Az online lapozási problémára adott FIFO és LRU algoritmusok átlagos eset vizsgálata

Csernai Kornél

2012. február 24.

1. Feladat

Adott k méretű gyorsmemória és n méretű háttérmemória (általában $n \gg k$). A háttérmemória használata költséges, ezért egyes részeit a gyorsmemóriában is eltároljuk. A rendszer folyamatosan kéréseket kap háttérmemóriára vonatkozóan, amelyeket ki kell elégítenie, lehetőleg a gyorsmemóriából. Ha a gyorsmemória betelik és egy kérésnek nem tudunk eleget tenni, akkor a *laphiba (page fault)* lép fel. A cél a laphibák számának minimalizálása.

Az egyes stratégiák abban különböznek egymástól, hogy a laphibák esetén a kért lapot melyik, már bent levő lap helyére teszi. Az így kiválasztott lapot a gyorsmemóriából el kell távolítani, helyére a kért lap kerül.

A feladat *online*, ha az algoritmus a kéréseket egyesével kapja meg, és minden kérésnél rögtön, helyben dönteni kell annak sorsáról. Ezzel szemben az *offline* feladatnál az algoritmus a teljes bemenettel rendelkezik.

A FIFO és LRU algoritmusok az online lapozási problémát oldják meg. A FIFO algoritmus mindig a legrégebben felvett elemet dobja ki. Ezzel szemben az LRU a legrégebben használt elemet dobja ki. Az offline feladat esetében az optimális megoldás hatékonyan megtalálható.

Ismert, hogy mindkét online algoritmus a legrosszabb esetben az offline optimális megoldás k-szorosát adják (k-versenyképes). Azt viszont nehezebb megtalálni, hogy átlagos esetben hogyan teljesítenek. Az átlagos eset azért érdekes, mert a gyakorlatban használt a lapcímek valamilyen eloszlást követnek. Most csak az egyenletes eloszlásra koncentrálunk. A feladatunk, hogy a két adott online algoritmust megvizsgáljuk empirikus átlagos eset szempontjából.

2. Futási környezet

A mérésekhez implementáltam a FIFO, LRU és az OPT algoritmusokat és elkészítettem egy szimulátor környezetet.

A teszteknek a következő paraméterei vannak:

- k, a gyorsmemória mérete

- n, a háttérmemória mérete
- m, a kérések száma
- $I = \sigma = \{\sigma_1, \ldots, \sigma_m\}$, a kérések sorozata

Minden futás az $\{1, 2, \ldots, n\}$ kérésekkel indul, amely a gyorsmemóriát inicializálja. Ezek nem számítanak bele az m kérésbe. Minden további kéréshez egy-egy véletlenszámot generáltam egyenletes eloszlás szerint 1 és n között.

Egy egyszeri mérésnél az (k, n, m) paraméterek rögzítettek és következő történik:

1. Véletlen sorozat generálása.
2. Optimum megkeresése az OPT algoritmussal. Amennyiben az optimum 0, újra végrehajtom az 1. lépést.
3. Az algoritmusok lefuttatása, online szimuláció.
4. Az egyes algoritmusok eredményeit elosztom az optimummal, ez lesz az algoritmus teljesítménye.

Ilyen egyszeri futásokból 100 ismételt próbát teszek és veszem a teljesítmények átlagát, minimumát, maximumát. A

$$P = \{(k, n, m) \mid 2 \leq k \leq k_{max},\ k + 1 \leq n \leq n_{max},\ 3 \leq m \leq m_{max}\}$$

halmaz adja az összes futás paramétereit, tipikusan $k_{max} = 99$, $n_{max} = 100$, $m_{max} = 100$ konstansokkal.

3. Technikai részletek

A programot Python scriptnyelven, részben objektum-orientált módon valósítottam meg. A futtatási környezet egy interfészt biztosít, amelybe bármely online lapozási algoritmus beilleszthető. Két ilyen algoritmus valósítottam meg: FIFO és LRU.

Az interfész legfontosabb része a `request` függvény. A keretrendszer a soron következő kéréssel paraméterezi a függvényt, amelyet az algoritmus lefuttat, és visszaadja azt a cella indexet, amelyre kicserélné a kért blokkot. Ha nincs cserére szükség, akkor a megtalált blokk indexét kell visszaadni. A keretrendszer számontartja a gyorsmemória aktuális állapotát és feljegyzi a laphibák számát.

A futtatókörnyezet lényege, hogy az összes $p \in P$ paramétert többször lefuttatja, az értékeket aggregálva. Vegyük észre, hogy különböző p paraméterek felett a bemeneti sorozat nem rögzített. A környezet feljegyzett eredményeket fájlokban tárolja, az m paraméter szerint csoportosítva: `run_003.txt, run_004.txt`, stb. Az így készült fájlok soronként oszlopokat tartalmaznak, szóközzel elválasztva. Minden egyes sor az adott p paraméter mellett kapott eredményeket tartalmazza.

Az első sor tartalmazza a fejléc oszlopokat:

- k, a k paraméter aktuális értéke
- n, az n paraméter aktuális értéke
- m, az m paraméter aktuális értéke
- *FIFO_avg*, a FIFO algoritmus átlagos teljesítménye
- *FIFO_min*, a FIFO algoritmus legjobb (minimális) teljesítménye
- *FIFO_max*, a FIFO algoritmus legrosszabb (maximális) teljesítménye
- *LRU_avg*, az LRU algoritmus átlagos teljesítménye
- *LRU_min*, az LRU algoritmus legjobb (minimális) teljesítménye
- *LRU_max*, az LRU algoritmus legrosszabb (maximális) teljesítménye

A megvalósítás részletei

```
1  class FIFO(Algorithm):
2      def init(self, n, k):
3          self.k = k
4          self.store = [i + 1 for i in range(k)]
5          logger.debug(self.store)
6
7          self.lastpos = 0
8
9      def find(self, piece):
10         try:
11             pos = self.store.index(piece)
12             return pos
13         except ValueError:
14             return -1
15
16     def request(self, piece):
17         pos = self.find(piece)
18
19         if pos == -1: # piece not found, page fault
20         # FIFO behavior, changing the i%k-th  element in
               round i
21             drop = self.lastpos
22             self.store[drop] = piece
23             self.lastpos += 1
24             self.lastpos %= self.k
25             return drop
26         else: # piece found
27             return pos
```

```
28
29      def __str__(self):
30          return "FIFO"

 1  class LRU(Algorithm):
 2      def init(self, n, k):
 3          self.k = k
 4          self.store = [i + 1 for i in range(k)]
 5          self.usage = [i for i in range(k)]
 6
 7          # round counter
 8          self.round = k - 1
 9
10          self.lastpos = 0
11
12      def find(self, piece):
13          try:
14              pos = self.store.index(piece)
15              return pos
16          except ValueError:
17              return -1
18
19      def request(self, piece):
20
21          self.round += 1
22
23          pos = self.find(piece)
24
25          if pos == -1: # piece not found, page fault
26              # find the least recently used slot:
27              minpos = -1
28              mintime = self.round
29
30              for i, j in enumerate(self.usage):
31                  if j < mintime: # found a new minimum
32                      minpos = i
33                      mintime = j
34
35              self.store[minpos] = piece
36              self.usage[minpos] = self.round
37              return minpos
38          else: # piece found
39              # actualize last used time for this piece
40              self.usage[pos] = self.round
41              return pos
42
```

```
43      def __str__(self):
44          return "LRU"
```

4. Futtatás

A futtatáshoz Python értelmező szükséges. A következő futtatásokat végeztem el:

`./single.py 5 20 3` – Egyszeri, részletezett futás a megadott N, M, K paraméterekkel.

`./average.py 1000 100000 100 100` – Az utolsó paraméter megadja, hogy hány egyszeri futást végezzünk. Itt már nem részletes.

`./env.py` – A 3. részben leírtakat végzi el.