Plancha 3: Programas de usuario y multiprogramación

2021 – Sistemas Operativos 2

Licenciatura en ciencias de la computación

1. Introducción

Véase el documento *Notas para la Plancha 3*.

2. Ejercicios

1. Al ejecutar programas de usuario, tendremos **dos espacios de direcciones**. El primero es el que utiliza el núcleo de Nachos, el cual corre en la máquina x86. El segundo es el del proceso de usuario que corre **sobre la máquina MIPS simulada**, por lo tanto serán direcciones en la máquina simulada. Por tanto, los punteros (arreglos y cadenas) no pueden ser intercambiados entre estos dos espacios de direcciones.

   La cabecera `userprog/transfer.hh` ofrece funciones para copiar datos desde el núcleo al espacio de memoria virtual del usuario y viceversa. Las hay de dos tipos, unas que leen/escriben cadenas terminadas por ceros y otras que leen un número fijo de bytes. Las signaturas son:

   ```
   void ReadBufferFromUser(int userAddress, char *outBuffer,
                           unsigned byteCount);
   bool ReadStringFromUser(int userAddress, char *outString,
                           unsigned maxByteCount);
   void WriteBufferToUser(const char *buffer, int userAddress,
                          unsigned byteCount);
   void WriteStringToUser(const char *string, int userAddress);
   ```

   Solo `ReadStringFromUser` está implementada. Agregue implementaciones para las demás.

   Sugerencia: para acceder a la memoria del usuario puede usar los métodos `Machine::ReadMem` y `Machine::WriteMem`.

2. Implemente las llamadas al sistema y la administración de interrupciones. Se deben proveer todas las llamadas al sistema definidas en `syscall.h` exceptuando `Fork` y `Yield`.

   Note que la implementación debe ser "a prueba de balas", o sea que un programa de usuario no debe poder hacer nada que haga caer el sistema operativo (con la excepción de llamar explícitamente a `Halt`). Esto implica que hay que tener cuidado con dónde se usa la macro `ASSERT`.

Se sugiere implementar las llamadas a sistema en el siguiente orden:

   a) `Create`, `Remove` y `Exit`.

   b) `Read` y `Write` para la consola. Puede ser útil implementar una clase `SynchConsole` que provea la abstracción de acceso sincronizado a la consola. En `userprog/prog_test.cc` hay una función que es sirve como comienzo de la implementación. La clase de acceso sincronizado a disco (`SynchDisk`) puede servir de modelo. Tenga en cuenta que a diferencia del disco, en este caso un hilo queriendo escribir no debería bloquear a un hilo queriendo leer.

   c) `Open` y `Close`. Será necesario que cada hilo disponga de su propia tabla de archivos abiertos.

   d) `Read` y `Write` para archivos.

   e) `Join` y `Exec`. Para poder probar `Exec` y `Join` se debe implementar primero multiprogramación. Para ello deberá proponer una manera de ubicar los marcos de memoria física para que se puedan cargar múltiples programas en la memoria (ver `lib/bitmap.hh`). Recuerde además marcar la memoria como libre cada vez que finalice un programa.

3. Agregue una función interna del núcleo que imprima el estado actual del planificador.

   Opcional: agregue una nueva llamada al sistema `Ps` que permita a los programas de usuario invocar la función.

4. Implemente multiprogramación con rebanadas de tiempo ("time slicing"). Será necesario forzar cambios de contexto después de cierto número de tics del reloj. Note que, ahora que está definido `USERPROG`, `Scheduler` almacena y recupera el estado de la máquina en los cambios de contexto.

5. La llamada `Exec` no provee forma de pasar parámetros o argumentos al nuevo espacio de direcciones. Unix permite esto, por ejemplo, para pasar argumentos de línea de comando al nuevo espacio de direcciones. Implemente esta característica.

   Hay dos formas de hacerlo, puede elegir cualquiera de las dos:

   a) Una es al estilo de Unix: copiar los argumentos en el fondo del espacio de direcciones virtuales de usuario (la pila) y pasar un puntero a los argumentos como parámetro a `main`, usando el registro *4* para pasar la cantidad y *5* para pasar el puntero.

   Los registros *4* y *5* (también llamados *a0* y *a1*) se usan para el primer y segundo parámetros de función, de acuerdo a la convención de llamada de MIPS que usamos. Es decir, son el `argc` y el `argv` del `main`, respectivamente.

   El esquema de la figura 1 (en la página siguiente) ilustra la organización de los datos en la pila.

   Sugerencia: puede aprovechar las funciones que provee Nachos en el archivo `userprog/args.cc`.

   b) La otra forma es agregar una nueva llamada al sistema, que cada espacio de direcciones llame como primera cosa en `main` y obtenga los argumentos para el nuevo programa.

   Sugerencia: para esta forma también puede aprovechar `userprog/args.hh`, pero deberá hacer algunas modificaciones a la función `WriteArgs`.

6. Programas de usuario:

   a) Implemente una biblioteca de funciones para programas de usuario, en `userland/lib.c`. Debe incluir al menos las siguientes funciones:

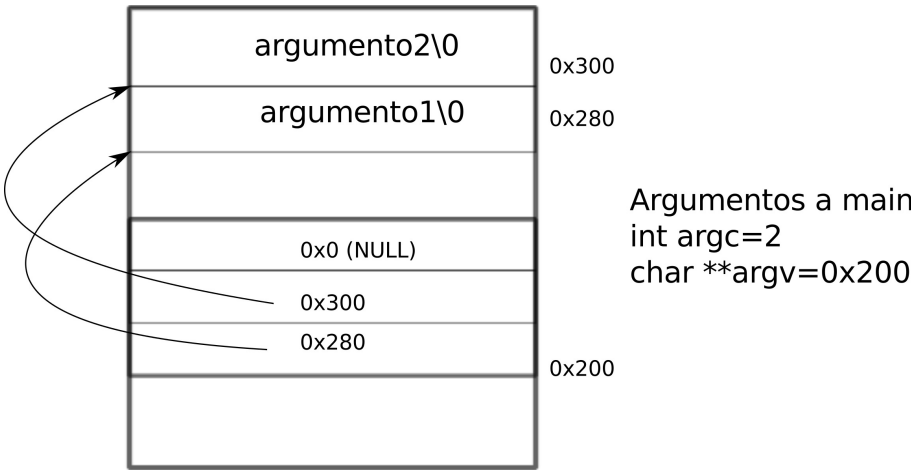

Figura 1: Esquema de la pila al pasar argumentos a `main`.

```
unsigned strlen(const char *s);
void puts(const char *s);
void itoa(int n, char *str);
```

b) Escriba los programas utilitarios `cat`, `cp` y `rm`. Puede aprovechar la biblioteca del punto anterior para facilitar la implementación. Guarde los archivos en el directorio `userland` como los que ya vienen con Nachos; tenga en cuenta que deberá modificar `userland/Makefile` para que se compilen al ejecutar `make`.

Sugerencia: puede tomar como referencia el archivo `userland/tiny_shell.c` provisto por Nachos.

c) Nachos incluye dos intérpretes de comandos: `userland/shell.c` y `userland/tiny_shell.c`. Ambos leen líneas de comando de la consola y las ejecutan usando la llamada a sistema `Exec`. El primero es más avanzado, fundamentalmente en el manejo de argumentos.

Ambos programas tienen el mismo comportamiento al ejecutar una línea de comando: se quedan esperando a que termine la ejecución de esta para permitir ingresar un nuevo comando. Este comportamiento se llama ejecución en primer plano. Uno podría en cambio desear ejecutar tareas en segundo plano: que su ejecución no bloquee al intérprete, sino que este siga aceptando entradas y ejecutando otras tareas en forma paralela a la ejecución de la primera.

Modifique `shell` para que ejecute en segundo plano aquellas líneas de comando que empiecen con el caracter `&`.