

NAMA: Abu zaki

NIM: 240202892

KELAS: 2IKRB

Hasil modul 1

Tujuan utama modul ini adalah:

- a. `getpinfo(struct pinfo *ptable)`: System call ini dirancang untuk mengambil informasi tentang proses-proses aktif yang sedang berjalan di sistem. Informasi yang dikembalikan meliputi PID (Process ID), ukuran memori yang digunakan, dan nama dari setiap proses. Ini memungkinkan program user-level untuk mendapatkan "gambaran" tentang kondisi proses di sistem.
- b. `getReadCount()`: System call ini bertujuan untuk melacak dan mengembalikan total jumlah pemanggilan fungsi `read()` yang telah terjadi sejak sistem xv6 pertama kali di-boot. Ini adalah contoh sederhana dari "instrumentasi kernel", di mana Anda menambahkan kode ke dalam kernel untuk memonitor aktivitas tertentu.

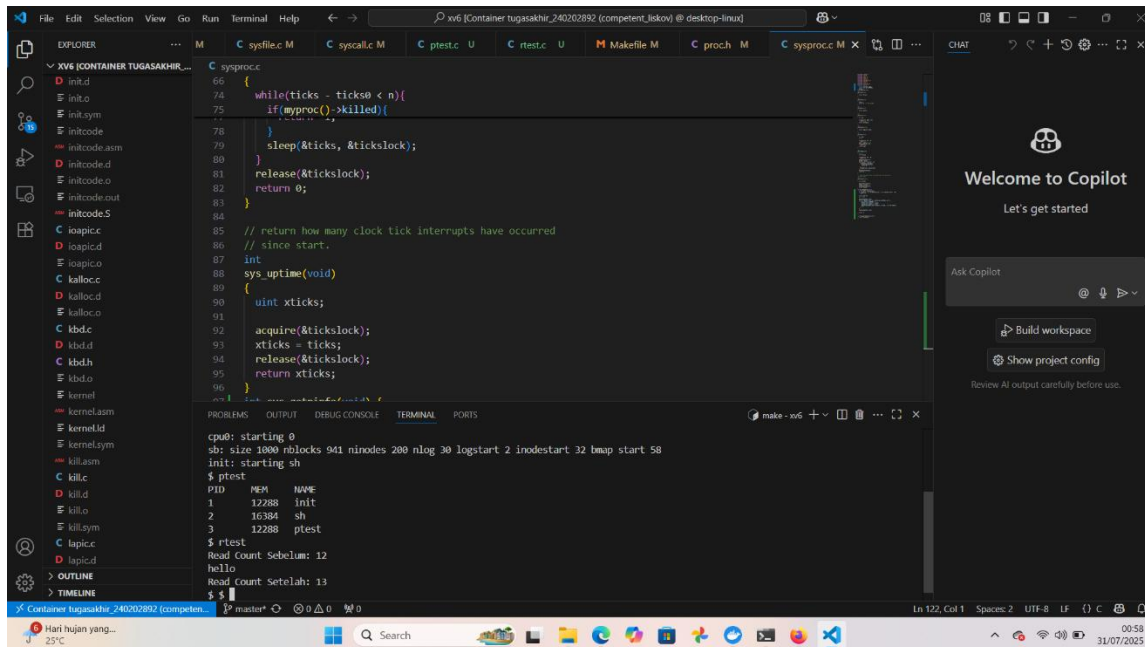
Rincian implementasi:

1. Definisi Struktur Data Baru: Membuat struktur `pinfo` di `proc.h` untuk menyimpan informasi proses, dan variabel global `readcount` di `sysproc.c` untuk menghitung panggilan `read`.
2. Pendaftaran System Call Baru: Menentukan nomor unik untuk setiap system call baru di `syscall.h` dan mendaftarkannya dalam tabel system call di `syscall.c`.
3. Antarmuka User-Level: Mendeklarasikan fungsi system call di `user.h` dan membuat entri di `usys.S` agar program user-level dapat memanggilnya.
4. Implementasi Fungsi Kernel: Menulis kode aktual untuk `sys_getpinfo` dan `sys_getreadcount` di `sysproc.c`, termasuk penanganan argumen dan akses ke struktur data kernel (seperti tabel proses).
5. Instrumentasi `read()`: Menambahkan `readcount++` di fungsi `sys_read` di `sysfile.c` untuk menginkrementasi penghitung setiap kali `read()` dipanggil.
6. Program Penguji (User-Level): Membuat program `pctest.c` dan `rctest.c` untuk menguji fungsionalitas system call yang baru dibuat dari sisi user-level, serta mendaftarkannya di `Makefile`.
7. Build dan Jalankan: Instruksi untuk mengkompilasi ulang xv6 dan menjalankan program penguji.

Pengujian:

- a. `Ptest`
- b. `Rtest`

Hasil uji



```
File Edit Selection View Go Run Terminal Help
xv6 [Container tugasakhir_240202892 (competent_tokov) @ desktop-linux]

EXPLORER
xv6 [CONTAINER TUGASAKHIR_...
  D init.d
  F init.o
  F init.sym
  F initcode
  F initcode.asm
  D initcode.d
  F initcode.o
  F initcode.out
  F initcode.S
  C ioapic.c
  D ioapic.d
  F ioapic.o
  C kalloc.c
  D kalloc.d
  F kalloc.o
  F kalloc.S
  C kbd.c
  D kbd.d
  F kbd.o
  F kbd.S
  C kernel
  F kernel.asm
  F kernel.d
  F kernel.sym
  F kill.asm
  C kill.c
  D kill.d
  F kill.o
  F kill.sym
  C lapic.c
  D lapic.d
  F lapic.o
  F lapic.S
  F OUTLINE
  F TIMELINE

C sysproc.c
66
67
68
69
70
71
72
73
74 while(ticks - ticks0 < n){
75   if(myproc() > killed){
76     // ...
77   }
78   sleep(&ticks, &tickslock);
79 }
80
81 release(&tickslock);
82 return 0;
83 }
84
85 // return how many clock tick interrupts have occurred
86 // since start.
87 int
88 sys_uptime(void)
89 {
90   uint xticks;
91
92   acquire(&tickslock);
93   xticks = ticks;
94   release(&tickslock);
95   return xticks;
96 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
cpu0: starting 0
sh: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ pstest
PID    MEM    NAME
1      12288   init
2      16384   sh
3      12288   pstest
$ pstest
Read Count Sebelum: 12
hello
Read Count Setelah: 13
$
```

Hasil modul 2

Tujuan utama dari modul ini adalah:

1. Menambahkan Prioritas: Setiap proses akan memiliki nilai prioritas (priority). Nilai ini akan digunakan oleh scheduler untuk menentukan proses mana yang harus dijalankan.
2. Mengatur Prioritas: Anda akan membuat sebuah system call baru bernama `set_priority(int priority)`. System call ini memungkinkan program dari user-level untuk mengubah prioritasnya sendiri saat runtime.
3. Mengubah Algoritma Penjadwalan: Bagian terpenting dari modul ini adalah memodifikasi fungsi `scheduler()` di kernel. Alih-alih mencari proses `RUNNABLE` berikutnya secara berurutan (Round Robin), scheduler yang baru akan mencari proses `RUNNABLE` dengan nilai prioritas tertinggi (angka terkecil) dan menjalankannya sampai selesai.
4. Non-Preemptive: Algoritma ini bersifat non-preemptive, yang berarti begitu sebuah proses dengan prioritas tinggi mulai berjalan, ia akan terus berjalan sampai selesai atau secara sukarela menyerahkan kontrol CPU (misalnya, saat menunggu I/O). Proses lain, bahkan yang memiliki prioritas sama-sama tinggi, harus menunggu.

Rincian implementasi:

1. Modifikasi Struktur: Tambahkan field `priority` pada `struct proc` di `proc.h`.

2. System Call Baru: Ikuti langkah-langkah standar untuk menambahkan system call (syscall.h, user.h, usys.S, syscall.c) untuk fungsi set_priority(), yang akan diimplementasikan di sysproc.c.
3. Logika Scheduler: Tulis ulang fungsi scheduler() di proc.c. Logika utamanya adalah mencari proses RUNNABLE dengan prioritas terendah (paling tinggi) dan menjalankannya.
4. Program Uji: Buat program ptest.c di user-level untuk menguji fungsionalitas ini. ptest akan membuat dua proses anak dengan prioritas berbeda dan menampilkan hasilnya, membuktikan bahwa proses dengan prioritas yang lebih tinggi (angka lebih kecil) akan selesai lebih dulu.

Pengujian:

- a. ptest

Hasil uji

The screenshot shows a VS Code editor with the following files open: Welcome, C proch, M, C procc, M, C ptest.c, U, M Makefile, M, C syscall.h, M, C user.h, M, and C usy. The main editor displays the implementation of the scheduler in proc.c, which includes comments about the scheduler's purpose and the implementation of the scheduler function. The terminal output shows the booting process from a hard disk, starting the CPU, kernel, and init, and then running the ptest program. The ptest program creates two child processes and prints their completion status.

```

367 // intena because intena is a property of this
368 // kernel thread, not this CPU. It should
369 // be proc->intena and proc->ncli, but that would
370 // break in the few places where a lock is held but
371 // there's no process.
372 void
373 sched(void)
374 {
375     int intena;
376     struct proc *p = myproc();
377
378     if(!holding(&ptable.lock))
379         panic("sched ptable.lock");
380     if(mycpu()->ncli != 1)
381         panic("sched locks");
382     if(p->state == RUNNING)
383         panic("sched running");
384     if(readeflags() & FL_IF)
385         panic("sched interruptible");
386     intena = mycpu()->intena;
387     switch(&p->context, mycpu()->scheduler);
388     mycpu()->intena = intena;
389 }

```

Bootting from Hard Disk..xv6...
cpu0: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 32 bmap start 58
init: starting sh
\$ ptest
Child 1 selesai
Child 2 selesai
Parent selesai
\$

Hasil modul 3

Tujuan utama modul ini adalah:

1. Copy-on-Write (CoW) Fork: Secara default, saat Anda memanggil fork(), seluruh ruang alamat memori proses induk akan disalin ke proses anak. Ini boros jika proses anak tidak

langsung memodifikasi semua memori tersebut. Dengan CoW, fork() tidak lagi menyalin data secara fisik. Sebaliknya, proses induk dan anak akan berbagi halaman memori yang sama, tetapi dengan izin read-only. Jika salah satu proses mencoba menulis ke halaman yang dibagi, ini akan memicu page fault. Saat page fault terjadi, kernel akan menyadari bahwa ini adalah halaman CoW dan akan membuat salinan fisik dari halaman tersebut. Baru setelah itu, operasi tulis diizinkan pada salinan baru. Untuk melacak berapa banyak proses yang berbagi halaman fisik, reference count digunakan. Halaman fisik hanya akan dibebaskan (kfree()) ketika tidak ada lagi proses yang mereferensikannya (reference count-nya nol).

2. Shared Memory (ala System V): Ini adalah mekanisme yang memungkinkan dua atau lebih proses untuk berbagi satu halaman memori fisik yang sama secara eksplisit.

Rincian implementasi:

1. shmget(int key): Digunakan untuk mendapatkan akses ke halaman memori bersama. Jika key sudah ada, proses akan "melekatkan" halaman yang sudah ada. Jika key belum ada, halaman baru akan dialokasikan dan dilekatkan.
2. shmrelease(int key): Digunakan untuk melepaskan akses ke halaman memori bersama. Seperti CoW, halaman fisik hanya akan dibebaskan ketika tidak ada lagi proses yang melekat padanya (menggunakan reference count terpisah).

Pengujian:

- a. cowtest
- b. shmtest

kendala yang dihadapi:

pada pengujian shmtest untuk parent tidak bisa menampilkan huruf B atau membaca dari child dengan hasil parent bentuk diamond, saya melakukan perubahan untuk outputnya

hasil uji

