Planificator de procese

Structuri de date (2016-2017)
Tema 1

13.03.2017
Deadline: 2.04.2017

Responsabil temă: Oprea Bogdan <oprea.bg@gmail.com>
Ultima modificare: 16.03.2017

1 Introducere

Bogdan este student în anul 1 la ACS și tocmai a terminat cu succes cursul de USO. Până acum a fost mereu fascinat de modul în care calculatorul său putea face atât de multe lucruri în același timp. Însă după ce a aflat de procese și-a dat seama că a trăit în minciună. Calculatorul lui nu face totul deodată, ci programează procese pe care le rulează pentru scurte cuante de timp și îl păcălesc astfel că ele au loc simultan. Pentru că vrea să înțeleagă mai bine mecanismul ca să nu se mai lase păcălit și altă dată, Bogdan vrea să afle cum exact decide procesorul ce proces urmează să se execute. Dar pentru că nu a fost pasionat de programare, vă roagă pe voi să duceți la bun sfârșit acest task.

2 Algoritmi de planificare

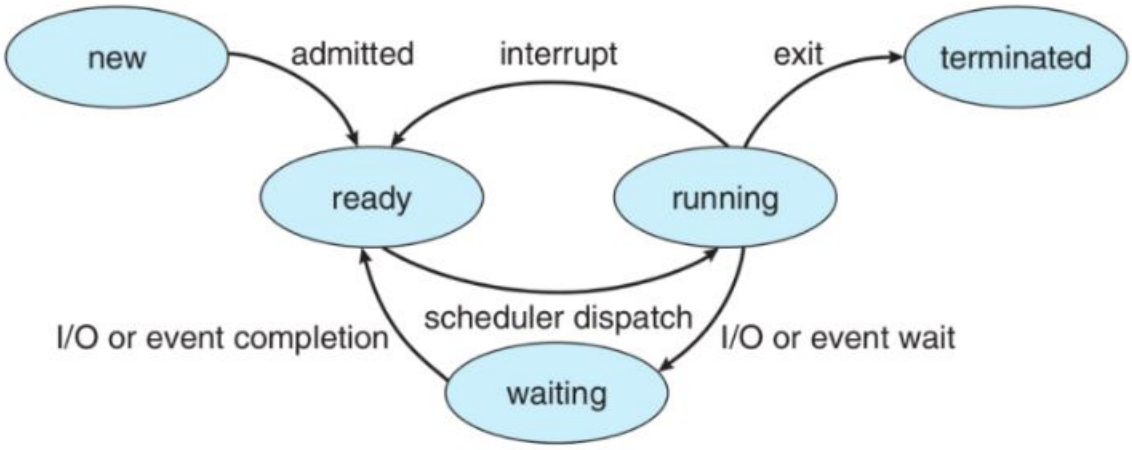
Atunci când se alege tipul de planificare pe procesor se ține cont de mai multe criterii: **fairness** (să ajungă toate procesele pe procesor), **gradul de ocupare** (procesorul să nu aibă timpi morți), **productivitate** (să fie terminate cât mai multe procese), **timp de așteptare** (cât mai mic).

O parte din aceste criterii sunt contradictorii și nu pot fi îndeplinite simultan. Spre exemplu, nu putem avea o productivitate mare dacă timpul de așteptare este mic. În funcție de ce ne interesează, avem următoarele tipuri de planificări:

- planificarea sistemelor batch, care pune accentul pe productivitate
- planificarea sistemelor interactive, care pune accentul pe fairness
- planificarea proceselor real-time (nu face obiectul acestei teme)

Un proces se poate afla în una din următoarele stări:

- **new**
- **ready** (procesul poate intra pe procesor. Aici aplicăm algoritmii de planificare)
- **running** (un singur proces poate rula pe un procesor la un moment dat)
- **waiting** (o listă a proceselor care așteaptă evenimente sau acțiuni I/O)
- **terminated**



Tranzițiile dintr-o stare în alta au loc astfel:

- WAITING → READY are loc când, pentru procesul în cauză, a avut loc evenimentul așteptat.
- RUNNING → TERMINATED are loc atunci când procesul s-a terminat.
- RUNNING → WAITING are loc atunci când procesul are nevoie de input de la utilizator sau de la un alt proces.
- READY → RUNNING este dată de planificator.
- RUNNING → READY poate sau nu să aibă loc, funcție de planificator.

2.1 Batch: First Come First Served (FCFS)

În cadrul acestui algoritm, procesele vor pleca în RUNNING în ordinea în care au sosit în READY. Ele vor sta în RUNNING până când se termină execuția lor, moment în care vor pleca în starea TERMINATED și vor face loc altui proces în RUNNING.

2.2 Batch: Shortest Job First (SJF)

Pentru acest algoritm este necesar să știm de la început durata exactă necesară pentru un proces să se finalizeze. Atunci când un proces trebuie să treacă din READY în RUNNING, se va alege acel proces care va dura cel mai puțin să se termine.

ATENȚIE: algoritmul nu este unul preemptiv. Asta înseamnă că un proces nu îl poate scoate pe altul din starea RUNNING. Dacă un proces a fost ales să treacă în RUNNING, el va rămâne acolo până când se va termina, indiferent dacă între timp în coada READY a sosit un alt proces care ar dura mai puțin să se termine.

2.3 Interactiv: Round Robin

Acest algoritm seamănă cu FCFS, doar că este preemptiv. Asta înseamnă că un proces poate fi întrerupt din starea RUNNING și trimis înapoi în starea READY (fără ca procesul să se fi terminat). Acest lucru se întâmplă dacă un proces petrece prea mult timp în starea RUNNING, blocând astfel alte procese să fie și ele rulate pe procesor.

Pentru aceasta există o cuantă de timp stabilită de la început. Atunci când un proces ajunge în RUNNING, se inițializează un contor. Când contorul ajunge să aibă valoarea cuantei, procesul va fi preemptat, adică va fi trimis în coada READY și un alt proces îi va lua locul în RUNNING.

2.4 Interactiv: Planificare cu priorități

Round Robin este un algoritm care ne asigură că toate procesele vor ajunge rapid pe procesor. Cu toate acestea, fiind din aceeași clasa cu FCFS, consideră toate procesele ca fiind egale. Există însă situații în care nu ne dorim acest lucru. Planificarea cu priorități rezolvă problema. Vom avea mai multe nivele de priorități (cu cât nivelul este mai mare, cu atât prioritatea procesului este mai bună). Atunci când se află în starea NEW, procesele vor primi priorități inițiale. Procesele vor intra în starea RUNNING în ordinea nivelelor (mai întâi toate cele de pe nivelul 4, apoi toate cele de pe nivelul 3 și tot așa până la nivelul 1). Algoritmul este tot unul preemptiv, deci la finalizarea cuantei, un proces neterminat va trece înapoi în starea READY.
Un plus de preemțiune pe care îl aduce acest algoritm este că, dacă la un moment dat în starea RUNNING se află un proces de nivel 3 și în READY sosește un proces de nivel 4, procesul aflat în RUNNING va fi preemptat, chiar dacă nu i-a expirat cuanta, și va face loc celui de nivel 4.
Pe același nivel, procesele vor fi tratate FCFS.

3 Cerințe

Pentru rezolvarea temei veți avea de urmărit o serie de comenzi pe care le veți primi dintr-un fișier de intrare. Numele fișierului de intrare este primul argument din linia de comandă. Rezultatele voastre vor fi scrise într-un fișier de ieșire, al cărui nume este al doilea argument din linia de comandă.

3.1 Comenzi

COMANDĂ	DESCRIERE
t	**tick** - trece un timp de ceas. Mai întâi se scurge o unitate de timp și apoi se fac modificările posibile. În cadrul acestei operații se iau decizii referitoare la starea RUNNING: preemptare, terminare, trimitere în READY, trimitere proces din READY în RUNNING
a nume x y	**add** - vine un nou proces din NEW, unde nume = nume_proces, x = timpul necesar pentru rulare și y = prioritatea (ignorați y pentru FCFS, SJF și Round Robin)
ma nume1 x1 y1 nume2 x2 y2 ...	**multiple add** - asemănător comenzii add, doar că vin mai multe procese deodată
w	**wait** - procesul din RUNNING trece în WAITING
e nume	**event** - un event trezește procesul din WAITING, unde nume = nume_proces care primește evenimentul
s	**show** - afișează informații despre procesul din RUNNING, în această ordine: nume_proces, timp rămas până la terminarea procesului. Dacă nu există un proces în RUNNING se afișează linie goală

Pentru comenzile **a**, **ma**, **w** și **e** se va aplica și un **t** implicit. Mai întâi se execută operațiile pentru comanda respectivă, apoi se execută **t** și se fac modificările necesare dacă este nevoie.

***Exemplu pentru SJF**: Presupunem că în starea RUNNING se află un proces care mai are 1t până se termină și în READY se află 3 procese cu următorii timpi: P1 (6t), P2(7t), P3(8t). Dacă se primește comanda **a P4 3 1** (procesul cu numele P4, timpul de viață 3 și prioritatea 1, care va fi ignorată la acest algoritm), atunci mai întâi se va băga P4 în coada READY și tot la această comandă se va executa t,*

moment în care procesul din RUNNING se va termina și pe procesor va intra P4 (are cel mai mic timp de rulare).

***Exemplu pentru FCFS**: Presupunem că în starea RUNNING și în READY nu se află niciun un proces. Dacă se primește comanda **a P1 3 1** (procesul cu numele P1, timpul de viață 3 și prioritatea 1, care va fi ignorată la acest algoritm), atunci mai întâi se va adaugă P1 în coada READY și tot la această comandă se va executa t. Se scade 1t din timpul de viață al procesului din RUNNING (nimeni) și apoi se adaugă procesul P1 în RUNNING. La sfârșitul acestui comenzi P1 se va afla în RUNNING și va avea timpul de viață 3. Același lucru s-ar fi întâmplat și dacă RUNNING și READY erau goale, P1 se afla în WAITING și s-ar fi primit comanda e P1 (e ar fi trimis P1 din WAITING în READY și apoi t implicit ar fi trimis P1 din READY în RUNNING.*

***Exemplu pentru planificare cu priorități**: Presupunem că în starea RUNNING se află un proces de prioritate 3, în READY nu se află niciun proces de prioritate mai mare (altfel s-ar fi aflat acela în RUNNING) și se primește comanda **a P1 3 4** (procesul cu numele P1, timpul de viață 3 și prioritate 4). Mai întâi se adaugă P1 în coada READY corespunzătoare priorității 4 și apoi se va executa t, în care prima oară se scade 1t din timpul de viață al procesului din RUNNING. Dacă în urma acestei scăderi procesul din RUNNING s-a terminat, atunci este eliminat și în locul său va sosi procesul P1 pentru că are cea mai mare prioritate. Dacă procesul din RUNNING nu s-a terminat, el va fi trimis în READY și în locul său va fi introdus procesul P1 în RUNNING. Dacă procesul din RUNNING ar fi avut de asemenea prioritatea 4, el nu ar fi fost preemptat din starea RUNNING.*

3.2 Cerințe

Scopul temei este să implementați cele 6 comenzi de la punctul anterior pentru toate cele 4 tipuri de planificari. Pentru aceasta, va trebui să vă creați o structură care să țină minte date pentru fiecare proces în parte. Această structură trebuie să fie aceeași, indiferent de tipul de planificare. **Creearea de structuri diferite pentru planificări diferite va duce la depunctări.**

3.2.1 Bonus

În cadrul planificării cu priorități putem avea priorități statice sau dinamice. Prioritățile statice sunt cele care, odată primite la intrare, nu se mai modifică. Avem însă posibilitatea de a modifica prioritățile în funcție de cum se comportă procesele de-a lungul timpului. Astfel, un proces care a fost scos de două ori din RUNNING va fi pedepsit și îi va scădea prioritatea cu un nivel (dacă este scos de 4 ori cu 2 nivele etc.). Un proces care iese singur de două din RUNNING și se duce în WAITING va fi premiat și îi va crește prioritatea cu un nivel. Dacă un proces a fost preemptat din RUNNING și apoi a trecut singur din nou în WAIT, atunci va fi premiat dacă următoare dată va trece din nou în WAIT.

3.3 Fișiere intrare/ieșire

Fișierele de intrare vor avea următorul format:

```
1/2/3 x/4 x y/5 x y
     command1
     command2
     command3
        ...
```

unde: 1 = FCFS, 2 = SJF, 3 = Round Robin (x = cuanta), 4 = Planificare cu priorități (x = cuanta, y = numărul de nivele de prioritate), 5 = Planificare cu priorități - bonus (x = cuanta, y = numărul de nivele de prioritate), iar comenzile sunt cele specificate la punctul **3.1**.

Fișierul de output va conține doar informațiile afișate de comanda **s**, câte una pe linie, respectiv linii goale pentru atunci când comanda **s** nu are ce afișa.

4 Punctaj

Cerința	Punctaj
First Come First Served	10p
Shortest Job First	20p
Round Robin	30p
Planificare cu priorități	30p
Cosing style, README, warning-uri	10p
Bonus	20p

5 Precizări și restricții

- Temele trebuie să fie încărcate pe vmchecker. **NU** se acceptă teme trimise pe e-mail sau altfel decât prin intermediul vmchecker-ului.
- **NU** folosiți platforma vmchecker pentru debugging. Pentru debug vă încurajăm să folosiți checker-ul atașat cerinței.
- O rezolvare constă într-o arhivă de tip **zip** care conține toate fișierele sursă alături de un **Makefile**, ce va fi folosit pentru compilare și un fișier **README**, în care se vor preciza detaliile de implementare. Toate fișierele se vor afla în rădăcina arhivei.
- Makefile-ul trebuie să aibă obligatoriu regulile pentru **build** și **clean**. Regula **build** trebuie să aibă ca efect compilarea surselor și crearea binarului **planificator**.
- Nu se pot folosi vectori. Implementarea se va face în mod exclusiv cu liste (liste de liste etc.). Singurele excepții sunt un buffer (care poate fi alocat static) de dimensiune maxim 100 pentru citirea comenzilor și, de asemenea, șiruri de caractere pentru reținerea numelor proceselor de dimensiune maxim 21.
- Atașat temei găsiți 5 fișiere cu exemple explicate referitoare la cum rulează fiecare tip de planificare.

Rularea temei se va face astfel: **./planificator input output**, unde scopul celor două fișiere a fost deja explicat anterior.

6 Change log

- **(13.03)** Am corectat exemplul pentru PP. Am adăugat o mențiune despre șirurile de caractere.
- **(15.03)** Am corectat testele si referintele pentru *PP/in21*, *PP/in27*, *Bonus/in30* si *Bonus/in31*.
- **(16.03)** Am modificat numele binarului din checker.