Chapter 4 – Homework

Die Aufgaben wurden im X2GO Container auf dem Linux System bearbeitet und gelöst.

1.

Run process-run.py with the following flags: -l 5:100,5:100. What should the CPU utilization be (e.g., the percent of time the CPU is in use?) Why do you know this? Use the -c and -p flags to see if you were right.

Die CPU-Nutzung liegt bei 100%, da bei jedem Prozess jeweils 5 Instructions ausgeführt werden und 100% der Instructions von der CPU ausgeführt werden, statt der I/O.

```
to852sch@ct-bsys-ss20-15:~/Desktop/HW-CPU-Intro$ python process-run.py -l 5:100,
5:100 -c -p
Time     PID: 0     PID: 1        CPU        IOs
  1     RUN:cpu      READY          1
  2     RUN:cpu      READY          1
  3     RUN:cpu      READY          1
  4     RUN:cpu      READY          1
  5     RUN:cpu      READY          1
  6        DONE    RUN:cpu          1
  7        DONE    RUN:cpu          1
  8        DONE    RUN:cpu          1
  9        DONE    RUN:cpu          1
 10        DONE    RUN:cpu          1

Stats: Total Time 10
Stats: CPU Busy 10 (100.00%)
Stats: IO Busy  0 (0.00%)

to852sch@ct-bsys-ss20-15:~/Desktop/HW-CPU-Intro$
```

Wenn man nun das Python Programm mit dem flag -c -p laufen lässt, sieht man in welchem Prozess wann die CPU benutzt wird. Dies ist im Bild Rot unterstrichen. Der erste Prozess wird von dem Prozessor durchlaufen. Im Anschluss wird direkt der zweite Prozess laufen gelassen vom Prozessor, während der andere im Status „Done“ verweilt.

2.

Now run with these flags: ./process-run.py -l 4:100,1:0. These flags specify one process with 4 instructions (all to use the CPU), and one that simply issues an I/O and waits for it to be done. How long does it take to complete both processes? Use -c and -p to find out if you were right.

Die beiden Prozesse haben eine Dauer von 10 clock ticks.

Der erste Prozess mit einer 100% CPU Usage hat eine Zeit von 4 clock ticks. Der zweite Prozess benötigt im Start 1 Tick und nochmal 4 weitere Ticks bis er fertig gewartet hat, da ein I/O Call gemäß Readme eine Standardlänge von 5 Ticks hat. Der Simulator hängt wegen dem I/0 Call ein leeren Tick am Ende des I/O Calls hinzu, daher ist dieses mit einem Sternchen versehen und von uns Blau markiert.

```
lu851not@ct-bsys-ss20-15:~/z-drive/3.Semester/BSYS/ProcessRun-HW 1$ ./process-run.py -l 4:100,1:0 -c -p
Time     PID: 0     PID: 1        CPU        IOs
  1     RUN:cpu      READY          1
  2     RUN:cpu      READY          1
  3     RUN:cpu      READY          1
  4     RUN:cpu      READY          1
  5        DONE     RUN:io          1
  6        DONE    WAITING                    1
  7        DONE    WAITING                    1
  8        DONE    WAITING                    1
  9        DONE    WAITING                    1
 10*       DONE       DONE

Stats: Total Time 10
Stats: CPU Busy 5 (50.00%)
Stats: IO Busy  4 (40.00%)
```

3.

Switch the order of the processes: -l 1:0,4:100. What happens now? Does switching the order matter? Why? (As always, use -c and -p to see if you were right)

Ja, die Reihenfolge der Prozesse spielt eine Rolle. Die beiden Prozesse haben nur noch eine Dauer von 6 Ticks. Die Wartezeit des ersten Prozesses wird genutzt um den zweiten Prozess laufen zu lassen.

Nach dem fünften Tick ist sowohl die Wartezeit vorbei als auch der zweite Prozess fertig gelaufen. Der Simulator hängt auch hier wegen dem I/O Call ein Tick ans Ende der I/O und kommt anschließend auf eine Laufzeit von 6 Ticks.

```
lu851not@ct-bsys-ss20-15:~/z-drive/3.Semester/BSYS/ProcessRun-HW 1$ ./process-run.py -l 1:0,4:100 -c -p
Time     PID: 0     PID: 1        CPU        IOs
  1      RUN:io      READY          1
  2     WAITING    RUN:cpu          1          1
  3     WAITING    RUN:cpu          1          1
  4     WAITING    RUN:cpu          1          1
  5     WAITING    RUN:cpu          1          1
  6*       DONE       DONE

Stats: Total Time 6
Stats: CPU Busy 5 (83.33%)
Stats: IO Busy  4 (66.67%)
```

4.

We'll now explore some of the other flags. One important flag is -S, which determines how the system reacts when a process issues an I/O. With the flag set to SWITCH ON END, the system will NOT switch to another process while one is doing I/O, instead waiting until the process is completely finished. What happens when you run the following two processes (-l 1:0,4:100 -c -S SWITCH ON END), one doing I/O and the other doing CPU work?

Da mit den Flags der Prozess immer zu Ende ausgeführt wird, werden nun 9 Ticks benötigt, um die beiden Prozesse zu vollenden. 5 Ticks für den ersten Prozess mit I/O und 4 Ticks für den zweiten.

Beide Prozesse laufen erst vollständig zu Ende. Erst wenn der Prozess fertig ist, kann der andere anfangen.

```
to852sch@ct-bsys-ss20-15:~/Desktop/HW-CPU-Intro$ python process-run.py -l 1:0,4:100 -
c -S SWITCH_ON_END
Time      PID: 0     PID: 1        CPU        IOs
  1      RUN:io      READY          1
  2     WAITING      READY                     1
  3     WAITING      READY                     1
  4     WAITING      READY                     1
  5     WAITING      READY                     1
  6*       DONE    RUN:cpu          1
  7        DONE    RUN:cpu          1
  8        DONE    RUN:cpu          1
  9        DONE    RUN:cpu          1
to852sch@ct-bsys-ss20-15:~/Desktop/HW-CPU-Intro$
```

5.

Now, run the same processes, but with the switching behavior set to switch to another process whenever one is WAITING for I/O (-l 1:0,4:100 -c -S SWITCH ON IO). What happens now? Use -c and -p to confirm that you are right.

Wie erwartet springt die CPU um, wenn der erste Prozess in den Waiting Status übergeht und lässt den Prozess laufen.

```
lu851not@ct-bsys-ss20-15:~/z-drive/3.Semester/BSYS/ProcessRun-HW 1$ ./process-run.py -l 1:0,4:100 -c -p -S S
WITCH_ON_IO
Time      PID: 0     PID: 1        CPU        IOs
  1      RUN:io      READY          1
  2     WAITING    RUN:cpu          1          1
  3     WAITING    RUN:cpu          1          1
  4     WAITING    RUN:cpu          1          1
  5     WAITING    RUN:cpu          1          1
  6*       DONE       DONE

Stats: Total Time 6
Stats: CPU Busy 5 (83.33%)
Stats: IO Busy  4 (66.67%)
```

6.

One other important behavior is what to do when an I/O completes. With -I IO RUN LATER, when an I/O completes, the process that issued it is not necessarily run right away; rather, whatever was running at the time keeps running. What happens when you run this combination of processes? (Run ./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH ON IO -I IO RUN LATER -c -p) Are system resources being effectively utilized?

Nein, die Ressourcen wurden nicht richtig genutzt. Zuerst wurde der I/O Call ausgeführt. Als dieser in den Waiting Modus übergegangen ist, ist der erste Prozess eingesprungen und hat die Wartezeit genutzt. Allerdings wurden im Anschluss die weiteren beiden Prozesse mit CPU-Usage ausgeführt.

Zum Schluss wurden dann erst die letzten beiden I/O Calls ausgeführt. Die Beiden letzten I/O Calls hätten mit den anderen beiden Prozessen gepaart werden können.

So wurden nun 26 Ticks benötigt + 1 Tick für den Done Status nach dem I/O.

```
to852sch@ct-bsys-ss20-15:~/Desktop/HW-CPU-Intro$ python process-run.py -l 3:0,5:100,5
:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p
Time     PID: 0     PID: 1     PID: 2     PID: 3        CPU        IOs
  1      RUN:io      READY      READY      READY          1
  2     WAITING    RUN:cpu      READY      READY          1          1
  3     WAITING    RUN:cpu      READY      READY          1          1
  4     WAITING    RUN:cpu      READY      READY          1          1
  5     WAITING    RUN:cpu      READY      READY          1          1
  6*      READY    RUN:cpu      READY      READY          1
  7       READY       DONE    RUN:cpu      READY          1
  8       READY       DONE    RUN:cpu      READY          1
  9       READY       DONE    RUN:cpu      READY          1
 10       READY       DONE    RUN:cpu      READY          1
 11       READY       DONE    RUN:cpu      READY          1
 12       READY       DONE       DONE    RUN:cpu          1
 13       READY       DONE       DONE    RUN:cpu          1
 14       READY       DONE       DONE    RUN:cpu          1
 15       READY       DONE       DONE    RUN:cpu          1
 16       READY       DONE       DONE    RUN:cpu          1
 17      RUN:io       DONE       DONE       DONE          1
 18     WAITING       DONE       DONE       DONE                     1
 19     WAITING       DONE       DONE       DONE                     1
 20     WAITING       DONE       DONE       DONE                     1
 21     WAITING       DONE       DONE       DONE                     1
 22*     RUN:io       DONE       DONE       DONE          1
 23     WAITING       DONE       DONE       DONE                     1
 24     WAITING       DONE       DONE       DONE                     1
 25     WAITING       DONE       DONE       DONE                     1
 26     WAITING       DONE       DONE       DONE                     1
 27*       DONE       DONE       DONE       DONE

Stats: Total Time 27
Stats: CPU Busy 18 (66.67%)
Stats: IO Busy  12 (44.44%)
```

7.

Now run the same processes, but with -I IO RUN IMMEDIATE set, which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?

Das Ziel von IO_RUN_IMMEDIATE ist Laufzeit und Ressourcen zu sparen, da die Wartezeit von einem I/O Call genutzt werden kann, um einen anderen Prozess laufen zu lassen der eine CPU-Usage hat.

Wie man in der Simulation schön sieht, wird zuerst die I/O ausgeführt und in der Wartezeit der nächste Prozess.

Der Prozess wird allerdings nach der WAITING Zeit unterbrochen um wieder eine I/O auszuführen. Auch hier wird wieder die WAITING Time genutzt, um einen anderen Prozess laufen zu lassen.

Die CPU ist 100% der Laufzeit beschäftigt was für eine optimale Nutzung spricht. Dies spiegelt sich auch in der Tickzahl bzw. der Laufzeit dar. Statt 26 wurden nur 18 Ticks benötigt.

Ja, es ist eine gute Idee I/O Calls so früh wie möglich zu bearbeiten, um mögliche unnötige Wartezeiten zu vermeiden.

```
to852sch@ct-bsys-ss20-15:~/Desktop/HW-CPU-Intro$ python process-run.py -l 3:0,5:100,5
:100,5:100 -S SWITCH_ON_IO -I IO_RUN_IMMEDIATE -c -p
Time     PID: 0     PID: 1     PID: 2     PID: 3        CPU        IOs
  1      RUN:io      READY      READY      READY          1
  2     WAITING    RUN:cpu      READY      READY          1          1
  3     WAITING    RUN:cpu      READY      READY          1          1
  4     WAITING    RUN:cpu      READY      READY          1          1
  5     WAITING    RUN:cpu      READY      READY          1          1
  6*     RUN:io      READY      READY      READY          1
  7     WAITING    RUN:cpu      READY      READY          1          1
  8     WAITING       DONE    RUN:cpu      READY          1          1
  9     WAITING       DONE    RUN:cpu      READY          1          1
 10     WAITING       DONE    RUN:cpu      READY          1          1
 11*     RUN:io       DONE      READY      READY          1
 12     WAITING       DONE    RUN:cpu      READY          1          1
 13     WAITING       DONE    RUN:cpu      READY          1          1
 14     WAITING       DONE       DONE    RUN:cpu          1          1
 15     WAITING       DONE       DONE    RUN:cpu          1          1
 16*       DONE       DONE       DONE    RUN:cpu          1
 17        DONE       DONE       DONE    RUN:cpu          1
 18        DONE       DONE       DONE    RUN:cpu          1

Stats: Total Time 18
Stats: CPU Busy 18 (100.00%)
Stats: IO Busy  12 (66.67%)
```

8.

Now run with some randomly generated processes: -s 1 -l 3:50,3:50 or -s 2 -l 3:50,3:50 or -s 3 -l 3:50,3:50. See if you can predict how the trace will turn out. What happens when you use the flag -I IO RUN IMMEDIATE vs. -I IO RUN LATER? What happens when you use -S SWITCH ON IO vs. -S SWITCH ON END?

-I IO RUN IMMEDIATE vs. -I IO RUN LATER

IO_RUN_LATER und IO_RUN_IMMEDIATE ergeben bei den Beispielen keinen Unterschied in der Laufzeit. Dies liegt daran, dass die Prozesse meistens warten.

-S SWITCH ON IO vs. -S SWITCH ON END

SWITCH_ON_IO ist selbstverständlich viel schneller, da SWITCH_ON_END erst wartet bis der I/O Call fertig ist und nicht die Zeit im „Waiting“ Zustand nutzen kann.