LINKÖPINGS UNIVERSITET

TDIU16: Process- och operativsystemprogrammering

Lab 2: Grundläggande systemanrop

Filip Strömbäck, Klas Arvidsson, Daniel Thorén

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2023

2023-03-14

Mål

I och med att alla program kör i usermode så har de inte möjlighet att komma åt hårdvaran. Detta är bra ur ett säkerhetsperspektiv eftersom olika processer inte kan förstöra för varandra. Däremot måste de ha begränsad åtkomst till hårdvaran för att göra något produktivt, exempelvis skriva ut resultatet av sina beräkningar på skärmen. Systemanrop löser detta genom att ge processer ett kontrollerat sätt att kommunicera med kernel. De kan därmed be kernel att prata med hårdvaran åt dem, och kernel kan se till att detta sker på ett säkert sätt. För mer information om systemanrop i Pintos se Pintos-Wiki under rubriken ”Systemanrop i Pintos”.

Målet med laborationen är att implementera följande systemanrop i Pintos:

- `halt` – stänger av den virtuella maskinen
- `exit` – avslutar processen som anropar exit
- `create` – skapar en fil
- `remove` – tar bort en fil
- `open` – öppnar en fil
- `close` – stänger en fil
- `read` – läser data från en fil eller terminalen
- `write` – skriver data till en fil eller terminalen
- `filesize` – hämtar storleken på en fil
- `seek` – ändrar filpositionen i en öppen fil
- `tell` – hämtar nuvarande filposition

Dessa implementeras i tre steg — vi börjar med de enkla systemanropen för att se hur systemanrop fungerar, sedan går vi vidare med lite mer komplexa systemanrop som kräver att vi använder den associativa databehållaren från tidigare.

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

Efter att du har implementerat alla delar av laborationen ska programmet `src/examples/file_syscall_tests.c` fungera felfritt. Del A och B har däremot mindre testprogram som är lämpliga att använda för att testa de delarna innan du har börjat på del C.

Del A `halt` och `exit`

Börja med att implementera systemanropen `halt` och `exit` genom att lägga till nödvändig kod i interrupthanteraren för systemanrop (`src/userprog/syscall.c`).

Implementationen av funktionerna `halt` och `exit` som anropas ifrån usermode finns i filerna `src/lib/user/syscall.h` och `src/lib/user/syscall.c`. Målet med uppgiften är alltså att skriva kod som tar emot anropen från usermode i kernel, och gör det som usermode frågar efter.

Du behöver i det här skedet inte bygga ny funktionalitet i kernel. Funktionaliteten för `halt` finns redan implementerad i funktionen `power_off`. För `exit` kan funktionen `thread_exit` användas. Däremot finns nu inget bra att göra med parametern som skickas till `exit`. I den här labben räcker det med att ni skriver ut den, så att ni kan kontrollera att implementationen fungerar. Vi kommer att använda den på ”rätt” sätt i senare labbar.

A.1 Testprogram

Du kan testa din implementation med programmet `src/examples/halt.c`. Se rubriken ”Att köra Pintos” i Pintos-Wiki för detaljer om hur du kör program i Pintos.

För att testa `exit` kan du exempelvis modifiera `halt.c` så att det anropar `exit` i stället. Alternativt kan programmet `src/examples/sumargv.c` användas.

Notera: Eftersom vi bara har implementerat `halt` och `exit` än så länge så har program i usermode ingen möjlighet att skriva ut saker till skärmen. Därmed fungerar inte heller funktioner som `printf` i usermode.

Del B In- och utmatning

För att processer ska kunna göra mer intressanta saker så behöver vi systemanrop för in- och utmatning: `read` och `write`.

Målet med denna del är därmed att implementera `read` för att läsa från tangentbordet, och `write` för att skriva till skärmen. Samma systemanrop kommer användas i del C för att läsa från och skriva till filer, men vänta med den delen för tillfället.

Om `read` anropas med `fd == STDIN_FILENO` ska `read` läsa från tangentbordet. Detta kan implementeras med hjälp av funktionen `input_getc` i filen `src/devices/input.h`. Här måste också tecknet ”carriage-return” (`\r`) ersättas med ett nyradstecken (`\n`).

Om `write` anropas med `fd == STDOUT_FILENO` ska `write` skriva ut till terminalen. Detta kan göras med exempelvis `putbuf` som är deklarerad i `lib/kernel/stdio.h` och implementerad i `lib/kernel/console.c`. Notera att `printf` **inte** är lämplig här, då datan som skickas till `write` inte nödvändigtvis är en nullterminerad sträng. Du vill också se till så att du skriver ut all data med **ett** anrop till `putbuf` för att undvika problem senare.

Systemanropens beteende för in- och utmatning finns beskrivet i ytterligare detalj under rubriken ”Read och Write” under ”Filsystem” i Pintos-Wiki.

B.1 Testprogram

Du kan testa din implementation med programmet `src/examples/line_echo.c`. Programmet läser en rad text från terminalen och skriver sedan ut den igen. Naturligtvis vill användaren kunna se de tecken som matas in på tangentbordet direkt på skärmen under tiden denne skriver.

Om du inte kör Pintos med flaggan `-v` och därmed ser skärmen från den virtuella maskinen, var noggran med att skriva indata i terminalen. Den virtuella maskinen har fel tangentbordslayout, vilket kan leda till oväntade resultat. Utdata ska visas i **bägge** fönstren.

Del C Filhantering

Till sist är det dags att implementera resterande systemanrop för att hantera filer. Eftersom många av dessa systemanrop arbetar med så kallade *fildeskriptorer* (ofta förkortat `fd`) behöver du introducera ett sätt att hålla koll på dessa. Tänk på att en process endast ska kunna läsa från, skriva till, och stänga filer som den har öppnat. Det är alltså lämpligt att lagra listan över fildeskriptorer i en datastruktur som hör ihop med processen (tips: en process har exakt en tråd i Pintos). Tänk också på att varje öppnad fil behöver stängas exakt en gång, även om processen glömmer bort att anropa `close`, eller om `close` anropas fler än en gång för en viss fildeskriptor.

Stora delar av filsystemet finns redan implementerat i Pintos. Målet med laborationen är alltså att låta usermode-program anropa funktionaliteten som redan finns på ett säkert sätt. Du behöver alltså inte fundera på hur filsystemet är implementerat i detalj i den här laborationen.

Planera din implementation noga. Skriv funktioner för kod du behöver upprepa mer än en gång (exempelvis kontroll att en `fd` är giltig). Du kan behöva modifiera implementationen i del A och B, och även befintlig kod. För att hålla koll på öppna filer, och vilken `fd` som motsvarar en viss fil (`struct file *` i Pintos) är det lämpligt att använda din kod från uppgiften "Associativ container" i förra labben. Lägg implementationen i filerna `src/userprog/flist.h` och `src/userprog/flist.c` och anpassa koden utefter vad som behövs för att lagra öppna filer. Fundera också på vad som händer ifall något anrop misslyckas. Eftersom du skriver kod i kernel så vill du vara extra noga med att hantera fel, och se till så att resurser (minne, öppna filer, ...) alltid återlämnas korrekt, även om ett fel skulle inträffa.

För denna del finns användbar information på följande ställen i Pintos-Wiki:

- "Interna funktioner" i avsnittet "Tråd- och processhantering i Pintos" beskriver bland annat `process_cleanup` som kan vara intressant.
- "Filsystem" beskriver hur filsystemet ser ut och fungerar i allmänhet, men också de systemanrop som ska implementeras.

Titta även på funktionerna i `src/filesys/filesys.h`, samt i `src/filesys/file.h`.

C.1 Testprogram

När du har implementerat resterande systemanrop ska programmet `src/examples/file_syscall_tests.c` fungera felfritt (meddelandet "Main thread about to poweroff..." kommer dock finnas kvar, det löses i kommande laboration).

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. Innan du har gjort nästa labb finns en minnesläcka ifrån `process_execute`, men utöver det borde inga flera minnesläckor finnas om din implementation är korrekt.

Beskrivning av systemanropen

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 "Filsystem" i Pintos-Wiki.

- `void halt(void) NO_RETURN;`

  Stänger av *datorn* (emulatorn) med omedelbar effekt. Processen som anropade `halt` kommer inte fortsätta exekevera eftersom datorn stängs av (därav är funktionen markerad med `NO_RETURN`).

- `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`).

  `status` indikerar oftast om processen lyckades med sitt arbete eller inte. Om allt gick bra anropas `exit` med `status` satt till 0. Annars är `status` satt till en programspecifik felkod. Värdet på `status` kommer i en senare laboration göras tillgängligt till den som startade processen. I det här skedet räcker det med att `status` skrivs ut så att vi kan se så att allt fungerar som det ska.

I C- och C++-program anropas sällan `exit` uttryckligen. I stället returnerar man `status` från `main`, och låter kompilatorn anropa `exit` i stället (i Pintos finns detta beteende implementerat i filen `src/lib/user/entry.c`).

- `bool create(const char *name, unsigned size);`

  Skapar en fil i filsystemet med namn `name` (en C-sträng). I Pintos har alla filer en fast storlek som måste anges när man skapar filen. Efter att en fil har skapats så går det alltså inte att ändra storlek på filen (till skillnad från de flesta andra operativsystem). Funktionen ska returnera `true` om det gick att skapa filen, och `false` om något gick fel (exempelvis om disken blev full).

- `bool remove(const char *name);`

  Tar bort filen `name` (en C-sträng) ur filsystemet. Returnerar `true` om det gick bra, och `false` om något gick fel (exempelvis om filen inte fanns).

- `int open(const char *name);`

  Öppnar filen `name` (en C-sträng). Om det gick att öppna filen ska den öppna filen associeras med en fildeskriptor som inte redan används. Fildeskriptorn ska sedan returneras. Om det av någon anledning inte gick att öppna filen ska -1 returneras.

  Notera: Implementationen av `open` kan välja vilket positivt heltal som helst för filen som nyligen öppnades. Det enda kravet är att samma id (fildeskriptor) inte redan används i samma process.

- `void close(int fd);`

  Stänger en fil som tidigare har öppnats av `open`. Efter detta så får nya anrop till `open` återanvända fildeskriptorn `fd` (alltså: om ett program stänger fd nummer 3, och sedan öppnar en ny fil så får `open` återanvända fd nummer 3 till detta, men måste inte göra det).

- `int read(int fd, void *buffer, unsigned length);`

  Läser data från tangentbordet (om `fd == STDIN_FILENO`) eller en fil. Systemanropet ska skriva upp till `length` bytes till minnet som pekaren `buffer` pekar på. Anropet ska sedan returnera antalet bytes som skrevs till buffern. Om data läses från tangentbordet så kommer buffern alltid bli full (i och med att vi väntar på att exakt `length` bytes har matats in). Om data läses från en fil så kan det returnerade värdet vara lägre än `length` ifall vi kom till slutet av filen innan buffern blev full.

  Om `fd` inte är giltig ska -1 returneras.

- `int write(int fd, const void *buffer, unsigned length);`

  Skriver data till skärmen (om `fd == STDOUT_FILENO`) eller en fil. Systemanropet ska läsa upp till `length` bytes från minnet som `buffer` pekar på. Likt `read` så returnerar `write` antal bytes som skrevs till filen eller skärmen. Skrivs data till skärmen så kommer returvärdet alltid vara lika med `length`. Skrivs data till en fil så kan det vara lägre ifall vi kom till slutet av filen (och eftersom filer inte kan ändra storlek i Pintos).

  Om `fd` inte är giltig ska -1 returneras.

- `int filesize(int fd);`

  Returnera storleken av den öppnade filen `fd`. Om `fd` inte är giltig ska -1 returneras.

- `void seek(int fd, unsigned position);`

  Flytta filpositionen i filen `fd` till `position`. Det innebär att nästa byte som läses eller skrivs till/från filen kommer att vara `position`. Exempelvis kan vi hoppa till början av filen genom att anropa `seek(fd, 0)`.

- `unsigned tell(fd);`

  Hämta vilken position i filen `fd` som kommer att läsas härnäst. Det är okej att du returnerar -1 ifall `fd` är oiltig, trots att resultate är `unsigned`.