Hessam Koucheki

9812358**032**

LabOS WorkReport

Bu-Ali Sina University

Dr. Nazari

Spring 1401

02

فهرست مطالب

SJF Non-preemptive

در الگوریتم SJF یا shortest job first، در حالت غیرانحصاری یا غیرپیشگیرانه، زمان‌بندی سی‌پی‌یو به این صورت انجام می‌شود که تسکی که کمترین زمان برای اجرا را نیاز دارد در ابتدا اجرا میشود، و تا زمانی که اجرای آن به صورت کامل به پایان نرسیده باشد، سی‌پی‌یو را آزاد نمی‌کند. بعد از اینکه یک کار به پایان رسید، کار بعدی‌ای که دارای کمترین زمان اجرا است وارد سی‌پی‌یو میشود و اجرا میشود. و این عمل تا زمانی که اجرای همه کارهای موجود تمام شود ادامه می‌یابد.

در ادامه به بررسی سه سناریو از این الگوریتم میپردازیم:

سناریوی اول

با مقادیر زیر شروع میکنیم:

```
[[1, 2, 3, 4, 5, 6, 7, 8],
 [4, 1, 6, 5, 5, 2, 4, 3],
 [5, 8, 8, 13, 7, 13, 7, 14]]
```

پس از اجرای تابع arrangeArrival به نتیجه زیر میرسیم:

```
[[2, 6, 8, 7, 1, 5, 4, 3],
 [1, 2, 3, 4, 4, 5, 5, 6],
 [8, 13, 14, 7, 5, 7, 13, 8]]
```

که همانطورکه مشخص است، تمامی تسک‌ها بر اساس زمان ورودی‌شان به سی‌پی‌یو مرتب شده‌اند. بعد از اجرای الگوریتم SJF روی داده‌های بالا، و شبیه‌سازی اجرای تمام تسک‌ها، به خلاصه نتایج زیر میرسیم:

```
{'PID_2': 0, 'PID_6': 31, 'PID_8': 56, 'PID_7': 11,
'PID_1': 7, 'PID_5': 17, 'PID_4': 44, 'PID_3': 24}
 {'PID_2': 9, 'PID_6': 49, 'PID_8': 76, 'PID_7': 21,
'PID_1': 14, 'PID_5': 28, 'PID_4': 62, 'PID_3': 36}
Avg wait time:  23.75
Avg finith time:  36.875
```

سناریوی دوم

با اجرای مجدد کدهای موجود، و تولید داده‌های جدید، شروع میکنیم:

```
[[1, 2, 3, 4, 5, 6, 7, 8],
 [0, 2, 5, 1, 3, 5, 1, 4],
 [6, 14, 5, 8, 15, 7, 14, 8]]
```

سپس دوباره آن‌ها را مرتب میکنیم:

```
[[1, 4, 7, 2, 5, 8, 3, 6],
 [0, 1, 1, 2, 3, 4, 5, 5],
 [6, 8, 14, 14, 15, 8, 5, 7]]
```

با اجرای الگوریتم روی داده‌های بالا، نتایج زیر حاصل میشود:

```
{'PID_1': 0, 'PID_4': 16, 'PID_7': 30, 'PID_2': 43,
'PID_5': 57, 'PID_8': 23, 'PID_3': 5, 'PID_6': 10}
 {'PID_1': 6, 'PID_4': 26, 'PID_7': 48, 'PID_2': 62,
'PID_5': 77, 'PID_8': 34, 'PID_3': 11, 'PID_6': 18}
Avg wait time:  23.0
Avg finith time:  35.25
```

سناریوی سوم

برای بار سوم و آخر، همه‌ی کدها را دوباره ران میکنیم تا داده‌های تازه تولید شود:

```
[[1, 2, 3, 4, 5, 6, 7, 8],
 [6, 3, 3, 6, 1, 5, 6, 3],
 [8, 12, 12, 15, 15, 7, 7, 6]]
```

با مرتب سازی داریم:

```
[[5, 3, 2, 8, 6, 4, 7, 1],
 [1, 3, 3, 3, 5, 6, 6, 6],
 [15, 12, 12, 6, 7, 15, 7, 8]]
```

و همچنان بعد از اجرای الگوریتم، به نتایج زیر دست پیدا میکنیم:

```
{'PID_5': 0, 'PID_3': 38, 'PID_2': 50, 'PID_8': 13,
'PID_6': 19, 'PID_4': 62, 'PID_7': 26, 'PID_1': 31}
 {'PID_5': 16, 'PID_3': 56, 'PID_2': 68, 'PID_8': 22,
'PID_6': 29, 'PID_4': 83, 'PID_7': 36, 'PID_1': 44}
Avg wait time:  29.875
Avg finith time:  44.25
```

نتیجه گیری

در این سه سناریو، دومی زمان انتظار و پایان کمتری نسبت به سایر داشت؛ سناریوی سوم از همه میانگین زمان انتظار و پایان بیشتر داشت که به دلیل زمان‌های طولانی‌تر تسک‌های موجود، همین انتظار هم از آن میرفت. سناریوی اول در وسط این دو قرار میگیرد.

FIFO

در الگوریتم FIFO یا همان First in First OUT، طرز کاری که داریم به این صورت است که هر تسکی که ابتدا وارد سی‌پی‌یو شود، ابتدا هم خارج میشود. به بیان دیگر، صف‌هایی که ما در دنیای واقع داریم مانند صف نانوایی همه از نوع فیفو هستند، به این صورت که هرکسی که ابتدا وارد صف شده باشد، زودتر از بقیه هم سرو میشود و از صف خارج میشود. این نوع الگوریتم، یکی از ساده‌ترین الگوریتم‌های موجود است و نیاز به پیاده‌سازی خیلی پیچیده‌ای ندارد. در اینجا به بررسی این نوع الگوریتم در سه سناریوی رندوم میپردازیم.

سناریوی اول

بعد از ساخت دیتای تصادفی و مرتب کردن آن‌ها براساس زمان ورود آنها، به نتایج زیر میرسیم:

```
[[3, 4, 6, 7, 2, 1, 5, 8],
 [1, 1, 2, 2, 3, 4, 5, 5],
 [10, 6, 8, 5, 5, 15, 12, 10]]
```

پس از اجرای الگوریتم فیفو روی این دیتا، نتایج زیر حاصل میشود:

```
{'PID_3': 0, 'PID_4': 10, 'PID_6': 15, 'PID_7': 23,
'PID_2': 27, 'PID_1': 31, 'PID_5': 45, 'PID_8': 57}
 {'PID_3': 11, 'PID_4': 17, 'PID_6': 25, 'PID_7': 30,
'PID_2': 35, 'PID_1': 50, 'PID_5': 62, 'PID_8': 72}
Avg wait time:  26.0
Avg finish time:  37.75
```

سناریوی دوم

دیتا های زیر تولید و بر اساس زمان ورود مرتب شده‌اند:

```
[[1, 3, 4, 8, 6, 5, 7, 2],
 [0, 2, 2, 2, 4, 4, 5, 5],
 [14, 15, 7, 9, 11, 13, 7, 11]]
```

با اجرای الگوریتم روی آن، نتایج زیر را داریم:

```
{'PID_1': 0, 'PID_3': 12, 'PID_4': 27, 'PID_8': 34,
'PID_6': 41, 'PID_5': 52, 'PID_7': 64, 'PID_2': 71}
 {'PID_1': 14, 'PID_3': 29, 'PID_4': 36, 'PID_8': 45,
'PID_6': 56, 'PID_5': 69, 'PID_7': 76, 'PID_2': 87}
Avg wait time:  37.625
Avg finish time:  51.5
```

سناریوی سوم

برای بار سوم و آخر، تعدادی دیتای رندوم تولید میکنیم:

```
[[2, 4, 5, 1, 3, 6, 7, 8],
 [4, 4, 4, 5, 5, 5, 5, 6],
 [12, 6, 10, 13, 8, 13, 8, 10]]
```

با اعمال الگوریتم، به این نتایج دست پیدا میکنیم:

```
{'PID_2': 0, 'PID_4': 12, 'PID_5': 18, 'PID_1': 27,
'PID_3': 40, 'PID_6': 48, 'PID_7': 61, 'PID_8': 68}
 {'PID_2': 16, 'PID_4': 22, 'PID_5': 32, 'PID_1': 45,
'PID_3': 53, 'PID_6': 66, 'PID_7': 74, 'PID_8': 84}
Avg wait time:  34.25
Avg finish time:  49.0
```

نتیجه گیری

الگوریتم فیفو، علی‌رغم اینکه از نظر پیاده‌سازی بسیار ساده است، اما اگر تسک‌های اولیه دارای زمان اجرای زیادی باشند در حالی که تسک‌های آتی دارای زمان اجرای کم باشند، بدترین عملکرد را از خود دارد و موجب زمان انتظار بسیار زیاد تسک‌های بعدی، و در نتیجه کاهش راندمان برنامه و گرسنگی بسیار زیاد تسک‌ها میشود. این الگوریتم زمانی در استفاده خوب عمل میکند که:

- تسک‌ها از نظر اهمیت اجرا با یکدیگر تفاوتی نکنند، یعنی همگی دارای یک اولویت باشند.
- تسک‌ها همگی از نظر زمان مورد نیاز برای اجرا در یک سطح باشند، یعنی مدت زمان اجرای کارها با یکدیگر تفاوت زیادی نداشته باشد.
- زمان ورود تسک‌ها، تقریبا دارای یک فاصله مساوی باشد، یعنی یک تسک در ثانیه 1 و تسک بعدی در ثانیه 5 و بعدی در 25 نباشند، بلکه چیزی شبیه به 1 – 3 – 5 – 8 باشد.
- اختلاف عددی بین زمان ورود تسک‌ها و مدت زمان اجرای آن‌ها زیاد نباشد. یعنی بتوانیم به نوعی مطمئن باشیم که اگر تسکی که اول آمده در حال انجام شدن است، با ورود تسک بعدی، آن تسک مدت زمان زیادی منتظر نخواهد ماند و ما میتوانیم به خیال راحت به تکمیل تسک اول بپردازیم.

همانطورکه در این سناریو ها مشاهده کردیم، اعداد میانگین انتظار و تکمیل در فیفو بستگی زیادی به نوع تسک‌ها و داده‌ها دارند و میتوانند اختلاف زیادی با یکدیگر داشته باشند.

با توجه به گفته‌ها، بهتر است در پروژه‌هایی که از نوع داده‌ها و در مورد تسک‌ها اطلاعات دقیقی نداریم و آن‌ها تقریبا رندوم هستند، از این الگوریتم استفاده نکنیم.

در نوشتن این الگوریتم، به نظر یک اشتباه وجود دارد که برای بعضی از ورودی‌ها، خطای index out of range رخ میدهد که مربوط به این خط است:

```
running[currentTime] = currentTask
```

تفاوت این الگوریتم، با نسخه‌ی اول آن که با هم بررسی کردیم این است که در این الگوریتم، اگر کاری به سی‌پی‌یو واگذار شد، قابل بازپس‌گیری است و میتوانیم بعد از اینکه یک کاری وارد سی‌پی‌یو شد و در حال اجرا بود که ناگهان تسکی با اولویت بالاتر وارد شد، تسک فعلی را تا جایی که انجام شده‌است در جایی ذخیره کنیم و تسک مهم‌تر را وارد سی‌پی‌یو کرده و مشغول به تکمیل آن شویم.

این الگوریتم در سیستم‌هایی که در آن، تسک‌ها دارای اولویت هستند بسیار کاربردی است و مطلوب است از این روش برای زمان‌بندی سی‌پی‌یو در آن‌ها استفاده کنیم.

Round Robin

در الگوریتم Round Robin، ما با استفاده از یک کوانتوم زمانی، مدام به صورت دایره‌وار بین تسک‌ها میچرخیم و هربار به اندازه یک کوانتوم به آن‌ها زمان اجرا میدهیم. در این الگوریتم که پیاده‌سازی کامل آن در فایل main.ipynb در کنار همین گزارش موجود است، به هر تسک فارغ از اهمیت و یا زمان رسیدن آن، یک کوانتوم زمانی اختصاص میدهیم و آن کار در سی‌پی‌یو اجرا میشود تا زمانی که کوانتوم آن تمام شود. در این الگوریتم به دلیل تعدد سوییچ‌های سی‌پی‌یو، سربار بسیار زیادی وجود دارد و هزینه‌ی زیادی برای تغییر تسک‌های موجود در سی‌پی‌یو متحمل میشویم. این نوع الگوریتم برای سیستم‌هایی که تسک‌های قابل اجرا در آن همگی از اهمیت یکسانی برخوردار هستند مناسب میباشد. برای مثال در سیستم‌های محاسبات ابری، که همه کاربران از اهمیت یکسانی برخوردار هستند این الگوریتم پیشنهاد میشود. اگر در این کلود های ابری، کاربران قادر بودند با پرداخت پول و حق اشتراک بیشتر، قدرت پردازش بیشتری هم در اختیار داشته باشند، میتوان با اختصاص دو یا چند کوانتوم زمانی به آن‌ها، زمان بیشتری از سی‌پی‌یوی مشترک را به آن‌ها اختصاص داد و کارهای آن‌ها را سریع‌تر از بقیه انجام داد.

در ادامه به بررسی چند سناریو از اجرای این الگوریتم میپردازیم:

سناریوی اول

```
Processes Burst Time     Waiting Time Turn-Around Time
  1               6                 28                34
  2               15                62                77
  3               9                 53                62
  4               13                63                76
  5               7                 48                55
  6               6                 38                44
  7               10                56                66
  8               11                62                73

Average waiting time = 51.25000
Average turn around time = 60.87500
```

سناریوی دوم

```
Processes Burst Time     Waiting Time Turn-Around Time
  1               8                40                48
  2               10               48                58
  3               6                32                38
  4               13               56                69
  5               5                36                41
  6               15               57                72
  7               5                39                44
  8               10               54                64

Average waiting time = 45.25000
Average turn around time = 54.25000
```

سناریوی سوم

```
Processes Burst Time     Waiting Time Turn-Around Time
  1               5                28                33
  2               11               61                72
  3               9                54                63
  4               7                45                52
  5               12               62                74
  6               11               64                75
  7               14               67                81
  8               14               69                83

Average waiting time = 56.25000
Average turn around time = 66.62500
```

با تشکر و احترام

حسام کوچکی

9812358032