Trabajo Práctico 2: Segunda Entrega

72.11 - Sistemas Operativos

Segundo Cuatrimestre 2024

Integrantes del **Grupo 21**:

62872	Matías LEPORINI
63382	Camila LEE
63074	Ana NEGRE

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. Para utilizarlo, dentro del contenedor 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
```

Decisiones de Desarrollo

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:
    - El quantum asignado a cada proceso se calcula en función de la prioridad del proceso usando la siguiente fórmula:
      ```
      QUANTUM + processTable[currentPID].priority);
      ```
    - Esto resulta en 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.
5. Cambio de contexto:
    - Se agregó al handler del timer tick la función *switchP()*, para evaluar la necesidad del cambio de contexto, además de para poder preservar el *rsp* de cada proceso.
    - La función *switchP()* maneja el cambio de contexto entre procesos y fue implementado 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.

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.

Observaciones

A lo largo del trabajo, se enfrentaron varios obstáculos, siendo el mayor un error en la creación de procesos desde Userland. Para solucionarlo, se agregó la flag -fno-pie a Userland, para asegurar que las direcciones de las funciones de Userland accedidas por las syscalls fueran las correctas.

Adicionalmente, cabe aclarar que dada la naturaleza parcial de la entrega, nos enfocamos únicamente en la creación y éxito de los tests, por lo que los comandos "extra" como *ps* no fueron implementados.

Cabe destacar que solo *test_prio* se implementó para ejecutarse como un proceso. A diferencia de este test, se decidió no ejecutar *test_mm* como proceso independiente debido a que, dado que este test se ejecuta en un loop infinito y no se ha implementado el mecanismo para ejecutarlos en background, sería imposible ejecutarlo concurrentemente con otros procesos; con lo cual se consideró que no era muy útil implementar su ejecución como proceso para esta entrega parcial. Este caso es el mismo que el caso de *test_proc*, por más de que este sí fue implementado como proceso como parte del alcance de la entrega.

En la siguiente entrega se modificará enteramente la shell, implementando los *file descriptors*, pipes, etc. necesarios para ejecutar cualquier proceso en background y poder monitorear los procesos activos desde la terminal.

Como último comentario, se aclara que, dada la limitación de la terminal actual (no acepta argumentos), se decidió establecer por el momento como constante la máxima cantidad de procesos a crear por *test_proc*, la cual puede modificarse y recompilar el sistema.

Fragmentos de Código Reutilizado

En esta entrega, la implementación de las funciones de procesos, context switching y scheduling se basó en los conceptos discutidos en las clases prácticas y teóricas.