

Unix: Control sobre archivos Pipes Procesos

Modificando Archivos

- Existe la función:

```
int utime(char *path, struct utimbuf *utp);
```

- Cambia la fecha de acceso y modificación del i-nodo asociado al archivo "path"
- Si "utp" es NULL, se asigna la fecha actual
- La estructura "utimbuf" tiene 2 elementos:
 - actime: tiempo de acceso
 - modtime: tiempo de modificación
- Ejemplo de directorio: listdir.c (clase pasada)

Control sobre archivos

- Para realizar operaciones sobre un descriptor de un archivo existe la función:

```
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

- Dependiendo del comando "cmd" es la cantidad de argumentos que se utilizan
- Esta definida en "<fcntl.h>", retorna -1 en caso de error, en caso de éxito depende del comando

Control sobre archivos

- Usando "fcntl" podemos hacer "lock" de ciertas partes de un archivo, tanto para lectura como escritura
- Otros procesos pueden "observar" estos locks
- El sistema no supervisa los programas que leen o escriben en archivos con lock
- El registro de locking es mantenido por la estructura: "flock", con los siguientes elementos:
 - l_type: es una de las siguientes constantes:
 - F_RDLCK: indica lock de lectura (compartido)
 - F_WRLCK: indica lock de escritura (exclusivo)
 - F_UNLCK: indica que se debe borrar el lock

Control sobre archivos

- Estructura "flock":
 - l_whence: una de las siguientes constantes:
 - SEEK_SET: l_start se mide desde el inicio
 - SEEK_CUR: l_start se mide desde la posición actual
 - SEEK_END: l_start se mide desde el final
 - l_start: el "offset" en bytes, desplazamiento relativo
 - l_len: número de bytes a bloquear, debería ser positivo, si es 0 indicará bloqueo hasta EOF
 - l_pid: identificador del proceso dueño del bloqueo, usado sólo por la función F_GETLK

Control sobre archivos

- Los comandos a usar son:
 - F_SETLKW: asigna o borra un lock para un registro:


```
fcntl(fd, F_SETLKW, flock_ptr);
```
 - fd indica el archivo a bloquear y "flock_ptr" es un puntero a una estructura "flock".
 - Esta llamada puede asignar o borrar locks compartidos o exclusivos, dependiendo de flock_ptr
 - Si el lock no está disponible, esta llamada esperará hasta que otro proceso borre el lock

Control sobre archivos

- Comandos de fcntl:
 - F_SETLK: igual que F_SETLKW, pero cuando el lock no está disponible, en vez de esperar, retorna -1
 - F_GETLK:


```
fcntl(fd, F_GETLK, flock_ptr);
```

 - Buscará un "lock" para el segmento descrito en "flock_ptr"
 - Si no se encuentra, el valor de "l_type" será F_UNLCK
 - Si lo encuentra, la estructura flock será sobrescrita con la información del lock

Pipes

- Canal de comunicación entre 2 procesos
- Se comporta como una cola (FIFO) en donde lo que se escribe por un extremo se lee por el otro.
- Para crear un pipe se hace:


```
pipe(fds);
```
- "fds" es un arreglo de dos enteros de largo 2
- Después de ejecutar esta llamada, se obtienen 2 fds:


```
fds[0] permite leer
fds[1] permite escribir
```

Pipes

- Generalmente, este arreglo se entrega a 2 procesos:
 - uno cierra el canal de lectura, y escribe en el otro
 - otro cierra el canal de escritura, y lee del otro



- Ejemplo: pipe.c

Conversión entre stream y fds

- Para utilizar un “stream” de datos como un file descriptor, existe la función:

```
FILE *fp = fdopen(int st, char *modo);
```

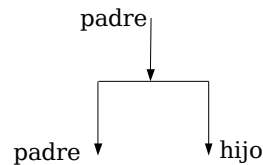
- Asocia el stream “st” al file pointer resultante, con el modo “modo”
- Se mantiene la posición de “st” en “fp”
- Devuelve NULL en caso de error, si acaba bien devuelve el FILE * correspondiente
- Se debe cerrar con “fclose”

Procesos

- Un proceso es creado con la llamada de sistema:

```
pid_t fork();
```

- Se crea un clon del proceso actual, pero cada proceso puede averiguar si el padre o el hijo



Procesos

- Cada proceso se identifica con un número único llamado “process id” (PID)
- Si la llamada a “fork” falla retorna -1, si no, retorna el identificador del proceso hijo al padre y 0 al proceso hijo
- Estructura tipo:

```
if ((pid = fork()) != 0){
    /* proceso padre */
} else {
    /* proceso hijo */
}
```

Procesos

- El proceso hijo es una copia exacta del proceso padre excepto por:
 - El proceso hijo tiene un único PID
 - El PID del padre es asignado por `fork()`, tomando el PID del proceso que ejecutó `fork()`
 - El hijo tiene su propia copia de los fd's del padre, por lo que tiene acceso a todos los archivos abiertos por el padre
 - El tiempo de ejecución del proceso hijo parte de cero
 - No hereda: alarmas, señales ni locks sobre archivos
- Un programa que llama a `fork()`, típicamente revisa los valores retornados y ejecuta las funciones dependiendo si es el padre o el hijo

Procesos

- Un proceso hijo puede ejecutar otro programa
- Existe una familia de funciones para ejecutar programas:

```
int execve(char *path, char *argv[], char *env[]);
int execl(char *path, char *arg, ...);
int execlp(char *file, char *arg, ...);
int execl_e(char *path, char *arg, ...,
            char * envp[]);
int execv(char *path, char *const argv[]);
int execvp(char *file, char *const argv[]);
```

Procesos

- La llamada a “`exec`” cambia la ejecución del proceso por un nuevo programa
- Las llamadas al nuevo programa deben llevar los argumentos necesarios para su ejecución
- El primer parámetro de las funciones “`exec`” es el nombre de archivo del programa que se quiere ejecutar
- Para el caso de “`*file`”, la ubicación de ese programa en el sistema se obtiene a partir de la variable de ambiente `PATH`

Procesos

- Por ejemplo la llamada:

```
execlp("more", "more", NULL);
```

- Busca el comando “`more`” en cada directorio incluido en el `PATH`
- Para los casos que el primer argumento es “`*path`”, no se realiza búsqueda
- El parámetro “`* envp`” se encarga de comunicar un “ambiente” al programa que se ejecutará:
 - Apunta a un arreglo de strings de la forma: `nombre=valor`
 - Debe terminar con `NULL`

Procesos

- Ejemplos de variables de ambiente:

HOME PATH PAGER EDITOR

- Para buscar un nombre de variable en el ambiente se usa:

```
char *p = getenv(char *name);
```

- El parámetro “name” apunta al nombre de la variable y “p” recibe el puntero a la línea name=value respectiva, o bien NULL si no existe

Procesos

- Para usar getenv es necesario agregar “<stdlib.h>”
- El programa que se ejecuta en la llamada a “exec” debe tener la forma:

```
int main(int argc, char *argv[])
```

- “argc” es el contador de argumentos y “argv” es un arreglo de punteros a los argumentos que recibe el programa
- El último argumento de “argv” debe ser NULL

Procesos

- Si el proceso padre ejecuta la llamada:

```
int = wait(int *status);
```

- Espera a que un hijo muera y retorna el PID del hijo
- La variable “status”, tendrá la causa del deceso:
- Existen macros para analizar el status:
 - WIFEXITED(status): es distinto de cero si el hijo terminó normalmente
 - WIFSIGNALED(status): será verdadero si el proceso hijo terminó a causa de una señal no capturada

Procesos

- Macros para analizar el status:
 - WTERMSIG(status): entrega el número de la señal que causó la muerte del proceso hijo, se puede usar sólo si WIFSIGNALED retorna verdadero
 - (ver manual)

- Como usar estas macros:

```
pid = wait(&s);
if ((WIFEXITED(s) != 0) && (WEXITSTATUS(s) != 0))
    fprintf(stderr, "Hijo finalizo con codigo %d\n",
            WEXITSTATUS(s));
if (WIFSIGNALED(s))
    fprintf(stderr, "Hijo murió con la señal: %d\n",
            WTERMSIG(s));
```

Procesos

- Si al momento de llamar “wait” ya había algún proceso muerto (zombie), la llamada retorna inmediatamente
- Un “wait” también termina si llega una señal que termina el proceso o la invocación de una función de manejo de señales
- Aparte de wait, también existe “waitpid”, para un manejo más específico:

```
pid_t waitpid(pid_t pid, int *status, int flag);
```

Procesos

- Si pid es -1 se espera a cualquier proceso (como hace wait). Si es mayor que 0 se espera al proceso que posee ese pid.
- El resto de valores posibles (0 y <-1) están relacionados con identificadores de grupos de procesos
- El parámetro “flag”, puede ser:
 - WNOHANG: hace que la llamada no se bloquee esperando el retorno
 - WUNTRACED: Usado normalmente por programas “shell”, para soportar tareas de control

Procesos

- Las funciones “wait” y “waitpid” liberan cualquier recurso que el proceso hijo esté usando
- Aunque no se revise el estado final de un hijo, es recomendable utilizar “wait” o “waitpid”:
 - Se llama a fork(), tenemos el padre y el hijo, el hijo genera otro fork(), tenemos padre, hijo y nieto, el hijo finaliza, capturado en un “wait” del padre.
 - El nieto se sigue ejecutando sin que exista un “wait” esperando por él
 - El nieto liberará todos los recursos que está utilizando una vez que finalice su ejecución

Procesos

- Como terminar un proceso:
 - Retornando desde el “main()” (return)
 - Llamando a “exit(valor)”, idéntico a “return valor”
 - Llamando a “abort()”, indica una finalización anormal, podría generar un archivo “core”, podría borrar los archivos abiertos
- Para casos normales se usa return o exit con valor 0
- La función “exit()” y “return” realizan (entre otras cosas):
 - Escribir todos los datos que están en algún buffer y cierra todos los “stream” abiertos
 - Borra los archivos temporales creados por “tmpfile()”

Procesos

- Como terminar otros procesos:

```
int kill(pid_t pid, int sig);
```

- Eliminará el proceso identificado por “pid”, retornando 0 en caso de éxito y -1 en caso de error
- En general es seguro eliminar los procesos hijos y nietos generados
- Ejemplo: fork.c