Trabajo Práctico 2: Entrega Final

72.11 - Sistemas Operativos

Segundo Cuatrimestre 2024

Integrantes del **Grupo 21**:

62872	Matías LEPORINI
63382	Camila LEE
63074	Ana NEGRE

Índice

Instrucciones de Compilación y Ejecución

Instrucciones de Compilación

Para crear el contenedor dentro del directorio que contiene el repositorio:

```
docker pull agodio/itba-so-multi-platform:3.0
cd $MyDir
docker run -v "{$PWD}:/root" --security-opt seccomp:unconfined -ti
agodio/itba-so-multi-platform:3.0
```

Una vez iniciado el contenedor, correr los comandos:

```
cd root
cd TP2-SistemasOperativos-72.11
```

Se provee para la compilación un script que compila el proyecto en su totalidad. El mismo compila tanto el kernel como el módulo de usuario y la toolchain.

Opciones de Compilación

En tiempo de compilación, el usuario puede elegir el tipo de administrador de memoria:

- Free List (default)
- Buddy

Para compilar con Buddy MMU, dentro del contenedor puede correr el comando:

```
bash ./compile.sh BUDDY
```

Para compilar con Free List MMU (default), puede correr el comando:

```
bash ./compile.sh
```

Instrucciones de Ejecución

Para ejecutar el proyecto, puede hacer uso del script provisto para tal fin. Fuera del contenedor, debe ejecutar:

```
bash ./run.sh
```

Introducción

En el presente informe se detalla el proceso de desarrollo del Trabajo Práctico 2 de Sistemas Operativos. El objetivo de este trabajo era la creación de un kernel, usando como base el trabajo final de Arquitectura de Computadoras, y agregándole nuevas características vistas en clase: administración de memoria, procesos, scheduling, mecanismos de IPC y sincronización.

A continuación se listan las decisiones de desarrollo tomadas a partir de la última entrega –para crear las nuevas funcionalidades requeridas o para adaptar las creadas anteriormente para soportar nuevos requerimientos–, justificaciones de los warnings arrojados por PVS-Studio y CPPCheck, limitaciones que presenta el trabajo y la verificación de funcionalidades.

Para concluir el informe, se agregaron observaciones relevantes al sistema y fragmentos de código reutilizado.

Decisiones de Desarrollo

---

1. Memory Manager y Buddy

Se implementó un sistema buddy con énfasis en la simplicidad y compatibilidad con la interfaz existente del memory manager. Es por esto que se optó por usar un array de bloques libres por tamaño, donde cada índice del array representa un orden (potencia de 2) de tamaño de bloque. Se decidió utilizar como **MIN_BLOCK_SIZE = 64** bytes para minimizar el overhead en asignaciones pequeñas como *malloc*s de nombres de funciones/procesos. Además, como **MAX_ORDER** se asignó **28** (máximo tamaño de un bloque ~128MB) pues se consideró un tamaño suficiente para nuestro sistema. Por otro lado, Tanenbaum menciona que Linux implementa slab allocation o object caches, pero se decidió que la sobrecarga de gestión adicional no justificaba la complejidad para nuestro caso de uso. Las operaciones críticas (*malloc*/*free*) son *O(log n)* en el peor caso y la búsqueda de buddies es *O(1)* lo cual sugiere un rendimiento adecuado del sistema.

Vale mencionar una mejora realizada a la implementación original (entrega parcial 1) de *free list* de la MMU: la incorporación de una función de coalescencia que unifica bloques contiguos libres. Esta modificación se realizó con el fin de reducir la fragmentación de memoria, y fue particularmente efectiva en escenarios de liberación frecuente de memoria (por ejemplo, la eliminación de filósofos); se puede verificar su efectividad habilitando los prints dentro de la función *coalesce_free_blocks*. Para la implementación del memory manager *free list* se tomó como base la solución propuesta por Kernighan y Ritchie (ver “Fragmentos de Código Reutilizados”).

2. Scheduling/Process Management

Se implementó un sistema de multitasking preemptivo usando como base el algoritmo de *Priority Based Round-Robin*. Las principales decisiones de diseño fueron:

1. Rango de prioridades:

- Se estableció un rango de prioridades del **0** al **10** donde 0 representa la prioridad más baja y 10 la más alta.
- Se estableció la prioridad por defecto (**DEFAULT_PRIORITY**) en 3 para los nuevos procesos.

2. Cálculo del quantum:
   - Se asignó un quantum que se calcula en función de la prioridad del proceso usando la siguiente fórmula:

     ```
     QUANTUM + processTable[currentPID].priority);
     ```

   - Esto produjo un rango de quantum de 5ms a 20ms. Si bien esto implica un quantum menor al estándar, se consideró aceptable teniendo en cuenta la cantidad de procesos que pueden estar activos simultáneamente en el sistema y la baja complejidad de los mismos.
3. Selección de procesos:
   - Se implementó una función *getNextReady()* para seleccionar el siguiente proceso en un esquema Round-Robin manteniendo un orden cíclico.
4. Prioridad en desbloqueo:
   - En la función *unblock()* se implementó un mecanismo para asegurar que procesos de alta prioridad (≥ MAX_PRIORITY / 2) sean seleccionados inmediatamente al ser desbloqueados. Esto también se tuvo en cuenta en el comando *nice()* a la hora de cambiar la prioridad de los procesos en ejecución.
5. Cambio de contexto:
   - Se agregó al handler del timer tick la función *switch_process()*, para evaluar la necesidad del cambio de contexto, además de para poder preservar el *rsp* de cada proceso.
   - La función *switch_process()* maneja el cambio de contexto entre procesos y fue implementada de forma tal que pueda balancear la ejecución prioritaria y la equidad round-robin:
     - Continuar el proceso actual si el quantum no expiró.
     - Cambiar al proceso de prioridad alta recién desbloqueado si es que existe.

    - Seleccionar un nuevo proceso usando *getNextReady*() si es necesario.

6. Manejo de pseudo proceso del kernel
    - Se estableció un pseudo PID (**PID_KERNEL**) para el kernel, el cual se ejecuta de no haber otros procesos disponibles. El main del kernel en sí, sin embargo, luego de las configuraciones iniciales solo ejecuta *yield()* para permitir la ejecución de los procesos subsiguientes.
    - Además, se decidió inicializar la shell como primer proceso, de manera que al habilitarse las interrupciones comience a ejecutarse. Este proceso se creó como "inmortal", con lo cual siempre debería estar ejecutándose.

Por otro lado, para la implementación de los procesos se utilizó un arreglo estático para almacenarlos. Cada proceso es representado con una estructura que encapsula toda la información esencial: nombre, puntero al inicio del stack del proceso, puntero al final del stack del proceso, argumentos, estado (*foreground* o *background*), y otros campos de administración de memoria dinámica.

Posteriormente a la entrega parcial de procesos, se decidió agregar una estructura jerárquica entre los procesos (previamente, cada proceso era independiente del resto). Esta modificación permitió asemejar la shell del proyecto al funcionamiento de una terminal "real": al ejecutar un proceso, la shell se bloquea hasta que el proceso creado termina. Adicionalmente, se implementó la posibilidad de que un proceso padre se bloquee hasta la muerte de todos sus hijos, lo cual resultó de especial utilidad a la hora de correr los tests que crean varios procesos.

3. Semáforos

Se implementó la posibilidad del uso de semáforos a través de syscalls. Para la implementación de los mismos se utilizaron las funciones proporcionadas por la cátedra para la obtención del acceso exclusivo a la zona crítica (ver "Fragmentos de Código Reutilizados"). Además, se decidió contar con un *lock* global para impedir *race-conditions* entre procesos al intentar modificar variables de los semáforos, y un *lock* individual para cada semáforo, el cual es el mutex "en sí".

Desde la última entrega se identificó y resolvió una limitación crítica en el manejo de procesos en semáforos. La implementación original utilizaba una única

lista (*interestedProcesses*) para manejar todos los procesos, lo que generaba conflictos al no poder distinguir entre procesos que solo habían abierto el semáforo y aquellos en espera bloqueante.

La solución separa las responsabilidades mediante dos listas distintas:

1. ***interestedProcesses***: gestiona el ciclo de vida del semáforo (*sem_open*/*sem_close*).
2. ***waitingProcesses***: maneja la sincronización (sem_wait/sem_post) con orden FIFO.

Esta separación permitió identificar precisamente qué procesos bloquear o desbloquear consultando *waitingProcesses*, manteniendo el orden correcto de las esperas y verificando que pertenezcan en *interestedProcesses*.

Una característica de la implementación a destacar es el funcionamiento de la apertura y el cierre de los semáforos. Con la finalidad de asemejarse lo más posible al funcionamiento de los semáforos en Linux, manteniendo la simplicidad para facilidad de implementación, se decidió lo siguiente:

Cuando un proceso abre un semáforo con cierto nombre (los cuales funcionan como key), este semáforo continúa “vivo” (es decir, puede ser accedido mediante esa *key*) hasta que el último proceso interesado cierra el semáforo. De esta manera, garantizamos la posibilidad de que distintos procesos accedan al mismo semáforo de manera “espontánea”, sin necesidad de sincronizar la liberación de recursos entre ellos (más allá del recurso a ser accedido).

Adicionalmente, se decidió que solo puedan modificar los semáforos aquellos procesos que lo hayan abierto previamente. Si bien esto implicó ciertos desafíos adicionales en la creación de procesos padre-hijos que utilicen semáforos (por ejemplo, en algunos tests o en el programa de filósofos), se consideró que no poner en práctica esta medida podría ser un problema de seguridad importante en un sistema más complejo, por lo que decidió utilizarse.

4. Pipes

Se identificaron dos casos de uso a la hora de pensar la solución a implementar: en el primer caso, el pipe se utiliza como medio de comunicación explícita entre dos procesos, compartiendo la memoria fija alocada por el pipe, a la cual varios procesos pueden escribir y leer abriendo el pipe previamente. En este

sentido, la implementación es muy similar a los semáforos. Para sincronizar la lectura y escritura se utilizaron dos semáforos, poniendo en práctica un problema de *consumers y producers*.

Luego en el segundo caso de uso se utiliza al pipe como forma de "juntar" procesos, de manera que la concatenación de comandos utilizando el pipe (símbolo "|") desde la shell genere la salida del primer proceso como entrada del segundo. Se tuvo especial atención a la siguiente directiva: "debe ser transparente para un proceso leer o escribir de un pipe o de la terminal". Para lograr esto, se añadió a la información de cada proceso el file descriptor de input y output. Al crearse el proceso, debe indicarse dónde será su salida y entrada en la estructura de creación. De esta manera, la shell puede abrir un pipe que será utilizado para comunicar ambos procesos. Luego, se modificó las funciones *read()* y *write()* para tener en cuenta los *file descriptors* de entrada y salida de cada proceso. En el caso de un proceso con entrada y salida estándar, se obtiene la información del teclado (read), y se imprime la información en pantalla (write); en otro caso se llama a *read_from_pipe()* o *write_to_pipe()* según corresponda. Esto permite comunicar a los procesos sin necesidad de modificar su comportamiento.

Adicionalmente, a diferencia del funcionamiento usual de los sistemas, y a modo de simplificar el desarrollo, se consideraron a los *file descriptors* estándar como "reservados", no permitiendo que los mismos se comporten de igual manera que los pipes (no pueden cerrarse ni modificarse).

5. Manejo de Procesos Background

Para la implementación de procesos en background, se decidió modificar la lógica de impresión para únicamente permitir a los procesos en *foreground* imprimir a pantalla. La única excepción a este comportamiento se tuvo en cuenta en caso de error (impresión con file descriptor ***STDERR***). Esto se decidió en base a que, en caso de error, es de especial interés para el usuario final poder conocerlo (caso idealmente lejano).

Adicionalmente, en el caso de comandos *pipeados*, la indicación de background actúa sobre el comando *consumer* únicamente, para permitir que éste reciba lo generado por el comando *producer*.

Para ejecutar un comando en background basta con utilizar el símbolo “-b” en la línea de comandos.

6. Filósofos Comensales

Durante la implementación del problema de los filósofos comensales se presentó un desafío en el manejo de los semáforos para los filósofos: no podían señalizar a sus vecinos creados posteriormente porque sus *PID*s no estaban en la lista de *interestedProcesses* del semáforo correspondiente. Por ejemplo, cuando el filósofo 2 intentaba ejecutar *try_to_eat(RIGHT(2))*, no podía realizar *sem_post* al semáforo del filósofo 3 porque su PID no estaba en la lista de procesos interesados de ese semáforo, ya que este había sido creado después; esto se debe a la forma en la que se implementó el semáforo: los procesos que abren los semáforos son los únicos que luego tendrán acceso a las funciones de *wait* y *post* (ver Sección 3: Semáforos).

Tras evaluar múltiples soluciones, incluyendo la inicialización total de recursos y o incluso cambiar la implementación actual a un método EVEN-ODD, implementamos una solución basada en la naturaleza circular de la mesa:

- Filósofo 0: acceso a todos los PIDs posibles
- Demás filósofos: acceso al filósofo a su izquierda (id-1) y derecha (id+1), y al filósofo 0 en el caso de que su mismo el mismo filósofo sea el último en la lista (es decir que su id == num_philosophers).

Esta implementación, aunque implica un overhead en el manejo de PIDs, garantiza la sincronización circular y previene deadlocks, asegurando el acceso equitativo a los recursos cuando están disponibles.

Errores de PVS-Studio y CPPCheck

El análisis con PVS-Studio y CPPCheck revela algunos falsos positivos: advertencias sobre manipulación de punteros void* (para direcciones de memoria) y detección de loops infinitos (while (1) del shell y filósofos). Estos fueron ignorados pues se tratan de casos que forman parte del diseño y funcionamiento esperado del sistema.

Limitaciones

Se implementaron límites estáticos para varios recursos del sistema, priorizando la funcionalidad básica (y tiempo) sobre la escalabilidad:

- Procesos: MAX_PROCESSES = 15
- Semáforos: MAX_SEMAPHORES = 255
- Pipes: MAX_PIPES = 20 (buffer de 1024 bytes)
- Filósofos: MAX_PHILOSOPHERS = 10
- Nombres de proceso: MAX_NAME_LENGTH = 20 caracteres
- Stack por proceso: STACK_SIZE = 4096 bytes

La decisión de utilizar arreglos estáticos responde al alcance del proyecto y simplifica la gestión de memoria. Por ejemplo, el límite de 10 filósofos fue establecido considerando el uso total de procesos del sistema (incluyendo shell y proceso principal), garantizando recursos suficientes para la demostración de la funcionalidad sin comprometer la estabilidad del sistema.

Una limitación en el caso de la implementación de pipes es la lectura múltiple. Con el fin de evitar la obligación de guardar varios índices de lectura (uno para cada proceso) y simplificar la implementación, solo se cuenta con un índice de lectura. Esto resulta un limitante puesto a que, de querer varios consumidores leer de un mismo recurso, al terminar la lectura un proceso, el siguiente comenzará a leer desde donde dejó el proceso anterior, y no desde el principio de lo escrito por el productor.

Otra limitación encontrada es el uso de file descriptors en las syscalls de lectura y escritura. En el sistema actual, solo se aceptan *STDIN, STDERR y STDOUT* como file descriptors válidos. Si bien esta decisión fue tomada en base a los requerimientos del trabajo, se considera que no debería resultar demasiado complejo aumentar el potencial de uso de los *file descriptors* en las syscalls, dada la lógica establecida en el kernel.

Verificación de Funcionalidad

Al iniciar el sistema operativo puede hacerse uso de la línea de comandos. Si bien al comenzar se proporciona un listado de los comandos posibles, a continuación se listan los mismos con mayor detalle:

1. Tests de funcionalidad

- **testmm**

Ejecuta un test de memoria en un ciclo eterno. Toma la máxima cantidad de memoria a ser pedida en un mismo bloque como parámetro.

- **testproc**

Ejecuta un test de creación/bloqueo/asesinato de procesos en un ciclo eterno. Toma como parámetro la cantidad máxima de procesos a ser creados en un ciclo del test.

- **testpipe**

Ejecuta un test sobre la utilización de los pipes implementados. Se utiliza un string compartido entre dos procesos, los cuales acceden a él mediante el pipe creado. El texto es lo suficientemente largo como para que no pueda leerse/escribirse en pantalla en una sola pasada, lo cual fuerza la sincronización del pipe. No recibe parámetros

- **testsync**

Ejecuta un test de sincronización. Toma como parámetro el número el cual cada proceso va decrementando / incrementando, y una flag de uso de semáforos. El comportamiento esperado es: en caso de la flag de uso de semáforos estar en cero, ocurre una race condition no resuelta, por lo que el resultado final es distinto de cero; en caso de utilizar semáforos el resultado final es siempre cero.

- **testprio**

Ejecuta un test de prioridades. El objetivo es la visualización del *timeslice* asignado a cada proceso según su prioridad. No toma parámetros.

2. Comandos “pipeables”

Los siguientes comandos pueden utilizarse tanto como *consumer end* de un pipe o solos.

- **cat**

Escribe en pantalla lo ingresado por teclado (o como resultado del output de un proceso) hasta recibir *EOF*. No recibe parámetros.

- **wc**

Filtra el input recibido (por teclado o por un proceso), imprimiendo en pantalla la cantidad de líneas escritas hasta la recepción del *EOF*. No recibe parámetros.

- **filter**

Filtra las vocales del input. recibido (por teclado o por un proceso), imprimiendo lo filtrado en pantalla hasta la recepción del *EOF*. No recibe parámetros.

3. Comandos de Procesos

- **ps**

Imprime por pantalla la información de cada proceso. No recibe parámetros.

- **kill**

Mata un proceso. Recibe el pid del proceso en cuestión por parámetro.

- **block**

Bloquea un proceso. Recibe el pid del proceso en cuestión por parámetro.

- **unblock**

Desbloquea un proceso en caso de estar bloqueado. Recibe el pid del proceso en cuestión por parámetro.

- **nice**

Cambia la prioridad de un proceso. Recibe por parámetro el pid del proceso y la nueva prioridad del proceso en cuestión. Cabe destacar que no se permite cambiar la prioridad de la shell por cuestiones de performance.

4. Comando de Memoria

- **mem**

Imprime por pantalla el estado actual de la memoria. No recibe parámetros.

5. Generales

- **phylo**

Imprime por pantalla el resultado del problema de filósofos visto en clase. Recibe por parámetro la cantidad de filósofos con los que inicializar el problema. Adicionalmente, pueden utilizarse las teclas **a** y **r** para añadir y quitar un filósofo a la vez respectivamente. La cantidad de filósofos debe estar entre 3 y 10 inclusive.

- **loop**

Imprime un saludo por pantalla en un ciclo infinito. No recibe parámetros.

- **clear**

Limpia la pantalla. No recibe parámetros.

- **help**

Imprime el listado completo de comandos disponibles. No recibe parámetros.

Otros comandos, como *scale_up* o *time,* no resultan de interés puesto que fueron desarrollados para Arquitectura de Computadoras, y no exhiben la funcionalidad agregada para Sistemas Operativos.

Observaciones

A modo de observación, se detalla el comportamiento de **^C**:

Por la forma en la que se decidieron implementar los procesos, resultó conveniente que **^C** no sólo mate al proceso padre de foreground, sino que también mate a todos los procesos hijos de éste. Esto se decidió por conveniencia, ya que resultaría indeseable que los procesos creados por un test, por ejemplo, continúen vivos luego de matar al proceso que inició el test. Además, de caso contrario, pueden provocar errores en el normal funcionamiento de los comandos, puesto a que una gran mayoría de ellos demuestran la comunicación entre procesos.

Por el otro lado, se decidió que todos los comandos disponibles se ejecuten como procesos. No se encontró ningún comando con la necesidad de ser built-in, por lo que se optó por la opción más conveniente (que además resulta más fiel al comportamiento estándar de una terminal).

Fragmentos de Código Reutilizado

Al utilizar la implementación de *Kernighan & Ritchie* para la creación del primer memory manager, se tuvo en fuerte consideración la lógica del código provisto en la sección 8.7 del libro:

```
typedef long Align;
union header{
        struct{
        union header *ptr;
        unsigned size;
}s;
Align x;
};
```

```
static Header base;
static Header * freep = NULL;
void * malloc(unsigned nbytes){
Header *p, *prevp;
Header *morecore(unsigned);
unsigned nunits;
nunits = (nbytes+sizeof(Header)-1)sizeof(Header)+1;
if(prevp=freep)==NULL){
base.s.ptr = freep = prevp= &base;
base.s.size = 0;
}
for(p=prevp->s.ptr; ; prevp=p, p=p->s.ptr){
if(p->s.size>=nunits){
if(p->s.size == nunits){
prevp->s.ptr = p->s.ptr;
}
else{
p->s.size -=nunits;
p += p->s.size;
p->s.size = nunits;
}
freep = prevp;
return (void*)(p+1);
}
if(p == freep){
if((p = morecore(nunits)) == NULL)
}
}

}
```

```
void free(void *ap){
        Header * bp, *p;
        bp = (Header *) ap-1;
        for(p= freep; !(bp>p && bp < p->s.ptr); p = p->s.ptr){
if(p>= p->s.ptr && (bp>p || bp < p->s.ptr)){
break;
}
```

```
}
if(bp+bp->s.size == p->s.ptr){
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
} else
        bp->s.ptr = p->s.ptr;
if(p+p->s.size == bp){
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
} else
        p->s.ptr = bp;
freep = p;
}
```

Luego para la implementación de semáforos se utilizaron las funciones vistas en clase para tal fin:

```
GLOBAL acquire
GLOBAL release

acquire:
    mov al, 0
.retry:
    xchg [rdi], al
    test al, al
    jz  .retry
    ret

release:
    mov byte [rdi], 1
    ret
```