Godkendelsesopgave 2 i "Styresystemer og multiprogrammering"

Generelt

Denne ugeopgave stilles fredag den 11. februar 2011 og skal afleveres senest mandag den 22. februar 2011 klokken 23:59:59. Den kan løses af grupper på op til 2 personer (3 kan tillades, men frarådes). Besvarelsen af opgaven vil resultere i enten 0, 1/2 eller 1 point. Pointene uddeles efter følgende retningslinjer:

- 0 point: besvarelsen har flere store mangler.
- 1/2 point: besvarelsen opfylder i store træk kravene, men har flere mindre mangler.
- 1 point: besvarelsen opfylder kravene til opgaven med kun få eller ingen mangler.

Det er en betingelse for at gå til eksamen på kurset at man har opnået mindst 4 point i alt.

Besvarelsen skal indleveres elektronisk via kursushjemmesiden (Absalon). Besvarelsen bør ske ved aflevering af en enkelt fil. Brug 'zip' eller 'tar.gz' til at samle flere filer. Opgavenummer og navne på gruppemedlemmer skal fremgå tydelig af første side i besvarelsen. Når I indleverer bør efternavne på alle gruppemedlemmer indgå i filnavnet - desuden skal opgavenummer fremgå af navnet:

```
efternavn1-efternavn2-efternavn3-G1.<endelse>
```

Jeres aflevering skal være i et format der kan læses på DIKUs systemer uden problemer (således bør formatet f.eks. ikke være MS Word dokumenter). Se i øvrigt "Krav til G-opgaver" siden på kursushjemmesiden under "G-opgaver" menupunktet.

I skal kun aflevere én fuld besvarelse per gruppe. Personen, som afleverer skal markere sine gruppemedlemmer via Absalon.

Om dokumentation af jeres løsning

Afleveringen skal indeholde en kortfattet rapport der dokumenterer hvilke vigtige observationer I har gjort jer, antagelser jeres løsninger afhænger af (som I enten har valgt, eller som er valgt af designerne af BUENOS) og desuden skal I redegøre kortfattet for design-beslutninger i kode I har implementeret. I skal også huske at kommentere jeres kildetekst så den er let at forstå. Især er det vigtigt at dokumentere hvis I har foretaget ændringer i allerede implementerede dele af BUENOS.

1 Denne uges tema: BUENOS

Denne og de følgende G-opgaver tager udgangspunkt i en simpel kerne kaldet BUENOS, som kører på en virtuel maskine kaldet YAMS. YAMS simulerer en MIPS32 arkitektur. En del "standard funktionalitet" er allerede implementeret i BUENOS og andre dele vil I blive bedt om at implementere. I denne G-opgave er det overordnede formål at introducere jer til BUENOS kernen. Det anbefales at læse guiden til BUENOS(tilgængelig på Absalon under 'Undervisningsmateriale'). I særdeleshed er kapitel 6 relevant for denne G-opgave.

G2.1: Implementering af proces abstraktion

Opgave 1 Den nuværende udgave af BUENOS har et begreb om kerne-tråde defineret i filen `kernel/thread.h` med typen `thread_table_t`, men ikke noget egentlig begreb om bruger-processer. Første del af G2 er derfor:

- Definér en datastruktur til at repræsentere bruger-processer (proces abstraktion). Datastrukturen skal som minimum indeholde navnet på den tilknyttede brugerprocess. I skal foretager jeres ændringer i filerne `proc/process.h` og `proc/process.c`.

  I får formentlig brug for flere oplysninger i jeres proces abstraktion for at kunne løse den næste opgave.

- Kernen skal kunne holde styr på alle bruger-processer og derfor kan det være at fordelagtigt at gemme alle proces abstraktioner for ikke-døde processer i en tabel eller lignende. Det er *tilladt* at have en statisk bestemt øvre grænse for hvor mange bruger-processer, der kan håndteres af gangen.

- I skal sikre at tilgang til jeres proces abstraktioner foregår atomisk. D.v.s. hvis én kernetråd læser fra stukturen, må en anden ikke kunne skrive samme sted.

Det anbefales at man som en del af denne opgave implementerer nogle funktioner til at håndtere processer. F.eks. er mindst følgnde funktioner nyttige at implementere:

```
/* Run process in new thread, returns PID of new process. */
process_id_t process_spawn(const char *executable);

/* Run process in this thread, only returns if there is an error. */
int process_run(const char *executable);

process_id_t process_get_current_process(void);

/* Stop the current process and the kernel thread in which it runs. */
void process_finish(int retval);


/* Wait for the given process to terminate, returning its return
   value, and marking the process table entry as free. */
uint32_t process_join(process_id_t pid);

/* Initialize process table. Should be called before any other
   process related calls. */
void process_init(void);
```

Det være en hjælp at starte med at se på `init/main.c` for at se hvordan det første program (`initprog`) startes i `init_startup_thread`.

G2.2: Implementering af udvalgte systemkald

Systemkald er en måde hvorpå brugerprogrammer kan kalde kernefunktioner. I det følgende afsnit beskrives kort hvordan systemkald er implementeret i BUENOS. Se også afsnit 6.3 og 6.4 i guiden til BUENOS.

Systemkald i BUENOS Systemkald i BUENOS er implementeret ved brug af MIPS instruktionen `syscall` under antagelse af at register `a0` indeholder nummeret på den kernefunktion, som skal kaldes og registrene `a1`, `a2` og `a3` indeholder argumenter til kernefunktionen. Effekten af at udføre `syscall`-instruktionen er at der generes en *exception* (eller trap, som det også kaldes i OS sammenhænge), interrupts slåes fra og der hoppes til en forudbestemt adresse som gemmer den nuværende "context". Dernæst hoppes til en (C) funktion, som håndterer systemkald. Funktionen, som håndter systemkald i BUENOS hedder `syscall_handle` og er defineret i filen `proc/syscall.c`. Funktionen er i udgangspunktet defineret således:

```
/**
 * Handle system calls. Interrupts are enabled when this function is
 * called.
 *
 * @param user_context The userland context (CPU registers as they
 * where when system call instruction was called in userland)
 */
void syscall_handle(context_t *user_context)
{
    /* When a syscall is executed in userland, register a0 contains
     * the number of the syscall. Registers a1, a2 and a3 contain the
     * arguments of the syscall. The userland code expects that after
     * returning from the syscall instruction the return value of the
     * syscall is found in register v0. Before entering this function
     * the userland context has been saved to user_context and after
     * returning from this function the userland context will be
     * restored from user_context.
     */
    switch(user_context->cpu_regs[MIPS_REGISTER_A0]) {
    case SYSCALL_HALT:
        halt_kernel();
        break;
    default:
        KERNEL_PANIC("Unhandled system call\n");
    }

    /* Move to next instruction after system call */
    user_context->pc += 4;
}
```

Når `syscall_handle` returnerer bliver returværdien fra kernefunktionen placeret i register `v0` og der returneres til brugerprogrammet. Funktionen `_syscall` (som er defineret i MIPS

assembler i filen `tests/lib.c`) fungerer som en wrapper til MIPS `syscall` instruktionen og kan kaldes fra C-kode. Dette benyttes i filen `tests/lib.c` benyttes `_syscall` til at definere et "systemkalds-bibliotek".

Opgave 2 Den anden del-opgave af G2 går ud på udfylde `syscall_handle` sådan at den også håndterer følgende systemkald (se afsnit 6.4 af BUENOS guiden for beskrivelse af hvad systemkaldene skal gøre):

- **int** syscall_read(**int** fhandle, **void** *buffer, **int** length)
- **int** syscall_write(**int** fhandle, **const** **void** *buffer, **int** lenght)
- **void** syscall_exit(**int** retval)
- **int** syscall_exec(**const** **char** *filename)
- **int** syscall_join(**int** pid)

For at gøre opgaven mere overkommelig har vi valgt at indføre nogle simplifikationer.

Generel begrænsning Det er *ikke nødvendigt* at bekymre sig om at gøre de implementerede funktioner "skudsikre" (kaldes "bulletproof" i guiden til BUENOS). At de implementerede systemkald er skudsikre betyder at den eneste måde et brugerprogram kan få kernen til at stoppe er ved halt systemkaldet og der er ingen mulighed for at to brugerprocesser læser hinandens data eller på anden vis kommer "i vejen" for hinanden.

syscall_read: Det må kun være tilladt at læse fra konsollen/`stdin`. D.v.s. jeres implementation af syscall_read behøver kun håndtere det tilfælde at fhandle == FILEHANDLE_STDIN (som defineres i `proc/syscall.h`).

syscall_write: Det må kun være tilladt at skrive til konsollen/`stdout`. D.v.s. jeres implementation af syscall_write behøver kun håndtere det tilfælde at fhandle == FILEHANDLE_STDOUT (som defineres i `proc/syscall.h`).

Når I implementerer read og write kan det være en fordel at kigge på hvordan startop-koden i `init/main.c` skriver til konsollen: se f.eks. init_startup_fallback.

syscall_exit: Hvis this_thread er den kernetråd, som håndterer den brugerprocess, der skal afsluttes, da skal man have følgende stump kode før et kald til thread_finish:

```
vm_destroy_pagetable(this_thread->pagetable);
this_thread->pagetable = NULL;
```

Sidste linje er nødvendig fordi thread_finish undersøger om feltet pagetable er NULL for den (kerne)tråd, der skal afsluttes. I fald feltet ikke er NULL, kaldes _kernel_panic. Den første linje er nødvendig for at frigøre det lager, som var reserveret til pagetable. Feltet pagetable er relateret til virtuel hukommelse og er således først relevant senere i kurset.

syscall_exec: Hvis I har implementeret et lille bibliotek af process-håndteringsfunktioner i opgave 1, burde dette systemkald være nemt at implementere.

syscall_join : Et kald til syscall_join(pid) skal vente på at en process identificeret ved pid afsluttes. Resultatet af afslutning af process pid (som angives med syscall_exit) skal være resultatet af kaldet til syscall_join. Det er vigtigt at bemærke at en process også skal kunne vente på afslutning af en process, som allerede er afsluttet.