Deluppgift 17 Processhantering: exec, sleep, exit, plist

Inledning

För att få ett praktiskt användbart operativsystem behövs stöd för att starta flera program. Då behövs minst två systemanrop för att hantera processer. En process måste kunna avsluta, något du redan implementerat, och en process måste kunna starta nya processer.
Dessutom behöver operativsystemet ett sätt att hålla reda på aktiva processer och visa dem för användaren (jämför aktivitetshanteraren i Windows eller skalkommandot `top` i Solaris/Linux).

Lägga till ett systemanrop

För att lägga till ett systemanrop i Pintos behöver fyra filer modifieras.

1) Öppna `lib/syscall-nr.h` och lägg till ett symboliskt namn för anropet sist i enum-listan. Det tilldelas nu automatiskt ett nummer vid kompilering.
2) Öppna `lib/user/syscall.h` och lägg till en funktions-deklaration av systemanropet så som användarprogram skall kunna anropa det.
3) Öppna `lib/user/syscall.c` och lägg till en implementation av ovan deklaration genom att använda ett av de fördefinerade makron som finns beroende på antalet parametrar (se hur de andra anropen är gjorda).
4) Öppna `userprog/syscall.c` och lägg till kernel-sidans implementation av systemanropet (precis som förut).

Beskrivning av uppgiftens systemanrop

Systemanropen `exec` och `exit` är deklarerade i `lib/user/syscall.h` och används av användarprogrammen därefter:

`pid_t exec (const char *command_line);`
: Anropar endast `process_execute`. Men nu skall den nya processen läggas till i en lista över aktiva processer. Ett lämpligt process-id är nu processens index i listan, men det går att lösa på många sätt (valfritt). För att kunna hålla reda på relationen mellan processer behöver varje process hålla reda på sitt namn, sitt eget process-id, sin förälders process-id, samt eventuell extra data som en process behöver skicka tillbaka till sin förälderprocess eller tvärtom (jämför med vad du gjorde i uppgift 10).

`void sleep(int millis);`
: Anrop av denna funktion “pausar” processen i ett antal millisekunder. Funktioner för att låta en tråd “sova” i PintOS finns i `devices/timer.c`.

  Den befintliga implementationen av `timer_sleep` använder en implementation som hela tiden byter till en annan tråd. Detta klassas som “busy wait”. En bra implementation placerar processen i en väntekö under hela denna tid så processen inte förbrukar onödig datorkraft på att hela tiden kontrollera om det är dags att fortsätta. Det räcker att du implementerar systemanropet utan att rätta till “busy-wait”.

`void plist (void);`
: Skriver ut hela listan med processinformation (id, namn, förälder-id, exit-status) på ett snyggt och städat format. Denna funktion kan vara bra att ha i debug-syfte, så gör utskriften tydlig och komplett.

`void exit (int status) NO_RETURN;`

Ser till att processer som inte längre behövs i processlistan rensas bort så att nya processer får plats. Eventuellt allokerat minne måste frigöras. Detta implementeras i `process_cleanup` så att det utförs vid alla anrop som leder till att en tråd avslutas. Avslut kan ske av kernel när något allvarligt fel uppstår i en process, och inte bara av systemanrop.
För att förbereda för enkel delning av data mellan en process och dess förälder skall en process tas bort ur listan exakt när *både* processen *och* dess förälder har avslutat. En process kan alltså finnas kvar i listan ända tills även dess förälder avslutat (*OBS! även om barn till processen fortfarande exekverar kan processen tas bort ur listan. Processen delar endast sina data med sin förälder, ej med sina barn!*). Detta tillvägagångssätt garanterar att processens plats i listan alltid är giltigt oavsett om processen eller föräldern försöker använda den. Tänk noga igenom vilka processer som kan tas bort och när det skall göras.
Parametern `status` kan du som förut bara skriva ut, alternativt spara värdet med övrig information på processens plats i listan.

Förberedelse

Vid implementering 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 exekvera 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.** Om du planerar använda tråd-id som process-id (vilket inte behövs, det finns andra lösningar), tänk då på att `thread_tid` *alltid* ger tråd-id för den tråd som *anropar.* Tänk även på att du redan har ett sätt att kommunicera värden till och från den nya tråden sedan uppgift 10. 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 uppgift 10 för intentionen och ansvarsfördelningen mellan de olika

tråd- och processfunktionerna, 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 `userprog/process.h` används internt i `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 alternative 1-4 indikerade.

Uppgift

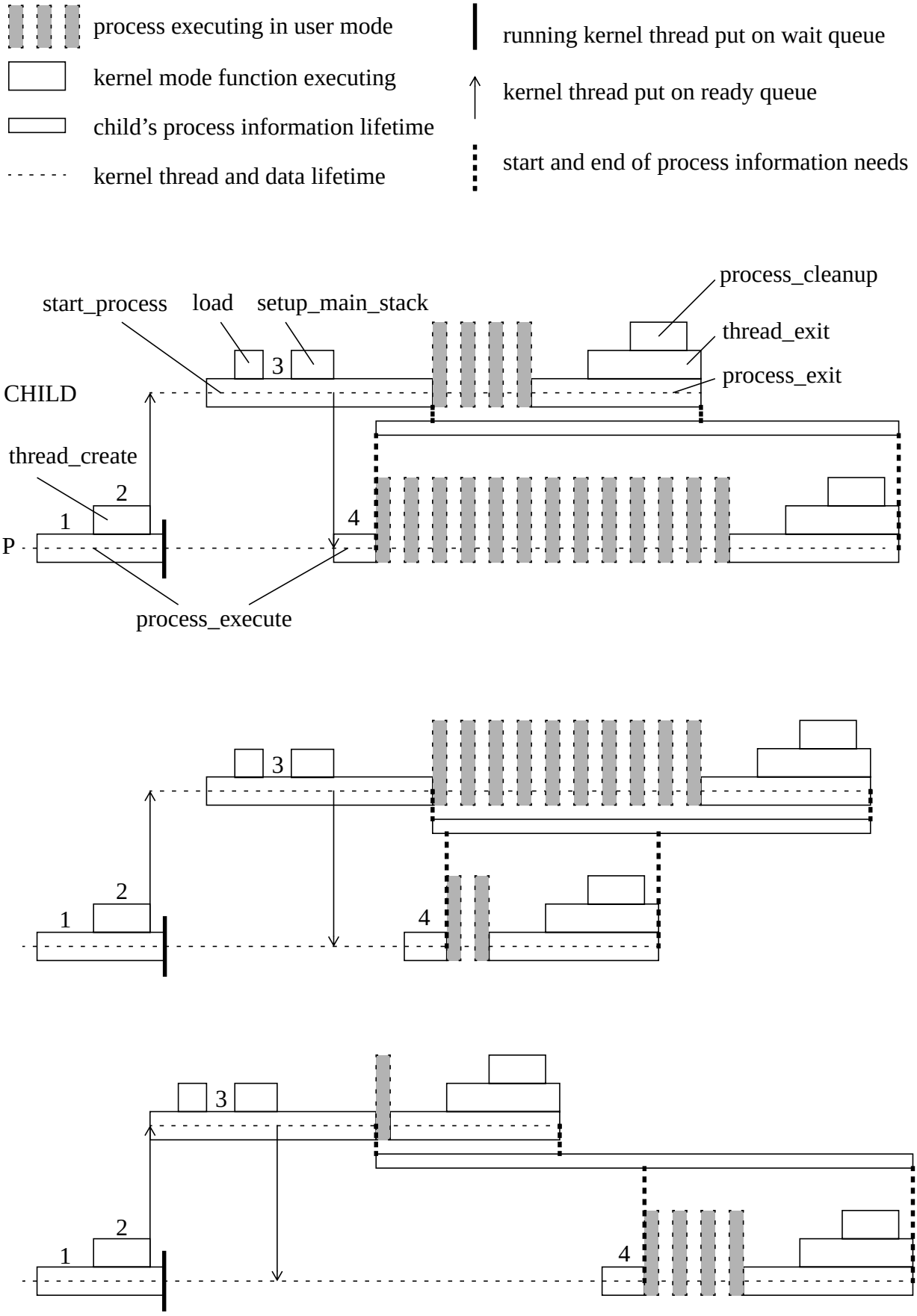
Redovisa för din handledare var du planerar att placera koden enligt förberedelsen. Att välja rätt plats här underlättar arbetet senare (du slipper göra om göra rätt). Lägg till systemanropen `plist` och `sleep` i PintOS och implementera sedan systemanropen `exec`, `exit` och `plist`.

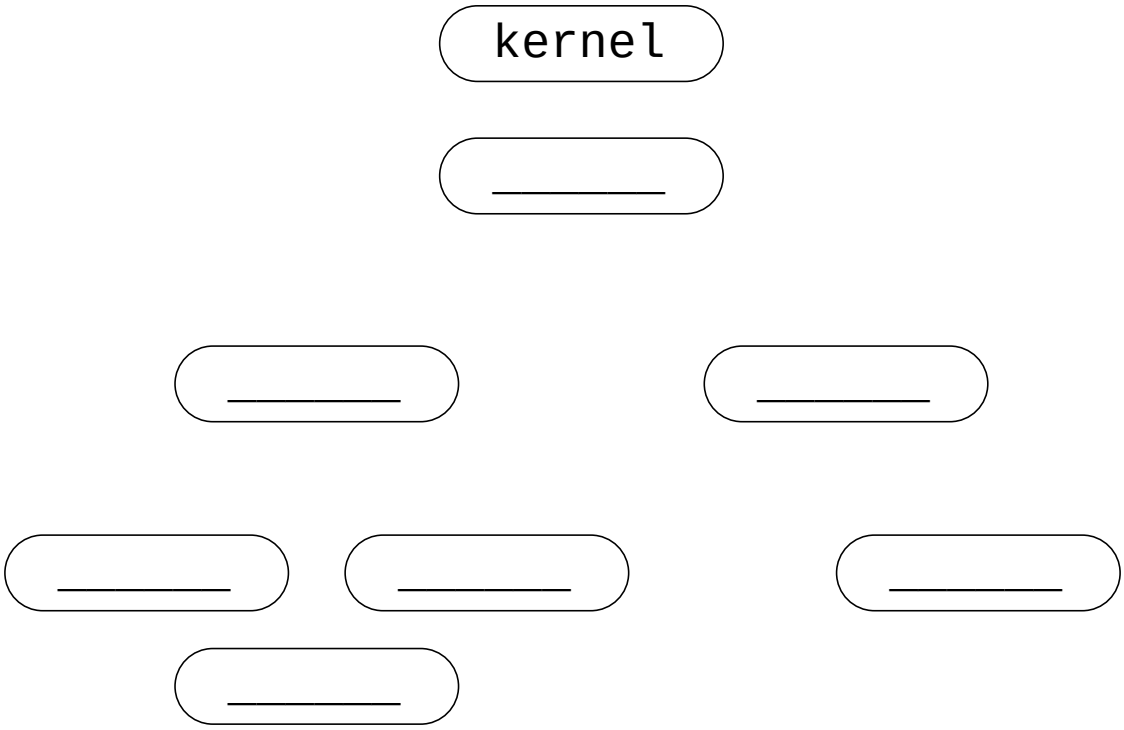
Tänk på att placera din kod i filerna `userprog/process.c` och `userprog/plist.c`. Koden i `userprog/syscall.c` skall endast innehålla ett anrop till dessa. Att du måste göra så beror på att den första processen inte startas via ett systemanrop, se informationen från uppgift 10.

I grafen på kommande sida ser du under vilken tidsperiod processinformationen skall finnas tillgänglig relativt både processen (`CHILD`) och dess förälder (`P`). Som du ser kan dessa data inte lagras i någon tråd då de kan behövas efter det att tråden ej längre finns. Du behöver naturligtvis synkronisera data som kan komma att användas samtidigt av en process och dess förälder.

Testa din implementation

Filen `examples/longrun_nowait.c` innehåller ett lämpligt testprogram för att testa systemanropen för processhantering. Se information i filen för att se hur det kan startas.


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
3
CHILD
thread_create
2
1
4
P
process_execute
3
2
1
4
3
2
1
4



free	proc_id	parent_id	exit_status	alive	parent_alive