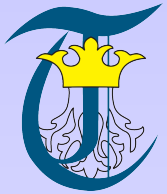


Universitatea *Transilvania* Brașov
Facultatea de Inginerie Electrică și Știința Calculatoarelor
Departamentul de Electronică și Calculatoare

Sisteme de operare

– curs 6 –

2012.04.03

ș.l. dr. ing. Kertész Csaba-Zoltán

2.5. Planificarea proceselor

- componenta SO care determină care dintre procesele din sistem va deveni activ:

scheduler

- schedulerul implementează un algoritm de planificare (scheduling algorithm)
- acest algoritm permite alegerea uneia din procesele aflate în starea ready și marcarea acesteia ca fiind activ
- după alegere schedulerul realizează schimbarea contextului între procesul care a fost activ și cel care este marcat să devină activ

Criteriile planificării

- **fairness**: fiecare proces să ocupe procesorul în mod cinstit
- **eficient**: să mențină utilizarea CPU cât mai aproape de 100%
- **timp de răspuns**: minim
- **turnaround**: minimizare timpului de așteptare
- **throughput**: maximizarea numărului de joburi pe oră

Problemele planificatorului

- criteriile sunt contradictorii
  - de ex. pentru a minimiza timpul de răspuns ar trebui rulate cât mai puține procese
- o complicație prezintă faptul că fiecare proces este **unic** și **impredictibil**
  - unele procese petrec foarte mult timp așteptând operații IO, altele pot rula ore întregi fără așteptare
  - la activarea unui proces schedulerul nu poate să știe cât timp va rula procesul înainte de a se bloca

Strategiile planificatorului

- strategii de planificare **preemtive:** permit suspendarea temporară a proceselor care rulează
  - la fiecare întrerupere de ceas se rulează schedulerul care decide dacă procesul curent continuă să ruleze sau nu, dacă nu procesul va fi suspendat și controlul dat altui proces
- strategii de planificare **non-preemptive:** procesele rulează până când își termină execuția sau se vor bloca în așteptarea unei operații IO
  - la apelul sistem de terminare sau cel blocant se rulează schedulerul

2.5.1. First Come First Served Scheduling

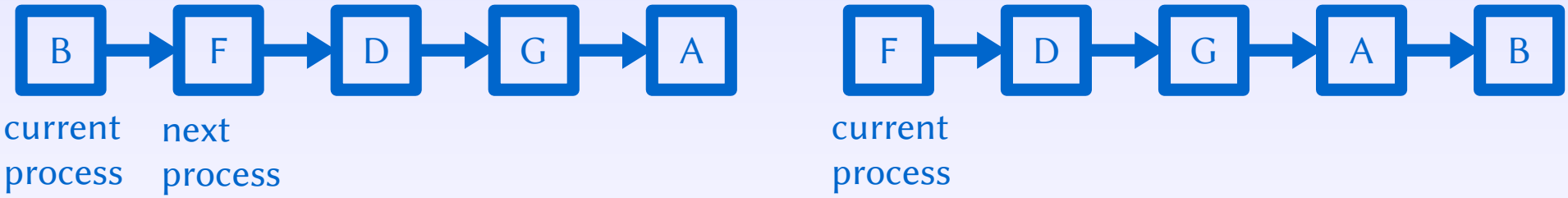
- planificarea cea mai simplă
- CPU va fi alocat proceselor în ordinea în care acestea se pornesc
- procesele continuă până la terminare sau blocare, moment în care următorul proces din listă va fi planificat
- se poate implementa foarte ușor cu un FIFO
- este o planificare non-preemptivă

2.5.2. Round Robin Scheduling

- este unul din cele mai vechi și simpli algoritmi de planificare preemptive
- fiecărei proces i se atribuie un interval de timp numit **cuantă**, în care procesul se poate executa
- dacă procesul nu-și termină execuția până la expirarea cuantei de timp alocate, el va fi întrerupt iar CPU este alocat altui proces
- dacă procesul termină înainte de expirarea cuantei, se va planifica alt proces fără a se aștepta expirarea cuantei

Round Robin

- algoritmul este ușor de implementat
  - planificatorul va menține o listă a proceselor ce pot fi rulate (ready)
  - la expirarea cuantei, procesul activ va fi trecut la sfârșitul listei, iar următorul proces din listă va obține CPU



- mean turnaround time
  - de ex. procesele A,B,C,D cu timpi de execuție 4,2,5,3, mtt=10.75; iar pentru 6,1,4,2, mtt=8.25

Alegerea cuantei

- o problemă importantă în proiectare este alegerea cuantei
- comutarea proceselor necesită un anumit interval de timp pentru salvarea și încărcarea registrelor, actualizarea tabelelor și listelor, etc.
- de ex. cuantă 20ms, timp de comutare 5ms => overhead de 20%
  - pentru a mări eficiența trebuie micșorat overheadul, de ex. cuantă 495ms, overhead 1%
  - dar cu cuanta mare timpul de răspuns crește: de ex. 10 utilizatori apasă Enter, ultimul va trebui să aștepte 5s

Alegerea cuantei

- stabilirea unei cuante mici va conduce la comutări multe scăzând eficiența CPU
- stabilirea unei cuante mari cauzează un timp de răspuns mare pentru utilizator
- valorile uzuale pentru cuante sunt 1-100ms
- informații despre eficiență

  `/proc/cpuinfo`

  `/proc/stat`

  `/proc/schedstat`

2.5.3. Priority scheduling

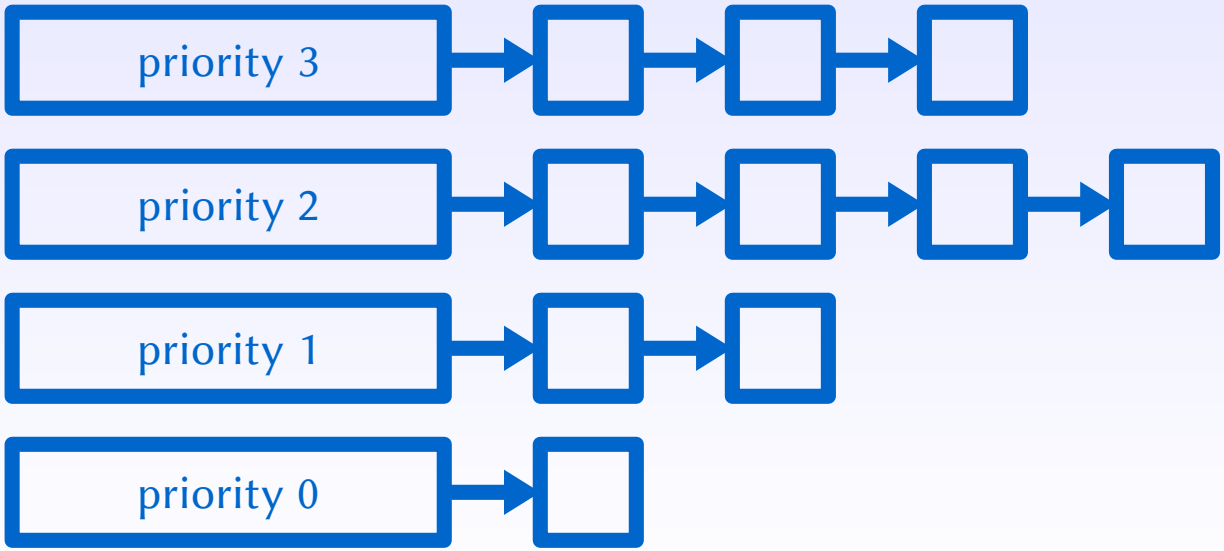
- în cazul Round Robin, procesele aveau aceeași prioritate
- uneori însă va trebui să ținem cont de prioritățile în rezolvarea unor probleme
- fiecărei proces i se alocă o prioritate
- la un moment dat va fi rulat procesul cel mai prioritar
- pentru a previne rularea indefinită a proceselor prioritare schedulerul poate descrește prioritatea procesului activ la fiecare întrerupere

Atribuirea priorităților

- atribuire statică
  - prioritate constantă definită pentru un proces sau pentru utilizatorul care pornește procesul
- atribuire dinamică
  - modificare a priorității în timpul rulării
  - de ex. comanda nice permite unui utilizator să-și descrească prioritatea pentru a fi drăguț cu ceilalți :)
  - SO poate acorda prioritate mai mare unui proces care așteaptă foarte mult pentru IO și utilizează foarte puțin CPU

Gruparea priorităților

- procesele pot fi grupate în clase de priorități
- se va utiliza priority scheduling între clase cu diferite priorități
- în cadrul fiecărei clase se va utiliza round-robin (între procese cu același prioritate)



Priority inversion

- o problemă mare la algoritmii bazate pe priorități este posibilitate apariției fenomenului de inversare a priorităților
- dacă un proces prioritar așteaptă după un proces mai puțin prioritar (pentru sincronizarea secțiunilor critice), atunci procesele cu priorități intermediare vor bloca procesul prioritar astfel devenind aparent mai prioritari
  - soluția poate fi schimbarea temporare a priorității procesului la prioritatea celui care este blocat în așteptarea sincronizării

2.5.4. Multiple queues

- se folosesc clase de priorități
- procesele din clasa cea mai prioritară vor avea o cuantă de timp, cele din următoarea clasă 2 cuante, următoarea 4 cuante, ş.a.m.d.
- dacă un proces și-a utilizat toate cuantele dintr-o clasă, va fi trecut într-o clasă inferioară
- de ex. un proces necesită 100 cuante:
  - rulează 1 cuantă, întrerupt, după care 2, 4, 8, 16, 32, 37 => 7 comutări în loc de 100
- un proces lung se va rula din ce în ce mai rar

2.5.5. Shortest job first

- algoritmii de până acum au fost proiectați pentru sisteme interactive
- algoritmul SJF este proiectat pentru sisteme în care timpul de rulare este cunoscut în avans
- de exemplu patru procese de aceeași prioritate:

Procese:	A	B	C	D
Timp execuţie:	8	4	4	4

- $mtt_{FCFS} = 14$, $mtt_{SJF} = 11$
- primul job rulat contribuie cel mai mult la mtt

Shortest job first

- SJF conduce la un timp de răspuns mediu minim
- ar fi bine de utilizat și la procese interactive
  - problema: cum se estimează timpul de rulare pentru astfel de procese
- **ageing**:
  - timpul inițial estimat = T_0, după rulare timpul măsurat = T_1
  - următoarea estimare = $aT_0 + (1 - a)T_1$
  - $T_0, T_0/2 + T_1/2, T_0/4 + T_1/4 + T_2/2, T_0/8 + T_1/8 + T_2/4 + T_3/2$

Shortest job first

- SJF este optim doar dacă toate job-urile sunt disponibile
- de exemplu:

Procese:	A	B	C	D	E
Timp execuţie:	2	4	1	1	1

- dacă inițial numai A și B pot fi planificate, restul fiind blocate: mtt = 4.6
- dacă toate procesele pot fi planificate: mtt = 4.4

2.5.6. Planificare garantată

- o abordare diferită de planificare:
  - se face promisiuni utilizatorului legat de mărimea timpului CPU atribuit
  - de exemplu dacă sunt n utilizatori logați în sistem, fiecare va primi 1/n din timpul CPU
- sistemul calculează mereu cât timp CPU a utilizat fiecare utilizator și împarte cu n, apoi calculează raportul între timpul utilizat și timpul de la ultimul calcul
- rulează tot timpul procesul cu cel mai mic raport

2.5.7. Politici și mecanisme

- nici unul din algoritmii prezentați nu acceptau informații de la procesele utilizator, pe care să le folosească în luarea deciziilor
  - schedulerul nu va lua întotdeauna decizia cea mai bună
- soluția este separarea mecanismului planificării de politica planificării
- mecanismul este realizat de kernel, dar politica este stabilită de procesul utilizator

Politici și mecanisme

- algoritmul de planificare este parametrizat într-un anume mod astfel încât parametrii să pot fi setați de procesele utilizator
- de exemplu:
  - kernelul folosește priority scheduling și asigură un apel sistem prin care un proces poate seta prioritățile proceselor fii, astfel procesul părinte va controla planificarea proceselor fii

2.5.8. Two-level scheduling

- până acum am presupus că procesele ready se află în memorie
- dacă însă nu dispunem de suficientă memorie, unele din procese ready va trebui plasate pe disc
- comutarea proceselor de pe disc va lua un timp mult mai mare decât celor din memorie
- vom utiliza un scheduler cu 2 nivele

Two-level scheduling

- inițial o submulțime a proceselor ready vor fi încărcate în memorie
- schedulerul va planifica numai aceste procese pentru o perioadă de timp
- periodic un scheduler de nivel înalt este utilizat pentru a elimina procesele din memorie (care sunt de mult timp acolo) și a încărca procese noi de pe disc
- apoi se folosește schedulerul de nivel jos pentru planificare între aceste procese noi

Criteriile folosite de schedulerul de nivel înalt

- cât de mult a stat un proces în memorie sau pe disc
- cât timp CPU a utilizat procesul
- cât de mare este procesul
- care este prioritatea procesului