Deluppgift 18 Processhantering: wait

Beskrivning av uppgiftens systemanrop

Systemanropet `wait` är deklarerat i `lib/user/syscall.h` och används av användar-programmen därefter:

```
int wait(pid_t id);
```

Tar emot ett process-id och kontrollerar att det är en barnprocess till processen som kör. Väntar därefter tills sagda barn-process har avslutat och returnerar dess `status` (se systemanropet `exit`). Avlutningsvis ska barn-processen tas bort från listan över aktiva processer eftersom den inte behövs mer. Det skall endast gå att vänta på en process en gång.

Uppgift

Implementera systemanropet `wait` 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`. Har du utfört uppgift 17 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 på att placera din kod i filen `userprog/process.c`. Koden i `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, se informationen från uppgift 10.

När du implementerat `wait` korrekt skall du kunna köra pintos med flaggan `-q` som gör att pintos avslutar när första processen är klar. Varför avslutar PintOS direkt om din `wait` inte fungerar?

Filerna `examples/longrun*.c` innehåller lämpliga testprogram för att testa systemanropen för processhantering. Se kommentarer i testprogrammen för att se hur de kan startas.

Testning

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/longrun`. När `wait` fungerar korrekt kan du även börja använda PintOS egna tester.

OBS! 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 till följa av något fel.) Den `printf` som behövs finns i `process_cleanup` men du måste *se till att variabeln status har rätt värde innan utskrivten sker.*

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

För att starta PintOS inbyggda tester använder du kommandot:

```
make -j8 check
```

Flaggan `-j8` är inte nödvändig, den instruerar endast make att köra nödvändiga kommandon i flera processer så att upp till 8 processorer kan nyttjas samtidigt. Detta är ju den fördel man får av att ha ett OS som tillåter att köra många processer samtidigt (multiprogramming). Du kan ju jämföra om det går fortare än att köra sekventiellt (utan `-j8`).

Ovan kommando kommer köra och utvärdera alla tester. Till slut får du se en testlog där det står `pass` eller `FAIL` för varje testprogram som kördes. Första gången kommer det att stå FAIL på de flesta tester. Det beror oftast på att du glömt kvar någon debug-utskrift som inte börjar med “# “, eller att `wait` inte alltid fungerar, eller att testet skickar in felaktiga parametrar till dina systemanrop. De testen heter i stil med *-bad-*, *-null och du kommer korrigera koden i senare uppgift. Det kan även uppstå synkroniseringsfel, som känns igen på att det inte alltid blir samma fel när du testar igen. Även det löser du i senare uppgift. Alla tester behöver alltså inte fungera direkt.

När du gjort ändringar i din kod och vill testa igen kör du bara ovan kommando en gång till. Koden kompileras om och alla tester körs igen automatiskt. Det spelar ingen roll om du står i `userprog` eller `userprog/build`. Eftersom du ändrat koden kan några av testerna som förut fick `pass` nu få `FAIL`. Därför är det lämpligt att köra alla tester igen för att upptäcka det direkt. Dock tar det tid. För att spara tid (när du är säker på att dina ändrignar inte förstör något fungerande test) finns möjligheten att köra endast de test som misslyckades igen:

```
make -j8 recheck
```

Det är viktigt att du ändrar i någon fil innan du kör testerna igen. Om alla filer är omodifierade kommer testen endast att visa föregående testresultat. Det märker du i regel på att testningen gick orimligt fort.

Utdata från varje testprogram sparas automatiskt på fyra olika filer. Antag att du står i katalogen `userprog/build` och följande rad står i testresultatet:

```
FAIL tests/userprog/halt
```

Utdata från testprogrammet finns nu i följande fyra filer:

`tests/userprog/halt.allput`
: Visar all utdata från körningen av PintOS och testprogrammet som skrevs ut på `stdout` (motsvarar `cout` i C++ program).

`tests/userprog/halt.output`
: Som ovan men utan de rader som börjar med “# “. Detta är indata till det program som kontrollerar om testet lyckades eller ej.

`tests/userprog/halt.errors`
: Här visas eventuella fel som skrevs ut på `stderr` (motsvarar `cerr` i C++ program).

`tests/userprog/halt.result`
: Här visas resultatet av kontrollen om testet lyckades eller ej. Rader som skrivs ut med plus (+) framför är rader som skrevs ut men var fel. Rader med minus (-) framför är rader som *inte* skrevs ut men *borde* ha skrivits ut för att testet skulle få godkänt. Logiken är: plus = “överflödig utdata” och minus = “saknad utdata”.