ES3/IPC - Opdracht 7. Process Scheduling

In deze opdracht raak je vertrouwd met het begrip process scheduling, d.w.z. hoe het operating systeem de processortijd verdeeld tussen processen.

# Voorbereidingen

Bestudeer Stallings paragraaf 9.2, 10.2, 10.3.

# Taken

## Process Scheduling

Ga naar de volgende link: <http://courses.cs.vt.edu/csonline/OS/Lessons/Processes/ProcessStateDiagram.swf>

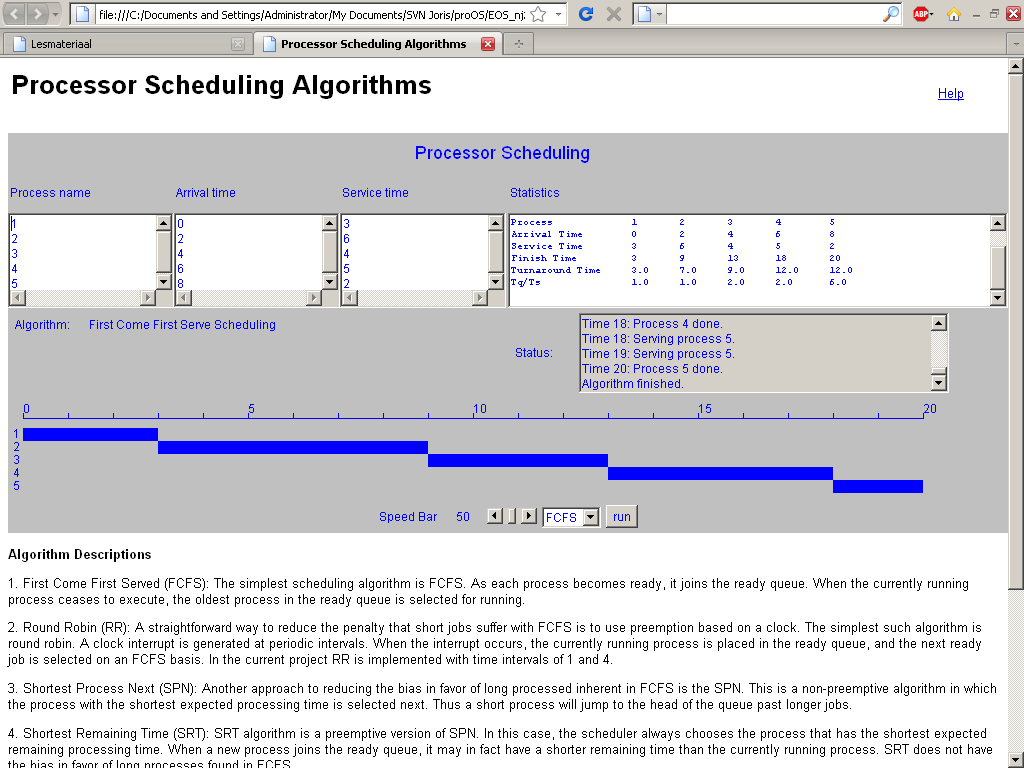
en beantwoord de volgende vragen:

* welk process schedulings algoritme wordt toegepast?
  + O(n) / time slice / epoch algoritme, elk proces heeft een time slice om zijn ding te doen.
* voor welke activiteiten komt het OS (de pinguïn "Tux") in de Running state?
  + Processen opstarten/laden, schedulen, kiezen welke runt, processen uit memory gooien

## Process Scheduling Algorithms

Downloadl IPC32\_pscheduler.zip en start de applet pscheduling.html (onderaan de html pagina staat uitleg).

Let er op dat je een grijs veld ziet zoals op dit screenshot, anders staat je beveiliging van Java Applets te streng. Loop je daar tegen aan kijk dan hier hoe je die settings kunt aanpassen: <https://www.java.com/en/download/help/jcp_security.xml>

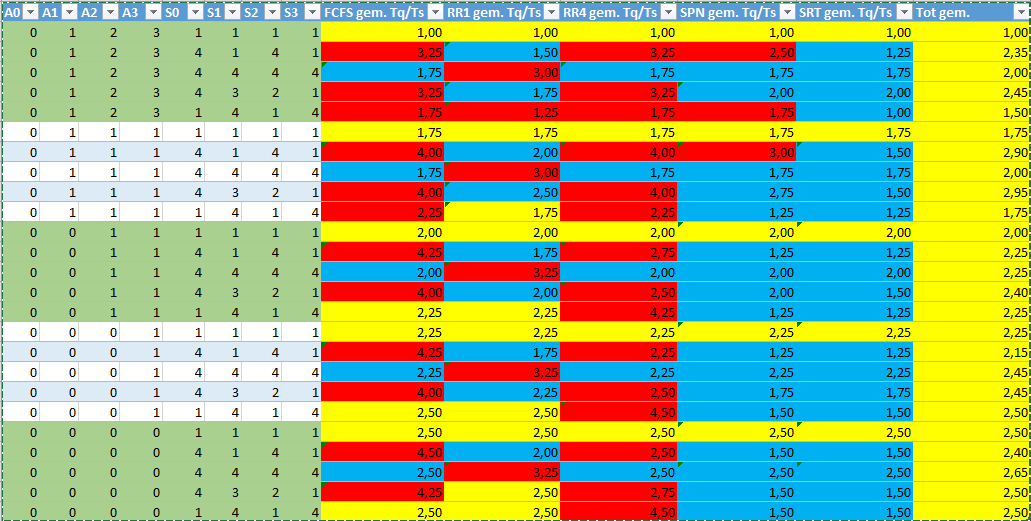


Opdrachten:

* Maak voor elk algoritme een scenario waarmee dit algoritme *wint* van de anderen. En maak voor elk algoritme een scenario waarmee dit algoritme *verliest* van de anderen. Er worden verschillende arrival times (4x) en verschillende service times (4x) uitgekozen, alle mogelijke combinaties staan in onderstaand tabel, deze worden getest:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A0** | **A1** | **A2** | **A3** | **S0** | **S1** | **S2** |  | **S3** |
| 0 | 1 | 2 | 3 | 1 | 1 | 1 |  | 1 |
| 0 | 1 | 2 | 3 | 4 | 1 | 4 |  | 1 |
| 0 | 1 | 2 | 3 | 4 | 4 | 4 |  | 4 |
| 0 | 1 | 2 | 3 | 4 | 3 | 2 |  | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 |  | 1 |
| 0 | 1 | 1 | 1 | 4 | 1 | 4 |  | 1 |
| 0 | 1 | 1 | 1 | 4 | 4 | 4 |  | 4 |
| 0 | 1 | 1 | 1 | 4 | 3 | 2 |  | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 |  | 1 |
| 0 | 0 | 1 | 1 | 4 | 1 | 4 |  | 1 |
| 0 | 0 | 1 | 1 | 4 | 4 | 4 |  | 4 |
| 0 | 0 | 1 | 1 | 4 | 3 | 2 |  | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |  | 1 |
| 0 | 0 | 0 | 1 | 4 | 1 | 4 |  | 1 |
| 0 | 0 | 0 | 1 | 4 | 4 | 4 |  | 4 |
| 0 | 0 | 0 | 1 | 4 | 3 | 2 |  | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 |  | 1 |
| 0 | 0 | 0 | 0 | 4 | 1 | 4 |  | 1 |
| 0 | 0 | 0 | 0 | 4 | 4 | 4 |  | 4 |
| 0 | 0 | 0 | 0 | 4 | 3 | 2 |  | 1 |

* Kies zelf een norm voor "verliezen" en "winnen": gaat het om de *gemiddelde* Tq/Ts, of om een *maximale* Tq/Ts, of …?
  + Winnen = laagste gemiddelde(E{Tq/Ts}), dit is de relatieve vertraging, die wil je zo klein mogelijk houden
  + Dit is het resultaat van de metingen met Ax de Arrival time van proces met naam x en Sx de Service time van proces x
  + Resultaten (in het rood penalties voor te groot gemmidelde, in blauw uitstekende gemmidelden == plus punt):



Te zien is dat SRT altijd de laagste gemmidelde heeft en is hier overal winnaar, maar elke scheduling strategie is bedoeldt voor een ander doel. Indien er threads en prioriteiten zouden spelen zou SRT of SPN mogelijk niet zulke goede resultaten behalen. RR4 had de meeste uitweiking en laagste gemmidelden, deze was in de meeste gevallen de slechtste. FCFS schijnt ook veel uitweiking te hebben (bv in een run waren de tq/ts 1 4 5 10). RR1 is nogal iets tussen SRT en FCFS, het doet het gewoon overal gemmideld en mogelijk zou dat ook het meest geschikte kunnen zijn voor een universele approach.

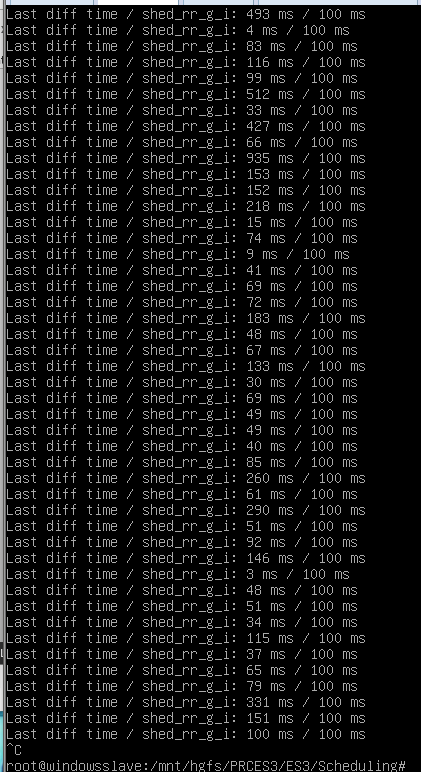
* De Java applet is niet helemaal natuurgetrouw.   
  Welk belangrijk aspect ontbreekt in de simulatie?   
  Wat heeft dat voor gevolgen?
  + Prioriteiten in processen ontbreken, zo kan je dus een heel belangrijk signaal missen. Ook missen de interrupts zodat een bepaald proces bij een bepaald signaal wordt opgewekt, deze functionaliteit mist dus en zo kan je niet een heel efficiënt slapend proces schrijven. Die zal dus volgens de berekeningen lage service hebben. Ook ontbreekt er multi core support, threading, etc, de resultaten zouden hierdoor helemaal anders kunnen zijn.

Voor de liefhebber (niet verplicht):

* Verbeter de tekortkomingen van de Java applet.

## Timeslicing

Schrijf een programma om de volgende vraag te onderzoeken:

* hoe lang duurt een time slice op jouw machine wanneer je de SCHED\_RR scheduling policy gebruikt?
  + 0.1 seconde
  + 

//Code:

#include <stdio.h>

#include <unistd.h>

#include <time.h>

#include <pthread.h>

static unsigned long long

convert\_time (struct timespec \* ts)

{

// convert 'ts' to microseconds

long long nsec;

long long sec;

// ensure that all items in the expression are 'long long'

nsec = ts->tv\_nsec;

sec = ts->tv\_sec;

return (nsec + sec \* 1000000000LL);

}

static unsigned long long

get\_time (void)

{

// get the elapsed time (in microseconds)

struct timespec ts;

clock\_gettime (CLOCK\_REALTIME, &ts);

return (convert\_time (&ts));

}

extern int

main ()

{

unsigned long long time\_last = get\_time();

unsigned long long difference = 0;

while (1)

{

long long t\_now = get\_time();

difference = get\_time() - t\_now;

if (difference > 100000)

{

struct timespec ts;

sched\_rr\_get\_interval(getpid(), &ts);

printf("Last diff time / shed\_rr\_g\_i: %llu ms / %llu ms\n", (t\_now - time\_last) / 1000000, convert\_time(&ts) / 1000000);

time\_last = t\_now;

}

}

return (0);

}

Je kunt bijvoorbeeld het programma dutycycle van de eerste week nemen en slim uitbreiden. Het programma heeft een while lus waarin continu de huidige tijd wordt opgevraagd. Zolang het proces met zijn time slice bezig is, zullen de tijden erg dicht bij elkaar liggen. Als er een “gat” (van tenminste enkele tientallen milliseconden) is tussen twee opeenvolgende tijden, dan hebben blijkbaar in dat tijdsgat één of meer andere processen gedraaid. Hoe lang heeft het proces wel mogen draaien?

Let op: meet gedurende een bepaalde tijd maar schrijf niet elke tijdmeting direct met printf naar de output, want dat kost veel tijd en dat verstoort de meting.

Je kunt vanuit de shell een proces met een bepaalde scheduling policy en prioriteit opstarten door middel van het Linux commando *chrt* (zie: <http://linux.die.net/man/1/chrt>). Doe dit met sudo.

Ook kun je in het programma de POSIX functie *sched\_setscheduler* aanroepen (<http://linux.die.net/man/2/sched_setscheduler>).

Je moet een aantal instanties (tenminste 2) van dit programma opstarten. Bedenk daarbij ook dat je wellicht meer dan één CPU-core hebt.

Start bij voorkeur de processen vanuit één terminal window, zodat je zeker weet dat de scheduler ze met elkaar laat concurreren. Je kunt ook de kernel instrueren om processen uit verschillende terminals met elkaar te laten concurreren: *echo 0 > /proc/sys/kernel/sched\_autogroup\_enabled*

Een andere commando wat hetzelfde doet: *sysctl -w kernel.sched\_autogroup\_enabled=0*

Meer over scheduler tweaking kun je hier vinden: <http://tweaked.io/guide/kernel/>

Een ander idee is om een eigen testprogramma te schrijven met een aantal threads die elk in een while lus de tijd opvragen en tijdsgaten registreren zoals boven beschreven. Hoe je een aantal threads met een bepaalde threadpolicy moet opstaren kun je in het volgende voorbeeldprogramma vinden: <http://www.eg.bucknell.edu/~cs315/Fall13/code/scheduling/posix-rt.c>

Je hoeft niet in de Linux kernel te hacken; en de getallen hoeven ook niet tot op de laatste decimaal nauwkeurig te zijn.

De echte timeslice (time quantum) van de SCHED\_RR scheduler kun je opvragen door middel van de *sched\_rr\_get\_interval* call. Dit moet natuurlijk kloppen met de uitkomst van jouw programma. Hiermee kun je dus je programma ijken.

# Opleveren

Je dient een document op te leveren waarin je de bovenstaande experimenten uitwerkt. Laat zien dat je een en andere goed onderzocht hebt. Voeg, waar nodig, screendumps toe om een en andere duidelijk te maken. Lever ook de bijbehorende source code in van de programma’s die je gemaakt of aangepast hebt.