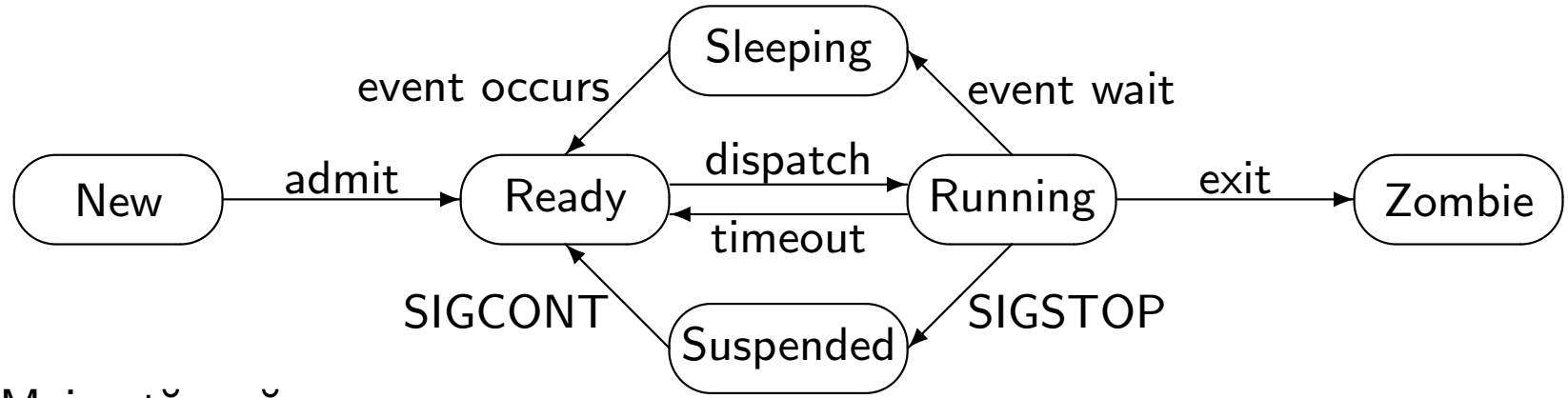
În timpul executării (intercalate), un proces (aflat în tabela de procese) se poate afla în mai multe stări:



Mai notăm că:
- în starea sleeping se intră/iese automat, când sunt întrunite anumite condiții externe (ex: solicitarea unor resurse indisponibile/anumite resurse devin disponibile);
- în starea suspended se intră/iese la comandă (prin trimiterea semnalelor SIGSTOP/SIGCONT); aceste tranziții sunt exploatate în cadrul mecanismului de job control al anumitor shell-uri, pentru oprirea/(re)pornirea la comandă a anumitor procese lansate de utilizator.

Stările unui proces

În timpul executării (intercalate), un proces (aflat în tabela de procese) se poate afla în mai multe stări:



Mai notăm că:
- un semnal este o entitate având ca unică informație relevantă un cod întreg și poate fi trimis de la un proces la altul (cu apelul sistem "kill()"); la primirea unui semnal procesul își întrerupe execuția normală și execută o funcție handler asociată tipului respectiv de semnal (poate fi un handler sistem sau unul utilizator), dupa care revine (handler-ul poate cere însă și terminarea procesului);

semnalele nu sunt tratate când procesul este în kernel mode - daca vin între timp, sunt puse în așteptare (pending) la proces și abia la trecerea din kernel mode în user mode sunt tratate (li se apelează handlerele);

există și alte aspecte care condiționează tratarea (sau chiar pierderea) semnalelor - a se vedea cursul 6.

Cuprins

Implementarea proceselor

Conceptul de proces este implementat cu ajutorul unor structuri de date și mecanisme specifice.

Crearea procesului (în general ca urmare a unui apel sistem efectuat de un alt proces, ex. "fork()" în UNIX/Linux) presupune următoarele operații (efectuate de sistem):
- asignarea unui identificator de proces unic;
- alocarea spațiului pentru imaginea procesului;
- inițializarea PCB al procesului;
- inserarea procesului în evidențele sistemului (de ex. în listele folosite de planificator).

Implementarea proceselor

Imaginea procesului (**process image**) conține date descriptive ale acestuia; în general, ea este formată din:
- programul executat;
- datele accesibile din program (proprii sau partajate cu alte procese);
- stiva/stivele folosite pentru apelarea rutinelor;
- contextul de execuție: informații despre proces folosite de SO pentru manipularea acestuia (incluzând informații de identificare, stare, control).

Pentru a fi în executie, procesul trebuie sa aibe imaginea încarcată în memoria principală. Cand este oprit din diverse cauze, imaginea poate fi swap-ata pe disc, în memoria virtuală.

Zona de memorie accesibilă prin executarea instrucțiunilor din programul utilizator s.n. **spațiul de adrese** al procesului. El include segmente de text (cod), date (proprii sau partajate cu alte procese), stiva - ca parte a imaginii procesului salvată pe disc, acestea au adrese virtuale.

O zona de date partajate între mai multe procese este unică fizic, dar prin mecanismul memoriei virtuale ea pare că este în spațiul de adrese al fiecărui proces care o folosește.

Implementarea proceselor

PCB (**Process Control Block**) este **intrarea din tabela de procese** (**process table entry**) corespunzătoare procesului și conține informații utilizate de SO pentru controlul acestuia; în general, aceste informații se referă la:
- gestiunea procesului:
identificatorul procesului, identificatorul procesului părinte, grupul procesului,
utilizatorul proprietar, grupul utilizatorului proprietar,
valorile regiștrilor (mai ales PC, SP și cuvântul de stare),
starea procesului (running, ready, etc.), prioritate, parametri de planificare,
momentul creării procesului, timpul procesor consumat,
momentul următoarei alarme, pointeri către alte PCB,
tabela de gestiune a semnalelor primite;
- gestiunea memoriei: indicatori spre segmentele de text, date, stivă;
- gestiunea fișierelor: tabela de descriptori, directorul curent.
Tabela de descriptori conține referințe către fișierele curent deschise de proces - acestea pot fi desemnate (de exemplu pentru a le aplica apeluri sistem) prin indicele lor în tabelă (descriptor).

Notăm că în diverse implementări unele din informațiile de mai sus sunt menținute în aria U a procesului (a se vedea mai jos), nu în PCB.

Implementarea proceselor

Ansamblul informațiilor pe care sistemul le deține despre un proces s.n. **contextul procesului**. El este format din:

- **contextul utilizator**: cod, datele procesului (proprii, partajate), stiva utilizator, din spațiul de adrese;
- **contextul registru**: valorile regiștrilor;
- **contextul sistem**, format din:
  - **partea statică**, conținând:
    - PCB;
    - **aria U** (**user area**): informații suplimentare despre proces (în completarea PCB), folosite doar de nucleu și doar când ruleaza în contextul acestui proces (când procesul este activ);
      de ex. aici pot fi situate tabelele de descriptori și directorul curent;
    - **aria P** (**proc area** sau **Per Process Region Table**): tabele în pagini ce definesc modul de mapare al adreselor virtuale, definind astfel concret zonele de cod, date, stivă;
  - **partea dinamică**, formată din locații pe stiva nucleului (corespunzatoare rutinelor nucleului aflate în apel pentru procesul respectiv); în PCB se află un pointer către vârful zonei dinamice; în diverse implementari fiecare proces are o copie proprie a stivei nucleului situata în aria U.

Implementarea proceselor

De obicei:
- în PCB sunt menținute informații folosite de nucleu care pot fi accesate mereu, chiar și când procesul nu este activ (de ex. când nucleul efectuează operații de planificare): prioritatea, etc.; de aceea PCB este păstrat doar în memoria principală;
- în aria U sunt menținute informații folosite de nucleu necesare doar când procesul este activ (rulează în cadrul contextului său) și de aceea ea se swap-eaza pe disc în cadrul imaginii procesului;

În general, partea contextului care nu este accesata când procesul nu este activ (se exclude deci PCB) se swap-eaza pe disc sub forma imaginii procesului.

Cuprins

Thread-uri - generalități

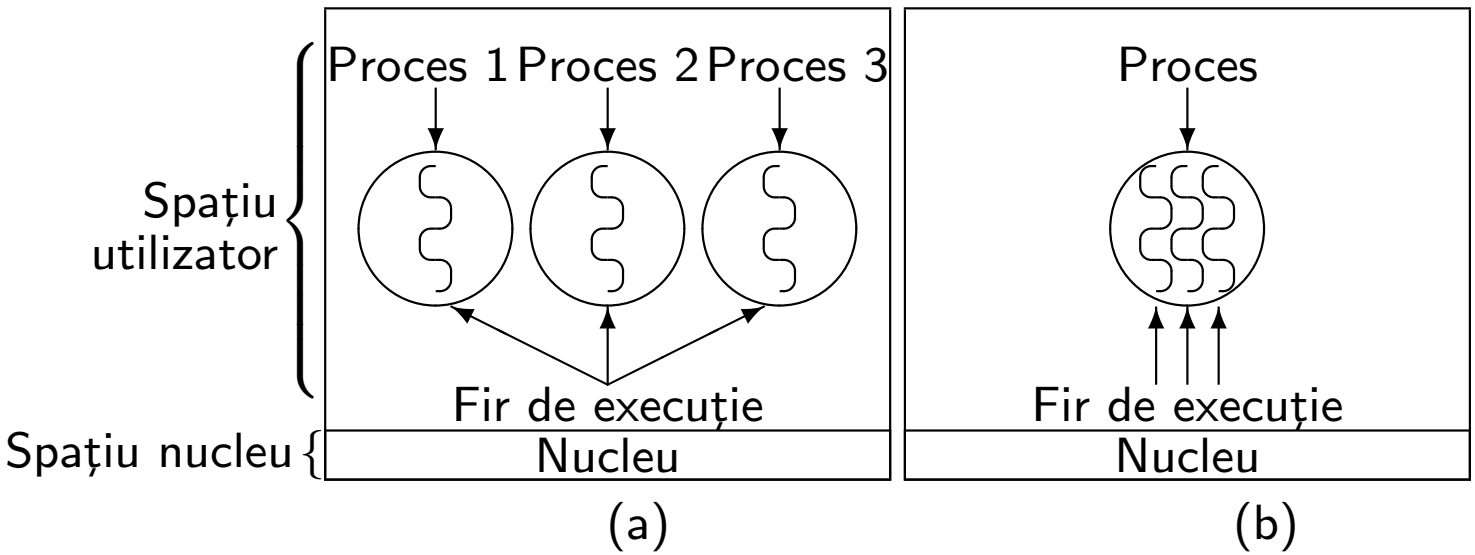
Modelul orientat pe procese discutat pană acum se bazează pe două concepte distincte: gruparea resurselor și execuția.

Un proces tradițional are un spațiu de adrese, un grup de resurse externe și un fir de excuție.
Putem modela existența mai multor **fire de execuție** (**thread-uri**) care se execută pseudo-paralel (intercalat) în același spațiu de adrese și operând pe același grup de resurse externe; procesorul comută între thread-uri (așa cum comută și între procese) creând iluzia că thread-urile se execută în paralel, dar pe procesoare mai lente.

Astfel apar două concepte distincte:
procesul - este o grupare de resurse;
thread-ul - este o entitate planificabilă la execuție.

Thread-urile se mai numesc și procese ușoare (lightweight process).
Programarea cu fire de execuție multiple (**multithreading**) se referă la situația când se permit mai multe fire de execuție în cadrul aceluiași proces.

Thread-uri - generalități



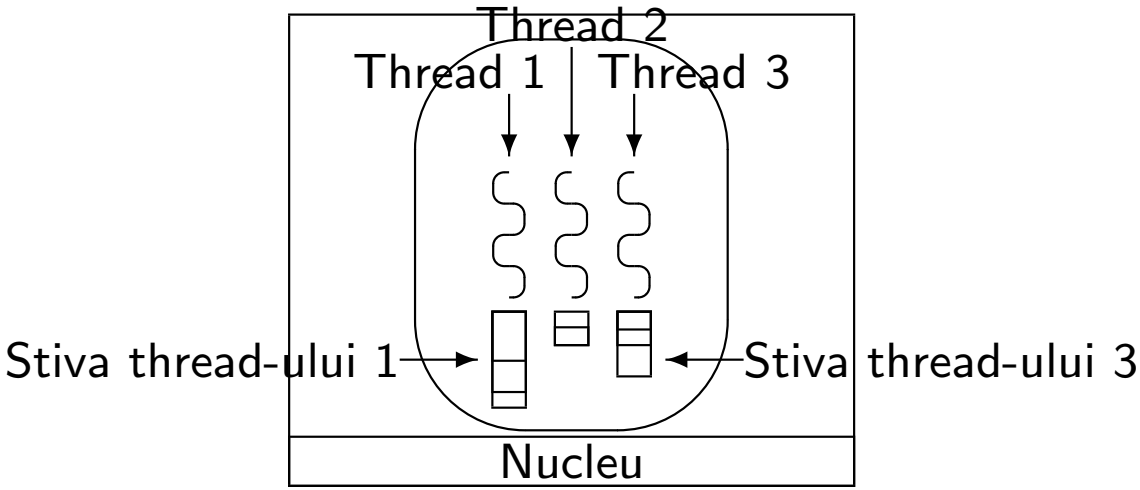
(a) Trei procese cu câte un fir de execuție.
(b) Un proces cu trei fire de execuție.

Thread-uri - generalități

Procesele distincte pot proveni de la utilizatori diferiți și pot fi ostile unul altuia; thread-urile unui proces provin de la același utilizator și în principiu ele sunt făcute să coopereze.

De aceea implementarea unor activități sub forma de procese este de preferat când activitățile sunt esențial diferite, iar implementarea lor sub formă de thread-uri în același proces este de preferat când activitătile fac parte din aceeași sarcină și cooperează la îndeplinirea ei.

Thread-uri - generalități



Thread-urile unui proces folosesc în comun spațiul de adrese și resursele procesului (fișiere deschise, semnale primite, etc.) dar pentru fiecare thread se reține un context propriu format din valorile regiștrilor, o anumită stare (asemănătoare stărilor procesului, cu tranziții asemănătoare), și o stivă proprie (alocată în spațiul de adrese al procesului gazdă) - acestea pentru a permite comutarea între thread-uri a.î. la reluare fiecare thread să continue consistent de unde a rămas.

Thread-uri - generalități

Elemente repartizate **"per proces"**:
- codul programului, variabilele globale, în general spațiul de adrese;
- tabela de descriptori de fișiere;
- colecția proceselor copil;
- tabela de gestiune a semnalelor primite;
- handler-ele asociate tipurilor de semnal;
- etc. (a se vedea secțiunea "Implementarea proceselor" mai sus).

Elemente repartizate **"per thread"**:
- **regiștri** (inclusiv PC);
- **stiva** (parte a spațiului de adrese al procesului gazdă);
- **stare**: **running**, **ready**, **blocked** (așteaptă întrunirea anumitor condiții - anumite resurse să devină disponibile, un alt thread să-l deblocheze, etc.), **terminated**.

Deci, toate thread-urile unui proces văd aceleași cod, aceleași variabile globale, iar dacă un thread deschide un fișier, acesta este accesibil și celorlalte thread-uri, care pot citi/scrie în el; în schimb, fiecare thread are propria evidență a rutinelor apelate de el (stiva).

Thread-uri - generalități

Deoarece thread-urile nu au resurse alocate, sunt mai ușor de creat/distrus/comutat decât procesele; acest lucru se simte mai ales când numarul firelor de execuție este dinamic și se schimbă rapid - astfel, este mai eficient să implementăm aceste fire ca thread-uri decât ca procese.

Thread-uri - generalități

În cadrul programării multithreading, un proces începe de regulă cu un singur thread, care generează apoi alte thread-uri. Pentru manevrarea thread-urilor, se folosesc anumite funcții de bibliotecă:

thread_create() $\Longrightarrow$ crează un thread nou; primește ca parametru numele unei proceduri care se va executa în noul thread; thread-ul creator primește un identificator pentru thread-ul nou (în unele sisteme se pot stabili astfel ierarhii de thread-uri);

thread_exit() $\Longrightarrow$ thread-ul curent se termină (nu mai e planificabil la execuție);

thread_wait() $\Longrightarrow$ thread-ul curent așteaptă (se blochează) terminarea unui anume alt thread;

thread_yield() $\Longrightarrow$ thread-ul renunță momentan la procesor (permițând altui thread să se execute); este importantă mai ales când mecanismul planificării thread-urilor este implementat în programul rulat (nu în nucleu) și astfel nu există o întrerupere de ceas care să determine comutarea thread-urilor (ceasul este folosit la comutarea între procese);

În UNIX/Linux există biblioteca "pthread" cu implementarea POSIX a thread-urilor (detalii se pot obține cu comanda shell "man pthread.h").

Thread-uri - generalități

Exemple de programe ce se pot implementa eficient cu thread-uri multiple:

• un editor de texte, având:
- un thread care interacționează cu utilizatorul, așteptând input;
- un thread care face toate modificările implicate de o corecție - de ex. eliminarea unei fraze impune translatarea/reformatarea tuturor paginilor următoare existente;
- un thread care salveaza periodic pe disc;
(toate operează pe aceeași porțiune de document din memorie și cu aceleași fișiere de pe disc conținând restul capitolelor - aceste resurse sunt grupate la nivelul procesului editor);

Thread-uri - generalități

Exemple de programe ce se pot implementa eficient cu thread-uri multiple:

• un server web, având:
- un thread dispecer, care preia cererile de pagini venite prin rețea, găsește un thread lucrător liber și-i pasează cererea (în particular, îl trece din starea blocked în starea ready);
- mai multe thread-uri lucrător; fiecare așteaptă (blocked) primirea unei cereri de la dispecer, apoi caută pagina cerută în memoria cache, dacă nu o găsește o încarcă de pe disc, apoi o livrează clientului, apoi iar trece iar în așteptare;
(toate operează pe aceeași memorie cache cu pagini încărcate și pe aceleași fișiere cu pagini de pe disc - aceste resurse sunt grupate la nivelul procesului server web).

Thread-uri - generalități

Exemple de programe ce se pot implementa eficient cu thread-uri multiple:

• un joc RTS în care fiecare personaj (unitate) este manevrat de un thread, care în mod independent investighează imprejurimile, ia deciziile de AI, interacționează cu alte unități - toate operând pe aceeași tablă, alocată la nivelul procesului joc.

Cuprins

Implementarea thread-urilor

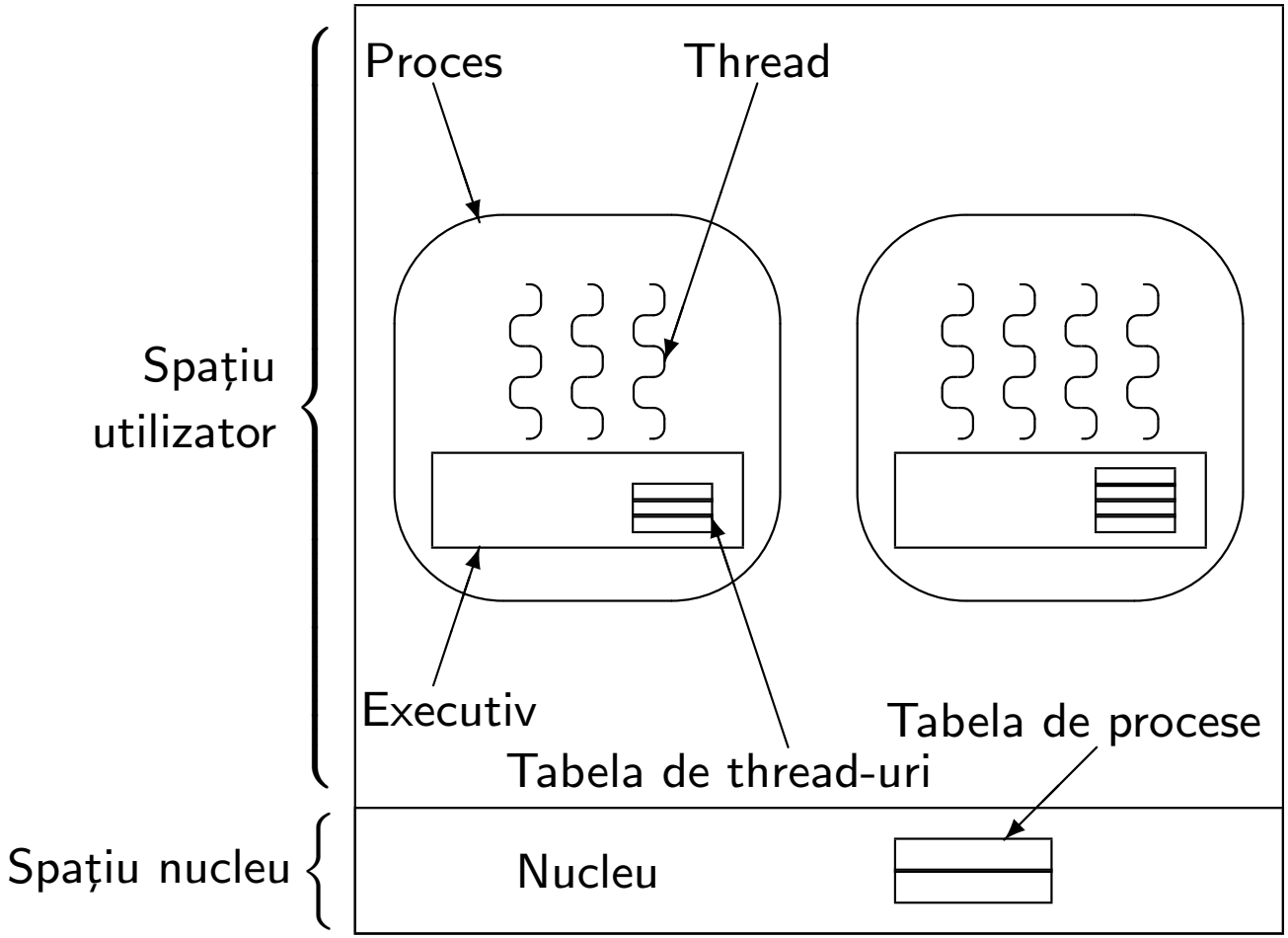
Implementarea thread-urilor se poate face:
- în spațiul utilizator;
- în nucleu;
- hibrid.

Implementarea în spațiul utilizator

Implementarea în spațiul utilizator:
- gestiunea thred-urilor este făcută de proces (nucleul nu știe de thread-uri, el gestionează procesele);
- procesul are un mecanism ce mimează planificatorul nucleului, format din:
  - un **executiv** (**run-time system**), care planifică thread-urile și conține o colecție de rutine pentru gestiunea lor, de tipul thread_create(), thread_exit(), thread_wait(), thread_yield();
  - o **tabelă de thread-uri** (**thread table**), gestionată de executiv, care deține evidența thread-urilor procesului - o intrare în ea ține evidența elementelor repartizate "per thread" ale unui thread: regiștri (mai ales PC si SP), stare, etc;
  - pentru a efectua acțiuni ce privesc starea lor (blocare, deblocare, etc.) thread-urile apelează rutinele executivului;

Mecanismul de gestiune a thread-urilor poate fi adăugat programului rulat de proces direct de programator sau atașat de compilator dintr-o bibliotecă.

Implementarea în spațiul utilizator



Un pachet de thread-uri în spațiul utilizator.

Implementarea în spațiul utilizator

Avantajele implementării în spațiul utilizator:

- se poate implementa în SO existente, care nu suportă nativ thread-uri;
- planificarea thread-urilor efectuată de proces (prin executiv) este mai rapidă decât cea efectuată de nucleu (care presupune TRAP-uri, salvări/restaurări de context, etc.);
- în fiecare proces se poate implementa un algoritm propriu de planificare;
- implementarea este mai scalabilă (implementarea în nucleu ar necesita mult spațiu pentru tabelă și stivă în interiorul nucleului, pentru toate thread-urile tuturor proceselor).

Implementarea în spațiul utilizator

Dezavantajele implementării în spațiul utilizator:

- dacă un thread efectuează un apel sistem blocant (ex: "read()" de la consolă și încă nu s-a tastat nimic) tot procesul (deci toate thread-urile lui) se blochează;

o soluție: reproiectarea SO a.î. toate apelurile sistem să fie neblocante (de ex. "read()" de la consola să nu blocheze procesul ci să returneze 0 dacă nu sunt disponibile caractere tastate), sau adăugarea de **cod jachetă** (**jacket**) sau **înveliș** (**wrapper**) pentru apelurile sistem (ex. apelul "select()" în anumite versiuni de UNIX), care verifică dacă apelurile respective se vor bloca, a.î. utilizatorul (thread-ul curent) să nu le mai facă (și să comute pe alt thread, iar la revenire să verifice din nou);

Implementarea în spațiul utilizator

Dezavantajele implementării în spațiul utilizator:

- unele sisteme se pot configura a.î. nu tot programul rulat de un proces trebuie să fie în memoria principală; când este necesară o altă porțiune a sa, sistemul (nucleul) o încarcă automat; atunci, dacă un thread efectuează un salt la o instrucțiune ce nu este în memoria principală, apare un **defect de pagină** (**page fault**) și întregul proces (deci toate thread-urile sale) este blocat de nucleu până se încarcă porțiunea de cod respectivă;

Implementarea în spațiul utilizator

Dezavantajele implementării în spațiul utilizator:

- deoarece nucleul nu știe de existența thread-urilor, nu poate declanșa (cu ajutorul ceasului) oprirea și comutarea între ele; de aceea thread-urile trebuie să ceară explicit renunțarea la procesor și redarea controlului către executiv pentru planificarea altui thread (cu thread_yield());

o soluție: în unele sisteme procesul (în particular un thread) poate programa ceasul pentru a-i trimite un semnal (alarmă) după un anumit interval de timp (apelul "alarm()" din UNIX/Linux); la primirea semnalului se execută un handler utilizator, care poate reda controlul executivului; astfel însă ceasul nu mai poate fi programat de proces/thread și pentru alte scopuri, iar multitudinea alarmelor (care presupun în particular întreruperi și apeluri sistem) încetinesc mult procesul.

Implementarea în spațiul utilizator

Dezavantajele implementării în spațiul utilizator:

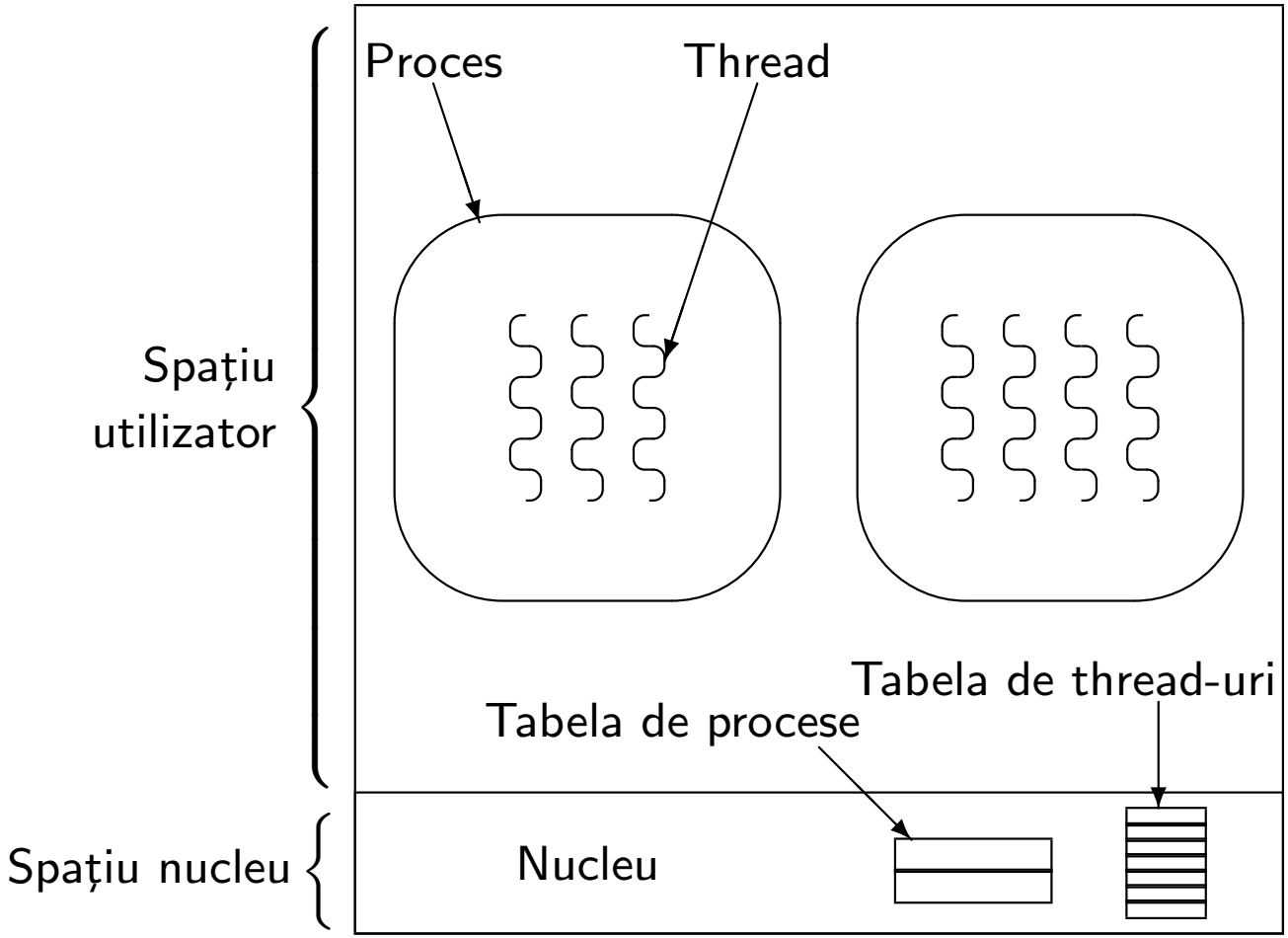
- de obicei programatorii doresc mai multe thread-uri în aplicații care fac multe apeluri sistem și se blochează des; ori, odată efectuat TRAP-ul către nucleu, acestuia i-ar fi mult mai ușor să comute și între thread-uri; astfel, n-ar mai fi necesare codurile jacheta pentru a verifica dacă apelurile sistem sunt sau nu blocante.

Implementarea în nucleu

Implementarea în nucleu:
- gestiunea thred-urilor este facută de nucleu concomitent cu gestiunea proceselor; procesele sunt cele obișnuite - nu mai au mecanisme proprii (executiv, tabelă de thread-uri) pentru gestionat thread-urile;
- nucleul are o **tabelă de thread-uri** (pe langă tabela de procese) pentru menținerea evidenței tuturor thread-urilor din sistem; informația din intrările tabelei este asemănatoare ca și în cazul implementării în spațiul utilizator, dar este acum stocată în nucleu, ca parte a contextului procesului;
- rutinele apelabile din procese/thread-uri pentru manevrarea thread-urilor (thread_create(), thread_exit(), thread_wait(), thread_yield(), etc.) sunt implementate acum ca apeluri sistem;

Când un thread se blochează, nucleul poate planifica alt thread, din același proces sau altul.
În cazul implementării în spațiul utilizator, executivul planifica thread-uri din același proces pană ce nucleul îi lua procesorul.

Implementarea în nucleu



Un pachet de thread-uri gestionat de nucleu.

Implementarea în nucleu

Avantaje/dezavantaje/soluții ale implementării în nucleu:

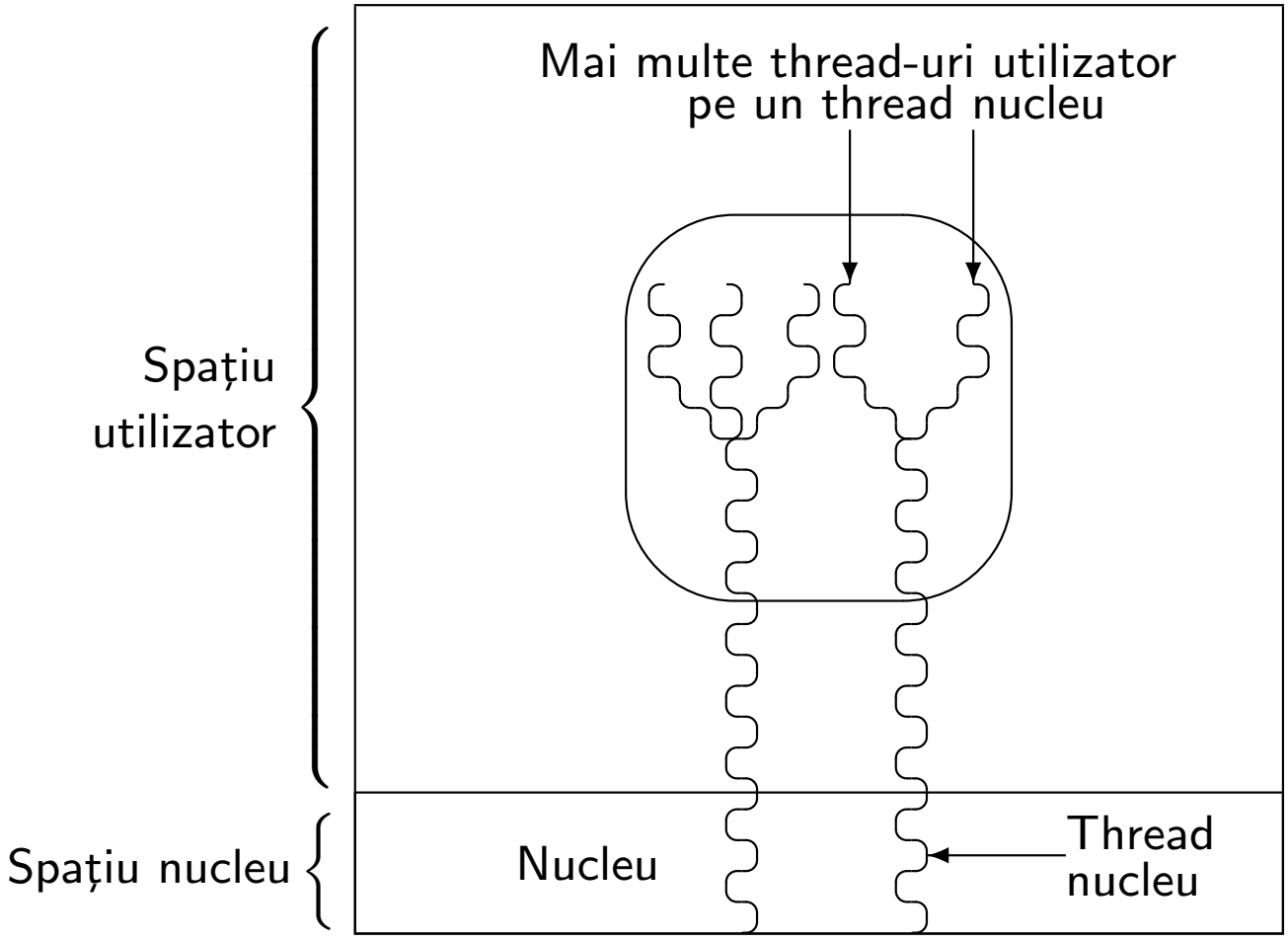
- thread-urile implementate în nucleu nu necesită apeluri sistem neblocante (în urma unui apel blocant venit de la un proces, nucleul, care are acces acum la thread-urile sale, poate face să fie blocat doar thread-ul care a efectuat apelul, nu tot procesul); de asemenea, dacă un proces (printr-un thread al lui) cauzează un defect de pagină, nucleul (care are acum acces la thread-urile procesului) va bloca doar thread-ul care a cerut instrucțiunea inexistentă în memorie (nu tot procesul) până se va încărca pagina dorită de pe disc în memoria principală (și poate planifica alte thread-uri să se execute în acest timp).
- apelurile ce pot crea/distruge/bloca un thread, ca apeluri sistem, consumă mai mult timp decât apelarea rutinelor executivului;
  o soluție: anumite sisteme **reciclează** thread-urile - când un thread este distrus, structurile sale de date din nucleu nu sunt alterate ci doar thread-ul este marcat ca neexecutabil; ulterior, când se cere crearea unui thread nou, se poate reactiva unul mai vechi.

Implementări hibride

Implementări hibride:

Pentru a combina avantajele celor două implementări de mai înainte, unele sisteme folosesc atât thread-uri la nivel nucleu cât și thread-uri la nivel utilizator.

Implementări hibride



Multiplexarea thread-urilor la nivel utilizator pe thread-uri la nivel nucleu.

Implementări hibride

Mai multe thread-uri utilizator se pot multiplexa în cadrul mai multor thread-uri nucleu - astfel, un thread nucleu poate fi utilizat pe rând de mai multe thread-uri utilizator.

Implementări hibride

O abordare hibridă proiectată de Anderson et al. (1992) s.n. **activarea planificatorului** (**scheduler activations**).

În acest model, dacă un thread se blochează (a efectuat un apel sistem blocant, a cauzat un defect de pagină, etc.), nucleul nu blocheaza tot procesul ci doar informează executivul acestuia, printr-un **apel către nivelul superior** (**upcall**), transmițând ca parametri pe stivă numarul thread-ului și natura evenimentului apărut; odată informat, executivul marchează ca blocat thread-ul respectiv și își poate planifica alt thread; ulterior, când nucleul constată că thread-ul își poate continua execuția, face iar upcall către executiv pentru a-l informa; atunci executivul anulează marcajul și trece firul în starea ready sau running.

O obiecție la modelul activării planificatorului este folosirea upcall-urilor, care încalcă structura internă a unui SO organizat pe niveluri.

Cuprins

Dificultăți

Dificultăți privind implementarea thread-urilor și programarea în manieră multithread a unor activități programate de obicei într-un singur thread:

- Dacă un proces are mai multe thread-uri și generează un proces copil, aceleași thread-uri sunt prezente și în copil ?

Dacă nu, copilul ar putea să nu funcționeze corect, deoarece toate firele (în cooperarea lor) ar putea fi esențiale.

Dacă da, ce se întamplă dacă un thread din părinte era blocat într-un "read()" de la tastatură ? Există acum două thread-uri blocate (în parinte și copil) ? Primesc ambele thread-uri o copie a liniei tastate ? Dacă nu, cine o primește ? Similar în cazul conexiunilor de rețea deschise.

Dificultăți

Dificultăți privind implementarea thread-urilor și programarea în manieră multithread a unor activități programate de obicei într-un singur thread:

- Întrucât thread-urile unui proces folosesc în comun resursele procesului, ce se întâmplă dacă un thread închide un fișier (deci el se închide la nivel de proces) în timp ce alt thread încă citește din el ?

Dar dacă un thread, constatând că este alocată prea puțina memorie, începe să aloce, între timp apare o comutare între thread-uri, iar noul thread, constatând și el că este alocată prea puțină memorie, începe sa aloce, aceeași memorie se va aloca de două ori ?

Dificultăți

Dificultăți privind implementarea thread-urilor și programarea în manieră multithread a unor activități programate de obicei într-un singur thread:

- utilizarea inconsistentă a variabilelor globale;
  de exemplu, în UNIX/Linux apelurile sistem setează variabila globală "errno" cu un număr ce indică succesul sau, în caz de eșec, codul erorii; dar, următorul scenariu:
  - thread-ul 1 face apelul sistem 1, care eșuează și seteaza "errno";
  - planificatorul comută pe thread-ul 2;
  - thread-ul 2 face apelul sistem 2, care suprascrie "errno";
  - planificatorul comută pe thread-ul 1;
  - thread-ul 1 testează errno;

  va face ca thread-ul 1 să citească un cod greșit și să se comporte incorect (citește codul furnizat de apelul 2, crezând că e cel furnizat de apelul 1);
  o soluție: implementarea unui concept de variablile globale la nivel de thread (concept intermediar între variabilele globale și cele locale).

Dificultăți

Dificultăți privind implementarea thread-urilor și programarea în manieră multithread a unor activități programate de obicei într-un singur thread:

- multe proceduri de bibliotecă nu sunt reentrante (nu se garantează funcționarea corectă dacă se face un al doilea apel înainte de a se termina primul); de exemplu "malloc()" menține tabele de folosire a memoriei, care se pot afla în stare de inconsistență, cu indici invalizi, în timp ce "malloc()" le modifică; dacă în timpul unui apel "malloc()" se comută pe un alt thread care face și el "malloc()", acesta va manevra adrese invalide putându-se ajunge la **căderea programului** (**program crash**);
  o soluție: crearea de jachete pentru fiecare procedură, care dă valorile 1/0 unui bit pentru a marca faptul că biblioteca este folosită; cât timp bitul este setat, orice încercare a unui alt thread de a apela o procedură a bibliotecii va bloca thread-ul respectiv; această abordare însă elimină în mare masură paralelismul.

Dificultăți

Dificultăți privind implementarea thread-urilor și programarea în manieră multithread a unor activități programate de obicei într-un singur thread:

- tratarea semnalelor primite este mai dificilă;
dacă un thread programează ceasul pentru a primi după un interval de timp o alarmă (semnalul SIGALRM în UNIX/Linux) sau încearcă să acceseze o locație în afara spațiului de adrese (fapt de natură să îi atragă primirea semnalului SIGSEGV în UNIX/Linux), ar fi normal ca semnalul să fie direcționat spre thread-ul respectiv; dacă însă thread-urile sunt implementate în spațiul utilizator, semnalele nu se pot direcționa la nivel de thread - le primește procesul ca un tot iar handlerul asociat se va executa în cadrul thread-ului activ în momentul primirii.

Dificultăți

Dificultăți privind implementarea thread-urilor și programarea în manieră multithread a unor activități programate de obicei într-un singur thread:

- în unele sisteme, când stiva unui proces crește prea mult, nucleul îi alocă mai mult spațiu; thread-urile au însă fiecare stiva proprie, în cadrul spațiului de adrese al procesului; dacă thread-urile sunt implementate în spațiul utilizator, nucleul nu știe de aceste stive și astfel, dacă ele cresc prea mult, nu va aloca memorie suplimentară - dealtfel este posibil ca nucleul să nu realizeze că o eroare de memorie este legată de creșterea stivei.

Cuprins

UNIX/Linux - generalități

Așa cum am mai spus, un proces se poate afla:
- în kernel mode: atunci poate excuta instrucțiuni hardware privilegiate și are acces la datele nucleului; în acest mod se află când execută un apel sistem sau întrerupere;
- în user mode: atunci poate executa doar instrucțiuni hardware neprivilegiate și are acces la spațiul său de adrese; în acest mod se află cand execută instrucțiuni scrise în programul executat.

UNIX/Linux - generalități

Spațiile de adrese ale proceselor sunt disjuncte (nu poate un proces să acceseze spațiul de adrese al altui proces); excepție: memoriile partajate IPC (a se vedea cursul 6). Spațiul de adrese al unui proces conține, printre altele, locațiile pe care procesul le are pentru variabilele din program și stiva utilizator a procesului.

Dacă un proces încearcă, prin instrucțiuni din programul executat (fiind deci în user mode) să acceseze o zonă din afara spațiului său de adrese, nu reușește, iar procesul primește semnalul SIGSEGV (handler-ul implicit termină procesul, dar se poate asigna un handler utilizator).

UNIX/Linux - generalități

Dacă un proces se află în kernel mode (de exemplu execută un apel sistem) și primește niște semnale, acestea nu sunt tratate ci ramân în așteptare (pending) la proces. Abia când procesul trece din kernel mode în user mode (revine din apelul sistem) tratează semnalele aflate în așteptare (ținând cont și de alte aspecte, a se vedea cursul 6); tratarea unui semnal constă în întreruperea cursului normal al execuției și executarea handler-ului asociat semnalului.

UNIX/Linux - generalități

Apelurile sistem pot fi blocante (pot pune procesul în starea sleeping până când sunt întrunite anumite condiții, apoi fac return și lasă procesul să continue) sau nu.

De exemplu citirea cu "read()" dintr-un fișier tub care este vid dar are scriitori (a se vedea cursul 6) pune procesul apelant în sleeping până când cineva scrie date în tub (și atunci "read()" efectuează citirea, face return, iar procesul continuă) sau tubul pierde scriitorii (și atunci "read()" returnează 0 iar procesul continuă). Dacă însă tubul a fost deschis cu "open()" și opțiunea "O_NONBLOCK", apelul "read()" asupra tubului respectiv nu va bloca niciodată procesul apelant (va returna întotdeauna imediat, eventual un cod de eroare).

UNIX/Linux - generalități

Compilatoarele de C (de ex. "gcc" din Linux) adaugă executabilelor create cod de interfațare cu apelurile sistem - astfel, din program putem apela un apel sistem ca pe o funcție C.

Codul respectiv poate proveni dintr-o bibliotecă cu legare statică și integrat în executabil, sau dintr-o bibliotecă cu legare dinamică și doar referit din executabil direct în bibliotecă în timpul rulării (de ex. "libc.so").

UNIX/Linux - generalități

Acest cod conține în principal:
- o variabilă globală "`int errno`", care va fi setată automat după fiecare apel sistem cu un număr ce indică succesul sau, în caz de eșec, codul erorii;
  valorile posibile ale lui "errno" pot fi desemnate prin constante simbolice predefinite: EACCES, EAGAIN, etc.;
  fișierul "errno.h" conține o declarație "`extern int errno;`" și definiția constantelor simbolice ce desemnează valorile sale;
  variabila "errno" poate fi accesată direct din programul utilizator, însă pentru asta trebuie să declarăm în el "extern int errno;" (sau "`#include<errno.h>`", și atunci avem acces și la constantele simbolice de mai sus);
- o funcție "`void perror(const char *s)`" (necesită "`include<stdio.h>`"), care afișază pe stderr stringul "s", urmat de ":" și un mesaj ce descrie eroarea a cărui cod se află curent în "errno" (deci motivul eșecului ultimului apel sistem efectuat);
- câte o funcție C pentru fiecare apel sistem apelat - aceasta este apelată din program în maniera C (cu parametri prin stivă, etc.) și realizează interfațarea cu apelul sistem în maniera assembler: ia parametrii din stivă și îi încarcă conform cerințelor apelului (în regiștri și stivă), efectuează apelul în maniera assembler ("`int $0x80`"), apoi recuperează rezultatele (din regiștri), stetează variabila "errno" conform acestor rezultate și returneaza o valoare în maniera C;
  în general, în caz de eșec aceste funcții returnează -1.

UNIX/Linux - generalități

În cele ce urmează, prin "apeluri sistem" ne vom referi la funcțiile C de interfațare de mai sus.

Obs: 1. Apeluri sistem diferite pot da o aceeași valoare lui "errno" (de ex. EACCES = Permission denied) dacă motivul eșecului seamănă, chiar dacă operația încercată a fost diferită;

2. Diverse funcții de bibliotecă pot seta "errno", nu numai cele de interfațare cu apelurile sistem.

UNIX/Linux - generalități

Listă de valori posibile ale lui "errno" (obținute cu "man errno.h"):

E2BIG = Argument list too long.
EACCES = Permission denied.
EADDRINUSE = Address in use.
EADDRNOTAVAIL = Address not available.
EAFNOSUPPORT = Address family not supported.
EAGAIN = Resource unavailable, try again (may be the same value as [EWOULDBLOCK]).
EALREADY = Connection already in progress.
EBADF = Bad file descriptor.
EBADMSG = Bad message.
EBUSY = Device or resource busy.
ECANCELED = Operation canceled.
ECHILD = No child processes.
ECONNABORTED = Connection aborted.
ECONNREFUSED = Connection refused.
ECONNRESET = Connection reset.
EDEADLK = Resource deadlock would occur.
EDESTADDRREQ = Destination address required.
EDOM = Mathematics argument out of domain of function.

EDQUOT = Reserved.
EEXIST = File exists.
EFAULT = Bad address.
EFBIG = File too large.
EHOSTUNREACH = Host is unreachable.
EIDRM = Identifier removed.
EILSEQ = Illegal byte sequence.
EINPROGRESS = Operation in progress.
EINTR = Interrupted function.
EINVAL = Invalid argument.
EIO = I/O error.
EISCONN = Socket is connected.
EISDIR = Is a directory.
ELOOP = Too many levels of symbolic links.
EMFILE = Too many open files.
EMLINK = Too many links.
EMSGSIZE = Message too large.
EMULTIHOP = Reserved.
ENAMETOOLONG = Filename too long.
ENETDOWN = Network is down.
ENETRESET = Connection aborted by network.

ENETUNREACH = Network unreachable.
ENFILE = Too many files open in system.
ENOBUFS = No buffer space available.
ENODATA = No message is available on the STREAM head read queue.
ENODEV = No such device.
ENOENT = No such file or directory.
ENOEXEC = Executable file format error.
ENOLCK = No locks available.
ENOLINK = Reserved.
ENOMEM = Not enough space.
ENOMSG = No message of the desired type.
ENOPROTOOPT = Protocol not available.
ENOSPC = No space left on device.
ENOSR = No STREAM resources.
ENOSTR = Not a STREAM.
ENOSYS = Function not supported.
ENOTCONN = The socket is not connected.
ENOTDIR = Not a directory.
ENOTEMPTY = Directory not empty.
ENOTSOCK = Not a socket.
ENOTSUP = Not supported.

ENOTTY = Inappropriate I/O control operation.
ENXIO = No such device or address.
EOPNOTSUPP = Operation not supported on socket.
EOVERFLOW = Value too large to be stored in data type.
EPERM = Operation not permitted.
EPIPE = Broken pipe.
EPROTO = Protocol error.
EPROTONOSUPPORT = Protocol not supported.
EPROTOTYPE = Protocol wrong type for socket.
ERANGE = Result too large.
EROFS = Read-only file system.
ESPIPE = Invalid seek.
ESRCH = No such process.
ESTALE = Reserved.
ETIME = Stream ioctl() timeout.
ETIMEDOUT = Connection timed out.
ETXTBSY = Text file busy.
EWOULDBLOCK = Operation would block (may be the same value as [EAGAIN]).
EXDEV = Cross-device link.

UNIX/Linux - generalități

Exemplu de utilizare a apelurilor sistem (prin interfațarea C):

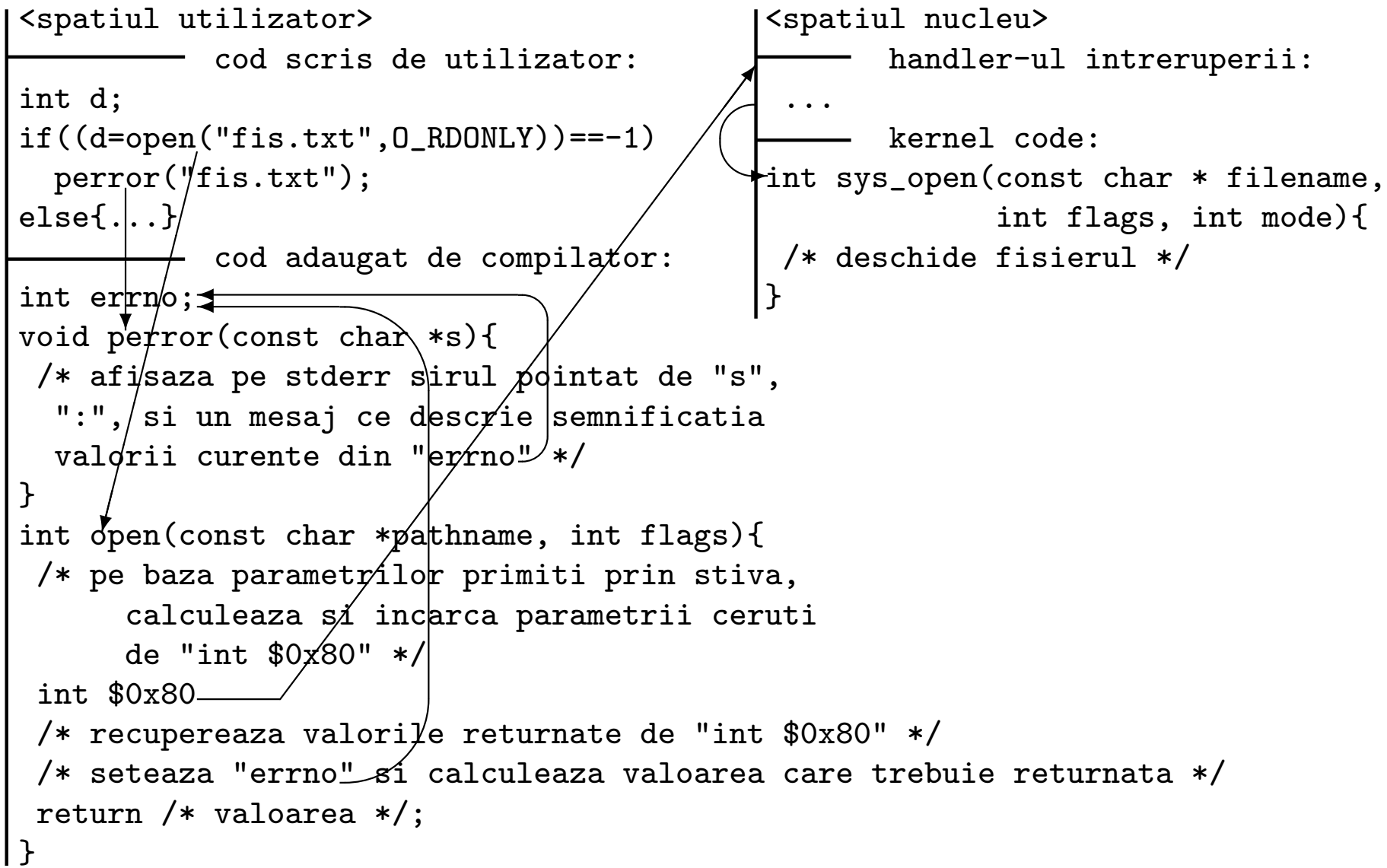
```
int d;
  ...
/* incercam sa deschidem fisierul "fis.txt" in citire */
if((d=open("fis.txt",O_RDONLY))==-1)
  perror("fis.txt"); /* cazul de eroare */
else{
  ...              /* accesam fisierul cu ajutorul descriptorului "d" */
}
```

Dacă fișierul "fis.txt" nu există în directorul curent, "open" eșuează returnând -1 și setează "errno" cu valoarea "ENOENT"; atunci se intră pe prima ramură "if", iar "perror" va afișa pe stderr următorul text (în concordanță cu valoarea "ENOENT" pe care a găsit-o în "errno"):

```
fis.txt: No such file or directory
```

UNIX/Linux - generalități

Practic, s-au executat următoarele:



UNIX/Linux - generalități

Câteva caractersistici ale unui proces (reținute în PCB sau aria U):

- **PID** (**process id**): identificatorul numeric al procesului;
  este unic în instanța UNIX/Linux din care face parte procesul
  și este mijlocul uzual prin care ne putem referi la proces;
- **PPID** (**parent process id**): identificatorul procesului părinte;
- **proprietarul real**, **proprietarul efectiv**,
  grupul proprietarului real, **grupul proprietarului efectiv**
  (identificatorii lor numerici: **UID**, **EUID**, **GID**, **EGID**):
  proprietarul real este moștenit de la procesul părinte,
  proprietarul efectiv este cel ale cărui drepturi sunt luate
  în considerație când procesul încearcă sa facă ceva
  (de exemplu să deschidă un fișier);
  în general, proprietarul efectiv coincide cu proprietarul real
  sau cu proprietarul fișierului executat de proces, în funcție
  de valoarea 0/1 a bitului SETUID din i-nodul fișierului
  executat de proces (a se vedea cursul despre gestiunea fișierelor);

UNIX/Linux - generalități

Câteva caractersistici ale unui proces (reținute în PCB sau aria U):

- **directorul curent**: la el își raportează procesul căile relative de specificare a fișierelor;
- **terminalul de control**: în general, procesele interpretoare de comenzi (shell) au stdin, stdout, stderr pe terminalul lor de control;
  procesele fiu lansate în urma unor comenzi shell externe moștenesc atât terminalul cât și redirectarea stdin, stdout, stderr, iar dacă n-am specificat alte redirectări în linia de comandă vor avea și ele stdin, stdout, stderr pe terminalul respectiv ("printf", "scanf" vor fi la terminal);
- **environment-ul**: colecție de variabile asignate la valori string;

UNIX/Linux - generalități

Câteva caractersistici ale unui proces (reținute în PCB sau aria U):

- **tabla de descriptori**: cu ajutorul descriptorilor procesul își referă fișierele deschise;
- **tabela de gestiune a semnalelor primite**
- **tabela de handlere asociate semnalelor**
- **prioritatea**
- **starea**
- **timpul procesor consumat**
- **regiștrii** (dacă nu este running)

UNIX/Linux - setare/consultare caracteristici

Apeluri sistem cu care procesul își poate consulta/modifica unele dintre aceste caracteristici:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

$\Longrightarrow$ returnează PID/PPID-ul procesului;

```
#include <unistd.h>
#include <sys/types.h>
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

$\Longrightarrow$ returnează UID/EUID/GID/EGID-ul procesului;

`pid_t`, `uid_t`, `gid_t` sunt definite ca tipuri întregi.

UNIX/Linux - setare/consultare caracteristici

```
#include <sys/types.h>
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

$\Longrightarrow$ setează UID, EUID / GID, EGID-ul procesului;
un parametru -1 lasă ID-ul respectiv neschimbat;
dacă vreunul din ID-uri este setat la altă valoare decât
vechiul UID, noul EUID este salvat ca "saved set-user-ID";
procesele neprivilegiate (de ex. care nu au proprietar pe "root")
pot seta doar EUID la UID, EUID, saved set-user-ID,
și pe UID la UID, EUID;
(prin asemenea schimbări, un proces poate renunța temporar)
la privilegii, efectuează o activitate neprivilegiată,
apoi recapată privilegiile (revenind la saved set-user-ID);
la succes returnează 0, la eșec returnează -1
errno posibile: EPERM;

alte apeluri înrudite: `setuid()`, `seteuid()`, `setgid()`, `setegid()`.

UNIX/Linux - setare/consultare caracteristici

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

$\Longrightarrow$ copiază calea absolută a directorului curent în zona pointată de "buf", presupusă a fi de lungime size; dacă este necesară o lungime mai mare, returnează NULL și setează errno la valoarea ERANGE; în caz de succes returnează "buf"; în caz de eroare conținutul final al zonei pointate de "buf" este nedefinit;

errno posibile: EACCES, EFAULT, EINVAL, ENOENT, ERANGE.

```
#include <unistd.h>
int chdir(const char *path);
```

$\Longrightarrow$ schimbă directorul curent în cel specificat de "path"; la succes retrurnează 0, la eșec returnează -1;

errno posibile: EACCES, EFAULT, EIO, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR.

Din interior un proces își poate accesa terminalul de control folosind specificatorul generic "/dev/tty".

UNIX/Linux - setare/consultare caracteristici

Un proces își poate accesa environment-ul folosind:

- al treilea parametru al lui "main", care trebuie să fie de tip "`char **`";

- variabila globală predefinită "`environ`" (adăugată automat de compilator, ca și "`errno`"), pentru care trebuie să includem în program declarația "`extern char **environ;`";

(în ambele cazuri este accesat un șir de șiringuri de forma "variabilă=valoare", terminat cu o componentă NULL)

UNIX/Linux - setare/consultare caracteristici

- apelurile:

```
#include <stdlib.h>
char *getenv(const char *name);
```

$\Longrightarrow$ caută în environment un string de forma "variabilă=valoare" unde "variabilă" este specificat de "`name`" și returnează un pointer către valoarea corespunzătoare "valoare", sau NULL dacă variabila nu este găsită;

```
#include <stdlib.h>
int putenv(char *string);
```

$\Longrightarrow$ adaugă la environment stringul specificat de "`string`", care trebuie să fie de forma "variabilă=valoare"; dacă există deja o variabilă cu numele respectiv, doar îi schimbă valoarea;

la succes returnează 0, la eșec returnează -1;

errno posibile: ENOMEM (dacă nu mai e loc în environment);

```
#include <stdlib.h>
int unsetenv(const char *name);
```

$\Longrightarrow$ elimină din environment variabila cu numele specificat de "`name`" (împreună cu valoarea ei), dacă există (altfel nu face nimic);

la succes returnează 0, la eșec returnează -1;

errno posibile: EINVAL (dacă numele variabilei conține un "=").

UNIX/Linux - setare/consultare caracteristici

Obs: în practică se constată ca modificările asupra environmentului efectuate cu "`putenv()`" și "`unsetenv()`" sunt sesizate integral doar prin variabila "environ", nu și prin al 3-lea parametru al lui "main".

UNIX/Linux - fork()

Putem crea procese copil cu apelul:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

$\Longrightarrow$

procesul se duplică;

noul proces este copil al primului;

fiul va executa același program ca părintele, punctul de plecare fiind ieșirea din "`fork()`";

copilul moștenește o copie a datelor părintelui (cu valorile din momentul ieșirii din "`fork()`");

de asemenea moștenește toate caracteristicile părintelui (proprietarul real și cel efectiv, directorul curent, terminalul de control, environmentul, descriptorii de fișiere, prioritatea, sesiunea, grupul, etc.), cu excepția semnalelor în pending și blocajelor asupra fișierelor;

returnează: în părinte: PID-ul copilului (> 0);
în fiu: 0;
în caz de eșec, procesul inițial nu se duplică iar "`fork()`" returnează (în el) -1;

errno posibile: EAGAIN, ENOMEM.

UNIX/Linux - exec()

Putem înlocui un proces cu un alt proces, folosind apelurile numite generic "exec()":

```
#include <unistd.h>
int execve(const char *filename, char *const argv [],
           char *const envp[]);
int execl(const char *path, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
           char * const envp[]);
int execv(const char *path, char *const argv[]);
```

$\Longrightarrow$

cu aceste apeluri procesul curent se înlocuiește cu un altul (practic se înlocuiește imaginea procesului cu alta); toate se bazează de fapt pe "`execve()`";

noul proces ia locul celui vechi în sistem, moștenind același PID, același părinte, și o mare parte din celelalte caracteristici; nu moștenește acei descriptori setați să se închidă la "exec()", semnalele în pending și handlerele de tratare a semnalelor instalate de utilizator în primul proces; de asemenea, poate diferi proprietarul efectiv sau/și grupul acestuia dacă în i-nodul fișirului executat de noul proces este setat bitul SETUID/SETGID;

UNIX/Linux - exec()

Putem înlocui un proces cu un alt proces, folosind apelurile numite generic "exec()":

```
#include <unistd.h>
int execve(const char *filename, char *const argv [],
           char *const envp[]);
int execl(const char *path, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
           char * const envp[]);
int execv(const char *path, char *const argv[]);
```

$\Longrightarrow$

primul parametru este specificatorul fișierului executat de noul proces;

parametrii "`arg`" desemnează argumentele în linia de comandă pentru noul proces, în ordinea argv[0]=programul, ..., argv[argc-1], argv[argc]=NULL; în varianta cu "`...`" argumentele trebuie date în clar (funcții cu număr variabil de parametri (variadic functions)); în celelalte variante argumentele se pun într-un vector și "`arg`" este adresa primei componente; primul argument trebuie să fie chiar fișierul executat iar ultima componentă NULL trebuie să apară efectiv (în clar scris ca "`(char *) NULL`", respectiv pe ultima componentă a vectorului);

UNIX/Linux - exec()

Putem înlocui un proces cu un alt proces, folosind apelurile numite generic "exec()":

```
#include <unistd.h>
int execve(const char *filename, char *const argv [],
           char *const envp[]);
int execl(const char *path, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
           char * const envp[]);
int execv(const char *path, char *const argv[]);
```

$\Longrightarrow$

parametrii "envp" sunt asemănători, dar se referă la strigurile din environment-ul transmis (de forma "variabilă=valoare"), după ultimul string trebuind să apară o componentă NULL; în lipsa parametrului "envp" se va transmite environmentul indicat de variabila "environ" a procesului apelant;

UNIX/Linux - exec()

Putem înlocui un proces cu un alt proces, folosind apelurile numite generic "exec()":

```
#include <unistd.h>
int execve(const char *filename, char *const argv [],
           char *const envp[]);
int execl(const char *path, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
           char * const envp[]);
int execv(const char *path, char *const argv[]);
```

$\Longrightarrow$

la succes nu există return (procesul apelant nu mai există); la eșec returnează -1;

errno posibile: E2BIG, EACCES, EFAULT, EINVAL, EIO, EISDIR, ELIBBAD, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOEXEC, ENOMEM, ENOTDIR, EPERM, EPERM, ETXTBSY.

UNIX/Linux - system()

Putem executa din program o comandă shell cu apelul:

```
#include <stdlib.h>
int system(const char *command);
```

$\Longrightarrow$ se lansează un proces copil care execută linia de comandă shell specificată de "`command`" (cu ajutorul apelului "`/bin/sh -c`"); după ce comanda s-a terminat, se revine din apel;

comanda este executată cu semnalul `SIGCHLD` blocat și cu semnalele `SIGINT`, `SIGQUIT` ignorate.

la succes returnează return status-ul comenzii în formatul utilizat de "`wait()`" (a se vedea mai jos) (deci codul de retur poate fi obținut cu "`WEXITSTATUS()`"), la eșec returnează -1;

dacă "`command`" este NULL, returnează $\neq 0$ dacă shell-ul este disponibil și 0 dacă nu este.

UNIX/Linux - wait() și waitpid()

Putem aștepta terminarea unui copil și putem afla informații despre terminarea lui, inclusiv codul de retur, cu apelurile:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

$\Longrightarrow$

• apelul "`wait()`":

dacă procesul n-are copii, returnează -1 și setează errno = ECHILD;

dacă procesul are cel puțin un copil zombie, este ales unul dintre copii săi zombie, în "`*status`" (daca "`status`" nu e NULL) se pun informațiile referitoare la terminarea lui (return status-ul, format din codul de retur și detaliile privind modul de terminare) și se returnează PID-ul său; copilul respectiv este eliminat efectiv din sistem;

dacă procesul are copii dar nici unul nu e zombie, procesul adoarme până se întâmplă unul din următoarele evenimente:

- unul din copii devine zombie - atunci comportarea e ca mai sus;
- apelul "wait()" este întrerupt de un semnal (nu este posibil oricând - a se vedea cursul 6).

UNIX/Linux - wait() și waitpid()

Putem aștepta terminarea unui copil și putem afla informații despre terminarea lui, inclusiv codul de retur, cu apelurile:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

$\Longrightarrow$

- apelul "`waitpid()`":

testează terminarea unui copil / unor copii, în mod blocant sau nu; dacă apelul este neblocant, el se termină imediat (procesul nu adoarme);

`pid` = PID-ul copilului testat; dacă este -1, este vorba de un copil oarecare, ca în cazul apelului "`wait()`";

`status` = dacă este diferit de NULL, la această adresă se vor pune informațiile referitoare la terminarea fiului (return status-ul);

`options` = poate fi 0, sau poate fi o combinație de opțiuni (constante simbolice pentru care se poate include "`sys/wait.h`") legate prin "|"; câteva opțiuni:

`WNOHANG` = apelul este neblocant;

`WUNTRACED` = dacă fiul testat este suspendat, în "`*status`" se vor pune informații referitoare la această suspendare;

UNIX/Linux - wait() și waitpid()

Putem aștepta terminarea unui copil și putem afla informații despre terminarea lui, inclusiv codul de retur, cu apelurile:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

$\Longrightarrow$

comportamentul lui "`waitpid()`":

dacă s-a produs o eroare sau dacă procesul n-are copii, apelul se termină imediat și returnează -1, iar errno este setat corespunzator (în cazul absenței copiilor, errno devine ECHILD);

UNIX/Linux - wait() și waitpid()

Putem aștepta terminarea unui copil și putem afla informații despre terminarea lui, inclusiv codul de retur, cu apelurile:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

$\Longrightarrow$

comportamentul lui "`waitpid()`":

dacă copilul testat e zombie, apelul se termină imediat, în "`*status`" se pun informațiile referitoare la terminarea lui și se returnează PID-ul său; copilul este eliminat efectiv din sistem;

dacă parametrul "`pid`" a fost -1, atunci acest comentariu este valabil în cazul când există cel puțin un copil zombie, caz în care se alege unul dintre copii zombie și i se aplică lui cele menționate;

UNIX/Linux - wait() și waitpid()

Putem aștepta terminarea unui copil și putem afla informații despre terminarea lui, inclusiv codul de retur, cu apelurile:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

$\Longrightarrow$

comportamentul lui "`waitpid()`":

dacă copilul testat este activ (sau, în cazul când parametrul "`pid`" a fost -1, dacă există copii dar toți sunt activi), atunci:

- dacă am folosit `WNOHANG`, apelul se termină imediat și returnează 0, iar în "`*status`" nu se pune nimic;
- dacă nu am folosit `WNOHANG`, procesul apelant adoarme până se întră într-una din situațiile celelalte;

UNIX/Linux - wait() și waitpid()

Putem aștepta terminarea unui copil și putem afla informații despre terminarea lui, inclusiv codul de retur, cu apelurile:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

$\Longrightarrow$

comportamentul lui "`waitpid()`":

dacă copilul respectiv este suspendat (în cazul când parametrul "`pid`" a fost -1, comentariul este valabil atunci când există copii suspendați și toți ceilalți copil sunt activi, alegându-se un copil suspendat), atunci:

- dacă am folosit `WUNTRACED`, apelul se termină imediat, returnează PID-ul copilului respectiv, iar în "*status" se pun informații referitoare la suspendarea acestuia (ele se pot analiza ulterior cu macro-urile `WIFSTOPPED()` și `WSTOPSIG()` - a se vedea mai jos;

- dacă n-am folosit `WUNTRACED` dar am folosit `WNOHANG`, apelul se termină imediat, returnează 0, iar în "*status" nu se pune nimic;

- dacă n-am folosit nici `WUNTRACED` nici `WNOHANG` (deci "`options`" a fost 0), procesul adoarme pâna se intră într-una din situațiile celelalte;

UNIX/Linux - wait() și waitpid()

Putem aștepta terminarea unui copil și putem afla informații despre terminarea lui, inclusiv codul de retur, cu apelurile:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

$\Longrightarrow$

comportamentul lui "`waitpid()`":

dacă în timp ce procesul este adormit într-un "`waitpid()`" primește un semnal, comportamentul este ca la "wait";

UNIX/Linux - wait() și waitpid()

Putem aștepta terminarea unui copil și putem afla informații despre terminarea lui, inclusiv codul de retur, cu apelurile:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

$\Longrightarrow$

De fapt, apelul

```
wait(&status)
```

este echivalent cu

```
waitpid(-1, &status, 0)
```

UNIX/Linux - wait() și waitpid()

Putem aștepta terminarea unui copil și putem afla informații despre terminarea lui, inclusiv codul de retur, cu apelurile:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

$\Longrightarrow$

errno posibile (la ambele apeluri): ECHILD, EINTR, EINVAL.

UNIX/Linux - wait() și waitpid()

În continuare, pentru analizarea informației recuperate în "*status" de apelurile "`wait()`" sau "`waitpid()`" se pot folosi următoarele macro-uri (definite în "sys/wait.h"), aplicate întregului conținut în "`*status`":

`WIFEXITED()` ⇒ copilul s-a terminat normal ?

`WEXITSTATUS()` ⇒ furnizează codul de retur al fiului (are semnificație doar dacă acesta s-a terminat normal);

`WIFSIGNALED()` ⇒ copilul s-a terminat ca urmare a primirii unui semnal ?

`WTERMSIG()` ⇒ furnizează semnalul care a provocat terminarea copilului (are semnificație doar dacă copilul s-a terminat ca urmare a primirii unui semnal);

`WIFSTOPPED()` ⇒ copilul a fost suspendat ? (această informație este pusă de "`waitpid()`" în "`*status`" doar dacă s-a apelat cu `WUNTRACED`);

`WSTOPSIG()` ⇒ semnalul care a provocat suspendarea fiului (are sens doar dacă fiul a fost suspendat, iar aceste informații sunt puse de "`waitpid()`" în "`*status`" doar dacă s-a apelat cu `WUNTRACED`).

UNIX/Linux - exit()

Codul de retur (exit status) furnizat de un proces către procesul său părinte poate fi precizat ca valoare returnată de "main()" (dacă "main()" este recursiv, este vorba de valoarea returnată de apelul initial) sau ca parametru al apelului "exit()":

```
#include <stdlib.h>
void exit(int status);
```

$\Longrightarrow$ determină terminarea normală a procesului apelant, iar byte-ul low din "status" (adică "status & 0377") este furnizat ca cod de retur procesului părinte;

sunt apelate funcțiile înregistrate cu "`atexit()`" și "`on_exit()`" în ordinea inversă înregistrării, sunt evacuate bufferele stream-urilor deschise apoi acestea se închid, sunt șterse fișierele create cu "tmpfile()";

în general, "exit()" este modul cel mai "curat" de a termina forțat un proces;

funcția "exit()" nu returnează (deoarece procesul apelant nu mai există).

Putem specifica orice cod de retur dorim, uzanța este însă ca în caz de succes se returnăm 0 iar în caz de eșec o valoare $\neq 0$ care să desemneze și motivul eșecului; standardul C menționează două constante, EXIT_SUCCESS și EXIT_FAILURE, care pot fi transmise ca parametru lui "exit()", pentru a indica succesul, respectiv eșecul.

UNIX/Linux - exemplu shell

Dăm în continuare un exemplu de folosire a apelurilor de mai sus la simularea unui shell, care suportă comanda internă "exit" și poate lansa sub formă de comenzi externe fișiere executabile, eventual cu redirectarea stdout către un fișier:

UNIX/Linux - exemplu shell

```
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<string.h>

char ldc[256], c0[256],
     c1[256], c2[256],
     *a[256];

int main(){
  while(1){
    printf(">>");
    gets(ldc);
    c0[0]=c1[0]=c2[0]=0;
    sscanf(ldc,"%s%s%s",c0,c1,c2);
    if(!strcmp(c0,"exit")){
      return 0;
    }
```

```
else if(fork()){
  wait(NULL);
}else{
      if(!strcmp(c1,">")){
        int d;
        if((d=open(c2,
            O_WRONLY|O_CREAT|O_TRUNC,
            S_IRUSR|S_IWUSR))==-1)
          return 1;
        close(1); dup(d); close(d);
      }
      a[0]=c0; a[1]=NULL;
      execv(c0,a);
      return 1;
    }
  }
  return 0;
}
```

UNIX/Linux - exemplu shell

Shell-ul de mai sus se poate folosi sub forma:

```
>>./program
```

```
>>./program > f
```

```
>>exit
```

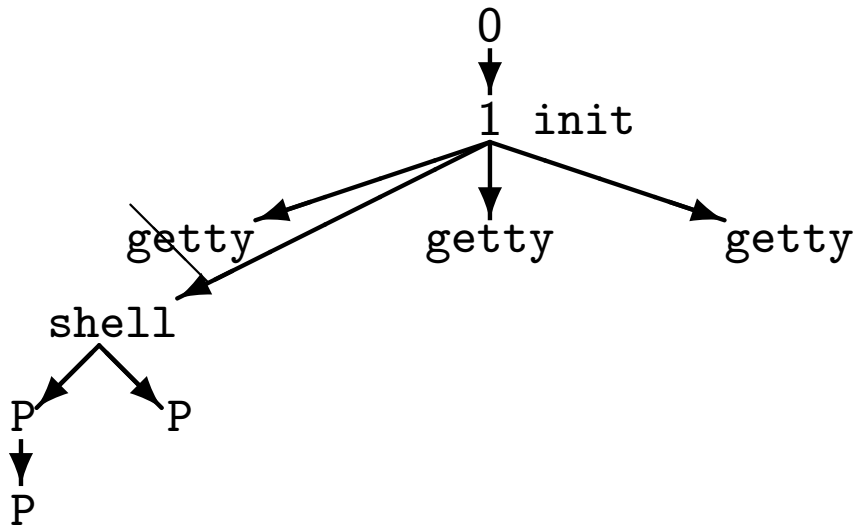
În al doilea caz, apelul "open()" deschide fișierul "`f`" în scriere returnând un descriptor "`d`" asociat acestuia, "`close(1)` închide stdout-ul shell-ului copil (1 devine astfel primul descriptor liber al acestuia), "dup(d)" alocă primul descriptor liber (adică 1=stdout) către același fișier ca descriptorul "`d`" (adică "`f`"), "`close(d)`" închide descriptorul "`d`" (nu mai este necesar, spre "`f`" deja duce stdout), iar "execv()" înlocuiește shell-ul copil cu un proces ce execută "`program`" și care va moșteni astfel stdout redirectat către fișierul "`f`".
Alte detalii - în cursul despre gestiunea fișierelor.

UNIX/Linux - arborescența proceselor

Într-o instanță UNIX/Linux procesele, legate prin relația părinte - copil, formează o arborescență.

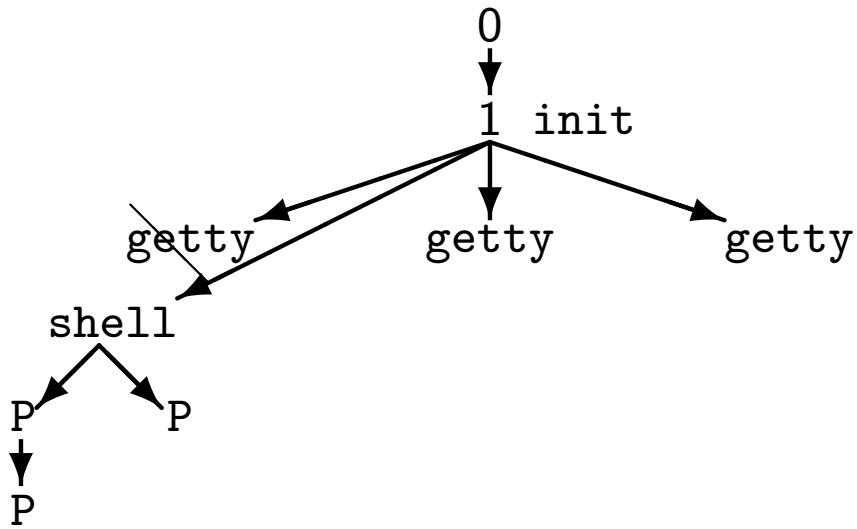
O parte din această arborescență este standard și o vom desena în cazul unui sistem UNIX clasic (în sistemele Linux moderne, mai ales dacă au instalată și interfața X-Windows, se mai pot interpune și alte procese):

UNIX/Linux - arborescența proceselor



Există un proces cu PID=0, care apare la boot-are; este un proces mai neobișnuit, de ex. nu i se pot transmite semnale; pe unele sisteme UNIX (System V) el se ocupă cu planificarea la execuție.

UNIX/Linux - arborescența proceselor



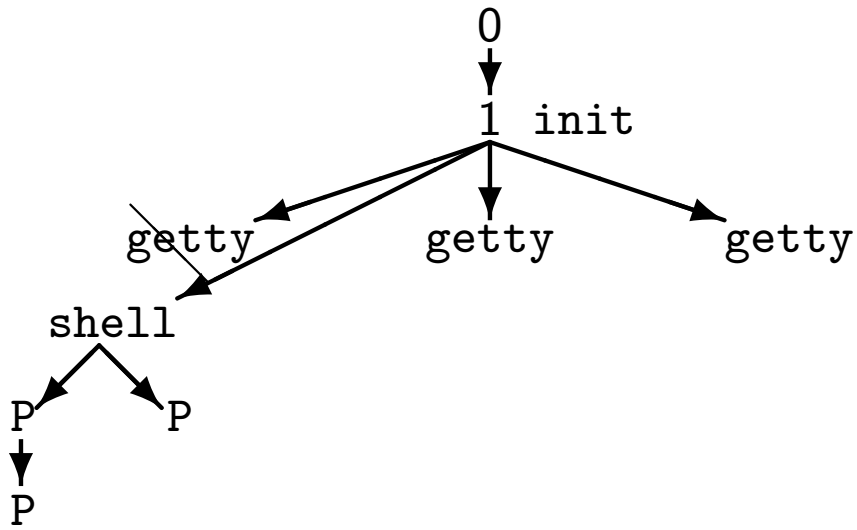
Procesul cu PID=0 are un copil cu PID=1, numit procesul "`init`"; acesta are mai multe sarcini, printre care:
1 - lansează la fiecare terminal text definit în sistem un proces copil "`getty`", care cere cont și parolă;

când cineva se loghează la un terminal, "`getty`"-ul respectiv se înlocuiește (prin "`exec()`") cu un proces shell;

shell-ului i se pot da în continuare comenzi interne și atunci le execută prin propriile instrucțiuni, sau externe și atunci lansează procese copil ce execută diverse programe care le îndeplinesc;

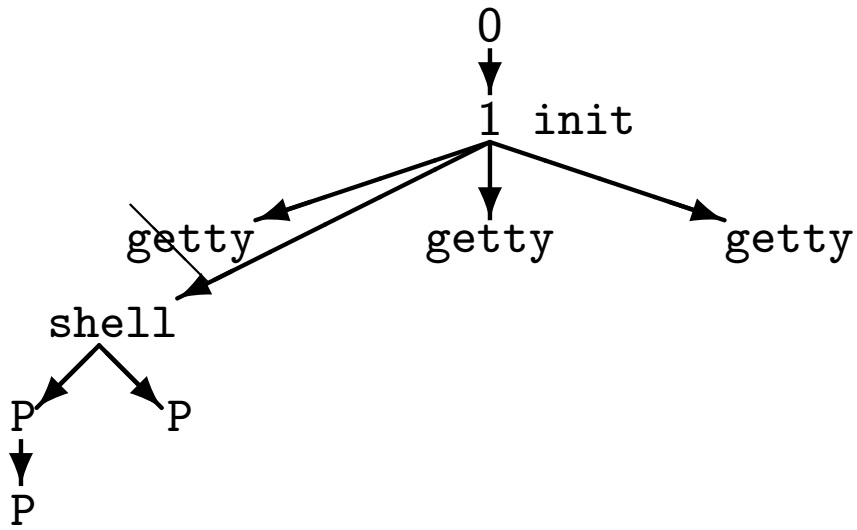
shell-ul poate lansa mai mulți copii în paralel, acești copii pot lansa alți copii, etc. - astfel arborescența crește;

UNIX/Linux - arborescența proceselor



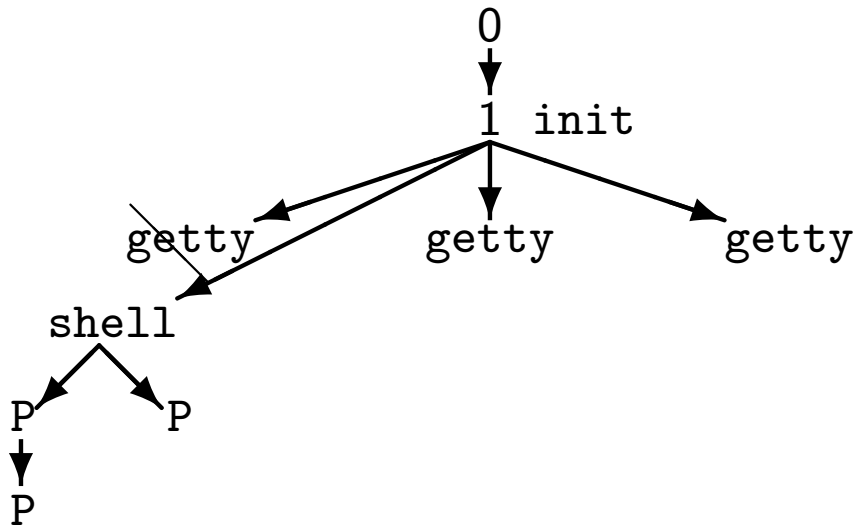
Procesul cu PID=0 are un copil cu PID=1, numit procesul "`init`"; acesta are mai multe sarcini, printre care:
2 - când parintele unui proces se termină, acesta (indiferent dacă este încă activ, blocat, zombie) devine copilul lui "`init`"; "`init`" face periodic apeluri de tip "`wait()`" sau "`waitpid()`" și îi elimină când aceștia ajung zombie.

UNIX/Linux - arborescența proceselor



Detaliem mecanismul logării (pașii esențiali):
- utilizatorul introduce username="ion" și password corespunzătoare;
- `getty` citește din fișierul "`/etc/passwd`" (care conține informații despre toți utilizatorii) informațiile despre "ion", inclusiv directorul său home și login shell-ul (specificatorul executabilului pe care "ion" îl dorește ca shell la logare);
- `getty` își schimbă proprietarul real și efectiv din "root" în "ion" (cu apeluri gen "`setuid()`", "`seteuid()`") și la fel grupurile acestora;
- `getty` își schimbă directorul curent în directorul home al lui "ion" (cu apelul "chdir()");
- se înlocuiește prin "`exec()`" cu un proces ce execută login shell-ul lui "ion".

UNIX/Linux - arborescența proceselor



Astfel, utilizatorul primește un shell, care:
- moștenește de la "getty" terminalul de control, de aceea va da comenzi de la același terminal unde s-a logat;
- moștenește de la "getty" proprietarii, dar acesta și i-a schimbat în prealabil în "ion" - deci proprietarii shell-ului sunt "ion";
- moștenește de la "getty" directorul curent, dar el și l-a schimbat în prealabil în directorul home al lui "ion" - astfel, primul director curent primit la logare este directorul home al user-ului;
- rulează login shell-ul lui "ion".

UNIX/Linux - setjmp/longjmp

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

$\Longrightarrow$

"`setjmp()`" salvează contextul de execuție și stiva (în principiu registrul contor de program și regiștrii de stivă) în "`env`"; tipul "`jmp_buf`" este definit ca un tip vector, deci "`env`" este de fapt un pointer către zona în care se vor salva aceste informații (parametru de ieșire); de asemenea, rezultă că entitățile declarate de tip "`jmp_buf`" sunt nume de vectori, deci nu sunt lvalues (nu pot apărea în stânga atribuirii);

"`longjmp()`" restaurează contextul de execuție și stivă memorat în "`env`"; aceasta face ca execuția sa continue de la ieșirea din apelul "`setjmp()`" care a salvat aceste informații; notăm că o entitate "`jmp_buf`" memorează doar poziția pe stivă, nu și conținutul ei, deci dacă între timp s-au apelat și alte funcții cu aceeași zonă curentă a stivei, ele poate au suprascris conținutul din locul respectiv, a.î. la revenirea la ieșirea din "`setjmp()`" execuția sa nu se mai facă corect (inclusiv pot apărea erori de execuție fatale);

UNIX/Linux - setjmp/longjmp

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

$\Longrightarrow$

constatăm că într-un apel "`setjmp()`" se poate intra/reveni (return) de mai multe ori: o dată prin apelarea normală și mai multe ori în urma unui "`longjmp()`"; când se revine în urma apelului normal, "`setjmp()`" returnează 0; când se revine în urma saltului cu un "`longjmp()`", "`setjmp()`" returnează valoarea parametrului "`val`" al acelui "`longjmp()`"; "`val`" nu poate fi 0; dacă îl dăm 0, se va considera 1;

"`longjmp()`" nu returnează niciodată (dintr-un apel "`longjmp()`" se sare direct într-un apel anterior al lui "`setjmp()`" prin restaurarea contextului).

UNIX/Linux - setjmp/longjmp

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savesigs);
void siglongjmp(sigjmp_buf env, int val);
```

$\Longrightarrow$

sunt similare lui "`setjmp()`"/"`longjmp()`", cu următoarele diferențe:
- o entitate de tip "`sigjmp_buf`" poate stoca nu numai un context de execuție și stivă, ci și o mulțime de semnale;
- dacă "`savesigs`" este $\neq 0$, "`sigsetjmp()`" salvează în "`env`" și masca de semnale blocate a procesului (a se vedea cursul 6);
- "`siglongjmp()`" restaurează ce a fost stocat în "`env`"; dacă apelul "`sigsetjmp()`" a stocat acolo și masca de semnale blocate, o va restaura și pe aceasta, altfel va restaura doar contextul de execuție și stivă.

Existența acestor apeluri este datorată faptului că standardul POSIX nu specifică dacă "`setjmp()`"/"`longjmp()`" salvează/restaurează și masca de semnale blocate (comportamentul poate diferi de la o implementare de UNIX/Linux la alta); dacă vrem să fim siguri că se salvează/restaurează și masca de semnale blocate, vom folosi "`sigsetjmp()`"/"`siglongjmp()`".

UNIX/Linux - setjmp/longjmp

Apelurile "`setjmp()`"/"`longjmp()`" (și generalizările lor "`sigsetjmp()`"/"`siglongjmp()`") se pot folosi pentru a implementa o formă de **corutine** - componente de program ce generalizează subrutinele (proceduri, funcții) în sensul că permit existența mai multor puncte de intrare și ieșire dintr-un același apel, pentru întreruperea și reluarea execuției apelului în anumite locuri (iar la fiecare revenire se regăsesc datele locale apelului, cu aceleași valori); într-un apel de subrutină se intră/iese doar o dată, prin mecanismul de apel/revenire.

Practic, plantând în diverse locuri apeluri "`setjmp()`"/"`longjmp()`", putem scrie funcții a.î. din interiorul apelului uneia să se sară în interiorul apelului celeilalte și invers, la fiecare revenire regăsindu-se variabilele locale automatice ale apelului cu aceleași valori. Funcțiile respective trebuie să fie însă apelate normal în prealabil, pentru a-și crea cadrul de apel pe stivă, și trebuie să avem grijă ca aceste cadre să nu fie suprascrise accidental - de ex. să începem salturile înainte ca din funcțiile respective să facem return, sau să avem grijă să nu fie apelate între timp alte funcții cu zonele respective din stivă ca zone curente (ca să-și creeze acolo cadrul de apel).

UNIX/Linux - thread-uri cu setjmp/longjmp

Exemplu: prezentăm o modalitate de implementare manuală (făcută explicit de programator) a thread-urilor în spațiul utilizator, folosind "setjmp()/longjmp()" (alternativ, se pot folosi "sigsetjmp()/siglongjmp()").

În acest exemplu, thread-urile cedează voluntar procesorul prin apeluri "thread_yield()"; în cursul 6 va fi prezentată o variantă a acestei implementări în care thread-urile sunt comutate automat la anumite intervale de timp, folosind semnale SIGALRM și apeluri "alarm()".

UNIX/Linux - thread-uri cu setjmp/longjmp

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<stdarg.h>
#include<setjmp.h>

/* executiv */

#define MAXTHREADS 100

struct thread_table_entry{
  jmp_buf j[2]; int s;
  void (*f)();
} thread_table[MAXTHREADS];
int nthreads, tcurrent;
jmp_buf jscheduler, jmain;

unsigned int thread_set[MAXTHREADS], nthreadset;
```

UNIX/Linux - thread-uri cu setjmp/longjmp

```
int thread_create(void (*pf)()){
  int i,k;
  if(nthreadset==nthreads)return -1;
  for(i=0;i<nthreads;++i){
      for(k=0;k<nthreadset;++k)if(thread_set[k]==i)break;
      if(k==nthreadset)break;
  }
  thread_table[i].f=pf;
  thread_table[i].s=0;
  thread_set[nthreadset]=i; ++nthreadset;
  return i;
}
void thread_yield(){
  if(tcurrent==-1?!setjmp(jmain):!setjmp(thread_table[tcurrent].j[1]))
    longjmp(jscheduler,1);
}
void thread_terminate(){
  int k;
  if(nthreadset==0)return;
  for(k=0;k<nthreadset;++k)if(thread_set[k]==tcurrent)break;
  --nthreadset; thread_set[k]=thread_set[nthreadset];
  longjmp(jscheduler,1);
}
```

UNIX/Linux - thread-uri cu setjmp/longjmp

```
void schedule_threads(){
  if(nthreadset==0){tcurrent=-1; longjmp(jmain,1);}
  tcurrent=thread_set[rand()%nthreadset];
  if(thread_table[tcurrent].s==0)
    {thread_table[tcurrent].s=1; longjmp(thread_table[tcurrent].j[0],1);}
  else longjmp(thread_table[tcurrent].j[1],1);
}
void initialize_threads(unsigned int nt, unsigned int dim){
 static int dimaux=0;
 unsigned char buf[1024];
 if(dimaux==0){
   if(nt>MAXTHREADS || dim==0)exit(1);
   srand(time(NULL));
   nthreads=nt; dimaux=dim; nthreadset=0; tcurrent=-1;
 }
 if(dim>0)initialize_threads(nt,dim-1);
 else if(nt>0)initialize_threads(nt-1,dimaux);
 if(dim==0)
  if(nt==nthreads){dimaux=0; if(setjmp(jscheduler)) schedule_threads();}
  else if(setjmp(thread_table[nt].j[0])){
            (*thread_table[nt].f)();
          thread_terminate();
        }
}
```

UNIX/Linux - thread-uri cu setjmp/longjmp

```
/* aplicatie */

int vglobal;
void f2(){
  int vf2=20,i;
  for(i=0;i<4;++i){
     thread_yield();
     printf("f2: vf2=%d, vglobal=%d\n",++vf2,++vglobal);
  }
}
void f1(){
  int vf1=10; int j;
  thread_create(f2);
  for(j=0;j<2;++j){
     thread_yield();
     printf("f1: vf1=%d, vglobal=%d\n",++vf1,++vglobal);
  }
}
int main(){
  initialize_threads(2,1);
  vglobal=0;
  thread_create(f1); thread_yield();
  return 0;
}
```

UNIX/Linux - thread-uri cu setjmp/longjmp

Comentarii:

1) Fiecare funcție "fi()" efectuează un ciclu în care dă variabilei sale locale automatice "vfi" valorile 11, 12 (în cazul "f1()"), resp. 21, 22, 23, 24 (în cazul "f2()") și incremenetază variabila globală "vglobal", care va primi per total valorile 1, 2, 3, 4, 5, 6; la fiecare iterație variabila locală și cea globală sunt afișate; iterațiile din cele două funcții sunt execuate intercalat (multithread), iar la rulare programul poate afișa de exemplu:

```
f1: vf1=11, vglobal=1
f2: vf2=21, vglobal=2
f2: vf2=22, vglobal=3
f2: vf2=23, vglobal=4
f1: vf1=12, vglobal=5
f2: vf2=24, vglobal=6
```

("f1" și "f2" pot alterna în diverse feluri și observăm că la revenirea în contextul fiecăreia dintre cele două funcții variabila ei locală "vfi" și-a continuat incrementarea de unde a rămas, în timp ce variabila globală "vglobal" a fost incrementată în continuare).

UNIX/Linux - thread-uri cu setjmp/longjmp

2) Executivul constă din:
• o tabelă de thread-uri "`thread_table`" cu "`nthreads`" elemente, indicele elementului (thread-ului) curent fiind "`tcurrent`";
o intrare în tabelă contine adresa "`f`" a unei funcţii executate de thread şi elementele de tip "`jmp_buf`" "`j[0]`" şi "`j[1]`", care reţin punctul (contextul) iniţial din care se execută thread-ul, respectiv punctul în care thread-ul este întrerupt, pentru a se comuta pe alt thread; "`s`" reţine care din cele două elemente este cel folosit curent pentru thread-ul respectiv;
• elementele de tip "`jmp_buf`" "`jscheduler`" şi "`jmain`", care reţin punctul din care este reluat planificatorul, respectiv din care se pleacă/revine din "`main()`" cand se executa thread-urile;
• o mulţime "`thread_set`", având "`nthreadset`" elemente, care conţine indicii intrărilor ocupate din tabela de thread-uri (este folosită la planificare);

UNIX/Linux - thread-uri cu setjmp/longjmp

• functia "`thread_create()`", care înregistrează un nou thread; în acest scop găsește o intrare liberă în tabela de thread-uri "`thread_table`" (i.e. care nu este în mulțimea "`thread_set`"), o inițializează cu funcția dată ca parametru, și o adaugă mulțimii "`thread_set`"; returnează indicele thread-ului, ca un identificator unic al său ce poate fi exploatat într-o extindere a acestei implementări, pentru a manevra thread-ul respectiv din funcțiile utilizator;
notăm că "`thread_create()`" nu lansează efectiv thread-ul în execuție ci doar informează planificatorul despre existența lui, iar planificatorul îl va lua în considerație ca o variantă posibilă de continuare atunci când un alt thread îi va transfera controlul cu "`thread_yield()`";
• funcția "`thread_yield()`", prin care thread-ul curent (sau "`main()`") cedează procesorul, transferând controlul planificatorului pentru a alege un (alt sau același) thread; în acest scop memorează contextul thread-ului curent (indicat de "`tcurrent`") in elementul său "`j[1]`" (sau "`jmain`" dacă este vorba de "`main()`"), apoi transferă controlul la începutul planificatorului (reținut în "`jscheduler`");
• funcția "`thread_terminate()`", care elimină thread-ul curent, eliminându-l din mulțimea "`thread_set`" și transferând controlul la începutul planificatorului (reținut în "`jscheduler`"), pentru a comuta pe un thread din cele rămase;

UNIX/Linux - thread-uri cu setjmp/longjmp

• funcția "`schedule_threads()`" este planificatorul thread-urilor; el alege noul thread care va continua (și va fi indicat de "`tcurrent`") în mod aleator din mulțimea thread-urilor existente "`thread_set`" (în loc de alegerea aleatoare se poate implementa orice algoritm de planificare descris în cursul 7), apoi ii va transfera controlul restaurând contextul memorat în elementul său "`j[0]`" sau "`j[1]`" (după cum thread-ul se afla la început sau nu), consultând/modificând corespunzător și "`s`"-ul său; dacă nu mai erau thread-uri ("`nthreadset==0`") se revine la "`main()`" (restaurând contextul memorat în "`jmain`");

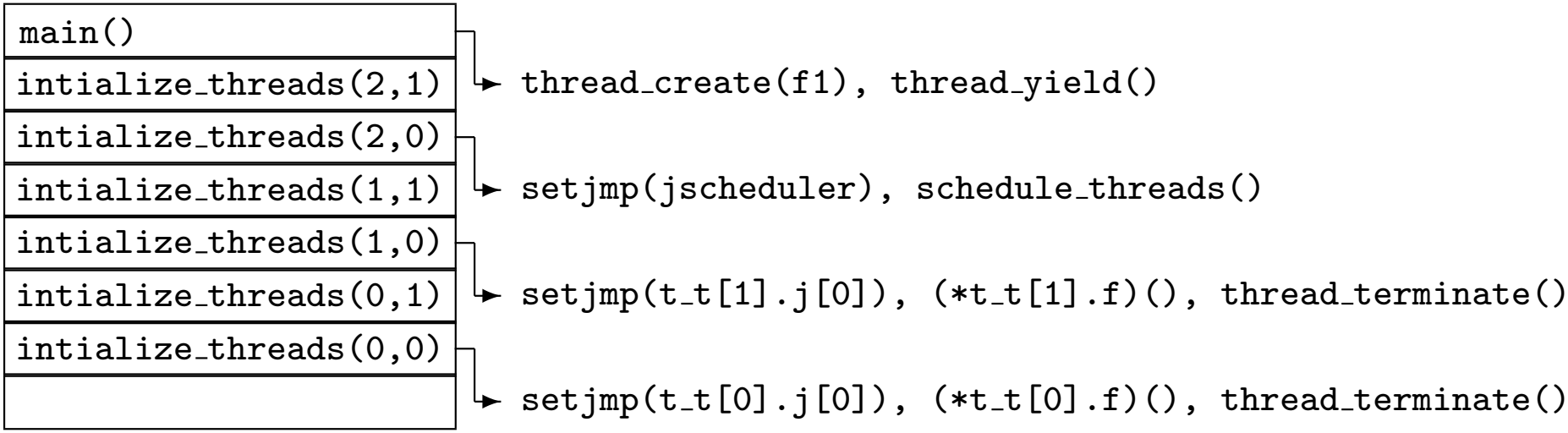
• funcția "`initialize_threads()`" inițializează executivul pentru a suporta maxim "`nt`" thread-uri, alocând pentru fiecare o stivă proprie (parte a stivei procesului) de aprox. "`dim * sizeof(buf)`" octeți; în acest scop se apelează recursiv de "`dim * nt`" ori (variabila locală statică "`dimaux`" reține valoarea inițială a lui "`dim`", deoarece aceasta se va modifica pe parcursul recursiei), iar la revenirea din apeluri, din "`dim`" în "`dim`" ori, consemnează pe stiva procesului pozițiile thread-urilor și planificatorului (în elementele "`j[0]`" și respectiv "`jscheduler`"); notăm că această consemnare trebuie să se facă la ieșirea din apeluri și nu la intrarea în ele, deoarece aceste apeluri ar putea suprascrie informațiile consemnate pe stivă; valoarea inițială a lui "`dim`" dă dimensiunea stivei unui thread și cu cât e mai mare, cu atât thread-ul poate lansa (obișnuit) apeluri îmbricate mai multe/mari; nu se oferă un mecanism automat de protecție la depășirea stivelor, trebuie să aibe grijă programatorul.

UNIX/Linux - thread-uri cu setjmp/longjmp

3) "`main()`" se execută secvențial (nu poate fi paralel cu alte thread-uri); odată apelat "`thread_yield()`" din "`main()`", "`main()`" se întrerupe (reținându-se contextul în "`jmain`") și se inițiază modul de rulare multithread, în care se excută deocamdata o singură funcție (un singur thread); în continuare, funcțiile apelate obișnuit se execută în același thread cu apelantul, cele apelate via "`thread_create()`" se execută în thread-uri paralele cu apelantul; când nu mai sunt thread-uri se revine la punctul în care s-a întrerupt "`main()`" și se continuă rularea secvențială a acestuia; din "`main()`" se pot apela și funcții în mod obișnuit, dar acestea nu vor forma thread-uri paralele cu "`main()`".

4) În exemplul nostru, "`main()`" inițializează executivul pentru a suporta maxim 2 thread-uri, fiecare având o stivă proprie de aprox. 1024 octeți, apoi crează un thread ce execută "`f1()`" și care la rândul său crează un thread ce execută (în paralel cu el) "`f2()`"; stiva procesului va arăta astfel:

UNIX/Linux - thread-uri cu setjmp/longjmp



fiecare dreptunghi este o zonă de aprox. 1024 ("`sizeof(buf)`") octeți, rezervată pe stiva procesului de un apel "`intialize_threads()`"; săgețile arată că din contextul apelurilor "`intialize_threads()`" cu "`dim==0`" se vor lansa funcții ca "`f1()`", "`f2()`", "`thread_terminate()`", "`setjmp()`", etc., dar aceste funcții vor consuma spațiu pe stivă începând din zona rezervată de apelurile cu "`dim==1`" (putând ajunge și suprascrie, dacă consumă prea mult spațiu, zonele următoarelor apeluri cu "`dim==0`", provocând un dezastru); de aceea "`dim`" trebuie să fie ≥ 1 și cu cât este mai mare cu atât este mai bine;

atenție că și "`setjmp()`"/"`longjmp()`" sunt funcții (nu macro-uri), deci consumă loc pe stivă !

UNIX/Linux - pthread

TODO: pthread.