Introduktion till processer

Grundläggande begrepp

Vi skriver ner ett par preciseringar av några begrepp som vi hittills stött på.

* En **fil** är en abstraktion som döljer krångliga detaljer om maskinvaran. En fil refererar ofta till ett fysiskt lagringsutrymme på någon form av lagringsmedium (sekundärminne såsom hårddisk, CD, DVD, Magnetband, floppy etc.) Men en fil kan också ligga i datorns primärminne då den behandlas.

Förtydligande: Inom *UNIX* är ovanstående alldeles för smalt. En **fil** i *UNIX* refererar till något som man hittar i filhierarkin och det kan vara *många många olika saker*. Mycket, väldigt mycket av ett *UNIX*-system är åtkomligt via filhierarkin. *UNIX* filhierarki är, som ni kanske minns, ett träd med roten `/` högst upp, under den finns katalogerna `/bin/` som innehåller körbara systemprogram som `ls` etc. Katalogen `/home/` innehåller hemkatalogerna för systemets användare. Troligtvis har ni redan något hum om detta från InfoMet.

Några exempel som illustrerar att filhierarkin är stor och omfattande är (som vi tidigare sett):

`/proc/`: Denna katalog innehåller underkataloger som är numrerade, vi ser till exempel katalogerna 1000, 1001 och 1002, dessa kataloger ligger inte på någon hårddisk utan är genererad information direkt från operativsystemets kärna (det innersta) som beskriver attribut och egenskaper hos processerna med nummer 1000, 1001 och 1002. På samma sätt har varje process en underkatalog i `/proc/` som beskriver dessa attribut och egenskaper.

`/dev/`: Denna katalog innehåller *device-nodes*, alltså *enhetsnoder* som möjliggör kontakt med de anslutna enheterna till systemet, här hittar vi periferienheter som nätverkskort, hårddiskspartitioner serieportar etc.

Med kommandot `man hier` får man fullständig information om var filhierarkin innehåller. Som vi ser så är begreppet "fil" i UNIX mycket vidare än begreppet "fil" i Windows. I våra laborationer kommer vi att stifta bekantskap med UNIX filbegrepp och se att vi kan få mycket information från det. Kommandot `ps` (som visar information om processer) gräver direkt i filstrukturen under `/proc/`.

På senare år har fokus mer hamnat på katalogen `/sys/` eftersom så mycket bara skyfflats in i `/proc/` så blev den till slut stor och ostrukturerad. Det kan hända att vi kommer att undersöka detta i kursens systemprogrammeringsdel.

Vi går vidare till nästa begrepp:

* En **process** är (ungefär) ett program som körs. Program finns ofta på datorns hårddisk i filer och man kan starta program genom att tex dubbelklicka på ikoner osv, men när ett program ligger på hårddisken är det inte en process. Det är först när programmet ***kör*** som man kan tala om en process. En process är alltså något som ***pågår***. En *aktivitet*. (Vidare kan ett program starta flera processer, men mer om det senare.)

Ytterligare förtydligande: En process pågår, ja, det är sant. Men det finns också många olika sorters processer, vi kommer att ge mening åt begreppen, *demon-process*, *zombie-process*, *barn-process*, *föräldra-process*, *system-process*, *användar-process* etc. Det finns en även en populär uppfattning om *UNIX* att "allt är filer", till och med processer är filer, ja, det är en sanning med modifikation, vi kan komma åt allt i ett datorsystem som som kör *UNIX* via noder i filhierarkin, alltså filerna, men det betyder inte att vi kan identifiera allt i ett *UNIX*-system som filer. Ett tydligt exempel på detta är processerna, ja, de finns alla representerade som filer under `/proc/` men jag tycker att vi mer får anse det som en *representation* av processerna, det *är* inte processerna själv. En mer precis fomrulering av "Allt är filer i *UNIX*" skulle kunna vara att "man kan komma åt och påverka det mesta i ett *UNIX*-system via filhierarkin".

Den sista termen var vad ett operativsystem var, vi avstår dock från precisering just nu (hela kursen kan ju uppfattas som en precisering av vad ett operativsystem är):

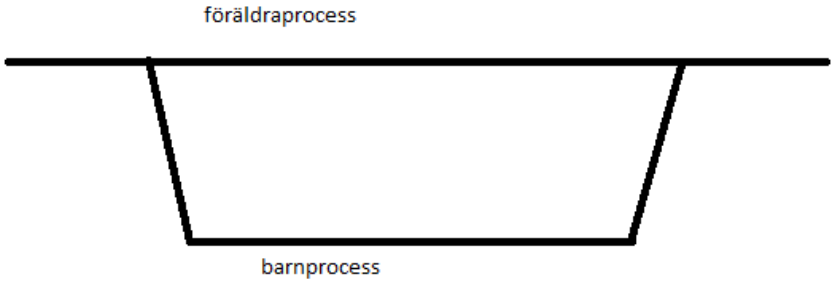
* Ett **operativsystem** (OS) är ett grundprogram som startar då datorn slås på. OS administrerar alla resurser till datorsystemet. (Resurser: mus, skärm, minne, CPU, ja allt.) Alla program som använder någon av datorsystemets resurser måste ha en pågående dialog/korrespondens med OS.

Operativsystemets kärna

Ovan har vi nämnt begreppet *kärna*. Det är den centrala delen av operativsystemet. Den viktigaste uppgiften hos kärnan är att skapa processer – det finns flera uppgifter som kärnan har – men att skapa processer är den centrala. Det finns också olika filosofier och designer kring operativsystem, man kan skapa OS med stor kärna som gör mycket och OS med liten kärna som gör lite (men delegerar uppgifter åt olika delsystem) men uppgiften att skapa processer ligger alltid hos kärnan.

Varje process har ett unikt identifikationsnummer, det så kallade process-id:t. Varje process har också en upplevelse att ha en bit av datorns arbetsminne där processens data och instruktioner lagras, det kallas processens *processbild – process image*.

I *UNIX* är alla processer (utom en) också *barnprocesser*, då en process skapas så är det alltid i respons till att en annan process begär av kärnan att en process skapas, den som begär att en process skapas kallas då *förälder* (*parent*) och den process som skapas kallas då *barn* (*child*). Det finns endast ett undantag från den här regeln och det är den speciella processen `init` som normalt är den första processen som skapas då ett *UNIX*-system startar. Den har normalt process-id 1. Eftersom det är den första processen som skapas så kan den inte ha någon förälder, men varje annan process som skapas har en förälder. Jag har hittat på ett tidsdiagram som kan användas för att illustrera förloppet då en process skapats. Det ser ut så här:



Båda processer (förälder och barn) symboliseras med en linje som har en utsträckning i tiden.

Normalt sett avlutas barnprocessen innan föräldern och det ser vi i diagrammet ovan på att att barnets linje avslutas innan förälderns linje avslutas. Barnets linje smälter då ihop med förälderns.

I och med den här strukturen (föräldrar/barn) så skapas ett släktträd där processen `init`, men process-id 1, är ett slags urförälder till alla processer. Detta träd kan vi se med kommandot `pstree`. Det här är resultatet av ett anrop av `pstree` på min dator då jag skrev dessa anteckningar:

```
init─┬─ModemManager───2*[{ModemManager}]
     ├─NetworkManager───3*[{NetworkManager}]
     ├─VBoxSVC─┬─VirtualBox───23*[{VirtualBox}]
     │         ├─VirtualBox───21*[{VirtualBox}]
     │         └─9*[{VBoxSVC}]
     ├─VBoxXPCOMIPCD
     ├─VirtualBox───3*[{VirtualBox}]
...
     │
     ├─lxpolkit───{lxpolkit}
     ├─lxterminal─┬─bash───pstree
     │            ├─gnome-pty-helpe
     │            └─{lxterminal}
     ├─menu-cached───{menu-cached}
...
```

Överst ser vi processen `init` som är förälder till allihop, vi ser att VirtualBox kör (eftersom jag också vill köra Windows, än så länge skriver jag dessa anteckningar i *LibreOffice* under *Windows*). Det som är särskilt intressant då är att `pstree` själv är en process, som består av programmet som körs för att ta fram alla processer. Vi ser att `pstree` är barn till `bash` som är barn till `lxterminal` som är barn till `init`.

Ett annat sätt att presentera körande processer är via kommandot `ps`. Här är ett resultat av kommandoraden `ps -e -o pid,ppid,comm`:

```
  PID  PPID COMMAND
    1     0 init
    2     0 kthreadd
    3     2 ksoftirqd/0
    6     2 migration/0
...
 4172  4139 VirtualBox
 4260     1 lxterminal
 4261  4260 gnome-pty-helpe
 4262  4260 bash
 4334  4139 VirtualBox
 4467  4260 bash
 4569  4467 ps
```

Vi ser `ps` (med processid 4569) själv som är en process som (liksom `pstree`) är ett barn till `bash` (med processid 4569) som är barn till `lxterminal` (med processid 4569) som är barn till `init` (med processid 4569).

Processen `lxterminal` är det program som ger oss den svarta rutan att skriva kommandon i. Den är en del av fönstersystemet som kallas *X-Windows-System* (som kom mycket tidigare innan *Microsoft Windows*). *Microsoft Windows* är dessutom ett helt operativsystem, *X-Windows-System*, är bara en del i ett gränssnitt och man kan installera ett *UNIX*-system med eller *utan* grafiskt gränssnitt. Det är en egenhet hos just operativsystemet *Microsoft Windows* att det grafiska gränssnittet är en del av operativsystemet – det finns till och med inbakat i kärnan. (I alla fall i tidigare versioner av *Windows*. Hur det är idag vet jag inte men jag tror inte att det är mycket bättre ställt. "Nya *Windows*" brukar mest vara ett nytt grafiskt gränssnitt och mer säkerhet.)

Skalprocesser

Men vad är det för en process som heter `bash`? Jo `bash` är en förkortning för *Bourne Again Shell*. Det är en så kallad *skalprocess* (*shell process*) och en skalprocess har flera funktioner. Dels erbjuder den en möjlighet att ge körande processer en miljö. Som nämnt tidigare så är processer barn till en process och om en skalprocess har flera barn så bildar dessa barn en grupp som har en gemensam miljö. Detta kan koordinera körningen av flera processer och användas för att få dem att samarbeta runt en uppgift. Vi kommer att se enkla exempel på det snart. Men skalprocesser har också en annan mycket viktig funktion och det är att de är ett gränssnitt till operativsystemets kärna. En skalprocess kan utföra så kallade *systemanrop* som är det man använder för att begära tjänster från operativsystemet. En vanlig tjänst är att skapa en ny process. Ordet "skal" används då som kontrast mot ordet "kärna" och skalet är ett gränssnitt mot kärnan. Skalet innehåller ofta ett helt programmeringsspråk och man kan alltså *programmera* i `bash`. Detta brukar kallas *script-programmering* och är ett sätt att automatisera olika uppgifter. Kommandona i detta scriptspråk är de vanliga *UNIX*-kommandona `ls`, `cp`, `mkdir` etc, men plus kontrollstrukturer som loopar och villkorssatser.

Enkel processkommunikation

Ett enkelt sätt att få processer att kommunicera är att använda den så kallade *pipen*. Det engelska ordet *pipe* betyder rör på svenska och en pipe är ett kommunikationsinstrument mellan två processer där den ena processen skickar data och den andra tar emot data. Ett exempel på detta är om vi ger följande kommando vid en prompt:

```
ls -l | wc -l
```

Den första processen `ls -l` listar alla filer i nuvarande katalog. Pipen är det lodräta strecket (`|`) mittemellan och tar resultatet av `ls -l` och skickar det vidare till processen `wc -l`. Kommandot `wc` heter "word count" och räknar det den får till sig så om man skriver `wc -l`, dvs lägger på ett `-l` på "`wc`" så räknar den istället antal rader (`l` = lines). På detta sätt skapar vi två processer som var för sig gör något enkelt, men tillsammans gör de något mer komplicerat. Resultatet av de två processerna blir alltså att vi räknar antalet filer i arbetskatalogen eftersom `ls -l` levererar en utskrift av så många rader som stämmer överens med antalet filer och `wc -l` räknar antalet rader som kommer från `ls -l`.

Andra egenskaper hos processer

Processer har som sagt process-id och föräldraprocesser. Men det finns flera egenskaper som blir tydliga när vi kopplar ihop processer med pipes. Då två processer kommunicerar via en pipe så

måste de ingå i något som kallas samma *processgrupp*. Att dela in processer i grupper är ett sätt att införa strukturer på processerna. Flera processgrupper kan dessutom ingå i något kallas en session som är nästa indelningsnivå. Vi kan se hur det här fungerar genom att betrakta följande anrop:

```
$ ps -o comm,pid,ppid,pgid,sess | cat | cat
```

Vi ska börja med att förklara denna kanske lite konstiga kommandorad. Först och främst första kommandot: `ps -o comm,pid,ppid,pgid,sess`. Vad gör detta kommando? Jo, det skriver ut alla processer och anger namnet på processen (`comm`), processid (`pid`), föräldraprocessid (`ppid`), processgrupper (`pgid`) och sist sessionsnummer (`sess`). Detta resultat skickas sedan vidare med första pipen till processen `cat` som faktiskt också bara skickar vidare resultatet via andra pipen till andra processen `cat` som skriver ut resultatet på skärmen. Resultatet ser ut sä här:

```
COMMAND           PID  PPID  PGID  SESS
bash             4738  4673  4738  4738
ps               4802  4738  4802  4738
cat              4803  4738  4802  4738
cat              4804  4738  4802  4738
```

Vi ser då framställningen av kommandot `ps` som hamnar i samma processgrupp som de andra processer som den kommunicerar med, dvs gruppen 4802, där alltså de båda `cat`-processerna finns. Alla dessa processer har skalprocessen `bash` som förälder och alla processerna, ingår i samma session som är 4738.

Dessa indelningsstrukturer är till för att kunna avgöra vilka processer som kan tala med varandra och påverkas av varandra. Vi kommer att diskutera sessioners betydelse mer då vi talar om så kallade demonprocesser. Men det ingår i del II av kursen.

Processers tillstånd och virtualisering av CPU:n

Den första delen av kurslitteraturen OSTEP, Operating Systems Three Easy Pieces, har titeln *Virtualization*, alltså virtualisering. Det är en mycket väl vald titel för virtualisering är en central princip inom operativsystemteknik. Vi har en fysisk resurs, datorn, men fysiska delar, CPU:n, hårddisken, pekdonet, minnet, och alla dessa fysiska resurser måste kunna användas av alla körande processer. Men enskilda processer kan inte förväntas ta hänsyn till andra processer, därför måste operativsystemet själv ta ansvaret för att koordinera och fördela resurserna mellan de körande processerna. Den centrala resursen i ett datorsystem är CPU:n. Utan den kan förstås ingenting ske. Operativsystem är idag skapade så att varje process upplever sig ha exklusiv tillgång till CPU:n, dvs operativsystemet virtualiserar CPU:n och ger alltså varje process en virtuell CPU. Det här är absolut inget nytt inom operativsystemtekniken, virtualisering har har varit en central del inom operativsystemtekniken sedan 60-talet.

Ni får själva läsa i kurslitteraturen detaljer om virtualisering, men vi ska kort nämna vilka följder det får för enskilda processer här. En process kan, bland annat med anledningen av virtualiseringen befinna sig i fundamentalt sett 3 olika tillstånd.

1. Körande (*Running*). 2. Blockerad (*Blocked*). 3. Redo att köra (*Ready*).

Då en process är *körande/running* är den i besittning av CPU:n. Virtualiseringen av CPU:n innebär att alla processer som finns i operativsystemet får turas om om CPU:n. När alltså en process är körande så exekveras dess instruktioner på CPU:n.

Då en process är *blockerad/blocked* väntar den på att något ska hända. Den *kan inte* köra, och den vanligaste anledningen brukar vara att den väntar på in/utmatning, (I/O). Det är meningslöst att köra vidare om inte de data finns tillgängliga som behövs för att köra så det här är ett naturligt tillstånd. En process som är blockerad kallas också *sovande/sleeping*.

Då en process är *redo att köra*/*ready* har den just kört slut på sin tid på CPU:n och måste vänta tills det blir dess tur igen. Det här är också ett naturligt tillstånd och är det som möjliggör att vi kan ha upplevelsen att flera processer kör på datorn samtidigt.

Egentligen är det en illusion att processerna kör precis samtidigt, det är ju bara en process åt gången som kan köra på CPU:n. Visserligen sker äkta parallellitet eftersom datorerna idag har flera processorer/processorkärnor men problematiken kvarstår: vi har inte en CPU per process, men virtualiseringen får det att verka som om varje process har en egen CPU.

En körning av kommandot `ps -l` nedan illustrerar tillstånden Running och Sleeping (blocked),

```
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  3347  3345  0  80   0 -  5255 -      pts/0    00:00:00 bash
0 R  1000  3392  3347  0  80   0 -  2154 -      pts/0    00:00:00 ps
$
```

Vi har alltså gett kommandot "`ps`" som frågar vilka körande processer som finns och `ps` svarar då att det finns två processer körandes, dels skalprocessen som `ps` körs i, `bash`, med PID = 3347, men också då `ps` själv som är den process som just ger oss upplysningen om vilka processer som kör. Den har PID = 3392 och har `bash` som förälder. Vi ser båda processernas tillstånd näst längst till vänster, `S` på `bash` markerar att `bash` är sovande (den väntar på att `ps` ska bli klar) och `R` på `ps` markerar att `ps` är en körande process.

Övriga tillstånd

Det finns tre tillstånd till som vi inte ska fördjupa oss i så mycket här, vi tar upp dem senare. Dessa tillstånd är

4. *Zombie*/*Defunct*, 5. *Avslutad*/*Terminated*, 6. *Ny*/*New.*

När en process är avslutad tas den bort ur systemet, och på samma sätt när en process är precis nyskapad, så läggs den in på tur för körning så dessa två tillstånd (5 och 6) är inte aktiva under så lång tid. Däremot kan en process vara en *Zombie* under en längre tid. Detta tillstånd är inte bra, vi måste undvika att processer blir så kallade zombier, men vi ska inte fördjupa oss i precis vad det är här. Vi kommer att lära oss mer av det när vi går in i nästa del av kursen: systemprogrammering, där ska vi med flit skapa zombier för att bättre förstå vad de är för något.