Deluppgift 9 Installera PintOS

Uppgift

Läs *först* följande avsnitt och utför sedan de kommandon som krävs för att du skall installera PintOS och bekanta dig med hur det används och ser ut.

Lägga till de verktyg som behövs för PintOS

För att få tillgång till diverse script och verktyg som behövs för att kompilera, starta och hantera PintOS behöver du installera modulen `/home/TDDI81/lab/modules/pintos`. Så här långt kommen i din utbildning bör du vara van att hantera modulsystemet, men för säkerhets skull kommer det här igen. För att lägga till modulen, använd kommandot:

```
module add /home/TDDI81/lab/modules/pintos
```

För att lättare hantera olika kodversioner och kunna spara arbetet efter varje delsteg behöver du subversion:

```
module add prog/subversion
```

Observera att dessa module-kommandon måste göras i *varje* terminal (kommandoskal) där du vill ha åtkomst till PintOS. För att få systemet att automatiskt lägga till modulen i varje terminal (endast efter återinloggning) använder du som vanligt `initadd` istället för `add`. Observera att om du gör `initadd` på en felaktigt specificerad modul är det stor risk att du saboterar filen `.login` i din hemkatalog. Kontrollera därför alltid att `add` fungerar som förväntat innan du gör `initadd`:

```
module initadd /home/TDDI81/lab/modules/pintos
module initadd prog/subversion
```

Från och med *nästa* gång du loggar in kommer du nu ha automatisk tillgång till ovanstående moduler i alla terminaler.

Kopiera källkoden utan versionshantering (enkelt)

```
gcp -a /home/TDDI81/lab/skel/ ~
chmod -Rf o+rX ~/skel
chmod 711 ~/skel
mv ~/skel ~/passphrase
```

Passphrase ovan är ett lösenord du hittar på. Om du väljer denna installationsvariant kan du nu hoppa direkt till “Gör koden tillgänglig för kurspersonal och labpartner”.

Kopiera källkoden med svn versionshanterng (om du vill vara lite mer avancerad)

Kör först följande rader. För att automatisera dem för senare inloggningar kan du lägga in dem i filen `.cshrc.private` i din hemkatalog:

```
setenv PINTOSURI file://$HOME/svnpintos
setenv PINTOSSKEL /home/TDDI81/lab/skel/pintos
alias make gmake
setenv PAGER less
```

Detta är i korthet de kommandon som behövs för att skapa ett eget svn-repository med PintOS:

```
svnadmin create $HOME/svnpintos
svn import $PINTOSSKEL $PINTOSURI -m "original pintos source"
chmod go-rwx $HOME/svnpintos
```

För att sedan skapa en lokal arbetskopia av PintOS gör du följande:

```
svn checkout $PINTOSURI $HOME/pintos
set-svn-ignore $HOME/pintos
chmod go-rwx $HOME/pintos
```

Arkivera ändringar och klara uppgifter

Nu skall du har en `pintos`-katalog i din hemfolder där du skall göra alla ändringar. Du har även en `svnpintos`-katalog (även kallat svn-repository) som du inte skall röra. När du är färdig med en uppgift arkiverar du den i repositoryt med kommandon:

```
cd $HOME/pintos
svn -m "uppgift 09 klar" commit
```

Ytterligare subversion-kommandon

Följande kommandon utgår från att du står i arbetskatalogen `$HOME/pintos`.

För att se vilken revision du är på eller vilka revisioner som finns, prova:

```
svn up
svn log
```

För att lista alla ändringar efter revisionen REV, prova (byt REV mot t.ex. 1 för revision 1):

```
svn diff -x -wup -r REV
```

För att lära dig mer om olika svn-kommandon, prova:

```
svn help [kommando]
```

Ytterligare information och tutorials finns tillgängliga på Internet.

Gör koden tillgänglig för kurspersonal och labpartner (passphrase)

Om du valt den enkla installationsvarianten kan alla som känner till din passphrase komma åt din kod. Be din labpartner verifiera detta genom att logga in på sitt konto och testa:

```
ls -la ~jambo007
ls -la ~jambo007/passphrase/pintos/src/userprog
```

Det första kommandot bör nu misslyckas med ett meddelande liknande detta (`jambo007` skall naturligtvis ersättas med ditt användarnamn):

```
ls: /home/jambo007: Du har inte rätt behörighet
```

Det andra kommandot bör däremot lista filerna i `userprog`-katalogen. Filerna går inte att modifiera på plats, men kan lätt kopieras. Du kan nu ge din passphrase till din labpartner eller kurspersonal för att få hjälp vid speciella problem.

Gör koden tillgänglig för kurspersonal och labpartner (access lists)

Antag att din kopia av PintOS finns i katalogen `$HOME/pintos`. Du kan ge tillträde till valda personer via så kallade access lists.

```
chmod -R o+rX $HOME/pintos
setfacl -s u::rwx,g::---,o:--- $HOME/pintos
setfacl -r -m g:TDDI81:r-x $HOME/pintos
setfacl -r -m u:jambo007:r-x $HOME/pintos
```

Det första kommandot ger alla access till alla (PintOS) filer. Det andra gör att endast du kan komma förbi den första katalogen i trädet. Den tredje gör att även kurspersonal kommer åt katalogen, och den sista raden (som kan upprepas med olika användare) gör att den namngivna användaren får tillgång. I vanliga fall är det bara din labpartner. Ovan är endast läsrättighet satt, men det går naturligtvis att ge skrivrättighet också. Det är dock på egen risk.

Gör ditt konto tillgänglig för din labpartner

Det är möjligt (t.o.m. enkelt) att ställa in ett konto så att specificerade användare kan logga in utan lösenord på kontot om de redan loggat in med sitt eget lösenord. Detta kan ibland vara praktiskt när flera personer arbetar på ett projekt. Man bör då se till att använda svn och checka ut koden till var sin arbetskatalog för att undvika problem som uppstår när samma filer ändras samtidigt av flera personer. Vill du dela ditt konto med din labpartner, fråga examinator hur du skall göra. Tänk på att din labpartner då får tillgång till ALLT på kontot, och kan agera gentemot andra användare, platser och skrivare som vore han/hon du. Det är antagligen bättre att använda någon av ovan metoder.

Bläddra i PintOS källkod bekvämt med emacs

Den version av PintOS som laborationerna utgår från finns tillgänglig i kurs-katalogen:

```
/home/TDDI81/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.

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.

Om du installerat enligt svn:

```
cd $HOME/pintos/src
gmake TAGS
```

Om du installerat enligt den enklare varianten:

```
cd $HOME/passphrase/pintos/src
gmake TAGS
```

Kompilera och kör PintOS

Följande kommandon utgår från att du följde ovan svn-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/pintos/src
gmake -j8 -C examples
gmake -j8 -C userprog
cd userprog/build
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 systemanropsimplementationen.

Funktionen `process_execute` stänger av datorn istället för att eturnera 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
```

Programmet ovan skiver ut de två översta värdena på user-stacken (stack top). 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/build
pintos-mkdisk fs.dsk 2
debugpintos -p ../../examples/sumargv    \
            -a sumargv -- -f run sumargv
```

Det första som är nytt är att disken som innehåller PintOS först skapas med ett eget kommando, `pintos-mkdisk`. Denna disk stannar sedan kvar tills du tar bort den (tar relativt stor plats på kontot), och kan användas till många körningar av PintOS. Det andra 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/build
pintos-gdb 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 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "--host=sparc-sun-solaris2.10 --
target=i386-elf"...

(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_wait`, 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
```