Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department

Zeinab Zali

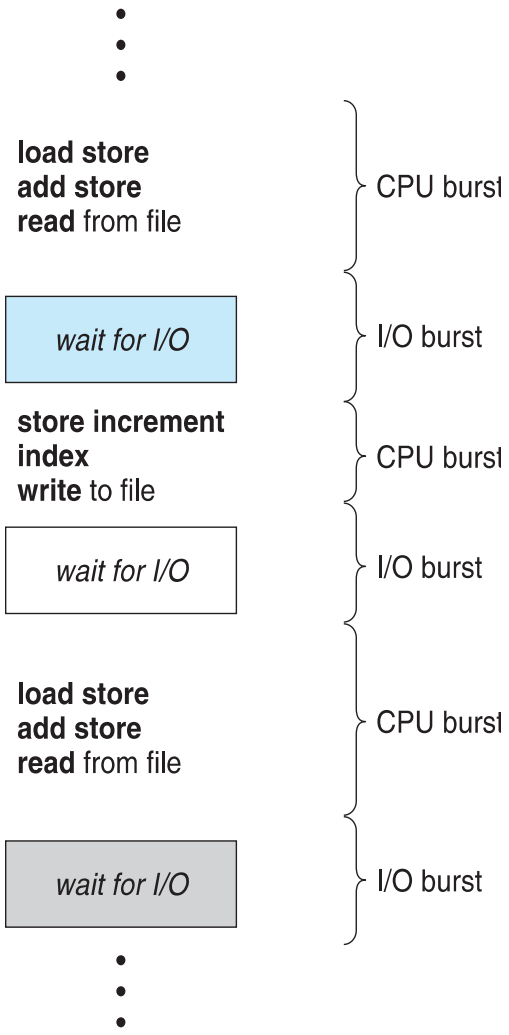
CPU Scheduling

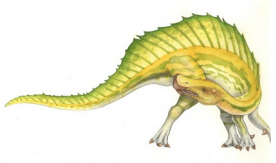
برنامه ریزی CPU

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern

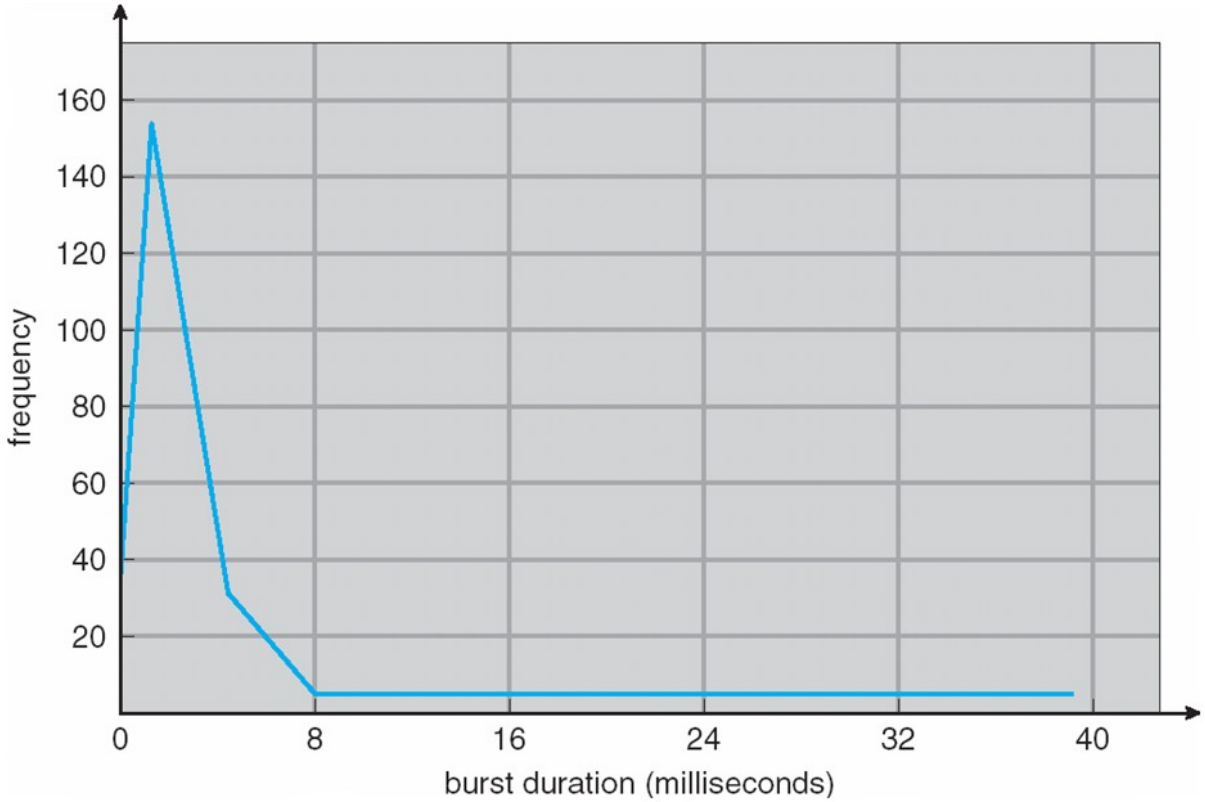
IO-bound and CPU-bound





-
توی اینجا می خوایم بگیم اگر از cpu در حد نهایت توانش ازش استفاده بکنیم ینی نزدیک 100 درصد مال وقتی است که multiprogramming داشته باشیم یا سیستممون مالتی تسک باشه و ما بخوایم همه اون پروسس هارو همزمان توی سیستم داشته باشیم
اگر برنامه cpu bound اکثر burst هاش burst های بلند cpu هستند ینی همینطور داره cpu استفاده میکنه و به ندرت به I/O نیاز داره یه همچین برنامه خودش هم به تنهایی می تونه CPU utilization رو ببره بالا
معمولا سیستم هایی که توی کامپیوتر داریم مالتی تسک هستند و برنامه هایی نداریم که خیلی Cpu رو مصرف کنند و بعضی اوقات میشه که همچین برنامه ای داشته باشیم ولی گاهی اوقات نداریم
برنامه هایی معمول به این صورت اند که : یک burst سی پی یو دارند و cpu burst ینی زمان متوالی که cpu رو می تونن استفاده بکنند ینی عملیاتی که دارند انجام میدن عملیاتی است که cpu توش استفاده میشه : شکل روبه رو
I/O burst ینی توی اون تیکه نیازه بره عملیات I/O رو انجام بده و وقتی که برگشت دوباره می تونه بره cpu رو استفاده بکنه
ممکنه I/O burst ها هم متفاوت باشه ینی ممکنه چندتا خط I/O پشت سر هم داشته باشیم
حالا اگر برنامه ای که داریم به صورت شکل روبه رو است و توی اون زمان هایی که داریم I/O انجام میدیم cpu چی کار بکنه؟ و باید بیکار بمونه؟ خیر اینجا می تونیم cpu رو استفاده بکنیم برای یک پروسس دیگه و این کار باعث میشه که cpu توی همه زمان ها استفاده بکنیم
مسئله مهمی که داریم اینجا اینه که CPU burst ها چقدر هستند برای هر برنامه مشخصی

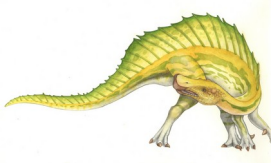
Histogram of CPU-burst Times



-
اینجا توی این نمودار میگه که:
اکثر burstهای پروسس های معمول توی 5 این هاست ینی durationشون خیلی کوتاهه و اون هایی که تعداد burstشون طولانی تره خیلی تعدادشون کمه پس همون چیزی است که صفحه قبل گفتیم ینی burst سی پی یوهامون خیلی طولانی نیست و هی وسطش I/O داریم و از همین واقعیتی که وجود داره می تونه استتفاده بکنیم اسکجولینگ سیستم ینی بیایم زمان هایی که یک پروسس burst سی پی یوش تموم شده و منتظر I/O بیایم cpu رو بدیم به یک پروسس دیگه ای و اون که burstاش تموم شد بدیمش به یک cpu دیگه ای که این یک حالت است و حتی می تونیم توی اسکجولینگ یک کار دیگه هم انجام بدیم ینی اینطور نیست که اگر burst سی پی یو یک پروسس تموم شد cpu رو ازش بگیریم ممکنه که هنوز پروسس نیاز به cpu داشته باشه ولی ازش می گیریم

CPU Scheduler

- **Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them**
  - **Queue may be ordered in various ways**
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive** قبضه نشدنی یا انحصاری
- All other scheduling is **preemptive** قبضه شدنی یا غیرانحصاری
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities



-
پس یک Short-term scheduler داریم که قراره بین پروسس هایی که توی صف ready قرار دارند انتخاب بکنه که cpu رو به کدومش بده
خب الان بر چه اساسی انتخاب بکنیم که به کدوم پروسس cpu بدیم؟ این سوالی هست که اسکجولینگ باید جوابشو بده ینی اسکجولینگ باید انتخاب بکنه که در هر لحظه cpu رو به کدوم پروسس بده
توی زمان های خاصی ممکنه که CPU scheduling فعال بشه و تصمیم بگیره که cpu رو به کی بده:
1- وقتی که میخوایم یک برنامه رو از حالت running to watting ینی تصمیم میگیره کدوم برنامه رو به جای این برنامه قرار بده این حالت ینی داشته cpu استفاده میکرده و خورده به I/O
2 و 3 رو از روی اسلاید ببین
4- یک موقعی هم هست که یک پروسس اجراش تموم شده

این 4 تا حالت بالا یه فرق هایی با هم دارند: توی حالت 1 برنامه ما به دلخواه خودش رفته توی حالت watting و کسی cpu رو به زور ازش نگرفته و توی حالت 4 هم همینطور برنامه تموم شده و cpu رو می تونیم بدیم به یک برنامه دیگه پس 1 و 4 حالت هایی هستند که یک پروسس به صورت دلخواه و با اختیار cpu رو ازاد میکنه ولی حالت های 2 و 3 حالت هایی هستند که برنامه ای در حال اجرا است ولی اسکجولر تصمیم میگیره که cpu ازش گرفته بشه پس به حالت 1 و 4 میگیم nonpreemptive ینی cpu در اختیار اون پروسس بوده و ما به زور نگرفتیمش ینی قبضه اش نکردیم و به حالت 2و3 میگیم preemptive ینی اسکجولر تصمیم گرفته که cpu ازش گرفته بشه و به یک کس دیگه ای داده بشه چون cpu انحصاری نیست اینجا و غیر انحصاری است

اگر preemptive باشه ما یکسری مسائلی هم خواهیم داشت:

1- اینجا ممکنه اون پروسس اول در حال استفاده از یک shared data باشه و اگر وسط کار cpu رو ازش بگیریم ممکنه که اون دیتا به صورت نادرستی تغییر بکنه یا هنوز تغییراتش تموم نشده و پروسس بعدی هم از اون دیتا استفاده بکنه و اینجا مشکل پیش بیاد

2- توی حالت کرنل مد هم ممکنه مشکل پیش بیاد مثلا توی حالت کرنل مد هستیم و یک عملیاتی داره توی سطح کرنل انجام میشه و یکسری دیتا استراکچرهای کرنل تغییر میکنه و ما وسطش می خوایم cpu رو بگیریم و بدیم به کس دیگه ای و توی اون پروسس دوم ممکنه مشکل ایجاد بشه چون دیتا استراکچرهای کرنل توی پروسس اول به درستی اپدیت نشده بودند

3- یا مثلا هر فعالیت خاص سیستم عامل که وسطش یک وقفه ای اتفاق بیوفته ممکنه یک مشکلی واسه اون برنامه ای که توی cpu در حال اجرا بوده پیش بیاد

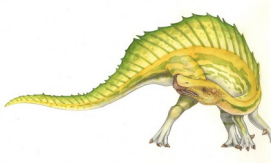
پس توی این حالت دوم باید حواسمون رو جمع بکنیم و کد نویسی به صورتی باشه که مشکلی پیش نیاد که این رو توی بحث همزمانی می گیم

Dispatcher

- **Dispatcher** module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

vmstat 1 3

cat /proc/pid/status



-
وقتی که اسکجولر تصمیم میگیره که cpu رو از یکی بگیره و به یکی دیگه بده یا ... اینجا یک ماژول دیگه هم باید کار بکنه به اسم Dispatcher
Dispatcher چیه؟ Dispatcher کنترل cpu رو از پروسس اول می گیره و میده به پروسس بعدی ینی همون پروسسی که الان اسکجولر انتخابش کرده
پس اسکجولر انتخاب میکنه که چه پروسسی الان باید cpu دستش باشه و Dispatcher میاد این جابه جایی cpu رو انجام میده که این Dispatcher شامل یه سری چیزا میشه که توی اسلاید نوشته
تاخیر اجرای این Dispatcher چقدره مهم میشه ینی اگر بخوایم بین دوتا پروسس سوییچ کنیم این زمان انتقال cpu بین دو تا پروسس که وجود داره و Dispatcher داره اجرا میشه این زمان باید زمان کوتاهی باشه پس باید این Dispatcher ما latency پایینی داشته باشه
دستور vmstat : یکسری اطلاعاتی در مورد ویرچوال ممووری میده و هم در مورد مقدار cpu که داره استفاده میشه و هم context switching ینی چند بار context switching کرده این پروسس یا ...
دستور cat/proc/ipd/status یک همچین فایلی رو برای هر پروسسی داریم که توش می تونیم یکسری اطلاعات ببینیم که یکیش اینه که چند بار این پروسس خاص context switching کرده

Scheduling Criteria

- **CPU utilization :** keep the CPU as busy as possible درصد مصرف پردازنده
- **Throughput:** number of processes that complete their execution per time unit بازده
- **Turnaround time:** amount of time to execute a particular process زمان برگشت
- **Waiting time:** amount of time a process has been waiting in the ready queue زمان انتظار
- **Response time:** amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment) زمان پاسخ
- **Fairness:** give each process a fair share of CPU عدالت
- **Overhead:** computation resource required for implementing the scheduling algorithm سربار

-
یکسری متریک هایی میگیم که توی بحث اسکجولینگ ازشون استفاده میکنیم برای اینکه مشخص کنیم اون الگوریتم اسکجولینگی که داریم ارائه میکنیم چقدر خوبه، چه کارایی داره و کجا می شه ازش استفاده کرد:
CPU utilization : ما دوست داریم تا جایی که می تونیم cpu رو مشغول نگه داریم ینی چقدر Cpu فعال بوده
Throughpu : توی یک سیستمی که مالتی تسک است می تونیم تعداد پروسس هایی که در یک واحد زمانی تموم میشه رو به عنوان Throughput در نظر بگیریم
Turnaround time : زمانیه که از شروع استارت یک پروسس تا پایانش نیاز داریم مثلا پروسس p0 از زمان 10 شروع شده و تا زمان 200 رفته و تموم شده و در این زمان طی شده یکسری مواقع cpu رو بهش داریم و یکسری مواقع هم ازش گرفتیم پس در حالت کلی 10-200 میشه Turnaround time ینی کل زمانی که توی سیستم وجود داشته با وجود اینکه یکسری جاها بهش cpu ندادیم
Waiting time : ینی یک پروسسی که توی سیستم بوده از ابتدا تا اخری که تموم بشه توی چه زمان هایی cpu بهش ندادیم و توی حالت wait بوده
Response time : ینی زمانی هست که انتظار می کشه تا برای اولین بار cpu رو بگیره و بتونه شروع به کار بکنه که این معیار خیلی مهمی است توی بحث اسکجولینگ
Fairness : ینی ما یک cpu یا هر تعداد cpu که داریم چه سهمی از این cpu رو به بقیه پروسس ها میدیم اگر سهمی که میدیم یکسان باشه می گیم این عدالت است (ینی توی اینجا منظور اینه که یک سهمی عادلانه ای از cpu به پروسس های مختلف بدیم ینی سهمی که متناسب با نیازهاشون و نوع اون پروسس باشه)
Overhead : چقدر زمان می بره که این الگوریتم اجرا بشه پس میخوایم Overhead مون پایین باشه

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
- Max Fairness
- Min Overhead

-
ما برای الگوریتم های اسکجولینگ دنبال الگوریتم هایی هستیم که ویژگی های صفحه روبه رو داشته باشند

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3

-
Burst Time همان cpu burst است مثلا الان پروسس p1 ما cpu burst اش 24 واحد زمانی است
می خوایم تصمیم بگیریم به یکییشون cpu بدیم اگر از روش FCFS بریم ینی کی اول اومد؟
مثل اگر p1 اول وارد سیستم شد اول به p1 مون cpu میدیم و بعد p2 و بعد p3
FCFS ینی اونی که اول اومد رو اول بهش سرویس میدیم
حالا می خوایم براساس اون متریک هایی که گفتیم بریم جلو... صفحه بعدی...

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

نمودار گانت برای برنامه به شرح زیر است:

...

زمان انتظار برای P1 = 0; P2 = 24; P3 = 27

میانگین زمان انتظار: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

-
توی این حالت ترتیب ورودها فرق میکنه اول p2 اومده بعد p3 و بعد p1

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

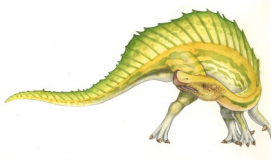
-
اینجا که ترتیب ورود عوض شده Average waiting time ما کمتر شده پس نتیجه میگیریم که این الگوریتم به شدت وابسته است به ترتیب ورود پروسس ها

اتفاقی که می افته اینجا اینه که: وقتی هست که ما یک پروسس خیلی بزرگی داریم و بعدش یکسری پروسس های کوچکی داریم و پروسس های کوچک مجبور میشن که منتظر اتمام اون پروسس بزرگه بمونن این اتفاقی که می افته رو بهش میگیم تاثیر Convoy
کی این اتفاق می افته ینی Convoy می افته؟ وقتی که یک پروسس cpu bound داریم و خیلی پروسس کوچک I/O bound در این صورت اگر cpu bound زودتر برسه I/O bound ها خیلی باید انتظار بکشن

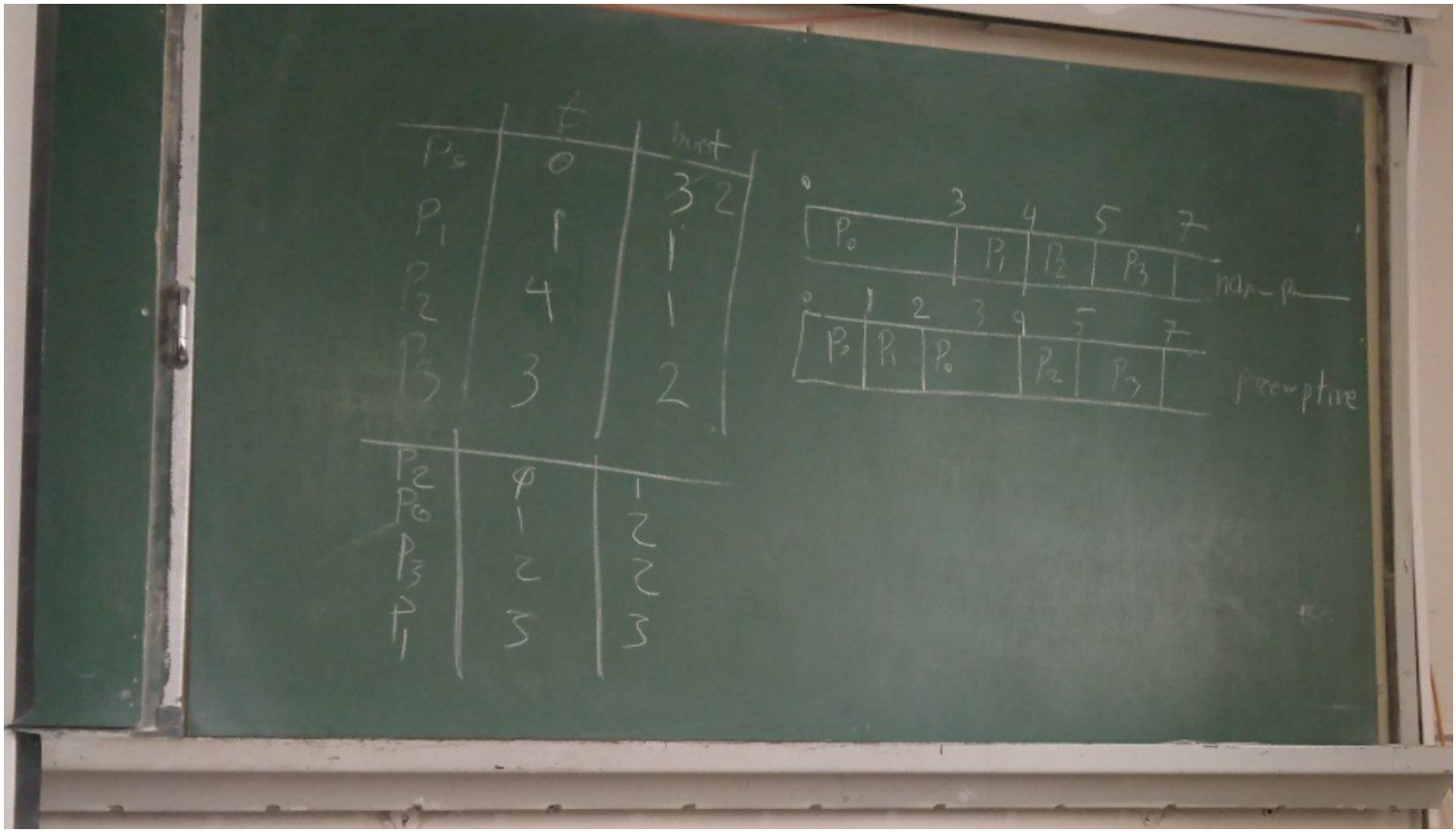
FCFS Scheduling (Example)

پروسس	زمان ورود	زمان پردازش
A	0	3
B	1	3
C	4	3
D	6	2

Waiting_time=(0+2+2+3)/4=7/4



یک مثال دیگه هم سر کلاس زد که عکسش پایین است:
نکته nonpreemptive و preemptive توش ببین



FCFS Properties

- Non-preemptive
- No starvation
- More suitable for short jobs compared to long ones
- The least overhead

-
ویژگی های الگوریتم FCFS:
1- Non-preemptive است
2- starvation ینی ممکنه به پروسسی هیچ وقت cpu نرسه و گرسنگی پیش بیاد؟ نه نمیشه چون ما داریم به ترتیب بهشون سرویس میدیم پس No starvation است (این یکی از بهترین ویژگی های این الگوریتم است)
3- اگر ما یکسری پروسس هایی با burst کوچک و burst بزرگ داشته باشیم معمولا به نفع کی میشه؟ اگر ما بیشتر پروسس هامون کوچک باشند این الگوریتم خوب عمل میکنه ولی اگر وسطش یکسری پروسس های بزرگ داشته باشیم دیگه خوب عمل نمیکنه و وابسته به این است که کی اول می رسه و اگر بزرگه اول برسه اون پروسس های کوچیکتر باید خیلی منتظر بمونن پس این الگوریتم وقتی خوبه که اون burst های ما کوچیک باشن
4- این الگوریتم overhead خیلی کمی داره چون با بیگ او یک یا O(1) می تونیم پیاده سازیش بکنیم

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- **SJF is optimal** – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request

-
توی الگوریتم قبلی زمانی سیستم نفع میکرد که پروسس های کوچیک داشته باشه ولی اگر سیستم پروسس بزرگتر داشت به ضرر پروسس های کوچک تر میشد
حالا اگر صف رو جوری بچینیم که پروسس کوچیک هارو بذاریم اول ینی اول به پروسس هایی cpu بدیم که کوچکتر است burstشون پس الگوریتمی که میگیم SJF ینی اونی که job اش کوچیکتره رو اول انتخاب بکن
ینی اول نگاه میکنیم burst های cpu کی کوچیکتر است و به اون cpu میدیم که اجرا بشه
و توی این حالت ما minimum average waiting time داریم چون خودمون پروسس های کوچیک رو چیدیم اول و اونی قدری برای پروسس کوچیک waiting time شون کم میشه پروسسهای بزرگ ها waiting time شون زیاد نمیشه
پس اینجا average waiting time مون از همه حالات کمتر است : نقطه مثبت
مشکل این حالت چیه؟ ما از کجا بدونیم burst سی پی یو یک پروسس چنده مثلا توی حالت قبلی cpu رو بهش می دیم که burst اش تموم بشه و هرچقدر زمان برد و بعد از اینکه تموم شد ما می فهمیم burstاش چنده ولی وقتی که یک پروسس اجرا نشده ما نمیدونیم burst اش چنده که بخوایم براساس اونا بچینیم و بریم جلو پس مشکل اینه که ما نمیدونیم burst بعدی هر پروسسی چقدره که بخوایم اون کوچکتره رو انتخاب بکنیم

Example of SJF

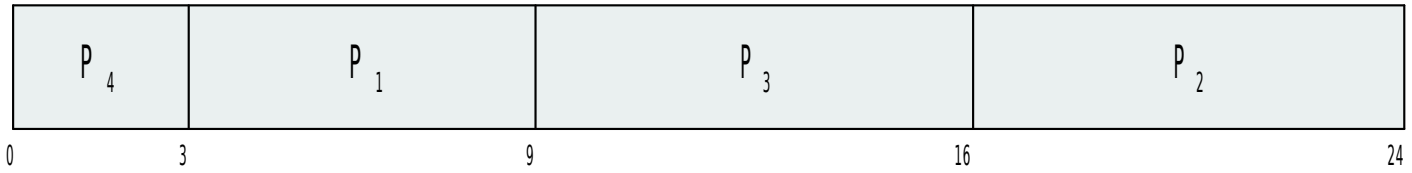
Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

فرض میکنیم همشون در زمان صفر وارد سیستم شدند

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

SJF Properties

- Non preemptive
- Maybe starvation for long processes
- The least average waiting time

-
Non preemptive : چون می ذاریم burst اش کامل اجرا بشه

اینجا ممکنه ایا starvation رخ بده؟مثلا توی مثال قبلی هنوز p2 اجرا نکردیم و یک پروسسی بیاد که burst اش یک باشه پس اول اون اجرا میشه با اینکه p2 زودتر اومده ولی چون burst اون کمتره اون اجرا میشه و ممکنه باز این اتفاق بیوفته قبل از اینکه p2 اجرا بشه پس اگر هی پروسس های کوچیک اینجا وارد بشه پروسس p2 ممکنه هیچ وقت اجرا نشه و همش منتظر بمونه و دچار گرسنگی بشه

مشخصه خوبش اینه که average waiting time اش کمه و بهینه است و بهترین حالتو داره

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

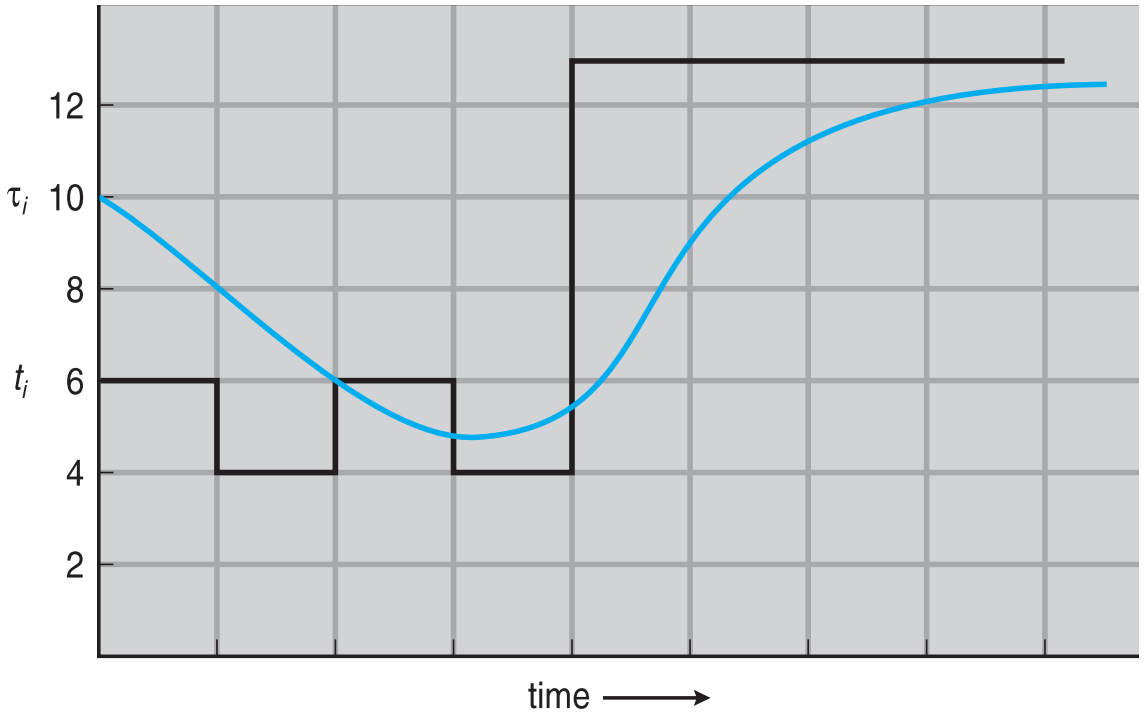
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α, $0 \le \alpha \le 1$
  4. Define: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$.

- Commonly, α set to ½

-
یک روشی برای اینکه بتونیم ببینم burst بعدی چنده اینه که حدس بزنیم
چجوری حدس بزنیم؟ براساس burst های قبلی -->
مثلا اگر بدونیم یک برنامه ای نرماله و burst های مختلفش تقریبا یه جوری متناسب با هم اند می تونیم به اندازه burst های قبلی burst جدیدش رو حدس بزنیم

مثلا یکی از این روش هایی که گفتیم exponential averaging است

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

توی نمودار مشکیه ما burst های واقعی پروسس رو می بینیم و ابیه اون چیزیه که پیش بینی شده

Example of Shortest-remaining-time-first

- Preemptive version of SJF is called **shortest-remaining-time-first**
- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

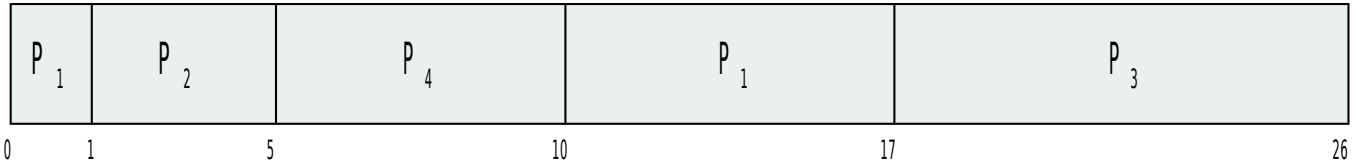
-
مثال:
توی این مثال به جای SJF می تونیم بگیم shortest-remaining-time-first ینی اینجا cpu رو می گیریم ازش به این صورت که توی زمان صفر اول میدیم به p1 بعد توی زمان 1 می بینیم burst پروسس p2 ما 4 است و p1 ما 7 پس p2 رو می ذاریم و به همین ترتیب می ریم جلو که شکلش توی صفحه بعدی است
پس اینجا الگوریتم رو preemptive اجرا کردیم
اینجا همچنان می تونه گرسنگی رو داشته باشه و حتی می تونه بدتر باشه ینی cpu از یکی بگیریم و دیگه بهش برنگرده
فقط خوبیش اینه که اگر یک پروسسی در حال اجرا بود و یک پروسسی با burst کوچکتر رسید cpu رو به اون پروسس جدیده می دیم

Example of Shortest-remaining-time-first

- Preemptive version of SJF is called **shortest-remaining-time-first**
- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

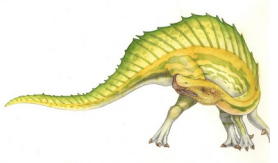
- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem: **Starvation** – low priority processes may never execute
  - When shut downing IBM 7094 at MIT in 1973, there is a waiting job from 1967!
- Solution: **Aging** – as time progresses increase the priority of the process

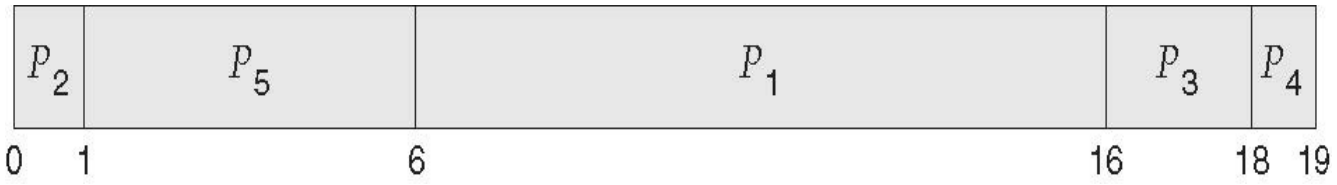


-
Priority ینی اولویت
بعضی از پروسس ها اولویت بالاتری دارند مثل پروسس های سیستمی که باید اولویت بالاتری داشته باشه
توی این روش به هر پروسسی یک Priority یا یک عدد می دیم و cpu براساس اینکه کدوم یکی از پروسس ها Priority شون بیشتر است بهشون cpuمیده ینی اونی که Priority اش بالاتره باید بیشتر بهش cpu داده بشه
این الگوریتم میتونه دو حالت باشه یا preemptive or nonpreemptive
SJF هم میتونیم بگیم یک الگوریتم Priority است ینی اونی که برست زمانیش کوتاه تره بهش اولویت داده شده که عددش رو میذاریم 1 روی برستش که هرچی این برست کم باشه کل کسر بزرگتر میشه و اولویتش بیشتر میشه
مشکلش چیه مخصوصا اگر preemptive باشه؟ گرسنگی زیاده می تونه ایجاد بکنه ینی اون پروسس های که اولویت کمتری دارند می تونن هیچ وقت اجرا نشن برای nonpreemptive هم میتونه همین باشه
راه حل: agingبه سیستم اضافه کنیم ینی هرچی عمر یک پروسس توی سیستم بالاتر می ره ما بیایم Priority اش ببریم بالا

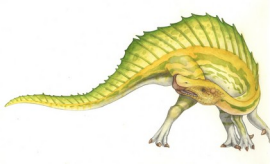
Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

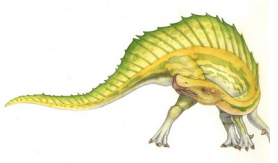


نمونه ای از برنامه ریزی اولویت

یک نکته داره: اگر زمان ورود هم می ذاشت باید به این دقت می کردیم که علاوه بر اینکه کدوم پروسس Priority بالا تر است باید میدیدیم اصلا توی اون زمان اون پروسس هنوز وارد شده یا نه ینی در کل ببینیم مثلا توی زمان 1 کدوم پروسس ها هستن و از بین اونا اونی که Priority بالاتری داره رو انتخاب کنیم

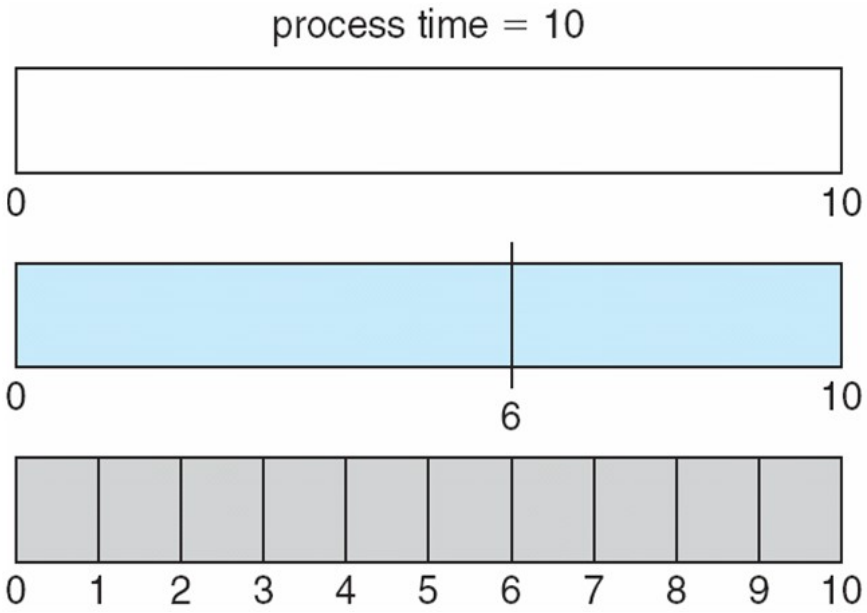
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once.
- **What is the maximum response time?**
  - **No process waits more than (*n*-1)*q* time units.**
- Timer interrupts every quantum to schedule next process
- Performance
  - Large q: FIFO
  - Small q: *q* must be large with respect to context switch, otherwise overhead is too high



-
مشهور است
یک بازه زمانی بهشون اجازه میدیم اجرا بشن مثل حلقه می شه ینی مثلا از p0 شروع میشه تا p10 بعد دوباره برمیگرده به p0 و همین ترتیب و هرکدوم که تموم شدن از حلقه میاد بیرون و دوباره حلقه ادامه پیدا میکنه
هر پروسسی توی بازه زمانی time quantum سرویس میگیره و cpu رو میگیره
پس اگر n تا پروسس داشته باشیم و time quantum هم q باشه می تونیم بگیم هر پروسس سهمش از cpu به اندازه n/1 است که به تعداد پروسس ها بستگی داره
زمان پاسخگوییشون بیشتر از (q)1-n نمیشه که اون پروسس اخر هست که اینطوری میشه
تایمر روی q ست میکنیم که پروسس رو عوض کنیم
اگر q بزرگ بگیریم --> برست پروسس تموم بشه توی اون q ازش خارج میشه - حالا q رو چجوری انتخاب کردیم که همه اون پروسس ها توی یکبار اجراشون برستشون تموم میشه که به این معناست که q خیلی بزرگ است انگار داریم FIFO عمل میکنیم
اگر Q کوچک بگیریم --> قبل از برست یک cpu رو ازش گرفتیم و دادیم به یک پروسس دیگه و هی داریم با یک q کوچیک این پروسس ها رو جابه جا میکنیم - این زمان جابه جایی بین دو تا پروسس یا زمان context switch که این وسط دوتا پروسس است یک overhead داره حالا اگر زمان context switch بزرگتر از q باشه این برامون بهینه نیست و انگار بیشتر زمان رو صرف context switch میکنیم

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

q usually 10ms to 100ms, context switch < 10 usec

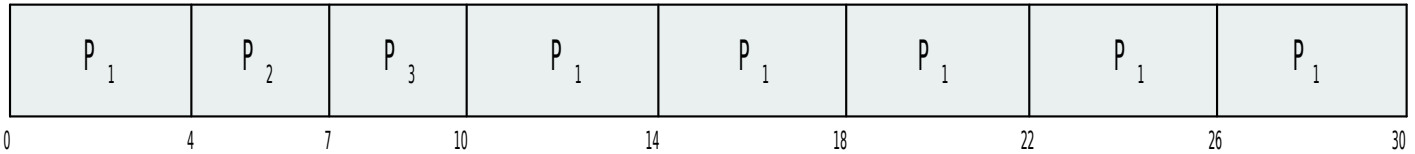
Time Quantum and Context Switch Time

اگر کوانتوم زمان بین 10 میلی ثانیه تا 100 میلی ثانیه باشه context switch باید کمتر از 10 میکرو ثانیه باشه وگرنه برامون نمی صرفه که این کوانتوم زمانی رو در نظر بگیریم

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Compare average waiting time, turnaround time and response time with SJF
  - Typically, higher average waiting time and turnaround than SJF, **but better *response***

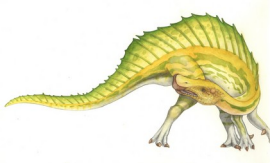
-
average waiting time برای RR بیشتر از SJF است
turnaround time هم توی RR بیشتر است
response time توی RR بهتر از SJF است و مهمترین مشخصه RR هم response time اش است
اینجا فرض شده که برست تایم ها بیشتر از Q است

RR Properties

- Preemptive
- No starvation
- Suitable for time sharing systems
- Low throughput with short quantum
- Same as FCFS with a quantum larger than CPU bursts

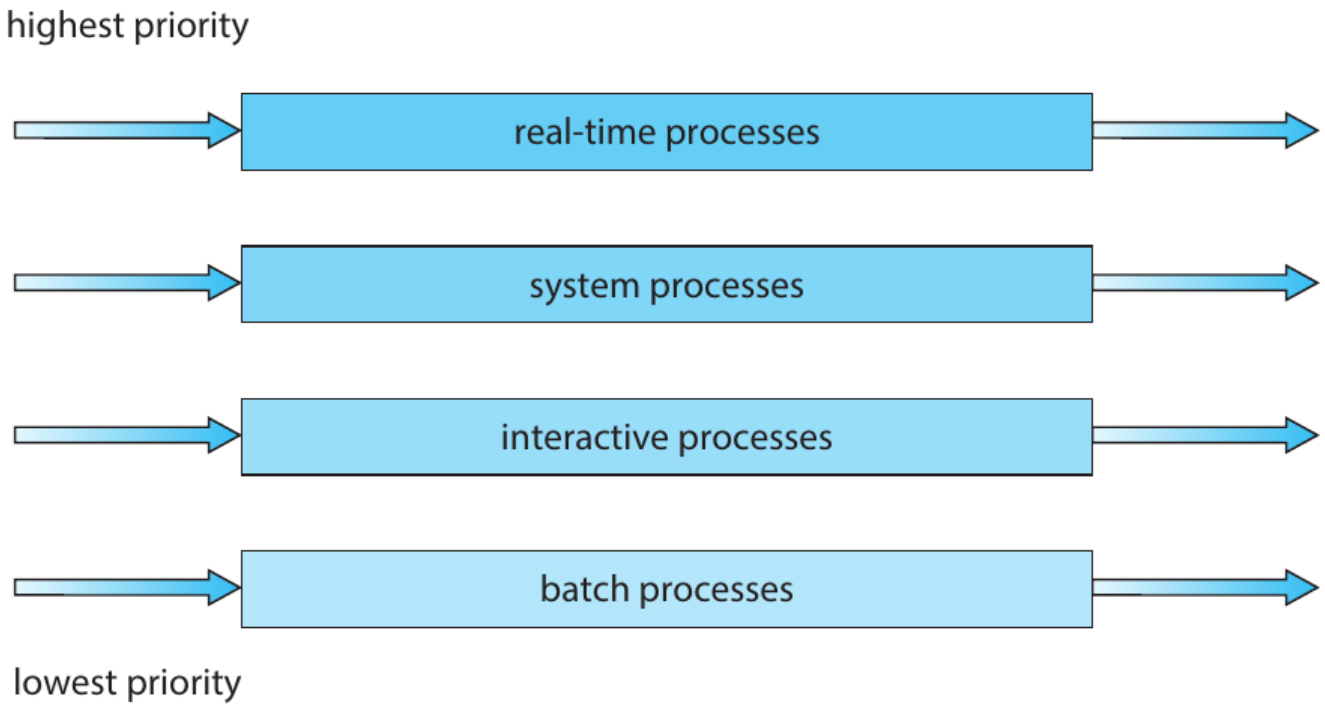
Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS



-
توی این الگوریتم صف های متفاوت هایی در نظر میگیریم تا اینجا یک صف برای پروسس ها در نظر گرفتیم - اینجا بیشتر از یک صف در نظر میگیریم
اینجا دو تا صف میگیریم:
صف interactive که خیلی مهمه که هی با کاربر در ارتباطه و یک صف batch
برای interactive زمان پاسخگویی کمی براش مهمه پس از الگوریتم RR
برای batch چون فقط مهمه که فقط اجرا بشه پس از الگوریتم FCFS
توی یکی از صف ها الگوریتم RR داریم و توی یکی دیگه FCFS داریم
نکته: توی یک لحظه زمانی که میخوایم cpu رو بهش پروسس بدیم باید تصمیم بگیریم از کدوم صف برداریم پس یک اسکجولینگی برای انتخاب این صف ها هم باید داشته باشیم :
1- مثلا یک priority ثابتی برای این دو صف در نظر بگیریم مثلا تا زمانی که توی صف foreground پروسس است اول اینو در نظر بگیریم و بعد back.. که این باعث میشه گرسنگی برای back.. میشه
یک کار بهتر اینه که ما تعیین کنیم چه سهمی از cpu رو به foreground بدیم و کدوم رو به back.. بدیم چون foreground ها مهم ترند می تونیم زمان های بیشتری رو به اینا اختصاص بدیم ینی 80 درصد cpu برای foreground و 20 درصد دیگه برای back..

Multilevel Queue Scheduling

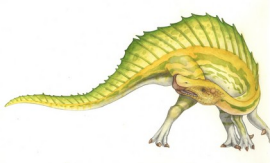


زمانبندی صف چندسطحی

4 صف داریم

Multilevel Feedback Queue

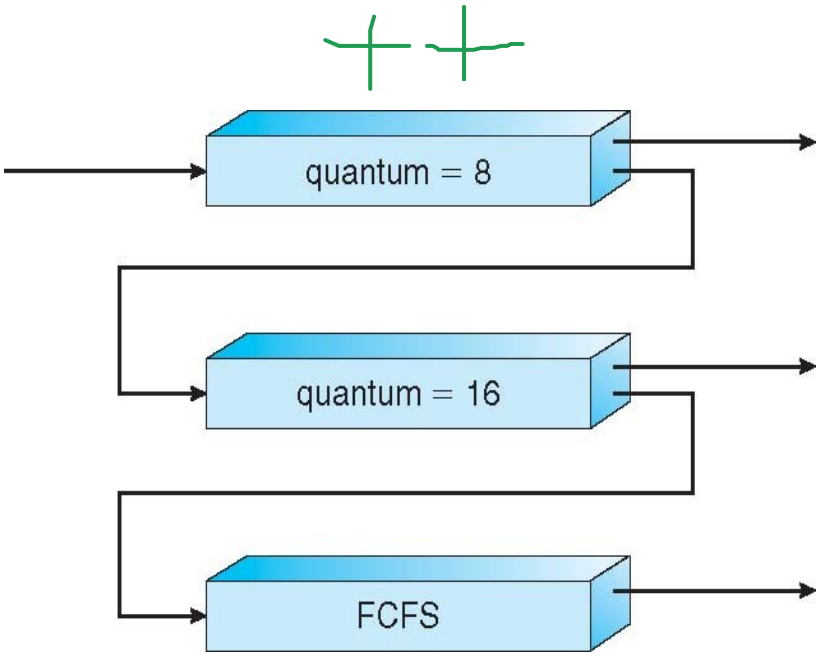
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

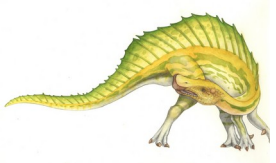


-
نکته: interactive بیشتر I/O هستند و اونایی که batch هستند cpu bound هستن و برست cpuشون بزرگه
یک پروسسی داریم که توی batch است چون برست cpuاش بزرگ بوده و ممکنه از یه جایی به بعد همش I/O پیدا کنه ینی ممکنه یک سری پروسس هایی اولویتشون تغییر بکنه توی زمان در این صورت ما باید چی کار کنیم ؟
یک راه حل Multilevel Feedback است ینی همون Multilevel رو داریم ولی یک فیدبکی هم از اجرای پروسس ها میگیریم
شاید بخوایم بر یک اساسی این پروسس ها رو بین صف ها جابه جا کنیم ینی اولویتشون رو عوض کنیم در این موقعیت ها باید به یکسری از مسائل فکر کنیم:
تعداد صف چندتا بگیریم
برای هر کدوم از این صف ها چه الگوریتمی در نظر بگیریم
کی تصمیم بگیریم که اولویت یک پروسس رو ببریم بالا و چه زمانی بیاریم پایین
اگر یک پروسسی توی یک زمان وارد سیستم میشه توی اون لحظه اول توی کدوم صف بذاریمش

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2





-
سه تا صف داریم
صف اول و دوم rr اجرا میشن و با کوانتوم زمان متفاوت یکی 8 و یکی 16
پس اونی که 8 ینی زمان پاسخگویی توشون برامون مهمتر است ولی 16 ها نه ینی یه مقداری بعد از اون 8 هستند
اخری هم fcfs میگیریم که به batch ها نزدیکترند
یک سیاست برای اسکجولینگ کردن:
به این صورته که وقتی که پروسس جدید به سیستم میاد اول ببریمش به این صف Q0 یک بازه زمانی اجراش کنیم یا برستش تموم میشه که کلا از سیستم خارج میشه یا برستش تموم نمیشه بعد از 8 در این صورت بعد از 8 واحد زمانی می بریمش توی صف بعدی
یه زمانی هم صف Q1 داره اسکجول میشه و دوباره این پروسس از این صف برداشته میشه و به اندازه کوانتوم زمانی این صف ینی 16 بهش cpu میدیم یا کلا تموم میشه برستش که خارج میشه یا نه وبزرگتر از این 16 بوده در این حالت ما می فهمیم این جز اونایی بوده که برستشون بزرگ بوده پس می بریمش به صف اخری ینی Q2 پس این مشکل قبلی رو حل کرد که ما نمیدونستیم این پروسس batchهست یا چی
پس اول با همه یه جور برخورد می کنیم ینی همه رو اول بهشون یک کوانتوم زمانی کمی میدیم که اجرا بشه اگه توی این حالت برستش تموم شد ینی نشون میده که خیلی interactive بوده و یک پروسس I/O بوده و اگر همین برستش تموم نشد بازم یکم بهش مهلت دادیم ینی بردیمش توی صف 16 که شاید interactive باشه ولی اگه دیگه باز تموم نشد می فهمیم از نوع batch است
پس اون مشکلی که نمی دونیم از اول اون پروسس جز کدوم یکی از این صف ها هست رو با این کار حل کردیم

++: یک مشکلی داره که یک ادم بدخواه بخواد هی سی پی یو رو بگیره
پس یک کدی میده که مدام داره I/O استفاده میکنه و برستش خیلی کوچیک است
اکثر مواقع هم صف بالایی که 8 هست رو داریم اجرا میکنیم
راه کار: میتونیم از طریق فیدبک بفهمیم ینی اگر ببینیم همچین اتفاقی مدام داره برای کی پروسس می افته بیایم اولویتش رو بیاریم پایین ینی خودمون بذاریم توی صف های پایینی

Real-Time CPU Scheduling

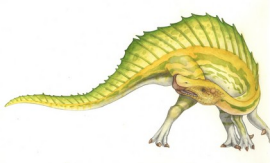
- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Real-time characteristics:
  - Periodic (p value)
  - Processing time (t)
  - Deadline (d)
- Some algorithms:
  - Priority-based

Linux command to check process priority and scheduling
chrt

-
یک دسته دیگه از پروسس ها هستند که باید دید متفاوتی نسبت بهشون داشته باشیم که اونها Real-Time ها هستند و باید یکسری الگوریتم های مجزا برای این ها بدیم :
دو دسته داریم:
ما تا اینجا راجع به soft Real-Time صحبت کردیم شاید یه تاخیر کوچیکی داشته باشه ولی خب مشکلی نداره و کاربر شاید فقط کسل بشه
hard Real-Time : ینی برای انجام یک کاری ما یک ددلاینی داریم ک از این ددلاین گذشتن دیگه اسکجول کردن اون تسک فایده ای برای ما نداره
ویژگی های Real-Time:
یک رخدادی به صورت متناوب اتفاق می افته و ما باید سر اون بازه زمانی به اون اتفاق رسیدگی کنیم و یک مقدار زمانی داریم که بهش میگیم ددلاین و اگر توی اون ددلاین به این رخداد رسیدیم اوکی هست ولی اگر از این زمان گذشت دیگه برامون فایده ای نداره پردازش اون
زمان d ینی ددلاین پردازشش چقدر است
برای جواب یک Processing time براش داریم ینی وقتی رخداد اتفاق افتاد اون عملیاتی که در مقابل این رخداد باید انجام بدیم یک Processing time داره
این پارامترها توی اون الگوریتمی که داریم انتخاب میکنیم باید بهش توجه کنیم
یکی از روش ها روش Priority-based است ینی اون الگوریتم هایی که Hard real-time هستند رو بهشون بالاترین اولویت رو بدیم که هر تسک دیگه ای که توی cpu در حال اجرا بود وقتی که این رخداد اتفاق افتاد ینی اون پروسس وارد سیستم شد ما دیگه cpu رو بدیم به این پروسس
اگر پروسس های متنوعی با اولویت بالا داشته باشیم چجوری باید بهشون رسیدگی کنیم؟ باید به ددلاین دوتاشون دقت کنیم و اونی که کمتر بود ددلاینش رو باید رسیدگی کنیم - همینطور باید به t هم دقت کنیم که t به اضافه اون ددلاینش باید کمتر از ددلاین دوم اون یکی پروسس باشه چون اگر بیشتر از ددلاین دوم باشه باعث میشه اون پروسس رو از دست بدیم

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**



-
دو مدل مالتی پروسسور داریم:
Asymmetric : ولی توی این یک coreرو میگیم منیجر که کارهای مدیریتی بقیه core هارو انجام میده و بقیه core ها بقیه تسک های معمولی رو انجام میدن و کارهای مدیریتی روی اون یه دونه core است
SMP که اکثر سیستم های فعلی این است ینی core های مختلفش رو به یک نوع استفاده میکنیم ینی همه core همه علمیاتی انجام میدن

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor
- What is the problem of load balancing according to affinity?

Linux command to check process affinity
taskset

-
مسئله مالتی پروسسور سیستم ها هست که اون بحث Load Balancing است توی یک سیستمی که ما چندتا هسته cpu داریم دوست داریم همشون به یک اندازه سرشون شلوغ باشه پس دوست داریم Load Balancing انجام بدیم به طوری که workload کل سیستم رو تقریبا مساوی تقسیم کنیم بین core سی پی یوهامون
هر core صف های جداگانه ای برای اجرا داره ممکنه یک core سرش خیلی شلوغ بشه و بقیه core ها سرشون خلوت در این صورت یک Load Balancing میتونه بیاد اونی که سرش شلوغه رو بیاد تکسش رو منتقل بکنه به صف اونی که سرش خلوت تره به دو حالت میتونه اتفاق بیوفته:
push ینی یک core که سرش شلوغه میاد پروسس های خودش رو به core های دیگه که خلوت اند push میکنه
pull اگر یک core دید سرش خلوته از بقیه core ها که سرشون شلوغه میاد از پروسس های اونها pull میکنه توی خودش
پروسس ها وابسته میشن به اون core چرا این اتفاق می افته؟ چون کش اون core با اطلاعات اون پروسس پر میشه و حالا اگر برستش تموم شد و اگر بیاریمش روی کی core دیگه اون کشی که اونجا ایجاد کرد بدردش نمیخوره و باید بیاد از اول اطلاعاتش رو از ممووری کش کنه توی کش خود این core و اون کش هم توی اون core قبلی جایگزین میشه با اطلاعات پروسس های دیگه پس از این کش بهینه استفاده نکردیم پس اون پروسس ها می خوان توی همون core بمونن

Thread Scheduling

- Distinction between user-level and kernel-level threads
- **When threads supported, threads scheduled, not processes**
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

-
پس دوتا اسکوپ می تونیم برای اسکجولر داشته باشیم:
PCS: اسکجولر وقتی میخواد انتخاب بکنه بین تردها و تردهای یک پروسس رو با هم ببینه و فکر کنه بین تردهای یک پروسس بخواد یکیش رو انتخاب بکنه برای اسکجولر شدن
SCS: اسکجولر همه تردهای یک سیستم رو یکجا ببینه و رقابت بین همه تردهای یک سیستم باشه سر CPU

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - **PTHREAD_SCOPE_PROCESS** schedules threads using PCS scheduling
  - **PTHREAD_SCOPE_SYSTEM** schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM
- Scheduling policies
  - FIFO
  - RR
  - OTHER

Linux scheduling

- pthread_attr_init(&attr)
- Config Scope
  - pthread_attr_getscope(&attr,&scope)

    pthread_attr_setscope(&attr,PTHREAD_SCOPE_PROCESS)
- Config scheduling algorithm
  - pthread_attr_getschedpolicy(&attr,&policy)

    pthread_attr_setschedpolicy(&attr,SCHED_FIFO)
- Shell commands
  - chrt

-
دو تابع هست که میتونیم از طریقشون فیچرهای اون تابع اولی رو تغییر بدیم:
اسکوپ اسکجولینگ
پالیسی اسکجولینگ ینی الگوریتم اسکجولینگی که به کار می ره هست

set ینی خودمون یه کاری رو بکنیم

Linux Scheduling in Version 2.6.23 +

- Features:
  - constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
    - Real-time range from 0 to 99 and time sharing range from 100 to 140
  - **Higher priority gets larger q**

-
تمرکز روی الگوریتم های لینوکس است
برای 2.6 به بعد هست
ویژگی های اصلیش:
از مهمترین اهداف زمان اسکجولینگ روی اردر یک بره ینی با اردر یک انتخاب بکنیم که توی این لحظه کدوم یکی از پروسس ها می تونن cpu بگیرند
الگوریتم Preemptive, priority based --> به همه پروسس های سیستم به یک نگاه دیده نمیشه ینی اولویت در نظر گرفته شده و اونی که اولویت بالاتری زودتر میتونه cpu رو بگیره
دو تا رنج priority تعریف میشه توی این الگوریتم:
یکیش برای تایم شیرینگ و یکیش هم برای ریل تایم است
دیفالت سیستم روی تایم شیرینگ است
پروسس هایی که اولویت بالاتری دارند کوانتوم طولانی تری برای اجرا دارند توی cpu

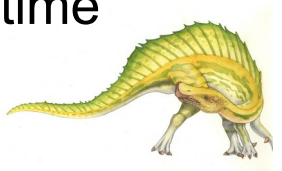
Linux Scheduling in Version 2.6.23 +

- Scheduling in the Linux system is based on scheduling classes
  - Each class is assigned a specific priority
  - To decide which task to run next, the scheduler selects the highest-priority task belonging to the highest-priority scheduling class
- By using different scheduling classes, the kernel can accommodate different scheduling algorithms based on the needs of the system and its processes.
  - Ex. The scheduling criteria for a Linux server, may be different from those for a mobile device running Linux.

-
اینجا دو تا کلاس اسکجولینگ داریم که هر کدومش priority میگیره مثلا اون ریل تایمه باید priority بالاتری داشته باشه و اسکجولر وقتی میخواد انتخاب بکنه از اون صفی که priority بالاتری داره اون پروسسی رو انتخاب میکنه که priority بالاتری داره
این امکان رو ایجاد کرده که کرنل بتونه الگوریتم های اسکجولینگ متفاوتی رو اجرا بکنه یا اضافه بکنیم توی کرنل لینوکس و این وابسته به شرایط است

Completely Fair Scheduler (CFS)

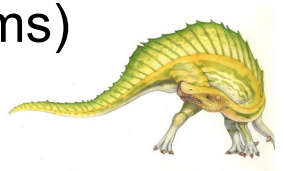
- Standard Linux kernels implement two scheduling classes
  - a default scheduling class using the CFS scheduling algorithm
  - a real-time scheduling class
  - New scheduling classes can, of course, be added
- ***Completely Fair Scheduler*** (CFS)
  - assigns a fair proportion of CPU processing time to each task (the quantum value)
  - **Scheduling decision:** CFS will pick the process with the lowest vruntime to run next
  - **Scheduling quantum (Time Slice):** Tasks with lower nice values receive a higher proportion of CPU processing time



-
CFS برای اون بخش غیر ریل تایم در نظر گرفته شده -- > توی سیستم ها به عنوان default scheduling class می شناسیم
با دستور chrt می تونیم الگوریتم های اسکجولینگ رو ببینیم
بخش اساسی که در مورد CFS داریم اینه که هدف ما توی این اینه که یک سهم عادلانه ای از CPU به هر کدوم از تسک ها بتونیم بدیم منظور از سهم همون کوانتوم زمانی است که به هر پروسسی میدیم که می خواد اجرا بشه --> سهم عادلانه ای به همه پروسس ها بدیم و البته priority هم در نظر بگیریم ینی متناسب با priorityاشون سهم عادلانه ای بدیم
الگوریتم ما باید دوتا چیز رو تعیین کنه:
Scheduling decision: ینی الان اگر اسکجولینگ باید یکی رو انتخاب کنه کدوم رو انتخاب بکنه --> پس الگوریتم ما باید این سوال رو جواب بده
Scheduling quantum: اگر انتخاب شد کوانتوم زمانی که بهش میدیم اجرا بشه چقدره
اگر در مورد CFS بخوایم اون دوتا سوال بالا رو جواب بدیم:
CFS پروسسی رو انتخاب میکنه که vruntime پایینتری نسبت به بقیه پروسس ها داشته باشه :
سوال اول
سوال دوم: تسک هایی که nice values کمتری دارند سهم بیشتری از CPU رو به خودشون اختصاص میدن
خلاصه: اونی که priority بالاتری داره رو هم زودتر انتخاب بکنیم و هم تایم اسلایس بیشتری بهمش بدیم و در عین حال می خوایم عادلانه باشه

CFS: Nice Value

- Lower nice value indicates a higher relative priority
- Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values.
  - If a task increases its nice value from, say, 0 to +10, it is being nice to other tasks in the system by lowering its relative priority (allow other tasks to scheduled earlier and for longer quantum)
- sched_latency: (Targeted latency) : an interval of time during which every runnable task should run at least once.
  - Proportions of CPU time are allocated from the value of sched latency
  - But what if there are “too many” processes running? Wouldn’t that lead to too small of a time slice, and thus too many context switches?
    - Yes! Solution is considering min granularity (Ex: 6ms)



-
Nice Value: میخوایم اینو جوری تعریف بکنیم که مشخص کننده priority ما باشه ولی برعکس priority است ینی هر چه مقدار نایس کمتر باشه priority بیشتر است
به عبارتی میگیم اونی که مقدار نایسش بیشتره زودتر کنار می کشه و حق خودشو میده به بقیه و priorityاش کمتره
Targeted latency: این مقدار، مقدار اون بازه زمانی است ک میخوایم مطمئن بشیم توی این بازه زمانی همه پروسس های که الان توی سیستم ready هستند حداقل یک بار cpu می گیرند-
این مقدار ممکنه متغییر باشه
اگر تعداد پروسسها خیلی زیاده بشه --> تعداد context switches ها خیلی زیاده میشه و بهینه نباشه برامون
اگر توی این بازه زمانی خیلی تعداد پروسس ها زیاد میشه و مثلا همشون سهم مساوی می گیرند از cpu ینی اگر طولش l است تقسیم بر n میشه ینی l/n حالا اگر این l/n خیلی کوچیک بشه ینی به نسبت به context switches خیلی کوچیک بشه در این حالت توی سیستم min granularity تعریف می کنن و min granularity ینی چقدر ما خورد کنیم مثلا 6 میلی ثانیه ینی میگیم اگر اون تایم اسلایسی که به اون پروسس می رسه کمتر از min granularity شد ما به اندازه همین min granularity بهش cpu میدیم ینی مقدارشو میاریم بالا مثلا اگر اون مقدار شد 1 میلی ثانیه ما 1 میلی ثانیه نمی ذاریم و می ذاریمش مثلا 6 میلی ثانیه

CFS: Virtual runtime

- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
  - For runtime= 200, what is **virtual runtime** for normal, high, low priority?
- To decide next task to run, scheduler picks task with lowest virtual run time
- Which one has higher priority?
  - IO bound or CPU bound process?

-
run time پروسس ها رو ینی چقدر تا حالا توی cpu اجرا شدن رو همیشه نگه داریم ینی چقدر cpu گرفته تا الان و زمانی که می خواست اسکجولر یکی رو انتخاب بکنه اونی رو انتخاب میکنیم که run time کمتری داشته باشه
Virtual runtime: میخوایم اینجا priority هم در نظر بگیریم توی بالایی در نظر نگرفتیم - حالا اگر بخوایم priority هم در نظر بگیریم اون وقت میخوایم یک مقدار جدیدی از روی run time تعریف میکنیم که اون مقدار جدید وابسته به priority تسک است
اگر پروسسی نرمال باشه ینی مقدار نایسش صفر است در این صورت انتظار داریم که run time اش 200 است virtual runtime اش هم همین باشه
ولی اگر پروسسی priority بالاتر باشه ینی مقدار نایسش کم باشه پس virtual runtime نسبت به run time کمتر باشه ولی اگر یک پروسسی مقدار نایسش بالاتر باشه در این حالت virtual runtime به نسبت run time واقعی خودش بیشتر است پس بازم دیرتر انتخابش میکنیم
پس اسکجولر اونی رو انتخاب میکنه که virtual runtime اش از همه کمتر است

Calculating time slice and vruntime

- CFS maps the nice value of each process to a weigh

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

$$\text{time\_slice}_k = \frac{\text{weight}_k}{\sum_{n=0}^{n-1} \text{weight}_i} \cdot \text{sched\_latency}$$

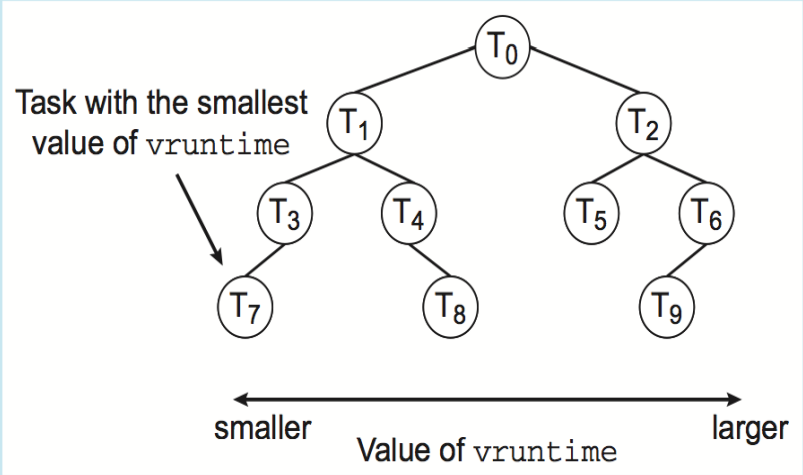
$$\text{vruntime}_i = \text{vruntime}_i + \frac{\text{weight}_0}{\text{weight}_i} \cdot \text{runtime}_i$$

-
توی کلاس گفت این بخش مهمه مثل این که مثلا بگه اگه اون سیگما نبود چی میشد اینا پس خوب این صفحه رو درک بکن
خلاصه از سیاست کلی توی الگوریتم CFS: هر بار اسکجولر باید یکی از پروسس هارو انتخاب بکنه و اونی رو انتخاب میکنه virtual runtime کمتری داره نسبت به بقیه که اماده اجرا هستند و تا چه زمانی بهش CPU میده به اندازه یک تایم اسلایسی که این تایم اسلایس وابسته به اون مقدار نایس است ینی هر چه مقدار نایس کمتر باشه این تایم اسلایس می تونه بیشتر باشه

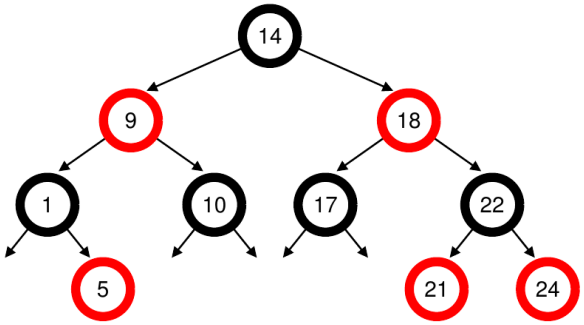
وزن همون پروسس تقسیم بر مجموع وزن های پروسس هایی که الان اماده اجرا هستند

CFS: red-black tree

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

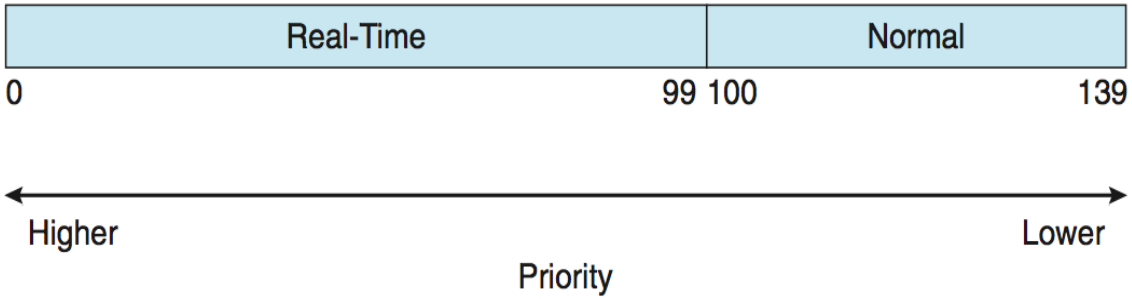


CFS: درخت قرمز-سیاه

نکات:
بدون اولویت میشه runtime
اونی که نایس تره دیرتر cpu میگیره
q خیلی بیشتر از context switch باید باشه که بصرفه
برای vruntime از دیتااستراکچر red-black می ریم که کلید نودها براساس vruntime چیده شده ینی سمت چپ تر ها vrumtime کمتری دارند و سمت راستی ها بیشترند
پس اسکجولر سمت چپ رو انتخاب میکنه و پیدا کردن نود سمت چپ میشه بیگ او logn و چون ما میدونیم هر سری سمت چپی ترین است و اگر ایندکسش رو ذخیره بکنیم میشه از اردر یک
اگر یک پروسسی داشته باشیم که خیلی I/O bound داشته باشه وقتی که برمیگرده چون خیلی مدت توی این درخت نبوده پس cpu از همه کمتر گرفته شده و اگر اولویتش هم زیاد باشه هی cpu به این داده میشه و باعث گرسنگی میشه برای این کار بهتره وقتی که میاد داخلش vrumtime اش برابر با کوچکترین vruntime باشه که هی نخواد cpu رو بگیره

Linux Scheduling: realtime

- Linux also implements real-time scheduling using the POSIX standard
  - SCHED FIFO
  - SCHED RR
- real-time policy runs at a higher priority than normal (non-realtime)
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Top command represents PR and NI values for processes
for normal processes (sched_other): PR = 20 + NI
(NI is nice and ranges from -20 to 19)
for real time processes (sched_fifo or sched_rr): PR = - 1 – real_time_priority
(real_time_priority ranges from 1 to 99)
Try chrt -m

--
برای قسمت ریل تایم دوتا الگوریتم FIFO , RR داریم و اینجا هم priority براشون در نظر گرفته میشه
و شکل رو ببین که داخلش است
قسمت نورمال با CFS اجرا میشه

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
- **Queueing Models**
- **Simulations**
- **Implementation**

-
چجوری ارزیابی میکنیم الگوریتم هارو؟
مدل کنیم اون الگوریتمی که داریم رو روی یکسری حالت هایی و براش محاسبه بکنیم پارامترهای
مدل نظر رو: Deterministic modeling
این Queueing Models در کنار Deterministic modeling استفاده میشه
Simulations: یک کد می نویسیم ک اون کد انگار داره یک سیستم کامپیوتری رو شبیه سازی
میکنه
Implementation: اون حالت واقعی میشه ینی توی سیستم های واقعی بیایم الگوریتم ها رو
بذاریم اجرا و ببینیم چجوری داره عمل میکنه