الگوریتم برنامه ریزی Round-Robin

مقدمه:

در این مقاله تلاش خواهیم کرد تا الگوریتم Round-Robin که در برنامه‌ریزی فرآیندهای CPU به کار می‌رود را به طور کامل توضیح دهیم. ابتدا با مفهوم برنامه‌ریزی CPU و اینکه اصلاً چرا نیاز است آشنا می‌شویم و سپس با ایده‌ی کلی الگوریتم Round-Robin آشنا می‌شویم. در ادامه تلاش می‌کنیم تا انواع پیاده سازی الگوریتم را شرح دهیم: ۱- بر مبنای FCFS ۲- بر مبنای SJF و مزایا و معایب هر کدام را نام می‌بریم. در انتها نیز مثال‌ای از چند فرآیند را بر روی FCFS اجرا می‌کنیم و آن‌ها را باهم مقایسه می‌کنیم.

برنامه‌ریزی CPU (Cpu Scheduling):

CPU یا همان پردازنده، قلب تپنده‌ی تمامی کامپیوترها می‌باشد چرا که مسئول انجام تمامی فرآیندهای کامپیوتر است. در کامپیوترها فرآیندهای زیادی نیاز دارند تا همزمان انجام شوند اما پردازنده نهایتا میتواند چند فرآیند را به صورت همزمان انجام دهد. حال فرض کنید که یک فرآیند به CPU باید داده شود اما CPU سرش شلوغ است و نمی‌تواند فرآیند را قبول کند، چه اتفاق ای برای آن فرآیند می‌افتد؟ منصرف می‌شود؟ منتظر می‌ماند؟ اگر بله، کجا؟ کی اجرا می‌شود؟ به چه ترتیب‌ای؟ جواب تمامی این سوال‌ها توسط برنامه‌ریز CPU داده می‌شود. برنامه ریز CPU مسئول برنامه‌ریزی فرآیندهای وارده به CPU است. برنامه‌ریز CPU تمامی فرآیندها را تا زمانی که به طور کامل انجام شوند نزد خود نگه می‌دارد و سپس با الگوریتم‌های متعددای آنها را به CPU می‌دهد تا انجام شوند. حال در این مقاله ما تلاش می‌کنیم تا روش(الگوریتم) Round-Robin را که یکی از بهترین و پر کاربردترین روش‌های برنامه‌ریزی CPU است را توضیح دهیم.

کلیت الگوریتم Round-Robin:

ایده‌ی کلی به این صورت است که برنامه‌ریز نمی‌گذارد تا یک فرآیند طولانی مدت CPU را درگیر کند. برای انجام این کار یک متغیر عمومی در الگوریتم وجود دارد به نام **time quantum** که مشخص می‌کند یک فرآیند نهایتا چقدر می‌تواند CPU را درگیر کند و زمانی که این زمان به پایان برسد، آن فرآیند از CPU گرفته می‌شود و دوباره تصمیم گیری می‌شود که چه پفرآیندی به CPU داده شود. این تصمیم گیری دوباره میتواند روش های متفاوتی داشته باشد که ما در این مقاله دو روش: FCFS و SJF را توضیح می‌دهیم.

رویکرد FCFS:

ابتدا باید بدانیم FCFS اصلا به چه معنا است؟ FCFS مخفف شده‌ی جمله‌ی First Come First Service است که به معنای این است که:" هرکه زودتر آمده است، زودتر سرویس بشود". حال چگونه می‌توان الگوریتم Round-Robin را با تصمیم گیری FCFS پیاده سازی کرد؟ به روش زیر توجه کنین:

- اولین عضو صف (قدیمی ترین فرآیند انجام نشده) را بردار
- آن را از صف حذف کن.
- انجامش بده تا زمانی که یا پایان یابد یا time quantum سر رسد.
- اگر فرآیند تمام نشده بود به انتهای صف اضافه اش کن.

حال اگر دقت کنین متوجه می‌شوین که هر فرآیند بعد از اینکه هربار CPU را می‌بیند به انتهای صف می‌رود. یا به بیان دیگری از آنجا که دیگر قدیمی ترین کسی که CPU را دیده است نمی‌باشد و جدید ترین کسی که CPU را دیده است می‌باشد از سر صف به انتهای صف می‌رود، پس به نوعی می‌توان گفت که فرآیندها به ترتیب آخرین باری که CPU را دیدن در صف قرار دارند. یا به نوعی دیگر به ترتیب قدیمی ترین کسانی که کارشون انجام نشده است در صف قرار دارند. یا به نوعی دیگر به ترتیب زودترین کسانی که رسیدند در صف قرار دارند. که میدانیم این همان رویکرد FCFS است.

حال داریم که الگوریتم بالا از قاعده‌ی FCFS استفاده می‌کند اما چرا از قاعده‌ی Round-Robin نیز پیروی می‌کند؟ اگر دقت کنید در مرحله سوم گفته شده است که "انجامش بده تا زمانی که یا پایان یابد یا time quantum سر رسد." که این به این معنا است که هر فرآیندی که به CPU داده می‌شود اگر کمتر از time quantum طول بکشد، تا پایان انجام می‌شود اما اگر بیشتر از time quantum طول بکشد، توسط برنامه‌ریز متوقف می‌شود و به انتهای صف فرستاده می‌شود، پس هر فرآیند نهایتا به اندازه‌ی یک time quantum در CPU می‌ماند، و این همان قاعده Round-Robin است پس قاعده‌ی Round-Robin نیز برقرار است.

حال که مطمئن شدیم روشمان تمامی خواص‌ای که می‌خواهیم را دارد، به سراغ پیاده سازی اش می‌رویم.

پیاده سازی کامل در فایل "Round-Robin-FCFS.cpp" موجود است و در ادامه جزء به جزء توضیحش می‌دهیم.

تعریف یک RoundRobin:

```
struct RoundRobin{
    int timeQuantum;
    queue <Process *> readyQueue;
};
```

در متغیر timeQuantum مقدار time quantum که قبلا درباره اش گفتیم ذخیره میشود و در متغیر readyQueue یک صف از پراسس ها که جلوتر با تعریف اشان آشنا میشویم ذخیره می‌شود. این صف همان صف ای است که فرآیندها به ترتیب آخرین باری که CPU را دیدن در آن قرار دارند که جلوتر با نحوه پر شدنش آشنا می‌شویم.

تعریف یک Process:

```
struct Process{
    int pId;
    int burstDuration;
    int burstLeft;
    int arrivalTime;
    int endingTime;
    int waitingTime;
    int timeAround;
    int responseTime;
    bool seen = false;
};
```

در Process تمامی اطلاعات مربوط به یک فرآیند قرار می‌گیرد.

- pId: id هر فرآیند (عددی که در هر فرآیند متفاوت است و فرآیندها با آن از یکدیگر مجزا می‌شوند.)
- burstDuration: مدت زمانی که طول می‌کشد تا یک فرآیند انجام شود.
- burstLeft: مدت زمان باقی‌مانده از فرآیند.
- arrivalTime: زمانی که درخواست فرآیند ارسال می‌شود.

- endingTime: زمانی که فرآیند تمام می‌شود.
- waitingTime: مدت زمانی که فرآیند در صف باقی می‌ماند.
- timeAround: فاصله زمانی ارسال درخواست فرآیند تا پایان آن.
- responseTime: فاصله زمانی ارسال درخواست تا اولین باری که CPU را می‌بیند.
- Seen: آیا تا به کنون این فرآیند توسط CPU دیده شده است یا خیر که چون در ابتدا قطعا دیده نشده است مقدار false گرفته است.

ارزش دهی به فرآیندها:

```
bool operator<(const Process &process1, const Process &process2){
    return process1.arrivalTime < process2.arrivalTime;
}
```

در اینجا برای فرآیندها یک تعریف کوچکتر بزرگتر ای تعریف می‌کنیم. به این صورت که هر فرآیندی که arrivalTime زودتر ای داشت باشد، کوچکتر است. حال چرا اینگونه تعریف می‌کنیم؟ چون که جلوتر وقتی می‌خواهیم هر فرآیند را در زمان آمدن درخواست اش به صف Round-Robin اضافه کنیم، اگر داشته باشیم که درخواست ها به چه ترتیب ای آمده اند خیلی راحت تر و سریع تر می‌توانیم عمل کنیم. حال برای اینکه بتوانیم فرآیندها را بر حسب زمان درخواست اشان مرتب کنیم باید به cpp بگوییم که هر فرآیند بر حسب زمان درخواستش سنجیده می‌شود که این هم ارز با این است که هر که arrivalTime کوچکتر ای داشته باشد کوچکتر است که این همان تابع بالا می‌باشد.

حال از اینجا به بعد تعاریف امان کم می‌شوند و کم کم شروع به ورودی گرفتن می‌کنیم.

ساخت یک Round-Robin:

```
RoundRobin RR;
cout << "Determine time quantum:";
cin >> RR.timeQuantum;
```

در اینجا یک Round-Robin میسازیم و مقدار time quantum را از کاربر می‌خواهیم تا برای ما تئیین کند.

ورودی گرفتن اطلاعات فرآیندها:

```
int n;
cout << "Enter #of processes :";
cin >> n;
Process * Processes = new Process[n];
for(int i = 0; i < n; i++){
    cout << "Enter process id:";
    cin >> Processes[i].pId;
    cout << "Enter arrival time:";
    cin >> Processes[i].arrivalTime;
    cout << "Enter Burst duration:";
    cin >> Processes[i].burstDuration;
    Processes[i].burstLeft = Processes[i].burstDuration;
}
```

ابتدا برای تعاریف داریم که:

- n: تعداد فرآیندها.
- Processes: یک آرایه به طول n از فرآیندها برای نگه داری تمامی فرآیندها در یک جا.

حال بدین صورت عمل می‌کنیم که ابتدا مقدار n را از کاربر وروردی می‌گیریم و سپس آرایه‌ی Processes را تعریف می‌کنیم و سپس عضو به عضو مقادیر arrivalTime و burstDuration را از کاربر وروردی می‌گیریم و چون فرآیند تا به حال انجام نشده است میدانیم که مقدار burstLeft با مقدار burstDuration در حال حاظر برابر است و مقدار آن را هم مشخص می‌کنیم. اما باقی اطلاعات مربوط به هر فرآیند وابسطه به این می‌باشد که ما به چه ترتیب ای فرآیندها را انجام می‌دهیم پس در حال حاظر قادر به مقدار دهی به آنها نمی‌باشیم.

مرتب سازی فرآیندها:

```
sort(Processes, Processes + n);
```

در اینجا فرآیندها را بر حسب arrivalTime مرتب می‌کنیم چرا که در بالاتر دیدیم ارزش هر فرآیند را برابر با زمان رسیدن اش قرار داده ایم. پس بعد از مرتب سازی صعودی، زودترین درخواست در ابتدای آرایه می‌باشد و دیرترین درخواست در انتهای آرایه قرار می‌گیرد.

حال از اینجا به بعد ورودی گرفتنمان هم پایان می‌یابد و شروع به برنامه ریزی فرآیند ها می‌کنیم.

شروع حلقه:

```
int Time = 0;
int pnt = 0;
Process* curProcess = NULL;
int curProcessTimeLeft = 0;
vector <int> log;
while(pnt < n || curProcess != NULL || !RR.readyQueue.empty()){
```

ابتدا برای تعاریف داریم که:

- Time: زمان در هر دور از حلقه
- Pnt: اندیس اولین عضو آرایه که هنوز درخواست اش داده نشده است.
- curProcess: فرآیندی که هم اکنون توسط CPU در حال اجرا است.
- curProcessTimeLeft: مقدار زمان باقی مانده از time quantum فرآیندی که در حال حاظر در حال اجرا است.
- Log: آرایه ای با اندازه‌ی متغیر که در آن ریخته میشود که در هر ثانیه چه فرآیندی در حال اجرا می‌باشد.

حال در شرط حلقه داریم که این حلقه ادامه پیدا میکند تا زمانی که حداقل یکی از شروط زیر برقرار باشد:

- Pnt < n یا به این معنی که هنوز درخواست ای باشد که داده نشده است.
- curProcess != NULL یا به این معنی که فرآیندی بر روی CPU در حال اجرا باشد
- !RR.readyQueue.empty() یا به این معنی که همچنان فرآیندی در صف امان باقی مانده است.

در کل هر ۳ شرط به اینکه هنوز فرآیندی برای انجام شدن وجود دارد یا خیر اشاره می‌کند و هر ۳ نیاز است چرا که یک فرآیند میتواند در هر یک از این ۳ جا باشد و از جاهای دیگر نتوان متوجه حضورش شد، پس برای اینکه مطمئن شویم دیگر فرآیندی برای انجام شدن وجود ندارد باید هر ۳ را بگردیم.

افزودن درخواست‌های آمده:

```
while(pnt < n && Processes[pnt].arrivalTime <= Time){
    RR.readyQueue.push(&Processes[pnt]);
    pnt ++;
}
```

به این صورت عمل می‌کنیم که تمامی فرآیندهایی که زمان آمدنشان از زمان حال کمتر مساوی است را به صف Round-Robin امان اضافه میکنیم و چون داریم که هر فرآیندی که اضافه می‌شود به انتهای صف می‌رود، به انتهای صف اضافه اش می‌کنیم. و با هر بار اضافه کردن یک فرآیند، اندیس آخرین اضافه شده را نیز افزایش می‌دهیم. و از طرفی دیگر مراقب هستیم که عضوی از آرایه که وجود ندارد را مقایسه نکنیم (pnt < n).

قرار دادن فرآیند جدید روی CPU در صورت خالی بودن CPU:

```
if(curProcess == NULL && !RR.readyQueue.empty()){
    curProcess = RR.readyQueue.front();
    curProcessTimeLeft = RR.timeQuantum;
    RR.readyQueue.pop();
    if(!(*curProcess).seen){
        (*curProcess).seen = true;
        (*curProcess).responseTime = Time - (*curProcess).arrivalTime;
    }
}
```

داریم که فرآیندی که بر روی CPU می‌باشد داخل متغیر curProcess قرار دارد، پس اگر curProcess = NULL باشد، داریم که CPU بیکار شده است و باید فرآیندی بر رویش قرار بدهیم. از طرفی دیگر فرآیندی که باید قرار بگیرد اولین عضو صف Round-Robin امان می‌باشد، پس صف نباید خالی باشد تا عضوی برای قرار دادن بر روی CPU وجود داشته باشد. حال در داخل شرط ابتدا عضو اول را به curProcess می‌دهیم و سپس curProcessTimeLeft اش را که همان quantum time است را مقدار دهی می‌کنیم و در انتها آن عضو را از صف حذف می‌کنیم. سپس نگاه می‌کنیم که آیا این فرآیند تا به حال CPU را دیده است یا خیر، اگر ندیده باشد مقدار responseTime اش مشخص می‌شود و باید ان را مقدار دهیم. مقدار responseTime برابر میشود با Time - arrivalTime چرا که مدت زمان آمدن درخواست تا اولین دیدن CPU دقیقا برابر است با زمان آمدن درخواست تا زمان حال که اولین دیدن CPU توسط فرآیند می‌باشد.

انجام تغییرات یک واحد زمان بر روی curProcess:

```
if(curProcess != NULL){
    (*curProcess).burstLeft --;
    curProcessTimeLeft --;
    log.push_back((*curProcess).pId);
}
else{
    log.push_back(-1);
}
Time ++;
```

اگر فرآیندی بر روی CPU وجود داشته باشد بعد از یک واحد زمان از مدت زمان اجرا باقی مانده اش یکی کم می‌شود و به همچنین از مدت زمان مجازش برای استفاده از CPU نیز یکی کم می‌شود و در log داریم که اندیس فرآیند حال حاضرمان اضافه می‌شود. از طرفی دیگر برای اینکه بتوانیم مشخص کنیم چه زمان هایی CPU کاری برای انجام دادن نداشته است، در آن زمان ها در log مقدار منفی یک می‌ریزیم.(یک مقداری که قطعا اندیس هیچ فرآیندی نیست.) و چه فرآیندی داشته باشیم چه نداشته باشیم قطعا زمان یک واحد می‌گذرد.

تعویض فرآیندهای پایان یافته / حد مجاز زمان گذشته:

```
if(curProcess != NULL && (*curProcess).burstLeft == 0){
    (*curProcess).endingTime = Time;
    curProcess = NULL;
    curProcessTimeLeft = 0;

}
else if(curProcessTimeLeft == 0 && curProcess != NULL){
    RR.readyQueue.push(curProcess);
    curProcess = NULL;
    curProcessTimeLeft = 0;
}
```

حال اگر فرآیند حال حاظر خاتمه یافته باشد، دیگر زمان اجرایی برایش باقی نمانده است و burstleft=0 میشود. در این صورت endingTime فرآیند امان مشخص می‌شود که همان زمان حال حاضر می‌باشد.

از طرفی دیگر چون فرآیند خاتمه یافته است دیگر CPU باید بیکار شود و این به آن معنا است که دیگر فرآیند پایان یافته روی CPU نباشد که این به آن معناست که فرآیند پایان یافته را از روی curProcess پاک کنیم که این هم ارز با NULL قرار دادن مقدار curProcess است. و چون فرآیندی باقی نمانده است پس زمان مجاز باقی مانده‌ای هم نمی‌ماند.

از طرفی دیگر اگر فرآیند پایان نیابد اما زمان مجاز مصرف CPU اش پایان یابد، ابتدا باید به انتهای صف برود و سپس باید از روی CPU برداشته شود که داریم این همان NULL قرار دادن curProcess و صفر کردن curProcessTimeLeft می‌باشد.

حال اینجا حلقه پایان می‌یابد و تمام کار های داخل حلقه دوباره اجرا می‌شوند تا زمانی که دیگر فرآیندی باقی نماند.

تکمیل کردن داده های مربوط به هر فرآیند:

```
for(int i = 0; i < n; i++){
    Processes[i].waitingTime = Processes[i].endingTime -
                               Processes[i].burstDuration -
                               Processes[i].arrivalTime;
    Processes[i].timeAround = Processes[i].endingTime -
                              Processes[i].arrivalTime;
}
```

داریم که تنها داده های پر نشده، waitingTime و timeAround می‌باشند. حال برای waitingTime داریم که هر مدت زمانی که در صف بوده است و در حال اجرا نبوده است، منتظر بوده است. حال داریم که مدت زمانی که در صف بوده است برابر است با زمانی که درخواست اش ارسال شده است تا زمانی که تمامی اجرایش انجام شده است. از طرفی دیگر مجموع تمام مدت ای که در حال اجرا بوده است برابر است با burstDuration پس داریم که:

waitingTime = (endingTime - arrivalTime) - burstDuration

حال برای timeAround هم داریم که برابر است با مدت زمان ارسال درخواست تا پایان یافتن فرآیند که برابر است با مدت زمانی که فرآیند در صف بوده است که از بالا داریم که برابر است با:

endingTime-arrivalTime

پس داریم که:

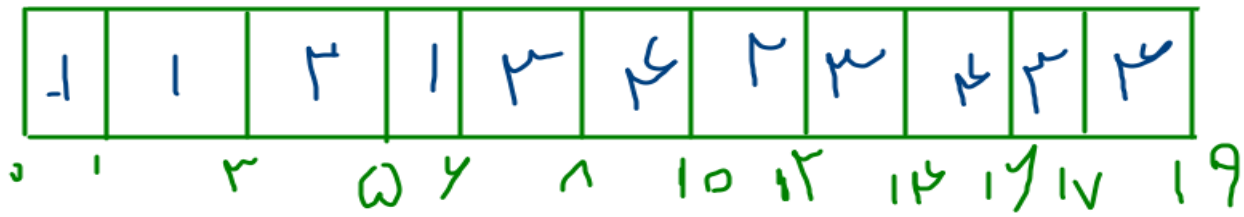
timeAround = endingTime—arrivalTime

الگوریتم در اینجا پایان می‌یابد و در ادامه به تحلیل راه حل می‌پردازیم.

حال فرض کنید فرآیندهای ورودی امان به صورت زیر باشند:

pId	burstDuration	arrivalTime
۱	۳	۱
۲	۴	۲
۳	۵	۲
۴	۶	۴

حال اگر با quantum time برابر با ۲ برنامه را اجرا کنیم برای هر فرآیند داریم که:



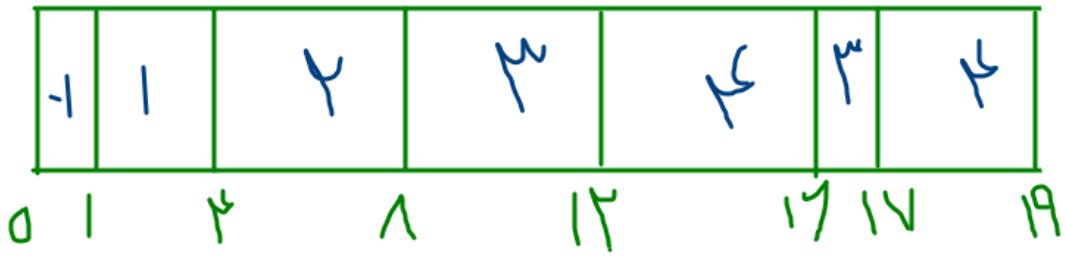
pId	endingTime	waitingTime	Timearound	responseTime
۱	۶	۲	۵	۰
۲	۱۲	۶	۱۰	۱
۳	۱۷	۹	۱۴	۳
۴	۱۹	۹	۱۵	۴

و برای میانگین waitingTime وaroundTime هم داریم که:

waitingTime avg. = 6.5

responseTime avg. = 2

و اگر با quantum time برابر با ۴ برنامه را اجرا کنیم برای هر فرآیند داریم که:



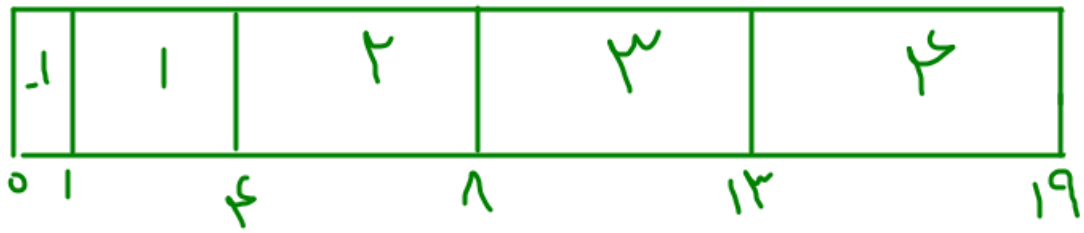
pId	endingTime	waitingTime	Timearound	responseTime
۱	۴	۰	۳	۰
۲	۸	۲	۶	۲
۳	۱۷	۹	۱۴	۵
۴	۱۹	۹	۱۵	۸

و برای میانگین waitingTime وaroundTime هم داریم که:

waitingTime avg. = 5

responseTime avg. = 3.75

و در آخر اگر با quantum time برابر با ۶ برنامه را اجرا کنیم برای هر فرآیند داریم که:



pId	endingTime	waitingTime	Timearound	responseTime
۱	۴	۰	۳	۰
۲	۸	۲	۶	۲
۳	۱۳	۵	۱۰	۵
۴	۱۹	۹	۱۵	۹

و برای میانگین waitingTime و aroundTime هم داریم که:

waitingTime avg. = 4

responseTime avg. = 4

حال با استفاده از تحلیل‌های آماری می‌توان به نتایج مهمی رسید، به طور مثال می‌توان دید که هر چه time quantum افزایش می‌یابد responseTime ها هم افزایش می‌یابند و در نتیجه میانگین responseTime ها هم افزایش می‌یابد. حال اثبات اینکه با کاهش time quantum کاهش responseTime هم داریم نسبتا ساده است اما بعضی از نتایج که در ادامه می‌گوییم اثبات اشان نیاز به دانش آماری دارد و از اثبات آنها صرف نظر می‌کنیم. حال چرا با کاهش time quantum کاهش

responseTime را هم داریم، چرا که وقتی time quantum کم باشد فرآیندها زود به زود عوض می‌شوند و در نتیجه هر کسی که به انتهای صف برای بار اول وارد می‌شود مدت زمان کمتری را باید صبر کند تا افراد جلوتر اش به انتهای صف بروند و در نتیجه زودتر برای بار اول به CPU می‌رسد.

حال باهم برخی از نتایج ای که در مورد مقدار دهی time quatum وجود دارد را ببینیم:

- هر چه time quantum کمتر باشد، میانگین responseTime کاهش می‌یابد.
- بهترین مقدار برای time quantum زمانی است که تقریبا از ۸۰ درصد burstTime ها بیشتر باشد. (بهینه بودن از لحاظ تعامل میان waitingTime و responseTime است)
- اگر time quantum از تمامی burstTime ها بیشتر باشد، دیگر شرط Round-Robin معنی نخواهد داشت و الگوریتم تنها یک FCFS ساده می‌شود.

رویکرد SJF :

در رویکرد FCFS تمامی فرآیندها تنها اولویتی که نسبت به هم داشتن زمان رسیدنشان بود. اما در یک کامپیوتر واقعی ممکن است بعضی مواقع فرآیندی درخواست داده شود که نتواند منتظر بماند و به سرعت باید اجرا بشود و از طرفی دیگر ممکن است فرآیندی باشد که اصلا مهم نباشد و بتواند منتظر بماند. حال در رویکرد قبلی داشتیم که به راحتی ممکن است یک فرآیند با اولویت منتظر یک فرآیند کم اولویت بماند در حالی که اگر زودتر انجام می‌شد بازده کامپیوتر ما بیشتر می‌شد. حال همیشه فقط دو اولویت نداریم و در کامپیوتر های امروزی معمولا بیش از ۱۰۰ اولویت متفاوت داریم. زمانی که تعداد اولویت ها انقدر زیاد است دیگر نمی‌توان با شروط تو در تو حالت‌های خاص را پیاده سازی کرد و نیاز داریم که الگوریتمی جدا ارائه دهیم.

یکی از این اولویت دهی‌ها بر حسب طول burstDuration است. به این منظور که ابتدا کار های کوتاه تر را انجام می‌دهیم و سپس به سراغ کار های طولانی تر می‌رویم. این الگوریتم به الگوریتم Shortest Job First یا به اختصار SJF معروف است. هدف از انجام این کار بهینه کردن waitingTime است. اما این کار چطور باعث کمینه شدن میزان waitingTime می‌شود؟ فرض کنید تمام فرآیندها انجام شده باشند، اولین فرآیندی که پایان می‌یابد درون waitingTime تمامی فرآیندهای بعدی کل burstTime اش میاید.

پس داریم که اولین فرآیندی که خاتمه می‌یابد در n-1 تا از waitingTime ها میاید، دومین فرآیند در n-2 و به همین ترتیب i امین فرآیند در n-i-1 فرآیند دیگر میاید، پس هرچه فرآیندها به ترتیب صعودی در طول باشند مجموع تمام waintingTime ها کاهش می‌یابد و در نتیجه میانگین waitingTime ها کاهش میابد و هر چه میانگین waitingTime ها پایین تر باشد داریم که بازده کامپیوتر بیشتر است. از طرفی دیگر هر موقع که یک فرآیند دارد صبر می‌کند تا فرآیند دیگری انجام شود یا فرآیندها در حال جابجایی اند یا فرآیند دیگری دارد انجام می‌شود. حال اگر زمان جابجایی میان فرآیندها را نادیده بگیرم، داریم که مجموع تمام waitingTime ها دقیقا برابر عبارت بالا است و ترتیب صعودی بهترین جواب برای کمینه میانگین waitingTime است. حال روش SJF چه ربطی به Round-Robin دارد؟ مساله ای که وجود دارد این است که کوتاه ترین زمان ها لزوماً یکتا نیستند و میان آنها چگونه باید انتخاب کرد؟ الگوریتم زیر را در نظر بگیرید:

- به ازای هر طول یک صف بساز.
- فرآیندها را به صف متناظر با طول اشان اضافه کن.
- کار های بعدی را تا زمانی که فرآیندی باقی مانده است انجام بده:
- از اول صف کوتاه ترین طول که خالی نیست یک فرآیند بردار.
- از صف اش خارج اش کن.
- اجرایش کن تا زمانی که یا تمام شود یا time quantum اش سر برسد.
- اگر تمام نشده بود به انتهای صف طول اجرایش اضافه‌اش کن.

حال الگوریتم بالا قطعا در هر مرحله یکی از کوتاه ترین فرآیندها را بر می‌دارد پس طبق اصول SJF کار می‌کند و از طرفی دیگر هر فرآیند را حداکثر به طول time quantum انجام می‌دهد و سپس در آن صف، آنرا در انتها می‌گذارد و دوباره از اول عمل می‌کند. به نوعی داریم که در صف هر طول مدت یک Round-Robin با رویکرد FCFS قرار دارد. پس می‌توان گفت که هم از قواعد SJF پیروی می‌کنیم هم از قواعد Round-Robin.

از طرفی دیگر به این روش صفت non-preemptive هم می‌دهند چرا که تنها در زمانی که یا اجرا تمام شود یا محدودیت زمانی سر برسد فرآیندش را عوض می‌کند، اگر هر وقت که فرآیند با اولویت بالاتری وارد شد، اجرا حال حاضر را قطع کند و آن را اجرا کند، از رویکرد preemptive بهره برده ایم.

حال از پیاده سازی و تحلیل این روش صرف نظر می‌کنیم و مستقیم به سراغ مقایسه این دو روش با هم می‌رویم. در مقایسه تلاش می‌کنیم بیشتر روی تفاوت های این دو روش تمرکز کنیم و در نتیجه گیری به مزایا و معایب مشترک اشان میپردازیم.

مقایسه دو رویکرد FCFS و SJF:

مزایای FCFS به SJF:

- پیاده سازی سریع
- تاثیر بیشتر مقدار time quantum برای موازنه بین waitingTime و responseTime وابسته به نیاز کامپیوتر (ریسپانس دهی بهتر و عملکرد کلی بهتر)
- پیدا کردن فرآیند بعدی با سرعت بیشتر
- لزوما نیازی نیست burstDuration ها را از اول داشته باشیم

مزایای SJF به FCFS:

- بهینه بودن در میانگین waitingTime
- قابلیت بهتر کردن responseTime تا حدودی توسط time quantum
- قابلیت اولویت دادن به فرآیندها (قرار دادن اولویت به جای burstDuration)

نتیجه گیری:

در این مقاله به تفکیک با انواع روش استفاده از الگوریتم Round-Robin آشنا شدیم. الگوریتم Round-Robin در کل زمانی بیشتر استفاده می‌شود که responseTime مهم باشد مانند سیستم های real-time یا سیستم هایی که intrupt (وقفه)های زیادی دارند. پس اگر دارید برای سیستم ای که قرار است فرآیندهای سنگینی را اجرا کند برنامه‌ریز برنامه نویسی می‌کنین این روش اصلا پیشنهاد نمی‌شود اما اگر برای کامپیوتری برنامه نویسی می‌کنین که قرار است ساختارهای زیادی داشته باشد و این ساختار ها به هم دیگر نیاز زیادی داشته باشند و یا قرار است با کاربر ارتباط زیادی داشته باشد این الگوریتم را به شما پیشنهاد می‌دهیم. و انتخاب اینکه از کدام رویکرد استفاده کنین وابسته به ملزومات کامپیوترتان است و از روی مزایا و معایب آن‌ها به راحتی میتوانین رویکرد مد نظرتان را انتخاب کنین.

اشکان زرخواه

۶۱۰۳۹۹۱۹۶