Aarhus Universitet
Datalogisk Institut
Ny Munkegade
8000 Århus C

11. december 2003

Arkitektur og Operativsystemer

Obligatorisk opgave

Obligatorisk opgave besvaret af	Clausen, Christoph	årskort 20020180	hold 4
	Jonigkeit, Christian	årskort 20021232	hold 4
	Winther Lauritzen, Rasmus	årskort 20021859	hold 4.

Besvarelsen omfatter 27 nummererede sider.

Rettidigt afleveret ved forelæsningen tordag den 11. december 2003 kl. 9.15

1 Indledning

Denne rapport er besvarelse af obligatorisk opgave i kurset Arkitektur og Operativsystemer, efteråret 2002. Læseren forudsættes at kende opgaveformuleringen samt bogen Modern Operating Systems, 2. udgave, af Andrew S. Tanenbaum.

2 Ændringer i Linux kernen

Linux-kernen version 2.2.14 er blevet modificeret ved at tilføje systemkaldene `darkos_vm_stat` og `darkos_vm_set`. En patch der beskriver samtlige modifikationer til den originale kerne-kildetekst er vist i appendiks A.

2.1 Tilføjelse af systemkaldet `darkos_vm_stat`

Systemkaldet `darkos_vm_stat` er tilføjet Linux kernen. For et givet proces-id aflæser det processens `nswap`, som angiver hvor mange sider processen pt. har swappet ud, og `rss`, der fortæller, hvor mange sider processen har i det fysiske lager.

Systemkaldet tager tre argumenter: proces-id samt to pointere til variable af typen long, hvori de aflæste værdier for `nswap` og `rss` gemmes. Descriptorstrukturen (`task_struct`) for det givne proces-id findes vha. kernefunktionen `find_task_by_pid`, der laver et opslag i en hashtabel, hvilket kan gøres i konstant tid. Hvis processen ikke findes, returnerer `find_task_by_pid` en null-pointer, og `darkos_vm_stat` returnerer fejlkoden `ESRCH` (No Such Process). Ellers har vi nu en descriptor-struktur, hvorfra vi får `rss` og `nswap`, som derefter kopieres til userspace vha. `copy_to_user`. Sker der en fejl under kopieringen, returneres fejlkoden `EFAULT` (Bad Address). Sker der ingen fejl, er alt gået som forventet, og der returneres 0.

I `arch/i386/kernel/entry.S` og `include/asm-i386/unistd.h` er systemkaldet blevet tildelt nummeret 191. I `kernel/sched.c` er tilføjet følgende:

```
asmlinkage int darkos_vm_stat(pid_t pid, unsigned long *usrrss,
    unsigned long *usrnswap)
{
  unsigned long rss, nswap;
  struct task_struct *task;
5 int err = 0;

  task = find_task_by_pid(pid);

  if(task == NULL) return -ESRCH; // pid doesn't exist
10
  rss = task->mm->rss;
  nswap = task->nswap;
```

```
    // copying values to userspace
15  err = copy_to_user(usrrss, &rss, sizeof(rss));
    err += copy_to_user(usrnswap, &nswap, sizeof(nswap));

    // on error return "Bad Address"
    return (err != 0)? -EFAULT : 0;
20 }
```

I appendiks B er vist et C-program, `darkos_vm_stat.c`, der tager et proces id. (PID) som kommandolinie-argument og udskriver `nswap` og `rss` for den tilsvarende proces, hvis den findes.

2.2 Tilføjelse af systemkaldet `darkos_vm_set`

Systemkaldet `darkos_vm_set` er tilføjet Linux kernen. Systemkaldet tager et positivt heltal k som parameter og sætter `darkos_swap_frac` til k. `darkos_swap_frac` er en global variabel, som vi har indført som modifikation til Linux' swapping-strategi.

Swap-algoritmen i Linux-kernen virker således, at processen med det største `swap_cnt` skal frigive en side i det fysiske lager, jf. opgaveformuleringen. Hvis en proces har et højt `swap_cnt`, vil den således gentagne gange skulle afgive sider i den fysiske hukommelse, indtil den ikke kan afgive flere og `swap_cnt` sættes til 0.

Man kan opnå en mere "fair" swapping-strategi ved at reducere `swap_cnt`, når en af processens sider er blevet swappet ud. Fair betyder her, at det ikke altid er den samme proces, der skal afgive sider[1].

Da `darkos_swap_frac` er unsigned, behøver vi ikke tage hensyn til tilfældet, hvor `darkos_swap_frac` er mindre end 0. I tilfældet, hvor det er lig med 0, ændres `swap_cnt` ikke, så det svarer til den oprindelige strategi. Er `darkos_swap_frac` større end nul, opdateres `swap_cnt` som følgende.

$$swap_cnt = swap_cnt - \begin{cases} swap_cnt & \text{hvis } \dfrac{swap_cnt}{darkos_swap_frac} = 0 \\ \dfrac{swap_cnt}{darkos_swap_frac} & \text{ellers} \end{cases}$$

Det synes umiddelbart meget vilkåligt bare at sætte `swap_cnt` = 0, når heltalsdivisionen giver nul. Man kan også håndtere problemet med heltalsdivisionen på andre måder. For eksempel kan man vælge at trække en 1 fra – den mindst mulige værdi. Man kan også bare lade `swap_cnt` være uændret. I begge tilfælde vil `swap_cnt` før eller senere blive nul, fordi processen ikke kan swappe flere sider ud, eller fordi man trækker et lille tal fra. Vi sørger således bare for, at det sker

[1] Bemærk, at hverken den nye eller den gamle strategi er særligt effektive, idet de ikke tager hensyn til, om de sider, der swappes ud, måske bliver brugt indenfor den nærmeste tid.

hurtigere. For de små værdier af `darkos_swap_frac`, som vi bruger, betyder det ikke så meget.

I `arch/i386/kernel/entry.S` og `include/asm-i386/unistd.h` er systemkaldet blevet tildelt nummeret 192[2]. I `mm/vmscan.c` er tilføjet følgende:

```
...

unsigned int darkos_swap_frac = 0;

...

static int swap_out(unsigned int priority, int gfp_mask)
{
  ...
  if (swap_out_process(pbest, gfp_mask)) {

    /* DARKOS JULEOPGAVE
     *
     * rss for the process has been reduced,
     * since swap_out_process has returned 1.
     *
     * We now want to reduce the probability
     * that another page is taken from the same
     * process the next time a page has to be
     * swapped out.
     *
     * This is accomplished by reducing swap_cnt
     * with a fraction darkos_swap_frac,
     * because it is always the
     * process with the highest swap_cnt that has
     * to release a page.
     */

    if((darkos_swap_frac) && (pbest->mm->swap_cnt > 0))
    pbest->mm->swap_cnt = pbest->mm->swap_cnt -
    (pbest->mm->swap_cnt/darkos_swap_frac == 0
    ? pbest->mm->swap_cnt : pbest->mm->swap_cnt/darkos_swap_frac);

    return 1;
  }
  ...
}

...

void darkos_vm_set(unsigned int k)
{
```

[2] Der var allerede et systemkald med dette nummer, men da antallet af totale systemkald var sat til noget lavere, gik vi ud fra, at det ikke bruges, hvilket vi fik bekræftet af en udtømmende `fgrep`

```
  darkos_swap_frac = k;
}
```

I appendiks C er vist et C-program, `darkos_vm_set.c`, der tager et heltal som kommandolinie-argument og sætter systemets swapping-strategi herefter.

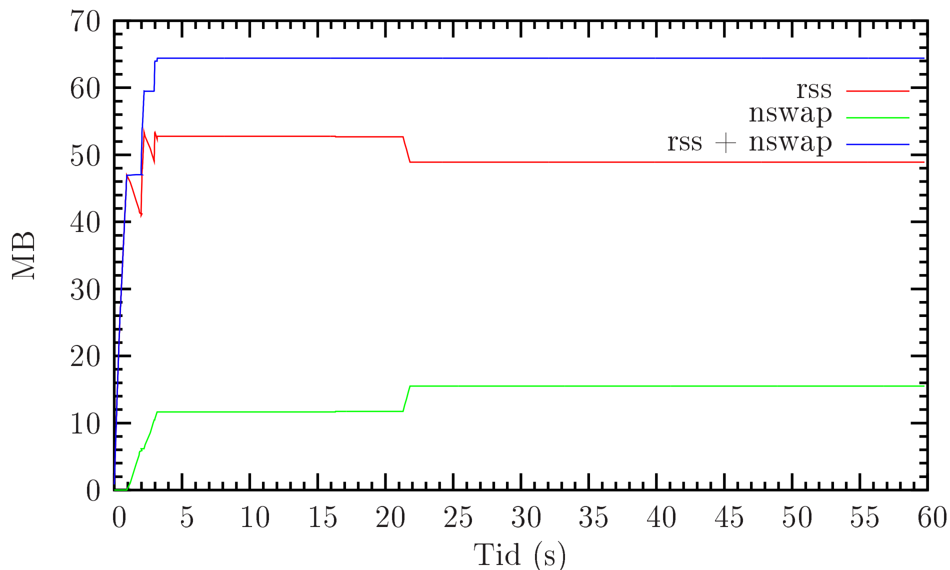
3 Eksperimenter

Der er blevet gennemført en række eksperimenter hvor der vha. det tilføjede systemkald `darkos_vm_stat` er blevet målt lagerforbrug og swap-aktivitet under afvikling af C-programmet `greedy.c` vist i appendiks D.

Programmet tager tre argumenter. Det første angiver, hvor mange megabyte hukommelse, `greedy` skal allokere i det hele. Anden parameter angiver, hvor mange sekunder programmet skal køre. Når den angivne mængde hukommelse er allokeret, går programmet i en uendelig løkke, indtil tiden er udløbet. Den sidste bestemmer, hvor mange sekunder `greedy` skal vente med allokeringen.

3.1 Swap-aktivitet for en enkelt proces

Programmet `greedy` med argumenter `64 60 0` er blevet afviklet på en dArkOS-Linux maskine med en uændret swapping-strategi. Grafen nedenfor afbilder de målte værdier for `nswap` og `rss` omregnet til Mb – en side svarer til 4096Kb – som funktion af målingstidspunktet.



Betragter man først den blå kurve, ses det, at der i alt allokeres 64Mb hukommelse til `greedy`, som forventet. Den røde viser, hvor meget fysisk lager, der er brugt.

Der er åbenbart en øvre grænse for, hvor meget fysisk lager en proces kan allokere, idet `rss` aldrig kommer over de 52Mb. Det passer godt med, at den virtuelle maskine har en samlet hukommelse på 64Mb, idet det lyder fornuftigt, at kernen bruger omkring 12Mb til sig selv.

Processen begynder dog at swappe ud, allerede inden denne grænse er nået, hvilket der kan ses på det første knæk. Her kan man aflæse, hvor meget fri hukommelse der var, da `greedy` startede.

Det interessante ved de to knæk i starten er, at `rss` aftager, medens `nswap` vokser og summen af de to er konstant. De tidsrum går altså med at swappe sider ud i stedet for at allokere ny hukommelse.

3.2 Swap-aktivitet for to processer

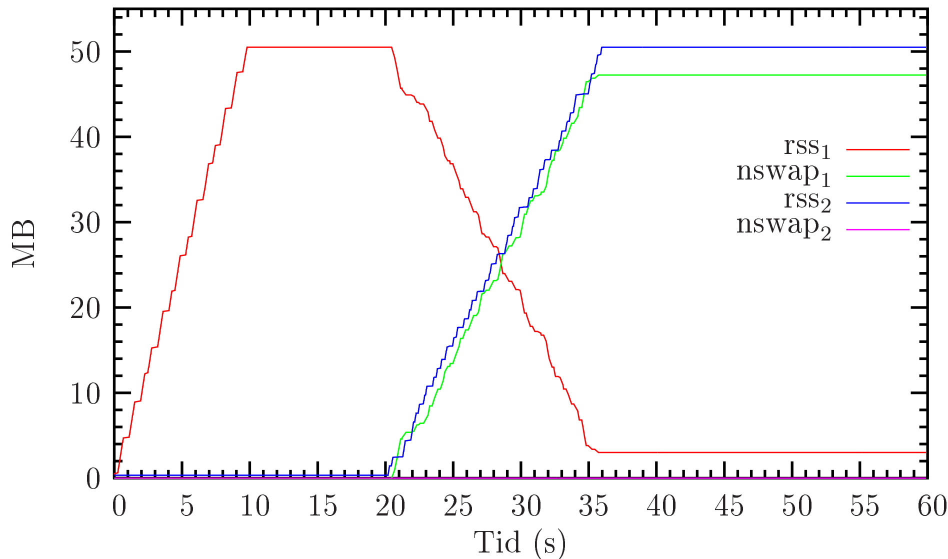
To instanser af programmet `greedy` som konkurrerer om pladsen i det fysiske lager er blevet afviklet på en dArkOS-Linux maskine med en uændret swapping-strategi. Den første instans allokerer plads indtil det fysiske lager er fyldt, hvorefter den anden instans begynder at allokere et tilsvarende antal Mb. Eksperimentet er blevet gennemført ved at afvikle nedenstående shell-script.

```
#!/bin/sh
./greedy 50 60 0 > greedy1.out &
./greedy 50 60 20 > greedy2.out
```

Først startes en instans af `greedy` som umiddelbart begynder allokering af ialt 50 Mb. Umiddelbart herefter startes en instans af `greedy`, som efter 20 sekunder begynder allokering af ialt 50 Mb. Den første instans startes i baggrunden, så de to instanser kommer til at konkurrere om siderne i det fysiske lager.

Ved at vælge ($x =$) 50Mb som allokeringsstørrelse sikrer vi, at den fysiske hukommelse fyldes, før anden instans af `greedy` starter, og at de to instanser konkurrerer om *hele* hukommelsen. Idet den anden instans af `greedy` først startes efter ($y =$) 20 sekunder, kan vi være sikre på, at den første greedy er blevet færdig med at allokere sin del af hukommelsen.

Grafen nedenfor afbilder de målte værdier for `nswap` og `rss` for de to instanser af `greedy` som funktion af målingstidspunktet.



På grafen ses, at anden instans af `greedy` starter efter 20 sekunder. Da hele hukommelsen er fyldt op, skal der gøres plads til hver enkelt side. Omkring dette tidspunkt tages der sandsynligvis et snapshot, da der ikke er mere fri hukommelse. Det er usandsynligt, at der er blevet lavet et snapshot før, da der var nok plads i det fysiske lager til den første `greedy`. Det vil altså sige, at den første `greedy` får tildelt et `swap_cnt` svarende til 50Mb.

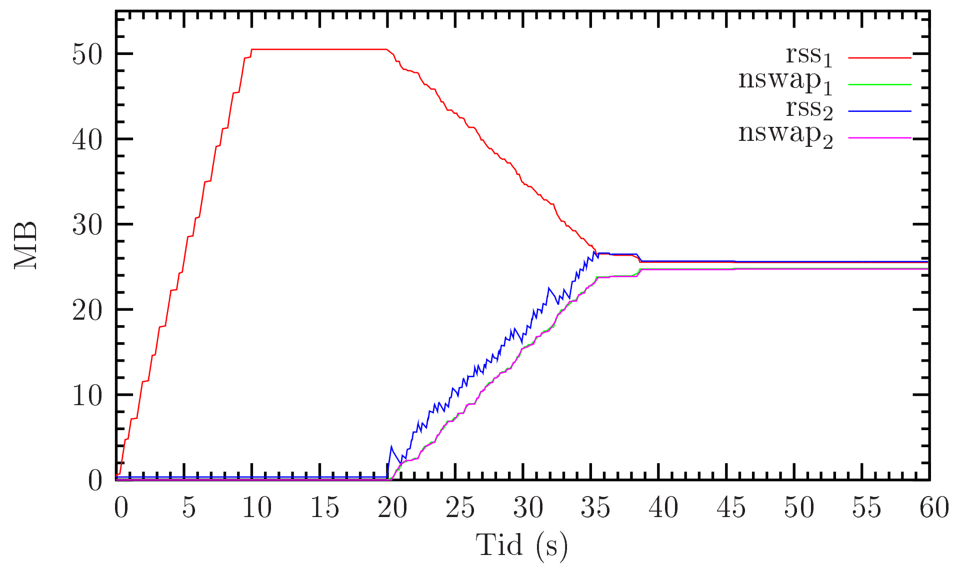
Den første instans af `greedy` må nu tydeligvis være den proces med det højeste `swap_cnt`, så det er dens sider, der bliver swappet ud. Da der således er nok sider at swappe ud, for at den anden instans af `greedy` kan få sin mængde hukommelse, laves der ikke et nyt snapshot i mellemtiden. Dette passer fint med, at den blå kurve og den grønne ligger nydeligt op ad hinanden, samtidigt med at den røde kurve aftager med omtrent samme hældning. Var der blevet lavet et nyt snapshot kort efter, at den anden `greedy` havde allokeret lidt over halvdelen af sit lager, ville dens egne sider være blevet swappet ud.

At den tilgængelige hukommelse til sidst rent faktisk er større end de 50Mb kan ses på, at den anden `greedy` kan allokere hele sine 50Mb, selvom den første stadig har 3 Mb i det fysiske lager. Dette forklarer også højdeforskellen mellem den grønne og den blå kurve dér, hvor de er vandrette.

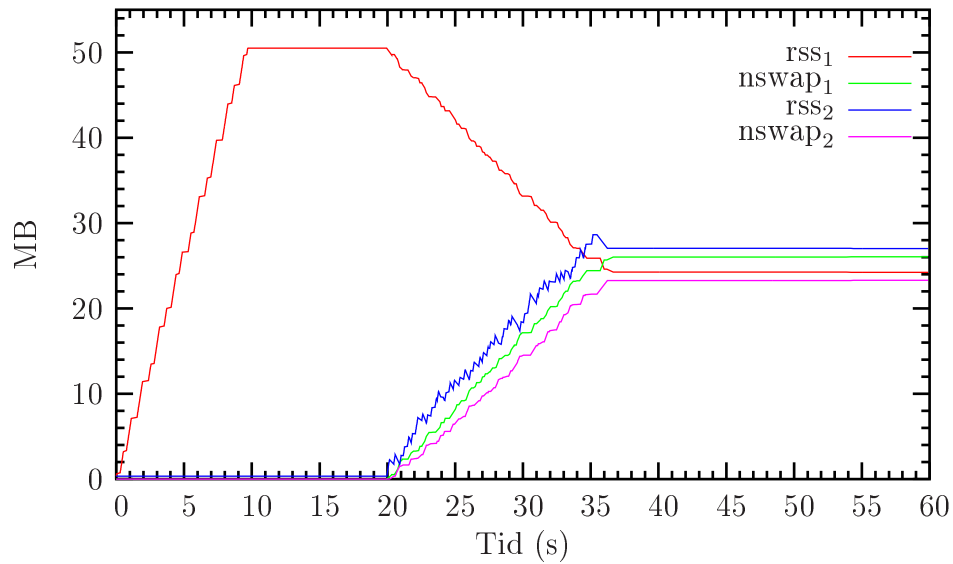
3.3 Swap-aktivitet for to processer med ny swapping-strategi

Programmet `darkos_vm_set.c` vist i appendiks C kan bruges til at ændre swapping-strategien som forklaret i afsnit 2.2. Måling og afbildning af swap-aktivitet for to processer, som beskrevet i forrige afsnit, er blevet gennenført for fem forskellige swapping-strategier svarende til `darkos_vm_set` med argument 1, 2, 4, 8, og 16.

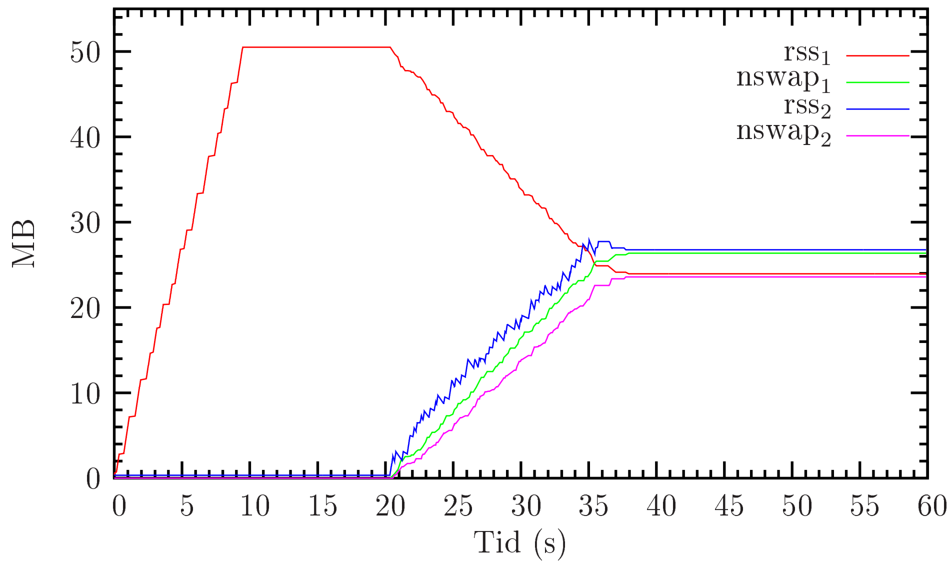
Resultat med swapping-strategi svarende til `darkos_vm_set 1`:



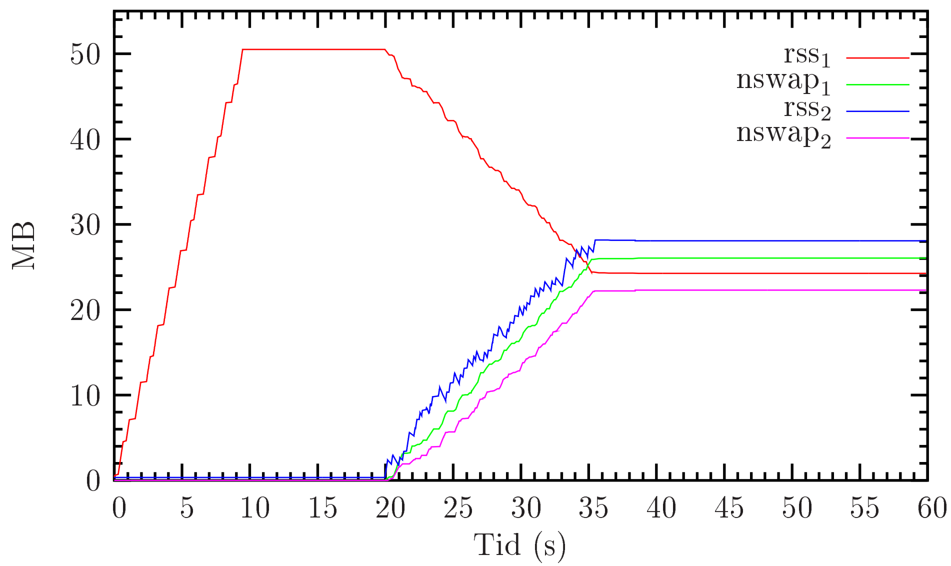
Resultat med swapping-strategi svarende til `darkos_vm_set 2`:



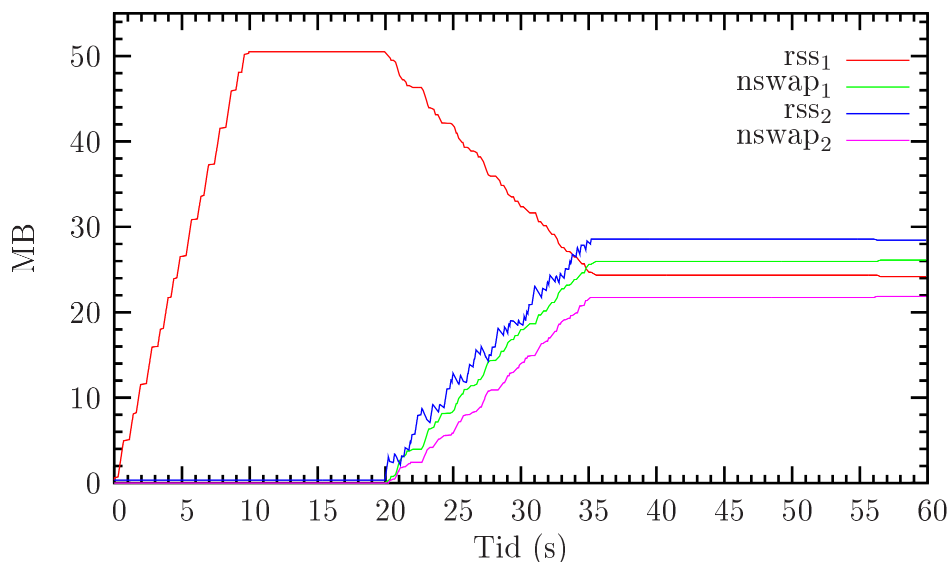
Resultat med swapping-strategi svarende til `darkos_vm_set 4`:



Resultat med swapping-strategi svarende til `darkos_vm_set 8`:



Resultat med swapping-strategi svarende til `darkos_vm_set 16`:



For `darkos_vm_set 1` forventer vi, at processerne alternerer strengt i afgivelse af sider, idet `swap_cnt` skiftevis sættes til nul. Dvs., først swapper første instans af `greedy` en side ud, så swapper anden instans en side ud, og derefter bliver der taget et nyt snapshot, så der startes forfra.

Vi forventer således, at `nswap`'erne ligger oven på hinanden, da begge processer hele tiden vil have swappet stort set det samme antal sider ud. Herfra kan vi også slutte, at de to `rss`'er vil have samme værdi, når begge processer har allokeret deres hukommelse ($rss + nswap = mbmax$).

Dette er præcist det, som grafen afbilder.

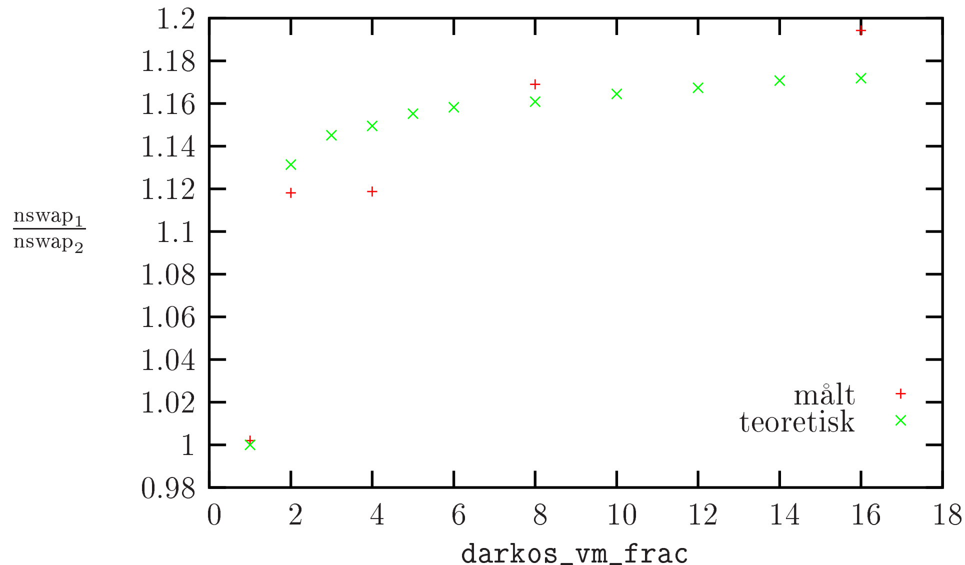
I stedet for at sætte processens `swap_cnt` helt til 0, halveres den for `darkos_vm_set 2`, når der er blevet swappet sider ud. Proces 1 swapper sider ud, indtil dens `swap_cnt` er mindre end `swap_cnt` fra den anden, vil det tage $\lceil \log_2(s_1/s_2) \rceil$ cykler[3] for efter et snapshot, inden der skiftes imellem de to. s_1 og s_2 angiver deres `swap_cnt`, regnet i antal sider, til tidspunktet af snapshottet. Det forklarer også, hvorfor den anden `greedy` næsten med det samme kommer til at swappe ud (se grafen), da der kun vil gå ca. $2\lceil \log_2 12000 \rceil = 28$ cykler, inden $\texttt{swap_cnt}_2 > \texttt{swap_cnt}_1$ (først efter andet snapshot vil s_2 være større 0). Når den første proces så at sige har indhentet den anden, vil der igen være streng alternering imellem de to, indtil der laves et nyt snapshot.

Konsekvensen er, at den proces, der starter med det største `swap_cnt` kommer til swappe flest sider ud (den bliver "straffet" mere).Grafen bekræfter dette, idet den endelige værdi af $\texttt{nswap}_1$ ligger højere end $\texttt{nswap}_2$.

[3]Skulle det ske, at s_1/s_2 er en potens af to, kræver det evt. en cykel mere. I starten er $s_2 = 0$, men kan til ovenstående beregning sættes til 1, da vi alligevel trækker den sidste fra pga. heltalsdivisionen (se ovenfor).

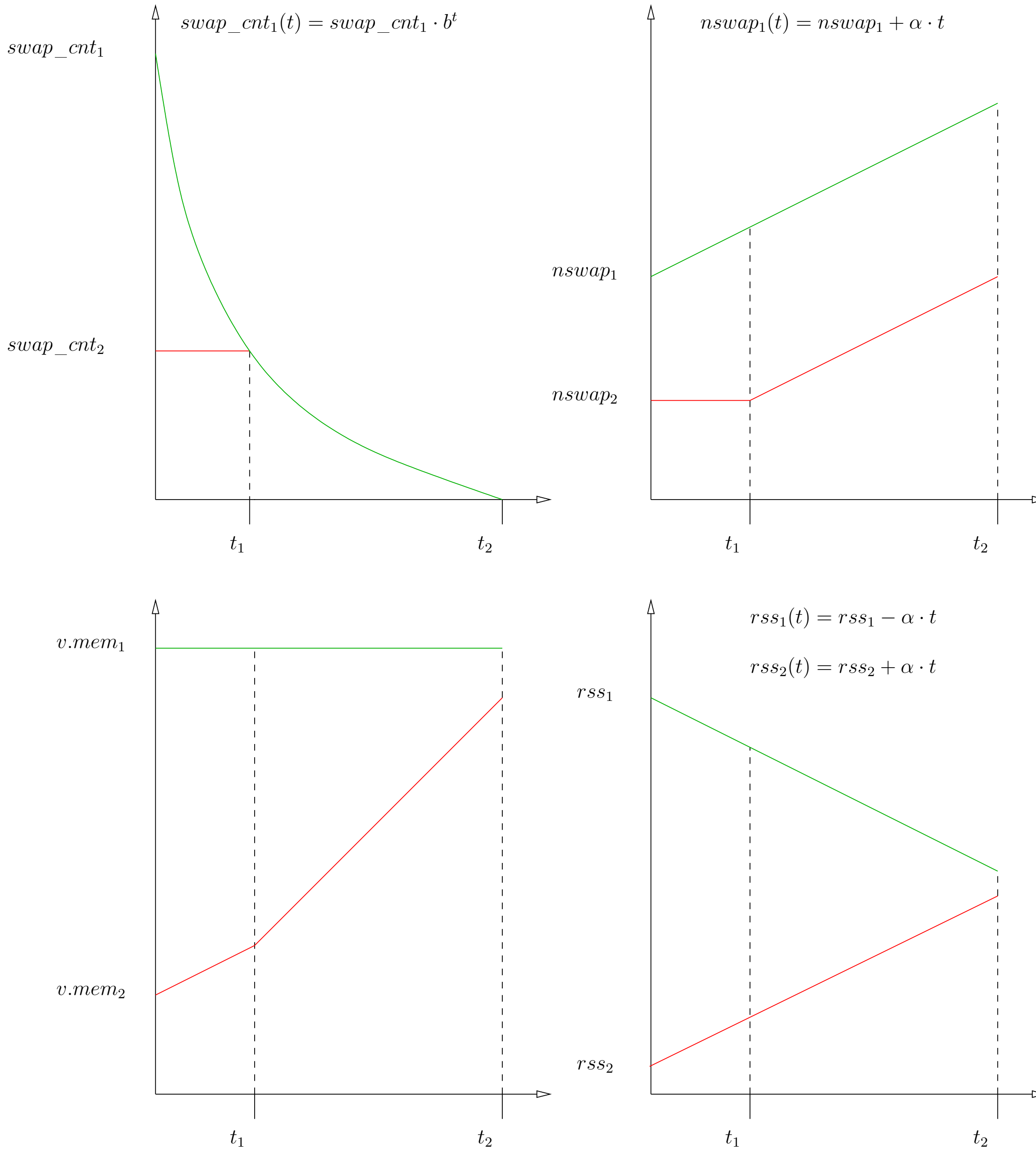
For større værdier vil der gå endnu flere cykler, inden der skiftes fra første instans af `greedy` til anden med hensyn til swapping. Vi forventer derfor, at afstanden mellem slutværdierne af `nswap` for de to processer vil blive større for voksende `darkos_vm_frac`.

At afstanden bliver større er lidt svært at se på graferne ovenfor. Derfor har vi plottet forholdet mellem de to værdier af `nswap` som funktion af `darkos_vm_frac` nedenfor. Man skal lige være bevidst om, at tallene kan variere for hver gang processerne udføres, så der er et par usikkerheder, der ikke er afbildet på grafen. Alligevel kan man se, at forholdet vokser som forventet.



De teoretiske værdier i ovenstående graf er fremkommet ved en simpel modellering af swapping-opførslen. Modellen gennemgåes i næste afsnit.

3.4 Simulering af swapping-strategi



Vores swapping algoritme gik ud på at trække en swap_frac'te del fra den proces' swap_cnt som lige havde swappet en side/sider ud. Da det altid er den proces med det højeste swap_cnt der skal swappe sider ud, vil i vores tilfælde process 1 swappe sider ud indtil dens swap_cnt er reduceret til en værdi som er ens eller mindre end den fra proces 2. Herefter vil begge processer skiftevis (strict alternating) swappe sider ud indtil deres swap_cnt hver især er reduceret til 0, hvor der tages et nyt snapshot. Dette anskueliggør grafen øverst til venstre.

Vi kan nu imidlertid beregne hvor langt tid(t_1) det tager, før første proces' swap_cnt er mindre eller lig den fra anden proces, samt hvor langt tid(t_2) der

yderligere går, før de er lig nul. Den første proces' swap_cnt vil til tiden t have værdien:

$$swap_cnt_1(t) = swap_cnt_1 \cdot \frac{swap_frac - 1}{swap_frac}^t$$

Størrelsen $\frac{swap_frac-1}{swap_frac}$ svarer til basen b på vores graf. For at beregne tiden t_1 må der altså gælde at $swap_cnt_1(t) = swap_cnt_2$, hvilket ved lidt manipulation giver at:

$$t_1 = log_{b^{-1}} \left(\frac{swap_cnt_1}{swap_cnt_2} \right)$$

Helt analogt er den anden tid:

$$t_2 = log_{b^{-1}} \left(swap_cnt_2 \right)$$

Hvis vi nu ser på hvad der sker i tidsrummet $t_c = t_1 + t_2$ anskueliggør de andre grafer hhv. hvordan processernes nswap, virtual memory og rss forholder sig.

Vi ved at proces 1 hele tiden, altså t_1 og t_2, swapper sider ud for at gøre plads til proces 2, mens proces 2 kun swapper sider ud i tidsrummet t_2. Vi har modeleret det således, at en proces swapper sider ud med en konstant hastighed α. Så proces 1. swapper i tiden t_c ud men hastighed α, mens proces 2. kun swapper ud i tidsperioden t_2. Således vil der i vores model gælde at:

$$nswap_1(t_c) = nswap_1 + \alpha \cdot (t_1 + t_2) \text{ og } nswap_2(t_c) = nswap_2 + \alpha \cdot t_2$$

Da vi nu har fået etableret et model af hvordan sider swappes ud, kan vi nu begynde at se på, hvorledes processernes virtuelle lagerforbrug ændrer sig med tiden. Grafen nederst til venstre anskueliggør dette, idet vi ved at proces 1 har allerede allokeret alt sit lager, mens proces 2 forsøge allokere mere lager, indtil den har fået allokeret det lager den har brug for. Igen opfatter vi at proces 2 vil prøve at allokere lager med en konstant hastighed β som sikker er meget større, omtrent størrelsorden XXX, end swappingshastigheden a, da accesstiden for fysik lager er mindre end for et drev. Imidlertid vil proces 2 aldrig komme op på at allokere lager med en hastighed der højere end den hvormed maskinen swapper ud, det vi sige en hastighed på α i t_1 og 2α til t_2. Vi får således:

$$v.mem_1(t_c) = v.mem_1 \text{ og } v.mem_2(t_c) = v.mem_2 + \alpha \cdot (t_1 + 2t_2)$$

Det sidste vi ser på er hvad der sker med processernes rss. Der er den trivielle sammenhæng at:

$$v.mem_1(t) = rss_1(t) + nswap_1(t) \text{ og } v.mem_1(t) = rss_1(t) + nswap_1(t)$$

som giver os den sidste graf og de dertil hørende formler:

$$rss_1(t_c) = rss_1 - \alpha \cdot t_1 + t_2 \text{ og } rss_2(t_c) = rss_2 + \alpha \cdot t_1 + t_2$$

Nu ved vi altså hvorledes de enkelte egenskaber forholder sig mens proces 2 prøver at allokere lagerplads. Videre ved vi at det hele går i stå, når proces 2 ikke længere vil allokere hukommelse, og dette sker netop når $v.mem_2$ har den værdi som vi har givet med som parameter til greedy programmet.

Alle disse informationer har vi brugt til at lave en simulering af, hvordan vores grafer kommer til at forløbe ved givne startværdier. Kildekoden ses i appendiks E

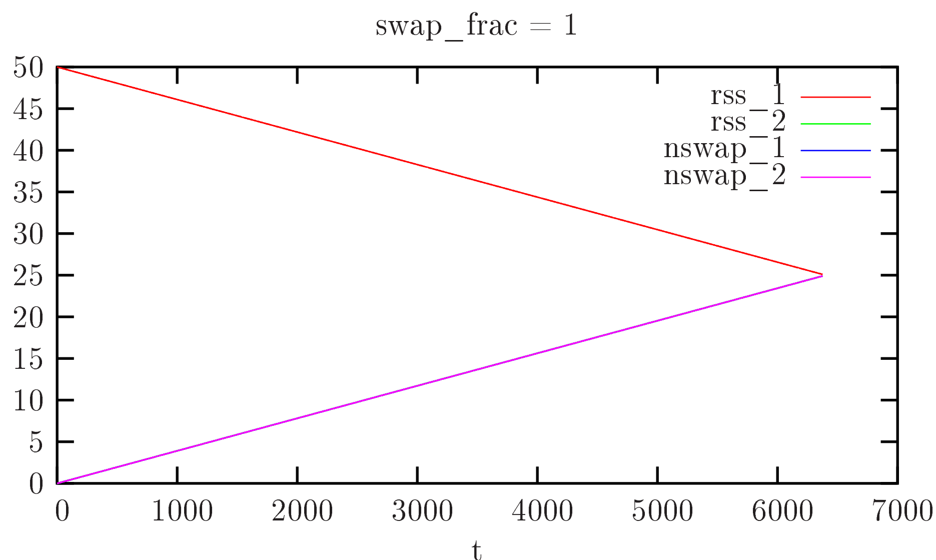
Programmet sim_swap tager 3 argumenter af typen unsigned int.

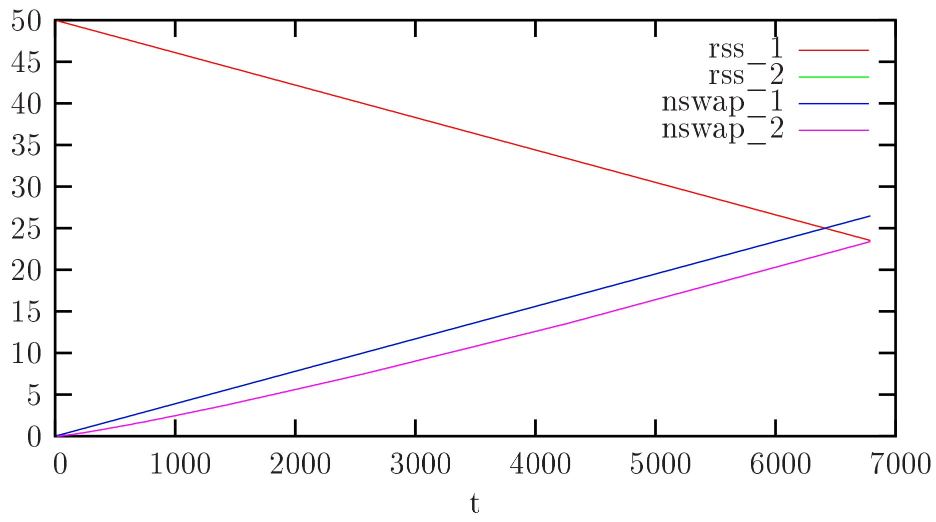
maxpages	antallet af sider hver proces 2 skal allokere hhv. proces 1 har allokeret
swap_rate	hastigheden hvormed processer kan swappe ud (svarer til α)
swap_frac	den del som en proces' swap_cnt skal reduceres med.

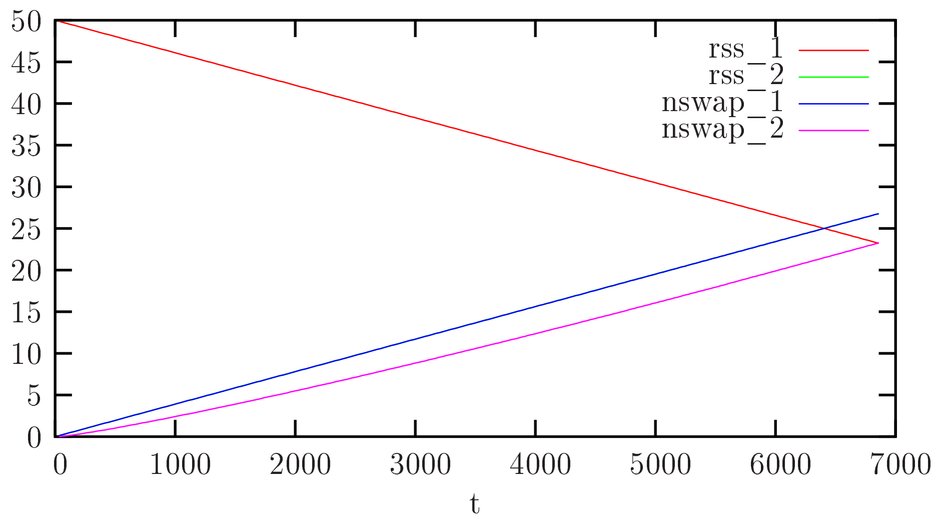
Programmets algoritme virker sådan, at værdierne rss_1, rss_2, $nswap_1$, $nswap_2$, t_1 og t_2 opdateres, vha. de udledede transistioner, for hver gang der er gået er tidsrum t_c. Når proces 2 har allokeret sit lager terminerer algoritmen.

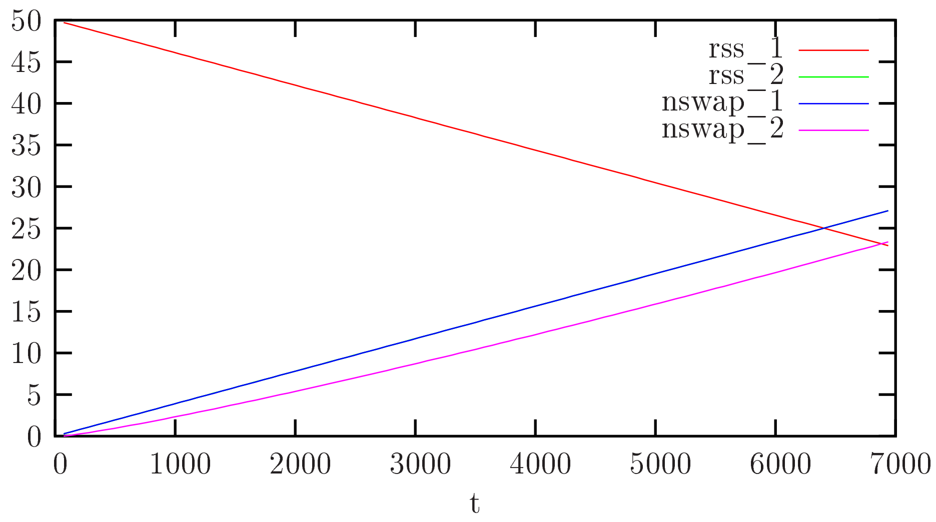
Programmet giver for hvert tidsinterval t_c output på rss_1, rss_2, $nswap_1$ og $nswap_2$.

Vi har kørt programmet med swap_frac $\in \{1, 2, 4, 8, 16\}$, og plottet data i gnuplot.

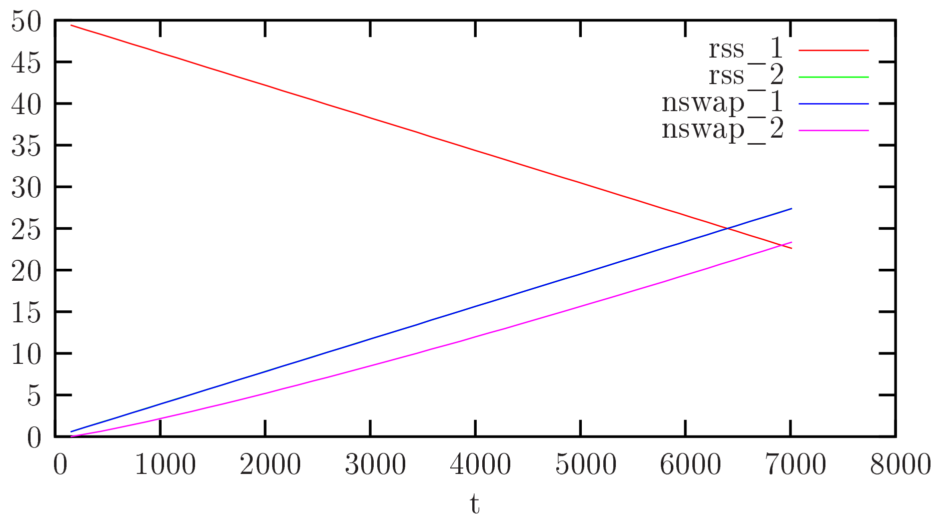



swap_frac = 2
rss_1
rss_2
nswap_1
nswap_2
50
45
40
35
30
25
20
15
10
5
0
0
1000
2000
3000
4000
5000
6000
7000
t


swap_frac = 4
rss_1
rss_2
nswap_1
nswap_2
50
45
40
35
30
25
20
15
10
5
0
0
1000
2000
3000
4000
5000
6000
7000
t



swap_frac = 8
50
45
40
35
30
25
20
15
10
5
0
0
1000
2000
3000
4000
5000
6000
7000
t
rss_1
rss_2
nswap_1
nswap_2



swap_frac = 16
50
45
40
35
30
25
20
15
10
5
0
0
1000
2000
3000
4000
5000
6000
7000
8000
t
rss_1
rss_2
nswap_1
nswap_2

4 Afslutning

Vi har i denne opgave set, hvorledes man kan undersøge og ændre i swapping-strategien i Linux-2.2.14. Vi har så gennemført en konkret ændring, der skulle medføre, at der vælges mere ligeligt mellem de processerne i stedet for kun at swappe en enkelt proces' sider ud. Efterfølgende undersøgelser viste, at swapping-opførslen derefter svarede til det, vi forventede.

A Patch til Linux kernen

En kopi af kildeteksten til Linux-kernen version 2.2.14 er blevet modificeret ved at tilføje systemkaldende `darkos_vm_set` og `darkos_vm_stat`. En patch mellem den modiferede kildetekst og den originale kildetekst blev lavet med følgende kommando:

```
# diff -Naur -X /root/dontdiff Linux-2.2.14 uge49 > patch-uge49
```

```
diff -Naur -X /root/dontdiff linux-2.2.14/arch/i386/kernel/entry.S
    uge49/arch/i386/kernel/entry.S
--- linux-2.2.14/arch/i386/kernel/entry.S    Mon Sep  9 18:03:12 2002
+++ uge49/arch/i386/kernel/entry.S   Fri Dec  5 07:24:26 2003
@@ -562,13 +562,14 @@
     .long SYMBOL_NAME(sys_ni_syscall)        /* streams1 */
     .long SYMBOL_NAME(sys_ni_syscall)        /* streams2 */
     .long SYMBOL_NAME(sys_vfork)             /* 190 */
-
+    .long SYMBOL_NAME(darkos_vm_stat)                /* 191 juleopg.
    */
+    .long SYMBOL_NAME(darkos_vm_set)                 /* 192 juleopg.
    */
     /*
      * NOTE!! This doesn't have to be exact - we just have
      * to make sure we have _enough_ of the "sys_ni_syscall"
      * entries. Don't panic if you notice that this hasn't
      * been shrunk every time we add a new system call.
      */
-    .rept NR_syscalls-190
+    .rept NR_syscalls-192
         .long SYMBOL_NAME(sys_ni_syscall)
     .endr
diff -Naur -X /root/dontdiff linux-2.2.14/include/asm-i386/unistd.h
    uge49/include/asm-i386/unistd.h
--- linux-2.2.14/include/asm-i386/unistd.h  Mon Sep  9 18:03:01 2002
+++ uge49/include/asm-i386/unistd.h Fri Dec  5 07:29:50 2003
@@ -195,8 +195,10 @@
 #define __NR_getpmsg        188 /* some people actually want streams
      */
 #define __NR_putpmsg        189 /* some people actually want streams
      */
 #define __NR_vfork      190
+#define __NR_darkos_vm_stat      191
 /* #define __NR_ugetrlimit      191 SuS compliant getrlimit */
-#define __NR_mmap2      192
+/* #define __NR_mmap2       192 */
+#define __NR_darkos_vm_set       192
 #define __NR_truncate64         193
 #define __NR_ftruncate64    194
 #define __NR_stat64         195
```

```
 diff -Naur -X /root/dontdiff linux-2.2.14/init/version.c uge49/init/
     version.c
--- linux-2.2.14/init/version.c Fri Nov 14 03:05:14 2003
+++ uge49/init/version.c     Mon Sep  9 18:04:27 2002
@@ -24,22 +24,3 @@
40   const char *linux_banner =
       "Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
       LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n";
-
-
45 -
-
-
-
-
50 -
-
-
-
-
55 -
-
-
-
-
60 -
-
 diff -Naur -X /root/dontdiff linux-2.2.14/kernel/sched.c uge49/
     kernel/sched.c
--- linux-2.2.14/kernel/sched.c Mon Sep  9 18:04:28 2002
+++ uge49/kernel/sched.c     Wed Dec 10 07:21:55 2003
65 @@ -2064,3 +2064,27 @@
       init_bh(TQUEUE_BH, tqueue_bh);
       init_bh(IMMEDIATE_BH, immediate_bh);
  }
+
70 +asmlinkage int darkos_vm_stat(pid_t pid, unsigned long *usrrss,
     unsigned long *usrnswap)
+{
+  unsigned long rss, nswap;
+  struct task_struct *task;
+  int err = 0;
75 +
+  task = find_task_by_pid(pid);
+
+  if(task == NULL) return -ESRCH; // pid doesn't exist
+
80 +  rss = task->mm->rss;
+  nswap = task->nswap;
+
+  // copying values to userspace
```

```
+  err = copy_to_user(usrrss, &rss, sizeof(rss));
85 +  err += copy_to_user(usrnswap, &nswap, sizeof(nswap));
+
+  // on error return 1
+  return (err != 0)? -EFAULT : 0;
+}
90 +
+
+
diff -Naur -X /root/dontdiff linux-2.2.14/mm/vmscan.c uge49/mm/
    vmscan.c
--- linux-2.2.14/mm/vmscan.c     Mon Sep  9 18:04:28 2002
95 +++ uge49/mm/vmscan.c    Wed Dec 10 07:22:39 2003
@@ -21,6 +21,8 @@

 #include <asm/pgtable.h>

100 +unsigned int darkos_swap_frac = 0;
+
 /*
  * The swap-out functions return 1 if they successfully
  * threw something out, and we got a free page. It returns
105 @@ -297,7 +299,8 @@
        for (;;) {
                int result = swap_out_vma(p, vma, address, gfp_mask);
                if (result)
-                       return result;
110 +                   return result;
+
                vma = vma->vm_next;
                if (!vma)
                        break;
115 @@ -370,8 +373,32 @@
                goto out;
        }

-       if (swap_out_process(pbest, gfp_mask))
120 -               return 1;
+       if (swap_out_process(pbest, gfp_mask)) {
+
+           /* DARKOS JULEOPGAVE
+            *
125 +            * rss for the process has been reduced.
+            * If swap_out_process has returned 1, a
+            * physical page has been freed, else
+            * the page was shared.
+            *
130 +            * We now want to reduce the probability
+            * that another page is taken from the same
+            * process the next time a page has to be
+            * swapped out.
```

```
+            *
135 +        * This is accomplished by reducing swap_cnt
+            * with a fraction k, because it is always the
+            * process with the highest swap_cnt that has
+            * to release a page.
+            */
140 +
+           if((darkos_swap_frac) && (p->mm->swap_cnt > 0))
+             p->mm->swap_cnt = p->mm->swap_cnt -
+                (p->mm->swap_cnt/darkos_swap_frac == 0
+                 ? p->mm->swap_cnt : p->mm->swap_cnt/darkos_swap_frac)
    ;
145 +       return 1;
+         }
      }
  out:
      return 0;
150 @@ -534,3 +561,7 @@
      return retval;
  }

+void darkos_vm_set(unsigned int k)
155 +{
+  darkos_swap_frac = k;
+}
```

B darkos_vm_stat.c

```
#include <stdlib.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

_syscall3(int, darkos_vm_stat, pid_t, pid, unsigned long *, rss,
    unsigned long *, nswap);

int main(int argc, char *argv[])
{
  unsigned long rss, nswap;
  pid_t pid;

  if (argc != 2)
    {
      printf("usage: darkos_vm_stat pid \n");
      return 1;
    }
  pid = atoi(argv[1]);

  if(darkos_vm_stat(pid, &rss, &nswap) == -1)
    {
      // Output kernel error message
      perror("darkos_vm_stat");
      exit(1);
    }

  printf("pid: %d, nswap: %d, rss: %d\n", pid, nswap, rss);

  return 0;
}
```

C darkos_vm_set.c

```c
#include <stdlib.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

_syscall1(int, darkos_vm_set, unsigned int, k)

int main(int argc, char *argv[])
{
  unsigned int k;

  if (argc != 2)
    {
      printf("usage: darkos_vm_set k \n");
      return 1;
    }
  k = atoi(argv[1]);

  darkos_vm_set(k);

  return 0;
}
```

D greedy.c

```
/*

 greedy.c

 Skal afvikles paa en Linux maskinen hvor kernen er udvidet med
 systemkaldet darkos_vm_stat beskrevet i opgaveformuleringen. Stub
    og
 kald af darkos_vm_stat skal tilfoejes inden programmet kan bruges.

*/

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

/* system call stub for darkos_vm_stat */
_syscall3(int, darkos_vm_stat, pid_t, pid,unsigned long *, rss,
    unsigned long *,nswap);

#define SAMPLEINT 10000
struct itimerval samplet = {{0,SAMPLEINT},{0,SAMPLEINT}};

int cursample = 0;
int totaltime = 0;
long elapsedtime = 0;

struct timeval tvstart;

unsigned long report(pid_t pid)
{
  unsigned long nswap = 0;
  unsigned long rss = 0;
  long msecs = 0;
  struct timeval tv;

  /* Da pid er gyldig, og det er meget usandsynligt, at
   * der sker en anden fejl, antages følgende kald
   * altid at gå godt.
   */
  darkos_vm_stat(pid, &rss, &nswap);

  gettimeofday(&tv,NULL);
```

```
  msecs = ((tv.tv_sec - tvstart.tv_sec) * 1000000 + (tv.tv_usec -
     tvstart.tv_usec)) / 1000;

  printf("%ld %d %lu %lu # time sample nswap rss \n", msecs,
     cursample, nswap, rss);
50
  return msecs;

}

55 static void sample_sighandler(int signo)
{
  long msecs;
  cursample++;

60  msecs = report(getpid());

  if (msecs >= (totaltime * 1000))
    exit(1);

65  elapsedtime = msecs;

}

void greedy(int maxalloc)
70 {

  int i,j;
  char *p;

75  for (j = 0; j < maxalloc; j++) {
    p = (char *)malloc(1024*1024);
    for (i = 0; i < 1024*1024; i++)
      p[i] = 42;
  }
80 }

int main(int argc, char *argv[])
{
  int maxmb;
85  long starttime;

  if (argc != 4)
    {
      printf("Usage: greedy mbmax totaltime starttime");
90      exit(0);
    }

  maxmb = atoi(argv[1]);
  totaltime = atoi(argv[2]);
95  starttime = atol(argv[3]) * 1000;
```

```
    gettimeofday(&tvstart ,NULL);

    setitimer(ITIMER_REAL,&samplet ,NULL);
100 signal(SIGALRM, sample_sighandler);

    while ((starttime - elapsedtime) > 0);

    greedy(maxmb);
105
    while(1);
}
```

E sim_swap.c

```
#include <stdio.h>
#include <math.h>

int main(int argc,char *argv[]) {

  if(argc!=4) {
    printf("usage: sim_swap maxpages swap_rate swap_frac\n");
    exit(1);
      }

  unsigned int maxpages, swap_frac, rss_1, rss_2, nswap_1, nswap_2,
      swap_1,swap_2,t_1,t_2,k,swap_rate, cycle;
  maxpages=atoi(argv[1]);
  swap_rate=atoi(argv[2]);
  swap_frac=atoi(argv[3]);
  rss_1=maxpages;
  rss_2=1;
  nswap_1=nswap_2=0;
  cycle=0;
  if(!swap_frac || !maxpages || !swap_rate) {
    printf("not legal startvalues\n");
    exit(1);
  }

  printf("# cycles\trss_1\trss_2\tnswap_1\tnswap_2\n"); //
      generating header output for gnuplotdatafile

  while(nswap_2+rss_2<maxpages) {
    swap_1=rss_1; swap_2=rss_2; // taking snapshot

    t_1=(int)((log(swap_1)-log(swap_2))/(log(swap_frac)-log(
        swap_frac-1))); // recalculating timeintervals
    t_2=(int)((log(swap_2))/(log(swap_frac)-log(swap_frac-1)));

    if(t_2==0) t_2=1; // making sure we have at least 1 operation

    rss_1=rss_1-swap_rate*(t_1+t_2); // updating values
    rss_2=rss_2+swap_rate*(t_1+t_2);
    nswap_1=nswap_1+swap_rate*(t_1+t_2);
    nswap_2=nswap_2+swap_rate*(t_2);

    cycle+=t_1+t_2; // updating cycles

    printf("%d\t%d\t%d\t%d\t%d\n",cycle,rss_1,rss_2,nswap_1,nswap_2)
        ; // generating output for gnuplotdatafile
  }
  printf("# rss_1: %d\tnswap_1: %d\t1.total pages: %d\n rss_2: %d\
     tnswap_2: %d\t2.total pages: %d\n"
     ,rss_1,nswap_1,rss_1+nswap_1,rss_2,nswap_2,rss_2+nswap_2);
```

```
45    return 0;
}
```