LINKÖPINGS UNIVERSITET

# TDIU16: Process- och operativsystemprogrammering

## Processhantering

Klas Arvidsson, Daniel Thorén, Filip Strömbäck

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2018

2020-03-06

# 1 Mål

Målet med den här laborationen är att lägga grunden till processhanteringen i Pintos. I slutändan vill vi att en process ska kunna starta en annan process (en barnprocess) och ta reda på om processen lyckades med sin uppgift eller ej genom att undersöka statuskoden som barnprocessen skickade till systemanropet `exit`. Exempelvis skulle det kunna se ut så här:

```
int main() {
  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 det.
  return 0;
}
```

I och med denna abstraktionen kan man se barnprocesser som ett lite mer komplicerat funktionsanrop. Föräldraprocessen startar en process och ger den parametrar. Barnprocessen gör sedan sitt och returnerar ett resultat.

I den här laborationen fokuserar vi på `exec` och `exit`. Det är dock bra att känna till det slutgiltiga målet redan nu, annars kommer det vi gör här kännas onödigt komplicerat. `wait` kommer i nästa laboration.

# 2 Bakgrund

Som vi kan se från exemplet ovan så ska `exec` returnera ett värde av typen `pid_t`, som är ett alias för `int`. Detta ID är oftast ett så kallat process-id, som är unikt för varje process i hela systemet. För att hålla koll på vilka process-id som finns, och för att en process senare ska kunna hämta resultatet från sina barnprocesser så måste vi hålla koll på alla processer i systemet. För detta ändamål kommer vi att bygga en processlista där vi lagrar information om alla processer. En stor del av laborationen går ut på att implementera processlistan och hålla den uppdaterad.

Du kan själv välja vilket process-id en process får, så länge det är positivt och unikt i hela systemet. Ett alternativ är att använda det tråd-id som varje tråd i Pintos redan får, ett annat är att generera heltal själv på något sätt (exempelvis via din `map`-implementation från ”Associativ container”). Det finns såklart andra alternativ också, men dessa är de vanligaste.

## 2.1 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 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 funktionen `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 uppgiften ”Den första processen”. 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 `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.

# 3 Uppgift

Implementera följande systemanrop:

- `pid_t exec(const char *command_line);`
- `void sleep(int millis);`
- `void plist(void);`
- `void exit(int status) NO_RETURN;`

För mer information om systemanropen se Pintos-Wiki under rubriken ”Tråd- och processhantering i Pintos”, se särskilt underrubriken ”Systemanrop”.

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. Tänk också på att föräldern till den första processen inte har startats via `process_execute`, eftersom den bara är en tråd, och därmed kanske inte finns i processlistan.

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.

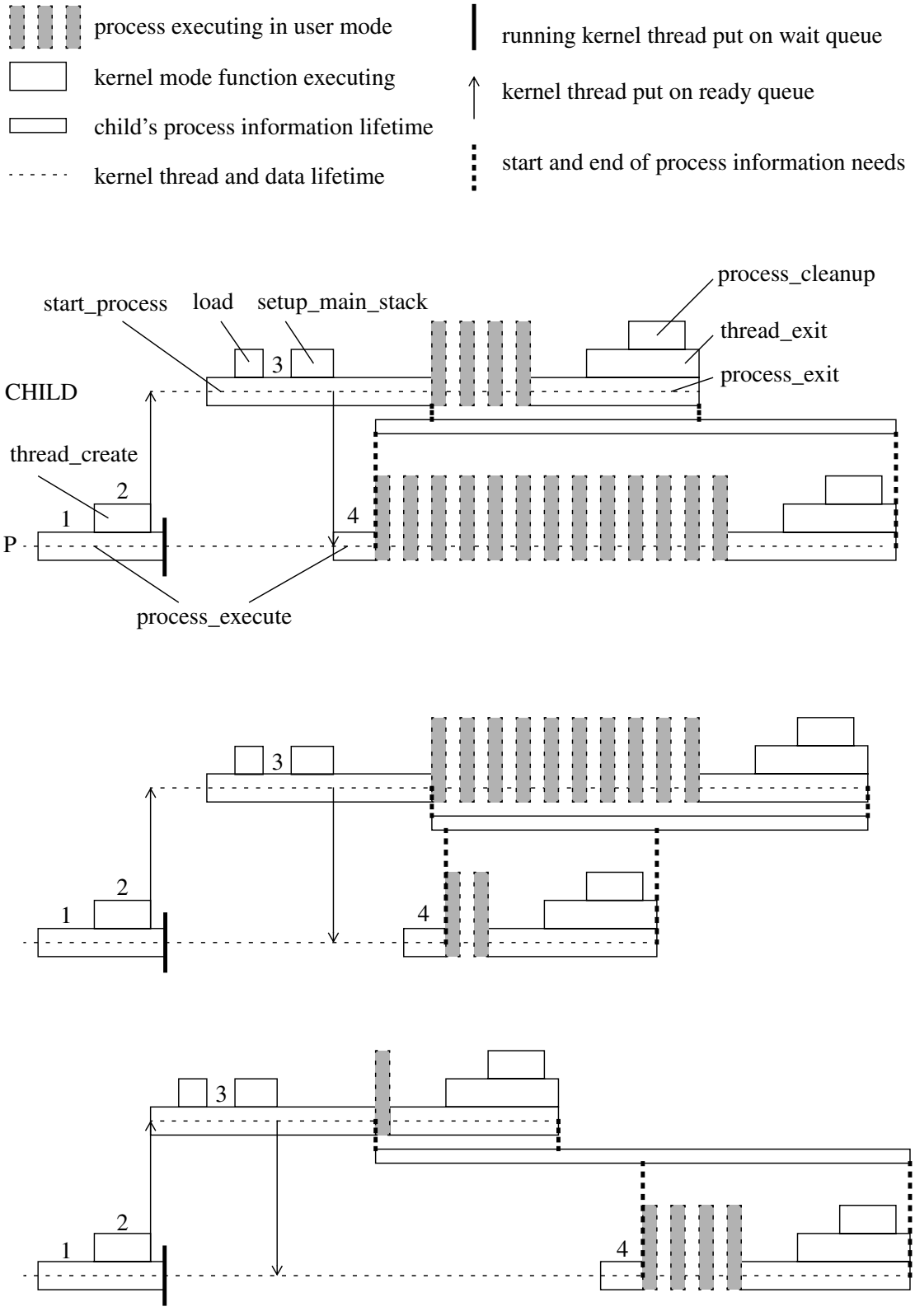
## 3.1 Testa din implementation

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.

Studera implementationen av `longrun_nowait.c` och se vad det gör. Lägg till ett anrop till systemanropet `plist` på lämpligt ställe i implementationen, kör programmet igen och studera hur din processlista ser ut (lägg tid på att göra utskriften snygg, det underlättar felsökningen otroligt), och fundera på om det stämmer överens med vad du borde se. I och med att vi ännu inte har implementerat `wait` går det inte att säga exakt vad som borde hända, och det är därmed inte möjligt att göra ett testprogram som garanterat säger om din implementation är korrekt eller ej i det här läget.

Du kan också testa programmet `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?


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
thread_create
P
process_execute
1
2
3
4