Operációs Rendszerek szorgalmi, MFQ ütemező implementálása, dokumentálása és tesztelése

Nevem ide

Feladat leírása

Készítsen egy programot Python nyelven, amely egy egyszerű MFQ ütemező működését szimulálja!

A globálisan preemptív, dinamikus prioritásos ütemező az alábbi ütemezési algoritmusokat futtatja az egyes szinteken:

- magas prioritású szint (prioritás = 2) RR ütemező, időszelet: 2
- közepes prioritású szint (prioritás = 1) RR ütemező, időszelet: 4
- alacsony prioritású szint (prioritás = 0) FCFS ütemező

A prioritás kezelése a következőképpen történjen:

- mindig a legmagasabb prioritású, futásra kész taszk fut
- hogyha az RR időszelet kihasználása előtt egy taszk I/O miatt felfüggeszti a futását, akkor eggyel nő a prioritása (ha tud)
- hogyha a taszk az RR időszeletét kihasználja, és még futna tovább, akkor eggyel csökken a prioritása (ha tud)
- ha éppen az RR időszelet kihasználásakor kezd I/O műveletet, akkor marad ugyanazon a prioritáson (eggyel csökkenne az időszelet kihasználása miatt, de eggyel nőne, mert I/O műveletet kezdeményezett)
- ha egy magasabb prioritású taszk beérkezése miatt vonták meg egy taszktól a CPU-t, a prioritása akkor is megmarad, és a taszk bekerül a megfelelő várakozási sor végére
- ha ez éppen akkor történik, amikor a taszk időszelete lejár, akkor csökkentjük a prioritást
- ha ez éppen akkor történik, amikor a taszk lelép I/O-ra, akkor növeljük a prioritást

Újonnan érkező taszkok legmagasabb prioritást kapnak (2), I/O-ról visszaérkező taszkok pedig a megfelelő prioritású sor végére kerülnek. Beérkezés és I/O-ról visszaérkezés a megfelelő időszelet vége felé, de az ütemezési feladatok elvégzése előtt történik. Az új taszkok előbb érkeznek, minthogy a már folyamatban levőek az I/O-ról visszatérnének. Az I/O-ról azonos időpillanatban visszatérő taszkok sorrendje az az I/O-ra lelépés sorrendjével egyezik.

Érdemes megjegyezni, hogy a statikus ütemező feladatnál fennálló vitatható eset (egyetlen taszk fut RR ütemezéssel, és sokadik köre közben bejön egy új taszk: egyből átadja a CPU-t, vagy csak az éppen aktuális RR időszelete végén), ebben a feladatban nem áll fent, mivel egy taszk egy RR szinten csak egyszer használhatja ki az időszeletet.

Bemenet (standard input, stdin)

Soronként egy taszk adatai. Egy sor felépítése (vesszővel elválasztva):

- a taszk betűjele (A, B, C…)
- a taszk indítási ideje (egész szám >= 0), a következő időszeletben már futhat (0: az ütemező indításakor már létezik), azonos ütemben beérkező új taszkok esetén az ABC-sorrend dönt

- a taszk CPU-löketideje (egész szám >= 1)
- a taszk I/O-löketideje (egész szám >= 1)
- az előző két sor tetszőlegesen sokszor ismétlődhet, de mindig CPU-lökettel fejeződik be a taszk

Példa:

```
A,0,6
B,1,7,6,7
C,5,2,4,1,5,2
D,9,3,1,4
```

Itt az A taszk 0. időszelet vége felé, de az átütemezés előtt érkezik (azaz az első időszeletben ő már futhat), és 6 időegységnyi CPU műveletet kell végrehajtania.

A B taszk az 1. időszelet vége felé, de az átütemezés előtt érkezik, és 7 időegységnyi CPU művelet után kell 6 időegységnyi I/O műveletet végrehajtania. Ezután kell még neki a CPU 7 egységig.

A C taszk 5. ütem végén érkezik, 2 egység CPU után 4 egység I/O-t végez, majd 1 egység CPU, 5 egység I/O, és végül 2 egység CPU kell neki.

A bemenet végét EOF jelzi (előtte soremelés biztosan van, és üres sor is előfordulhat).

Kimenet (standard output, stdout)

A kimenet 4 sorból áll: az első sorban az egész számok utolsó jegyei vannak 1-től, ameddig az alul levő sorok valamelyikében van még futó taszk (`123456789012345678901234...`).

A következő 3 sor a 3 prioritás szintnek felel meg. Minden oszlopban pontosan egy taszk-karakter szerepelhet, és a másik kettő szóköz kell legyen. Ha csak a taszk karaktereket nézzük oszloponként, akkor az első sorban szereplő szám-adik időszeletben éppen futó taszk betűjele kell legyen a karakter. A sor pedig az éppen aktuális prioritását mondja meg a taszknak: a felső sor a legmagasabb prioritású, az alsó a legalacsonyabb.

Hogyha az adott időszeletben nem fut érdemi taszk, akkor az idle taszkot X jelölje a legalacsonyabb prioritáson (legalsó sor).

Példa (a fenti bemenetre adott válasz):

```
123456789012345678901234567890123456
AABB CC  DDC     CC  DD    BB
    A  BB   AAABB  DB  DD    BBBB
                        XX      B
```

Itt beérkezik az A taszk, és kihasználja fent a két időszeletét. Közben a beérkezett B taszk kapja meg a CPU-t, szintén 2-es prioritáson. Neki is lejár a 2 időszelete, ezután az 1. prioritás szinten A taszk kapja meg a CPU-t. De csak egy körig, utána C taszk beérkezése miatt A bekerül B mögé a közepes prioritású sorban. C éppen akkor megy el I/O-ra, amikor elvennék tőle a CPU-t, tehát marad ugyanazon a szinten. Ezután B is fut 1-es prioritáson, de őt a D taszk beérkezése löki ki. Most megint A van elöl a közepes prioritás sorában, utána B, utána pedig D, miután lefutotta a magas prioritású időszeletét. Most tartunk a 12. ütemnél.

C taszk visszaérkezik I/O-ról, és egyetlen időszeletét rögtön le is futja. Upgradeelni kellene, de ennél nincs magasabb prioritás, így marad 2-es prioritáson. A közepes prioritáson A taszk van elöl a sorban, lefutja a hátralevő idejét, és befejeződik. B taszk elkezd futni, de közben C megszakítja: visszatér I/O-ról, lefutja a maradék szükséges idejét, és befejeződik. Ezután D lesz elöl a sorban, egy időszeletet fut, majd elmegy I/O-ra. B fut egyetlen ütemet, majd D visszatér I/O-ról. B éppen elment I/O-ra, így megnőtt a prioritása. Hogyha B nem ment volna el I/O-ra, D akkor is prioritást élvezne a 22. ütemben: itt látszik például, hogy egy rövid I/O lökettel hogyan lehet kijátszani ezt az algoritmust, hogy több CPU-időt kapjon a taszk: itt D azért kerül fel megint a magas prioritású sorba, mert egyetlen ütemig I/O-zott. D lefut, majd 2 ütemig tétlen a CPU, majd B visszatér I/O-ról, és ő is lefut.

Tesztadatok

(kézzel lettek előbb végigszámolva, a hibázás csökkentése érdekében)

1. teszt: a fentebbi, összetettebb példa

```
A,0,6
B,1,7,6,7
C,5,2,4,1,5,2
D,9,3,1,4

12345678901234567890123456789012345678901234
AABB CC  DDC     CC  DD    BB
    A  BB   AAABB  DB  DD    BBBB
                          XX      B
```

2. teszt: egy hosszú taszk, és több rövid érkezik közben

```
A,0,12
B,2,2,5,3
C,14,2

12345678901234567890123456789
AABB     BB   CC
    AAAA   B
        A   AA  AAA
```

Itt látszik, hogy a rövid CPU-löketű taszkok előnyt élveznek.

3. teszt: több, rövid CPU-löketű taszk osztozik

```
A,0,2,5,3,4,1
B,3,1,6,2
C,6,3,3,1
D,2,3,4,2

123456789012345678901
AADDB CCAADDBB    C A
     D        CA
               XX X
```

A rövid CPU-löketű taszkok körülbelül egyenlően osztoznak a CPU-n.

4. teszt: rövid taszkok kiéheztetnek egy CPU-intenzív taszkot

```
A,0,10
B,3,2,2,2,2,2
C,4,2,2,2,2,2

123456789012345678901 2
AA BBCCBBCCBBCC
  A             AAAA
                   AAA
```

Az általunk implementált változat egyik problémája: a CPU intenzív taszkok sok I/O intenzív taszk mellett éheznek.

5. teszt: egy program kijátssza az algoritmust, rövid I/O-löketekkel

```
A,0,10
B,1,5,1,5,1,5

123456789012345678901234 5
AABB         BB     BB
    AAAABBB    BBB    BBB
           A      A      AA
```

Egy másik felmerülő probléma: egy ártalmas B taszk csak azért generál I/O műveleteket, hogy magasabb prioritást kapjon, és előnyre tegyen szert a többi taszkkal szemben.

6. teszt: egy CPU-intenzív taszk I/O-intenzív lesz, illetve fordítva

```
A,0,13,4,2,4,1,3,2
B,5,3,6,2,1,12

12345678901234567890123456789012345 67
AA    BB            BB BB      A    AA
  AAA   AAAAB         A   ABBBB
             AAAAXX             BBB   BBB
```

Ez a példa bemutatja, hogy futás közben is tud az algoritmus alkalmazkodni a taszkok változó viselkedéséhez: az eredetileg CPU-intenzív A taszk először alacsony prioritásra süllyed, majd amikor I/O intenzívvé válik, akkor a szabályoknak köszönhetően magától magasabb prioritást kap. B taszk esetén pont fordítva.

Implementáció értékelése

Az előbbiekben bemutatott ütemező egy egyszerű változata az MFQ-nak. Egy valós rendszerben sokkal több paraméterrel (szintek száma, időszeletek hossza, stb.), és bonyolultabb prioritásszámítással kerülne az ütemező megvalósításra.

A hivatkozott forrásban[1.] a prioritásnövelést nem I/O-ra történő várakozás miatt kapják a taszkok, hanem bizonyos időnként minden taszk újra a legmagasabb szintre kerül. Ez megoldaná a 4. tesztben vázolt problémát. Továbbá ott esik szó az 5. teszben vázolt problémáról is, melyre azt a megoldást kínálja, hogy számon kell tartani az összes eddigi idejét a taszknak, amit az adott szinten töltött, és annak függvényében kell downgrade-elni.

Az [1.] forrásnak egy érdekes figyelemfelhívása az, hogy az ütemező is egy olyan rendszerkomponens, melynek a támadásokra felkészültnek kell lenni (lásd 5. teszt). A forrás viszont

egy másik fontos, tanult tényezőt nem említ: az ütemező futásának a költségét. Az is egy fontos szempont egy ütemezőnél, hogy minél kevesebb CPU-időt vegyen el a tényleges taszkoktól.

Elkészített programok, fájlok

mfq.py: A feladatra adott megoldás, Python 3-at használ, segédkönyvtárak nélkül. Standard inputon kapja a megadott formában a bemenetet, és a standard outputra írja a kimenetet, szintén a megadott formában.

tests/input mappa: tartalmazza a fentebb listázott tesztesetek bemenetét, 01-06 nevű fájlokban (kiterjesztés nélkül, hogy a fájlnév egyben megegyezzen a teszt sorszámával).

tests/expected mappa: az előzőhöz hasonlóan, 01-06 nevű fájlokban tartalmazza a tesztesetek elvárt kimenetét.

test_one.sh:
```
./mfq.py < tests/input/$1 | diff -s tests/expected/$1 -
```
Elindítja a python programot az első parancssori argumentumként kapott nevű fájlt átirányítva a standard bemenetére. A kimenetet átadja a diff parancsnak, amely összehasonlítja az ugyanolyan nevű elvárt kimenet fájljával, amit a python programtól kapott. Ha megegyezik, kiírja az egyezés tényét (-s).

test_all.sh:
```
for f in tests/input/*
do
./test_one.sh `basename $f`
done
```
Elindítja a test_one.sh programot az összes tests/input mappában található fájl nevével meghívva.

Források

[1.] https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf

Ez a cikk a https://pages.cs.wisc.edu/~remzi/OSTEP/ honlapról származik, és az MFQ ütemezők működését építi fel fokozatosan. Tartozik hozzá egy Python kód is (https://github.com/remzi-arpacidusseau/ostep-homework/blob/master/cpu-sched-mlfq/mlfq.py), melyet nem használtam a feladat elkészítése során.

[2.] https://en.wikipedia.org/wiki/Multilevel_feedback_queue, az algoritmus áttekintéséhez

Előadásdiák: a konkrét paraméterértékek (szintek, időszeletek), és az algoritmus főbb koncepciói (I/O növeli a prioritást)

A tárgy moodle oldalán található „fakultatív feladat: többszintű ütemező megvalósítása”: a feladatleírás szövege, és a szerkezete jelentős részben innét származik