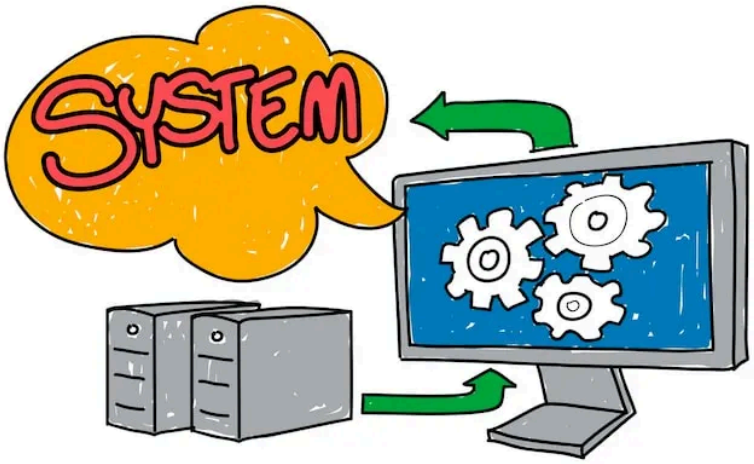
UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN
INGENIERIA EN INFORMATICA Y SISTEMAS



Curso
Sistemas Operativos

Ciclo
Sexto Ciclo

Proyecto
“Sistema Operativo UNIX (xv6)”

Colaboradores

Alexis Ricardo Erik Ricardo Condori 2023-119013

Yeferson Guerra Quispe_2023-119021

Tacna - Perú

2025

1. Introducción

Los sistemas operativos actúan como intermediarios fundamentales entre el hardware y las aplicaciones de usuario, gestionando recursos limitados mediante mecanismos de multiplexación y aislamiento. El presente proyecto se centra en la exploración y modificación de XV6, una reimplementación moderna del sistema operativo UNIX Versión 6 desarrollada por el MIT para fines educativos.

XV6, al ser un sistema operativo de núcleo monolítico con una base de código compacta, permite una comprensión profunda de conceptos clave como la gestión de procesos, la memoria virtual y las llamadas al sistema (syscalls). En este informe se documentan las extensiones realizadas al núcleo de XV6, las cuales incluyen la instrumentación de llamadas al sistema para su trazabilidad, la implementación de nuevas herramientas de monitoreo de procesos y memoria, y la creación de estadísticas de uso del sistema. Estas modificaciones buscan aplicar de manera práctica los conceptos teóricos abordados en la Unidad II del curso.

2. Objetivos

2.1. Objetivo General

Interactuar con un sistema operativo real tipo UNIX (XV6) para diseñar y desarrollar extensiones a nivel de núcleo y espacio de usuario, analizando su impacto en la gestión del sistema.

2.2. Objetivos Específicos

- **Relacionar la teoría** de procesos, estados de planificación y llamadas al sistema con su implementación en código C dentro de XV6.
- **Instrumentar el núcleo** para interceptar y visualizar la ejecución de llamadas al sistema en tiempo real.
- **Desarrollar nuevos comandos de usuario** (uptime, psmem, trace) que accedan a estructuras de datos internas del kernel (ptable) para exponer información sobre el estado del sistema. Ademas
- **Implementar un mecanismo de conteo de invocaciones** de llamadas al sistema, mediante la instrumentación del manejador de syscalls y la creación de una nueva syscall (`getsyscount`) accesible desde el espacio de usuario con el comando syscount <parámetro>, syscount sin parámetro.
- **Aplicar buenas prácticas** de programación en sistemas, incluyendo la modificación segura de estructuras del kernel y el uso de control de versiones.

3. Descripción De Las Modificaciones Realizadas

El desarrollo se dividió en tres componentes principales, abarcando modificaciones en archivos de cabecera (.h), código fuente del núcleo (.c) y programas de usuario.

3.1. Instrumentación de Llamadas al Sistema (Entregable 1)

Se modificó el mecanismo de despacho de llamadas al sistema en syscall.c para permitir el rastreo (tracing) de su ejecución.

- **Mapeo de Nombres:** Se creó un arreglo syscallnames[] en syscall.c para traducir los números de interrupción (ej. 1) a nombres legibles (ej. "fork").
- **Interruptor de Rastreo:** Se implementó una variable global trace_active y una nueva syscall sys_trace (número 22) que permite al usuario activar o desactivar la impresión de mensajes en consola.
- **Filtrado:** Se añadió lógica para omitir la syscall write del rastreo para evitar bucles de retroalimentación visual en la consola.

3.2. Comandos de Usuario Relacionados con la Unidad II (Entregable 2)

Se implementaron dos comandos para inspeccionar el estado del sistema:

- **Comando uptime extendido:**
  - Utiliza la syscall existente uptime() para obtener los *ticks* del reloj.
  - Se creó una nueva syscall getprocs() respaldada por la función get_process_count() en proc.c, la cual itera sobre la tabla de procesos (ptable) contando aquellos cuyo estado no sea UNUSED.
- **Comando psmem (Inspección de memoria):**
  - Se definió una estructura struct uproc para transportar datos (PID, estado, nombre, memoria) del kernel al usuario.
  - Se implementó getprocsinfo() en el kernel, que copia de manera segura la información de la ptable a la estructura de usuario.
  - El comando psmem formatea estos datos mostrando una tabla con el estado actual (RUNNING, SLEEPING, etc.) y el consumo de memoria en bytes.

3.3. Contador de Invocaciones (Entregable 3)

Se implementó un mecanismo en el kernel para contabilizar cuántas veces se ejecuta cada llamada al sistema y permitir que los programas de usuario puedan consultar esta información.

Arreglo de Contadores:

Se creó un arreglo global syscall_count en el kernel donde cada posición corresponde a una syscall específica. Cada vez que se ejecuta una syscall válida, se incrementa el contador correspondiente.

Nueva Syscall getsyscount:

Se implementó una nueva llamada al sistema que permite al usuario:

- Consultar el número de invocaciones de una syscall específica proporcionando su identificador.

- Obtener un valor de error si el identificador es inválido o no se puede leer correctamente.

Comando de Usuario syscount:

Se desarrolló un programa de usuario que permite:

1. Consultar una syscall específica:

- Si recibe como parámetro un identificador de syscall, muestra únicamente el número de invocaciones de esa llamada: syscount <parametro>.

2. Consultar todas las syscalls:

   - Si no recibe parámetros, muestra un resumen con todas las llamadas al sistema y su número de invocaciones, usando nombres legibles para cada syscall: syscount

4. Fragmentos Relevantes De Código Comentado

A continuación, se presentan las secciones críticas del código desarrollado en el contador de invocaciones.

Manejador de syscalls:

```
// Manejador principal de syscalls
void
syscall(void)
{
  int num;
  struct proc *p = myproc();   // Obtiene un puntero al proceso
actual

  num = p->tf->eax;            // El número de syscall se pasa en
el registro eax
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {  //
Verifica que la syscall exista

    syscall_count[num]++;       // Incrementa el contador de
llamadas a esta syscall

    if(trace_active){           // Si el tracing está activo,
imprime información de depuración
      cprintf("syscall %s(", syscall_names[num]);

      if(num == SYS_read || num == SYS_write){   // Si es read o
write
        int fd, n;
        argint(0, &fd);       // Obtiene el primer argumento (file
descriptor)
        argint(2, &n);        // Obtiene el tercer argumento
(número de bytes)
```

```
        cprintf("fd=%d, n=%d", fd, n);
      }
      else if(num == SYS_open){  // Si es open
        char *path;
        int mode;
        argstr(0, &path);       // Obtiene el primer argumento:
path del archivo
        argint(1, &mode);       // Obtiene el segundo argumento:
modo de apertura
        cprintf("path=%p, mode=%d", path, mode);
      }
      else if(num == SYS_exec){  // Si es exec
        char *path;
        argstr(0, &path);       // Obtiene el primer argumento:
path del programa
        cprintf("path=%p", path);
      }

      cprintf(")\n");           // Cierra el paréntesis en la
impresión
    }

    p->tf->eax = syscalls[num]();  // Llama a la función real de
la syscall y guarda el resultado en eax
  } else {
    // Si el número de syscall no es válido, imprime un error
    cprintf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->tf->eax = -1;           // Devuelve -1 al proceso
  }
}
```

Archivo syscall.h:

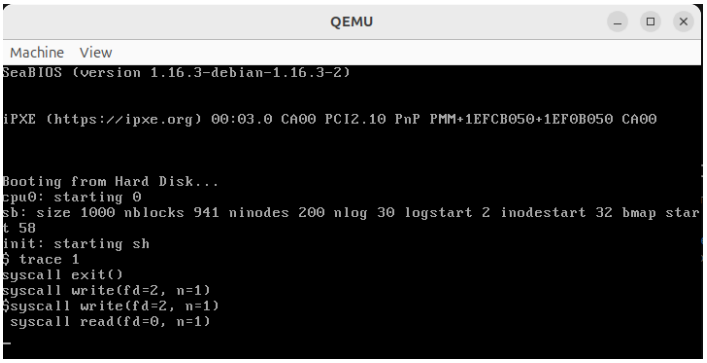
```
// System call numbers
// Definiciones de constantes que representan cada syscall en xv6
// Cada número corresponde a una función del kernel que puede ser
llamada
// desde un programa de usuario mediante la interfaz de syscalls
// NSYSCALLS indica el total de syscalls disponibles
#define SYS_fork    1
#define SYS_exit    2
```

```
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_trace  22
#define SYS_getprocs 23
#define SYS_getprocsinfo 24
#define SYS_getsyscount 25
#define NSYSCALLS 26  // Total de syscalls disponibles
```

5. Resultados De Pruebas

Prueba 1: Instrumentación de Syscalls

Descripción: Se activa el rastreo con trace 1 y se ejecuta un comando ls. Se observa cómo el sistema operativo reporta las llamadas internas como open, fstat y close.



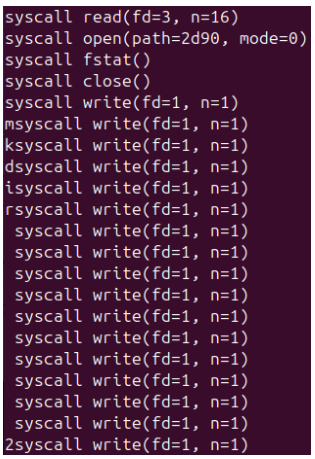


Fig 1. Consola QEMU mostrando la traza de ejecución.

Prueba 2: Comando Uptime Extendido

Descripción: Ejecución del comando uptime mostrando el tiempo transcurrido desde el arranque y el número de procesos concurrentes en la tabla.

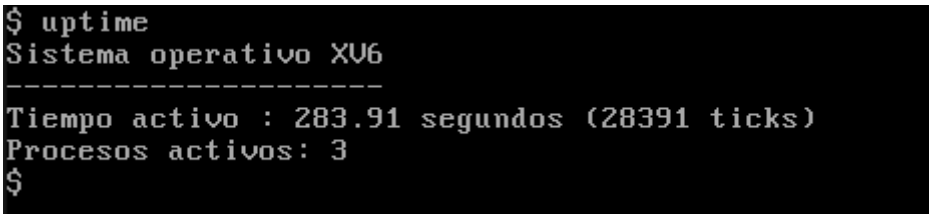


Fig 2. Salida del comando uptime con conteo de procesos.

Prueba 3: Inspección de Memoria (psmem)

Descripción: Listado de procesos activos. Se puede identificar el proceso init (PID 1) y sh (PID 2) en estado SLEEPING, y el propio comando psmem en estado RUNNING.

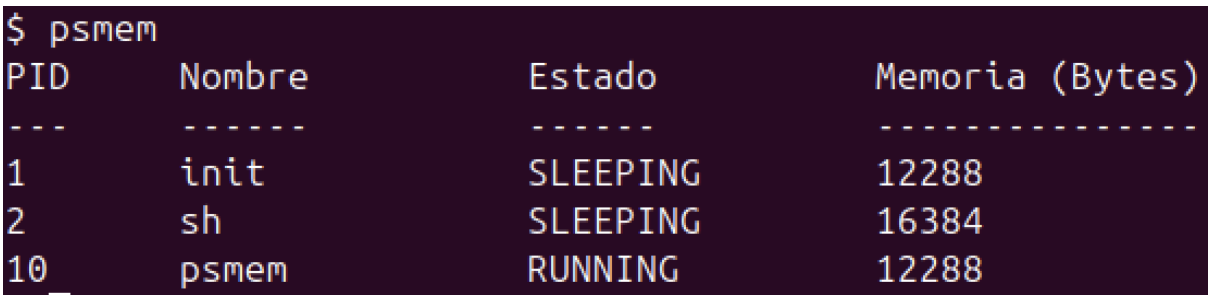


Fig 3. Salida del comando psmem con los PID, Nombre, Estado y Uso de Memoria.

Prueba 4: Consulta individual de syscalls (syscount)

Descripción:
Se ejecuta el comando syscount con un identificador de syscall como parámetro. El sistema muestra cuántas veces ha sido invocada dicha llamada desde el arranque.

Fig 4. Salida del comando syscount <parámetro> para mostrar cantidad de llamadas de la syscall.

Prueba 5: Resumen general de syscalls (syscount)

Descripción:
Se ejecuta el comando syscount sin parámetros. El sistema presenta un resumen con todas las llamadas al sistema y su número de invocaciones.

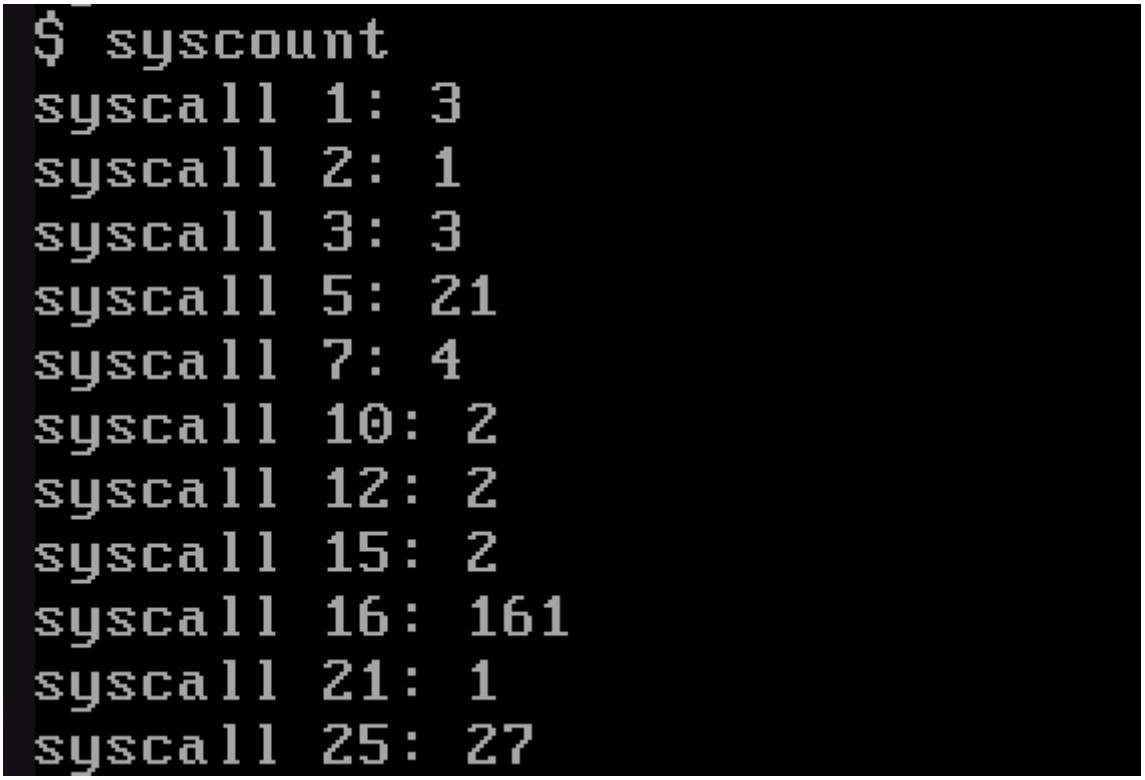


Fig 5. Muestra tabla de resumen las syscalls y su cantidad de llamadas.

6. Conclusiones Técnicas

1. Interacción Usuario-Núcleo: La implementación de nuevas llamadas al sistema (como sys_trace o sys_getprocs) demostró empíricamente el mecanismo de protección del procesador. El usuario no puede acceder directamente a la memoria del kernel (ptable); debe solicitarlo a través de una interrupción de software (trap), validando el modelo de protección de anillo estudiado en la Unidad II.
2. Gestión de Procesos: A través del comando psmem, se verificó la existencia del Bloque de Control de Procesos (PCB) en XV6 (struct proc). Se observó cómo los procesos pasan la mayor parte del tiempo en estado SLEEPING esperando eventos (E/S), lo cual justifica la necesidad de algoritmos de planificación para maximizar el uso de la CPU.
3. Sincronización: La manipulación de la tabla global de procesos requirió el uso de primitivas de sincronización (acquire y release de ptable.lock). Omitir esto podría causar inconsistencias si el planificador intenta cambiar el estado de un proceso mientras el comando psmem lo está leyendo, destacando la importancia de la exclusión mutua en sistemas operativos concurrentes.
4. La instrumentación del manejador de llamadas al sistema permite registrar de forma centralizada el número de invocaciones de cada syscall. El comando syscount evidenció cómo se muestra la cantidad de llamadas de cada syscall y una tabla de resumen de todas las syscalls y su cantidad de llamadas.

7. Referencias Bibliográficas

- [1] "Xv6, a Simple Unix-like Teaching Operating System," MIT PDOS, 2014. [En línea]. Disponible: http://pdos.csail.mit.edu/6.828/2014/xv6.html. [Accedido: 15-Dic-2025].
- [2] R. Cox, F. Kaashoek y R. Morris, "xv6: a simple, Unix-like teaching operating system" (book-rev8.pdf), MIT, 2014.
- [3] Código fuente de XV6 (xv6.pdf), MIT.
- [4] Documento de Evaluación de Producto Unidad 02, Curso Sistemas Operativos, UNJBG, 2025.