OS Project 1 Report
CPU Scheduling

B06902104 吳由由

1 設計

1.1 判斷排程策略

- 依照使用者輸入判斷該次排程要採用哪種algorithm
- Define
  - Method 0: FIFO
  - Method 1: RR
  - Method 2: SJF
  - Method 3: PSJF

1.2 CPU 使用設定

- 以setrlimit 將RLIMIT_CPU設為RLIM_INFINITY
  在linux系統中，RLIMIT_CPU限制當一個process使用CPU的時間超過設定的soft margin時，會送SIGXCPU來終止process, 但如果這個signal被catch且持續使用CPU時直到超過hard margin時，系統會送出SIGKILL來把process砍掉。因為這次實驗的目標是讓我的scheduler和child process用越多CPU時間越好，最好不要被其他task context switch掉，因此我把soft margin和hard margin都設成infinity。
- 以setrlimit 將RLIMIT_RTTIME設為RLIM_INFINITY
  在linux系統中RLIMIT_RTTIME限制一個在real time scheduling policy下可佔CPU的時間上限。當process使用超過soft margin時，會送出SIGXCPU來中止該程式，但這個signal可能被catch因此程式仍能繼續消耗CPU資源，直到消耗的資源超過hard margin時系統會送SIGKILL來直接中止該程式。這次本程式的scheduler使用了real time schedule policy，且希望每個process在cpu的時間能能由我控制，因此我把soft margin和hard margin都設成infinity來避免系統把我的process踢除。
- 設定scheduler process使用0號cpu, 使用SCED_FIFO real time排程方式，給定priority 90. 在linux系統中使用SCED_FIFO，process在做完之前都可以一直佔著cpu, 因為我不希望scheduler被context switch, 所以使用該排程方式。

1.3 將各個process依照ready time排序

- 程式開始後每個時間點我都會check有哪些process已經ready了並把process產生出來，因此要先依照ready time來排序

1.4 Scheduler main loop

- 當scheduler的時間點有child process該結束時，等該child的pid，並把該process從queue pop出來，queue的head移到下一個要執行的process
- 檢查scheduler的該時間點有沒有process已經ready, 當ready時就fork出child process。將child process設定在1號cpu上，並將此child process設成priority idle 並把nice value設為39。
- scheduler 依照指定的schedule strategy檢查這個時間點應該要執行的process編號，把要執行的process的priority以SCHE式排程，其他使用SCHED_IDLE方式，讓他得到很低的執行priority
- scheduler 執行1 unit of time的時間，running process的剩餘執行時間減1

1.5 Scheduling Algorithm

1.5.1 FIFO

- 當有新的process ready時，此process enqueue to ready queue最後面。當要執行process時，peap ready queue以得到目前執行的process. 當process 執行完成時把以完成的process dequeue。不斷重複此流程直到所有process都被執行完。

1.5.2 RR

- 當有新的process ready時，enqueue在queue最後方，每個process執行500 unit, 若還沒完成則將running process從queue中先dequeue再enqueue, 先換給下個process執行。重複此流程直到所有process都被順利執行完。

1.5.3 SJF

- 當現在時間點沒有running process時，在目前的ready queue裡找execution time 最短的process來執行。執行完後就將此process從queue中移除。重複此流程直到所有process執行完成。

1.5.4 PSJF

- 在目前的時間點中，找出剩餘執行時間最短的process來執行，如果找到了比目前running process不同剩餘時間更短的process就發生context switch. 將所有目標process執行完後結束

1.6 Self Design System Call

- time_pj1, System call number 333 可將系統目前時間回傳
- prink_pj1, System call number 334 將含Project1 tag的log印到dmesg當中

2 核心版本

Ubuntu 16.04.6-desktop-amd64.iso compile with kerne linux-4.14.25

3 比較實際結果與理論結果

- 對所有測資的實驗結果可在資料夾中ExpResult.txt找到理論值和真正值的差別
- 誤差來源:
  - CPU 執行同樣次數迴圈時，每次的時間還是略有不同
  - 當發生context switch時，舊的process離開，新的process進CPU會造成誤差時間，process的執行時間會比理想值還久
  - 當running porcess和idle process的priority差距不夠大的時候，idle process還是會有share 到cpu的時間，就造成偷跑的情形，這時process的完成時間就會比理論值短，但這個問題可以由調整好priority有效解決
- 實驗結果(Base on test data set)
  對實際process結束時間值和理想值做比較，並平均每筆測資的abs誤差時間

	test 1	test 2	test 3	test 4	test 5
FIFO	50.835	1958.475	161.610	30.534	165.979
RR	19.162	191.302	586.493	288.283	315.865
SJF	115.235	55.200	581.919	101.540	33.484
PSJF	210.254	145.553	12.663	97.860	130.210

- Average Error for different scheduling strategy

	FIFO	RR	SJF	PSJF
Average Error	473.4866	280.221	177.475	119.308

- 以上實驗數字單位皆是基於我在我的機器上測出的unit of time與理想值的計算
- 從實驗結果中可以發現與課本理論結果相同與不同之處
  原本預期RR的average error應該要最大，但發現因為FIFO是先到的先做，FIFO的第二筆測資時間很長，cpu在中間可能稍微被context switch幾次後累積下來就差了很多。因為前一筆proess的延遲，也導致排在長process後面的短process的delay也變得很大，導致整個平均值都被一筆測試拉高。也印證了FIFO的缺點是短的process會因為前面有長的process而受到拖延
  RR演算法在各個測試裡的平均error時間都比較大，應是因為做RR時會需要多次context switch, 而context switch時就無法好好運用CPU時間
  SJF有效改善FIFO的缺點，短的process先做，短process的delay也小，因此overall 的error time顯著降低
  PSJF 本來以為會因為需要context switch而造成效能下降，但看來在我們的test case 中context switch並不常發生，採用此種作法更能有效運用CPU,得到最好的效能