UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

FACULTAD DE INGENIERÍA

ESCUELA PROFESIONAL DE INFORMÁTICA Y SISTEMAS

INFORME TÉCNICO:

"PROYECTO XV6"

Docente: Hugo Manuel Barraza Vizcarra

Integrantes:

Alex Yasmani Huaracha Bellido 2023 - 119027

Jheral Jhosue Maquera Laque 2023 - 119037

Ciclo académico : 6 ciclo

Asignatura: Sistemas Operativos

TACNA - PERÚ

2023

1. Introducción

El sistema operativo XV6 es una reimplementación moderna del Unix Versión 6, diseñada con fines didácticos para ilustrar los conceptos fundamentales de la arquitectura de núcleos monolíticos. En el contexto de la Unidad II: Gestión de Procesos y Planificación, es crucial comprender no solo teóricamente cómo se administran los recursos, sino visualizar dinámicamente el comportamiento interno del sistema.

El presente informe detalla el proceso de modificación del kernel de XV6 para incorporar nuevas funcionalidades de instrumentación y diagnóstico. A diferencia de un sistema estándar que opera como una "caja negra", las modificaciones realizadas permiten al administrador inspeccionar la tabla de procesos en tiempo real, rastrear la ejecución de llamadas al sistema (syscalls) y recolectar estadísticas de uso. Estas herramientas son fundamentales para tareas de depuración, análisis de rendimiento y comprensión del ciclo de vida de los procesos (creación, ejecución, bloqueo y terminación).

2. Objetivos

2.1. Objetivo general

Instrumentar el núcleo del sistema operativo XV6 mediante la implementación de nuevas llamadas al sistema y comandos de usuario, permitiendo la visualización y análisis de la gestión de procesos y el manejo de interrupciones.

2.2. Objetivos específicos

- Visualizar la Tabla de Procesos: Implementar la syscall psmem para inspeccionar el estado (RUNNING, SLEEPING, ZOMBIE), identificadores y consumo de memoria de los procesos activos.
- Rastrear Llamadas al Sistema: Desarrollar la herramienta trace para interceptar e imprimir en consola los argumentos y valores de retorno de cada syscall ejecutada, facilitando la auditoría del sistema.
- Generar Telemetría de Uso: Diseñar una estructura de datos persistente en el kernel para contabilizar las invocaciones a cada syscall y exponer estos datos mediante el comando scount.
- Mejorar la Legibilidad del Tiempo: Modificar la herramienta uptime para traducir los "ticks" del reloj de hardware a unidades de tiempo humanas (horas, minutos, segundos).

3. Descripción de modificación realizadas

Las modificaciones se dividen en tres módulos principales, afectando tanto al espacio de usuario como al espacio del núcleo

3.1. Instrumentación

Para lograr el rastreo de las llamadas al sistema, se realizaron modificaciones específicas en el archivo syscall.c, que es el encargado de despachar las interrupciones. El proceso se dividió en tres pasos lógicos:

A. Variable de Control Global

En el archivo syscall.c, se declaró una variable global llamada trace_on inicializada en 0.

- Función: Actúa como un interruptor. Si es 0, el sistema funciona normal; si es 1, el sistema entra en "modo espía".
- Interacción: Esta variable es modificada por una nueva syscall sys_trace que recibe el valor deseado (0 o 1) desde el comando de usuario.

B. Mapeo de Identificadores a Texto

El kernel originalmente solo entiende números (ej: 1 es fork, 5 es read). Para que la salida en pantalla sea legible, se agregaron dos arreglos auxiliares en syscall.c:

- syscallnames[]: Un arreglo de cadenas que asocia cada índice numérico con su nombre real (ej: [SYS_read] "read").
- syscall_argc[]: Un arreglo de enteros que indica cuántos argumentos requiere cada función. Esto es necesario para saber cuántos datos leer de la pila (stack) al momento de imprimir.

C. Modificación del Dispatcher

Se modificó la función principal void syscall(void), que se ejecuta cada vez que un programa solicita un servicio. Se agregó una condicional lógica que verifica si trace_on == 1.

El flujo implementado es el siguiente:

- Verificación: Si trace_on está activo, se procede a la captura de datos.
- Impresión del Nombre: Se usa el índice de la syscall (eax) para buscar su nombre en syscallnames e imprimirlo.
- Recuperación de Argumentos: Mediante un ciclo for y el arreglo syscall_argc, se extraen los argumentos de la pila del proceso usando la función auxiliar get_syscall_arg.
- Resultado: Finalmente, se imprime el valor de retorno que la syscall dejó en el registro %eax.

Esta implementación permite ver en tiempo real qué está pidiendo cada programa al sistema operativo, transformando números crudos en una bitácora legible.

3.2. Inspección de Procesos y Nuevos Comandos

En esta etapa, el objetivo fue crear herramientas que permitieran al usuario obtener información interna del sistema que normalmente está oculta. Se implementaron dos nuevos comandos: psmem para la gestión de procesos y memoria, y uptime para la gestión del tiempo.

A. Monitor de Procesos y Memoria

Para este comando, fue necesario acceder a la Tabla de Procesos (ptable), una estructura crítica del kernel donde se guarda la información de cada programa en ejecución.

- Nueva Syscall (sys_psmem):
    - Como un programa de usuario no puede leer directamente la memoria del kernel por seguridad, se creó una syscall intermediaria.
    - Esta función recorre el arreglo ptable.proc (que contiene hasta 64 procesos).
- Manejo de Concurrencia (Locking):
    - Para evitar errores de lectura (por ejemplo, leer un proceso justo cuando está siendo borrado), se utilizó el mecanismo de cerrojos del kernel (acquire(&ptable.lock) y release). Esto asegura que la "foto" que tomamos de los procesos sea consistente.
- Formato de Salida:
    - Originalmente, el estado del proceso es un número (ej: 4 es RUNNING). Se creó un arreglo de cadenas (states[]) para traducir estos números a palabras legibles (UNUSED, SLEEPING, RUNNING, etc.).
    - Se utilizó un formato de impresión con tabuladores y espacios de relleno (padding) para alinear perfectamente las columnas de PID, Estado, Memoria y Nombre, solucionando problemas de visualización en la consola QEMU.

B. Tiempo de Actividad

El sistema operativo mantiene un contador interno de "ticks" (interrupciones de reloj) desde que arranca. El desafío fue traducir este número crudo a algo útil para el humano.

- Obtención de datos: Se utilizó la syscall existente uptime(), que devuelve el número total de ticks.
- Lógica de Conversión:

- Sabiendo que la frecuencia del reloj en XV6 es de aproximadamente 100 Hz (100 ticks por segundo), se implementó una lógica aritmética en el espacio de usuario.
- Se dividieron los ticks entre 100 para obtener segundos totales, y luego mediante divisiones sucesivas y módulo (%), se desglosó el tiempo en Horas, Minutos y Segundos.
- Esto permite al administrador saber exactamente cuánto tiempo lleva encendido el servidor sin tener que hacer cálculos mentales.

3.3. Contador de Syscalls

El objetivo de este módulo fue dotar al sistema operativo de una memoria histórica ("telemetría") capaz de registrar cuántas veces se invoca cada servicio del núcleo. A diferencia del rastreo (trace), que es momentáneo y vuelca datos en pantalla, este contador es persistente y silencioso, acumulando métricas durante toda la sesión del sistema.

A. Estructura de Datos Persistente

Para almacenar los conteos, se realizaron modificaciones en el archivo syscall.c dentro del espacio del núcleo.

- Variable Global: Se definió un arreglo estático syscall_counts[30] inicializado en 0.
- Función: Cada índice del arreglo corresponde directamente al ID de una syscall (ej: la posición 1 guarda las veces que se llamó a fork). Al residir en el kernel, estos datos persisten aunque los programas de usuario terminen.

B. Interfaz de Lectura Segura

Dado que los programas de usuario (como la shell) no pueden leer directamente las variables del kernel por protección de memoria, se implementó una nueva llamada al sistema: sys_get_count.

- Validación: Recibe un ID como argumento, verifica que esté dentro de un rango válido (para evitar desbordamientos de buffer) y retorna el valor actual acumulado.

C. Instrumentación del Manejador

Se intervino nuevamente la función principal void syscall(void) en syscall.c para realizar el conteo automático.

- Lógica de Inyección: Justo antes de ejecutar la función solicitada, se insertó la instrucción de incremento: syscall_counts[num]++;.

- Independencia: Esta operación se realiza incondicionalmente, garantizando que las estadísticas sean precisas y se acumulen siempre, independientemente de si el modo de rastreo (trace) está activado o desactivado.

D. Herramienta de Visualización (scount)

Se desarrolló un nuevo programa de usuario llamado scount para consultar y presentar estos datos.

- Modos de Operación: El comando permite consultar una syscall específica o mostrar una tabla resumen completa.
- Corrección Visual: Durante el desarrollo, se detectó un problema de desalineación en la consola VGA de QEMU al usar tabuladores estándar. Para solucionarlo, se implementó una técnica de relleno de espacios (padding) en los nombres de las funciones (ej: "read "), asegurando que todas las cadenas tengan la misma longitud y que las columnas de la tabla se mantengan perfectamente alineadas.

3.4. Descripción de modificaciones realizadas

Listado Extendido de Archivos (lsx) Como parte de los comandos de usuario adicionales sugeridos en la evaluación, se implementó la herramienta lsx. A diferencia del comando ls nativo de XV6, que solo lista los nombres de los archivos, lsx permite visualizar metadatos críticos del sistema de archivos almacenados en los inodos.

Detalles de Implementación: Esta herramienta se desarrolló enteramente en el espacio de usuario, por lo que no requirió nuevas llamadas al sistema (syscalls). Utiliza la syscall estándar stat para interrogar al sistema de archivos.

- Recuperación de Metadatos: Por cada entrada en un directorio, se invoca a stat(), lo que llena una estructura con datos como el número de inodo (st.ino), el número de enlaces duros (st.nlink), el tamaño en bytes (st.size) y el tipo de archivo.
- Formateo: Se implementó una lógica de impresión con encabezados y anchos de columna fijos (NAME, TYPE, INO, NLINK, SIZE) para garantizar una tabla legible en la consola.

4. Fragmentos Relevantes de Código

4.1. Modificación en syscall.c (Instrumentación y Conteo)

Se modificó la función central de despacho para integrar tanto el contador de estadísticas como el rastreo en tiempo real. La lógica asegura que el conteo sea constante, mientras que la impresión en pantalla depende de la activación del usuario.

```
struct proc *curproc = myproc();

  // Se obtiene el número de syscall desde el registro eax
del proceso
  num = curproc->tf->eax;

  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {

    // TELEMETRIA
    // Se incrementa el contador global antes de ejecutar la
función.
    // Esto asegura el registro histórico aunque el trace
esté apagado.
    syscall_counts[num]++;

    // Ejecución de la syscall real solicitada
    curproc->tf->eax = syscalls[num]();

    // INSTRUMENTACION
    // Verificación de la bandera 'trace_on'.
    // Si está activa (1), se procede a imprimir los
detalles.
    if (trace_on == 1) {
        // Se imprime el nombre usando el arreglo de mapeo
creado
        cprintf("%s(", syscallnames[num]);

        int count = syscall_argc[num];
        int i;
        // Ciclo para extraer y mostrar los argumentos desde
el stack
        for(i = 0; i < count; i++){
            // Función auxiliar para lectura segura de
memoria
            int arg = get_syscall_arg(curproc, i);
            cprintf("%d", arg);

            if(i < count - 1) cprintf(", ");
        }
        // Finalmente se muestra el valor de retorno
        cprintf(") -> %d\n", curproc->tf->eax);
    }
  }
}
```

4.2. Implementación de psmem en proc.c

Para mostrar la información de la memoria, se accede a la tabla de procesos (ptable). Es crítico utilizar mecanismos de bloqueo (locks) para garantizar que la lectura sea segura en un entorno concurrente.

```
void
psmem(void)
{
  struct proc *p;
  // Arreglo estático para traducir los estados numéricos a
texto.
  // Se agregan espacios al final para mantener la alineación
visual.
  static char *states[] = {
  [UNUSED]    "UNUSED  ",
  [SLEEPING]  "SLEEPING",
  [RUNNABLE]  "RUNNABLE",
  [RUNNING]   "RUNNING ",
  [ZOMBIE]    "ZOMBIE  "
  };

  // Se adquiere el candado lock de la tabla.
  // Esto evita condiciones de carrera si un proceso cambia
de estado mientras se lee.
  acquire(&ptable.lock);

  cprintf("PID\tEstado  \tMemoria \tNombre\n");

cprintf("-------------------------------------------------\
n");

  // Recorrido de los slots de procesos
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;

    // Se imprime solo si el estado es válido
    if(p->state >= 0 && p->state < NELEM(states) &&
states[p->state]){
      // Formato con tabuladores (\t) para separar columnas
      cprintf("%d\t%s\t%d\t\t%s\n", p->pid, states[p->state],
p->sz, p->name);
    }
  }
```

```
  // Se libera el candado para no bloquear el sistema
  release(&ptable.lock);
}
```

4.3. Lógica del comando de usuario scount.c

Este fragmento muestra cómo el programa de usuario interactúa con la nueva syscall sys_get_count. El código recorre los identificadores disponibles y filtra los resultados para mostrar únicamente las funciones que han tenido actividad.

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

    printf(1, "ID\tNombre      \tInvocaciones\n");
    printf(1, "-----------------------------------\n");

    int i;
    // Ciclo principal: Se iteran los IDs de las syscalls
monitoreadas (1 a 24)
    for(i = 1; i <= 24; i++){

        // CONSULTA AL KERNEL
        // Se llama a la nueva syscall 'get_count' pasando el
ID 'i'.
        // Esto trae el valor actual desde el arreglo
protegido del núcleo.
        int conteo = get_count(i);

        // FILTRADO DE DATOS
        // Solo se muestran en pantalla las syscalls que se
han usado al menos una vez (conteo > 0).
        // Esto evita llenar la consola con información
irrelevante (ceros).
        if(conteo > 0){
            // Se imprime la fila con el formato: ID - Nombre
- Cantidad
            printf(1, "%d\t%s\t%d\n", i, nombres_syscalls[i],
conteo);
        }
    }
    exit();
}
```

4.4. Cálculo de tiempo en uptime.c

Dado que el kernel retorna "ticks" de reloj, la lógica de conversión a tiempo humano se implementó en el espacio de usuario mediante operaciones aritméticas.

```
int
main(int argc, char *argv[])
{
  // Obtención de los ticks crudos del sistema
  int ticks = uptime();

  // Conversión: Se asume frecuencia de 100Hz (100 ticks = 1
seg)
  int total_segundos = ticks / 100;

  // Desglose matemático en horas, minutos y segundos
  int minutos = total_segundos / 60;
  int segundos = total_segundos % 60;
  int horas = minutos / 60;

  minutos = minutos % 60; // Residuo de minutos

  // Impresión formateada para el usuario
  printf(1, "Tiempo Activo : %d h, %d min, %d seg\n", horas,
minutos, segundos);
  exit();
}
```

4.5. Lógica de `lsx.c` (Espacio de Usuario)

```
void lsx(char *path) {
  // ... (apertura del directorio)

  // Impresión del encabezado de columnas
  printf(1, "NAME            TYPE  INO   NLINK SIZE\n");
  printf(1, "-------------------------------------\n");

  // Bucle de lectura de entradas del directorio
  while(read(fd, &de, sizeof(de)) == sizeof(de)){
    if(de.inum == 0) continue;

    // Se obtiene la información detallada del inodo usando
stat()
```

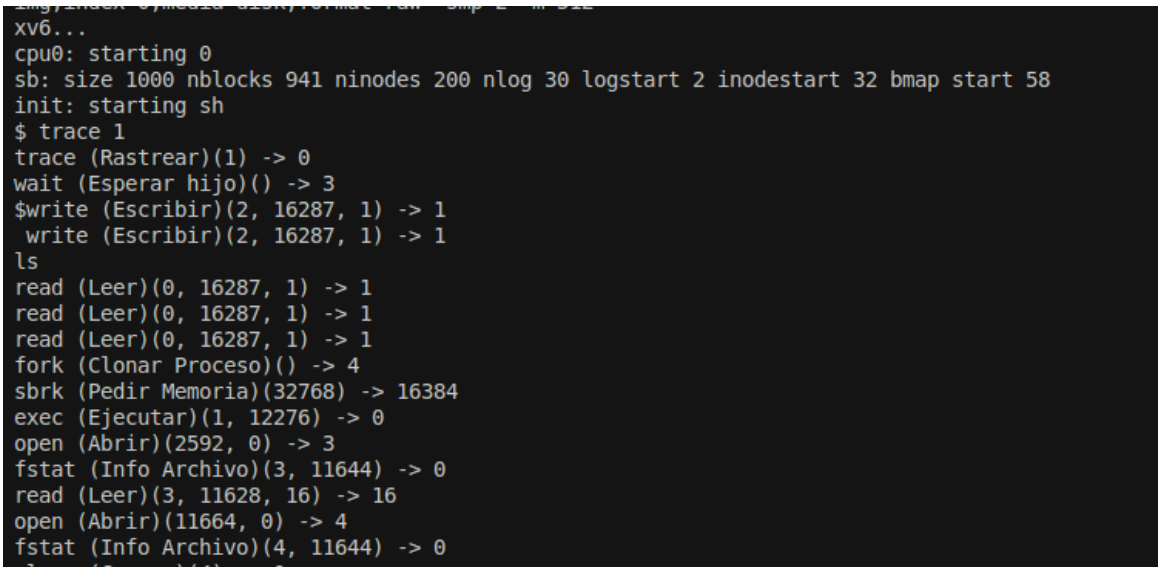
```
    if(stat(buf, &st) < 0){
      printf(2, "lsx: cannot stat %s\n", buf);
      continue;
    }

    // Impresión formateada: Nombre, Tipo, Inodo, Enlaces,
Tamaño
    printf(1, "%s %4s %5d %5d %5d\n",
           fmtname(buf), typename(st.type), st.ino, st.nlink,
st.size);
  }
  close(fd);
}
```

5. Resultados de pruebas

5.1. Prueba de Instrumentación (Trace)

Objetivo: Verificar la capacidad del núcleo para interceptar llamadas al sistema, decodificar sus nombres y argumentos, e imprimirlos en consola antes de su ejecución.

Figura 1. Ejecución del comando trace 1 seguido de una interacción con la shell.



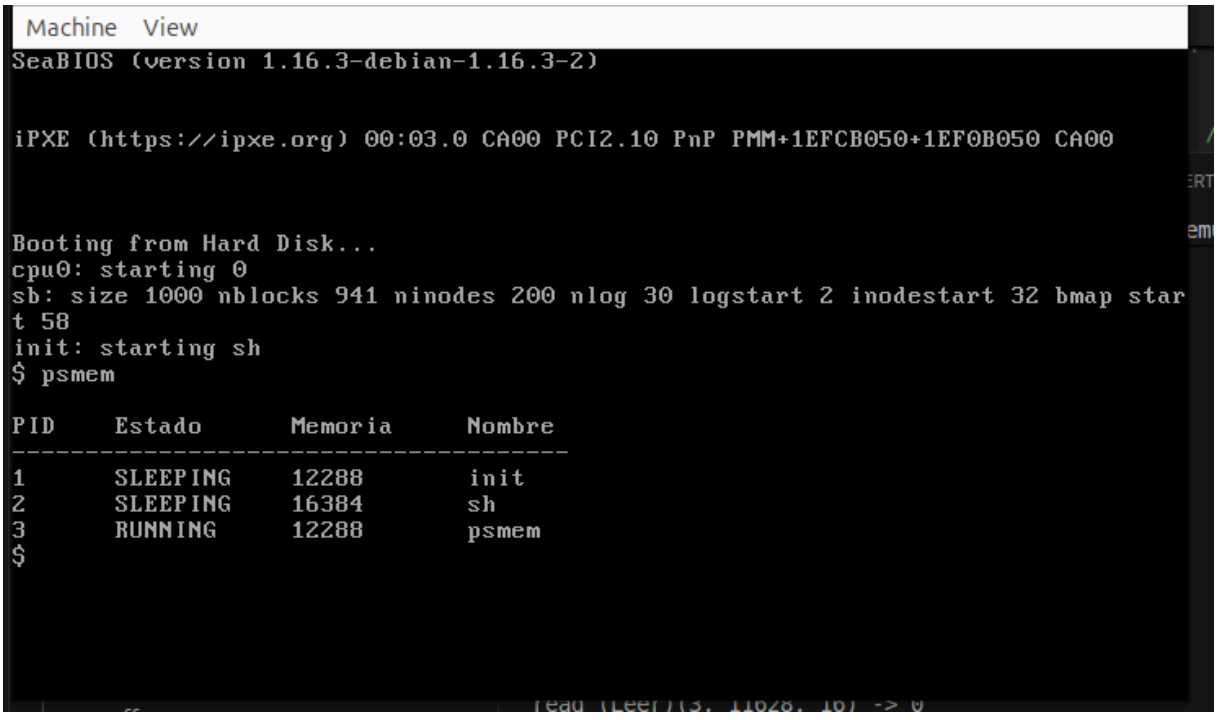
Análisis de Resultados: Como se aprecia en la Figura 1, al activar el rastreo con el comando trace 1, el sistema comienza a reportar inmediatamente la actividad interna.

- Se observa la propia llamada de activación: trace(Rastrear)(1) -> 0.
- Al escribir el comando ls en la terminal, se registran las llamadas read (leyendo el teclado) y write (haciendo eco en pantalla) de la shell.
- Es crucial notar la secuencia de creación del nuevo proceso ls: se detecta un fork(Clonar Proceso), seguido de un sbrk(Pedir Memoria) y finalmente el exec(Ejecutar).
- Posteriormente, se ven las llamadas open y read que realiza el comando ls para leer el directorio. Esta prueba confirma que el manejador de syscalls modificado está interceptando y mapeando correctamente los IDs numéricos a nombres descriptivos (ej. "Clonar Proceso").

5.2. Prueba de Inspección de Procesos (Psmem)

Objetivo: Validar la lectura segura de la tabla de procesos (ptable) y la correcta traducción y formateo de los estados internos.

Figura 2. Visualización de la tabla de procesos activa con `psmem`.



Análisis de Resultados: La Figura 2 muestra el estado del sistema en un momento dado. La herramienta psmem identifica correctamente los tres procesos existentes:

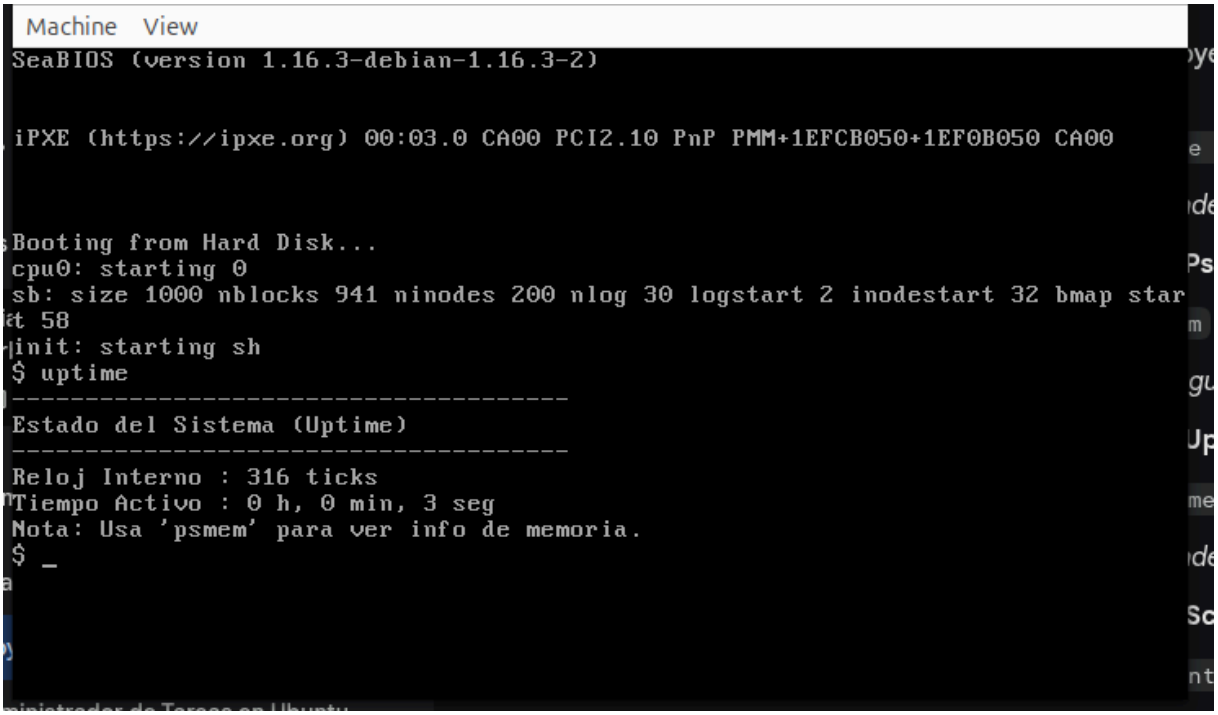
- PID 1 (init) y PID 2 (sh): Aparecen en estado SLEEPING, lo cual es consistente ya que están esperando eventos (como entrada del usuario).
- PID 3 (psmem): Aparece en estado RUNNING, lo que es correcto ya que es el proceso que se está ejecutando actualmente para mostrar la tabla. La prueba

demuestra que la traducción de estados numéricos a texto funciona y que la alineación de columnas mediante padding es efectiva para una visualización clara.

5.3. Prueba de Conversión de Tiempo (Uptime)

Objetivo: Verificar la conversión aritmética en el espacio de usuario de los "ticks" del reloj del hardware a un formato de tiempo legible.

Figura 3. Salida del comando mejorado uptime.

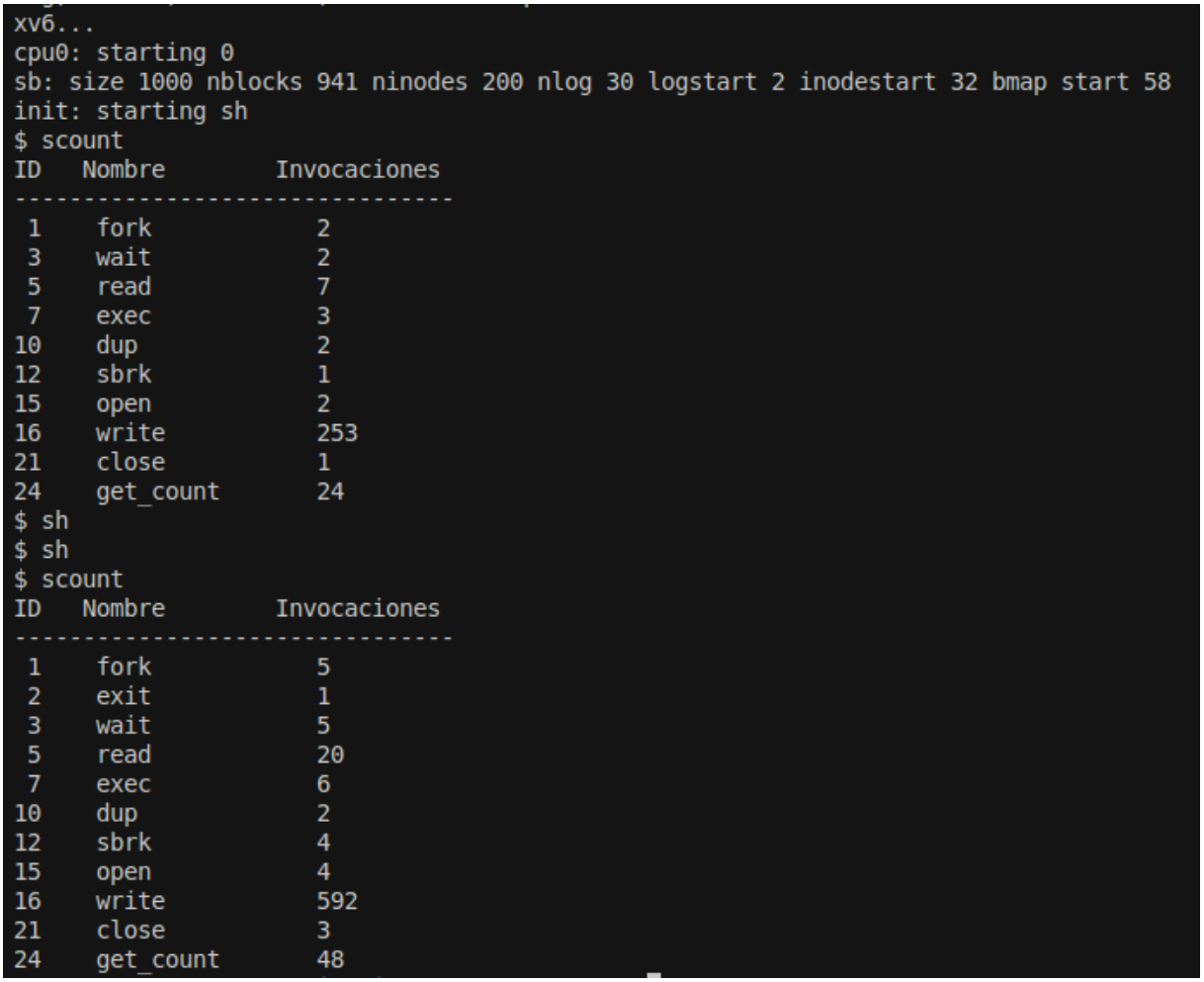


Análisis de Resultados: En la Figura 3, el comando uptime muestra que el reloj interno del kernel ha registrado 316 ticks desde el arranque. La lógica de conversión implementada en el programa de usuario traduce esto correctamente a 3 segundos de tiempo activo (asumiendo la frecuencia estándar de 100Hz de XV6). Esto valida que la abstracción del hardware se está realizando correctamente para el usuario final.

5.4. Prueba de Telemetría Acumulada (Scount)

Objetivo: Comprobar la persistencia del contador de syscalls en el kernel y su actualización tras la actividad del sistema.

Figura 4. Comparativa de estadísticas de uso (scount) antes y después de ejecutar comandos.



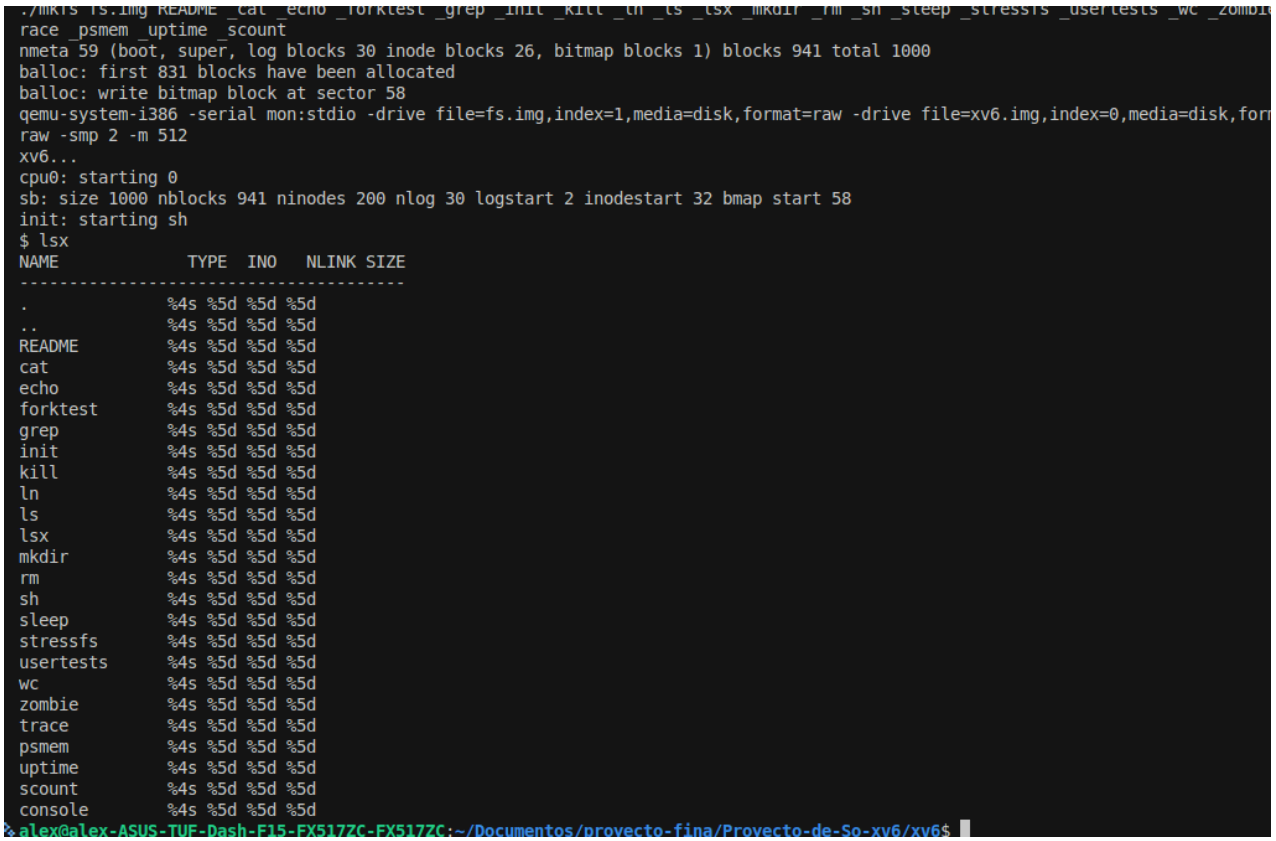
Análisis de Resultados: La Figura 4 demuestra la naturaleza persistente del contador:

- Tabla Superior (Estado Inicial): Se muestra un conteo inicial donde destacan write (253 invocaciones) y un uso moderado de fork, wait y exec.
- Actividad Intermedia: El usuario ejecuta el comando sh (nueva shell) en dos ocasiones. Esto debería generar nuevas llamadas de creación de procesos.
- Tabla Inferior (Estado Final): Al ejecutar scount nuevamente, se reflejan los cambios:
    - fork, exec y wait: Han aumentado proporcionalmente a la creación de las nuevas shells (ej. fork pasó de 2 a 5).
    - read y write: Muestran un aumento significativo debido a la interacción con las nuevas terminales. Esto confirma que el arreglo global syscall_counts dentro del núcleo está registrando de forma continua y acumulativa cada interrupción de software, independientemente de los procesos que las generen.

5.5. Prueba de Información de Archivos (lsx)

bjetivo: Verificar la capacidad de la herramienta para acceder a la tabla de inodos y mostrar atributos extendidos de los archivos (tamaño, enlaces e identificadores).

Figura 5. Ejecución del comando lsx comparando la información de archivos del sistema.



Análisis de Resultados: Al ejecutar el comando lsx, se despliega una tabla detallada que valida el correcto funcionamiento de la lectura de metadatos:

- Identificación (INO): Se observa que cada archivo posee un número de inodo único (columna INO), lo cual es la identificación interna del sistema de archivos.
- Tipos de Archivo: La columna TYPE distingue correctamente entre directorios (DIR), archivos regulares (FILE) y dispositivos (DEV, como la consola).
- Integridad de Datos: La columna SIZE reporta el tamaño en bytes, lo cual es fundamental para verificar el uso de disco, información que el comando ls original omitía.
- Enlaces (NLINK): Se visualiza el conteo de referencias a cada archivo, útil para entender la estructura del sistema de archivos (por ejemplo, los directorios suelen tener múltiples enlaces debido a . y ..).

6. Conclusiones

- Visibilidad del Núcleo (Kernel): A través de la implementación de trace y scount, se logró transformar la visión del sistema operativo de una "caja negra" a una "caja blanca". Se comprobó experimentalmente que cada interacción del usuario (incluso un simple ls) desencadena una serie compleja de interrupciones y cambios de modo (usuario a kernel), validando la teoría de llamadas al sistema.

- Gestión de Procesos y Planificación: La herramienta psmem permitió visualizar dinámicamente el ciclo de vida de los procesos. Al observar procesos en estado SLEEPING (como init y sh) coexistiendo con procesos RUNNING, se verificó la eficacia del planificador (scheduler) de XV6 para administrar la CPU y la memoria sin que los procesos inactivos consuman recursos de procesamiento innecesarios.

- Persistencia de Datos en el Kernel: Con el desarrollo de scount, se demostró que el núcleo puede mantener estructuras de datos (como el arreglo de contadores) que sobreviven a la terminación de los procesos de usuario. Esto es fundamental para entender cómo el SO gestiona la telemetría y la auditoría global del sistema.

- Abstracción del Sistema de Archivos: La implementación de lsx en el espacio de usuario evidenció la separación lógica entre el nombre de un archivo y sus metadatos (inodo). Se comprendió que atributos críticos como el tamaño y los enlaces no residen en el directorio, sino en la tabla de inodos, accesible mediante la syscall stat.

- Desarrollo a Bajo Nivel: La manipulación directa de la memoria de video (para alineación de tablas) y la aritmética de punteros en C reforzó las competencias de programación de sistemas, destacando la importancia del manejo cuidadoso de la memoria y la sincronización (uso de locks) para evitar inconsistencias en la tabla de procesos.

7. Anexos

ANEXO A: Enlace al Repositorio de Código

- *Descripción:* Link a GitHub/GitLab con el historial de commits que demuestra el trabajo colaborativo.
- *https://github.com/JheralMaquera/Proyecto-de-So-xv6.git*