opiframe
Knowledge equals power

Opiframe Oy

Files and Signals

Content

- Files and basic I/O
- Signals and signal handling

Files and Streams

opiframe
Knowledge equals power

- For historical reasons high-level streams in glibc are handled by FILE-datatype.
- Do not try to create FILE objects yourself.
- These are managed by the library functions and you should always point to that object by FILE *.

Files and Streams

opiframe
Knowledge equals power

- Three standard streams exists:
- Standard input (stdin) which is the input source normally associated with the process.
- Standard output (stdout) which is the same as stdin but for output.
- Standard error (stderr) which is the stream where the program issues errors and diagnostics.

- These are represented by numbers 0,1 and 2 respectively (0 being stdin and so on)

Files and Streams

opiframe
Knowledge equals power

- Standard opening is done by command:
- FILE * **fopen**(const char *filename, const char *opentype)

- You can open the file for reading (type ’r’), writing (type ’w’) or appeding (type ’a’). You can also open the stream with handle ’+’ (’r+’, ’w+’ or ’a+’) which basically tells the system to open the stream for both read and write.

Files and Streams

Closing the stream is done with command:

int **fclose** (FILE * stream);

- This function causes *stream* to be closed and the connection to the corresponding file to be broken. Any buffered output is written and any buffered input is discarded.

- Closing all open streams:

int **fcloseall**(void);

Files and Streams

- Reading from streams is handled by:
- int **fgetc** (*FILE *stream*)

- This function reads the next character as an unsigned char from the stream *stream* and returns its value, converted to an int. If an end-of-file condition or read error occurs, EOF is returned instead.

Files and Streams

opiframe
Knowledge equals power

- ssize_t **getline** (*char **lineptr, size_t *n, FILE *stream*)
- This reads a line from stream storing it in a buffer and storing the buffer pointer to *lineptr
- The *lineptr should be allocated size_t *n memory with malloc.
- If the line requires more memory than *lineptr holds **getline** will do a realloc and allocate enough memory.

Files and Streams

opiframe
Knowledge equals power

- char * **fgets** (*char *s, int count, FILE *stream*)
- The **fgets** function reads characters from the stream *stream* up to and including a newline character and stores them in the string *s*, adding a null character to mark the end of the string.
- You must supply *count* characters worth of space in *s*, but the number of characters read is at most *count* − 1.
- The extra character space is used to hold the null character at the end of the string.

Files and Streams

- ssize_t **getdelim** (*char **lineptr, size_t *n, int delimiter, FILE *stream*)
- This function is like getline except that the character which tells it to stop reading is not necessarily newline.
- The argument *delimiter* specifies the delimiter character; **getdelim** keeps reading until it sees that character (or end of file).
- The text is stored in *lineptr*, including the delimiter character and a terminating null.

Files and Streams

opiframe
Knowledge equals power

- Writing to the stream is done with:
- int **fputc** (*int c, FILE *stream*)

- The **fputc** function converts the character *c* to type unsigned char, and writes it to the stream *stream*. EOF is returned if a write error occurs; otherwise the character *c* is returned.

Files and Streams

opiframe
Knowledge equals power

- int **fputs** (*const char *s, FILE *stream*)

- The function **fputs** writes the string *s* to the stream *stream*. The terminating null character is not written. This function does *not* add a newline character, either. It outputs only the characters in the string.

- This function returns EOF if a write error occurs, and otherwise a non-negative value.

Files and Streams

opiframe
Knowledge equals power

- These are the usual suspects when dealing with files. For formatted output we have various versions of printf and scanf.

int **printf** (*const char *template, ...*)

int **fprintf** (*FILE *stream, const char *template, ...*)

int **sprintf** (*char *s, const char *template, ...*)

int **scanf** (*const char *template, ...*)

int **fscanf** (*FILE *stream, const char *template, ...*)

int **sscanf** (*const char *s, const char *template, ...*)

Files and Streams

- So for example scanning for coordinates in two dimension as formatted output you would write:

```
int x, y;
scanf(”%d,%d”,&x,&y);
```

- This requires the user to enter the information formatted as requested (for example : ”1,1”).

- The f-versions are for streams and s-versions are for strings. So fprintf writes to a stream and fscanf reads from a stream.

Files and Streams

opiframe
Knowledge equals power

- Streams come in two varieties: Seekable and non-seekable.
- Example of a seekable stream would be any ordinary file. You can seek a position within the stream and start your operations there.
- A live television broadcast of olympic final in ice hockey would be a non-seekable stream, atleast on for future.

Files and Streams

opiframe
Knowledge equals power

- Usually streams are represented by something called a file descriptor.
- This is a (usually) simple handle for the actual stream.
- These descriptors can be independent or they can be linked each other.
- This affects things like position in the stream.

Files and Streams

opiframe
Knowledge equals power

- The opening and closing of any stream can be done with so-called primitives.
- These primitives are the low-level operations that more high-level functions use.
- Thus **fopen** is actually only a more sophisticated wrapper function to the actual primitive **open**.

Files and Streams

opiframe
Knowledge equals power

- The primitive **open()**:
- int **open** (*const char *filename, int flags*[, *mode_t mode*])

- This function creates and returns a new file descriptor for the file named by filename.

- Initially the position is at the start of this file.

- Argument mode must be supplied when the file is created but can be supplied on other cases also.

Files and Streams

opiframe
Knowledge equals power

- The flags argument controls how the file is to be opened. These are:

- File access flags:
- **O_RDONLY**; for read only
- **O_WRONLY**; for write only
- **O_RDWR**; for read/write

Files and Streams

opiframe
Knowledge equals power

- Open-time flags:
- O_CREAT; file will be created if doesn’t exist.
- O_EXCL; if this and O_CREAT are set, **open** will fail if file exists.
- O_NONBLOCK, this prevents **open** from blocking when opening takes time. Usually this is only required when opening something like serial port.
- O_NOCTTY, if the named file is a terminal device do not make it a controlling one. Controlling terminal devices handle things like file descriptors and job control.

Files and Streams

opiframe
Knowledge equals power

- The operating modes are:
- **O_APPEND**, enables the append mode.
- If set, then all write operations write the data at the end of the file, extending it, regardless of the current file position. This is the only reliable way to append to a file.

- **O_NONBLOCK**, The bit that enables non blocking mode for the file.
- If this bit is set, read requests on the file can return immediately with a failure status if there is no input immediately available, instead of blocking. Likewise, write requests can also return immediately with a failure status if the output can't be written immediately.

- **O_ASYNC**, The bit that enables asynchronous input mode.
- If set, then SIGIO signals will be generated when input is available.

Files and Streams

opiframe
Knowledge equals power

- The primitive **close()**:
- int **close** (*int filedes*)

- The function close closes the file descriptor *filedes*. Closing a file has the following consequences:
- The file descriptor is deallocated.
- Any record locks owned by the process on the file are unlocked.
- When all file descriptors associated with a pipe or FIFO have been closed, any unread data is discarded.

Files and Streams

opiframe
Knowledge equals power

- The reading and writing primitives are called just that: **read()** and **write()**. For seeking a location within a stream there is a function **lseek()**.

- ssize_t **read** (*int filedes, void *buffer, size_t size*)
- This function reads up to size from filedes and stores it into buffer. Return value is the actual number of bytes read. Zero is returned if nothing is read. This happens on EOF or if the otherside of the opened stream no longer is there

Files and Streams

opiframe
Knowledge equals power

- ssize_t **write** (*int filedes, const void *buffer, size_t size*)
- This function writes up to size from buffer to stream descripted by filedes. Return value is the actual number of bytes written. The error return value is handled similarly to read.

- off_t **lseek** (*int filedes, off_t offset, int whence*)
- This function seeks the location in filedes specified by offset and whence. Offset can have values of unsigned long int. Whence has three values specified by macros SEEK_SET, SEEK_CUR and SEEK_END.

Files and Streams

- Finally there is a control function for all things streams:
- int **fcntl** (*int filedes, int command, ...*)

- This function is used to change the flags or perform other operations on the stream. The command is performed on filedes.

Files and Streams

opiframe
Knowledge equals power

- Commands are:
- **F_DUPFD**, returns a duplicate (linked!) file descriptor
- **F_GETFD**, returns the flags associated with file decs.
- **F_SETFD**, sets the flags for file decs.
- **F_GETFL**, returns the flags associated with the FILE!
- **F_SETFL**, sets the flags for the FILE!

- Note that the FD ended change the actual descriptor and FL ended the file.

Signals

- A signal is a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems.

- Essentially it is an asynchronous notification sent to a process in order to notify it of an event that occurred.

Signals

- When a signal is sent to a process, the operating system interrupts the process's normal flow of execution.
- Execution can be interrupted during any non-atomic instruction.
- If the process has previously registered a signal handler, that routine is executed. Otherwise the default signal handler is executed.

Signals

opiframe
Knowledge equals power

- Linux implements signals using information stored in the **task_struct** for the process.
- The currently pending signals are kept in the signal field with a mask of blocked signals held in blocked.
- With the exception of **SIGSTOP** and **SIGKILL**, all signals can be blocked.

Signals

- If a blocked signal is generated, it remains pending until it is unblocked.
- Linux also holds information about how each process handles every possible signal
- This is held in an array of sigaction data structures pointed at by the **task_struct** for each process.

Signals

- Amongst other things it contains is either:
- The address of a routine that will handle the signal

- Flag which tells Linux that the process wishes to ignore this signal

- Flag which tells Linux to let the kernel handle the signal for it.

Signals

- Not every process in the system can send signals to every other process, the kernel can and super users can.

- Normal processes can only send signals to processes with the same uid and gid or to processes in the same process group.

Signals

- Signals are not presented to the process immediately they are generated. They must wait until the process is running again.

- Every time a process exits from a system call its signal and blocked fields are checked and, if there are any unblocked signals, they can now be delivered.

Signals

opiframe
Knowledge equals power

- Processes can elect to wait for signals if they wish, they are suspended in state Interruptible until a signal is presented.

- States of a living process are:
- **TASK_RUNNING,** process is considered by the kernel scheduler for running.
- **TASK_INTERRUPTIBLE,** process is not considered by the kernel scheduler for running but can be interrupted by a signal.
- **TASK_UNINTERUPTIBLE,** like **TASK_INTERRUPTIBLE** but cannot be interrupted by a signal.

IPC - Signals

opiframe
Knowledge equals power

- Signal handlers can be installed with the **signal()** system call.
- **typedef void (*sighandler_t)(int);**
- **sighandler_t signal(int** *signum*, **sighandler_t** *handler*);

- The **signal**() system call installs a new signal handler for the signal with number *signum*. The signal handler is set to *sighandler* which may be a user specified function, or either **SIG_IGN** or **SIG_DFL**.

IPC - Signals

- If a signal handler is not installed for a particular signal, the default handler is used.
- Otherwise the signal is intercepted and the signal handler is invoked.
- Typing certain key combinations at the controlling terminal of a running process causes the system to send it certain signals. (CTRL+C for SIGINT f.ex)

IPC - Signals

- Because signals are asynchronous, another signal can be delivered when first signal is being handled.
- This can cause signals not to be handled properly unless proper care is taken.
- Signal handling can be made to wait or the whole signal ignored by the process.

IPC - Signals

- Signal handlers should be simple and contain no code that results in unwanted side-effects suchs as **errno** changes or signal mask alterations.

- Also using non-reentrant (meaning functions that cannot be accessed while one is already running) functions is discouraged.

Signals

opiframe
Knowledge equals power

- Most important signals in Linux are:
- **SIGHUP**(1) is a signal sent to a process when its controlling terminal is closed.
- **SIGINT**(2) is terminal interrupt signal.
- **SIGQUIT**(3) is terminal quit signal. Includes core dump.
- **SIGILL**(4) is the signal sent to a process when it attempts to execute a malformed, unknown, or privileged instruction.
- **SIGTRAP**(5) is the signal sent to a process when a condition arises that a debugger has requested to be informed of.

Signals

- **SIGFPE**(8) is the signal sent to a process when it performs an erroneous arithmetic operation.
- **SIGKILL**(9) kernel handled kill signal. Causes program to terminate immediately.
- **SIGUSR**(10) user defined signal.
- **SIGSEGV**(11) the ever wonderful segmentation fault. The process tried to access restricted or unavailable memory location.
- **SIGPIPE**(13) process wrote to a pipe when no one listened.
- **SIGALRM**(14) signal raised by alarm when certain time limit has been reached.

Signals

- **SIGTERM**(15) the nice termination request for the process. Can be ignored.
- **SIGCHLD**(17) raised when child process either dies or stops.
- **SIGCONT**(18) continue if stopped
- **SIGSTOP**(19) kernel handled pause signal for process. Cannot be ignored.
- **SIGTSTP**(20) stop from the controlling terminal.

Signals

opiframe
Knowledge equals power

- In POSIX signal API there is an important data type **sigset_t**. It is defined as an opaque handle for a set of signals.
- For handling signal sets you have:
- **int sigemptyset(sigset_t *_set_);**
- **int sigfillset(sigset_t *_set_);**
- **int sigaddset(sigset_t *_set_, int _signum_);**
- **int sigdelset(sigset_t *_set_, int _signum_);**
- **int sigismember(const sigset_t *_set_, int _signum_);**

Signals

- If you want to block or unblock a signal from delivery you'll need to use **sigprocmask():**
- **int sigprocmask(int** *how***, const sigset_t ***set***, sigset_t ***oldset***);**
- **sigprocmask**() is used to change the signal mask, the set of currently blocked signals. The behaviour of the call is dependent on the value of *how*, as follows:
- **SIG_BLOCK** The set of blocked signals is the union of the current set and the *set* argument.
- **SIG_UNBLOCK** The signals in *set* are removed from the current set of blocked signals.
- **SIG_SETMASK** The set of blocked signals is set to the argument *set*.
- If *oldset* is non-null, the previous value of the signal mask is stored in *oldset*.

Signals

- Use **sigpending()** to handle signals in a signal set that have been sent but are blocked at the moment:
- **int sigpending(sigset_t **set*);**
- **sigpending**() returns the set of signals that are pending for delivery to the calling process (i.e., the signals which have been raised while blocked). The mask of pending signals is returned in *set*.

Signals

- Command **sigsuspend()** blocks your process until signals in a set are delivered:
- **int sigsuspend(const sigset_t ****mask***);**

- **sigsuspend**() temporarily replaces the signal mask of the calling process with the mask given by *mask* and then suspends the process until delivery of a signal whose action is to invoke a signal handler or to terminate a process.

Signals

- Sending a signal to a process can be done with **kill()** system call:
- **int kill(pid_t *pid*, int *sig*);**
- The *kill*() function shall send a signal to a process or a group of processes specified by *pid*. The signal to be sent is specified by *sig.* These are specified in signal.h.
- You need to have permission to send signals to the target**.** No permission will lead to error for **kill().**

Signals

- For different first arguments:
- **pid** > 0, signal is sent to corresponding process.
- **pid** = 0, **sig** shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the process group ID of the sender
- **pid** = -1, **sig** shall be sent to all processes (excluding an unspecified set of system processes) for which the process has permission to send that signal.
- If **pid** is negative, but not -1, **sig** shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the absolute value of **pid**

Signal handler

- As stated before signal handlers can be assigned with signal() system call.
- There is however another way which allows for more control over your signal handler.
- You can even request something called extended signal handler which brings with it a host of data courtesy of the kernel.

Signal handler

- For more controlled approach on signal handlers, use sigaction() system call:
- **int sigaction(int** *signum*, **const struct sigaction ****act*, **struct sigaction ****oldact*);

- The **sigaction**() system call is used to change the action taken by a process on receipt of a specific signal.

- You install the handlers per signal basis. One sigaction() call per signal.

Signal handler

- The struct sigaction is defined as:

```
struct sigaction {
void (*sa_handler)(int);
void (*sa_sigaction)(int, siginfo_t *, void*);
sigset_t sa_mask;
int sa_flags;
void (*sa_restorer)(void); }
```

- You use either the **sa_handler**, for normal signal handlers or **sa_sigaction**, for extended signal handlers, not both.

Signal handler

- **sa_flags** are the flags specified for the operation. For example:
- **SA_SIGINFO,** Use the extended signal handler. The signal handler takes 3 arguments, not one. In this case, *sa_sigaction* should be set instead of *sa_handler.*
- **SA_RESETHAND** (**SA_ONESHOT**), use the handler only once and go back to the default handler.
- **SA_NOCLDWAIT,** If *signum* is **SIGCHLD**, do not transform children into zombies when they terminate. In short, we are not interested in our children.