Instituto Politécnico Nacional

Escuela Superior de Cómputo

Materia: Sistemas operativos

Grupo: 4SCM1

Profesor: Araujo Díaz David

Integrantes:

- Arcos Hermida Aldo Alejandro - 5
- Chávez Becerra Bianca Cristina - 9
- Islas Osorio Enrique - 20
- Juárez Cabrera Jessica - 21
- Palmerin García Diego - 28

Trabajo: Practica #9 Manejo de archivos y directorios

Objetivo

Aplicar la forma como se manejan archivos en los sistemas LINUX/UNIX. Programar la Técnica de mapeo de archivos.

Desarrollo

Por lo general, leemos un carácter o una línea a la vez. Al menos eso parece. La realidad es que, por lo general, hay bastantes búferes entre el usuario y el disco duro, por lo que su solicitud de un carácter puede desencadenar una lectura de 2048 caracteres y luego sus llamadas subsiguientes regresan desde el búfer.

Una computadora moderna puede hacerlo mucho mejor que leer usando cosas como fgetc. Dado que un programa tiene un gran espacio de direcciones virtuales y que la computadora tiene una unidad de administración de memoria perfectamente buena, se puede pedir al sistema operativo que simplemente asigne el archivo a su espacio de memoria. Luego, puede tratarlo como cualquier otra matriz de caracteres y dejar que el sistema operativo haga el resto.

El sistema operativo no necesariamente lee todo el archivo de una sola vez, solo reserva espacio para el archivo. Cada vez que se accede a una página que no está en la memoria, el sistema operativo la captura de forma invisible. Las páginas que no utilice con mucha frecuencia se pueden descartar y volver a cargar más tarde. Detrás de escena, el sistema operativo hace mucho para que pueda trabajar en archivos muy grandes sin ningún esfuerzo real. La llamada que lo hace todo es mmap.

Por supuesto, siempre hay una trampa. Si se tiene un archivo realmente grande, es posible que se deba trabajar para mapearlo parcialmente y luego mapearlo nuevamente. También crear o extender archivos es un poco más de trabajo usando el mapeo. Aún así, el mapeo de memoria es fácil de hacer en la mayoría de los casos comunes.

Lo primero que se debe decidir es si desea leer el archivo o leer y escribir en el archivo. Si solo necesita acceso de lectura, se puede solicitar un mapeo privado. Eso significa que se obtendrá el archivo tal como existe y cualquier cambio que realice simplemente copiará las páginas en su propia copia privada. Por lo general, no cambiará los archivos que abre de esta manera a menos que cree un nuevo archivo para escribir los cambios.

Sin embargo, si se desea escribir en el archivo como se hace en la memoria, se necesitará una asignación compartida. Esto se puede usar para compartir datos con

otros procesos, pero también se asegura de que el archivo reciba actualizaciones a medida que las crea.

El siguiente programa simple de conteo de palabras de un archivo. En lugar de usar llamadas de E/S estándar (stdwd), se usa open para abrir el archivo para leerlo y luego lo asigna al espacio de direcciones del programa. Necesitamos saber el tamaño del archivo:

```
int fd = open(filename, O_RDONLY); // open file
struct stat finfo;
char *b;
if ( fd == -1 )
  {
  perror(filename);
  return 2;
  }
if ( fstat( fd, &amp;finfo ) == -1 ) // learn size of file
  {
  perror(filename);
  return 3;
  }
b=mmap( NULL, finfo.st_size, PROT_READ, MAP_PRIVATE, fd, 0 ); // map to memory
if ( b == MAP_FAILED )
  {
  perror("mmap");
  return 4;
  }
```

Los argumentos para mmap son simples. La primera es una dirección. Casi nunca se necesita especificar la dirección. Si se hace, hay muchas reglas sobre cómo configurar la dirección que varían según la plataforma. Al especificar NULL, mmap elegirá un lugar. El siguiente argumento es una longitud seguida de banderas. En este caso, le decimos al sistema que solo nos importa leer el archivo. También especificamos un mapeo privado y luego el identificador de archivo y un desplazamiento desde el inicio del archivo.

Una vez que esta llamada tiene éxito, la variable b tiene un puntero a todo el archivo en la memoria. Imprimirlo sería tan fácil como:

```
while (--len) putchar *b++;
```

```
while (len--) // process each character
 {
 if ( *b == '\n' ) l++;
 if ( isspace(*b) )
  {
  if (state==1)
   {
   w++;
   state=0;
   }
  }
  else state=1;
  b++;
}
```

En teoría, la E/S asignada a la memoria debería ser más rápida que un programa que realmente realiza E/S de disco. Sin embargo, las bibliotecas pueden estar almacenando en búfer e incluso mapeando a sus espaldas, por lo que la diferencia de rendimiento podría ser muy pequeña. Pero la simplificación del código es una gran ventaja independientemente del rendimiento.

Conclusiones

Esto parece genial, pero debe tener en cuenta algunos problemas potenciales. Estamos acostumbrados a pensar que si leemos algunos datos del disco y nada sale mal, podemos olvidarnos de ello. Pero mapear un archivo no es lo mismo que leerlo. Suponga que mapea un archivo en una unidad de red y tal vez lea algunas páginas de él. Entonces la red se cae. La lectura de una nueva página provocará un error porque el archivo subyacente ya no está.

Referencias

https://hackaday.com/2020/03/20/linux-fu-mapping-files/
https://stackoverflow.com/questions/41529420/mmap-file-backed-mapping-vs-anonymous-mapping-in-linux
https://medium.com/i0exception/memory-mapped-files-5e083e653b1