Deluppgift 9 Installera Pintos

Uppgift

Målet med denna del är att skapa din egen klon av Pintos som du kan jobba med genom kursen, och se till att den verkar fungera. Du måste kopiera filer, lära dig hur du kompilerar, hur du startat program som kör i Pintos, och hur du felsöker. Den här delen har inget att redovisa, men desto mer som du kan behöva gå tillbaka till senare. Det är okej om du väntar med vissa delar tills du behöver dem, men kom då ihåg att gå tillbaka hit och försök själv innan du frågar.

Snabbstart för installation på ditt IDA-konto (Linux Mint)

Utför följande kommandon:

```
module add /home/TDIU16/lab/modules/pintos
pintos
```

Om du nu får en hjälputskrift som förklarar kommandot `pintos` så går du vidare:

```
module initadd /home/TDIU16/lab/modules/pintos
mkdtemp
setfacl -m g:TDIU16:r-x TDIU16*
cd ~/TDIU16*
git clone https://gitlab.ida.liu.se/klaar36/pintos.git
cd pintos/src/userprog
make -C ../examples
make
pintos -p ../examples/sumargv -a sumargv   \
       -v -k --fs-disk=2 -- -f -q run sumargv
```

Om du tillhör den skaran användare som är lämpad att bli systemadministratör har du redan innan du skrev in kommandona kontrollerat vad varje kommando egentligen gör och om det är lämpligt att exekvera. Dessutom fungerar din installation utan annat än väntade fel (se kommande exempel).

Tillhör du den mer normala skaran användare så har du bara kopierat och klistar in kommandona, och har nu en ofungerande installation av Pintos, samt ingen aning om vad som gick snett. Tack och lov finns då metoden “gör om gör rätt” som jag brukar tillämpa. Börja med att ta bort mappen vars namn börjar med TDIU16 från ditt konto. Ta reda på vad den heter:

```
ls -1d ~/TDIU16*
```

Utöva sedan **EXTREM** försiktighet med följande kommando:

```
\rm -rf vad_nu_mappen_vars_namn_borjar_med_TDIU16_heter
```

Skämtsamt kan man säga att ovan gör “remove real fast”, och går det snett så är det kört. Den som en gång provat slarva med detta vet vad jag menar. Den som inte provat slarva ännu rekommenderas göra det innan du har alltför viktiga filer som försvinner, eller helst låta bli helt. När du nu rensat kan du börja om. Var mer noggran med kommandona och mer observant på vad som eventuellt går galet.

Alternativ git-URL

Om du har loggat in på `gitlab.ida.liu.se` och satt upp nycklar för `ssh` kan du använda `ssh` i stället för `https`:

```
git clone git@gitlab.ida.liu.se:klaar36/pintos.git
```

Gitlab är en bra resurs att använda för både kursers laborationer och egna projekt, så jag rekommenderar du sätter dig in i hur `git` används och fungerar och sedan använder det.

Installera Pintos på egen dator (Linux)

Detta är inget vi supportar eftersom det finns så många varianter på datorer och distributioner. Är du händig och har lite tur kan du dock göra ett försök. Börja med följande kommandon. Nu är det speciellt viktigt att du okritiskt kopierar och kör kommandona rakt av, annars kanske du hittar de bakdörrar och trojaner som installeras på din dator. Okej, jag skojar. Det finns inget sådant vad jag vet. Dock kanske det finns anledning att inte köra kommandona hur som helst. Filer kan råka kopieras över, eller ändras på fel sätt även om jag tror de fungerar. Du kan alltså behöva anpassa till hur det ser ut på just din dator. Här är i alla fall kommandona:

```
sudo aptitude install qemu
sudo aptitude install git
mkdir -p ~/bin
PARLOMBA=parlomba1.ida.liu.se
scp -r <LiU-ID>@$PARLOMBA:/home/TDIU16/lab/bin ~/bin/pintos
echo "export PATH=$PATH:$HOME/bin/pintos" >> ~/.bashrc
mkdir ~/TDIU16
cd ~/TDIU16
```

Så! Nu är det **INTE** klart, utan nu bör du kunna fortsätta med installationen från och med kommandot `git clone` i snabbstarten.

Installera Pintos på egen dator (Ej Linux)

Installera först `VirtualBox`, sedan en virtuell maskin (i `VirtualBox`) med `Linux Mint`. Därpå installerar du enligt instruktionerna för `Linux` ovan.

Bläddra i Pintos källkod bekvämt med emacs

Den version av Pintos som laborationerna utgår från finns tillgänglig i git-repositoryt och i kurskatalogen:

```
/home/TDIU16/lab/skel/pintos/src/
```

Denna version är read-only. Du har alltså alltid ursprungsversionen tillgänglig att jämföra med även efter att du tagit bort eller ersatt kod i din version. ***Var alltså inte rädd att ta bort given kod eller kommentarer som "är i vägen" i Pintos.*** Vill du senare se vad det stod är det bara att "diffa" mot ursprungsfilen i kurskatalogen. Är du lite mer modern diffar du mot git-repositoryt:

```
git help diff
```

För att enkelt kunna använda emacs och bläddra i koden finns det i `src` katalogen en fil `TAGS` som berättar för emacs var olika deklarationer och definitioner finns i koden. Det innebär att du genom att placera markören vid ett funktionsanrop i koden och trycka M-. (Meta-punkt) i emacs kan hoppa direkt till definitionen av motsvarande funktion. M-* hoppar tillbaka. Första gången du använder detta måste du hjälpa emacs att lokalisera TAGS-filen. TAGS-filen skapas eller uppdateras från din `src` katalog:

```
cd $HOME/TDIU16*/pintos/src
make TAGS
```

Kompilera och kör Pintos

Följande kommandon utgår från att du följde ovan instruktioner till punkt och pricka. Om du inte gjorde det kan du behöva justera kommandon eller sökvägar nedan.

I resten av instruktionen kommer alla sökvägar att anges relativt `src`-katalogen i Pintos. Det testprogram du skall använda i exemplen nedan finns i `examples/sumargv.c`. Lokalisera programmets källkod och kontrollera varför det (ibland) avslutar med kod `111`.

Använd sedan följande kommandon för att kompilera Pintos och testköra programmet. Observera att sista kommandot är radbrutet här, men skall skrivas på en rad i terminalen. Det omvända snedtecknet sist på raden anger att raden fortsätter på nästa rad och skall inte tas med när du manuellt skriver av kommandot på en rad, men gör att det ibland fungerar om du klipper och klistrar.

```
cd $HOME/TDIU16*/pintos/src/userprog
make -j8 -C ../examples
make -j8
pintos -p ../examples/sumargv -a sumargv   \
       -v -k --fs-disk=2 -- -f -q run sumargv
```

När du kör Pintos enligt ovan kommer mycket status-information att skrivas ut. Var alltid mycket uppmärksam på eventuella felmeddelanden. Nedan är denna information uppbruten i delar med några korta kommentarer (inklusive hur du stoppar programkörningen med `Ctrl-a x`!).

Följande text beskriver status för uppsättningen av emulator och initial disk:

```
Copying ../examples/sumargv into /tmp/MAdPEpgFX5.dsk...
Writing command line to /tmp/Jh32hYvaz0.dsk...
qemu -hda /tmp/Jh32hYvaz0.dsk -hdb /tmp/InW2t5E1LN.dsk  \
     -hdc /tmp/MAdPEpgFX5.dsk                           \
     -p 1234 -m 4 -net none -monitor null -nographic
```

Därefter kommer Pintos boot-meddelanden:

```
Kernel command line: -f -q put sumargv run sumargv
Pintos booting with 4,096 kB RAM...
375 pages available in kernel pool.
374 pages available in user pool.
# main#1: thread_create("idle", ...) RETURNS 2
Calibrating timer...  16,460,800 loops/s.
hd0:0: detected 129 sector (64 kB) disk, \
       model "QEMU HARDDISK", serial "QM00001"
hd0:1: detected 4,032 sector (1 MB) disk, \
       model "QEMU HARDDISK", serial "QM00002"
hd1:0: detected 81 sector (40 kB) disk, \
       model "QEMU HARDDISK", serial "QM00003"
```

Rader som startar med tecknet # är ett debug-meddelande för att lättare kunna följa exekveringen av några centrala och viktiga funktioner.

Filsystemet formateras när flaggan -f anges, och filer kopieras in då flaggor -p och -a anges:

```
Formatting file system...done.
Boot complete.
Putting 'sumargv' into the file system...
```

Så följer starten av den första processen (run sumargv). Eftersom Pintos inte är fullt funktionellt ännu kommer inte så mycket att utföras:

```
Executing 'sumargv':
# main#1: process_execute("sumargv") ENTERED
# main#1: thread_create("sumargv", ...) RETURNS 3
ERROR: Main about to poweroff with 2 threads still running!
ERROR: Check your process_execute() and process_wait().
# sumargv#3: start_process("sumargv") ENTERED
# sumargv#3: start_process(...): load returned 1
# sumargv#3: start_process("sumargv") DONE
Executed an unknown system call!
Stack top + 0: 1
Stack top + 1: 111
# sumargv#3: process_cleanup() ENTERED
sumargv: exit(-1)
# sumargv#3: process_cleanup() DONE with status -1
```

Rader som inleds med # är debugutskrifter.

Nu kommer exekveringen att skriva ut några fel men i övrigt fungera. Felutskrifterna beror på att varken funktionen `process_execute` eller funktionen `process_wait` är korrekt implementerad. Det kommer du att göra i en senare uppgift. Nuvarande funktioner har bara grundfunktionalitet för att du skall kunna komma igång med systemanrops-implementationen.

Funktionen `process_execute` stänger av datorn istället för att returnera den nya processens id, och funktionen `process_wait` är bara implementerad som en stub som returnerar minus ett direkt. Funktionen `process_wait` anropas för att Pintos skall vänta tills `sumargv` (som är den första processen i detta fall) blir klar. När det inträffar avslutar Pintos. Om `process_wait` returnerar för tidigt så kommer operativsystemet avsluta, och kanske stänga av datorn (om flagga `-q` angavs) medan jobb fortfarande finns kvar att utföra. I nuläget kommer inte exekveringen så långt eftersom `process_execute` hinner stänga av datorn med `power_off` först.

Avstängningskoden har i vår version av Pintos felhanteringskod tillagd som gör att operativsystemet skriver ut ett fel men ändå väntar tills alla trådar är klara. Detta är praktiskt att ha medan du arbetar med Pintos, och är det enda som gör att det alls går att starta en process innan `process_execute` och `process_wait` korrigerats av dig. *Kom ihåg:* I vissa lägen kommer Pintos ändå att "låsa sig" utan att stänga av. Tryck då `Ctrl-a` och sedan `x` för att avsluta emulatorn QEMU. Detta bekräftas med följande meddelande:

```
QEMU: Terminated
```

Fungerar inte det brukar ett bra trick vara att öppna en ny terminal och i den köra:

```
killall pintos
```

Programmet `sumargv` *du körde på förra sidan skriver ut de två översta värdena på user-stacken (fetstilat). Kan du lista ut var det andra värdet kommer från? Kan du lista ut vad det första är? Studera koden för* `sumargv` *och fundera på vilket systemanrop som utförs då main returnerar. Med kännedom om hur systemanrop går till och anropas (lib/user/syscall.*) och numreras (lib/syscall-nr.h), stackens utseende från uppgift 8, och programmets kod (examples/sumargv.c) bör du kunna klura ut det.*

När Pintos sedan avslutar skrivs lite statistik ut:

```
Timer: 54 ticks
Thread: 0 idle ticks, 52 kernel ticks, 2 user ticks
hd0:0: 0 reads, 0 writes
hd0:1: 53 reads, 170 writes
hd1:0: 81 reads, 0 writes
Console: 1302 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
```

Om flagga -q angavs då Pintos startade kommer operativsystemet även stänga av emulatorn (datorn). Om trådar fortfarande kör kommer debug-kod att generera en felutskrift och vänta på att dessa avslutar. När emulatorn stängs av ser du följande meddelande:

```
Powering off...
```

Vi har nu gått igenom en hel programkörning med Pintos, från start av operativsystemet (boot-sekvens), via exekvering av ett program, till hur det (så småningom) skall se ut när Pintos avslutar. Kommandoraden du angav för att starta Pintos innehöll många flaggor och argument. I nästa avsnitt där felsökning introduceras används ett alternativt sätt att skriva kommandoraden, med minimalt antal flaggor. För att få mer information om vilka flaggor som kan användas och hur de fungerar kan du skriva:

```
pintos --help
```

Kort om x86 emulatorn QEMU

Det är även användbart att känna till lite om emulatorn `qemu` som är den emulator vi använder. Här följer några tangentbordskombinationer som kan vara användbara. Ytterligare information kan den intresserade hitta på `http://www.qemu.org/`.

```
<Ctrl-Alt> i det grafiska fönstret tar eller släpper kontroll
           över tangentbord och mus.
<Ctrl-a x> Avslutar emulatorn. Släpp control innan du trycker
           på x. Kan behöva tryckas flitigt för att ge effekt.
```

Felsökning med debugger

Att använda debugger kräver i princip samma kommandon och flaggor som att köra utan, samt en *extra terminal för debuggern*. Dock passar vi här på att gå igenom ett alternativt sätt att skapa Pintos disk, och skippar några flaggor vi kan klara oss utan. Detta gör kommandoraden lite kortare, men det blir ibland lite svårare att nyttja samma kommandorad igen genom att bara trycka upp-pil i terminalen. Kör följande kommandon:

```
cd $HOME/pintos/src/userprog
debugpintos --fs-disk=2 -p ../examples/sumargv    \
            -a sumargv -- -f run sumargv
```

Det som är nytt är att vi använder `debugpintos` istället för `pintos`, för att debuggern senare skall kunna ansluta. Pintos kommer nu vänta på att en debugger skall ansluta. ***I en annan terminal*** *startar du debuggern:*

```
cd $HOME/pintos/src/userprog
pintos-gdb build/kernel.o
```

Du får upp en hel del text när debuggern startar, följt av en prompt där du skriver `debugpintos`. Det du skall skriva är i kursiv stil nedan.

```
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/
licenses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from build/kernel.o...done.

(gdb) debugpintos
0x0000fff0 in ?? ()
```

Debuggern skriver ut lite frågetecken. Det är OK, då har den anslutit till emulatorn. Skriv `break main` för att skapa en initial brytpunkt:

```
(gdb) break main
Breakpoint 1 at 0xc0100008: file ../../threads/init.c, line 68.
```

Debuggern bekräftar att den är med på noterna. Fortsätt programkörningen med `continue`:

```
(gdb) continue
Continuing.
Breakpoint 1, main () at ../../threads/init.c:72
72      {
```

Pintos kommer nu köras fram till brytpunkten vid `main`. Nu kan du sätta valfria brytpunker eller använda valfria kommandon i debuggern för att utföra felsökning. I detta exempel nöjer vi oss med att skapa ytterligare en brytpunkt vid `process_execute`, och sedan fortsätta (`continue`) tills Pintos kommer dit.

```
(gdb) break process_execute
Breakpoint 2 at 0xc0108367: file ../../userprog/process.c, line
147.
(gdb) continue
Continuing.
Breakpoint 2, process_execute (command_line=0xc0007d91
"sumargv") at ../../userprog/process.c:147
147       int command_line_size = strlen(command_line) + 1;
```

Använd sedan kommandot `next` för att stega en rad i programmet:

```
(gdb) next
166       debug("%s#%d: process_execute('%s') ENTERED\n",
```

Nästa rad att exekvera skrivs ut. Om du inte skriver något kommando i debuggern, utan bara trycker `Enter` så kommer föregående kommando att upprepas. Prova:

```
(gdb)
179       arguments.command_line = malloc(command_line_size);
(gdb)
180       strlcpy(arguments.command_line, command_line,
command_line_size);
(gdb)
200       strlcpy_first_word (debug_name, command_line, 64);
(gdb)
204       thread_id = thread_create (debug_name, PRI_DEFAULT,
(gdb)
256       power_off();
```

Prova nu kommandot `backtrace`. Det ger information om hur programstacken ser ut, vilka funktioner som ledde till raden som exekveras:

```
(gdb) backtrace
#0  process_execute (command_line=0xc0007d91 "sumargv")
    at ../../userprog/process.c:256
#1  0xc0100559 in run_task (argv=0xc010f44c)
    at ../../threads/init.c:278
#2  0xc01005fd in run_actions (argv=0xc010f444)
    at ../../threads/init.c:330
#3  0xc01000bd in main ()
    at ../../threads/init.c:126
```

Stega vidare så att även `power_off` exekveras. Pintos avslutas. Sedan avslutar du debuggern:

```
(gdb) next
Watchdog has expired.  Target detached.
(gdb) quit
```

Du kan själv undersöka vilka andra felsökningsmöjligheter som finns. Här följer några av de mest vanliga kommandon du kan använda i debuggern. Kommandot `backtrace` är kanske det mest användbara. Mer ovanliga kommandon inkluderar kommandon för att skriva ut minnesinnehåll eller köra disassembler på funktioner. Sådant är kanske mest användbart för den som felsöker en kompilator eller assembler.

```
help
help bt
help next
backtrace
bt
next
nexti
step
stepi
break
clear
delete
display
undisplay
```