Lottery scheduling

### Увод

Процесорско распоређивање је метод по коме се нитима, процесима или токовима података даје приступ системским ресурсима ( нпр. процесорском времену ). Ово се углавном ради у сврху ефективног балансирања оптерећења на систему или постизања циљаног квалитета услуге. Потреба за алгоритмима распоређивања је произашла из потребе већине модерних система да изводе више послова(процеса) истовремено и мултиплексирање ( одашиљање вишеструких токова истовремено ).

Распоређивач углавном манипулише :

* Протоком - укупан број процеса који завршавају своје извршавање у јединици времена
* Кашњењем, специфично :
  + Временом окрета - укупно време између издавања процеса и његовог завршетка
  + Временом одзива - време које је потребно од слања захтева до првог одговора
* Правичност / Време чекања - једнако процесорско време сваком процесу / време током кога је процес у реду чекања

Сада ћемо испитати тип распоређивача познат као а распоређивач пропорционалног дељења, који се понекад назива и fair-share распоређивач. Пропорционални удио је заснован на једноставном концепту: уместо да се оптимизује вриеме обрта или вријеме одзива, распоређивач може покушајти да гарантује да сваки посао добије одређени проценат CPU времена. Откривен је одличан рани пример распоређивања пропорционалног дељења у истраживању Валдспургера и Веихла, познатије је као lottery-scheduling; међутим, идеја је свакако старија. Основна идеја је прилично једноставна: сваки пут држите лутрију да одредите који процес треба да иде даље, процеси који би требало да се чешће одвијају требало би да добију више шансе.

### Основни концепт: Тикети представљају удео

Основни концепт lottery-scheduling: **тикети**,се користе за представљање порције ресурса који процес (или корисник или шта год) треба да прими. Проценат тикета које процес има представља своје учешће у ресурсу система.

Погледајмо пример. Замислите два процеса, А и Б, затим рецимо да А има 75 карата, док Б има само 25. Дакле, оно што бисмо волели јесте А да прими 75% CPU и Б преосталих 25%. Lottery-scheduling постиже ово пробабилистички (али не детерминистички) одржавањем лутрије сваког пута (рецимо, сваки временски интервал). Лутрија је једноставна: распоређивач мора знати колико укупно постоји тикета (у нашем примеру има их 100). Распорешивач затим бира добитну карту, која је број од 0 до 99

1. Претпоставимо да А држи тикете 0 до 74 и Б 75 до 99, добитни тикет једноставно одређује да ли А или Б иде даље. Планер затим учитава стање тога победничког процеса и покреће га.

Ево примера добијања победничких тикета распоређивача:

63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49

Ево резултујућег распореда:

А Б А А Б А А А А А А Б А Б А А А А А

Као што можете видети из примера, кориштење случајности у lottery-scheduling доводи до пробабилистичке тачности у испуњавању жељене пропорције, али без гаранције. У нашем примеру горе, Б добија само 4 излаза од 20 временских интервала (20%), уместо жељене расподеле 25%. Међутим, што се ова два процеса дуже такмиче, већа је вероватноћа да ће се постићи жељени процент.

Lottery-scheduling такође пружа бројне механизме за манипулацију тикета на различите и понекад корисне начине. Један начин је са концептом вредности тикета. Вредност дозвољава кориснику да сет тикета додели међу својим пословима у било којој валути; систем тада аутоматски претвара поменуту вредност у исправну глобалну вредност.

На пример, претпоставимо да су корисници А и Б добили по 100 карата.

Корисник А изводи два посла, А1 и А2, и даје им по 500 тикета (од укупно 1000) у валути А. Корисник Б изводи само 1 посао и даје 10 карата (од укупно 10). Систем претвара алокације тикета А1 и А2 од 500 у вредности А на 50 у глобалној вредност; слично, 10 карата Б1 претворено је у 100 тикета. Лутрија се затим држи преко глобалних вредности тикета (укупно 200) да би се одредило који се посао обавља.

Још један користан механизам је **трансфер** тикета. Са трансферима, процес може привремено предати своје карте другим процесима. Овај механизам је посебно користан у клијент / сервер подешавањима, где клијентски процес шаље поруку на сервер тражећи од њега да уради неки посао у име клијента. Да би убрзао рад, клијент може да проследи тикете серверу и тако покуша максимизовати перформансе сервера док сервер решава захтеве клијента. Када заврши, сервер затим враћа тикете клијенту и све се враћа на старо. Коначно, **инфлација** тикета понекад може бити корисна техника. Са инфлацијом, процес може привремено повећати или смањити број тикета који поседује. Наравно, у такмичарском сценарију са процесима који не верују једни другима, то нема много смисла; могао би да има један похлепни процес који додели себи огроман број тикета и преузме машину. Уместо тога, инфлација се може применити у окружењу где група процеса верује један другом; у том случају, ако било који процес зна да му треба више CPU времена, може повећати вредност тикета као начин да се та потреба задовољи без комуникације са било којим другим процесима.

### Имплементација

Вероватно најневероватнија ствар у вези са lottery-scheduling алгоритмом је једноставност његове имплементације. Све што вам треба је добар насумични генератор бројева који бира добитну карту, структуру података за праћење процеса система (нпр. листу) и укупан број тикета.

Претпоставимо да процесе држимо у листи. Ево примера који се састоји од три процеса, А, Б и Ц, сваки са неким бројем каратa.

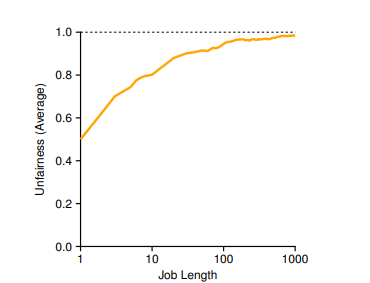
Да би донели одлуку о распореду, прво морамо изабрати насумичан број (**winner**) од укупног броја карата (400)2 .Рецимо да одаберемо број 300. Онда, ми једноставно прођемо кроз листу, са **counter**-ом који нам помаже да пронађемо **winner**-а.

Код пролази кроз листу процеса, додајући сваку вредност тикета на **counter** док вредност **counter** не буде већа од **winner**-а. Једном када је то случај, **current** елемент листе је победник. Наш пример победничке карте је 300, што значи: Прво, **counter** се повећава на 100 да би рачунао за А карте; пошто је 100 мање од 300, петља се наставља. Тада ће **counter** бити ажуриран на 150 (Б тикети), још мање од 300 и тако настављамо. Коначно, **counter** се ажурира на 400 (јасно више од 300), и на тај начин избијамо из петље са **current** који показује на Ц (победник).

Да би овај процес био најефикаснији, најбоље је организовати листу у сортираном редоследу, од највећег броја карата до најнижег. Редослед не утиче на исправност алгоритма; међутим, он генерално осигурава да се најмање итерира кроз листу, посебно ако постоји неколико процеса који поседују већину карата.

### Пример

Да бисмо учинили динамику lottery-scheduling разумљивијом, извршимо сада кратку студију времена завршетка два процеса један против другог, сваки са истим бројем тикета (100) и исто време извођења (Р, које ће варирати). У овом сценарију, желели бисмо да сваки посао заврши отприлике у исто време, али због насумичности lottery-scheduling, понекад један посао завршава пре другог. Да бисмо квантификовали ову разлику, дефинишемо једноставну **метрику неправичности**, Која је једноставно, време када се први посао заврши подељено са временом када се други посао заврши. На пример, ако је Р = 10, а први посао завршава у времену 10 (а други посао на 20), **U** = 10/20 = 0.5. Када се оба посла заврше скоро у исто време, **U** ће бити приближно 1. У овом сценарију, то је наш циљ: савршено фер распоред би постигао **U** = 1.



Слика приказује просечну неправичност како дужину два посла (**Р**) варира од 1 до 1000 у односу на тридесет испитивања (резултати се генеришу преко симулатора који ћемо касније објаснити). Као што можете да видите са графика, када дужина посла није јако дуга, просечна неправичност може бити прилично озбиљна. Само када се послови изводе за значајан број временских интервала lottery-scheduling се приближава жељеном исходу.

### Како расподелити тикете?

Један од проблема с којим се нисмо бавили је: како расподелити тикете процесима? Овај проблем је тежак, наравно, како се систем понаша, у великој мери зависи од начина на који се додељују тикети. Један приступ је претпоставити да корисници најбоље знају; у таквом случају, сваком кориснику се даје одређени број тикета, а корисник може да додели тикете процесима по жељи. Међутим, ово решење је нелогично: заиста вам не говори шта да радите. Тако, с обзиром на скуп послова, „проблем са расподелом тикета“ остаје отворен.

### Зашто не детерминистички?

Такође се можете запитати: зашто уопште користити насумичност? Као што смо видели

горе, док нам насумичност даје једноставан распоређивач, он повремено неће испоручити тачне пропорције, нарочито током кратких временских скала. Из тог разлога, Валдспургер је измислио **stride scheduling**, детерминистички планер расподеле по фер вредности. **Stride scheduling** је такође једноставан. Сваки посао у систему има корак, који је обрнуто пропорционалан броју тикета које има. У нашем пример изнад, са пословима А, Б и Ц, са 100, 50 и 250 тикета, респективно, можемо израчунати корак сваког од њих тако што ћемо поделити неки велики број са броју тикета за који је сваком процесу додељен. На пример, ако поделимо 10,000 са сваком од вредности тикета, добијамо следеће вредности корака за А, Б и Ц: 100, 200 и 40. Ову вредност зовемо корак сваког процеса; сваки пут када процес прође, ми ћемо

увећајти бројач за њега (то је његова пропусна вредност [**pass value**] ) за нјегов корак да бисмо пратили његов глобални напредак.

Распоређивач затим користи корак и **pass** да би одредио који процес треба да се изврши следећи. Основна идеја је једноставна: у било ком тренутку изаберемо процес који ће се извршавати и који има најнижу **pass** вредност до сада; када покренемо процес, повећамо његов **pass** за вредност његовог корака. Имплементацију псеудокода обезбеђује Валдспургер:

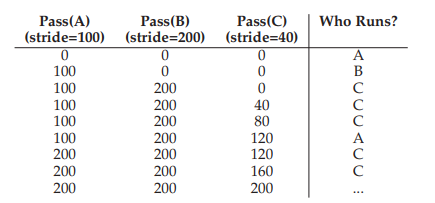
curr = remove\_min(queue); // pick client with min pass

schedule(curr); // run for quantum

curr->pass += curr->stride; // update pass using stride

insert(queue, curr); // return curr to queue

У нашем примеру, почињемо са три процеса (А, Б и Ц), са вредностима корака од 100, 200 и 40, и све са вредностима **pass** на почетку на 0. Дакле, прво, било који процес може да се покрене, пошто су њихове пролазне вредности једнако ниске. Претпоставимо да одаберемо А (произвољно; може се изабрати било који од процеса са једнаким вредностима ниског **pass**-а); Када завршимо са временским интервалом, ажурирамо његову вредност проласка на 100. Затим покрећемо Б, чија је вредност **pass**-а постављена на 200. Коначно, покрећемо Ц, чија вредност **pass** се повећава на 40. У овом тренутку, алгоритам ће изабрати најнижи **pass**, која је Ц-ова, и покренуће је, ажурирајући њен **pass** на 80 (корак Ц је 40, како се сећате). Тада ће се Ц поново покренути (и даље најнижи **pass**), подижући **pass** на 120. А ће се покренути сада, ажурирајући свој **pass** на 200 (сада једнак Б). Тада ће се Ц покренути још два пута, ажурирајући свој **pass** на 160, а затим 200. У овом тренутку, сви **pass**-ови су поново једнаке, овај поступак ће се поновити, бесконачно. Слика прати понашање распоређивача током времена.



Као што се може видети из слике, Ц је пет пута покренут, А два пута, а Б само једном, тачно у сразмери са вредностима њихових карата од 250, 100 и 50. Lottery-scheduling постиже пропорције пробабилистички током времена; Stride scheduling добија их тачно на крају сваког циклуса распоређивања. Дакле, можда се питате: с обзиром на прецизност распореда корака, зашто користити lottery-scheduling уопште? Па, lottery-scheduling има једну лепу особину да распоређивање корака: нема глобално стање. Замислите нови процес који улази у сред горе наведеног примера распоређивања корака; каква би била његова пролазност? Да ли би требало да буде постављено на 0? Ако је тако, то ће монополизовати CPU. Са lottery-scheduling, не постоји глобално стање процеса; једноставно додајемо нови процес са било којим тикетом, ажурирамо једну глобалну варијаблу да бисмо пратили колико укупних тикета имамо, и настављамо одатле. На овај начин, lottery-scheduling олакшава инкорпорирање нових процеса на разуман начин.

### Сажетак

Увели смо концепт пропорционалног распоређивача и укратко су разматрана два приступа: lottery-scheduling, stride scheduling. Lottery-scheduling користи насумичност на паметан начин за постизање пропорционалног учешћа; корак је детерминистички. Ниједан планер није савршен, а распоређивачи fair-share имају доста проблема. Једно питање је како такви приступи повезују У/И као што је горе поменуто, послови који обављају У/И повремено можда неће добити свој удео у CPU-у. Још једно питање је зашто они остављају отворен проблем са тикетима или приоритом, тј. како знате колико тикета треба да буде додељено вашем процесу? Други распоређивачи опште намене решавају ове проблеме аутоматски и могу се лакше применити.

Добра вест је да постоје многи домени у којима ови проблеми нису доминантна брига, а планери пропорционалног дељења се користе у великој мери. На пример, у виртуелизованом дата центру (или cloud-у), где бисте можда желели да доделите једну четвртину ваших CPU циклуса Windows VM-у, а остало на базну Linux инсталацију, пропорционално дељење може бити једноставно и ефикасно. Идеја се може проширити и на друге ресурсе.