LINKÖPINGS UNIVERSITET

TDIU16: Process- och operativsystemprogrammering

Lab 3: Processhantering

Filip Strömbäck, Klas Arvidsson, Daniel Thorén

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2023

2023-03-15

Mål

För att kunna utnyttja resurser effektivt i operativsystemet vill vi kunna köra flera processer samtidigt. Detta kräver att operativsystemet kan hålla koll på processerna så att de kan samarbeta där det behövs. Operativsystemet behöver också se till att synkronisera åtkomst till delade resurser så att problem inte uppstår. I den här laborationen kommer vi därför att implementera en processlista så att vi kan hålla koll på vilka processer som finns i systemet. Vi kommer också att titta på ett par specifika fall där vi måste synkronisera exekeveringen så att allt fungerar som det ska.

Målet med laborationen är att implementera följande systemanrop i Pintos:

- `exec` – starta en ny process
- `wait` – vänta på att en process avslutar
- `exit` – avsluta en process, och skicka status till föräldraprocessen

I slutet av handledningen finns en mer detaljerad beskrivning av vad systemanropen ska göra.

Notera: `exit` påbörjades i förra laborationen, men vi kommer att bygga färdigt hanteringen av `status`-parametern i den här labben.

Mer information om detta finns i avsnittet ”Tråd- och processhantering i Pintos” i Pintos-Wiki.

Efter att du har implementerat alla delar av laborationen ska programmet `src/examples/longrun.c` fungera felfritt. Varje del har också ett eget testprogram så att du kan testa din implementation innan allt är klart.

Del A Övning på semaforer

I mappen `src/standalone/lab3a` finns filen `exec-wait.c`. Filen innehåller bland annat en funktion `do_work` som vi tänker oss gör tunga beräkningar. Vi vill därför köra funktionen parallellt i flera trådar för att utnyttja flera kärnor i vår CPU.

För att göra detta innehåller filen funktionerna `exec` och `wait` för att enkelt hantera parallellismen i resten av programmet. Tanken är att `exec` ska starta en tråd som kör `do_work` i en separat tråd. Det ska sedan gå att vänta på att tråden är klar genom att anropa `wait`. Med hjälp av dessa funktioner kan vi sedan i `main` starta hur många trådar vi vill genom att anropa `exec` flera gånger och sedan vänta på att alla blivit klara genom att anropa `wait`. Detta fungerar som `exec` och `wait` kommer att fungera i Pintos senare.

Som du säkert märker fungerar inte programmet som det ska. Ofta kommer första och andra resultatet vara 0, vilket inte är korrekt. Detta beror så klart på att synkronisering helt saknas. Uppgiften är att lösa problemet med hjälp av semaforer.

Du kan lösa denna laboration på två sätt. Dels genom att använda visualiseringsverktyget, och dels genom att kompilera och köra koden i terminalen (separat från Pintos). Nedan beskrivs båda sätten:

A.1 Visualiseringsverktyget

På skolans datorer och ThinLinc kan du starta verktyget med kommandot `/courses/TDIU16/progvis/start` Du kan även köra verktyget på din egen dator om du vill. Instruktioner för detta finns på kurshemsidan (välj ”Visualiseringsverktyg” i vänstermenyn).

När du har startat verktyget kan du välja *File* → *Open...* och sedan välja källkodsfilen för labben. Koden kompileras automatiskt, så du behöver inte göra det manuellt. Du kan sedan välja i vilken ordning du vill att de olika trådarna ska köras. På så vis kan du se vad som händer i olika fall. Målet är att din implementation ska fungera korrekt oavsett i vilken ordning du kör trådarna. Verktyget kan också automatiskt testa alla

möjliga kombinationer åt dig om du väljer *Run* → *Look for errors*. Detta ger en indikation på om din lösning är korrekt eller inte.

Om du har gjort ändringar i källkoden kan du välja *File* → *Reload* (Ctrl+Shift+R) för att ladda in programmet igen.

A.2 I terminalen

Om du vill lösa labben med vanliga utvecklingsverktyg går även detta. Gå till mappen `src/standalone/lab3a` och skriv `make exec-wait` för att kompilera programmet. Detta skapar filen `exec-wait` som kan köras med `./exec-wait`. Du kan självklart använda GDB som vanligt: `gdb ./exec-wait` eller `gdb -tui ./exec-wait`.

Del B Starta processer (90% förarbete, 10% kodning)

I Pintos finns en funktion för att starta processer i filen `src/userprog/process.c` som heter `process_execute`. Den anropas i nuläget bara från `src/threads/init.c` för att starta den första processen. Vi kommer även använda den för att implementera systemanropet `exec` i nästa del.

Den nuvarande implementationen består i huvudsak av två funktioner: `process_execute` som har till uppgift att starta en ny kernel-tråd för den nya processen, och `start_process` som körs av den nya kernel-tråden och har till uppgift att ladda den körbara filen och starta programmet. Funktionen `process_execute` ska returnera ett process-id för den process som just skapats ifall allt gick bra, eller -1 om något gick fel.

Den nuvarande implementationen är inte komplett. Synkroniseringen mellan `process_execute` och `start_process` fungerar inte som den ska. För att kunna returnera korrekt information måste `process_execute` vänta på att `start_process` kommit tillräckligt långt i sitt arbete. Annars kommer inte `process_execute` kunna veta om det gick bra att starta processen eller inte, och kanske inte heller vilket process-id som den nya processen kommer att få. Ett annat problem är att `process_execute` kanske börjar städa upp saker som den nya processen behöver för att kunna starta om den inte väntar på `start_process`.

I Pintos-Wiki, under rubriken ”Tråd- och processhantering i Pintos” finns en mer ingående beskrivning av vad som händer. Där framgår att punkt 3 (vänta), 4 (se om allt gick bra) och e (skicka resultatet) inte fungerar korrekt i den nuvarande koden.

Implementera den synkronisering som behövs för att lösa dessa punkter korrekt och utan att använda ”busy-wait”. När du är klar skall det inte finnas några spår av `power_off` raden kvar i koden. Du skall se till att vänta precis så mycket som behövs (vare sig det är inget alls eller 7 timmar). Tänk på att det måste fungera korrekt för godtyckligt antal processer. Att använda globala synkroniseringsvariabler kommer *inte* att fungera (Varför?).

Det finns debugutskrifter i början på `process_execute`, i början av `start_process`, och i slutet av `process_execute`. De skall *alltid* skrivas ut i samma ordning som i föregående mening när du gjort rätt. Om `start_process` utskriften vid någon körning kommer sist har du gjort fel. Gör du fel är det även mycket troligt att du får minnesfel, då `start_process` kommer använda variabler som blivit ogiltiga sedan `process_execute` returnerat. Att allokera dynamiskt minne med `malloc` i `process_execute`, och sedan frigöra det minnet i `start_process` är **inte** en godkänd lösning på sådana problem (det är alltid en god vana att samma funktion, modul eller tråd som allokerar minne ansvarar för att frigöra detsamma). Lös allt genom att vänta lagom länge.

B.1 Testprogram

När du är klar, tänk noga igenom din lösning med ledning av ovan information, och testa sedan genom följande tre testfall:

- Starta ett användarprogram enligt föregående uppgifter, t.ex. `sumargv`. I debug-utskriften för returvärdet från `process_execute` skall du se ett giltigt id, t.ex. 3.
- Gör som ovan, men skriv fel programnamn, skriv t.ex. `sumsum` istället för `sumargv` sist på raden. Detta gör att `load` i `start_process` kommer misslyckas. I debug-utskriften för returvärdet från `process_execute` skall du se -1.
- Kör ett program med argument `-tcl=2`. Detta simulerar att `thread_create` misslyckas skapa tråden vid andra anropet. I debug-utskriften för returvärdet från `process_execute` skall du se -1, eftersom det inte blev någon process (inte ens en tråd!).

```
pintos -p ../examples/sumargv -a sumargv -v -k --fs-disk=2 \
        -- -f -q -tcl=2 run 'sumargv 1 2 3'
```

Notera: Eftersom det inte är deterministiskt i vilken ordning som trådarna körs så ger testfallen ovan bara en indikation på om din lösning är korrekt eller inte. Det vill säga: om något av testfallen misslyckas så har du garanterat en bugg i din implementation. Men om alla testfallen går igenom så är det inte garanterat att implementationen är korrekt — du kanske bara hade ”tur” när du körde testerna. För att garantera att implementationen är korrekt måste man fundera noga på vad som kan hända och se till så att koden alltid gör rätt, oavsett vad som händer. Är du osäker, be assistenten om hjälp att resonera kring din lösning!

Del C Processlistan

För att kunna implementera systemanropen `exec` och `wait` korrekt så behöver vi lagra information om alla processer som körs i systemet. Ett bra sätt att göra det är genom att använda en variant av vår associativa databehållare från tidigare.

Du kan själv välja hur du tilldelar process-id till processer. Det enda kravet är att varje process måste ha ett process-id som är unikt i hela systemet (se exemplet i slutet av dokumentet). Ett sätt är att låta process-id:t vara den nyckel som din associativa databehållare returnerar från sin `insert`-funktion. Ett annat är att återanvända det tråd-id som finns i Pintos. Det finns så klart andra alternativ också, men dessa är de vanligaste.

C.1 Förberedelse

Vid implementation av `exec` (modifiering av `process_execute` och `start_process`) finns ett antal möjligheter att placera koden som lägger till en process i listan. Tänk igenom följande 4 alternativ. De är listade ungefär efter i vilken ordning de olika alternativen exekveras i koden:

1. Lägg till nya processen i listan före anropet av `thread_create` i `process_execute`
2. Lägg till nya processen i listan inuti `thread_create`
3. Lägg till nya processen i listan inuti `start_process`
4. Lägg till nya processen i listan efter anropet av `thread_create` i `process_execute`

För att utvärdera varje alternativ skall du tänka igenom följande fem frågor (det ger alltså 5 svar per alternativ, dvs. 20 svar totalt). **För optimal placering bör svaret på varje fråga vara positivt.**

Tänk på att du sedan förra uppgiften har en mekanism för att kommunicera värden till och från den nya uppgiften. En annan detalj som är värd att tänka på är att funktionen `thread_tid()` returnerar tråd-id:t för tråden som *anropar* funktionen (detta är relevant om du använder tråd-id som process-id). Nu till de fem frågorna att utvärdera för varje alternativ (möjliga svar inom parentes):

- Kommer den nya tråden att lägga till sin egen process i processlistan?

  (Ja / Nej, det gör förälder-tråden)

- Är förälderns process-id tillgängligt när informationen om den nya processen skall läggas till i processlistan?

  (Ja, direkt / Ja, det kan lätt ordnas / Nej, det går absolut inte att få tag på)

- Är den nya processens process-id tillgängligt vid den placeringen?

  (Ja, direkt / Ja, det kan lätt ordnas / Nej, det går absolut inte att få tag på)

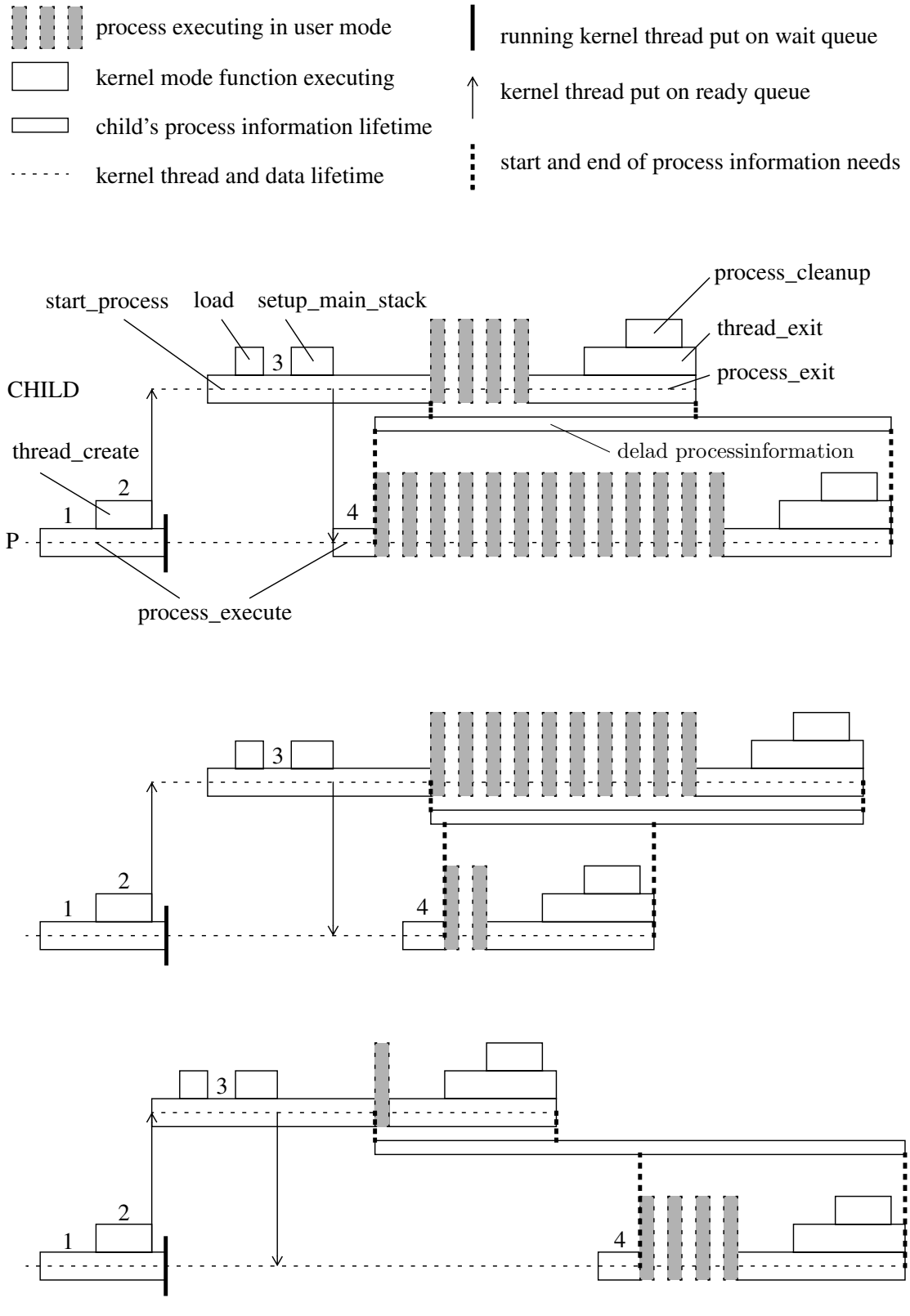
- Processens id kommer att användas senare, när barnprocessen når `process_cleanup`, för att kunna ta bort processen ur processlistan. Är det *garanterat* att koden som lägger till den nya processen i processlistan *alltid* kommer exekveras *innan* den nya tråden exekverar `process_cleanup`? *Detta är en viktig punkt.*

  (Ja / Nej, den nya processen kan hinna avsluta innan den läggs till i listan)

- Överensstämmer uppgiften att lägga till en ny process i processlistan med intentionen av den funktionen du utför det? (Se Pintos-Wiki under rubriken ”Tråd- och processhantering i Pintos”, se särskilt underrubriken ”Starta en process”, kort sammanfattat nedan)

  (Ja, absolut / Ja, ganska bra / Nej, inte alls)

Observera att det är mycket viktigt att följa Pintos intentioner eftersom funktionerna för process-hantering i `src/userprog/process.h` används internt i `src/threads/init.c`. Om de funktioner som anropas där inte sätter upp en komlett process uppstår fel senare. Intentionen med funktionerna `process_execute` och `start_process` är just att starta en process, d.v.s. göra allt som behövs för att starta och registrera en ny en process korrekt. Intentionen med `thread_create` är att starta en funktion i en egen kernel-tråd och inget utöver det. På nästa sida visas en figur över tre tänkbara exekveringsordningar (det finns fler) med alternativen 1–4 indikerade.


process executing in user mode
kernel mode function executing
child's process information lifetime
kernel thread and data lifetime
running kernel thread put on wait queue
kernel thread put on ready queue
start and end of process information needs
process_cleanup
start_process
load
setup_main_stack
thread_exit
process_exit
CHILD
delad processinformation
thread_create
P
process_execute
1
2
3
4

C.2 Uppgift

Gör en kopia av din associativa databehållare från tidigare laborationer och lägg kopian i filerna `src/userprog/plist.h` och `src/userprog/plist.c`. Eftersom alla namn i C är globala så behöver du byta namn på funktionerna i `plist`-filerna till något som inte kolliderar med namnen i `flist`-filerna. Glömmer du detta steget kommer implementationen att bete sig mycket konstigt (men du får varningar från länkaren).

I och med att processlistan är global i systemet (det finns bara en instans i hela Pintos) så finns det en poäng med att lagra den som en global variabel inuti `plist.c`, och att ta bort första parametern från alla funktioner i `plist.h` och `plist.c`. Eftersom det bara finns en instans av listan är det onödigt att skicka med den till alla funktioner. Dessutom kommer detta göra det enklare att resonera om vilken kod som kan göra ändringar i den globala variabeln, vilket underlättar i kommande labbar.

Lägg sedan till systemanropen `plist` och `sleep` i Pintos enligt definitionerna i slutet av dokumentet (de finns inte i `src/lib/syscall.h` från början). Implementera sedan systemanropen `plist`, `sleep` och `exec`. Tänk på att koden i `src/userprog/syscall.c` i princip bara ska anropa lämpliga funktioner i andra ställen av systemet. Detta beror på att den första processen inte startas via ett systemanrop.

Planera sedan din implementation av processlistan enligt förberedelsen ovan. Redovisa för din assistent var du planerar att placera koden som sätter in en process i processlistan. Att välja rätt plats från början underlättar arbetet senare (du slipper ”gör om gör rätt”).

Modifiera sedan implementationen av `process_execute` och `start_process` så att processer läggs till i din processlista. Lägg också till kod som uppdaterar processlistan när `exit` anropas (det finns en funktion som heter `process_exit` som kan användas för detta), och se till att element i processlistan som inte längre behövs tas bort.

Grafen på föregående sida illustrerar när processinformationen i processlistan behöver finnas tillgänglig relativt både processen själv (*CHILD*) och dess förälder (*P*). Som du ser så kan denna data inte lagras i någon av de två trådarna (varken *CHILD* eller *P*) eftersom den kan behövas av ena tråden när den andra inte längre finns. Ytterligare en komplikation är att tråden som startar den första processen via `process_execute` inte har startats via `process_execute` — den första processen har alltså ingen förälder.

C.3 Testprogram

Filen `src/examples/longrun_nowait.c` innehåller ett testprogram för att testa `exec` och `exit`. Kommentaren i början av filen innehåller en lämplig kommandorad för att köra programmet.

Studera implementationen av `longrun_nowait.c` och se vad programmet gör. Lägg till ett anrop till systemanropet `plist` på lämpligt ställe i implementationen (exempelvis i slutet av sista loopen), kör programmet igen och studera hur din processlista ser ut (lägg tid på att göra utskriften snygg, det underlättar felsökning). Fundera på om utskriften stämmer överens med hur det borde se ut baserat på hur processlistan borde bete sig.

I och med att vi ännu inte har implementerat `wait` går det inte säga exakt hur utskriften borde se ut (det varierar beroende på exakt hur din implementation ser ut, och varierar från körning till körning). Saker att titta på är:

- Blir processer markerade som avslutade när de har avslutat sig?
- Städas element upp ur processlistan som de ska?
- Kraschar systemet ibland?

Det är också värt att variera lite olika parametrar i testet. Du kan exempelvis variera antalet processer som startas genom att variera parametrarna till `nowait`-programmet (färre processer gör det enklare att se vad som händer, men risken att något ”konstigt” händer minskar också). Det är också en bra idé att variera

tiden som `longrun_nowait` väntar. Ökas tiden så är det större sannolikhet att alla startade processer avslutar innan `longrun_nowait` blir klar (och att processlistan borde vara näst intill tom i slutet av programmet). Minskas den så kommer flera processer vara kvar längre, vilket testar andra delar av din implementation.

Du kan också testa programmet `src/examples/wait_test.c`, men eftersom det programmet är tänkt att testa systemanropet `wait()` kan du tillfälligt behöva kommentera ut anropen till `wait` och lägga till anrop till `plist()` på lämpliga ställen.

Notera: Du kommer fortfarande se meddelandet om att Pintos försöker stänga av sig innan alla trådar har stängts av (`ERROR: Main thread about to poweroff..`) i och med att vi inte har implementerat `process_wait` ännu. Det är till och med så att det meddelandet kanske visas trots att `process_wait` är korrekt. Varför?

Del D Vänta på processer

Nu när processlistan (förhoppningsvis) fungerar som den ska kan vi till slut implementera systemanropet `wait`. Gör detta genom att utöka informationen i listan med aktiva processer med den information som behövs för att spara `status` i processinformationen när en process avslutar. När en process anropar `wait` skall statusinformationen *som sparats eller kommer att sparas* för angiven barnprocess returneras från `wait` när den finns tillgänglig. Har du implementerat `exec` och `exit` korrekt bör detta vara enkelt, eftersom processinformationen hela tiden finns tillgänglig för både processen som anropar `wait` (föräldern) och processen som sparar sin `status` (barnet). Tänk noga igenom var du placerar synkroniseringsvariabler som behövs för att låta föräldern vänta tills barnet är klart. Det får inte finnas någon risk att föräldern väcks av något annat barn än det den väntar på. Och det får inte finnas någon risk att variablerna är borttagna när de behövs. Tänk också på att det bara är föräldraprocesser som får vänta på sina barn. Det ska alltså inte gå att vänta på vilka processer som helst med hjälp av `wait`. I och med att det bara ska gå att vänta på ett barn en gång så kan processinformationen för barnet tas bort redan när föräldern har väntat på ett barn.

Tänk på att placera din kod i filen `src/userprog/process.c`. Koden i `src/userprog/syscall.c` skall endast innehålla ett anrop till relevant funktion. Att du måste göra så beror på att den första processen inte startas via ett systemanrop, och att kernel inte heller väntar på den första processen via ett systemanrop.

D.1 Testprogram

Filen `src/examples/wait_test.c` innehåller ett enkelt testprogram för att testa så att systemanropen `exec` och `wait` verkar fungera som de ska i en liten skala. Filen `src/examples/longrun.c` innehåller ett större testprogram som stresstestar implementationen lite mer. Kommentaren i början av `src/examples/longrun.c` innehåller en lämplig kommandorad för att köra programmet.

Hittills har du vid varje Pintos-körning fått en felutskrift om att huvudprogrammet (main) försöker stänga av datorn innan alla trådar är klara. När din lösning fungerar korrekt skall du inte längre få *några* `ERROR` vid körning av `examples/wait_test` eller `examples/longrun`.

Notera: programmet `examples/longrun_nowait` kan fortfarande ge `ERROR`. Varför?

Notera: ibland kan du se meddelandet *load: ... open failed*. Det beror på att filsystemet inte ännu är synkroniserat. Vi kommer lösa det i nästa laboration.

När `wait` fungerar korrekt kan du även börja använda Pintos egna tester. Testet `tests/userprog/wait-simple` kan då användas som ett bra förstasteg (du kan köra det med kommandot `pintos-single-test <testnamn>`, se Pintos-Wiki för mer information).

Notera: Pintos egna tester ställer tre viktiga krav på din implementation:

1. Det får *inte* förekomma några debug-utskrifter som *inte* startar med fyrkant+mellanslag (`"# "`).

2. Implementationen av wait *måste* fungera korrekt.
3. När en process avslutar *måste* den skriva ut sin *exit_status* (parametern *status* till systemanropet *exit* om processen avslutar via ett systemanrop, -1 om processen avslutas av kernel på grund av något fel.) Den `printf` som behövs finns i `process_cleanup` men du måste *se till att variabeln status har rätt värde innan utskriften sker*. Du måste också se till att utskriften sker *innan* föräldratråden notifieras (dvs. utskriften måste garanterat ske innan föräldern vaknar från anropet till `wait`).

   ```
   printf("%s: exit(%d)\n", thread_name(), status);
   ```

Se avsnittet ”Automatiska tester” i Pintos-Wiki för information om de automatiska testerna. I slutet av kursen ska alla automatiska tester fungera.

Tips: Testa även att lägga till flaggan `-L` (som i *leak check*) innan ordet `run` i kommandoraden. Detta gör att systemet kommer att visa dig minnesläckor i slutet av körningen. Vid den här punkten bör inga minnesläckor finnas för testprogrammet `longrun`. I och med att vi inte har synkroniserat filsystemet ännu så kan det dock ibland finnas minnesläckor ifrån `inode`.

Beskrivning av systemanropen

Systemanropen `exec` och `wait` introducerar konceptet *process-id*. Dessa fungerar mycket likt fildeskriptorer: de är ett heltal som vi använder för att referera till en process. Ett program kan inte anta särskilt mycket om ett process-id annat än att vi får ett unikt process-id från `exec`, och att vi sedan kan skicka det process-id:t till `wait`. Vi vill alltså kunna göra som i programmet nedan:

```
int main(void) {
  pid_t child = exec("sumargv 1 2 3"); // Starta en barnprocess som summerar argumenten
  int result  = wait(child);           // Vänta på att den blev klar och hämta resultatet
  printf("Sum: %d\n", result);         // Skriv ut resultatet.
  return 0;
}
```

Vi kan självklart starta flera processer samtidigt också:

```
int main(void) {
  pid_t child1 = exec("sumargv 1 2 3");
  pid_t child2 = exec("sumargv 2 3 4");
  int result1  = wait(child1);
  int result2  = wait(child2);
  printf("Sum1: %d\nSum 2: %d\n", result1, result2);
  return 0;
}
```

Gemensamt för båda programmen är att de inte bryr sig om vilket process-id som returneras från `exec` (annat än att det inte är -1). Detta för att programmet inte kan veta vilka process-id:n som är lediga när `exec` anropas. Detta innebär att Kernel kan välja hur den tilldelar pocess-id:n till processer. Det gör att vi kan välja ett sätt som gör vår implementation så smidig som möjligt. Eftersom vi kommer använda en version av vår associativa databehållare för att implementera processlistan så kan det vara smidigt att använda ID:t som den returnerar för att representera ett process-id, men det finns många möjligheter.

Mekanismen för hur systemanrop anropas finns beskriven i mer detalj under rubriken ”Systemanrop i Pintos” i Pintos-Wiki.

Mer detaljer över hur systemanropen ska fungera finns under rubriken ”Systemanrop” i avsnittet ”Tråd- och processhantering i Pintos” i Pintos-Wiki.

- `pid_t exec(const char *file);`

  Startar en ny process som kör programmet som finns i filen `file` (en C-sträng). Om allt gick bra returneras den nya processens *process-id*. Annars returneras -1.

  Notera: typen `pid_t` är ett alias för `int`, men gör det enklare att hålla koll på vad som är ett process-id.

- `void exit(int status) NO_RETURN;`

  Avslutar processen som anropade `exit` med omedelbar effekt. Processen kommer inte att fortsätta exekevera efteråt eftersom den stängdes av (därav är funktionen markerad med `NO_RETURN`).

  Värdet i `status` ska sparas undan i processlistan så att processens förälder senare ska kunna hämta det med hjälp av systemanropet `wait`.

- `int wait(pid_t child);`

  Väntar på att en process som tidigare startats med `exec` avslutas. Om processen redan har avslutats behöver `wait` inte vänta, och kan i stället returnera direkt. Värdet som `wait` returnerar är den `status` som den avslutade processen skickade som parameter till `exit`.

  Det är bara möjligt att anropa `wait` en gång för varje anrop till `exec`. Det vill säga: det går bara att vänta på varje process en gång. Efter att `wait` har returnerat får systemet återanvända process-id:t till en ny process.

  Om processen `child` inte finns, eller refererar till en process som inte startades av processen som anropade `wait`, ska `wait` direkt returnera -1.

- `void plist(void);`

  Skriv ut innehållet i processlistan till skärmen. Detta används för att enklare felsöka din implementation. I och med att vi kommer att använda systemanropet ganska mycket är det värt att lägga tid på att se till att utskriften är snygg och lättläst, lämpligtvis i form av en tabell.

- `void sleep(int millis);`

  Gör så att processen som anropade systemanropet sover i `millis` antal millisekunder. Detta kan implementeras exempelvis med funktionen `timer_msleep` i `src/devices/timer.h`. Denna funktion behövs av några av testprogrammen som används i den här labben.