Brugerprocesser i Buenos
OSM Anden opgave

Davy Eskildsen og Mads Lund

February 27, 2012

I denne opgave har vi implementeret håndtering af brugerprocesser i Buenos og nogle systemkald, så brugerprogrammer kan interagere med kernen i forhold til processerne.

Brugerprocesser

Brugerprocesserne og de tilhørende kernefunktioner er implementeret i filerne `proc/process.c` og `proc/process.h`.

En proces bliver representeret med datatypen `process_t` som indeholder en tilstand, adressen til et program og en returværdi. I figur 1 ses deklarationen fra `proc/process.h`.

```
1  typedef int process_id_t;
2
3  typedef enum {
4    PROC_RUNNING,
5    PROC_ZOMBIE,
6    PROC_FREE
7  } proc_state_t;
8
9  typedef struct {
10   proc_state_t state;
11   const char *executable;
12   int retval;
13   process_id_t first_child;
```

Figure 1: Typedelklerationer i `proc/process.h`

I `proc/process.c` er defineret en tabel som skal indeholde alle kørende processer. Til det har vi brugt et array af `process_t` med størrelsen `USER_PROC_LIMIT`. Det fungerer således som det maksimalie antal processer, som kan køres. Vi har sat `USER_PROC_LIMIT` = 64. Når tabellen bliver initialiseret sættes alle indgangenes tilstand til `PROC_FREE`. Dette gøres med funktionen `process_init`.

Da denne tabel er en ressource som kan tilgåes af flere processer, definere vi en spinlock, `proc_table_slock`.

Til håndtering af processerne har vi implementeret funktionerne: `process_spawn`, `process_run`, `process_finish` og `process_join`. Desuden har vi ændret funktionen `process_start` så den tager id'et på en proces i stedet for adressen til et program. `process_start` henter så adressen fra procestabellen i stedet. Da `process_start` skal kaldes af `thread_create` er nødvendigt at argumentet er af typen `uint32` og bliver så castet derfra.

`process_spawn`

Denne funktion kører et givent program som en process i en ny tråd. Først gennemsøges procestabellen efter en ledig plads (tilstanden er `PROC_FREE`). Der er ikke Implementeret nogen fejlhåndtering i tilfælde af, at der ikke er nogen ledig plads. I stedet dræbes systemet. Herefter initialiseret processen i tabellen ved at sætte tilstanden til `PROC_RUNNING` og adressen til programmet. Processen startes med funktionen `process_start` som køres i en ny tråd ved hjælp af `thread_create`.

`process_run`

Denne funktion kører et givent program som en process i den aktuelle. Tilsvarende `process_spawn` findes fri plads i procestabellen som initialiseres. Til sidst kaldes `process_start`.

`process_finish`

Denne funktion afslutter et proces. Den sørger for at få fjerne døde børn og registre levnde børn forældreløse. Hvis hvus en proces er forældreløse blive den sat "begravet" ellers bliver den til en zombi der venter på sin forælder kommer og begraver den.

`process_join`

Denne funktion venter på at en given proces slutter of returnerer dennes returværdi. Hvis processen ikke er færdig (tilstanden er `PROC_ZOMBIE`) så bruger vi en "sleep queue" til at vente på at tilstanden ændres. Hertil har vi fulgt eksemplet fra kapitlet om "Sleep Queue" i Roadmap to Buenos side 29. Når processen er færdig er dens returværdig gemt i procestabellen så vi kan hente der derfra. Inden vi returnerer frigives indgangen i tabellen.

Systemkald

Vi har implementeret systemkaldene `SYSCALL_EXEC`, `SYSCALL_EXIT`, `SYSCALL_JOIN`, `SYSCALL_READ` og `SYSCALL_WRITE` i funktionen `syscall_handle` i filen `proc/syscall.c`. De bliver kaldt ved hjælp af funktionerne beskrevet i haeder-filen `tests/lib.h`.

For de systemkald som tager argumenter, parses argumenterne fra MIPS-registrene i "user space" som argumenter til de relevante kerneoperationer. Tilsvarende gemmes returværdier fra kernoperationer i MIPS-registret for returværdier. `SYSCALL_EXEC`, `SYSCALL_EXIT` og `SYSCALL_JOIN` kalder stortset de tilsvarende kerneoperationer. Dem vil vi derfor ikke gennemgå yderligere her. `SYSCALL_READ` og `SYSCALL_WRITE` er implementeret ud fra testkonsolen fra i `init/main.c`.

Fejl på afleveringstidspunket

Vi har på afleveringstidspunktet en fejl der gør at vores kerne ikke kan lave nye processer. Vi havde fået den fjernet men den kom igen lige før afleveringen og vi havde ikke tid nok til at kunne finde og fjerne den igen.