گزارش دستورکار هشتم آزمایشگاه سیستم‌های عامل، قسمت دوم

نگار موقتیان، ۹۸۳۱۰۶۲

بهبود الگوریتم FCFS

الگوریتم FCFS ساده‌ترین نوع الگوریتم‌های زمانبندی بوده و به طور کلی برای زمانی مناسب است که پردازه‌هایی با waiting time کوتاه و نسبتاً یکنواخت داریم. اگر طول پردازه‌ها بسیار متفاوت باشد این خطر وجود دارد که پردازه‌های کوتاه مجبور شوند برای اجرا منتظر پردازه‌های طولانی مانده و در نتیجه اثر کاروان به وجود آمده و متوسط زمان اجرای برنامه‌ها زیاد شود. همچنین این الگوریتم ذاتاً با اجرای موازی پردازه‌ها و پایپ‌لاین سازگار نیست.

بنابراین برای بهبود این الگوریتم نیاز است آن را به صورتی تغییر دهیم که پردازه‌های طولانی نتوانند تا اتمام کار خود CPU را بدست بگیرند و پردازه‌های کوچک‌تر مجبور نباشند تا انتها منتظر آن‌ها بمانند. اگر این الگوریتم را به گونه‌ای تغییر دهیم که پردازه‌های کوچک‌تر در ابتدا اجرا شوند این الگوریتم حالت SJF به خود می‌گیرد و اگر آن را به گونه‌ای تغییر دهیم که به صورت قبضه‌ای عمل کند و نگذارد پردازه‌های طولانی‌تر بیش از یک حد مشخص اجرا شوند الگوریتم حالت RR به خود می‌گیرد.

می‌توان از تکنیک‌های دیگری نیز برای بهبود الگوریتم استفاده کرد. برای مثال می‌توانیم به این صورت عمل کنیم که اگر زمان اجرای یک پردازه پردازه‌ای کوتاه‌تر وارد سیستم شد شانس اجرا را به او داده و سپس به پردازهٔ قبلی بازگردیم و در غیر این صورت به اجرای پردازه‌ها به صورت FCFS ادامه دهیم.

یا تا مدتی مشخص به پردازه زمان اجرا داده و پس از آن پردازه را قبضه کرده، پردازنده را در اختیار پردازهٔ بعدی موجود در صف FCFS قرار دهیم و پس از سپری شدن مدت مشخص برای پردازهٔ دوم باز به پردازهٔ اول باز گردیم. در این صورت پردازه‌های متنوع‌تری فرصت اجرا پیدا خواهند کرد و احتمال اجرای پردازه‌های کوچک‌تر افزایش می‌یابد.

بهبود الگوریتم SJF

برای الگوریتم SJF اثبات می‌شود که می‌توان کمترین میانگین زمان انتظار را داشت و این الگوریتم از این نظر به خوبی عمل می‌کند. اما مشکل این الگوریتم این است که با این روش ممکن است پردازه‌های طولانی دچار قحطی شده و پردازنده به آن‌ها اختصاص نیابد.

می‌دانیم الگوریتم SJF را می‌توان گونه‌ای از الگوریتم مبتنی بر الویت در نظر گرفت که در آن الویت پردازه‌ها متناظر با زمان اجرای آن‌هاست (و لذا پردازه‌های طولانی الویت کم‌تری خواهند داشت). بنابراین برای اینکه از قحطی در این الگوریتم جلوگیری کنیم می‌توانیم از aging در پردازه‌ها استفاده کنیم. برای این کار در ابتدای ورود هر پردازه به عنوان الویت زمان burst time آن را تخصیص می‌دهیم و در طول زمان از این عدد الویت کم می‌کنیم تا الویت پردازه افزایش یابد. از این طریق پردازه‌های کوتاه‌تر در الویت قرار خواهند گرفت اما پردازه‌های طولانی نیز به مرور زمان شانس اجرا شدن پیدا خواهند کرد و لذا دچار قحطی نخواهند شد.

بهبود الگوریتم مبتنی بر الویت

الگوریتم مبتنی بر الویت بر خلاف الگوریتم‌های دیگر الویت پردازه‌ها را نیز در نظر می‌گیرد و از این جهت برای استفاده در سیستم‌های معمول (که در آن‌ها کارها الویت‌های متفاوتی دارند) برتری دارد. اما این روش نیز خطر قحطی برای پردازه‌های کم الویت را در پی دارد.

همانطور که در قسمت قبل نیز اشاره شد برای جلوگیری از قحطی پردازه‌های کم الویت می‌توانیم از aging استفاده کنیم. با استفاده از این تکنیک در هر مرحله به الویت پردازه‌ها اضافه خواهد شد و به طور طبیعی هر پردازه‌ای که قدیمی‌تر باشد به مرور الویت بیش‌تری کسب خواهد کرد و لذا به مرور شانس اجرا شدن آن افزایش یافته و از قحطی آن جلوگیری می‌شود.

بهبود الگوریتم RR

الگوریتم RR از این جهت اهمیت دارد که با چرخش نوبت میان پردازه‌ها تضمین می‌کند هیچ یک از آن‌ها دچار قحطی نخواهند شد. در عوض عملکرد این الگوریتم (از نظر میانگین زمان انتظار و اجرای پردازه‌ها) ممکن است به خوبی الگوریتم‌هایی مانند SJF نباشد. به علاوه راه حلی برای پردازه‌های با الویت بالا (و علی‌الخصوص NMI

ها) که باید پیش از پردازه‌های دیگر و به صورت یک‌تکه اجرا شوند ارائه نمی‌دهد و به عبارتی تمام پردازه‌ها را به چشم یکسان می‌بیند.

برای بهبود این مشکل پردازه‌هایی که می‌خواهند به شکل RR اجرا شوند باید یک الویت نیز داشته باشند. پس از آن اینکه چه time quantum ای برای اجرا به آن‌ها اختصاص داده شود و یا اساساً تکه تکه شوند یا خیر می‌تواند بر اساس این الویت تصمیم‌گیری شود. برای مثال می‌توان به پردازه‌هایی با الویت بالاتر زمان بیش‌تری برای هر دور اجرا اختصاص داد (می‌توان یک time quantum اولیه داشت و این مقدار بر الویت پردازه تقسیم شود. از این طریق پردازه‌های با شماره الویت کمتر و الویت بیش‌تر زمان بیش‌تری برای اجرا خواهند داشت). به علاوه می‌توان تعیین کرد که اگر الویت پردازه از مقدار خاصی بیش‌تر بود اصلاً آن را تکه‌تکه نکنیم و اجازهٔ اجرا تا به انتها را به آن بدهیم.

پیاده‌سازی این قسمت نیز با توجه به کدهای مربوط به آزمایش هشتم و توضیحات داده‌شده در این قسمت به پیوست ارسال می‌گردد. خروجی این برنامه مانند زیر خواهد بود.

```
E:\Uni\5th Semester\Operating Systems\Lab\Lab 08 - 2\OSLab08_RR with Priority.exe
Number of processes: 4
Base Time quantum: 20

Please enter the 'burst time' and 'priority' corresponding to each process:
15 4
5 2
23 1
7 5

> Process 1 runs for 5 time unit(s).
> Process 2 runs for 5 time unit(s).
> Process 3 runs for 23 time unit(s).
> Process 4 runs for 4 time unit(s).
> Process 1 runs for 5 time unit(s).
> Process 4 runs for 3 time unit(s).
> Process 1 runs for 5 time unit(s).

Process 1:
  Burst Time = 15 - Waiting Time = 35 - Turnaround Time = 50

Process 2:
  Burst Time = 5 - Waiting Time = 5 - Turnaround Time = 10

Process 3:
  Burst Time = 23 - Waiting Time = 10 - Turnaround Time = 33

Process 4:
  Burst Time = 7 - Waiting Time = 38 - Turnaround Time = 45


Average Waiting Time: 22.00
Average Turnaround Time: 34.50
```

همانطور که مشاهده می‌شود زمان اختصاص داده شده به پردازه‌ها بر حسب الویت آن‌ها بوده و برای پردازه با الویت ۱ هیچ محدودیتی قرار داده نشده‌است و لذا این پردازه توانسته در ۲۳ واحد زمانی متوالی (که بیش از base time quantum است) پردازنده را در دست بگیرد.