Report

資工三 B06902132 林栢衛

設計

- 直接將要print在dmesg的information印在一個一個叫dmes的file當中。這麼做的目的是為了增加實作上的便利性，讓C library負責有關I/O及操作file的一切問題。
- 使用sched_setaffinity(0)，使我的程式全都運行在單個核心上面。
- 使用sched_setscheduler()設定priority。Scheduler的process是隨時都在high priority；其他的child processes都保持low priority，只有輪到要跑時才暫時提升成high priority。
- 我使用sched_yield()來分配cpu資源。當scheduler算出某個child process現在應該要跑，則scheduler會將該child process之priority由low暫時升成high，接著使用sched_yield()，將cpu使用權交給child process。
- child process在跑一個時間單位之後就會使用sched_yield()，將cpu還給scheduler，然後scheduler會把child process的priority設回low。
- 理論值可以告訴scheduler當前這個child process應該要跑多少時間單位，因此child process一但沒有跑完應該要跑的時間單位就將CPU還給scheduler的話，下回loop還是同樣的child process會跑。直到跑完應該要跑的時間單位，才會真正換別的child process來跑。
- Scheduler process使用sched_yield()時，如果沒有ready的high priority process，也不會使cpu 落入任何low priority的process手中。因此這樣的設計可以有效保障scheduler對於唯一一顆cpu的掌控權。
- 一開始會對所有的processes針對ready time以及先到的順序sort過。
- FIFO policy：照著sorted ready queue中的順序一一跑完。
- RR policy：ready queue中的每個process，跑固定時間後，就要更新execution time，然後排到最後去，換別的process來跑。
- SJF policy：雖然ready queue中的每個process都已經依照ready time sort過了，但要實作SJF policy還需要對execution time sort才足夠（在這邊，execution time可以看作還剩多少單位要跑）。為了確保等等scheduler可以直接取得execution time最小之process，不管是有process跑完要更新queue、有

process到了要加入queue或任何會動到ready queue的動作，都要記得對execution time sort過。

- PSJF policy：與SJF類似，不同之處在於每個process不一定會一次跑完，而是在跑一定時間後把cpu還給scheduler，scheduler可以根據當前ready queue裡的狀況判斷現在哪個process應該要跑。

核心版本

- 4.15.0-96-generic

比較實際結果與理論結果，並解釋造成差異的原因

- 對於每筆test data，我計算出裡面所有processes理論上的turnaround time乘上Unit_Time的值，並加總起來，以下以X簡寫之。接著我會加總所有processes實際上的turnaround time，以下以Y簡寫之。計算誤差率的方式是100% * (Y-X)/X，以下以err代表誤差率的絕對值。這麼做只是折衷的辦法，因為實際上在執行TIME_MEASUREMENT時和後面執行各policy的test data時，系統的狀況可能已有不同。
- Unit_Time: 0.0023100228354
- FIFO
  1. X = 7500 * 0.0023100228354
     Y = (1.138611786 + 1.250176733 + 1.110741335 + 1.147794745 + 1.146197113)
     err = 66.56%
  2. X = 337400 * 0.0023100228354
     Y = (189.693810587 + 12.264202383 + 2.394459529 + 2.358792783)
     err = 73.48%
  3. X = 111500 * 0.0023100228354
     Y = (17.962130147 + 11.043113728 + 6.663727668 + 2.267852478 + 2.209081551 + 2.209281016 + 8.896738532)
     err = 80.10%
  4. X = 7900 * 0.0023100228354
     Y = (4.445139867 + 1.154915400 + 0.427468399 + 1.087021134)
     err = 61.01%

5. X = 111600 * 0.0023100228354
   Y = (17.836586046 + 10.611197930 + 6.769463322 + 2.066630203 + 2.047346623 + 2.271732834 + 8.922681710)
   err = 80.40%

- RR

1. X = 7500 * 0.0023100228354
   Y = (1.349278330 + 1.135610364 + 1.121555035 + 1.118200607 + 1.178862163)
   err = 65.93%
2. X = 16300 * 0.0023100228354
   Y = (17.900612310 + 20.318159247)
   err = 1.50%
3. X = 124700 * 0.0023100228354
   Y = (33.324937684 + 44.532459568 + 42.144518613 + 48.163664110+56.536483798 +60.073235267)
   err = 1.14%
4. X = 91500 * 0.0023100228354
   Y = (9.329708972 + 9.240715306 + 9.143122164 + 31.982740173 + 35.516975947 +
   46.505673630 + 55.419375538)
   err = 6.73%
5. X = 92,000 * 0.0023100228354
   Y = (9.704680795 + 9.524409658 + 9.252356613 + 31.932486361 + 35.703140561 + 46.626799996 + 55.147624040)
   err = 6.99%

- SJF

1. X = 25700 * 0.0023100228354
   Y = (4.889015490 + 2.526663146 + 9.787598748 + 16.181371336)
   err = 43.77%
2. X = 28000 * 0.0023100228354
   Y = (0.214683591 + 0.425798440 + 9.669510407 + 9.085004801 + 15.735065608)
   err =45.69%

3. X = 96770 * 0.0023100228354
   Y = (6.654235727 + 0.020539087 + 0.019896753 + 8.866122784 + 8.393193174 + 10.811830050 + 15.576216222 + 19.995157450)
   err = 68.53%
4. X = 20000 * 0.0023100228354
   Y = (6.667544524 + 2.244531725 + 8.778892419 + 2.252333731 + 4.374687916)
   err = 47.36%
5. = X = 8000 * 0.0023100228354
   Y = (4.345903964 + 1.141315406 + 1.106523799 + 1.094881326)
   err =58.40%

- PSJF

1. X = 51000 * 0.0023100228354
   Y = (6.080492886 + 17.101523369 +32.522402238 +54.283234341) err = 6.64%
2. X = 17000 * 0.0023100228354
   Y = (2.187238612 + 8.807637512+4.361699221+2.237191091+15.346574230)
   err = 16.12%
3. X = 5000 * 0.0023100228354
   Y = (1.093036530 + 1.093115353+1.094130519 +7.698804594)
   err = 4.94%
4. X = 24800 * 0.0023100228354
   Y = (2.161309404 + 4.451507947 + 8.769055558 + 14.873938566)
   err = 47.19%
5. X = 28000 * 0.0023100228354
   Y = (0.208988354 + 0.414773884 + 9.494214834 + 8.852926106 + 15.433960328)
   err = 46.81%

- 仔細分析我的結果後，令我意外的是，如FIFO及SJF這種non-preemptive的、理論上不需要太常做context switch的policy，在實際跑時，使用我的誤差公式計算竟然會得到比其他preemptive的policy還要差的結果。
  我覺得可能是因為：

1. 我的scheduler process和child processes都用同一個cpu，比起使用多個cpu的做法會有更多context switch的機會。
2. non-preemptive process在實作上應該可以直接跑完再把cpu還給scheduler，而不用每隔一段固定時間就把cpu資源放掉。我的寫法可能會導致一些不必要的overhead。
3. 某些processes turnaround time理論值本來就很小，用誤差公式後，理論值和實際值的差別看起來會很大。
4. 執行project_1的程式時，我的電腦可能同時也在做別的工作，因此導致額外的context switch。

- References:

https://users.pja.edu.pl/~jms/qnx/help/watcom/clibref/qnx/sched_yield.html

http://man7.org/linux/man-pages/man2/sched_yield.2.html

https://www.twblogs.net/a/5baad6452b7177781a0e9a6f