Institutionen för Elektronik och
Datorsystem vid ICT skolan

Laboration 2

Small-Shell v2.51 för UNIX

ID2206/ID2200 (fd. 2G1520/2G1504)
Operativsystem

ht 2007 - vt 2008

Institutionen för Elektronik och Datorsystem

miniShell v2.5 för UNIX

ID2206/ID2200 (fd. 2G1520/2G1504) Operativsystem

1.0 Introduktion

Den här laborationen har som syfte att ge kunskap om och förståelse för hur man skapar och använder processer i operativsystemet UNIX. Du skall också lära dig hitta, läsa och tillägna dig teknisk dokumentation i form av `man`-blad. Eftersom många andra operativsystem lånat många ideer från UNIX är de kunskaper du tillägnar dig i den här laborationen och de efterföljande användbara inom ett ganska brett område.

> **Förberedelser**: Innan du sätter igång med labben ska Du gå igenom exemplen nedan, samt lösa de ev. uppgifter/frågor som ges i anslutning till exemplen. ***Observera*** att Du måste utnyttja möjligheten att studera manualblad med hjälp av man-kommandot.

1.1 Uppgiften

Uppgiften går ut på att skriva ett program som fungerar som en enkel kommandotolk (shell) för UNIX. Programmet skall tillåta användaren att mata in kommandon till dess han/hon väljer att avsluta kommandotolken med kommandot "**`exit`**". Kommandotolken skall tillåta att kommandon exekveras i förgrunden eller som bakgrundsprocesser. Kommandotolken skall skriva ut information, med processid och om kommandot exekveras i förgrund/bakgrund, när processer skapas respektive terminerar. Efter varje kommando som körts i förgrunden skall ditt program skriva ut statistik för hur lång tid kommandot tog att genomföra (dvs. du behöver inte mäta exekveringstider för bakgrundsprocesser).

De flesta kommandon skall exekveras i en separat process som ditt program, eller miniShell, skapar för varje kommando. Emellertid skall ditt program ha två *inbyggda* kommandon:

- **`cd`** *`dir`* Som byter arbetsdirectory (working directory) till *`dir`*. Om *`dir`* inte finns eller är en icke giltig sökväg så skall ditt program upptäcka det och byta arbetsdirectory till användarens hemdirectory som det är definierat av environmentvariabeln **`HOME`**.
- **`exit`** för att avsluta ditt program.

Ditt program måste ha inbyggda felkontroller för returvärden från systemanropen. Om du upptäcker ett fel skall vettiga felmeddelanden skrivas ut till användaren.

Kommandotolken får naturligtvis inte lämna sk. zombie-processer efter sig! Den skall heller inte termineras om en förgrunds eller bakgrundsprocess termineras med <Ctrl-c>.

För förgrundsprocesser skall information med processid skrivas ut utan fördröjning både när de skapas respektive terminerar. För bakgrundsprocesser skall information om skapande skrivas ut omedelbart och för terminering senast i samband med att nästa kommando matas in i kommandotolken efter det att bakgrundsprocessen terminerats.

Programmet skall implementera två sätt för detektion av terminerade bakgrundsprocesser:

- Genom att läsa av förändrad statusinformation som OS-håller för barnprocesser via "pollning" med lämpligt systemanrop (wait()/waitpid())
- Genom att använda en signalhanterare (inte ett krav för den mindre kursen ID2200 fd.2G1504)

Vilken metod som används skall kunna väljas vid kompileringstillfället genom att ett makro SIGNALDETECTION definieras för att kompilera programmet så att bakgrundsprocessers terminering detekteras via signalhanterare.

Observera att barnprocesser kan förändra status, dvs signallera till föräldraprocessen av många olika anledningar förutom att de terminerat. Därför måste du kontrollera status för barnprocesserna - dvs. kontrollera varför de signallerat till föräldern så att du bara rapporterar att en barnprocess terminerat om den verkligen gjort det!.

Programmet skall, som alla program i kursen, vara skrivet i ANSI-standard C utan C++ utvidgningar.

OBS! Vi tillåter inte att du försöker kommunicera via globala variabler mellan en eventuell signalhanterare och huvudprogrammet.

1.2 En kortfattad beskrivning av en kommandotolk i UNIX

En kommandotolk i UNIX är det program som läser in och exekverar kommandon från användaren. Ett annat sätt att identifiera kommandotolken är att det är det program som ger dig prompten på kommandoraden. I det här fallet behöver vi inte bry oss om de mer avancerade möjligheter de flesta kommandotolkar brukar tillhandahåla för att t.ex länka ihop utmatning från ett kommando till inmatningen i ett annat genom pipes, möjligheter att dirigera om in och utmatning eller möjligheten att tolka (interpretera) inbyggda kommandospråk, sk. shell-scripts. Glömmer vi bort det så kommer vår kommandotolk att upprepa följande:

```
1) Läsa in ett kommando från kommandoraden
2) Parsa (tolka) kommandot för att se om det är ett inbyggt kommando
eller ej
3a) Om det var ett inbyggt kommando utför det t.ex genom en sekvens
av systemanrop - ta hand om eventuella fel.
3b) Det var inte ett inbyggt kommando. Skapa en ny process att exekvera
kommandot i.I den nya processen byter man exekverande program till det
program kommandot avsåg (första argumentet på kommandoraden). Se till
att alla parametrar till kommandot kommer med! Själva kommandotolken
skall sedan antingen vänta på att den nya processen avslutas om kom-
mandot exekveras i förgrunden eller direkt ge en ny prompt om det
exekveras i bakgrunden.
4) Innan den nya prompten ges skall man dock göra följande:
   i) Om det var ett förgrundskommando så skriv ut statistik om hur lång
      tid processen var igång
   ii) kontrollera om några bakgrundsprocesser avslutast och skriv ut
      information om dessa.
5) om kommandot exit ges avslutas kommandotolken.
```

1.3 Användbara systemanrop och biblioteksfunktioner

Följande systemanrop och biblioteksfunktioner kan vara användbara (för att utföra ett systemanrop använder man normalt en biblioteksfunktion som sedan utför ett systemanrop med samma namn):

- `fork()` - för att skapa en ny process
- `getrusage()` - för att få information om resursutnyttjandet (OBS! är inte standardiserad och fungerar inte bra på alla system)
- `gettimeofday()` - för att läsa av den “normala” klockan (wall-clock)
- `getenv()` - för att läsa av värdet på environmentvariabler
- `sigset()` - för att installera signalhanterare
- `sigaction(), sigset(), sighold()`, `sigrelse()` - för att (temporärt) blockera/tillåta signaler att nå processen (och en ev. signalhanterare). Observera att `sighold()/ sigrelse()` inte är tillgängliga på alla system
- `execve()` - för att exekvera ett nytt program i en process. Speciellt `execvp()` kan vara användbart

- `wait(), waitpid()` - för att vänta på barn-processer som har/skall terminera
- `chdir()` - för att byta arbetsdirectory för en process
- `exit()` - för att avbryta exekveringen (döda)av en process
- `fgets(), gets()` - för att läsa in kommandorader
- `strcmp(), strncmp()`- för att jämföra textsträngar
- `strlen()`- för att räkna antal tecken i textsträngar
- `strtok()` - kan underlätta att parsa/dela upp kommandoraden i textsträngar

1.4 Några förenklande antaganden

För att göra uppgiften lite enklare kan du anta att:

- Användaren inte skriver kommandon som är längre än 70 tecken
- Användaren inte ger kommandon med fler än 5 argument
- Du behöver inte parsa kommandoraden för att kunna hantera |, >, < eller **;**

1.5 Specialtecken i C

`'\n'` - är tecknet för “newline”

`'\0'` - är NULL-tecknet, dvs en nollställd byte som används för att terminera text-strängar i C.

1.6 Tips om arbetsgång

Innan du överhuvudtaget börjar fundera på att skriva kod bör du läsa man-sidorna för de system-anrop och biblioteksfunktioner du tror dig behöva använda!

- *Fortsätt och läs hela labPM - de flesta har nog nytta av att ha läst sektionerna 3 och 4 innan man ens börjar att fundera på hur labben skall lösas!*

Mitt förslag till hur du ska lösa uppgiften är att dela upp den i delmoment som du kan utveckla, förstå och lära av samt testa separat. På så sätt kan man enklare bygga upp grundfunkionaliteten coh sedan addera den mer avancerade funktionaliteten allt eftersom. Tanken är att från början, i steg ett, skriva två separata program, ett som kan läsa in och exekvera ett enkelt kommando i en och samma process och ett andra program som kan starta och hantera en barnprocess korrekt. Steg två är att kombinera dessa två program till ett som tillåter inläsning av ett kommando som exekveras i en separat process. I steg tre lägger du till en loop (slinga) som upprepar inläsningen av kommandon och har då skapat en enkel kommandotolk som kan hantera förgrundprocesser. I de avslutande faserna lägger du till hantering av bakgrundsprocesser och statistikutskrifter.

Det som brukar orsaka de flesta problemen i laborationen är att lägga till hanteringen av bak-grundsprocesser. Det man måste vara klar över är att skillnaden mellan bak-/förgrundsprocesser enbart ligger i hur/när man väntar på dem. När en process avslutas förändras dess status och den går til ett sk. zombie-läge. Förenklat betyder det att all information om processen finns kvar i pro-cesstabellen och att den informationen inte städas bort förrän någon process kontrollerar status för processen genom ett anrop till `wait()/waitpid()`. När en process terminerar skickas ochså en signal till föräldraprocessen. Signalen är SIGCHILD och skickas inte enbart då en process ter-minerar utan alltid då den förändrar status. Detta är anledningen att man alltid bör kontrollera sta-tus som man läser av via `wait()/waitpid()` och att du i den här laborationen alltid *måste* göra det!. Signaler köas normalt inte men i fallet att processer terminerat kan föräldraprocessen under vissa versioner av Solaris ta emot en signal för varje process som terminerat. Denna egen-skap bör man inte utnyttja.

Sammantaget ger detta åtminstonde två sätt att hantera väntandet på bakgrundsprocesser: man kan installera en signalhanterare som tar hand om väntandet kombinerat med pollning eller man bara polla information om det finns terminerade barnprocesser (bakgrundsprocesser). I bägge fallen krävs att man noga studera de manualblad som finns för relevanta systemanrop/bibliotekstfunktioner för att upptäcka de möjligheter som finns. Riktigt i detalj hur de här två angreppssätten skall implementeras vill jag inte gå in på - det tillhör de problem du förväntas lösa på egen hand i laborationen.

1.7 Förslag på arbetsgång.

- Gör ett enkelt program som inte gör något annat än en `execvp()` med ett ett-ordskommando som t.ex. `ls.` Börja med att hårdlöda in kommandot och utvidga det sedan till att ge en prompt och sedan kunna läsa in valfritt ett-ords kommando. (Du behöver inte använda just `execvp()` men jag tycker det är den enklaste i `exec`-familjen för den här uppgiften)t

---

- Bygg ett enkelt program som anropar `fork()`. Låt föräldra- och barnprocessen skriva ut olika meddelanden innan de avslutas och se till att föräldraprocessen hanterar barnprocessen korekt, dvs. kontrollerar dess status via `wait()/waitpid()`.

- Kombinera `execvp()` och `fork()` programmen så att barnprocessen utför kommandot `ls`.
- Lägg till att föräldraprocessen väntar på att barnprocessen skall terminera, dvs implementera förgrundsprocesser
- Lägg till en loop i föräldraprocessen för att kunna upprepa inläsningen av kommandon
- Lägg till det inbyggda `exit` kommando (loopen ska kunna avslutas med kommandot **`exit`**)

- Addera statistikutskrifter
- Tillåt mer avancerade kommandon som består av flera parametrar
- Lägg till det inbyggda kommandot för **`cd`**

- Utvidga koden till att kunna hantera både för- och bakgrundsprocesser (det här är för många det stora och jobbiga steget...)

- Se till att kommandotolken inte lämnar några zombie-processer efter sig
- Debugga och kommentera koden ordentligt, skriv rapporten och lämna in den

---

1.8 Vanliga fel och problemkällor

- Fundera över vilken inläsningsfunktion du bör välja för att läsa in kommandoraden men som undviker att du öppnar upp för en sk. “overflow”-attack!

- `fgets()` och `gets()` är snarlika men läser t.ex in olika mycket av en kommandorad innan de avbryter inläsningen vilket man inte alltid tänker på.
- Om du använder `strtok()` så bör du fundera på hur den hanterar “newline”-tecken, ‘\n’, i en textsträng.

De flesta problemen brukar trots allt uppkomma då man skall utvidga kommandotolken från att bara klara förgrundsprocesser till att också klara bakgrundsprocesser. Om du använder signalhanterare som fångar signaler så måste du fundera på:

- Man hanterar inte användningen av `wait()/waitpid()` korrekt, dvs. man funderar inte tillräckligt över om man ska vänta på någon barnprocess i största allmänhet, barnprocesser ur en viss mängd eller en specifik barnprocess.
- Vilka signaler skall fångas av signalhanterare? (för att se vilka signaler som finns kan du titta i `/usr/include/sys/signal.h`)

När man installerat en signalhanterare blir programmet definitivt asynkront - dvs. exekveringen kan avbrytas när som helst om en signal anländer som skall fångas av signalhanteraren. Att fundera på i det här sammanhanget kan vara:

- Ska du blockera vissa signaler i vissa delar av programmet för att få det att fungera korrekt och förenkla för dig?
- `wait()` och `waitpid()` kan returnera av två orsaker: att status ändrats för en barnprocess eller att processen som anropat `wait()/waitpid()` tar emot en signal - kan detta orsaka problem?
- Var återupptas exekveringen efter det att signalhanteraren exekverat klart?
- Har man otur avbryts systemanrop mitt i exekveringen då en process tar emot en signal i en signalhanterare - dessa systemanrop startas inte om! Därför måste man noga kontrollera returvärden från biblioteksfunktioner och systemanrop för att se om de blivit avbrutna och behöver återstartas![1]
- Hur väntar man bäst på förgrundsprocesser?

1.9 Testning

Ditt program skall naturligtvis klara de inbyggda kommandona `cd` och `exit` på ett korrekt sätt men också att hantera både förgrunds- och bakgrundsprocesser. Programmet skall naturligtvis kunna hantera felaktiga komandon, tomma kommandon och kommandon som inleds med blankslag. Speciellt bör du testa hur de olika processtyperna fungerar tillsammans t.ex genom att köra sekvenser som:

- start förgrundsprocess -> avslut förgrundsprocess -> start bakgrundsprocess -> avslut bakgrundsprocess
- start bakgrundsprocess -> start förgrundsprocess -> avslut förgrundsprocess -> avslut bakgrundsprocess

1. Egentligen borde du fundera på om detta kan ge problem om man t.ex avbryts mitt i en skrivning och sedan återstartar den. Kommer då t.ex delar av skrivningen att skrivas två gånger? Den här typen av synkroniseringsproblematik faller lite utanför kursens ramar och behandlas noggrannare i speciella kurser med namn som t.ex Programmering med Processer

- [start bakgrundsprocess]$^{+}$ -> start förgrundsprocess -> [avslut bakgrundsprocess]$^{+}$ -> avslut förgrundsprocess -> [avslut bakgrundsprocess]$^{+}$

Testa också att starta flera bakgrundsprocesser. Hitta gärna på flera tester!!! för testning kan man inte vara nog noggrann med!

1.10 Exempel på exekvering av ett “miniShell version 2.0”

```
imit.kth.se:Tue> miniShell.2.0
==>>pwd
/afs/it.kth.se/misc/courses/gru/2G1113.os/2G1504/
Spawned foreground process pid: 4988
Foreground process 4988 terminated
wallclock time:   3.124 msec
==>>netscape &
Spawned backgrond process pid: 4989
==>>loop
Give number:Spawned foreground process pid: 4991
4
Foreground process 4991 terminated
wallclock time:   6318.359 msec
==>>
Background process 4989 terminated
==>>netscape &
Spawned backgrond process pid: 4992
==>>netscape &
Spawned backgrond process pid: 4994
==>>loop
Give number:Spawned foreground process pid: 4996
4
Foreground process 4996 terminated
wallclock time:   2569.418 msec
==>>
Background process 4992 terminated
Background process 4994 terminated
==>>exit
imit.kth.se:Wed>
```

FIGUR 1. Exempel på exekvering av ett “miniShell version 2.0”

2.0 Att läsa “man”-blad

I UNIX kan man få information om b.la. UNIX-kommandon, systemanrop och biblioteksrutiner via manualblad (man-blad) genom att ge kommandot **`man`** följt av det man vill ha information om. Manualbladen är organiserade i sektioner där sektion 1 innehåller UNIX-kommandon, sektion 2 innehåller UNIX-systemanrop och sektion 3 innehåller biblioteksfunktioner. Kommandot **`man pwd`** ger t.ex information om UNIX-kommandot **`pwd`**, se FIGUR 2.

Observera att samma namn kan användas för ett UNIX-kommando, systemanrop och biblioteksfunktion. Det gör att man måste kunna ange vilken sektion man är intresserad av. För att t.ex. få information om systemanropet **`wait()`** snarare än UNIX-kommandot **`wait`** anger man kommandot **`man 2 wait`** eller **`man -s 2 wait`** (syntaxen skiljer lite mellan olika UNIX system/kommandotolkar).

Om man vill göra en fritextsökning för att hitta något man är intresserad av i manualsidorna ger man kommandot **`man -k`** *`sökord`*.

```
brax.imit.kth.se:Mon> man pwd
Reformatting page.  Wait... done

FSF                                    PWD(1)

NAME
    pwd - print name of current/working directory
SYNOPSIS
    pwd [OPTION]
DESCRIPTION
    Print the full filename of the current working directory.
        --help
        display this help and exit
         --version
        output version information and exit

REPORTING BUGS
    Report bugs to <bug-sh-utils@gnu.org>.

SEE ALSO
    The full documentation for pwd is maintained  as  a  Texinfo
    manual.  If the info and pwd programs are properly installed
    at your site, the command

info pwd

    should give you access to the complete manual.
```

FIGUR 2. Exempel på man kommando

Om man vill ange att man t.ex. avser biblioteksfunktionen `wait()` som finns i sektion 3 brukar man skriva detta som `wait(3)` när man skriver löpande text.

3.0 Att skapa processer

I operativsystemet UNIX har varje process ett unikt **p**rocess**id**entifikationsnummer som kallas för `PID`.

3.1 System-anropet fork (2)

En ny UNIX-process skapas med hjälp av systemanropet `fork(2)`. Systemanropet `fork` skapar en nästan exakt dubblett-kopia av den process som anropade `fork`, men med ett nytt unikt `PID`.

Processen som anropar `fork` kallas för förälder, *father* eller *parent-process* och den nya processen kallas för barn eller *child-process*. Efter genomförd fork-operation har både parent- och child-processen samma programkod och båda processerna kommer, efter retur från systemanropet fork, fortsätta exekveringen på samma plats i programmet, nämligen i programkoden omedelbart efter anropet av `fork()`.

För att kunna skilja de två processerna levererar systemanropet `fork()` vid returen olika parametervärden till parent och child-processerna. Värdet 0 lämnas som returparameter till child-processen medan parent-processen får ett returvärde som utgörs av `PID` för den nya processen dvs child-processens PID. En vanlig sekvens för att skapa en ny process visas i Figur FIGUR 3. nedan.

```
   /* Vanlig sekvens av fork() användning*/

int pid;
pid = fork();
if ( pid == 0 )
{  /* child-process kod */
   ...
}
else if ( pid > 0)
{  /* parent-process kod */
   ...
}
else /* if (pid < 0) */
{  /* SYSTEM ERROR */
   ...
}
```

FIGUR 3. fork() systemanrop

3.2 Systemanrop av typen `exec` (2)

För att kunna starta exekvering av en process med annan kod än father-processen används ett systemanrop av typen `exec.`

Vid varje systemanrop med någon av de olika `exec` ska parametrar levereras. Parametrarna ska ange vilken fil som ska exekveras i stället för den process som anropas av exec. Dessutom kan man leverera ett antal andra parametrar och därvid använda olika metoder för parameteröverföring dvs som värden eller pekare[2].

Om exec-anropet lyckas så “omvandlas” processen till en ny process som inte har någonting gemensamt med den process som den skapades från, förutom PID. Vilken process som helst kan anropa exec, dvs även en “father”-process.

Om `exec`-anropet misslyckas görs retur, med parametervärdet -1, till den anropande processen som då förväntas vidtaga lämpliga åtgärder, antingen rätta till felen och göra nytt försök eller ge larm.

2. Det finns flera rutiner som fungerar som interface till `execve`. Se `execve(2)` för mer information.

4.0 Kort om signaler

Signaler är ett sätt att kommunicera asynkront mellan processer i UNIX (de flesta OS har liknande mekanismer). Signaler kan liknas vid mjukvaruavbrott (software interrupts). Precis som med avbrott finns olika signaler, dessa har man gett olika nummer. Vill man på ett UNIX system ta reda på vilka signaler som finns tittar man lämpligen i header filen `/usr/include/signal.h` eller `/usr/include/sys/signal.h` där signalerna och deras nummer definieras.

Signaler kan genereras av olika orsaker:

- Användaren kan genom användarkommandot `kill(1)` skicka signaler till processer, t.ex. för att slå ihjäl processer man tappat kontakten med. T.ex skickar `kill -9 pid` en `SIGKILL` signal till processen med processid *`pid`* som då termineras.
- Processer kan skicka signaler till andra processer med systemanropet (biblioteksfunktionen) `kill(2)`. Anledningarna till att en process skickar en signal till en annan process kan vara flera: Man kan skicka `SIGKILL` till en annan process för att slå ihjäl den; man kan skicka en signal för att tala om att något har inträffat t.ex för att meddela en övervakande process att ett fel har inträffat; eller man kan använda signaler för att synkronisera processer.
- Operativsystemet ser till att automatiskt skicka signaler till processer då vissa händelser inträffar. T.ex skickar OS:et automatiskt en signal till föräldraprocessen då status för en av dess barnprocesser förändras, t.ex om barnprocessen terminerar.
  Skulle en process råka ut för ett exekveringsfel som upptäcks av hårdvaran, dvs då hårdvaran genererar ett avbrott pga. ett exekveringsfel, t.ex då "segmentation fault" och "bus error" inträffar, kommer detta att översättas av OS:et till en signal som skickas till den berörda processen. På det viset kan man, om man väljer att hantera signalen, skriva program som kan återhämta sig efter olika typer av exekveringsfel vilket kan vara nödvändigt i t.ex debuggers.
  (Tips: prova att skriva ett program som innehåller ett "segmentation fault" och prova att fånga signalen `SIGSEGV` i en signalhanterare).

Signaler hanteras och levereras till processer av operativsystemet. På liknande sätt som för avbrott, där varje avbrott bör ha en avbrottsrutin, så har man en signalhanterare per signal. En signalhanterare är en funktion som skall anropas då processen tar emot en signal. Varje process har sin egen uppsättning med signalhanterare till skillnad från avbrottsrutinerna som är gemensamma över hela maskinen. Operativsystemet tillhandahåller en "deafult"-uppsättning av signalhanterare för varje process. Vissa signaler ignoreras per default medan andra, som SIGSEGV, per default leder till att processen termineras (dessa beteenden kan man allstå ändra).

Om man själv vill installera en egen signalhanterare gör man det enklast via biblioteksfunktionen `sigset(3C)`. Med den kan man installera en egen funktion för varje signal (sk. signalhanterare). När processen tar emot en signal för vilken en signalhanterare installerats så avbryts processens normala exekvering och signalhanteraren körs. Då signalhanteraren är klar fortsätter processen sin normala exekvering från den punkt där den blev avbruten (om man inte valt att terminera processen i signalhanteraren).

En komplikation finns dock och det är att om processen blev avbruten i ett systemanrop så återstartas detta inte utan det kan avbrytas mitt i. Ofta får detta inga andra bieffekter förutom att systemanropet inte utförs. Ett vanligt exempel på när detta kan inträffa är om processen står och väntar på att läsa något, t.ex inmatning från användaren. Om då en signal kommer till processen kan systemanropet för I/O komma att avbrytas (eftersom man inte vet hur länge man skulle behövt vänta innan I/O avslutades). Effekten av det blir att biblioteksfunktionen man anropade, t.ex `gets()`, kommer att returnera utan att ha läst in något (eller bara läst in delar). Att detta inträffat kan man ta reda på genom att kontrollera returvärdet från biblioteksfunktionen som i

detta fall bör indikera ett fel. Exakt vilket fel som inträffat kan man få reda på via `errno` (se `<errno.h>`). När man tagit reda på vilket fel som inträffat kan man vidta åtgärder som t.ex att göra om inläsningen.

Om man inte vill bli avbruten, eller i delar av koden inte behöva fundera på att hantera situationer där systemanrop avbryts, kan man (tillfälligt) blockera mottagningen av signaler. Detta är analogt med att man (tillfälligt) kan stänga av avbrott. Hantering av signaler sker med systemanropet `sigaction(2)` enligt POSIX standarden. Det är ett av de mer komplicerade systemanropen i UNIX och eftersom tonvikten här är att förstå signaler snarare än enskilda systemanrop tillåter vi att man använder enklare biblioteksfunktioner som finns på en del system som SUN OS (Solaris) men t.ex inte under LINUX. Dessa funktioner är `sighold(3C)` som blockerar mottagningen av en signal och `sigrelse(3C)` som tar bort blockeringen av en signal. Man kan blockera eller ignorerar de flesta signaler utom `SIGKILL`. Blockerade signaler markeras i en bit-mapp där man har en bit per signal och tappas alltså inte bort utan levereras till processen då de inte längre är blockerade (t.ex. om man exekverar `sigrelse()`). Observera att de inte köas i egentlig mening eftersom det är en kö med bara en position (en bit), har flera signaler skickats till processen medan den haft den signalen blockerad kan man inte räkna med mer än att en signal når processen. Om man däremot väljer att ignorera signaler kommer dessa inte att “köas”. Det här sammantaget gör att det inte är helt enkelt att upptäcka när varje barnprocess terminerat.....

5.0 Förberedelsefrågor

1. Motivera varför det ofta är bra att exekvera kommandon i en separat process.
2. Vad händer om man inte i kommandotolken exekverar wait() för en barn-process som avslutas?
3. Hur skall man utläsa `SIGSEGV`?
4. Varför kan man inte blockera `SIGKILL`?
5. Hur skall man utläsa deklarationen `void (*disp)(int))`?
6. Vilket systemanrop använder `sigset(3C)` troligtvis för att installera en signalhanterare?
7. Hur gör man för att din kommandotolk inte skall termineras då en förgrundsprocess i den termineras med <Ctrl-c>?

8. Studera körningsexemplet nedan och förklara varför man inte har bytt “working directory” till `/home/ds/robertr` när man avslutat miniShell:et?

```
brax.imit.kth.se:Fri> miniShell
==>>pwd
/afs/it.kth.se/misc/courses/gru/2G1113.os/2G1504/
Spawned foreground process pid: 7691
Foreground process 7691 terminated
wallclock time:   2.937 msec
==>>cd
==>>pwd
/home/ds/robertr
Spawned foreground process pid: 7692
Foreground process 7692 terminated
wallclock time:   2.939 msec
==>>exit
brax.imit.kth.se:Fri> pwd
/afs/it.kth.se/misc/courses/gru/2G1113.os/2G1504/
```

6.0 Om rapporten

Rapporten skall vara utformad i enlighet med *“Allmänna instruktioner för LabPM”* som du hittar på kursens hemsida. Som alltid bör du tänka på att ha med:

- Ifyllt försättsblad
- En relativt utförlig beskrivning av uppgiften, hur du valt att lösa den och ditt program
- Svar på *samtliga* förberedelsefrågor från detta labPM
- Resultat från testkörningar
- Väl kommenterad källkod
- Uppgift om var källkoden finns och hur den skall kompileras (se till att vi kan accessa den!)

För att vi skall kunna förbättra labben och labbkursen vill vi också att du ägnar lite tid åt att också skriva in följande i din labbrapport:

- En uppskattning på hur mycket tid du lagt ned på att göra labben
- Betygssätt labPM, svårighetsgrad på labben och vad du fått ut av den på en skala 1-5
- Kan vi förbättra labben? Kom gärna med konstruktiv kritik!