Curs 8

CPU SCHEDULING

CPU scheduler => selecteaza in proces ready din coada si il aloca unu core CPU

=>aceste decizii se iau cand un proces trece din running in wait, din running in ready, din waiting in ready sau cand se termina

=> situatiile 1 si 4 = nonpreemptive

=> situatiile 2 si 3 = preemptive

Dispatcher => da control CPU-ului pt procesul selectat ales de CPU SCHEDULER

Dispatcher latency => timpul pe care il consuma dispatcher-ul pentru a opri un proces si pentru a porni unul nou

Programarea proceselor se face dupa urmatoarele criterii:

1. CPU utilization = cat mai ocupat posibil
2. Debit = cat mai mare
3. Timpul de exeutie pentru fiecare proces in parte (minim)
4. Timpul de asteptare al unui proces aflat in coada (minim)
5. Timpul de raspuns (minim)

First Come First Served (FCFS)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

La acest algoritm putem obtine cu totul alte rezultate in cazul in care procesele ajung in alta ordine, si putem obtine astfel in avg time mai bun.

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

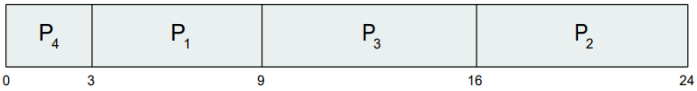
Shortest-Job-First Scheduling

Ideea pentru acest algoritm de programare a proceselor consta in alegerea celui mai scurt proces primul.
SJF este considerat a fi optimal (doar daca joburile sunt disponibile simultan).
Presupunem ca t1, t2 ,t3, ..., tn sunt timpii de rulare:
Job1 termina la t1
Job2 termina la t1+t2
.....
Jobn termina la t1+t2+...+tn

Deci timpul mediu de asteptare va fi egal cu 1/n(nt1 + (n-1)t2 + ... + tn)
Exemplu:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Pentru a putea prezice durata urmatorului proces avem (alfa = ½):

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \leq \alpha \leq 1$
4. Define:

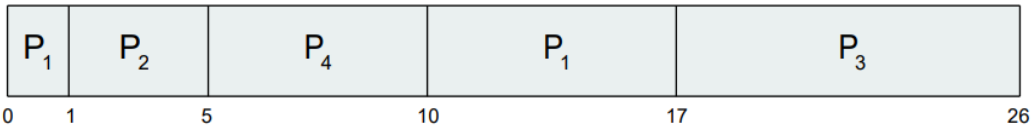
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

Shortest -remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5
- SJF nepreemptiv are average waiting time = 7.75 ms

Round Robin (RR)

Acest algoritm implica utilizarea unei cuante de timp care consuma o valoare din fiecare proces. Daca sunt n procese gata in coada iar cuanta de timp este q, atunci fiecare proces va primi 1/n din timpul CPU in grupe de cel mul q cuante odata. Niciun proces nu va astepta mai mult decat (n-1)*q.

Daca q=mare => FCFS q = mic =>RR

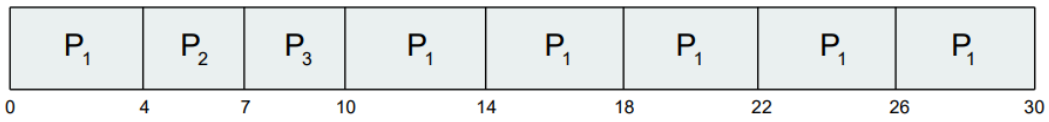
Observatii

Cuanta de timp trebuie aleasa in prinicipiu mai mare decat durata context switch-ului. (cuanta scurta => multe context switch-rui care reduc eficienta utilizarii CPU, iar cuanta prea mare => reduce timpul de raspuns pt programele interactive).

Exemplu de functionare a algoritmului RR cu q = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Average waiting time = 17/3 = 5.66 ms
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

OBS:

Round Robin presupune ca toate procesele au aceeasi importanta, fapt total nerealist. Se introduce notiunea de prioritate. Astfel procesul cel mai important are prioritate maxima si planificatorul va alege de fiecare data procesul cu prioritate maxima la acel punct. (pentru a se preveni rularea indefinita planificatorul poate reduce prioritatea procesului care ruleaza cu fiecare tact de ceas).

daca prioritatea scade sub cea a urmatorului proces prioritar => context switch

Asignarea prioritatilor:

Dinamic => procesul I/O-bound primeste prioritatea 1/f, unde f este fractiunea din cuanta CPU utilizata la ultima rulare de catre proces (ruleaza mai putin => prioritate mai mare)

Static => prioritatile sunt definite de utilizator

OBS:

Clasele de prioritate + RR => se ruleaza cu RR doar procesele din clasa de prioritate maxima. Daca nu exista procese in acea clasa se trece la urmatoarea, si trebuie ajustate prioritatile pt a evita starvation.

Starvation = procese cu prioritate mica nu se executa niciodata

Aging = solutie pt starvation = cresterea prioritatii odata cu trecerea timpului

Exemplu de priority scheduling simplu:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart

- Average waiting time = 8.2

Exemplu de priority scheduling cu round robin:

- Run the process with the highest priority. Processes with the same priority run round-robin
- Example:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Gantt Chart with time quantum = 2

NORMA Scheduling

in sistemele distribuite, fiecare procesor executa local algoritmul de planificare

Exemplu:

- Q: cum se comporta un grup de procese care interactioneaza strans si ruleaza pe masini diferite?
- ex: procesele A, B ruleaza pe P1; C si D pe P2
- masinile P1 si P2 ruleaza RR cu q=10ms
- A si C ruleaza in cuantele pare, B si D in cele impare
- A porneste si cheama D in cuanta para, D nu ruleaza (C ruleaza in cuantele pare)
- dupa 10ms, D primeste mesajul lui A si raspunde; cuanta e impara, A nu ruleaza (B ruleaza)
- dupa inca 10 ms, A primeste mesajul lui D

$\Rightarrow$ schimbul de mesaje are un overhead de 20 ms

Co-scheduling

Procesele care comunica frecvent trebuie sa ruleze simultan pe masini diferite.

Exemplu

- implementare co-scheduling cu matrici
  - fiecare coloana contine procesele rulate pe un anume procesor (masina)
  - fiecare linie contine procesele care ruleaza simultan procesoare
  - i.e., coloanele reprezinta procesoare(masini), liniile cuante de timp
  - fiecare procesor planifica RR local
  - procesoarele isi sincronizeaza cuantele RR folosind bcast (un ceas sincron ca la SIMD, posibil implementat prin NTP)
  - co-scheduling: toate procesele unui grup au alocate aceleasi cuante de timp (i.e., apar pe aceleasi linii ale matricii in diferite coloane) => maximizarea paralelismului disponibil si a throughput-ului comunicatiei

Planificare de timp real

Sistem de timp real = sistem in care corectitudinea executiei unui program nu depinde doar de corectitudinea rezultatelor ci si de timpul la care sunt livrate rezultatele (Hard => toate rezultatele trebuie livrate la timp
Soft => rezultatele calculate pot fi acceptate si cu intarziere)
Task = o singura executie a unui bloc de cod
Timp de sosire al taskului = T la care sis. constientizeaza ca taskul trb executat
OBS:
taskurile perioadice = taskurile care sosesc la intervale regulate de timp (T(t) = timp fix inter-sosiri (perioada), C(t) = timp de calcul)
taskuri aperiodice = caracterizate de rate de sosiri stochastice
pot veni in rafala => supraincarcare sistem
trebuie sa existe timp minim intre 2 sosiri succ ala taskului
= taskuri sporadice = pot veni si ele intr un timp scurt
= daca taskurile periodice incarca sistemul atunci spunem ca acesta se afla in stare de supraincarcare trecatoare
Timpul de eliberare = release time = timpul la care taskului i se permite pornirea executiei
Deadline = timpul pana la care trebuie sa se termine executia taskului
OBS: sosire -> release -> calcul -> deadline

Rate Monotonic (RM)

Este un algoritm pentru taskuri periodice, mai exact a unor taskuri care respecta urmatoarele constrangeri:

1. Orice task neperiodic din sistem e special: fie rutina de initializare, fie de recuperare din eroare (inlocuiesc taskurile periodice cand ruleaza dar nu deadline-uri hard)
2. Taskurile au asignate prioritati statice (taskuri cu rate de sosiri mari au prioritate mica)
3. Taskurile se executa preemptiv
4. Planificatorul alege intotdeauna taskul cu prioritate maxima

OBSERVATIE

- utilizarea *U* a procesorului de catre *n* taskuri

$$U = \sum_{i=1}^{n} \frac{C_i}{Ti}$$

- **Lema**: daca U < ln 2, planificarea RM e fezabila (toate deadline-urile sunt respectate)
- in practica, U < ln 2 e o cerinta conservatoire, exista seturi de taskuri care se pot planifica RM si a caror utilizare depaseste ln 2

Exemplu pentru Rate Monotonic

- sistem multimedia
  - task audio cu perioada T_a = 10 ms si C_a = 2 ms
  - task video cu perioada T_v = 30 ms si C_v = 10 ms

  => U_a = 2/10 = 0.2

  U_v = 10/30 = 0.33

  => U = 0.53 < ln 2 = 0.69 => taskurile sunt planificabile
- taskul audio are frecventa maxima (perioada cea mai scurta) => are si prioritate maxima
- pp. C_v = 20 ms => U_v = 20/30 = 0.66=> U = 0.86 > ln 2 ; Totusi, o diagrama de timp arata ca taskurile sunt planificabile RM daca taskul audio are prioritate maxima

Earliest Deadline First

Taskul cu deadline-ul cel mai apropiat are prioritate maxima.

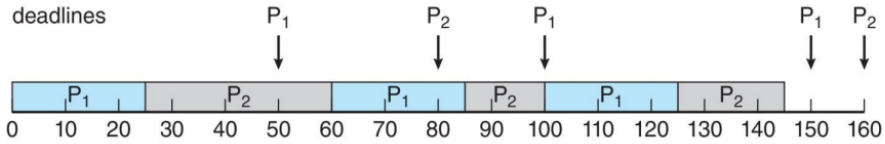
Exemplul de la RM e planificabil si EDF.

un set de *n* taskuri e planificabil daca

$$U = \sum_{i=1}^{n} \frac{C_i}{Ti} <= 1$$

Exemplu EDF

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority
- Figure



Least Slack First

Slack = durata maxima de timp cu care un task poate fi intarziat fara a-si pierde deadline-ul. LSF alege pt rulare task-ul cu slack-ul cel mai mic

EDF si LSF se pot folosi si pt planificarea taskurilor aperiodice.

Se poate demonstra ca EDF e optimal (daca orice alt algoritm poate produce o planificare fezabila pt un set de taskuri pr care stim timpii de sosire, de calcul si de deadline atunci si EDF o poate face) la fel si LSF

Alocarea proportionala de resurse

Noua abordare = se renunta la conditia ca toate procesele sa termina cu siguranta inainte de deadline

= planificatorul controleazaa rata de executie a proceselor variind portia de resurse pe care procesele o primesc

Planificare de tip loterie (randomizat)

Idee: drepturile asupra resurselor sunt exprimate prin tichete de loterie. Alocarea unei resurse se face prin organizarea unei loterii: resursa e acrodata procesului care detine tichetul castigator, alocarea e proportionala cu numarul de tichete detinut de proces.

Tichetele de loterie = dreptul de a folosi o fractiune din resursa

= abstracte SAU relative SAU uniforme

Tichete abstracte = cuantifica dreptul de folosire a resursei independent de detaliile HW

Tichete relative = fractiunea de resursa pe care o cuantifica variaza proportional cu numarul total de tichete detinute

Tichete uniforme = indiferent de tipul de resursa, procentele de alocare ale proceselor sunt reprezentate omogen prin tichete

Algoritm

La inceputul fiecarei cuante, planificatorul organizeaza o loterie, iar resursa e acordata castigatorului. Tichetul castigator e ales aleator conform unei distributii uniforme.

Se cauta in lista de procese gata de rulare participantul care detine tichetul castigator.

Exemplu

Ex: nr total de tichete = 20, 5 procese cu [10, 2, 5, 1, 2] tichete

random[0..19] = 15

S1 = 10; S1 > 15? NU

S2 = 12; S2 > 15? NU

S3 = 17; S3 > 15? DA => procesul cu nr. 3 (detine 5 tichete) primeste procesorul

Analiza algoritmului

Running time = generare de nr aleatoare e rapida
= traversarea liste e O(n)
= acumularea sumei partiale

Caracteristici LS

- ➔ Fair dpdv probabilistic
- ➔ Tichetele se pot transfera de la un proces la altul atunci cand acesta se blocheaza din motiv oarecare
- ➔ Inflatia de tichete
- ➔ Tichete de compensare