Sistemas Operativos

Departamento de Computación – FCEyN – UBA
Segundo cuatrimestre de 2018

Nombre y apellido: _____
Nº orden: 7 L.U.: _____ Cant. hojas: 4

1	2	3	4	Nota
B-	R	B	B	A

Segundo parcial – 15/11 - 2do. cuatrimestre de 2018

CARA

ACLARACIONES: 1) **Numere** las hojas entregadas. Esta hoja se entrega y es la hoja cero. Complete en la primera hoja la cantidad total de hojas entregadas (sin contar el enunciado). 2) Realice cada ejercicio en **hojas separadas** y escriba **nombre, apellido y L.U. en cada una**. 3) Cada ejercicio se califica con **Bien**, **Regular** o **Mal**. La división de los ejercicios en incisos es meramente orientativa. Los ejercicios se califican globalmente. El parcial se aprueba con 2 ejercicios bien y a lo sumo 1 mal/incompleto. 4) El parcial **NO** es a libro abierto. 5) **Justifique adecuadamente cada una de sus respuestas.**

Ejercicio 1.

Se desea crear un nuevo filesystem llamado YTFS (YouTubeFileSystem) que permita buscar y reproducir videos de Youtube utilizando archivos. Algunas de las características principales de este FS son:

a) **Búsqueda:** La búsqueda de videos se podrá realizar mediante la creación de directorios, cuyo nombre será las palabras claves a buscar. Por ejemplo, al ejecutar `mkdir "dc exactas uba"`, se creara el directorio correspondiente y dentro de él un archivo por cada video que concuerde con la búsqueda.
b) **Reproducción:** Los resultados podrán usarse como archivos de video normales. Por ejemplo:

```
mkdir "dc exactas uba"
cd dc\ exactas\ uba
mplayer "Bienvenida a los ingresantes 2017.mp4"
```

Durante la fase de diseño preliminar, les ingenieres a cargo del proyecto discuten acaloradamente la conveniencia de adoptar un enfoque inspirado en bloques contiguos, FAT o inodos. Indicar cuál de las tres opciones recomendaría, y por qué, para cada uno de los siguientes requerimientos. Considere cada uno de los requerimientos por separado a la hora de hacer el análisis:

a) Los archivos que resulten de una búsqueda inician vacíos, y sólo empiezan a descargarse al intentar reproducirlos. Por tal motivo, es importante que agregar datos a un archivo sea lo más rápido posible.
b) Dado que los videos pueden ser muy pesados, es importante que el tamaño máximo sólo esté limitado por el tamaño del disco.
c) Dado que el usuario puede querer ver un video arrancando desde una posición que no sea el principio, es importante que el acceso a una parte arbitraria del video sea rápida.
d) Por último, se quiere que la lectura secuencial de los archivos sea lo más rápida posible, para que no afecte la reproducción de los videos.

Ejercicio 2.

Se desea escribir un *driver* para la famosa impresora *Headaches Persistent*. El manual del controlador nos dice que para comenzar una impresión, se debe:

- Ingresar en el registro de 32 bits `LOC_TEXT_POINTER` la dirección de memoria dónde empieza el buffer que contiene el *string* a imprimir.
- Ingresar en el registro de 32 bits `LOC_TEXT_SIZE` la cantidad de caracteres que se deben leer del buffer.
- Colocar la constante `START` en el registro `LOC_CTRL`.

En este momento, si la impresora detecta que no hay suficiente tinta para comenzar, escribirá *rápidamente* el valor `LOW_INK` en el registro `LOC_CTRL` y el valor `READY` en el registro `LOC_STATUS`. Caso contrario, la impresora comenzará la impresión, escribiendo el valor `PRINTING` en el registro `LOC_CTRL` y el valor `BUSY` en el registro `LOC_STATUS`. Al terminar, la impresora escribirá el valor `FINISHED` en el registro `LOC_CTRL` y el valor `READY` en el registro `LOC_STATUS`.

Un problema a tener en cuenta es que, por la mala calidad del *hardware*, éstas impresoras suelen detectar erróneamente bajo nivel de tinta. Sin embargo, el fabricante nos asegura en el manual que "alcanza con probar hasta 5 veces para saber con certeza si hay o no nivel bajo de tinta".

El controlador soporta además el uso de las interrupciones: `HP_LOW_INK_INT`, que se lanza cuando la impresora detecta que hay nivel bajo de tinta, y `HP_FINISHED_INT`, que se lanza al terminar una impresión.

Se pide implementar las funciones `int driver_init()`, `int driver_remove()` y `int driver_write(void* data, int size)` del *driver*. Piense cuidadosamente si conviene utilizar *polling*, *interrupciones* o una mezcla de ambos. Justifique la elección. Además, debe asegurese de que el código no cause condiciones de carrera. Las impresiones deberán ser bloqueantes. No hace falta que implemente *spooling*.

Ejercicio 3.

Asuma que Bob crea un programa para utilizar el *driver* del Ejercicio 2:

```
int main (int argc, char *argv[]) {
  int impresora_fd = open("/dev/hp", "w");
  imprimir(impresora_fd);
  close(impresora_fd);
}
int imprimir (int impresora_fd) {
  char texto[250];
  printf("Ingrese el string a imprimir\n");
  gets(texto);
  resultado = write(impresora_fd, texto);
  if (resultado == 1) {
      printf("La impresión se realizó con éxito\n");
  } else {
      printf("Hubo un error al procesar la impresión\n");
  }
}
```

a) Explique qué problema de seguridad tiene el programa que escribió Bob, y cómo debería explotarlo un atacante.

b) Si este programa corriera con permisos de administrador, ¿Qué podría hacer un atacante que logra explotar la vulnerabilidad?

c) Describa algún mecanismo de protección que se podría implementar para evitar dicha vulnerabilidad.

Ejercicio 4.

Un sistema distribuido tiene cuatro nodos: A, B, C y D, conectados de la siguiente forma:

Responda las siguiente preguntas justificando:

a) Dados los siguientes escenarios:

- El nodo B se cae.
- El enlace entre A y B se cae.
- B está muy sobrecargado, y su tiempo de respuesta es 100 veces mayor a lo normal.

¿Puede A discernir entre cada uno de ellos?

b) Si A recibe un mensaje de D, a través de B, ¿Se puede asumir que D no está caído?

c) Si B recibe un mensaje de A y uno de C, ¿Se puede saber si A envío su mensaje antes que C, o viceversa? ¿Por qué?

Sistemas Operativos · 2do parcial
N.º 7

Ejercicio 1

a) Para que agregar datos a un archivo sea muy rápido, hay dos enfoques que pueden resultar útiles:

- bloques contiguos(*), ya que si hay memoria sin asignar después del último bloque del archivo, escribir un nuevo bloque después de él es de complejidad temporal O(1). (¡en escrituras a disco!)
- inodos, ya que podemos calcular en qué posición del "arreglo" de bloques del inodo correspondiente al archivo irían los bloques nuevos, escribir los datos en un bloque libre y agregar la dirección de dicho bloque. Pseudocódigo:

```
                                inode
void add_new_block ( [illegible] file, void * block_data) {
    blockaddr where_to_write = get_free_blockaddr();
    uint block_num = file.size / BLOCK_SIZE_IN_BYTES; //cte. del sistema
    if (block_num < 12) {
        // los primeros 12 bloques son directos
        file.blocks[block_num] = blockaddr;
    } else if (block_num < MAX_BLOCKS_FIRST_INDIRECTION) {
        buffer block_table_buffer[BLOCK_SIZE_IN_BYTES];
        read_block(file.blocks[12], block_table_buffer);
        [illegible]
        blockaddr* singly_indirect_blocks = (blockaddr*) block_table_buffer;
        [illegible]
        uint idx = block_num - 12;
        singly_indirect_blocks[idx] = where_to_write;
    } else ... // más casos para 2 o 3 indirecciones
...
}
```

No voy a ahondar en los detalles técnicos de cómo escribir en más indirecciones; lo importante es que calcular dónde va el bloque nuevo es de complejidad CONSTANTE con respecto a la cantidad de bloques del archivo.

En este caso particular, optaré por el segundo enfoque debido a que cada directorio creado podría tener muchos archivos (las búsquedas de Youtube retornan potencialmente millones de resultados), todos los cuales iniciarían vacíos y serán de tamaño final muy variable (podrían durar 10 segundos o 2 horas). Entonces, con asignación continua podría caer en un caso muy desfavorable donde la asignación no es O(1): quedarme sin espacio contiguo [illegible] descargo el video y tener que copiar TODOS LOS BLOQUES a un pedazo de memoria libre (¡lineal en la cantidad de bloques!). Con inodos esto no ocurre.

(*) Asumo obviamente que conozco de antemano el tamaño del archivo (es parte de la metadata).

b) El único método en el que el límite teórico para el tamaño de los archivos es el tamaño del disco es asignación continua. Sin embargo, el límite con inodos puede llegar a ser un número gigantesco (varias decenas de terabytes), lo cual es bastante mayor que la capacidad de muchos discos. Por lo tanto, dependiendo del tamaño del disco que tenga, podría preferir asignación continua o inodos, ya que asignación continua en la práctica puede introducir otras desventajas debido a que tiene fragmentación externa: podría tener suficiente espacio para el archivo, pero que no sea continuo y deba reubicar otros archivos (¡tal vez muy grandes!). (✡)

pero a nivel teórico con bloques contiguos es suficiente

c) Asignación continua es el método más conveniente para este caso de uso: acceder a cualquier bloque del archivo requiere exactamente una lectura al disco. Inodos también provee acceso constante en notación O grande, pero si el bloque es indirecto, podría requerir algunas lecturas más (aunque el número de lecturas sea acotado). Por lo tanto, me decidiré por asignación continua. (✡)

d) Asignación continua provee la lectura secuencial más rápida: es simplemente leer los bloques de corrido. FAT también es una buena opción, ya que puedo consultar las direcciones de los bloques en la FAT recorriendo sus entradas (como una lista enlazada) en memoria principal y leyendo los bloques del disco. El overhead que da leer la FAT es despreciable pues está cargada en memoria, así que cualquiera de los dos métodos podría ser conveniente; dado que podría tener muchos archivos en este contexto de uso y que eso empeora el efecto de la fragmentación externa, me inclinaría más por FAT, que no tiene fragmentación externa. Inodos no es una muy buena opción porque requiere lecturas extra al disco para los bloques indirectos.

Nuevamente, para la tarea específico pedida bloques contiguos es el que mejor lo resuelve.

(✡) FAT es definitivamente la peor opción en este caso ya que tiene límite de 2GB o 4GB según la versión. Mucho menor que la capacidad de cualquier disco moderno.

(✡) FAT también es una muy mala opción en este caso ya que acceder al N-ésimo bloque de un archivo requiere N consultas a la tabla.

Ejercicio 2

- Al inicializar la impresora mediante driver_init(), será necesario registrar las interrupciones HP_LOW_INK_INT y HP_FINISHED_INT, así como las variables compartidas (ej. los semáforos / mutexes para evitar race conditions).
- Al quitar el dispositivo, se deben liberar esas interrupciones en driver_remove().
- Asumo que driver_write() es la función que se usa para imprimir (o de más bien decirle a la impresora que lo haga). Básicamente debe copiar los datos del usuario para evitar race conditions, chequear hasta 5 veces si la tinta está baja, volver en caso de que definitivamente esté baja (asumo que la responsabilidad de recargar la impresora y volver a llamar a la función una vez que tenga tinta recae en el usuario) y esperar a que la impresora termine de imprimir si no está baja. Para chequear la tinta usa polling, porque lo hace rápido y no lo hace más de 5 veces así que no desperdiciará muchos ciclos de CPU. Para esperar a que la impresora termine usará interrupciones, porque eso sí podría tardar muchísimo para los tiempos de la CPU. ✓

Código del driver

```
sem printing_finished;
mutex mut_text;
mutex mut_ctrl;

int driver_init() {
    sem_init(printing_finished, 0);
    mut_init(mut_ctrl, 1);
    mut_init(mut_text, 1);

    attach_interrupt(HP_FINISHED_INT, print_handler);
    attach_interrupt(HP_LOW_INK_INT, low_ink_handler);
    // no usé esto en el código, pero por ahí hay otra parte
    // (que no estoy implementando) en la que es necesario
}

int driver_remove() {
    destroy_sem(printing_finished);
    destroy_mutex(mut_ctrl);
    destroy_mutex(mut_text);
    detach_interrupt(HP_FINISHED_INT);
    detach_interrupt(HP_LOW_INK_INT); return 0;
}
```

no lo necesitas porque lo resolv con polling

✓

```
int driver_write(void* data, int size) {
    void* to_write = copy_from_user(data);
(*) lock(mut_text); OUT(LOC_TEXT_POINTER, to_write); (*)
    int i = 0;
    bool low_ink = true;
    while (i < 0 && low_ink) { // Polling
        lock(mut_ctrl);
        if (IN(LOC_CTRL) != LOW_INK) {
            low_ink = false;
        }
        ++i;
        unlock(mut_ctrl);
    }

    if (low_ink) {
        // Realmente estaba baja. unlock(mut_text);
        return ERR_LOW_INK;
    } else {
        // Imprimir.
        wait(printing_finished); // Bloquearse hasta que llegue la interrupción.
        return ERR_OK;
    }
}
```

(*)
OUT(LOC_TEXT_SIZE, size);

hay que repetirlo para cada validación de la tinta y para efectivamente imprimir.

(*) Cambiar de lock(mut_text); falta
wait(printing_finished); // Bloquearse hasta poder imprimir.

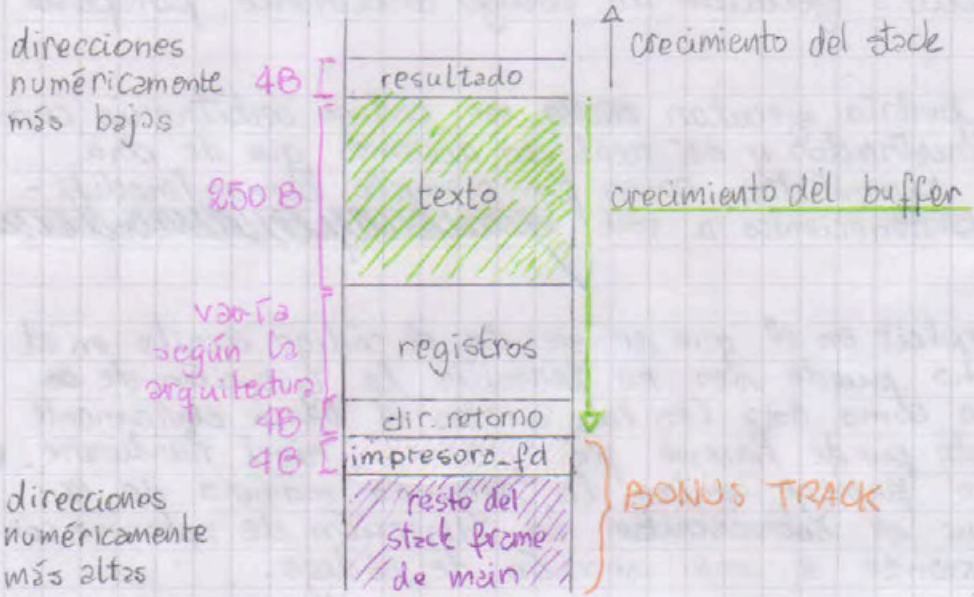
```
void print_handler() {
(*) if (IN(LOC_STATUS) == BUSY) {
        OUT(LOC_STATUS, READY);
        OUT(LOC_CTRL, FINISHED); unlock(mut_ctrl);
        signal(printing_finished);
    }
}
```

(*) lock(mut_ctrl);

están mal manejados los semáforos. Parecería que se espera que con un signal se levanten 2 wait, lo cual no ocurre

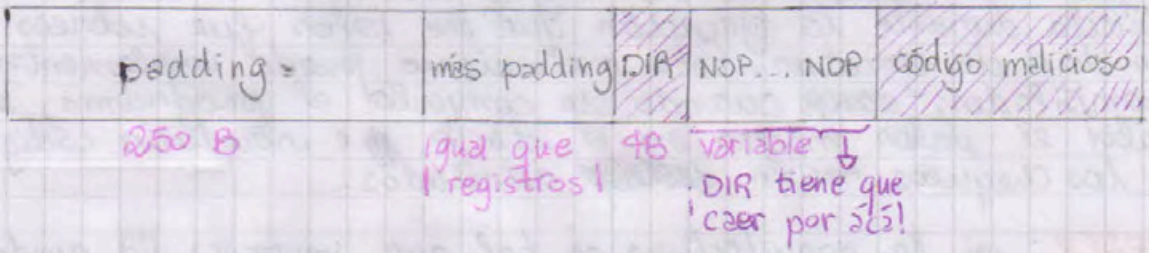
Ejercicio 3

a) Primero analicemos el estado de la pila en la llamada a imprimir:



El buffer texto mide 250 bytes, pero la función gets() no verifica el tamaño de la entrada antes de escribir en el buffer. Por lo tanto, si enviamos una entrada de más de 250 caracteres, se escribirá más allá del límite del buffer, hacia donde está la dirección de retorno. Esto se conoce como buffer overflow. ✓

Una manera de explotarlo es con la siguiente entrada:



(lo que está marcado en violeta es lo jugoso del exploit)

La idea es que esta entrada sobreescriba la dirección de retorno en el stack para que al volver de la función, el EIP pase a apuntar a DIR y no a la dirección que se suponía, y por lo tanto se ejecute lo que está contenido en DIR. En este caso, eso es alguna instrucción NOP de la entrada (que ahora está escrita en el stack). Como NOP no hace nada, el EIP continuará deslizándose por el stack hasta ejecutar el código malicioso.

DIR puede calcularse de antemano sabiendo cuánto ocupa aproximadamente el programa compilado y dónde está cargado en memoria. Los NOPs se ponen para evitar la molestia de calcular la dirección exacta del código malicioso.

Otra manera de explotar esta vulnerabilidad es, en vez de DIR, los NOPs y el código malicioso, enviar como entrada algo que, de manera análoga, sobreescriba la dirección de retorno con el punto de entrada de alguna otra función (desde ya que tendríamos que saber de antemano cuál es ese punto de entrada).

Cualquiera de estas dos maneras de explotar la vulnerabilidad tiene el mismo efecto: ejecución de código arbitrario por parte de un atacante.

b) Un atacante podría ejecutar ese código arbitrario con permisos de administrador y así realizar acciones que de otra manera no tendría permitidas, como por ejemplo borrar/modificar/leer archivos pertenecientes a root.

c) Para evitar el exploit en el que se ejecuta el código escrito en el stack, un mecanismo puede ser no permitir la ejecución de secciones marcadas como data (en las cuales el stack obviamente se incluye); esto puede hacerse por ejemplo a nivel hardware. Esto obviamente no protege contra la segunda manera de explotarla, en la que se sobreescribe la dirección de retorno con algo que sí corresponde a una sección de código.

Un mecanismo que sí puede proteger contra ambas formas es Address Space Randomisation: la posición de memoria en la que se carga cada programa (en memoria virtual) es aleatoria, por lo que el atacante no puede saber qué parámetro escribir en DIR.

Otro mecanismo que protege contra ambas es usar un stack canary, que es un número mágico escrito en el stack. Si se detecta durante la ejecución que ese valor fue sobreescrito, se deja de ejecutar. Este metanismo puede implementarse a nivel compilador; cuando se compila el programa, se debe escribir el valor mágico en el stack e introducir código que haga los chequeos recién detallados.

BONUS TRACK: si la arquitectura es tal que impresora_fd queda justo debajo del stack frame de imprimir (lo cual es posible, ya que imprimir se llama desde main y los frames de funciones llamadas anidadas quedan arriba), se podría sobreescribir impresora_fd.

Este efecto no parece ser tan interesante como el anterior; se podría por ejemplo hacer que cierre cualquier otro file descriptor (manteniendo la dirección de retorno de imprimir como estaba, por supuesto).

Ejercicio 4

Puede ocurrir que D envíe el mensaje y luego se caiga, entonces A no sabría que D está caído.

b) No porque como el problema de los generales bizantinos no tiene solución en una red son fallas y/o participantes maliciosos (y este podría ser el caso), A no tiene manera de saber si el mensaje realmente vino de D o si B lo falsificó. ✗

c) No porque el sistema es distribuido y los relojes de A y C podrían no estar sincronizados. Podría ocurrir que el reloj de A esté 5 minutos atrasado con respecto al de C, C mande un mensaje 1 segundo antes que A y los timestamps digan que el de A es anterior. Tampoco hay reloj centralizado así que no hay forma de saberlo. ✓

B tampoco puede basarse en el tiempo de llegada de los mensajes ya que no se puede garantizar que un mensaje enviado antes llegue antes por la red (esto ya depende del canal de comunicación, no de los nodos).

a) El primero y el segundo son indistinguibles (problema de los generales bizantinos), el tercero puede distinguirse si A espera más (asumiendo que no haya otro nodo haciéndose pasar por B, pero eso ya escapa al punto del ejercicio). ✓