Deluppgift 10 Starta den första processen (90% förarbete, 10% kodning)

Saker som kan vara bra att känna till

Ett operativsystem startar med något som brukar kallas boot-sekvens. Det är en benämning på allt som händer när operativsystemet startar. Kod för operativsystemest skall läsas från disk och startas, systemets olika moduler skall initieras (trådhantering, minneshantering, diskhantering etc.) och tillgänglig hårdvara skall detekteras och initieras. När du vill lägga till egna datastrukturer eller medlemmar i befintliga datastrukturer är det bra att känna till var och i vilken ordning olika moduler initieras. Du kan till exempel inte allokera minne med `malloc` förrän minneshanteringsmodulen är initierad.

När systemet är initierat är det dags att starta den första processen. Detta sker genom anrop av funktionen `process_execute` som finns i filen `userprog/process.c`. Denna fil innehåller allt som har med processhantering att göra. I PintOS är en process nästan synonym med en kernel-tråd, beroende av att varje process “drivs” av en kernel-tråd. Varje process har alltså exakt en kernel-tråd. Däremot är inte alla kernel-trådar processer. Funktioner för trådhantering finns i filen `threads/thread.c` och motsvarande header-fil. Trådmodulen är den viktigaste delen i PintOS.

PintOS från start till slut, grov översikt

Detta är huvuddragen av hur PintOS exekverar i ordning från start till slut. Mycket är naturligtvis utelämnat. PintOS huvudprogram (main) finns i `threads/init.c`.

```
thread_init ();  /* initiera trådsystem */
malloc_init ();  /* dynamisk minnesallokering */
timer_init ();   /* generera regelbundna avbrott */
intr_init ();    /* avbrottshanterare */
kbd_init ();     /* tangentbord */
syscall_init (); /* systemanrop */
thread_start (); /* skapa idle-tråd */
disk_init ();    /* diskhantering */

/* starta och vänta på första processen
 * den ny processen läggs på ready-kön
 * kan börja exekvera vilket ögonblick som helst
 * kan också dröja länge innan den startar exekveringen
 */
pid = process_execute (task);

/* vänta på att ovan process blir klar
 * om detta inte fungerar går Pintos direkt vidare och avslutar
 */
process_wait (pid);

/* om flagga -q användes, stäng av datorn (emulatorn) */
if (power_off_when_done)
    power_off ();
```

```
/* avsluta huvudtråden
 * (om poweroff kördes kommer vi naturligtvis inte hit)
 * andra trådar kan fortfarande finnas kvar och köra klart
 * till slut finns endast idle-tråden kvar
 */
thread_exit ();
```

Tråd- och processhantering i PintOS

Följande funktioner för tråd- och processhantering är centrala i PintOS. Du kommer behöva ändra alla process-funktioner någon gång. Funktionernas ansvarsfördelning som du förväntas följa:

`process_execute()`
: Skapar en tråd med uppgift att läsa in och starta exekveringen av en ny process.
Ser till att all data associerad med en process är korrekt uppsatt.
Returnerar ett unikt id som identifierar processen i framtiden *men endast om tråden lyckades starta den nya processen*. Annars returneras -1.

`start_process()`
: En hjälpfunktion till `process_execute` som gör större delen av jobbet i en egen tråd. Läser in processen från disk, allokerar all information relevant för hantering och spårning av en process, sätter upp processens initiala stack och startar processen i user-mode. Det är här vi upptäcker om en process kan startas eller ej.

`process_cleanup()`
: Avallokerar en process alla resurser som kernel reserverade under exekveringen av `start_process` och `process_execute`, samt resurser som allokerats via systemanrop. *Denna funktion anropas automatiskt av* `thread_exit` *så att trådar som är processer kan avallokera sina data.* Denna funktion i sig avslutar ingen tråd eller process, den avallokerar bara data, såsom processens minnesrymd. Följaktligen är det `thread_exit` som måste anropas för att stoppa exekveringen av en process.

`process_wait()`
: Denna funktion tar emot ett id för en skapad process och pausar exekvering av anropande tråd tills processen med det id-numret har avslutat. Om PintOS startas med flaggan -q är det av central betydelse att denna funktion fungerar korrekt (eller åtminstone hjälpligt) eftersom den används för att vänta på att första processen skall exekvera färdigt. Om den returnerar innan första processen avslutat kommer PintOS avslutas, och när flaggan -q används emulatorn att stängas av, med påföljd att de processer man tänkt sig köra kanske inte hinner bli klara eller ens starta.

`process_exit()`
: *Denna funktion är inte implementerad.* Eftersom varken `thread_exit` eller `process_cleanup` kan ta emot en `exit_status` (trådar har ingen exit-status) är det tänkt att denna funktion implementeras av dig för att göra det som behövs för att ta emot `exit_status` (uppgift 17) och avsluta processen (uppgift 13 med flera). Läs om `thread_exit` och övriga funktioner som beskrivs här innan du skriver någon kod.

`thread_create()`

Skapar en kernel-tråd som dediceras till att exekvera en viss funktion (i kernel-mode). Allt som har med processer att göra lämnas till funktionerna i filen `process.c`. Kan alltså användas för att skapa generella trådar, men skapar inga processer. Observera särskilt att denna funktion endast skapar tråden genom att allokera en stack, en trådstruktur och placera den i operativsystemets kö över trådar som ar redo att exekvera. Exakt när tråden börjar exekvera är odefinerat, och kan ske när som helst efter skapandet - ibland omedelbart, ibland efter lång tid.

`thread_current()`

Returnera en pekare till trådinformation (se `threads/thread.h`) om den tråd som exekverar just nu. *Denna behöver du använda ofta.* Det finns även funktioner som direkt tar fram specifika delar ur den struct som representerar en tråd, men via denna funktion kan du komma åt allt. *Titta vilken information som finns i threaad.h.*

`thread_exit()`

Anropas av en tråd som vill avsluta sig själv. Denna funktion returnerar inte, utan sätter en terminate-flagga och byter till nästa tråd, varpå nästa tråd raderar den avslutande tråden från systemet. (En tråd kan inte ta bort sig själv. Trådens data behövs när operativsystemed växlar till en ny tråd. Den kan inte såga av grenen den sitter på.)

Synkroniseringsmekanismer i PintOS

PintOS har ett antal synkroniseringsmekanismer tillgängliga. Dessa är deklarerade i `threads/synch.h` och definierade i `threads/synch.c`. Samtliga funktioner tar emot en pekare till en struct. Minne för structen måste i vanlig ordning vara skapat innan anrop, antingen genom att som argument ge adressen till en vanlig variabel (*rekommenderas*):

```
struct semaphore binary_semaphore_variable;
sema_init(&binary_semaphore_variable, 1);
```

eller genom att som argument ge en pekare till dynamiskt minne allokerat med `malloc` (måste alltid avallokeras med `free` när det inte behövs mer):

```
struct semaphore* binary_semaphore_pointer = malloc(...);
sema_init(binary_semaphore_pointer, 1);
...
free(binary_semaphore_pointer);
```

Här följer beskrivning av de viktigaste funktionerna som du kan behöva för att göra din kod tråd-säker. Samtliga av dessa är naturligtvis tråd-säkra själva, dvs. de kan anropas “samtidigt” från flera trådar utan problem.

`void sema_init (struct semaphore *sema, unsigned value);`

Används *alltid* för att initiera en nyligen deklarerad eller allokerad semafor *en gång*. Initierar semafor `sema` till att starta med `value` lediga resurser och en tom vänte-kö. Initiering till 1 ger en binär semafor som fungerar som ett lås, men utan felkontroller. Initiering till 0 ger en semafor där ingen resurs finns tillgänglig från början.

`void sema_down (struct semaphore *sema);`

Förbrukar *alltid* en resurs från `sema`. Finns ingen resurs tillgänglig så placeras anropande tråd på semaforens vänte-kö tills en resurs blir tillgänglig (tråden som anropar tar alltså paus). När funktionen returnerar är det *garanterat* att tråden fått tillgång till en resurs (att `value` plus antal exekverade `sema_up` minus antalet exekverade `sema_down` är positivt eller noll).

`void sema_up (struct semaphore *sema);`

Gör en *alltid* en resurs tillgänglig i `sema`. Om det finns någon tråd på `sema`'s vänte-kön kommer en av dessa att väckas och få tillgång till den nya resursen. Det *garanteras* att denna funktion aldrig kommer att vänta på något. Anropande tråd får fortsätta direkt.

`void lock_init (struct lock *lck);`

Initierar ett lås `lck` till att vara ledigt och med tom vänte-kö. Används *alltid* för att initiera ett nyligen deklarerat eller allokerat lås *en gång*.

`void lock_acquire (struct lock *lck);`

Om låset `lck` är ledigt markeras det som upptaget av anropande tråd. Om det är upptaget placeras anropande tråd på en vänte-kö tills låset blir ledigt. Det är *garanterat* att låset är markerat som upptaget av anropande tråd när funktionen returnerar. Det är *förbjudet* att anropa denna funktion när anropande tråd *själv* redan markerat låset som upptaget ("håller låset").

`void lock_release (struct lock *lck);`

Markerar låset `lck` som ledigt ("släpper låset") om exekverande tråd är den som håller låset. Om någon tråd finns på vänte-kön kommer den att väckas och få tillgång till låset. Det är *förbjudet* att anropa denna funktion om anropande tråd *inte* är den som håller låset.

`void cond_init (struct condition *cond);`

Initierar `cond` till att ha tom vänte-kö. Används *alltid* för att initiera ett nyligen deklarerat eller allokerat condition *en gång*.

`void cond_wait (struct condition *cond, struct lock *lck);`

Placerar *alltid* anropande tråd på `cond`'s vänte-kö och *släpper* sedan låset `lck`. Används för att vänta på att ett *externt* villkor skall uppfyllas "medan" anropande tråd hållet ett lås. Det är *garanterat* att anropande tråd inte blockerar låset *medan* den ligger på vänte-kön. Det är *garanterat* att anropande tråd "fortfarande" håller låset när funktionen returnerar. Det är **INTE** garanterat att det externa villkoret fortfarande är uppfyllt när funktionen returnerar. Det externa villkoret är något du testar för att avgöra om du behöver vänta på något. Det görs innan du anropar denna funktion.

`void cond_signal (struct condition *cond, struct lock *lck);`

Väcker en tråd från `cond`'s vänte-kö. Om vänte-kön är tom väcks *ingen* tråd. Anropande tråd skall ha gjort så resultatet av det externa villkoret förändrats innan anrop. Det är *förbjudet* att anropa denna funktion utan att hålla låset `lck`. (Låset används normalt för att skydda variabler som ingår i det externa villkoret.)

Hur process_execute och start_process skall fungera

Den initiala versionen av `process_execute` och `start_process` har flera problem och flera "hack" för att det ändå skall fungera hjälpligt. Följande beskriver hur implementationen *borde* (skall) fungera. Det är ganska nära hur det faktiskt fungerar, men inte riktigt.

Detta beskriver vad `process_execute` skall göra. Fetstilat är sådant som inte ännu sker:

1) En kommandorad tas emot som parameter
2) En tråd skapas (placeras på ready-kön) med uppgift att exekvera den nya processen.
   Som första argument anges ett namn på tråden, detta är första ordet på kommandoraden.
   Andra argument är trådens prioritet, `PRI_DEFAULT`.
   Tredje argument är en pekare till den funktion tråden skall exekvera.
   Fjärde argument är den pekare som skickas med när tråden startar funktionen i tredje argument.
   Fjärde argumentet är lämpligen en pekare till en struct som kan användas för att kommunicera olika data mellan föräldertråden och den nya tråden eller vice versa.
3) **Vänta på att tråden skall nå punkt e) nedan**. Tråden skall starta, läsa in processen (`load`), och göra den klar för start. Dessa tre steg kan gå galet på flera sätt, t.ex. att programfilen är korrupt eller att minnet tar slut. Resultatet hanteras i nästa punkt.
4) **Ta emot resultatet från nya tråden.** Om allt gick bra skall ett nummer som unikt identifierar processen returneras. **Gick något galet skall -1 returneras**. (-1 kan således inte användas för att identifiera en giltig process.)

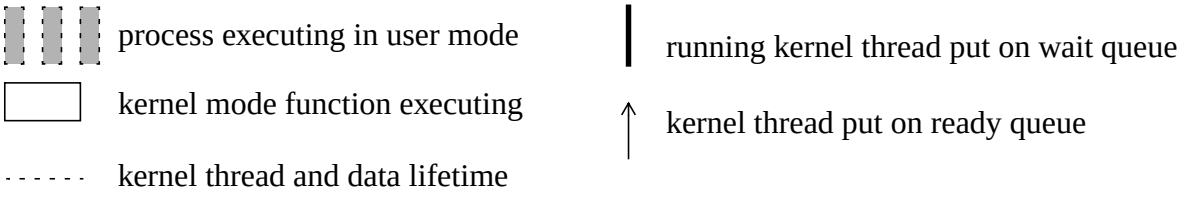
Detta beskriver vad `start_process` skall göra. Fetstilat är sådant som inte ännu sker korrekt. Under varje punkt visas relevant rad kod som referens (om det finns implementerat):

a) Ta emot pekare till data från `process_execute`
   `( struct process_arguments* pa = (struct process_arguments*)argument; )`
b) Extrahera namnet på programmet som skall köras
   `( strlcpy_first_word (file_name, pa->command_line, 64); )`
c) Läsa in programmet och allokera nödvändigt minne och stack
   `( success = load (file_name, &if_.eip, &if_.esp); )`
d) Placera kommandoraden på processens stack som parametrar till main
   `( HACK if_.esp -= 12; /* Unacceptable solution. */ )`
e) Skicka resultatet av punkter a) - d) (allt gick bra eller något gick galet) till punkt 4) ovan
f) Starta exekveringen av processen (läs kommentaren i koden, ej viktigt att förstå exakt)
   `( asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory"); )`

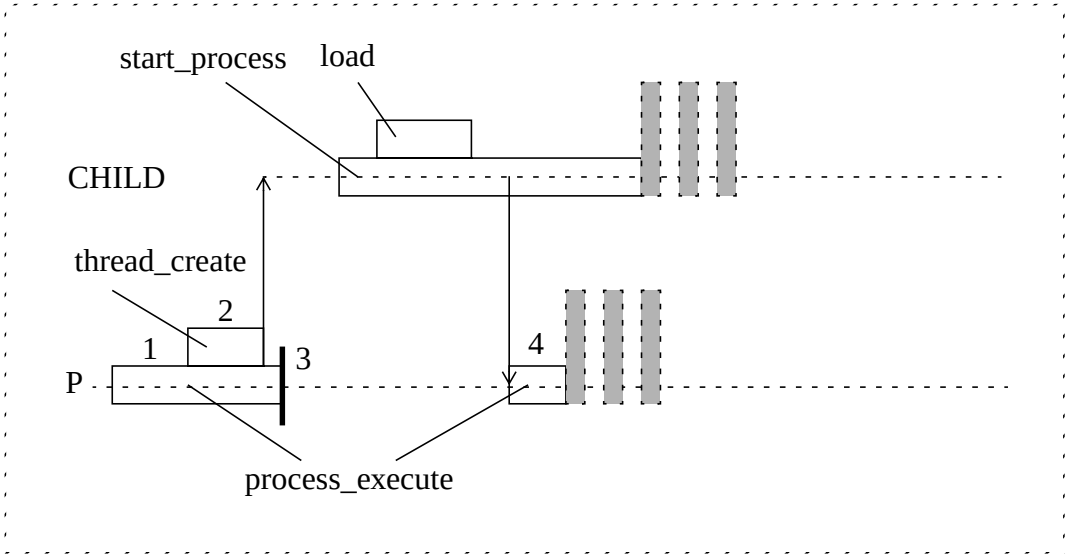
I `process_execute` utför nuvarande lösning inte hela punkt 3) och 4), istället stängs datorn av. I punkt 4) returneras just nu alltid att allt gick bra, men när något av stegen i 3) går galet skall det synas i punkt 4). Dessutom finns ett "race" eftersom kommandoraden kan hinna bli ogiltig innan `start_process` (som använder kommandoraden) exekverar (kommentera ut raden som gör `power_off` och se vad som händer). En korrekt lösning låter `process_execute` vänta *precis* tills `start_process` är klar med kommandoraden *och* har utfört punkt e).

I `start_process` utför originallösningen varken punkt d) eller punkt e). Punkt e) utförs inte alls. Och för punkt d) används istället en “hack” för att “låtsas” att den initiala stacken finns. Detta görs genom att helt enkelt subtrahera 12 från stackpekaren, vilket gör att det ser ut som om parametrar till main finns där. Fundera på varför just 12 byte måste användas. (Varför räcker det inte med 8 byte?)

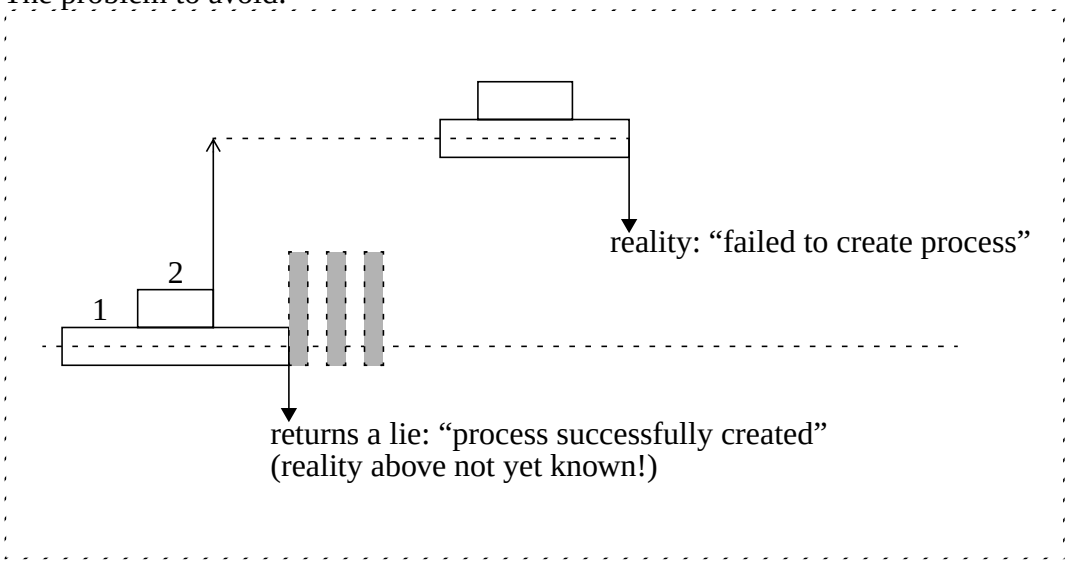
Figuren sammanfattar problemet grafiskt.



How it shall work:



The problem to avoid:



Uppgift

I ovan beskrivning framgår att punkter 3), 4) och e) inte fungerar korrekt i nuvarande kod. (Punkt d) löses i uppgift 11.) Implementera den synkronisering som behövs för att lösa dessa tre punkter korrekt och utan att använda “busy-wait”. När du är klar skall det inte finnas några spår av `power_off` raden. 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 synkroniserings-variabler 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.***

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’
```

När allt fungerar, be till slut assistenten kontrollera din lösning.