UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

FACULTAD DE INGENIERÍA

ESCUELA PROFESIONAL DE INGENIERIA EN INFORMATICA Y SISTEMAS

PRODUCTO UNIDAD II:

"EXTENSIÓN Y MONITOREO DEL SISTEMA OPERATIVO XV6"

DOCENTE: Ing. Hugo Manuel Barraza Vizcarra

FACULTAD: Ingeniería ESCUELA: ESIS

CURSO: Sistemas Operativos

CICLO: 6to TURNO: Mañana

ESTUDIANTES:

- Roger Cristian Huanca Pozo 2022-119009
- Nicolas Arturo Marin Gonzales 2022 - 119082

TACNA - PERÚ

2025

1. Introducción y Objetivos

Introducción

Los sistemas operativos constituyen uno de los componentes fundamentales de cualquier sistema de cómputo moderno, ya que son responsables de administrar y coordinar el uso eficiente de los recursos de hardware, tales como el procesador, la memoria principal, los dispositivos de entrada/salida y el sistema de archivos. Para cumplir este rol, el sistema operativo debe implementar mecanismos de planificación y protección que permitan la ejecución concurrente de múltiples procesos de manera segura y eficiente.

En este contexto, el sistema operativo XV6 representa una plataforma educativa ideal para comprender el funcionamiento interno de un sistema tipo UNIX. que conserva los principios fundamentales de diseño de los sistemas operativos reales, permitiendo al estudiante interactuar directamente con el núcleo, modificar su comportamiento e implementar nuevas funcionalidades.

El presente proyecto se desarrolla con la finalidad de extender el sistema operativo XV6 mediante la instrumentación de llamadas al sistema y la implementación de nuevos comandos de monitoreo. Estas extensiones permiten observar el comportamiento interno del sistema y relacionar los conceptos teóricos vistos en clase como planificación de procesos, gestión de memoria y llamadas al sistema con una implementación real.

Objetivos

Objetivo general

Diseñar e implementar extensiones al sistema operativo XV6 que permitan el monitoreo de llamadas al sistema, procesos y métricas básicas de planificación, relacionando su funcionamiento con los conceptos vistos en clase.

Objetivos específicos

- Instrumentar las llamadas al sistema para registrar y visualizar su ejecución en tiempo real.
- Implementar comandos de usuario que permitan consultar información relevante sobre el estado del sistema y los procesos.
- Analizar el impacto de la ejecución de procesos sobre métricas de planificación como los cambios de contexto.
- Aplicar buenas prácticas de programación en C, control de versiones con Git y documentación técnica.
- Relacionar los resultados obtenidos con los conceptos de memoria, planificación y gestión de procesos estudiados en clase.

2. Descripción de las Modificaciones Realizadas

2.1 Instrumentación de llamadas al sistema (Entregable 1)

Como parte del primer entregable, se modificó el núcleo de XV6 para permitir la instrumentación de llamadas al sistema, es decir, registrar y mostrar información detallada cada vez que un proceso invoca una syscall.

Se implementó una variable global en el kernel (syscall_tracing) que actúa como interruptor del logging. Cuando está activada, cada vez que un proceso invoca una syscall, se imprime en consola:

- Identificador del PID del proceso.
- Nombre de la llamada al sistema.
- Parámetros utilizados en la invocación.
- Valor de retorno de la syscall.

Para evitar saturación, se excluyen:

- La propia syscall trace (para evitar recursión)
- Procesos con PID ≤ 2 (init y sh)

Archivos modificados/creados

Archivo	Cambio realizado	Propósito
syscall.h	Añadido #define SYS_trace 22	Definir número de la nueva syscall
syscall.c	•VariableGlobal syscall_tracing • Array syscall_names[] •Función print_syscall_args() • Lógica de logging en syscall()	Núcleo del mecanismo de tracing
sysproc.c	Implementación de sys_trace()	Manejo del argumento on/off en el kernel
user.h	Declaración int trace(int);	Exportar syscall al espacio de usuario
usys.S	Añadido SYSCALL(trace)	Stub en ensamblador para invocar la syscall
strace.c	Programa de usuario nuevo	Comando para activar/desactivar tracing

Makefile Incluir _strace en UPROGS Compilar e instalar el comando

---

2.1.1 Fragmentos relevantes de código comentado

syscall.c – Variable global y nombres de syscalls

```
// Variable global para activar/desactivar el tracing
int syscall_tracing = 0;

// Nombres de todas las syscalls (índice = número de syscall)
char* syscall_names[] = {
  [SYS_fork]    "fork",
  [SYS_exit]    "exit",
  // ... otras syscalls
  [SYS_trace]   "trace",  // La que creamos
};
```

syscall.c – Función de logging mejorada

```
void print_syscall_args(int num, struct proc* p) {
  switch(num) {
    case SYS_fork:
    case SYS_getpid:
      cprintf("()");  // Syscalls sin argumentos
      break;
    case SYS_kill:
      cprintf("(pid=%d)", p->tf->ebx);  // Primer argumento en ebx
      break;
    case SYS_open:
      cprintf("(path=0x%x, mode=%d)", p->tf->ebx, p->tf->ecx);
      break;
    default:
      cprintf("(...)");
  }
}
```

syscall.c – Punto de entrada modificado

```
void syscall(void) {
  int num;
  struct proc *curproc = myproc();
  num = curproc->tf->eax;

  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // 1. Ejecutar la syscall
    int result = syscalls[num]();
    curproc->tf->eax = result;

    // 2. Mostrar logging si está activado y no es trace
    if(syscall_tracing && num != SYS_trace && curproc->pid > 2) {
      cprintf("[%d] %s", curproc->pid, syscall_names[num]);
      print_syscall_args(num, curproc);
      cprintf(" = %d\n", result);
    }
  } else {
    // Manejo de syscall desconocida
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

sysproc.c – Implementación de la syscall trace

```
int sys_trace(void) {
  int n;
  if(argint(0, &n) < 0)  // Obtener argumento (0 o 1)
    return -1;
  if(n != 0 && n != 1)   // Validación
    return -1;
  syscall_tracing = n;   // Activar/desactivar
  return 0;              // Éxito
}
```

strace.c – Comando de usuario

```
int main(int argc, char *argv[]) {
```

```
  if(argc != 2) {
    printf(2, "Uso: strace <on|off>\n");
    exit();
  }
  if(strcmp(argv[1], "on") == 0) {
    if(trace(1) < 0) {
      printf(2, "Error: No se pudo activar\n");
      exit();
    }
    printf(1, "Syscall tracing: ON\n");
  }
  // ... manejo de "off"
}
```

2.2 Comandos de monitoreo del sistema (Entregable 2)

En el segundo entregable se implementaron nuevos comandos de usuario orientados al monitoreo del sistema, alineados con los temas de la Unidad II.

Comando uptimex

El comando uptimex extiende la funcionalidad del comando uptime original de XV6. Este comando muestra:

- Tiempo de actividad del sistema en ticks y segundos.
- Conversión del tiempo a formato HH:MM:SS.
- Número de procesos activos en el sistema.
- Contador global de cambios de contexto, como métrica básica de planificación.
- Límite máximo de procesos soportados por el sistema.

Este comando permite observar cómo la ejecución de procesos y comandos afecta directamente al scheduler del sistema.

Comando psinfo

El comando psinfo permite inspeccionar el estado de los procesos en ejecución. Puede utilizarse de dos maneras:

- psinfo: muestra una lista completa de los procesos activos.

- psinfo <PID>: muestra información detallada de un proceso específico.

Archivos modificados:

2.2.1 **Comando uptimex - Extensión del tiempo de actividad**

Archivo	Cambio realizado	Propósito
uptimex.c	Creación de nuevo comando de usuario	Extender funcionalidad de uptime original con métricas del sistema
sysproc.c	Implementación de sys_getnproc() y sys_getcontextsw()	Proporcionar información del sistema al espacio de usuario
proc.c	Adición de contador context_switches e incremento en scheduler	Rastrear cambios de contexto para análisis de planificación
user.h	Declaración de nuevas syscalls: getnproc(), getcontextsw()	Interfaces para comunicación kernel-user
syscall.h	Definición de números: SYS_getnproc, SYS_getcontextsw	Identificadores únicos para syscalls
usys.S	Entradas: SYSCALL(getnproc), SYSCALL(getcontextsw)	Generación de stubs de ensamblador
Makefile	Adición de _uptimex a UPROGS	Inclusión del comando en el sistema de archivos

Funcionalidad:

- Tiempo en ticks y formato HH:MM:SS
- Número de procesos activos (getnproc())
- Contador de cambios de contexto (getcontextsw())
- Límite máximo de procesos (64 en XV6)

2.2.2 Comando psinfo - Información de procesos

Archivo	Cambio realizado	Propósito
psinfo.c	Creación de nuevo comando de usuario	Mostrar estado y detalles de procesos del sistema
sysproc.c	Implementación de sys_getprocinfo()	Copiar información de procesos desde kernel a user space
proc.h	Definición de estructura psinfo	Estructura de datos compartida kernel-user
user.h	Declaración de getprocinfo() y estructura psinfo	Interface tipo-safe para transferencia de datos
syscall.h	Definición de SYS_getprocinfo	Identificador único para syscall
usys.S	Entrada: SYSCALL(getprocinfo)	Stub de ensamblador para la syscall
Makefile	Adición de _psinfo a UPROGS	Inclusión en sistema de archivos

Funcionalidades:

- Modo lista: muestra tabla con todos los procesos activos
- Modo detalle: información específica de un PID
- Campos mostrados: PID, nombre, estado, memoria, PID padre
- Estados soportados: RUNNING, RUNNABLE, SLEEPING, ZOMBIE, UNUSED

2.2.3 FRAGMENTOS RELEVANTES DE CÓDIGO COMENTADO

1. IMPLEMENTACIÓN DE SYSCALLS EN sysproc.c

1.1 sys_getnproc() - Contar procesos activos

```
int sys_getnproc(void)
{
  struct proc *p;
  int count = 0;
```

```
  // Adquirir lock para acceso seguro a la tabla de procesos
  acquire(&ptable.lock);

  // Recorrer toda la tabla de procesos (NPROC = 64 procesos máximos)
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    // Solo contar procesos que NO estén en estado UNUSED (no utilizados)
    if(p->state != UNUSED) {
      count++;  // Incrementar contador por cada proceso activo
    }
  }

  // Liberar lock después de la operación
  release(&ptable.lock);
  return count;  // Retornar número total de procesos activos
}
```

Recorre la tabla global de procesos (ptable) y cuenta cuántos no están en estado UNUSED. Esto incluye procesos en estados: EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE.

1.2 sys_getprocinfo() - Obtener información detallada de un proceso

```
int sys_getprocinfo(void)
{
  int pid;                  // PID solicitado por el usuario
  struct psinfo *user_info;   // Puntero a estructura en espacio de usuario
  struct proc *p;           // Puntero para recorrer procesos
  int i;                    // Contador para copia de strings

  // Paso 1: Obtener parámetros de la syscall
  // argint(0, &pid) obtiene el primer parámetro (PID)
  // argptr(1, ...) obtiene el segundo parámetro (puntero a estructura)
  if(argint(0, &pid) < 0 ||
     argptr(1, (char**)&user_info, sizeof(struct psinfo)) < 0) {
    return -1;  // Error si los parámetros son inválidos
  }

  // Paso 2: Bloquear tabla de procesos para acceso exclusivo
  acquire(&ptable.lock);

  // Paso 3: Buscar proceso con el PID solicitado
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    // Verificar: mismo PID Y proceso no está UNUSED
    if(p->pid == pid && p->state != UNUSED) {

      // Paso 4: Copiar información básica del proceso
```

```
    user_info->pid = p->pid;  // PID del proceso

    // PID del padre (si existe, sino usar 1 para init)
    user_info->ppid = (p->parent != 0) ? p->parent->pid : 1;

    user_info->sz = p->sz;    // Tamaño de memoria del proceso

    // Paso 5: Convertir estado numérico a string descriptivo
    const char* state_str;
    switch(p->state) {
      case UNUSED:    state_str = "UNUSED"; break;     // 0
      case EMBRYO:    state_str = "EMBRYO"; break;     // 1
      case SLEEPING:  state_str = "SLEEPING"; break;   // 2
      case RUNNABLE:  state_str = "RUNNABLE"; break;   // 3
      case RUNNING:   state_str = "RUNNING"; break;    // 4
      case ZOMBIE:    state_str = "ZOMBIE"; break;     // 5
      default:        state_str = "UNKNOWN"; break;    // Por seguridad
    }

    // Paso 6: Copiar string de estado carácter por carácter
    // Se hace manualmente por seguridad (evitar desbordamientos)
    for(i = 0; i < 15 && state_str[i] != '\0'; i++) {
      user_info->state[i] = state_str[i];
    }
    user_info->state[i] = '\0';  // Terminar string con null

    // Paso 7: Copiar nombre del proceso
    for(i = 0; i < 15 && p->name[i] != '\0'; i++) {
      user_info->name[i] = p->name[i];
    }
    user_info->name[i] = '\0';  // Terminar string con null

    // Paso 8: Liberar lock y retornar éxito
    release(&ptable.lock);
    return 0;  // Éxito: proceso encontrado y copiado
   }
  }

  // Paso 9: Si llegamos aquí, el proceso no fue encontrado
  release(&ptable.lock);
  return -1;  // Error: proceso no encontrado
}
```

Busca un proceso específico por su PID y copia su información a una estructura en espacio de usuario.

1.3 sys_getcontextsw() - Obtener contador de cambios de contexto

```
// Variable global definida en proc.c
extern int context_switches;  // Declaración externa

int sys_getcontextsw(void)
{
  return context_switches;  // Simplemente retorna el contador global
}
```

Retorna el valor actual del contador global de cambios de contexto.
Cada cambio de contexto representa una interrupción, donde se guarda el estado de un proceso y se carga otro. Este contador es una métrica de actividad del planificador.

2. MODIFICACIONES EN EL SCHEDULER (proc.c)

2.1 Incremento del contador de cambios de contexto

```
// Variable global (definida al inicio de proc.c)
int context_switches = 0;  // Inicializada en 0 al arranque

void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;) {  // Loop infinito del scheduler
    sti();  // Habilitar interrupciones

    // Bloquear tabla de procesos para búsqueda segura
    acquire(&ptable.lock);

    // Buscar proceso RUNNABLE para ejecutar
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if(p->state != RUNNABLE)
        continue;  // Saltar procesos no listos

      // Proceso seleccionado para ejecución
      c->proc = p;           // Asignar proceso a CPU
      switchuvm(p);          // Cambiar espacio de memoria
      p->state = RUNNING;    // Actualizar estado
```

```
    // --- CAMBIO DE CONTEXTO ---
    swtch(&(c->scheduler), p->context);  // Guardar estado del scheduler
    context_switches++;                  // ¡INCREMENTAR CONTADOR AQUÍ!
    switchkvm();          // Restaurar memoria del kernel
    // --- FIN CAMBIO DE CONTEXTO ---

    c->proc = 0;  // Proceso ya no está en CPU
   }

   release(&ptable.lock);  // Liberar tabla de procesos
  }
}
```

Cada vez que el scheduler cambia de proceso (usando swtch()), incrementa el contador global context_switches. El incremento ocurre después de swtch() porque swtch() guarda contexto actual y carga nuevo contexto,cuando retorna, ya se ejecutó otro proceso ,el incremento cuenta ese cambio ya completado

3. COMANDOS DE USUARIO

3.1 uptimex.c - Monitoreo extendido del sistema

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void) {
  // --- OBTENER DATOS DEL SISTEMA MEDIANTE SYSCALLS ---
  int ticks = uptime();          // Llamada existente: tiempo en ticks
  int num_proc = getnproc();     // Nuestra syscall: procesos activos
  int context_sw = getcontextsw(); // Nuestra syscall: cambios de contexto

  // --- CONVERSIÓN DE UNIDADES ---
  // XV6: 100 ticks = 1 segundo (frecuencia del timer)
  int seconds = ticks / 100;     // Convertir ticks a segundos
  int minutes = seconds / 60;    // Segundos a minutos
  int hours = minutes / 60;      // Minutos a horas

  // --- PRESENTACIÓN FORMATEADA ---
  printf(1, "\n========================================\n");
  printf(1, "       TIEMPO DE ACTIVIDAD XV6          \n");
  printf(1, "========================================\n");
```

```
  // Tiempo en diferentes formatos
  printf(1, "Tiempo de actividad del sistema:\n");
  printf(1, "  • Ticks:     %d\n", ticks);      // Unidad base
  printf(1, "  • Segundos:  %d\n", seconds);    // Para humanos

  // Formato HH:MM:SS con ceros a la izquierda
  printf(1, "  • Tiempo:   %d:", hours);
  if((minutes % 60) < 10) printf(1, "0");  // Minutos con 2 dígitos
  printf(1, "%d:", minutes % 60);
  if((seconds % 60) < 10) printf(1, "0");  // Segundos con 2 dígitos
  printf(1, "%d\n", seconds % 60);

  // Información del scheduler y procesos
  printf(1, "----------------------------------------\n");
  printf(1, "Información del sistema:\n");
  printf(1, "  • Procesos activos:     %d\n", num_proc);
  printf(1, "  • Cambios de contexto:  %d\n", context_sw);
  printf(1, "  • Máximo de procesos:   64\n");  // NPROC en XV6

  printf(1, "========================================\n\n");

  exit();  // Terminar proceso correctamente
}
```

Es un **comando de diagnóstico** que muestra:

1. **Tiempo del sistema** en múltiples formatos
2. **Carga del sistema** (# procesos activos)
3. **Actividad del scheduler** (# cambios de contexto)

3.2 psinfo.c - Inspector de procesos

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[]) {
  struct psinfo p;      // Estructura para información de proceso
  int show_all = 1;     // Bandera: 1=mostrar todos, 0=mostrar uno
  int specific_pid = -1;// PID específico si se proporciona

  // --- ANÁLISIS DE ARGUMENTOS ---
  // Formato: psinfo          → mostrar todos
  //          psinfo <PID>    → mostrar proceso específico
```

```
if(argc == 2) {
  specific_pid = atoi(argv[1]);  // Convertir argumento a número
  show_all = 0;                  // Cambiar a modo detalle
}

// --- CABECERA ---
printf(1, "\n================================================\n");
printf(1, "         INFORMACIÓN DE PROCESOS XV6          \n");
printf(1, "================================================\n");

// --- MODO DETALLE (PID ESPECÍFICO) ---
if(!show_all && specific_pid > 0) {
  // Llamar a syscall con el PID solicitado
  if(getprocinfo(specific_pid, &p) == 0) {
    // Mostrar información detallada formateada
    printf(1, "PID:            %d\n", p.pid);
    printf(1, "Nombre:         %s\n", p.name);
    printf(1, "Estado:         %s\n", p.state);
    printf(1, "PID Padre:      %d\n", p.ppid);
    printf(1, "Memoria:        %d bytes\n", p.sz);
  } else {
    printf(1, "Proceso %d no encontrado\n", specific_pid);
  }
}
// --- MODO LISTA (TODOS LOS PROCESOS) ---
else {
  // Encabezado de tabla
  printf(1, "PID  ESTADO    NOMBRE       MEMORIA  PADRE\n");
  printf(1, "---  ---------  ------------  -------  -----\n");

  int count = 0;  // Contador de procesos encontrados

  // Buscar procesos con PID desde 1 hasta 100
  // Nota: En XV6, PIDs se asignan secuencialmente
  for(int pid = 1; pid <= 100; pid++) {
    // Intentar obtener información de cada PID
    if(getprocinfo(pid, &p) == 0) {
      // Mostrar en formato tabla (simple, sin alineación avanzada)
      printf(1, "%d   %s   %s   %d   %d\n",
             p.pid, p.state, p.name, p.sz, p.ppid);
      count++;  // Incrementar contador
    }
  }

  printf(1, "\nTotal de procesos: %d\n", count);
}

printf(1, "================================================\n\n");
```

```
  exit();  // Salir correctamente
}
```

Proporciona dos modos de visualización:

1. Modo lista: Tabla con todos los procesos activos
2. Modo detalle: Información completa de un proceso específico

Detalles técnicos:

- PID 1: Siempre es init (primer proceso del sistema)
- PID padre 1: Indica que el padre es init (procesos huérfanos)
- Estado RUNNING: Solo un proceso por CPU puede estar en este estado
- Estado ZOMBIE: Proceso terminado esperando que padre llame a wait()

4. ESTRUCTURAS DE DATOS COMPARTIDAS

4.1 struct psinfo (definida en user.h y proc.h)

```
// Esta estructura debe ser IDÉNTICA en kernel y user space
struct psinfo {
  int pid;        // Process ID (4 bytes)
  int ppid;       // Parent Process ID (4 bytes)
  char state[16]; // Estado como string (16 bytes max)
  char name[16];  // Nombre del proceso (16 bytes max)
  uint sz;        // Tamaño de memoria en bytes (4 bytes)
};
// Total: 4+4+16+16+4 = 44 bytes
```

2.3 Tabla de modificaciones para Entregable 3

Archivo	Cambio realizado	Propósito
syscall.c	1.Array syscall_count[] 2. init_syscall_counters() 3. syscall_count[num]++ en handler	Almacenar e incrementar contadores de cada syscall
sysproc.c	Implementación de sys_getsyscallcount()	Proporcionar interfaz para consultar contadores

syscall.h	Definición de SYS_getsyscallcount = 26 y NSYSCALLS = 27	Números únicos para syscalls y constante de tamaño
user.h	Declaración de getsyscallcount(int)	Prototipo para espacio de usuario
usys.S	Entrada SYSCALL(getsyscallcount)	Stub en ensamblador para la syscall
main.c	Llamada a init_syscall_counters() en main()	Inicializar contadores al arrancar el sistema
syscount.c	Creación de nuevo comando de usuario	Interfaz amigable para consultar estadísticas
Makefile	Adición de _syscount a UPROGS	Incluir comando en sistema de archivos

---

```
# Compilar
make clean
make

# Ejecutar en QEMU
make qemu


#Comandos para presentable 01:
$ strace on    # Activar tracing
$ ls          # Ver llamadas al sistema
$ strace off   # Desactivar tracing



#Comandos para presentable 02:
$ uptimex          # Tiempo del sistema y métricas de planificación
$ psinfo           # Lista completa de procesos activos
$ psinfo 1         # Información detallada del proceso init (PID 1)
$ psinfo 2         # Información del shell principal (PID 2)

# Demostración de funcionalidad
$ uptimex          # Anotar cambios de contexto iniciales
$ ls               # Ejecutar comandos para generar actividad
$ cat README       # Más actividad del sistema
$ uptimex          # Ver incremento en cambios de contexto
```

5. CONCLUSIONES:

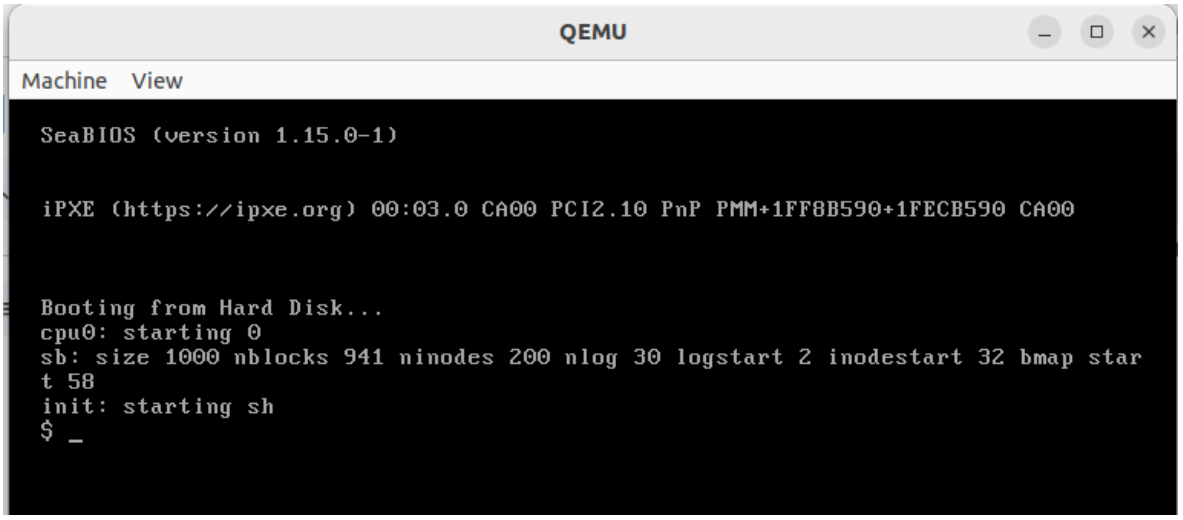
El desarrollo del proyecto permitió comprender de manera práctica los mecanismos internos de un sistema operativo real, a través de la extensión del kernel de XV6 y la creación de nuevas herramientas de monitoreo del sistema y de procesos. La implementación de instrumentación de llamadas al sistema, la obtención de métricas de planificación y el acceso a información detallada de cada proceso consolidaron la integración entre teoría y práctica, demostrando cómo conceptos esenciales como gestión de procesos, cambios de contexto, temporización, interrupciones y comunicación núcleo–usuario ocurren realmente dentro del sistema operativo.

Asimismo, el proceso requirió un análisis profundo del funcionamiento de las system calls, del scheduler y de la tabla de procesos, así como la modificación coordinada de estructuras de datos, syscalls, comandos de usuario y componentes del kernel. Esto fortaleció las habilidades de programación en C, el uso de estructuras del núcleo, el manejo de memoria, la sincronización con locks y la comprensión del ciclo de vida de los procesos.

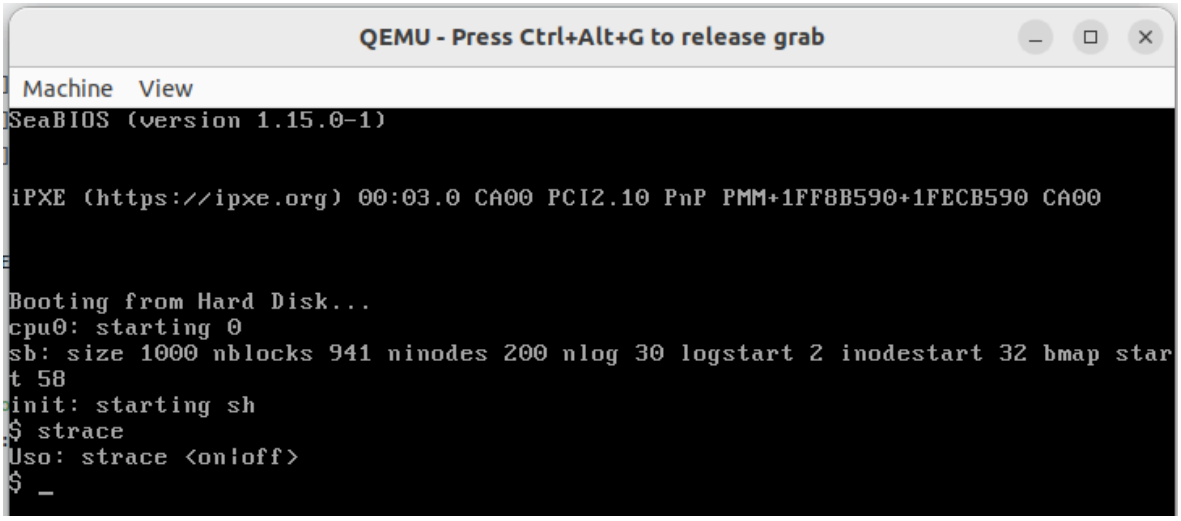
Finalmente, los resultados obtenidos evidencian que extender un sistema operativo es completamente alcanzable cuando se domina su arquitectura interna y se comprende el flujo de ejecución entre el espacio de usuario y el espacio del kernel. El proyecto no solo permitió implementar nuevas herramientas funcionales, sino también observar directamente el comportamiento del sistema en ejecución, convirtiendo la teoría en experiencia concreta y enriqueciendo significativamente el entendimiento de los sistemas operativos modernos.

Evidencia: Capturas de pantalla de entregable 1

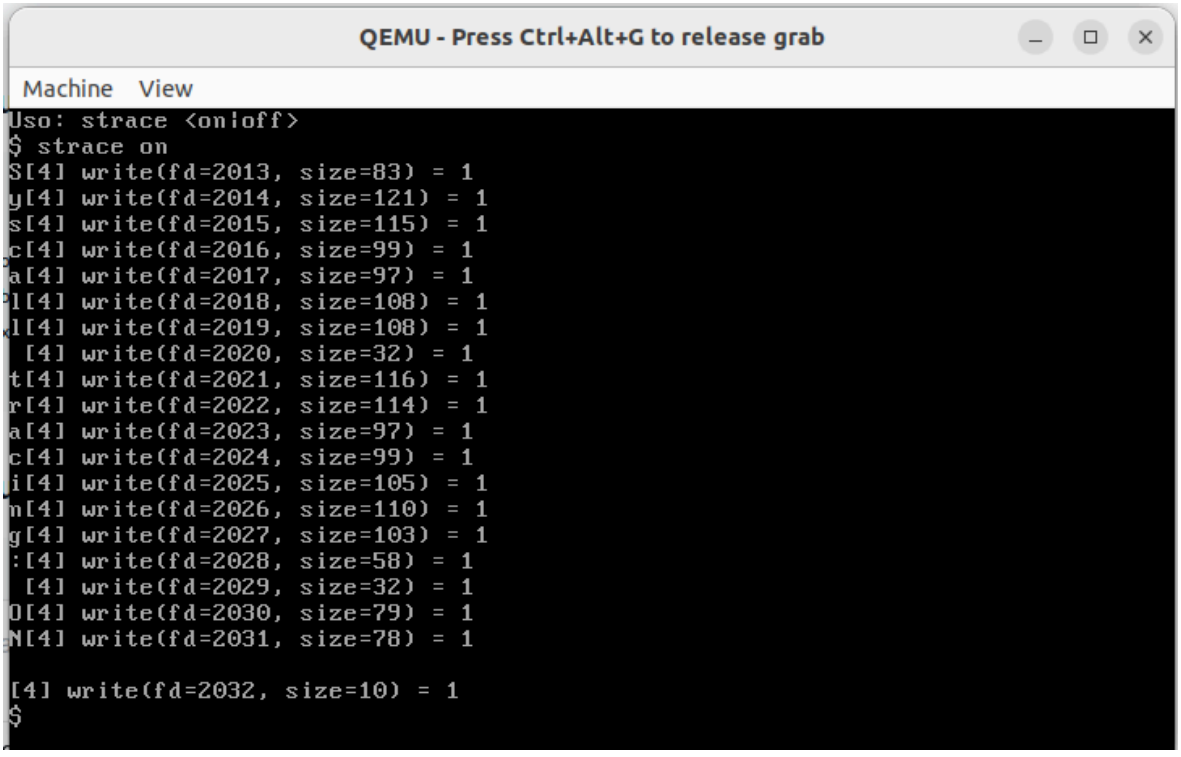
Captura de pantalla del sistema XV6



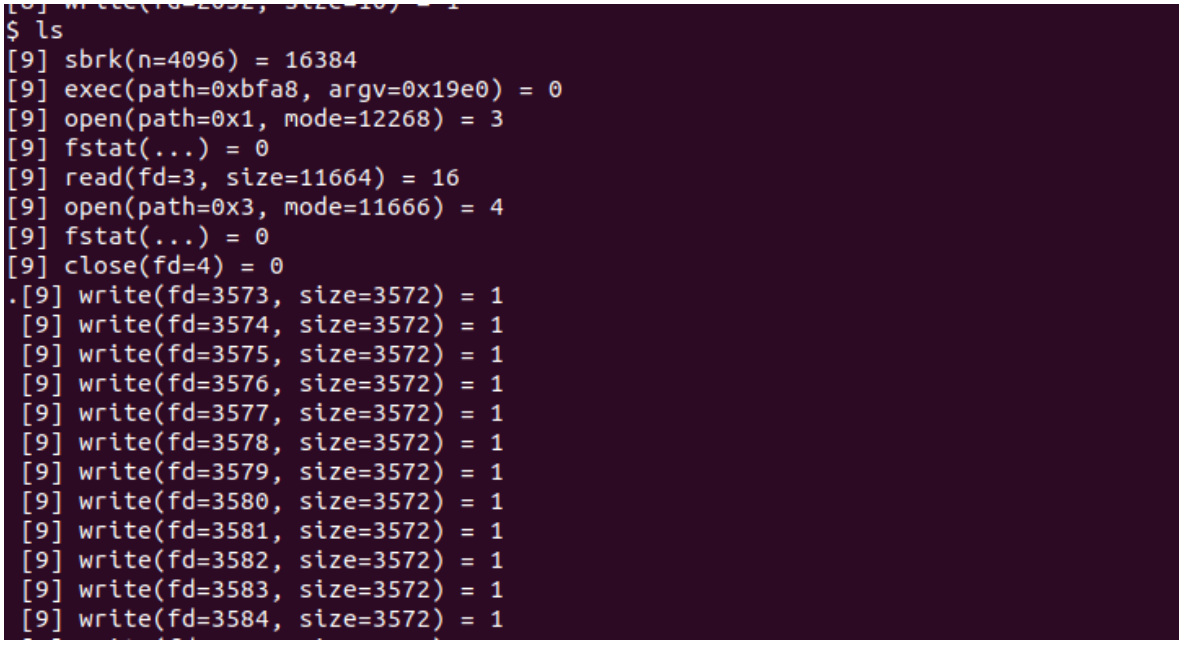
Captura 1: Ayuda del comando



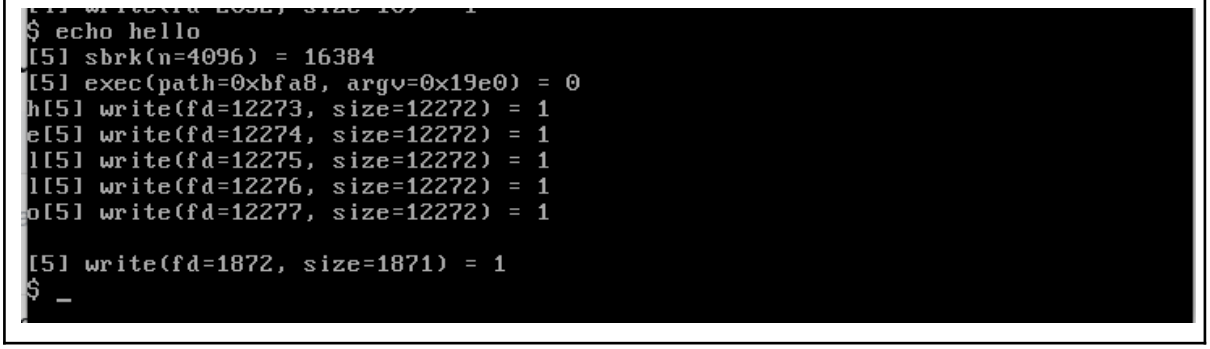
Captura 2: Activación



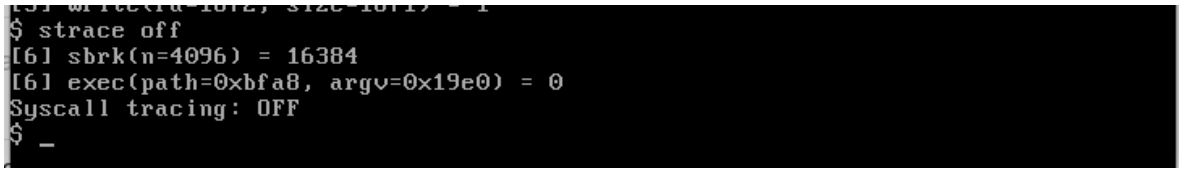
Captura 3: después de la activación mostrar ls



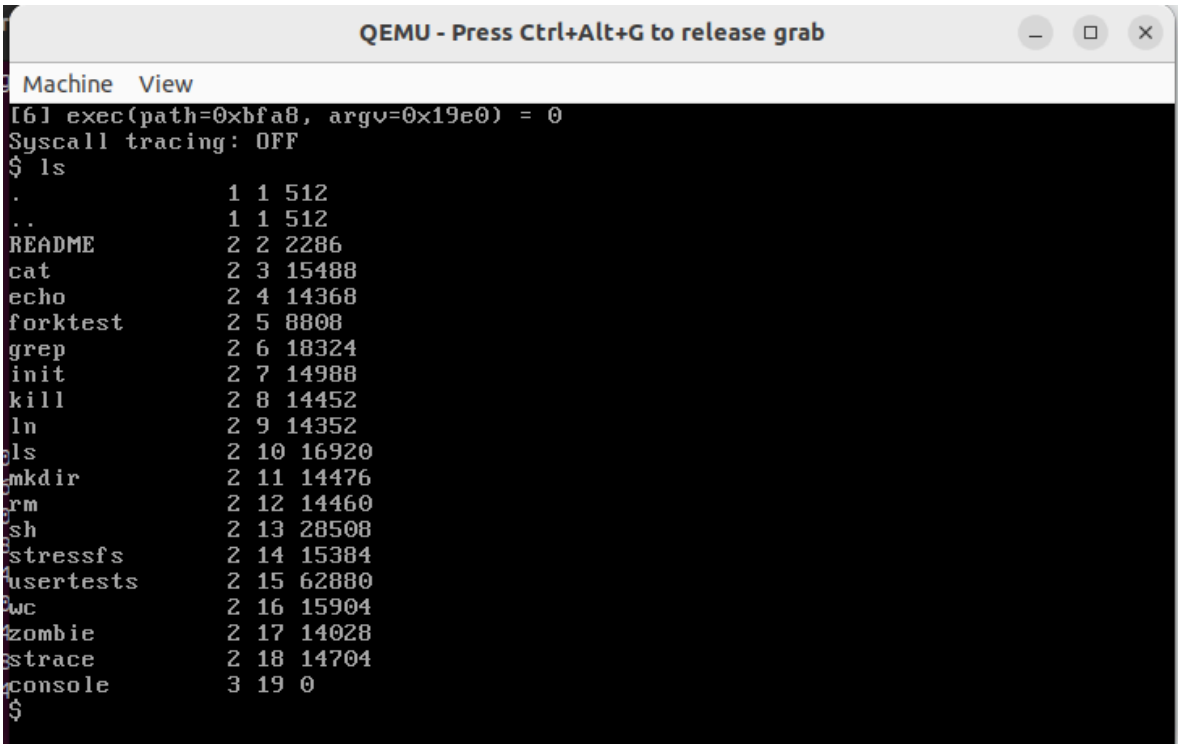
Captura 4: Otro comando



Captura 5: Desactivación y verificación

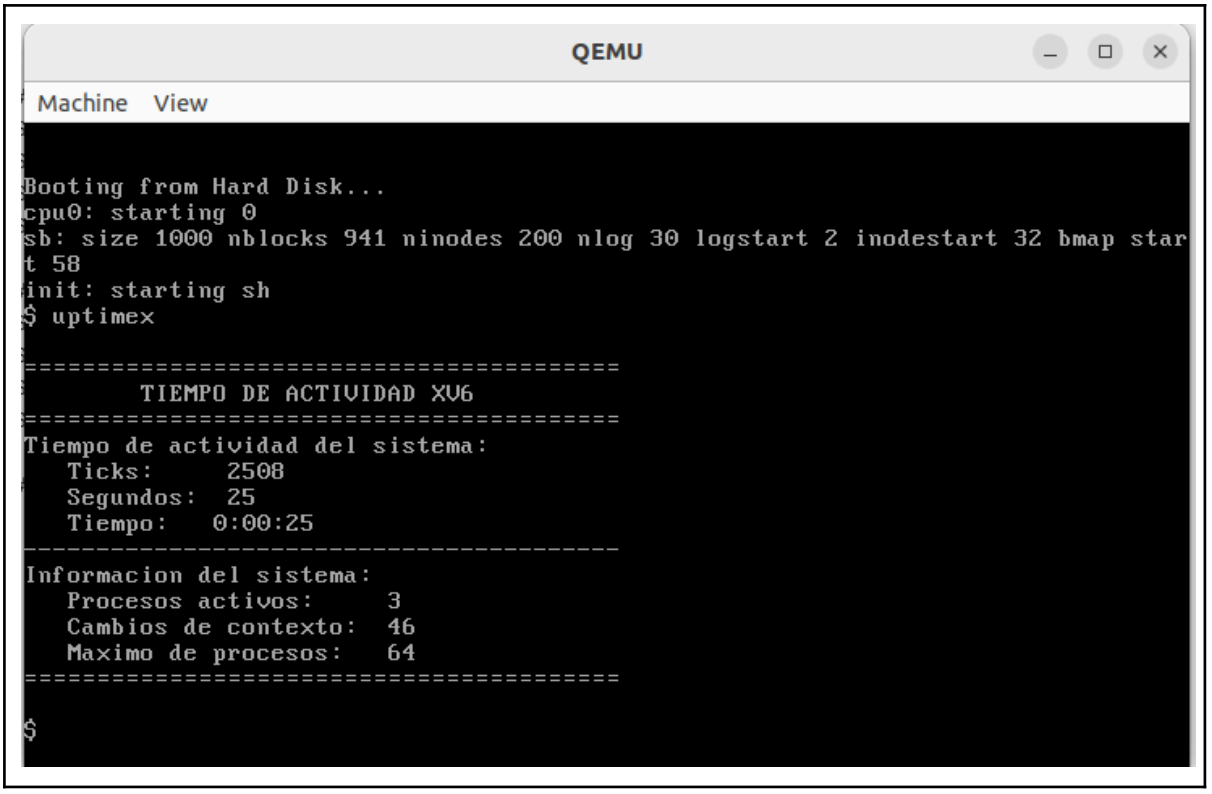


Captura 6:Mostrar ls despues de OFF



Evidencia: Capturas de pantalla de entregable 2

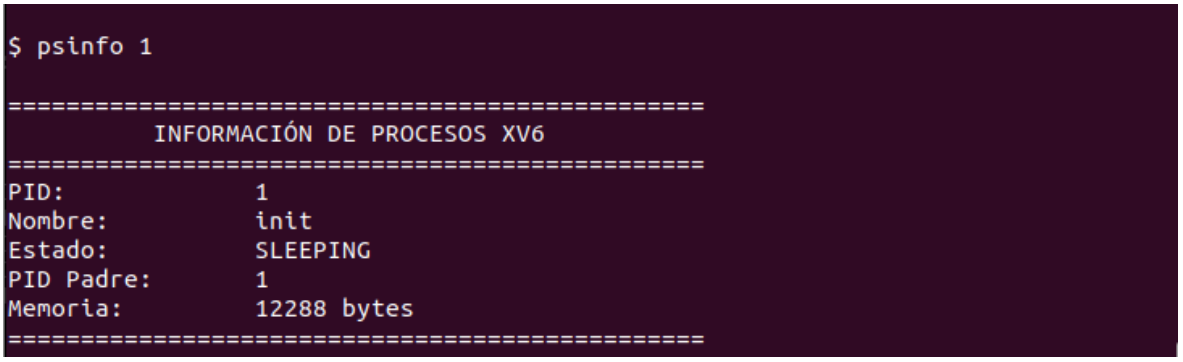
Captura 7: Visualización del tiempo de actividad del sistema



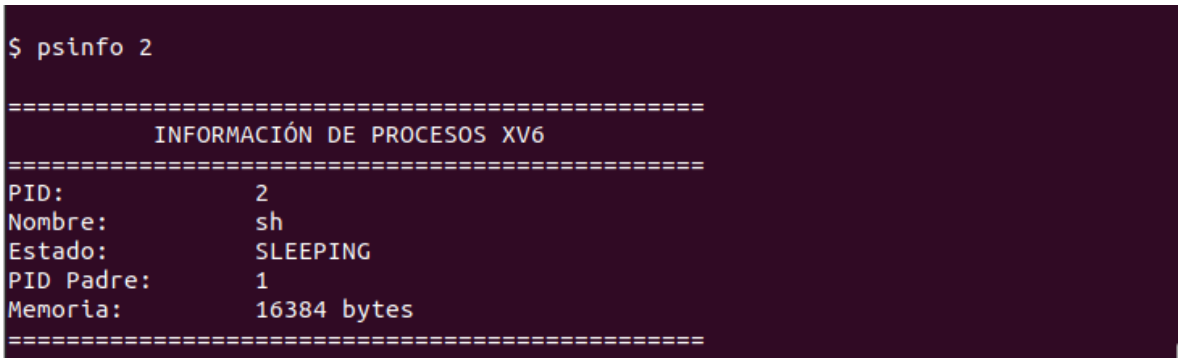
Captura 8: Listado completo de procesos activos

```
$ psinfo

================================================
          INFORMACIÓN DE PROCESOS XV6
================================================
PID  ESTADO     NOMBRE        MEMORIA  PADRE
---  ---------  ------------  -------  -----
1    SLEEPING    init    12288    1
2    SLEEPING    sh    16384    1
4    RUNNING    psinfo    12288    2

Total de procesos: 3
================================================
```

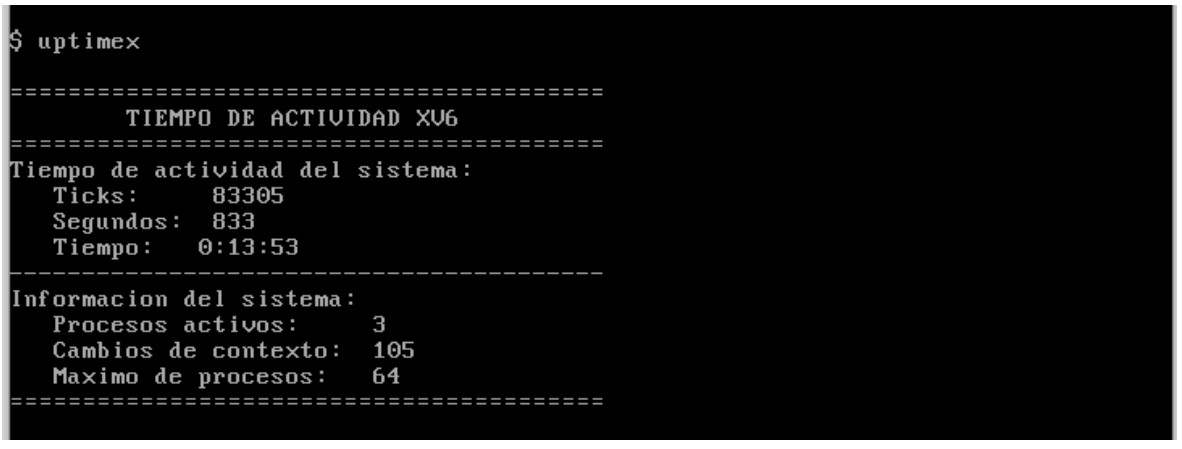
Captura 9: Información detallada del proceso init (PID 1)



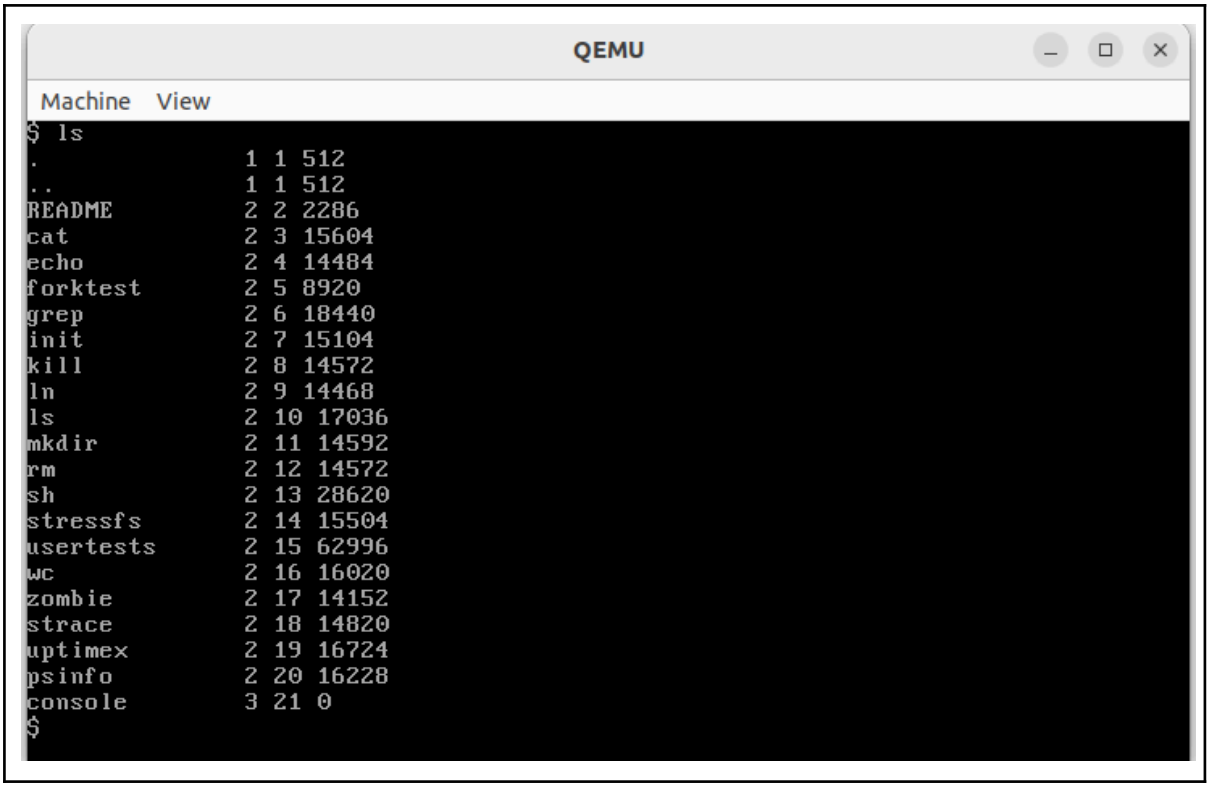
Captura 10: Información del proceso del shell principal (PID 2)



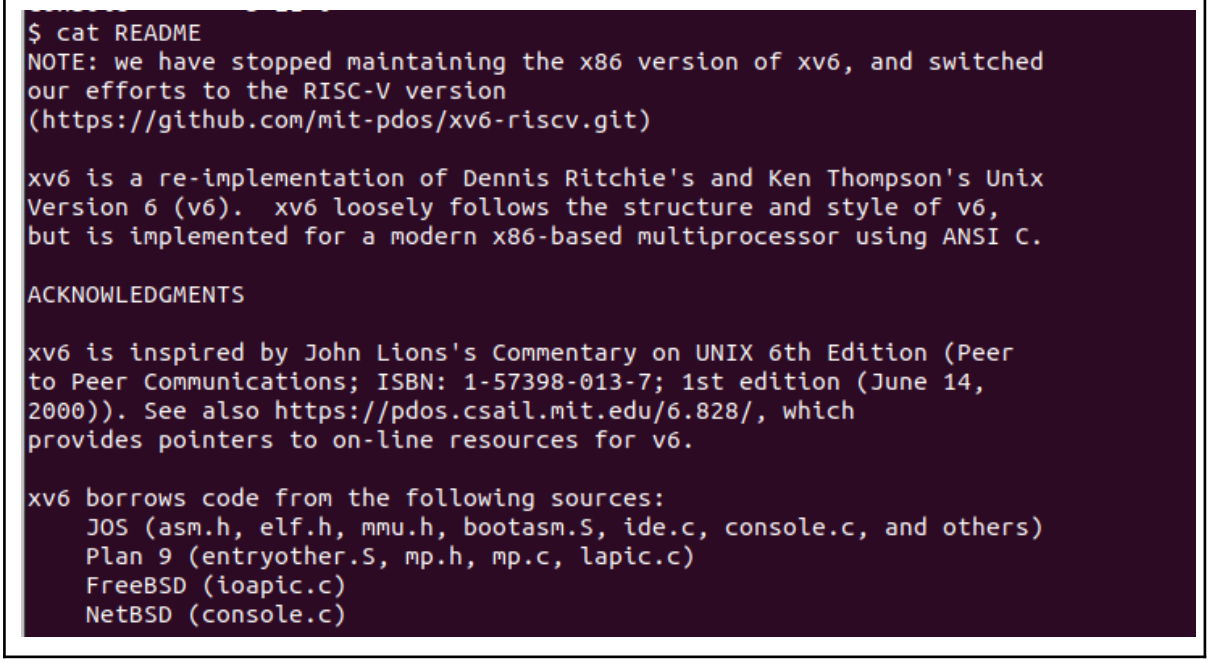
Captura 11: Estado inicial de los cambios de contexto



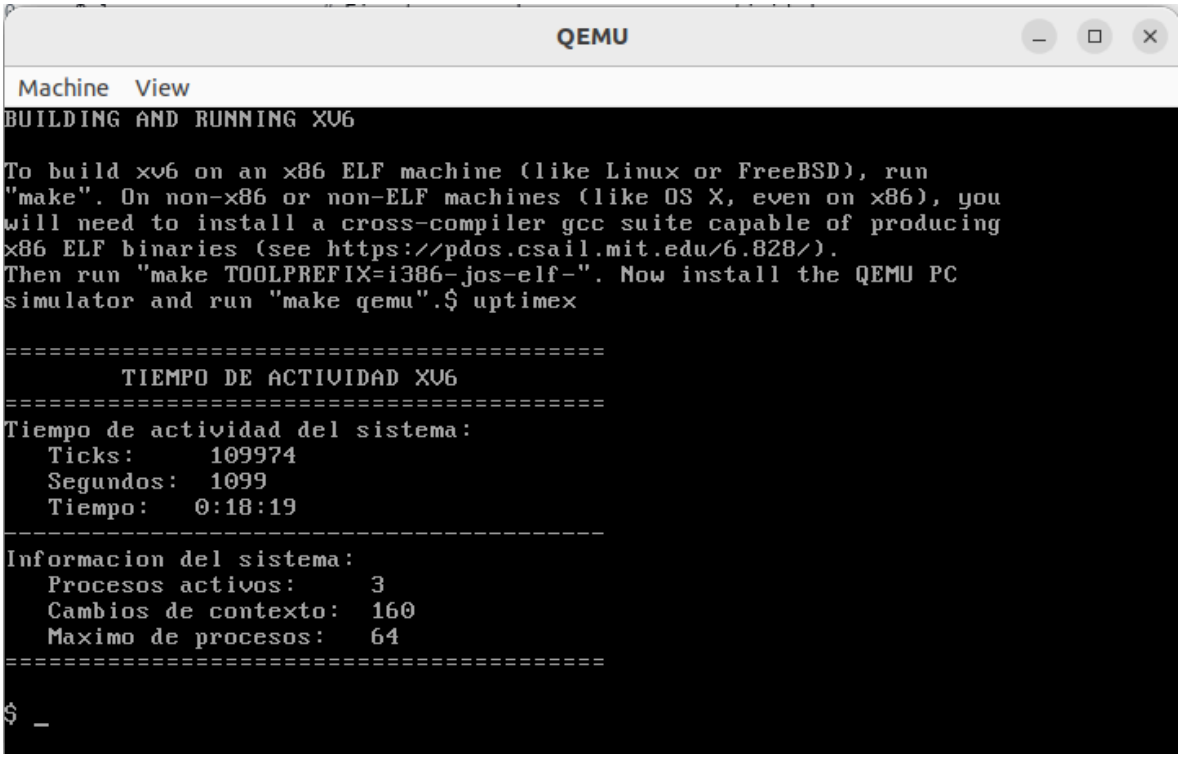
Captura 12: Generación de actividad del sistema (comando ls)



Captura 13: Generación de actividad del sistema (lectura de archivo)

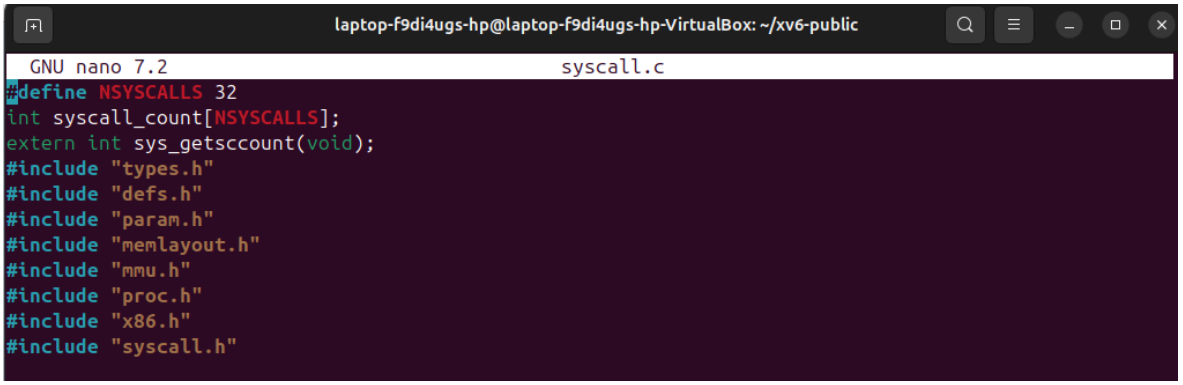


Captura 14: Verificación del incremento de cambios de contexto



Evidencia: Capturas de pantalla de entregable 3

Captura 15: Diseñar una estructura de datos que mantenga, en el núcleo, el número de invocaciones que ha tenido cada llamada al sistema (p.ej., un arreglo indexado por el número de syscall).



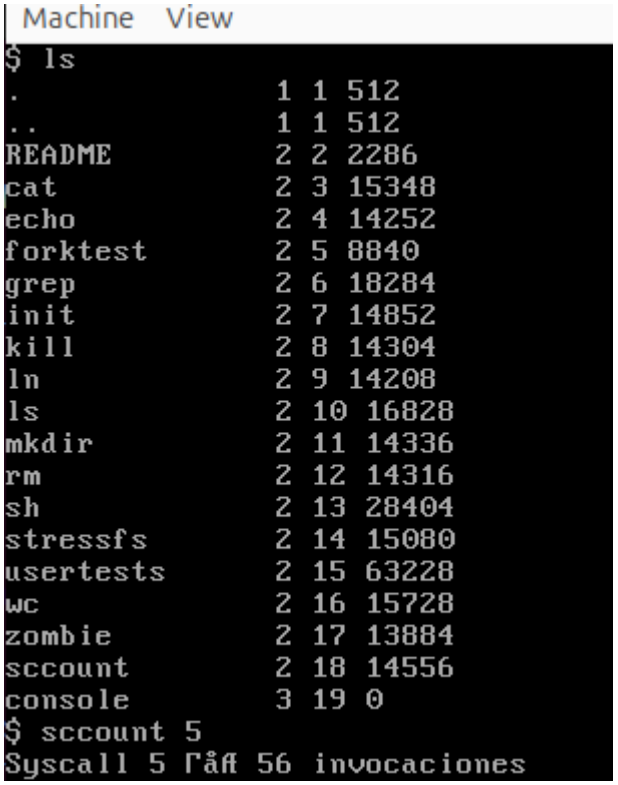
Captura 16: Diseñar una estructura de datos que mantenga, en el núcleo, el número de invocaciones que ha tenido cada llamada al sistema (p.ej., un arreglo indexado por el número de syscall).

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NSYSCALLS)
        syscall_count[num]++;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

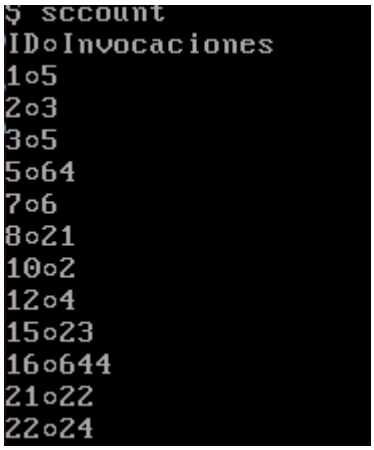
Captura 17: Implementar una nueva llamada al sistema que permita consultar esos contadores

```
  GNU nano 7.2                                        sysproc.c

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}

int
sys_getsccount(void)
{
int num;
if(argint(0, &num)< 0)
 return -1;

if(num < 0|| num>= NSYSCALLS)
 return -1;

return syscall_count[num];
}
```

Captura 18: Comando de usuario: con parámetro



Captura 19: Comando de usuario: sin parámetros



link de github :
https://github.com/Roger-cristian-huanca-pozo/xv6-proyecto-final-so.git

Recursos y Referencias

- XV6 Documentation - Documentación oficial
- XV6 Book - Libro de referencia
- GitHub Repository - Código fuente original