UNIX programming examples

lu. 18/12/2005

This page is for those who have no time (or are too lazy) to read lots of man pages. This is also a good starting point for getting introduced to some UNIX programming concepts and to UNIX IPC (that's Inter-Process Communication). The page covers several programming topics under UNIX in a practical fashion. Each topic has one or more full source code examples, for further clarification. Of course, when the tight details are needed, a complete reading of the appropriate man page is necessary.
While the Internet is rich with sites giving fragmentary examples on any particular subject, this page tries to wrap up more wholly information about the most common topics.

Subjects organization

The page is organized by *subjects*. One paragraph per subject, each paragraph having its *notes*, one *summary* and one or more *source code examples*. Notes are made of a brief intro on theory, and then infos about how UNIX implements the thing. Notes are meant not to be read by developers who already roughly know what they're searching about. There are summaries for that: they quickly list key aspects. POSIX conformance plays a big role here.
For every subject, some examples are reported in plain text/html. Examples are ready to be compiled an run on most OSes.
It is possible to download all the example sources in one package, and to download all the html highlighted sources in one package.

Usage of these sources and this page

Every source in this page, except where explicitly reported otherwise, has been invented, written and tested by myself. As far as I care, all these sources are public domain. In case you make conspicuous use of them, I like that you include some reference to this work in your credits. About the rest of the contents in the page, please avoid copying it fully or in parts.
In general, if you found this page useful, consider including a link to it somewhere on the web. This will make the page more easily accessible from search engines for others too.
Any comment or different feedback is welcome, just follow the link for the contact page in the top, right of this document.
I would like to thank Roberto Farina, who encouraged this work and gave me the possibility to carry it out in spite of the choking engagements in university.

General topics on UNIX programming

The following entries cover some general programming topics under UNIX:

Interprocess communication / UNIX IPC programming

The following entries cover topics specific to Inter-process communication mechanisms under UNIX:

Regular expressions

- *re_format(7)*
- *regex(3)*
- *regexp(3)*

Notes:
In the language theory, Regular Expressions are a metalanguage: a (formal) language to describe languages. Regular expressions (RE) can only describe the family of *regular* languages. Regular languages comprehend a small set of very simple languages whose properties are leftover. Talking about computers, RE refers to the language used for driving parser programs for recognizing strings. Recognize some text means verify it satisfies the properties expressed by the regular expression.
RE define a set of special characters whose meaning is shown below. Any character out of that list is matched as-is on the string against which you check the rule. The full RE is obtained by concatenating shorter regular expressions, starting from simple characters up to longer sub expressions. For example, you can define the patter "^hi.[it's me]" and it won't match the string "hi.it's me", but it will match "hi i" or "his" or even "hide". This is because the following characters have a meta meaning while not appearing escaped (prefixed by a "\"):

- `.` : one character (e.g. `ab.d` matches "abcd" and "abed")
- `^` and `$` : beginning and ending of the text (e.g. `^foo$` matches "foo" but not "fooa")
- `*` and `+` : last *atom* matched 0-or-more and 1-or-more times. An atom being any sub-regular-expression (e.g. `ab*` matches "a" or "abbb" with any number of Bs, and `head (elem )+ tail` matches an abstract "list" of at least one elem).
- `?` : last atom is "possible", appearing 0 or 1 times. (e.g. `Donald (Ervin )?Knuth` matches the name of the Professor with or without the midname part).
- `[content]` : one of the chars specified in content. Here content can be for example "abcde" or a range like "a-e". You can also use ctype's families through contents like "[:space:]" or "[:lower:]" (e.g. `[1-9][0-9]+` matches any natural number, that does not begin with 0).
- `( )`: brackets serve to "pack" sub-expressions, for readability or to associate them to quantifiers like `? * + {x,y}` (e.g. `(X (content)+ Y)*` identifies a list of 0 or more messages with header X and footer Y, but each having a non-empty payload).
- `{x,y}` : last atom matches x or y times. ",y" can be omitted. An atom is the nearest part of the rule to the bound {} by left. (e.g. `[a-z] {4,8}` may define a valid password as a lowercase word between 4 and 8 characters long)
- `|` : the string matches if it matches either the regex at left or right of this sign. (e.g `foo|bar` matches either "foo" or "bar")
- `\` : escape char, see below

You can make any of those special chars matching itself by escaping it, like "\?" for '?' or "\\" for '\'. Furthermore, you can create more complex expressions by composing REs using parenthesization. The bigger RE can be specified by several *atoms* like that: "me and (him|she)" and "((((this people:.* |that guys) are)|(noone is)) fried of mine)".

Text comparisons using REs are made easy by the regex library. See Summary.

Summary:

- what to #include: `regex.h`, (sometimes `sys/types.h` which uses off_t for ansi compiling)
- types used: `regex_t` which carries the RE pattern compiled, `regmatch_t` for reporting
- functions:
  1. `int` **`regcomp`**`(regex_t * restrict preg, const char * restrict pattern, int cflags)`
     compiles `pattern` into a binary format for performance purposes. `preg` will contain the result. `cflags` is commonly `REG_EXTENDED|REG_ICASE` (modern re format + case-insensitive matching). regcomp returns 0 on success, non-zero on failure (see `regerror`).
  2. `int` **`regexec`**`(const regex_t * restrict preg, const char * restrict string, size_t nmatch, regmatch_t pmatch[restrict], int eflags)`
     matches the rule in `preg` against `string`. `nmatch` and `cflags` are commonly 0. `pmatch` commonly NULL. 0 is returned if the string matched. If it did not match, `REG_NOMATCH` is returned. Elseway it returns an error you can check with REG_* error codes, or with the following function if you want to output human-readable messages.
  3. `size_t` **`regerror`**`(int errcode, const regex_t * restrict preg, char * restrict errbuf, size_t errbuf_size)`
     fills `errbuf` with an error message clearly describing the error `errcode`. `errbuf_size` limits the size of the error message. `preg` can be specified to make more precise the error message. `NULL` elseway.
  4. `void` **`regfree`**`(regex_t *preg)`
     frees space allocated by using `preg`. You could no longer use `preg` after this
- other things to know:
  - you can check a RE is well-formed by compiling it. The `regcomp()` function will return 0 for ok, REG_BADPAT for bad pattern.
  - regcomp compiles by default with the *OLD* RE format (REG_BASIC). You must explicitly specify the REG_EXTENDED flag to use the new format.
  - regexec supports a number of options. They are very often not needed (flag 0).

Examples:
1) Reimplementing egrep's basic behaviour. Obtained text from standard input, prints out those lines matching the regular expression got from command line. It can be useful to test REs too. Follows the source file:

```c
/*
 *  qegrep.c
 *
 *  compares its standard input against the regular
 *  expression given (on command line), and prints
 *  out those lines matching that RE.
 *
 *
 *  Created by Mij <mij@bitchx.it> on Mon Dec 29 2003.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 */

 /* max error message length */
#define MAX_ERR_LENGTH 80
 /* max length of a line of text from stdin */
#define MAX_TXT_LENGTH 600

#include <stdio.h>
 /* for --ansi (see off_t in regex.h) */
#include <sys/types.h>
 /* for index(): */
#include <string.h>
#include <regex.h>


int main(int argc, char *argv[]) {
     /* will contain the regular expression */
    regex_t myre;
    int err;
    char err_msg[MAX_ERR_LENGTH];
    char text[MAX_TXT_LENGTH];

     /* safe check on cmd line args */
    if ( argc < 2 || argc > 2 ) {
        printf("Usage:\n\tqegrep 'RE' < file\n\tOR\n\techo \"string\" | qegrep 'RE'\n");
        return 1;
    }

     /* compiles the RE. If this step fails, reveals what's wrong with the RE */
    if ( (err = regcomp(&myre, argv[1], REG_EXTENDED)) != 0 ) {
        regerror(err, &myre, err_msg, MAX_ERR_LENGTH);
        printf("Error analyzing regular expression '%s': %s.\n", argv[1], err_msg);
        return 1;
    }

     /* "executes" the RE against the text taken from stdin */
    while ( fgets(text, MAX_TXT_LENGTH, stdin) != NULL ) {
         /* we rely on the fact that text contains newline chars */
        *(index(text, '\n')) = '\0';
        if ( (err = regexec(&myre, text, 0, NULL, 0)) == 0 ) puts(text);
        else if ( err != REG_NOMATCH ) {
             /* this is when errors have been encountered */
            regerror(err, &myre, err_msg, MAX_ERR_LENGTH);
            return 2;
        }
    }

     /* meaningless here. Useful in many other situations. */
    regfree(&myre);
```

```
    return 0;
}
```

You can download the original ascii source file with this link: qegrep.c.
Compile with: `gcc --ansi --pedantic -Wall -o qegrep qegrep.c`
Run with: `./qegrep 'Newstyle Regular Expression'< textfile2scan` or `echo "string2scan" | ./qegrep 'Newstyle Regular Expression'`

Parsing the command line

- *getopt(3)*
- *getopt_long(3)*

Notes:
This hasn't to be done manually but through the `getopt()` and `getopt_long()` facilities. Both them process the command line in form of (option, argument) pairs.
The `getopt()` function is the simplest one. It parses *short options* (i.e. like "`-b -a -c -x option`" or equivalently "`-bcax option`"). The `getopt_long()` function parses *long options* (i.e. like "`--big-option --anotherone --check --extra=option`", case sensitive!).
Getopt_long also supports short options, and short-long options aliasing too: it is a actually super-set of getopt, a more difficult one.
The concept behind the getopt* functions is easy. They are both buffered functions (so they're not thread safe). They are given the array of command line parameters (`argc` and `argv`). They walk the array looking for any of the recognized options (a list specified by the programmer). When one is encountered, they return a reference to it, and possibly another one to its argument. The way they offer parsing is *reactive*: the programmer should specify the operations to be done after each call returned, wrt the option encountered.
For `getopt()`, the list of known options is a simple string like "`abx:c`". Every single character in the string is a known option. A character followed by a colon requires an argument. The order isn't regarded, but recommended in case of many options. Every time it's called, `getopt()` returns the next option in argv if it's recognized, "?" if it's not recognized, or "-1" if no more options have been found (exhaustion).
Several global variables provided by an header file, the `opt*` vars, are set for carrying more info about the option. `optopt` points out the index of the option in *argv, and `optarg` points to the string which is argument of the current option, if any (unspecified otherwise).
For `getopt_long()`, the list of recognized options is still the option string as formerly, plus a support array, whose elements are each a structure describing an option: (name, has_argument, short_name_or_val). In the latter, `name` and `has_argument` are intuitive; `short_name_or_val` can be set to a recognized *short* option (an short-long aliasing will then occur), or to an arbitrary value to which set a memory location (see summary and examples below for details). `getopt_long()` return value is like `getopt()`, but 0 is returned when a long option is recognized.
Some automatic debugging is provided by these functions. They autonomously warn the user about missing arguments and unrecognized options. This default can be set off by setting to 0 the `opterr` global variable. In this case the return value is used to inspect what error have been found, if any: "?" for unknown option, ":" for missing argument.

Summary:

- what you have to #include: `getopt.h`
- global variables and types used:

```
extern char *optarg;
extern int opterr;
extern int optind;
/* and others ... */

struct option {     /* only for getopt_long */
    const char *name;
    int has_arg;
    int *flag;      /* not null => val takes the value of what's pointed by it */
    int val;
}
```

- functions:
  1. `int` **`getopt`**`(int` **`argc`**`, const char` **`*argv[]`**`, const char` **`*optstring`**`)`
     parses short options. `argc` and `argv` are self explaining. `optstring` is the list of recognized options, one character per option and colons for arguments, as explained above in Notes. Each call returns the option recognized, "?" if the option isn't recognized, or "-1" when option string is exhausted, and sets several `opt*` variables including `optarg` (eventually points to the argument string for the option), `optopt` (index of the last option recognized in `argv`), and `opterr` (whether automatic error prompting is enabled to the end user).
  2. `int` **`getopt_long`**`(int` **`argc`**`, const char` **`*argv[]`**`, const char` **`*optstring`**`, struct option` **`options[]`**`, int` **`*index`**`)`
     parses long options. `argc` and `argv` are self explaining. `optstring` is the list of the "short" options recognized, one character per option and colons for arguments, as explained above in Notes. `options` is an array of option structures for "long" options, whose last element must be 0 filled (see example 2).
     There isn't any restriction for the relation occurring between the sets of long and short options. `index` is typically NULL; otherwise, the variable it points is set to the index of the long option recognized in the `option[]` array.

Returns -1 on errors, 0 when only-long options are recognized, and the option character when short or short-long aliased options (optstring ones, those without an entry in the option array and those whose entry has flag set to NULL).

Examples:
1) A simple program that parses the command line option the classic way. Recognizes and handles both argument-less and -ful options, and is sensible to option repeated. Try executing without options, with repeated options (e.g. "-a -b -a") and with/without arguments for "-x".

```
/*
 *  getopt-short.c
 *
 *  Parsing short-options command lines with getopt.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 07/08/05.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 *
 */


#include <stdio.h>
#include <unistd.h>

#define OPT_NUM     4       /* a, b, c and x (h is not relevant here) */


int main(int argc, char *argv[]) {
    int want_opt[OPT_NUM];     /* want option? */
    char ch;                   /* service variables */
    int i;
    char *my_argument;


    /* init want_opt array */
    for (i = 0; i < OPT_NUM; i++)
       want_opt[i] = 0;

    /* optstring: a, b, c, h; and x taking an argument */
    while ((ch = getopt(argc, argv, "abchx:")) != -1) {     /* getopt() iterates over argv[] */
        switch (ch) {       /* what getopt() returned */
            case 'a':       /* a has been recognized (possibly for the second or more time) */
                want_opt[0] = want_opt[0] + 1;     /* remember about a */
                break;
            case 'b':       /* b */
                want_opt[1]++;
                break;
            case 'c':
                want_opt[2]++;
                break;
            case 'x':
                want_opt[3]++;
                my_argument = optarg;   /* preserve the pointer to x' argument */
                break;
            case 'h':   /* want help */
            default:
                /* no options recognized: print options summary */
                printf("Usage:\n%s [-a] [-b] [-c] [-h] [-x]\n", argv[0]);

                /* typically here:
                exit(EXIT_FAILURE);
                */
        }
    }

    /* print out results */
    printf("You requested:\n");
    if (want_opt[0]) printf("a [%d]\n", want_opt[0]);
    if (want_opt[1]) printf("b [%d]\n", want_opt[1]);
    if (want_opt[2]) printf("c [%d]\n", want_opt[2]);
    if (want_opt[3]) printf("x [%d]: %s", want_opt[3], my_argument);  /* but only the last one rests */

    printf("\n");
    return 0;
}
```

You can download the original ascii source file with this link: getopt-short.c.
Compile with: `gcc --ansi --pedantic -Wall -o getopt-short getopt-short.c` , or via the makefile in the package.
Run with: `./getopt-short -a -r -b -x -a -x foo -x`

---

2) Parsing long options. This exemplifies the long-short option aliasing, and how to handle long options without a short counterpart.

```
/*
 *  getopt-long.c
 *
 *  Parsing long-option command lines with getopt.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 11/08/05.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 *
 */


#include <stdio.h>
/* for getopt_long() */
#include <getopt.h>


int main(int argc, char *argv[]) {
    char ch;                   /* service variables */
    int long_opt_index = 0;
    int longval;
```

```
    char *my_argument;
    struct option long_options[] = {        /* long options array. Items are all caSe SensiTivE! */
        { "add", 0, NULL, 'a'   },      /* --add or -a  */
        { "back", 0, NULL, 'b'  },      /* --back or -b */
        { "check", 0, &longval, 'c' },  /* return 'c', or return 0 and set longval to 'c' if "check" is parsed */
        { "extra", 1, &longval, 'x' },
        { 0,    0,    0,    0   }       /* terminating -0 item */
    };


    while ((ch = getopt_long(argc, argv, "abchx:", long_options, &long_opt_index)) != -1) {
       switch (ch) {
            case 'a':   /* long_opt_index does not make sense for these */
                /* 'a' and '--add' are confused (aliased) */
                printf("Option a, or --add.\n");
                break;
            case 'b':
                /* 'b' and '--back' are confused (aliased) */
                printf("Option b, or --back.\n");
                break;
            case 'c':
                /* 'c' and '--check' are distinguished, but handled the same way */
                printf("Option c, not --check.\n");
                break;
            case 'x':
                my_argument = optarg;
                printf("Option x, not --extra. Argument %s.\n", my_argument);
                break;
            case 0:     /* this is returned for long options with option[i].flag set (not NULL). */
                        /* the flag itself will point out the option recognized, and long_opt_index is now relevant */
                switch (longval) {
                    case 'c':
                        /* '--check' is managed here */
                        printf("Option --check, not -c (Array index: %d).\n", long_opt_index);
                        break;
                    case 'x':
                        /* '--extra' is managed here */
                        my_argument = optarg;
                        printf("Option --extra, not -x (Array index: %d). Argument: %s.\n", long_opt_index, my_argument);
                        break;
                    /* there's no default here */
                }
                break;
            case 'h':   /* mind that h is not described in the long option list */
                printf("Usage: getopt-long [-a or --add] [-b or --back] [-c or --check] [-x or --extra]\n");
                break;
            default:
                printf("You, lamah!\n");
       }
    }

    return 0;
}
```

You can download the original ascii source file with this link: getopt-long.c.
Compile with: `gcc --ansi --pedantic -Wall -o getopt-long getopt-long.c` , or via the makefile in the package.
Run with: `./getopt-long -a --foobar --back -x foo -b --check`.

Tasks, fork(), wait() and waitpid()

- *fork(2)*
- *wait(2)*
- *waitpid(2)*
- *wait4(2)*

Notes:
A task is an independent flow of instructions associated with a memory space and a private context. You can clone a task into another one which will consequently run concurrently and, once again, independently to it parent. The terms "parent", "child", "grandfather" etc. are common for describing the hierarchical relationship occurring between processes.
Immediately after the fork, the parent and the child are identical for both the code, the data and the context. They can modify everything in their environment independently because their contexts are separate. Since the code segment is identical, they will also run the same instructions just after the fork. Looking at the return value of the forking function is the only way to distinguish who's who. The parent is returned the Process ID of the child, whereas the child always gets the value "0".
The way to make this whole mechanism useful is actually to place a branch just after the fork and decide what to do depending on who is the parent and who is the child. The most common example to describe when tasks come useful is a server accepting connections. You can run a process that listens for incoming connections, and forks to a new child every time a new one occurs, delegating it to serve the client. This design is cleaner than creating a monolithic process which does everything itself.
POSIX defines *pid_t fork(void)* as function to fork a process. The fork() primitive can fail, for example, when the total number of concurrent processes allowed by the OS has been over-gone, or if some resource got exhausted. In this case no children are produced.
An important point to get is that nothing is said about which between the parent or the child executes first right after fork() returns. Well written code requires this is taken into account. If you postulate something about that, you're writing unreliable code.
Being processes, children can terminate and return exit values. The `wait()` function lets the parent fetch this value. It blocks the calling process until the first child terminates; then, it returns the pid of the child and stores its exit status. When a child terminates, the OS signals its parent with a SIGCHLD. When the parent produced several children and one particular is expected to terminate, the `waitpid()` function is to be used. Waitpid can also be instructed not to be blocking, thereby it is used frequently in place of `wait()` too. The SIGCHLD signal can also

be bound to a "purging" wait() function for skipping zombies accumulation.
If a process terminates before any of its children, the process with PID 1 (init(8)) receives the paternity of them.

Summary:

- what you have to #include: `sys/types.h`, `unistd.h`, `sys/wait.h`
- types used: `pid_t` for identifying processes via PID
- functions:
  1. `pid_t` **`fork`**`(void)`
     The caller process creates a clone of itself which will run independently after the call. The caller is returned the pid of the child. The child is returned 0. On errors `fork()` returns -1 without creating any child, and `errno` is set properly.
  2. `pid_t` **`wait`**`(int *status)`
     blocks until one of the children exits, stores its exit status in `status` and returns its pid, or -1 (setting `errno`) if errors happened.
  3. `pid_t` **`waitpid`**`(pid_t wpid, int *status, int options)`
     Does the same as `wait()`, but waiting for a specific child identified by `wpid`. When `wpid` is -1, it waits for any children the same way `wait()` does. The `options` argument can be used to drive the behaviour of the function. The most commonly used option is WNOHANG, whose meaning is "do not wait, just check and exit if no children terminated". The function returns the pid of the child awaited, 0 if WNOHANG has been requested and the queue of terminated children is empty, -1 setting `errno` on errors.
  4. `pid_t` **`wait4`**`(pid_t wpid, int *status, int options, struct rusage *rusage)`
     this is the most powerful function for getting child processes informations. It acts pretty the same way as `waitpid()`, but fills the `rusage` struct with statistics about the process awaited such as resources utilization and pages reclaimed.
  5. `pid_t` **`getpid`**`(void)`
     Returns the Process ID of the caller. Always successful.
  6. `pid_t` **`getppid`**`(void)`
     Returns the parent's Process ID of the caller. Always successful.
- macros (for `waitpid()` and `wait4()`):
  1. **`WIFEXITED`**`(status)`
     returns true (non-zero) if `status` denotes a normal process exit status
  2. **`WIFSIGNALED`**`(status)`
     returns true if the process terminated after receiving a signal
  3. **`WIFSTOPPED`**`(status)`
     returns true if the process is not running but has been stopped, and can be restarted
  4. **`WEXITSTATUS`**`(status)`
     returns the process exit status information carried by `status`
  5. **`WTERMSIG`**`(status)`
     returns the number of the signal which made the process terminate. If the process terminated normally, this is meaningless.
  6. **`WSTOPSIG`**`(status)`
     returns the number of the signal which put the process into stop.

Examples:
1) Practicing with paternities, pids, exit statuses. The process forks to a child, then waits for someone to terminate. The child forks into another child, and wait(pid) for it to terminate. This latter child forks into another child, doesn't expect for anyone and terminates.

```c
/*
 *  procsmess.c
 *
 *  A completely meaningless example for tasks.
 *  The parent process forks, then wait() for output
 *  the exit status of who exited. The child forks
 *  and does the same with waitpid(). The grandchild
 *  finally forks and suddendly terminate.
 *
 *
 *  Created by Mij <mij@bitchx.it> on Wed Dec 31 2003.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 */

 /* for printf() and perror() */
#include <stdio.h>
 /* for fork() */
#include <sys/types.h>
#include <unistd.h>
 /* for wait*() */
#include <sys/wait.h>

int main() {
    pid_t mypid, childpid;
    int status;

     /* what's our pid? */
    mypid = getpid();
    printf("Hi. I'm the parent process. My pid is %d.\n", mypid);

     /* create the child */
    childpid = fork();
    if ( childpid == -1 ) {
        perror("Cannot proceed. fork() error");
        return 1;
    }


    if (childpid  == 0) {
```

```
         /* then we're the child process "Child 1" */
        printf("Child 1: I inherited my parent's pid as %d.\n", mypid);

         /* get our pid: notice that this doesn't touch the value of parent's "mypid" value */
        mypid = getpid();
        printf("Child 1: getppid() tells my parent is %d. My own pid instead is %d.\n", getppid(), mypid);


         /* forks another child */
        childpid = fork();
        if ( childpid == -1 ) {
            perror("Cannot proceed. fork() error");
            return 1;
        }

        if (childpid == 0) {
             /* this is the child of the first child, thus "Child 2" */
            printf("Child 2: I hinerited my parent's PID as %d.\n", mypid);

            mypid = getpid();
            printf("Child 2: getppid() tells my parent is %d. My own pid instead is %d.\n", getppid(), mypid);

            childpid = fork();
            if ( childpid == -1 ) {
                perror("Cannot proceed. fork() error");
                return 1;
            }

            if (childpid == 0) {
                 /* "Child 3" sleeps 30 seconds then terminates 12, hopefully before its parent "Child 2" */
                printf("Child 3: I hinerited my parent's PID as %d.\n", mypid);

                mypid = getpid();
                printf("Child 3: getppid() tells my parent is %d. My own pid instead is %d.\n", getppid(), mypid);

                sleep(30);
                return 12;
            } else   /* the parent "Child 2" suddendly returns 15 */  return 15;
        } else {
             /* this is still "Child 1", which waits for its child to exit */
            while ( waitpid(childpid, &status, WNOHANG) == 0 ) sleep(1);

            if ( WIFEXITED(status) ) printf("Child1: Child 2 exited with exit status %d.\n", WEXITSTATUS(status));
            else printf("Child 1: child has not terminated correctly.\n");
        }
    } else {
         /* then we're the parent process, "Parent" */
        printf("Parent: fork() went ok. My child's PID is %d\n", childpid);

         /* wait for the child to terminate and report about that */
        wait(&status);

        if ( WIFEXITED(status) ) printf("Parent: child has exited with status %d.\n", WEXITSTATUS(status));
        else printf("Parent: child has not terminated normally.\n");
    }

    return 0;
}
```

You can download the original ascii source file with this link: procsmess.c.
Compile with: `gcc --ansi --pedantic -Wall -o procsmess procsmess.c`
Run with: `./procsmess`

---

2) Handling lots of children. The parent loops forking. Each child does nothing and suddenly terminates. The parent cycles on waitpid to keep the exit statuses queue the shortest it can. Just put a "sleep(1);" into children to see a memory exhaustion or MAXPROCS reached. A desktop user will never see such an occurrence besides bugs, while servers can occasionally run into problems on this topic.

```
/*
 *  forkvswaitpid.c
 *
 *  The parent loops forking and waiting.
 *  Notice the behaviour of both the post-fork()
 *  and post-waitpid() messages.
 *
 *
 *  Created by Mij <mij@bitchx.it> on Wed Dec 31 2003.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;
    int status;

    printf("Hi. Your OS would get a bit busy right now. Please kill pressing ctrl+C\n");

    while (1) {
        pid = fork();

        if ( pid == -1 ) {
            perror("Error on fork()");
            return 1;
        }

        if (pid == 0) {
             /* we're the child */
            printf("One more child born.\n");
            return 0;
```

```
        } else
             /* we're the parent and we want to purge the queue of childs exited */
            while ( waitpid(-1, &status, WNOHANG) > 0 ) printf("One more child dead.\n");
    }

    return 0;
}
```

You can download the original ascii source file with this link: forkvswaitpid.c.
Compile with: `gcc --ansi --pedantic -Wall -o forkvswaitpid forkvswaitpid.c`
Run with: `./forkvswaitpid`

Threads

- *pthread(3)*
- *and any function it lists*

Notes:
Threads are frequently reported as "alternatives to tasks". Since threads allow to satisfy a wider set of requirements than tasks, it's not fair to talk about them in these terms.
A thread is an independent stream of instructions running inside the scope of a process. A task can create several threads which will run concurrently. Threads live in the same context of their caller. Since their creation does not require the whole process context to be replicated in memory (as in fork()), they are way faster than tasks. The same questioning about context also plays for the lower scheduling overhead.
Since threads aren't independent of each other, they are also more difficult to use.
From the point of view of the developer, a thread is a function. The caller may thread the function to make it executing concurrently with the main stream of instructions. The function can access the whole context of the process it's running inside, e.g. global vars, open files and other functions.
Each instance has its own stack record, then an autonomous local scope. Multiple concurrent instances of a function will not share local variables then.
The real deal with threads is controlling overlaps and inconsistencies. Designing threaded applications is difficult. You must take care of your own code, but you must also take into account others' code. Libcalls you use must be thread-safe as well as *your* functions. In order to write thread-safe code, you must design thread-safe algorithms and involve thread-safe functions. Whether a library is thread-safe or not is normally reported explicitly on its man pages or in its source code.
*pthread*s are the standard thread API proposed by POSIX, and implemented by most of the POSIX-compliance OSes.
pthreads define a set of functions that let the programmer handle threads in a simple manner. They're partitioned into four main families depending on their goal: thread, attribute objects, Mut(ual)Ex(clusion) and condition variable routines. This page will just cover the first two families, the most commonly used. If you need to do something from the other ones, then you don't just need to quickly carry out a threaded program, and you might spend the time to carefully read the actual manpages.

Summary

- despite tasks, threads *do* share address-spaces with their caller
- to write safe code using threads, thread-safeness of every function involved ist something you must make sure
- since threads share their context with the creator, a certain attention should be paid to possible side effects
- what you have to #include: `pthread.h`
- types used: `pthread_t`, `pthread_attr_t`
- thread functions (one more time: -partial-):
    1. `int` **`pthread_create`**`(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`
       raises a thread executing the function `start_routine` (completely ignoring their arguments and return values). `thread` is set to the threadID assigned. Thread's attributes can be disposed by properly setting `attr`, or let defaults by passing NULL. `arg` is the only way to pass arguments to the routine, always passed by reference and casted to `void *`, or NULL for no arguments. On success 0 is returned.
    2. `void` **`pthread_exit`**`(void *value_ptr)`
       makes the calling thread terminate, offering `value_ptr` to who joins the exiting thread. (see below) In any thread except the one which runs `main()`, this is completely equivalent to what the `return` statement does.
    3. `pthread_t` **`pthread_self`**`(void)`
       returns the thread id.
    4. `int` **`pthread_join`**`(pthread_t thread, void **value_ptr)`
       suspends the calling stream of execution until `thread` terminates. `thread`'s return status made available by `return` of `pthread_exit` is stored in `value_ptr`. Returns 0 on success. Non-zero on failure
    5. `int` **`pthread_detach`**`(pthread_t thread)`
       sets `thread` non-joinable. When `thread` terminates, its return value will be ignored. Returns 0 on success, non-zero on failure. This does *not* put the target thread to termination.
    6. `int` **`pthread_cancel`**`(pthread_t thread)`
       cancels the target thread. The cancellation is done dependently on some attributes of `thread` itself.
- attribute obj functions:

1. `int` **`pthread_attr_init`**`(pthread_attr_t *attr)`
   initializes the attribute object `attr` with defaults thread attributes. These could then be customized with the functions below.
2. `int` **`pthread_attr_setdetachstate`**`(pthread_attr_t *attr, int detachstate)`
   sets `attr`'s detach status to detachstate. Default is commonly non-detachable, but it's recommended to be explicit to write portable code.
3. `int` **`pthread_attr_getdetachstate`**`(pthread_attr_t *attr, int *detachstate)`
   writes the detach status information contained in the thread attribute object `attr` into `detachstate`. 0 returned on success. Non-zero on error.
4. `int` **`pthread_attr_destroy`**`(pthread_attr_t *attr)`
   frees the space used for the attribute object `attr`.

Examples:
The following examples are from "Using POSIX Threads: Programming with Pthreads" by Brad nichols, Dick Buttlar, Jackie Farrell O'Reilly & Associates, Inc. I find they're quite complete, so I'm not going to spend time to write other ones. Unfortunately they're not documented. I added a short description of their behaviour on the top of their source code, and therein embedded some comment. Maybe in the future i'll write something more complete and clear (and better written) than them.

1) This shows out how to create and "join" threads. "Joining" for threads is analogue to "waiting" for tasks.

```
/*********************************************************
 * An example source module to accompany...
 *
 * "Using POSIX Threads: Programming with Pthreads"
 *     by Brad nichols, Dick Buttlar, Jackie Farrell
 *     O'Reilly & Associates, Inc.
 *
 *********************************************************
 * simple_threads.c
 *
 * Simple multi-threaded example.
 * Creates two threads. While doing different things, they
 * both access and modify variables with global scope.
 * Those vars have been tought to be modified this way, so
 * this is *not* an example of a thread side effect. If each
 * thread'd been accessing the same variable, they could create
 * such kind of problems.
 */
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

void do_one_thing(int *);           /* first function to thread */
void do_another_thing(int *);       /* second function to thread */
void do_wrap_up(int, int);          /* after joining threads... */

int r1 = 0, r2 = 0;

extern int
main(void)
{
   /* ids for the first and the second thread */
  pthread_t thread1, thread2;

   /* creating the first thread. retval != 0 means troubles */
  if (pthread_create(&thread1,
         NULL,
         (void *) do_one_thing,
         (void *) &r1) != 0)
    perror("pthread_create"), exit(1);

   /* creating the first thread. retval != 0 means troubles.
    * its argument is passed with a void * casting as requested
    * by pthread_create. The actual function expects an int. */
  if (pthread_create(&thread2,
         NULL,
         (void *) do_another_thing,
         (void *) &r2) != 0)
    perror("pthread_create"), exit(1);

   /* waiting for the first thread to terminate.
    * Thread's return(/exit) value gets discarded. */
  if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"),exit(1);

   /* waiting for the second thread */
  if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"),exit(1);

  do_wrap_up(r1, r2);

  return 0;
}

void do_one_thing(int *pnum_times)
{
  int i, j, x;

  for (i = 0;  i < 4; i++) {
    printf("doing one thing\n");
    for (j = 0; j < 10000; j++) x = x + i;
    (*pnum_times)++;
  }

}

void do_another_thing(int *pnum_times)
{
  int i, j, x;
```

```
  for (i = 0;  i < 4; i++) {
    printf("doing another \n");
    for (j = 0; j < 10000; j++) x = x + i;
    (*pnum_times)++;
  }

}

void do_wrap_up(int one_times, int another_times)
{
  int total;

  total = one_times + another_times;
  printf("All done, one thing %d, another %d for a total of %d\n",
    one_times, another_times, total);
}
```

You can download the original ascii source file with this link: simple_threads.c.
Compile with: `gcc --pedantic -Wall -o simple_threads simple_threads.c`
Run with: `./simple_threads`

---

2) A precious example on Mutual Exclusion.

```
/*********************************************************
 * An example source module to accompany...
 *
 * "Using POSIX Threads: Programming with Pthreads"
 *     by Brad nichols, Dick Buttlar, Jackie Farrell
 *     O'Reilly & Associates, Inc.
 *
 *********************************************************
 * simple_mutex.c
 *
 * Simple multi-threaded example with a mutex lock.
 * Does the same as the example above (see the link below)
 * but with mutual exclusion. Any time a thread starts, it
 * request the thread lock. Just one thread is executing
 * anytime. The others must wait it to unlock before proceeding.
 * quickly comment out by Mij <mij@bitchx.it> for
 * http://mij.oltrelinux.com/devel/unixprg/
 *
 * (also modified a bit on the code itself for clarity and
 * expressiveness purposes)
 */
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

void do_one_thing(int *);       /* first thread */
void do_another_thing(int *);       /* second thread */
void do_wrap_up(int, int);

int r1 = 0, r2 = 0, r3 = 0;
pthread_mutex_t r3_mutex=PTHREAD_MUTEX_INITIALIZER; /* for mutex locking */

extern int
main(int argc, char **argv)
{
   /* thread ids */
  pthread_t thread1, thread2;

  if (argc > 1)
    r3 = atoi(argv[1]);

   /* creating the first thread */
  if (pthread_create(&thread1,
         NULL,
         (void *) do_one_thing,
         (void *) &r1) != 0)
    perror("pthread_create"),exit(1);

   /* creating the second thread */
  if (pthread_create(&thread2,
         NULL,
         (void *) do_another_thing,
         (void *) &r2) != 0)
    perror("pthread_create"),exit(1);

   /* expecting the first thread to terminate */
  if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"), exit(1);

   /* expecting the second thread to terminate */
  if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"), exit(1);

  do_wrap_up(r1, r2);

  return 0;
}

void do_one_thing(int *pnum_times)
{
  int i, j, x;
  pthread_t ti;

  ti = pthread_self();      /* which's our id? */
  pthread_mutex_lock(&r3_mutex);
   /* this is the segment containing sensitive operations.
    * So we need to keep it alone from concurrency for safeness. */
  if(r3 > 0) {
```

```
    x = r3;
    r3--;
  } else {
    x = 1;
  }
   /* sensitive code end */
  pthread_mutex_unlock(&r3_mutex);

  for (i = 0;  i < 4; i++) {
    printf("doing one thing\n");
      for (j = 0; j < 10000; j++) x = x + i;
      printf("thread %d: got x = %d\n", (int)ti, x);
    (*pnum_times)++;
  }

}

void do_another_thing(int *pnum_times)
{
  int i, j, x;
  pthread_t ti;

  ti = pthread_self();
  pthread_mutex_lock(&r3_mutex);
  if(r3 > 0) {
        x = r3;
        r3--;
  } else {
        x = 1;
  }
  pthread_mutex_unlock(&r3_mutex);

  for (i = 0;  i < 4; i++) {
    printf("doing another \n");
    for (j = 0; j < 10000; j++) x = x + i;
    printf("thread %d: got x = %d\n", (int)ti, x);
    (*pnum_times)++;
  }

}

void do_wrap_up(int one_times, int another_times)
{
  int total;

  total = one_times + another_times;
  printf("All done, one thing %d, another %d for a total of %d\n",
    one_times, another_times, total);
}
```

You can download the original ascii source file with this link: simple_mutex.c.
Compile with: `gcc --pedantic -Wall -o simple_threads simple_mutex.c`.
Run with: `./simple_mutex`

IPC: signals

- *kill(2)*
- *raise(3)*
- *signal(3)*

Notes:
A signal is a very primitive kind of message. Process A can send process B a signal. If B associated an handler to the given signal, that code will be executed when the signal is received, interrupting the normal execution.
Originally, signals were simply unary values (received or not, like a beep): when a process received the signal, it always reacted the same manner (historically, terminating, that also explains the current name of the function you'll see below for sending signals). Recently, signals have been "type-fied" (think about beeps of different frequencies): when a process receives a signal, it reacts differently depending on the signal's *type*. This type is an integer value. Some values have been conventionally standardized and associated with symbolic names: (15, TERM) and (20, CHLD) are popular examples. The full list is defined in `signal.h`.
Processes send signals with the `kill()` function. A process may associate an handler function to a specific signal type with the `signal()` function.
Signal dispatching is controlled by the operating system. A process A can signal another process only if they belong to the same user. Processes run by a superuser can signal every process. Signals are often sent by the OS itself, e.g. to kill a process when its execution caused problems like memory segment violation. For some of these signals, the OS inhibits custom handlers: SIGSTOP and SIGKILL will always make the process respectively stop and die.
Two standard handlers have been provided with the standard C library: `SIG_DFL` causing process termination, and `SIG_IGN` causing the signal to get ignored. When an handler fires for a signal, it won't be interrupted by other handlers if more signals arrive.
As last note: signals are not queued (somehow, "best effort"). In order to notify signals, the OS holds a bitmask for every process. When the process receives the signal *n*, the *n*th bit in this bitmask is set. When the handler function terminates, it's reset to 0. Thus, if the process is not scheduled for execution (some of its user-space code) before other signals of the same type arrive, only the last will be notified by the process. The others are lost.

Summary:

- what to #include: `signal.h`

- types used: `pid_t`, `sig_t`
- functions:
    1. `void` **`signal`**`(int sig_type, void (*sig_handler)(int signal_type))`
       `sig_type` is one of the signal values (or names) defined in `signal.h`. `sig_handles` is a pointer to a function taking an int argument. This function will be run in the process context when the process is hit by the signal of the specified type.
    2. `int` **`kill`**`(pid_t dest_pid, int sig_type)`
       sends a signal of type `sig_type` to the process running with PID `dest_pid`. Particular types 0 and -1 of `dest_pid` make kill() respectively sends all processes executing with the same group of the sender, and signals all the processes running in the system (only works with superuser credentials).
    3. `void` **`raise`**`(int sig_type)`
       signals itself with signal `sig_type`.
- Other things to know:
    - signals are only delivered between processes when either:
        1. the processes involved are run by the same user
        2. the raising process is run by a superuser
    - signal types and type names are defined in `signal.h`
    - signals are not queued
    - handlers "terminate process" (SIG_DFL) and "ignore signal" (SIG_IGN) are already defined in `signal.h`

Examples:
1) Handling signals coming from the operating system. This example implements a solution to the problem of purging the queue of exit statuses of dead children. Whenever a child changes its status, the parent is signaled (by the OS) with "SIGCHLD". A SIGCHLD handler can be associated with this event. There are 2 important things to note for this process:

1. the signal handler may catch all the dead children, or not. The random sequence delaying forks contributes to signals overlapping phenomenon.
2. the raise of the signal handler (`child_handler()`) turn the process back in a "running" state. In this case, it breaks the sleep() functions before the exhaust their module.

```
/*
 *  sig_purgechilds.c
 *
 *  The parent process pulls a child exit status whenever the OS notifies
 *  a child status change.
 *
 *  Created by Mij <mij@bitchx.it> on 04/01/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unixprg/
 *
 */

/* for printf() and fgetc() */
#include <stdio.h>
/* for fork() */
#include <sys/types.h>
#include <unistd.h>
/* for srandom() and random() */
#include <stdlib.h>
/* for time() [seeding srandom()] */
#include <time.h>
/* for waitpid() */
#include <sys/wait.h>
/* for signal(), kill() and raise() */
#include <signal.h>

/* how many childs to raise */
#define NUM_PROCS   5


/* handler prototype for SIGCHLD */
void child_handler(int);

int main(int argc, char *argv[])
{
    int i, exit_status;


    /* execute child_handler() when receiving a signal of type SIGCHLD */
    signal(SIGCHLD, &child_handler);

    /* initialize the random num generator */
    srandom(time(NULL));


    printf("Try to issue a \'ps\' while the process is running...\n");


    /* produce NUM_PROCS childs */
    for (i = 0; i < NUM_PROCS; i++) {
        if (! fork()) {
            /* child */

            /* choosing a random exit status between 0 and 99 */
            exit_status = (int)(random() % 100);
            printf(" -> New child %d, will exit with %d.\n", (int)getpid(), exit_status);

            /* try to skip signals overlapping */
            sleep((unsigned int)(random() % 3));

            /* choosing a value to exit between 0 and 99 */
            exit(exit_status);
        }
```

```
        /* father */
        sleep((unsigned int)(random() % 2));
    }

    /* checkpoint */
    printf("parent: done with fork()ing.\n");

    /* why this is not equivalent to sleep(20) here? */
    for (i = 0; i < 10; i++) {
        sleep(1);
    }
    /* all the child processes should be done now */
    printf("I did not purge all the childs. Timeout; exiting.\n");

    /* terminate myself => exit */
    kill(getpid(), 15);

    /* this won't be actually executed */
    return 0;
}


/* handler definition for SIGCHLD */
void child_handler(int sig_type)
{
    int child_status;
    pid_t child;
    static int call_num = 0;

    /* getting the child's exit status */
    child = waitpid(0, &child_status, 0);

    printf("<-  Child %d exited with status %d.\n", child, WEXITSTATUS(child_status));

    /* did we get all the childs? */
    if (++call_num >= NUM_PROCS) {
        printf("I got all the childs this time. Going to exit.\n");
        exit (0);
    }

    return;
}
```

You can download the original ascii source file with this link: sig_purgechilds.c.
Compile with: `gcc -o sig_purgechilds -Wall --ansi sig_purgechilds.c` .
Run with: `./sig_purgechilds.c` .

IPC: pipes

- *pipe(2)*

Notes:
A pipe is an uni-directional mean to send data. On UNIX systems, it's implemented with a couple of file descriptors. What is written on the first one can be read from the second one, with a FIFO order.
Pipes are one of the easiest and fastest way for implementing IPC between two processes. Commonly, a process creates two pipes X and Y and forks to a child. The child will inherit these data. The parent will message the child on X_w (the write descriptor of the X pipe), and the child will receive on X_r. In analogy, the child will write to the parent on Y_w; the parent can read from Y_r. Using the same pipe for bi-directional communication can bring high-quality messes, so the fair way to implement bi-directional communication is just to use two pipes (P1 to P2, and P2 to P1).
Thereby, it is clear that *n*(n-1)* pipes are needed for a *n* processes to intercommunicate together.
Atomicity of pipe operations is guaranteed by the OS if the length of the message is shorter than a specific value (carried by the `PIPE_BUF` macro), but if this information is relevant for you, pipes are probably not a good programming choice for your scenario.
When one of the two pipe ends is closed, the pipe becomes *widowed* (or *broken*). Writing to a widowed pipe is not permitted by the OS. On such a trial, the process is signalled with a SIGPIPE. Reading from widowed pipes always returns 0 bytes. This is often used to notify one end process the communication is finished.

Summary:

- what to #include: `unistd.h`
- functions:
    1. `int` **`pipe`**`(int *fd_couple)`
       creates a pipe and stores its file descriptors to `fd_couple[0]` (read end) and fd_couple[1] (write end). -1 is returned on errors while creating, 0 otherwise.
- Other things to know:
    1. pipes are uni-directional
    2. atomicity is guaranteed for messages shorter than `PIPE_BUF` bytes
    3. half-open pipes are told *widowed* (or *broken*) pipes. Writing on them causes a write error, reading from them always returns 0 bytes.

Examples:

1) a set of processes messaging together. Every process gets its own pipe (couple of file descriptors). When process *i* wants to message process *j*, it writes onto *j*'s pipe write end. Every process will get its messages reading from the read end of its own pipe. This is a de-genere example with processes only reading the first 2 messages they got. The full solution can be implemented with the help of the select() function.

```
/*
 *  pipes.c
 *
 *  A set of processes randomly messaging each the other, with pipes.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 05/01/05.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 *
 */

/* ... */
#include <stdio.h>
/* for read() and write() */
#include <sys/types.h>
#include <sys/uio.h>
/* for strlen and others */
#include <string.h>
/* for pipe() */
#include <unistd.h>
/* for [s]random() */
#include <stdlib.h>
/* for time() [seeding srandom()] */
#include <time.h>


#define PROCS_NUM               15          /* 1 < number of processes involved <= 255 */
#define MAX_PAYLOAD_LENGTH      50          /* message length */
#define DEAD_PROC               -1          /* a value to mark a dead process' file descriptors with */


        /*      ***                     DATA TYPES                  ***     */
/* a process address */
typedef char proc_addr;

/* a message */
struct message_s {
    proc_addr src_id;
    short int length;
    char *payload;
};


        /*      ***                     FUNCTION PROTOTYPES         ***     */
/* send message to process with id dest */
int send_proc_message(proc_addr dest, char *message);
/* receive a message in the process' queue of received ones */
int receive_proc_message(struct message_s *msg);
/* mark process file descriptors closed */
void mark_proc_closed(proc_addr process);


        /*              ***             GLOBAL VARS                 ***     */
/* they are OK to be global here. */
proc_addr my_address;                   /* stores the id of the process */
int proc_pipes[PROCS_NUM][2];           /* stores the pipes of every process involved */


int main(int argc, char *argv[])
{
    pid_t child_pid;
    pid_t my_children[PROCS_NUM];               /* PIDs of the children */
    int i, ret;
    char msg_text[MAX_PAYLOAD_LENGTH];      /* payload of the message to send */
    proc_addr msg_recipient;
    struct message_s msg;



    /* create a pipe for me (the parent) */
    pipe(proc_pipes[0]);

    /* initializing proc_pipes struct */
    for (i = 1; i < PROCS_NUM; i++) {
        /* creating one pipe for every (future) process */
        ret = pipe(proc_pipes[i]);
        if (ret) {
            perror("Error creating pipe");
            abort();
        }
    }


    /* fork [1..NUM_PROCS] children. 0 is me. */
    for (i = 1; i < PROCS_NUM; i++) {
        /* setting the child address */
        my_address = my_address + 1;

        child_pid = fork();
        if (! child_pid) {
            /* child */
            sleep(1);

            /* closing other process' pipes read ends */
            for (i = 0; i < PROCS_NUM; i++) {
                if (i != my_address)
                    close(proc_pipes[i][0]);
            }


            /* init random num generator */
            srandom(time(NULL));
```

```
            /* my_address is now my address, and will hereby become a "constant" */
            /* producing some message for the other processes */
            while (random() % (2*PROCS_NUM)) {
                /* interleaving... */
                sleep((unsigned int)(random() % 2));

                /* choosing a random recipient (including me) */
                msg_recipient = (proc_addr)(random() % PROCS_NUM);

                /* preparing and sending the message */
                sprintf(msg_text, "hello from process %u.", (int)my_address);
                ret = send_proc_message(msg_recipient, msg_text);
                if (ret > 0) {
                    /* message has been correctly sent */
                    printf("    --> %d: sent message to %u\n", my_address, msg_recipient);
                } else {
                    /* the child we tried to message does no longer exist */
                    mark_proc_closed(msg_recipient);
                    printf("    --> %d: recipient %u is no longer available\n", my_address, msg_recipient);
                }
            }

            /* now, reading the first 2 messages we've been sent */
            for (i = 0; i < 2; i++) {
                ret = receive_proc_message(&msg);
                if (ret < 0) break;
                printf("<--     Process %d, received message from %u: \"%s\".\n", my_address, msg.src_id, msg.payload);
            };

            /* i'm exiting. making my pipe widowed */
            close(proc_pipes[my_address][0]);

            printf("# %d: i am exiting.\n", my_address);
            exit(0);
        }

        /* saving the child pid (for future killing) */
        my_children[my_address] = child_pid;

        /* parent. I don't need the read descriptor of the pipe */
        close(proc_pipes[my_address][0]);

        /* this is for making srandom() consistent */
        sleep(1);
    }

    /* expecting the user request to terminate... */
    printf("Please press ENTER when you like me to flush the children...\n");
    getchar();

    printf("Ok, terminating dandling processes...\n");
    /* stopping freezed children */
    for (i = 1; i < PROCS_NUM; i++) {
        kill(my_children[i], SIGTERM);
    }
    printf("Done. Exiting.\n");

    return 0;
}


int send_proc_message(proc_addr dest, char *message)
{
    int ret;
    char *msg = (char *)malloc(sizeof(message) + 2);


    /* the write should be atomic. Doing our best */
    msg[0] = (char)dest;
    memcpy((void *)&(msg[1]), (void *)message, strlen(message)+1);

    /* send message, including the "header" the trailing '\0' */
    ret = write(proc_pipes[dest][1], msg, strlen(msg)+2);
    free(msg);

    return ret;
}


int receive_proc_message(struct message_s *msg)
{
    char c = 'x';
    char temp_string[MAX_PAYLOAD_LENGTH];
    int ret, i = 0;


    /* first, getting the message sender */
    ret = read(proc_pipes[my_address][0], &c, 1);
    if (ret == 0) {
        return 0;
    }
    msg->src_id = (proc_addr)c;

    do {
        ret = read(proc_pipes[my_address][0], &c, 1);
        temp_string[i++] = c;
    } while ((ret > 0) && (c != '\0') && (i < MAX_PAYLOAD_LENGTH));



    if (c == '\0') {
        /* msg correctly received. Preparing message packet */

        msg->payload = (char *)malloc(strlen(temp_string) + 1);
        strncpy(msg->payload, temp_string, strlen(temp_string) + 1);

        return 0;
    }


    return -1;
```

```
}


void mark_proc_closed(proc_addr process)
{
    proc_pipes[process][0] = DEAD_PROC;
    proc_pipes[process][1] = DEAD_PROC;
}
```

You can download the original ascii source file with this link: pipes.c.
Compile with: `gcc --ansi --pedantic -o pipes pipes.c`
Run with: `./pipes`

IPC: named pipes

- *mkfifo(2)*

Notes:
There isn't much to say about named pipes once you understood standard pipes. The former ones are just a simple extension to the latter. In order to create a common pipe you use the `pipe()` function, and get a reference to it in terms of file descriptors. This is the big limit of pipes: with file descriptors, they can only be shared between processes with parent-child relationship. Named pipes work much like common pipes, but they can be shared between independent processes.
You request a FIFO file to enter the filesystem with the `mkfifo()` function. This takes as argument the name you want the file to appear like. Every other process with opportune credentials could then handle this special file with the standard `open()`, `close()`, `read()` and `write()` file functions.
The first process who open the fifo is blocked until the other end is opened by someone. Of course, being named pipes actual *pipes* in first place, they can't be applied functions like `lseek()`. This is common to every stream file including pipes, FIFOs and sockets. As a difference, being named pipes actually outside of the inner, private scope of a process, they will survive the death of their creator: they are persistent. This means that you must explicitly handle the named pipe removal after you're done with it. `unlink()` is what you need for this.

Summary:

- what to #include: `sys/types.h` and `sys/stat.h`
- functions:
    1. `int` **`mkfifo`**`(char *path, mode_t mode)`
       `path` is the (path+)name of the FIFO file to create. `mode` is the file permission (see *umask(2)* and *chmod(2)*). It returns non-zero on errors.
- Other things to know:
    1. standard pipes' properties and features.
    2. pipes are accessed like standard files except for positioning which doesn't work (being they fifos...)
    3. File owner and groups are set respectively to the user the caller is running as and the owner of the directory where the fifo is created
    4. user space commands are also available for creating named pipes from the shell. This is actually the most common use, very suitable to get different processes unaware each the other to communicate.

Examples:
1) a simple example. There's not much more to address about fifos.
That's what you do to try this:

1. you create the named pipe from the command line: `mkfifo /tmp/named_pipe`
2. you compile this program, and run it
3. you write something on the pipe: `echo "this is a message" > /tmp/named_pipe`
4. you try what happens when skipping the string terminator: `echo -n "this is another message" > /tmp/named_pipe`
5. then try to load a terminator on the fifo: `echo "a third message" > /tmp/named_pipe`
6. when you're happy with your test, close the program. Then remove the pipe: `rm /tmp/named_pipe`

```
/*
 *  nampipes.c
 *
 *  simply opens a pre-created named pipe (a "fifo") and reads
 *  stuff from it as soon as there's something available.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 02/02/05.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 *
 */


#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
```

```c
#define MAX_LINE_LENGTH


int main(int argc, char *argv[]) {
    int pipe;
    char ch;


    /* we expect a named pipe has been created
     * in /tmp/named_pipe . The command
     *   $ mkfifo /tmp/named_pipe
     * is a friend to get to this point
     */
    pipe = open("/tmp/named_pipe", O_RDONLY);
    if (pipe < 0) {
        perror("Opening pipe");
        exit(1);
    }

    /* preparing to read from the pipe... */
    printf("Waiting data from the pipe... \n");

    /* reading one char a time from the pipe */
    while (1) {
        if (read(pipe, &ch, 1) < 0) {
            perror("Read the pipe");
            exit(2);
        }

        printf("%c", ch);
    }

    /* leaving the pipe */
    close(pipe);

    return 0;
}
```

You can download the original ascii source file with this link: nampipes.c.
Compile with: `gcc --ansi --Wall -o nampipes nampipes.c`
Read the instructions above, then run with: `./nampipes`

IPC: BSD sockets

- *socket(2)*
- *bind(2)*
- *listen(2)*
- *connect(2)*
- *sendto(2)*
- *recvfrom(2)*
- *[write(2) and read(2)]*

Notes:
Sockets are the representation that the operating system provides for network endpoints to processes. They originated in 4.2BSD and are today an essential feature of every UNIX system.
Sockets are given a *type* and a set of attributes that depend on it. These attributes include an address: the set of possible values this address is taken from is told *domain*. As domains are distinct for each type, sockets are typically distinguished just by the domain they are addressed in. The most commonly used domains are the UNIX domain, that uses filename-like addresses and is suitable only for in-host IPC, and the INET domain, that uses IP addresses and can be used for both in-host and inter-host IPC, but many more domains exist.
Sockets can be connected or not. Connectionful sockets require a setup phase before the actual communication occurs, and a tear down phase after. Connectionless ones don't, but require a slightly more complex semantic for sending and receiving every single message.
Connectionless sockets are point-to-anypoint and peer. This is the lifecycle of a non-connected socket:

1. it is created (here the belonging domain is specified); -- `socket()`
2. it's bound attributes; -- `bind()`
3. it is used for sending or receiving single messages; -- `sendto()` and `recvfrom()`
4. it is closed -- `close()`

Connectionful sockets are strictly point-to-point: once a connection is established, messages only travel from/to the two entities involved. Connection also makes these sockets master/slave or client/server (the one who initiate the connection is told slave or client). This is the lifecycle of a connected socket with a role of server:

1. it is created (and assigned a domain); -- `socket()`
2. it's bound attributes; -- `bind()`
3. it's open into a network port (listening); -- `listen()`
4. it accept connections; -- `accept()`
5. data is read and written on it (it represent a stream once connected); -- `write()` and `read()`
6. it is closed -- `close()`

The lifecycle of a client is similar, but features a connection request (`connect()`) in place of points {2,3,4}.
When connected, sockets appears much like a file: they can be read and written the same way that file descriptors are.
Sockets are blocking by default: functions are suspended until the requested job on the socket terminates. However, like regular file descriptors, sockets can be set non-blocking too, with `fcntl()`.

Summary:

- what to #include: `sys/types.h`, `sys/socket.h` (socket, bind, listen, accept, connect, sendto, recvfrom), `unistd.h` (write, read and close). Then, `netinet/in.h` for INET-domain sockets, or `sys/un.h` for UNIX-domain sockets
- types:
  1. **struct sockaddr**: a generic container for socket attributes; this is what expects `bind()`, but is actually instantiated every time with a domain-specific attribute structure like the ones below, eventually casted when necessary.
  2. **struct sockaddr_in**: for INET domain socket attributes; defined as follows:

```
struct sockaddr_in {
        u_char  sin_len;
        u_char  sin_family;         /* = AF_INET */
        u_short sin_port;           /* use htons() to set this */
        struct  in_addr sin_addr;
        char    sin_zero[8];        /* 8 zeros required here */
};

struct in_addr {
        in_addr_t s_addr;           /* either use a macro from netinet/in.h (eg INADDR_ANY), or inet_addr */
};
```

  3. **struct sockaddr_un**: for UNIX domain socket attributes; defined as:

```
struct  sockaddr_un {
        u_char  sun_len;                /* sockaddr len including null */
        u_char  sun_family;             /* AF_UNIX */
        char    sun_path[104];          /* path name */
};
```

- useful macros:
  1. **AF_***: the socket domain ("Address Family"); from `sys/socket.h`
  2. **INADDR_ANY**: for INET sockets, represents any possible address.
  3. **SUN_LEN(struct sockaddr_un *su)**: for UNIX sockets, returns the address of a sockaddr_un structure initialized with a pathname; from `sys/un.h`
  4. **htons** & Co.: for converting between machine and network byte order (necessary in inter-host communication)
- functions:
  1. `int` **`socket`**`(int domain, int communication_type, int protocol)`
     creates a socket and returns its identifier, or -1 on errors (and `errno` is set). Pick `domain` and `communication_type` from `sys/socket.h`, e.g. `AF_INET` and `SOCK_STREAM`. The `protocol` field is domain-dependant.
  2. `int` **`bind`**`(int socket, const struct sockaddr *saddr, int saddrlen)`
     binds the attributes carried in `saddr` to the socket. `saddrlen` is the size of the actual structure passed: e.g. SUN_LEN(saddr) for UNIX domain sockets, or sizeof(struct sockaddr_in) for INET domain sockets. Return -1 on failure (sets `errno`).
  3. `int` **`listen`**`(int socket, int maxqueuelen)`
     applicable for connected sockets (SOCK_STREAM). Makes the socket listening for connections. `maxqueuelen` tells how many pending connections are accepted at most.i Returns 0 on success, -1 on failure (`errno` set).
  4. `int` **`connect`**`(int socket, const struct sockaddr *servaddr, int servaddrlen)`
     applicable for connected sockets (SOCK_STREAM): connects the socket to the server described by `servaddr`. `servaddrlen` tells how long is the servaddr structure. Returns 0 on success, -1 on failure (and sets `errno`).
  5. **`read`**`()` and **`write`**`()` can be used for reading and writing to (already) connected sockets just the way they are used for file descriptors. They are standard functions, their description is omitted.
  6. `int` **`recvfrom`**`(int socket, void *buf, size_t len, int flags, struct sockaddr *from, int *fromlen)`
     for disconnected sockets: receives a message, and stores it in `buf` for at most `len` bytes. `flags` is normally 0. If `fromaddr` is non-NULL and its size is set in `fromlen`, the address structure of the message sender is written into it. Returns the number of bytes actually accepted, or -1 on errors (`errno` set).
  7. `int` **`sendto`**`(int socket, void *message, size_t len, int flags, struct sockaddr *toaddr, int toaddr_len)`
     for disconnected sockets: sends the first `len` bytes of the buffer `message`, to the recipient specified in `toaddr` (`toaddr_len` tells the size of the actual structure). Returns the number of bytes sent, or -1 on failure (`errno` set).
- other useful functions: `inet_addr()` & Co., `gethostbyname()` & Co.,

Examples:
N.B.: It is an habit (that I dislike) that examples about sockets include a sample client, a sample server, and mr. sample client contacts mr. sample server and they are happy and close the connection. These examples are not like that. Both connected and connectionless sockets are presented. Besides, "standalone" clients and servers are proposed: their functionality can be tested with standard tools like netcat.

1) an UDP server. A connectionless socket is opened on the local internet-protocol interface "127.0.0.1", port "12321", and a message is awaited. Once received, the message length and content are displayed, along with the IP address and port of the sender.

```c
/*
 * udpserv.c
 *
 * listens on an UDP port, accept one message and displays its content and
 * who's the sender
 *
 *
 * Created by Mij <mij@bitchx.it> on 18/12/05.
 * Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 */

#include <stdio.h>
/* socket(), bind(), recvfrom() */
#include <sys/types.h>
#include <sys/socket.h>
/* sockaddr_in */
#include <netinet/in.h>
/* inet_addr() */
#include <arpa/inet.h>
/* memset() */
#include <string.h>
/* close() */
#include <unistd.h>
/* exit() */
#include <stdlib.h>

/* maximum size available for an incoming message payload */
#define MAX_MSGLEN      100

int main() {
    int sock, err;
    char messagebuf[MAX_MSGLEN+1];
    struct sockaddr_in saddr, fromaddr;
    int fromaddr_len = sizeof(fromaddr);


    /* create a INET-domain, disconnected (datagram) socket */
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock < 0) {
        perror("In socket()");
        exit(1);
    }

    /* bind the following attributes to the socket: */
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = inet_addr("127.0.0.1");     /* localhost address */
    saddr.sin_port = htons(61321);          /* use port 61321 (with correct network byte-order */
    memset(&saddr.sin_zero, 0, sizeof(saddr.sin_zero)); /* always zero-fill this field! */

    err = bind(sock, (struct sockaddr *)&saddr, sizeof(saddr));
    if (err) {
        perror("In bind()");
        exit(1);
    }

    /* receive a message in "messagebuf" at most MAX_MSGLEN bytes long (1 is
     * spared for the trailing '\0'; store sender info in "fromaddr" */
    err = recvfrom(sock, messagebuf, MAX_MSGLEN-1, 0, (struct sockaddr *)&fromaddr, &fromaddr_len);
    if (err <= 0) {     /* otherwise, "err" tells how many bytes have been written in "messagebuf" */
        perror("in recvfrom()");
        exit(1);
    }
    messagebuf[err] = '\0';        /* NULL-terminator */

    /* close the socket */
    close(sock);

    /* displaying message length and content */
    printf("Message:\n\t@bytes: %d\n\t@payload: %s\n", err, messagebuf);
    /* displaying sender info: */
    printf("Sender:\n\t@address: %s\n\t@port: %d\n", inet_ntoa(fromaddr.sin_addr), ntohs(fromaddr.sin_port));

    return 0;
}
```

You can download the original ascii source file with this link: udpserv.c.
Compile with: `gcc -o udpserv udpserv.c`
Run the server in a terminal:
`$ ./udpserv`
Then, open another terminal and feed something on UDP on 127.0.0.1 port 61321 via UDP, e.g. with netcat:
`$ echo -n "Hello lame server." | nc -u 127.0.0.1 61321`
Also try omitting the newline (here, take "-n" away), and sending large messages (eg, cat a whole file).

---

2) A UNIX domain client. It connects to a given server, sends a message to it, expects a newline-terminated response, then displays all this and exits.

```c
/*
 * unixcli.c
 *
 * connects to an UNIX domain socket, sends a message to it, and disconnects.
 *
 *
 * Created by Mij <mij@bitchx.it> on 18/12/05.
 * Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 */

#include <stdio.h>
/* socket(), bind() */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
```

```c
/* write(), close() */
#include <unistd.h>
/* strlen() */
#include <string.h>
/* exit() */
#include <stdlib.h>

/* maximum size available for an incoming message payload */
#define MAX_MSGLEN      100
/* path of the UNIX domain socket in filesystem */
#define SERVERSOCK_PATH     "/tmp/mytmpunixsock"

int main() {
    int sock, err, i = 0;
    char ch, messagebuf[MAX_MSGLEN];
    struct sockaddr_un servaddr;


    /* create a UNIX domain, connectionful socket */
    sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("In socket()");
        exit(1);
    }

    /* connect the socket to the server socket described by "servaddr" */
    servaddr.sun_family = AF_UNIX;
    sprintf(servaddr.sun_path, SERVERSOCK_PATH);

    err = connect(sock, (struct sockaddr *)&servaddr, sizeof(servaddr));
    if (err) {
        perror("In connect()");
        exit(1);
    }

    /* write a message to the server */
    err = write(sock, "Hello server.\n", strlen("Hello server.\n"));
    if (err < 0) {
        perror("In write()");
        exit(1);
    }

    printf("Message sent:\n\t@length: %d bytes\n\t@content: %s\n", err, "Hello server.\n");

    /* receive the response from the server */
    do {
        err = read(sock, &ch, 1);       /* read one byte from the socket */
        if (err <= 0) {
            printf("Premature end-of-file (0) or read() error (<0)? %d\n", err);
            break;
        }
        messagebuf[i++] = ch;
    } while (ch != '\n');
    messagebuf[i] = '\0';

    /* close the socket */
    close(sock);

    printf("Response received:\n\t@length: %lu bytes\n\t@content: %s\n", strlen(messagebuf), messagebuf);

    return 0;
}
```

You can download the original ascii source file with this link: unixcli.c .
Compile with: `gcc -o unixcli unixcli.c`
Before running, establish a UNIX domain socket, e.g. with socat:
`$ socat UNIX-LISTEN:/tmp/mytmpunixsock -`
In another terminal, run the client with:
`$ ./unixcli`
You see the message incoming from the client to the server in the first terminal. The client expects a response now: type something (newline-terminated) and see what the client tells. Also try interrupting the connection prematurely by Ctrl-C 'ing the server. (Manually remove the socket created by socat after use).

IPC: POSIX Message queues

- *mq_open(3)*
- *mq_close(3)*
- *mq_unlink(3)*
- *mq_send(3)*
- *mq_receive(3)*
- *mq_getattr(3)*
- *mq_setattr(3)*

Notes:
Message queues are an handy way to get IPC when working with unrelated processes that run completely without synchronism. A message queue is a shared "box" where processes can drop and withdraw messages independently. Besides the content, each message is given a "priority". Message queues first appeared in System V. POSIX standardized this IPC mechanism with the 1003.1b ("realtime") standard in 1993, but the resulting standard is describing a slightly different behavior wrt the former historical implementation. This is about the POSIX interface.

Message queues objects are referenced by a unique POSIX object name in the system as usual. The opening function (`mq_open()`) is the only one to use that name, while an internal reference returned by it will be used to reference the queue for the remainder of the program.
Messages are sent to the queue with the `mq_send()` (blocking by default). They can be any kind of information, nothing is dictated on the structure of a message. On sending, the message is to be accompanied by a priority. This priority is used for delivering messages: every time a process request for a receive (invoking the `mq_receive()` function, also blocking by default), the oldest message with the highest priority will be given. This differs from the SysV API, in which the user explicitly request for a specific priority, causing this priority being frequently used as recipient address.
Several features of the message queue can be set on creation or, in part, on a later time with the `mq_setattr()` function. Between them, the maximum size of a message is worth particular attention: each process invoking a receive won't be serviced, and EMSGSIZE will be set in `errno` if the length of the buffer for the message is shorter than this maximum size.

Summary:

- message queues are available with the System V API and the POSIX API. This is about the latter
- what to #include: `mqueue.h` (`sys/stat.h` for using permission macros while creating queues)
- types:
  1. **mqd_t**: a message queue descriptor.
  2. **struct mq_attr**: a message queue attributes structure, defined as follows:

```
struct mq_attr {
    long int mq_flags;    /* Message queue flags.  */
    long int mq_maxmsg;   /* Maximum number of messages.  */
    long int mq_msgsize;  /* Maximum message size.  */
    long int mq_curmsgs;  /* Number of messages currently queued.  */
}
```

- functions:
  1. `mqd_t` **`mq_open`**`(const char *name, int flags, ... [ mode_t mode, struct mq_attr *mq_attr ])`
     opens the queue referenced by `name` for access, where `flags` can request: `O_RDONLY`, `O_WRONLY`, `O_RDWR` for access permission, and `O_CREAT`, `O_EXCL` or `O_NONBLOCK` as intuitive, respectively, for create queue, fail create if already existing, and non-blocking behavior. If O_CREAT is specified, two more fields are requested: `mode` expliciting object permissions (S_IRWXU, S_IRWXG etc you can inspect in chmod(2); include `sys/stat.h` for these), and `mq_attr` expliciting the characteristics wanted for the queue. The latter can be set to `NULL` for using defaults, but mind the message size problem discussed in Notes above.
     Returns the message descriptor, or (mqd_t)-1 on error.
  2. `int` **`mq_close`**`(mdq_t mqdes)`
     closes the queue described by `mqdes`. Returns 0 for ok, otherwise -1.
  3. `int` **`mq_unlink`**`(const char *name)`
     like `unlink`(2), but with the posix object referenced by `name`.
     Returns 0, or -1 on error.
  4. `int` **`mq_send`**`(mqd_t mqdes, const char *msgbuf, size_t len, unsigned int prio)`
     sends `len` bytes from `msgbuf` to the queue `mqdes`, associating a `prio` priority.
     Return 0 on succes, -1 otherwise.
  5. `ssize_t` **`mq_receive`**`(mqd_t mqdes, char *buf, size_t len, unsigned *prio)`
     takes the oldest message with the highest priority from `mqdes` into `buf`. The `len` field determines an important effect described in Notes above. `prio`, if not NULL, is filled with the priority of the given message.
     Returns the length of the message received, or -1 on error.
  6. `int` **`mq_getattr`**`(mqd_t mqdes, struct mq_attr *mq_attr)`
     provides the attributes associated to `mqdes` by filling at `mq_attr` with the structure shown above.
     Returns 0, or -1 on error.
  7. `int` **`mq_setattr`**`(mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr *omqstat)`
     for `mqdes`, sets the attributes carried by `mqstat`. In fact, only the `mq_flags` member of this structure applies (for setting O_NONBLOCK): the other ones are simply ignored and can be set just when the queue is created. The `omqstat` is commonly NULL, or it will be filled with the previous attributes and the current queue status.
     Returns 0, or -1 on error and the queue won't be touched.
- further notes on persistence: message queues both in the POSIX and System V implementation features kernel-level persistence
- further notes on System V message queues: while the semantic is quite the same as for POSIX, System V message queues use a different namespace for functions for opening, creating, sending, receiving etc. See the msgget, msgctl, msgsnd, msgrcv man pages for more about this older API.

Examples:
1) Handling creation/opening of message queues, sending/receiving of messages and the essence of queue attributes. Try running several times the message producer before running the consumer. Try running the consumer more times than the producer. Try looping the producer until the queue fills. Try running the consumer after decreasing MAX_MSG_LEN under what's displayed by the queue attributes.

```
/*
 *  dropone.c
 *
 *  drops a message into a #defined queue, creating it if user
 *  requested. The message is associated a priority still user
 *  defined
 *
 *
 *  Created by Mij <mij@bitchx.it> on 07/08/05.
```

```
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 *
 */

#include <stdio.h>
/* mq_* functions */
#include <mqueue.h>
#include <sys/stat.h>
/* exit() */
#include <stdlib.h>
/* getopt() */
#include <unistd.h>
/* ctime() and time() */
#include <time.h>
/* strlen() */
#include <string.h>


/* name of the POSIX object referencing the queue */
#define MSGQOBJ_NAME    "/myqueue123"
/* max length of a message (just for this process) */
#define MAX_MSG_LEN     70


int main(int argc, char *argv[]) {
    mqd_t msgq_id;
    unsigned int msgprio = 0;
    pid_t my_pid = getpid();
    char msgcontent[MAX_MSG_LEN];
    int create_queue = 0;
    char ch;            /* for getopt() */
    time_t currtime;


    /* accepting "-q" for "create queue", requesting "-p prio" for message priority */
    while ((ch = getopt(argc, argv, "qi:")) != -1) {
        switch (ch) {
            case 'q':   /* create the queue */
                create_queue = 1;
                break;
            case 'p':   /* specify client id */
                msgprio = (unsigned int)strtol(optarg, (char **)NULL, 10);
                printf("I (%d) will use priority %d\n", my_pid, msgprio);
                break;
            default:
                printf("Usage: %s [-q] -p msg_prio\n", argv[0]);
                exit(1);
        }
    }

    /* forcing specification of "-i" argument */
    if (msgprio == 0) {
        printf("Usage: %s [-q] -p msg_prio\n", argv[0]);
        exit(1);
    }

    /* opening the queue        --  mq_open() */
    if (create_queue) {
        /* mq_open() for creating a new queue (using default attributes) */
        msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRWXU | S_IRWXG, NULL);
    } else {
        /* mq_open() for opening an existing queue */
        msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR);
    }
    if (msgq_id == (mqd_t)-1) {
        perror("In mq_open()");
        exit(1);
    }

    /* producing the message */
    currtime = time(NULL);
    snprintf(msgcontent, MAX_MSG_LEN, "Hello from process %u (at %s).", my_pid, ctime(&currtime));

    /* sending the message      --  mq_send() */
    mq_send(msgq_id, msgcontent, strlen(msgcontent)+1, msgprio);

    /* closing the queue        -- mq_close() */
    mq_close(msgq_id);


    return 0;
}
```

```
/*
 *  takeone.c
 *
 *  simply request a message from a queue, and displays queue
 *  attributes.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 07/08/05.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 *
 */

#include <stdio.h>
/* mq_* functions */
#include <mqueue.h>
/* exit() */
#include <stdlib.h>
/* getopt() */
#include <unistd.h>
/* ctime() and time() */
#include <time.h>
/* strlen() */
#include <string.h>
```

```c
/* name of the POSIX object referencing the queue */
#define MSGQOBJ_NAME    "/myqueue123"
/* max length of a message (just for this process) */
#define MAX_MSG_LEN     10000


int main(int argc, char *argv[]) {
    mqd_t msgq_id;
    char msgcontent[MAX_MSG_LEN];
    int msgsz;
    unsigned int sender;
    struct mq_attr msgq_attr;


    /* opening the queue        --  mq_open() */
    msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR);
    if (msgq_id == (mqd_t)-1) {
        perror("In mq_open()");
        exit(1);
    }

    /* getting the attributes from the queue        --  mq_getattr() */
    mq_getattr(msgq_id, &msgq_attr);
    printf("Queue \"%s\":\n\t- stores at most %ld messages\n\t- large at most %ld bytes each\n\t- currently holds %ld messages\n", MSGQOBJ_NAME, msgq_a

    /* getting a message */
    msgsz = mq_receive(msgq_id, msgcontent, MAX_MSG_LEN, &sender);
    if (msgsz == -1) {
        perror("In mq_receive()");
        exit(1);
    }
    printf("Received message (%d bytes) from %d: %s\n", msgsz, sender, msgcontent);

    /* closing the queue    --  mq_close() */
    mq_close(msgq_id);


    return 0;
}
```

You can download the original ascii source files with these links: dropone.c and takeone.c.
Compile with:
`gcc --pedantic -Wall -o dropone dropone.c`
`gcc --pedantic -Wall -o takeone takeone.c`
(some systems wrap these functions into a posix realtime library: add `-lrt` for them -- e.g. Linux does)
Run as follows:
`./dropone -p some_positive_integer` (add `-q` the first time)
`./takeone`
and try some of the tests described above the sources.

---

2) Working with queue attributes. This creates a queue with custom attributes, and also tries to modify them at a later time. Look for what attributes are ignored, when and why.

```c
/*
 *  msgqattrs.c
 *
 *  creates a posix message queue requesting custom attributes,
 *  and displays what attributes are taken into account and what
 *  are not, while both creating and while setting at a later time.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 31/08/05.
 *  Original source file available on http://mij.oltrelinux.com/devel/unixprg/
 *
 */


#include <stdio.h>
#include <mqueue.h>
#include <sys/stat.h>
/* exit() */
#include <stdlib.h>


#define MSGQOBJ_NAME        "/fooasd1431"
#define MAX_QMSG_SIZE       30


int main(int argc, char *argv[]) {
    mqd_t myqueue;
    struct mq_attr wanted_attrs, actual_attrs;


    /* filling the attribute structure */
    wanted_attrs.mq_flags = 0;                  /* no exceptional behavior (just O_NONBLOCK currently available)  */
    wanted_attrs.mq_maxmsg = 100;               /* room for at most 100 messages in the queue */
    wanted_attrs.mq_msgsize = MAX_QMSG_SIZE;    /* maximum size of a message */
    wanted_attrs.mq_curmsgs = 123;              /* this (current number of messages) will be ignored */

    /* mq_open() for creating a new queue (using default attributes) */
    myqueue = mq_open(MSGQOBJ_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRWXU | S_IRWXG, &wanted_attrs);
    if (myqueue == (mqd_t)-1) {
        perror("In mq_open()");
        exit(1);
    }

    printf("Message queue created.\n");
```

```
    /* getting queue attributes after creation          --  mq_getattr()    */
    mq_getattr(myqueue, &actual_attrs);
    printf("Attributes right after queue creation:\n\t- non blocking flag: %d\n\t- maximum number of messages: %ld\n\t- maximum size of a message: %ld

    /* building the structure again for modifying the existent queue */
    wanted_attrs.mq_flags = O_NONBLOCK;
    wanted_attrs.mq_maxmsg = 350;              /* this will be ignored by mq_setattr() */
    wanted_attrs.mq_msgsize = MAX_QMSG_SIZE;   /* this will be ignored by mq_setattr() */
    wanted_attrs.mq_curmsgs = 123;             /* this will be ignored by mq_setattr() */

    /* trying to later set different attributes on the queue        --  mq_setattr()    */
    mq_setattr(myqueue, &wanted_attrs, NULL);

    /* getting queue attributes after creation */
    mq_getattr(myqueue, &actual_attrs);
    printf("Attributes after setattr():\n\t- non blocking flag: %d\n\t- maximum number of messages: %ld\n\t- maximum size of a message: %ld\n\t- curre

    /* removing the queue from the system       --  mq_unlink() */
    mq_unlink(MSGQOBJ_NAME);


    return 0;
}
```

You can download the original ascii source with this link: msgqattrs.c.
Compile with:
`gcc --pedantic -Wall -o msgqattrs msgqattrs.c`
Run with: `./msgqattrs`

IPC: POSIX Semaphores

- *sem_init(2)*
- *sem_open(2)*
- *sem_close(2)*
- *sem_post(2)*
- *sem_wait(2)*
- *sem_trywait(2)*

Notes:
In the real world today, POSIX semaphores don't still get full support from OSes. Some features are not yet implemented. The Web is nonetheless poor of documentation about it, so the goal here is to leave some concepts; you will map into this or the other implementation in the need. We'll hopefully enjoy better support during the next years.
Semaphores are a mechanism for controlling access to shared resources. In their original form, they are associated to a single resource, and contain a binary value: {`free`, `busy`}. If the resource is shared between more threads (or procs), each thread waits for the resource semaphore to switch `free`. At this time, it sets the semaphore to `busy` and enters the critical region, accessing the resource. Once the work is done with the resource, the process releases it back setting the semaphore to `free`. The whole thing works because the semaphore value is set directly by the kernel on request, so it's not subject to process interleaving.
Please mind that this mechanism doesn't provide mutex access to a resource. It just provides support for processes to carry out the job themselves, so it's completely up to the processes to take the semaphore up to date and access the resource in respect of it.
Semaphores can be easily extended to support several accesses to a resource. If a resource can be accessed concurrently by *n* threads, the semaphore simply needs to hold an integer value instead of a Boolean one. This integer would be initialized to *n*, and each time it's positive value a process decrements it by 1 and gets 1 slot in the resource. When the value reaches 0, no more slots are available and candidate threads have to wait for the resource to get available ("free" is no more appropriate in this case). In this case the semaphore is told to be *locked*.
This is the concept behind semaphores. Dijkstra has been inspired by actual train semaphores when he designed them in '60s: this justify their being very intuitive.
There are 2 relevant APIs for such systems today:

- System V semaphores
- POSIX semaphores

The former implementations being inherited by the System V, and the latter being standardized by POSIX 1003.1b (year 1993, the first one with real time extensions). SysV semaphores are older and more popular. POSIX ones are newer, cleaner, easier, and lighter against the machine.
Just like pipes, semaphores can be *named* or *unnamed*. Unnamed semaphores can only be shared by related processes. The POSIX API defines both these personalities. They just differ in creation and opening.

Summary:

- always keep an eye on platform-specific implementations. It is common to deal with partial implementations. Make your way with the help of *man*.
- what to #include: `semaphore.h`
- types:
    1. **`sem_t`**: a semaphore descriptor. Commonly passed by reference.

- functions for activating *unnamed* semaphores:
  1. `int` **`sem_init`**`(sem_t *sem, int shared, unsigned int value)`
     the area pointed by `sem` is initialized with a new semaphore object. `shared` tells if the semaphore is local (0) or shared between several processes (non-0). This isn't vastly supported.
     `value` is the number the semaphore is initialized with.
     Returns -1 if unsuccessful.
  2. `int` **`sem_destroy`**`(sem_t *sem)`
     deallocates memory associated with the semaphore pointed by `sem`.
     Returns 0 if successful, -1 otherwise.
- functions for activating *named* semaphores:
  3. `sem_t` **`sem_open`**`(const char *name, int flags, ...)`
     "`, ...`" is zero or one occurrence of "`mode_t mode, unsigned int value`" (see below).
     opens a named semaphore. `name` specify its location in the filesystem hierarchy. `flags` can be zero, or `O_CREAT` (possibly associated with `O_EXCL`). Non-zero means creating the semaphore if it doesn't already exist. O_CREAT requires two further parameters: `mode` specifying the permission set of the semaphore in the filesystem, `value` being the same as in `sem_destroy()`. `O_CREAT` causes an exclusive behaviour if it appears with `O_EXCL`: a failure is returned if a semaphore `name` already exists in this case. Further values have been designed, while rarely implemented: `O_TRUNC` for truncating if such semaphore already exists, and O_NONBLOCK for non-blocking mode.
     `sem_open` returns the address of the semaphore if successful, SEM_FAILED otherwise.
  4. `int` **`sem_close`**`(sem_t *sem)`
     closes the semaphore pointed by `sem`. Returns 0 on success, -1 on failure.
  5. `int` **`sem_unlink`**`(const char name *name)`
     removes the semaphore `name` from the filesystem. Returns 0 on success, -1 on failure.
- functions for working with open semaphores:
  6. `int` **`sem_wait`**`(sem_t *sem)`
     waits until the semaphore `sem` is non-locked, then locks it and returns to the caller. Even without getting the semaphore unlocked, `sem_wait` may be interrupted with the occurrence of a signal.
     returns 0 on success, -1 on failure. If -1, `sem` is not modified by the function.
  7. `int` **`sem_trywait`**`(sem_t *sem)`
     tries to lock `sem` the same way `sem_wait` does, but doesn't hang if the semaphore is locked.
     Returns 0 on success (got semaphore), EAGAIN if the semaphore was locked.
  8. `int` **`sem_post`**`(sem_t *sem)`
     unlocks `sem`. Of course, this has to be atomic and non-reentrant. After this operation, some thread between those waiting for the semaphore to get free. Otherwise, the semaphore value simply steps up by 1.
     Returns 0 on success, -1 on failure.
  9. `int` **`sem_getvalue`**`(sem_t *sem, int *restrict sval)`
     compiles `sval` with the current value of `sem`. `sem` may possibly change its value before `sem_getvalue` returns to the caller.
     Returns 0 on success, -1 on failure.

Examples:
1) Unnamed semaphores.
One process; the main thread creates 2 threads. They access their critical section with mutual exclusion. The first one synchronously, the second one asynchronously.

```
/*
 *  semaphore.c
 *
 *
 *  Created by Mij <mij@bitchx.it> on 12/03/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unixprg/
 *
 */


#define _POSIX_SOURCE
#include <stdio.h>
/* sleep() */
#include <errno.h>
#include <unistd.h>
/* abort() and random stuff */
#include <stdlib.h>
/* time() */
#include <time.h>
#include <signal.h>
#include <pthread.h>
#include <semaphore.h>


/* this skips program termination when receiving signals */
void signal_handler(int type);

/*
 * thread A is synchronous. When it needs to enter its
 * critical section, it can't do anything other than waiting.
 */
void *thread_a(void *);

/*
 * thread B is asynchronous. When it tries to enter its
 * critical section, it switches back to other tasks if
 * it hasn't this availability.
```

```
 */
void *thread_b(void *);


/* the semaphore */
sem_t mysem;

int main(int argc, char *argv[])
{
    pthread_t mytr_a, mytr_b;
    int ret;


    srand(time(NULL));
    signal(SIGHUP, signal_handler);
    signal(SIGUSR1, signal_handler);

    /*
     * creating the unnamed, local semaphore, and initialize it with
     * value 1 (max concurrency 1)
     */
    ret = sem_init(&mysem, 0, 1);
    if (ret != 0) {
        /* error. errno has been set */
        perror("Unable to initialize the semaphore");
        abort();
    }

    /* creating the first thread (A) */
    ret = pthread_create(&mytr_a, NULL, thread_a, NULL);
    if (ret != 0) {
        perror("Unable to create thread");
        abort();
    }

    /* creating the second thread (B) */
    ret = pthread_create(&mytr_b, NULL, thread_b, NULL);
    if (ret != 0) {
        perror("Unable to create thread");
        abort();
    }

    /* waiting for thread_a to finish */
    ret = pthread_join(mytr_a, (void *)NULL);
    if (ret != 0) {
        perror("Error in pthread_join");
        abort();
    }

    /* waiting for thread_b to finish */
    ret = pthread_join(mytr_b, NULL);
    if (ret != 0) {
        perror("Error in pthread_join");
        abort();
    }

    return 0;
}


void *thread_a(void *x)
{
    unsigned int i, num;
    int ret;


    printf(" -- thread A -- starting\n");
    num = ((unsigned int)rand() % 40);

    /* this does (do_normal_stuff, do_critical_stuff) n times */
    for (i = 0; i < num; i++) {
        /* do normal stuff */
        sleep(3 + (rand() % 5));

        /* need to enter critical section */
        printf(" -- thread A -- waiting to enter critical section\n");
        /* looping until the lock is acquired */
        do {
            ret = sem_wait(&mysem);
            if (ret != 0) {
                /* the lock wasn't acquired */
                if (errno != EINVAL) {
                    perror(" -- thread A -- Error in sem_wait. terminating -> ");
                    pthread_exit(NULL);
                } else {
                    /* sem_wait() has been interrupted by a signal: looping again */
                    printf(" -- thread A -- sem_wait interrupted. Trying again for the lock...\n");
                }
            }
        } while (ret != 0);
        printf(" -- thread A -- lock acquired. Enter critical section\n");

        /* CRITICAL SECTION */
        sleep(rand() % 2);

        /* done, now unlocking the semaphore */
        printf(" -- thread A -- leaving critical section\n");
        ret = sem_post(&mysem);
        if (ret != 0) {
            perror(" -- thread A -- Error in sem_post");
            pthread_exit(NULL);
        }
    }

    printf(" -- thread A -- closing up\n");

    pthread_exit(NULL);
}


void *thread_b(void *x)
```

```
{
    unsigned int i, num;
    int ret;


    printf(" -- thread B -- starting\n");
    num = ((unsigned int)rand() % 100);

    /* this does (do_normal_stuff, do_critical_stuff) n times */
    for (i = 0; i < num; i++) {
        /* do normal stuff */
        sleep(3 + (rand() % 5));

        /* wants to enter the critical section */
        ret = sem_trywait(&mysem);
        if (ret != 0) {
            /* either an error happened, or the semaphore is locked */
            if (errno != EAGAIN) {
                /* an event different from "the semaphore was locked" happened */
                perror(" -- thread B -- error in sem_trywait. terminating -> ");
                pthread_exit(NULL);
            }

            printf(" -- thread B -- cannot enter critical section: semaphore locked\n");
        } else {
            /* CRITICAL SECTION */
            printf(" -- thread B -- enter critical section\n");

            sleep(rand() % 10);

            /* done, now unlocking the semaphore */
            printf(" -- thread B -- leaving critical section\n");
            sem_post(&mysem);
        }
    }

    printf(" -- thread B -- closing up\n");

    /* joining main() */
    pthread_exit(NULL);
}


void signal_handler(int type)
{
    /* do nothing */
    printf("process got signal %d\n", type);
    return;
}
```

You can download the original ascii source with this link: semaphore.c.
Compile with: `gcc --pedantic -Wall -o semaphore semaphore.c`
Run with: `./semaphore`

IPC: POSIX shared memory

- *shm_open*
- *mmap*
- *shm_unlink*

Notes:
Shared memory is what the name says about it: a segment of memory shared between several processes. UNIX knows System V shared memory and POSIX shared memory. This covers the POSIX way, but the other one is just matter of different system call names for initializing and terminating the segment.
Shared memory in UNIX is based on the concept of *memory mapping*. The segment of memory shared is coupled with an actual file in the filesystem. The file content is (ideally) a mirror of the memory segment at any time. The mapping between the shared segment content and the mapped file is persistent. The mapped file can be moved (even replicated) arbitrarily; this turns often suitable for reimporting such *image* of the memory segment across different runs of the same process (or different ones too) [See Further notes on mapping, below].
In fact, keeping an image file of the memory segment is occasionally useless. *Anonymous mapping* is when a shared segment isn't mapped to any actual object in the filesystem. This behavior is requested either explicitly (BSD way) or via `/dev/zero`, depending on the implementation [See Further notes on anonymous mapping, below].
In POSIX shared memory, the memory segment is mapped onto a posix object; consequently, it suffers the same problem discussed in notes on POSIX objects below. The `shm_open()` function is used to create such object (or open an existing one). The core function `mmap()` is then used to actually attach a memory segment of user-defined size into the address space of the process and map its content to the one of the ipc object (in case of open existing object, the former content is also imported). Thereafter, if other processes shm_open and mmap the same memory object, the memory segment will be effectively *shared*.
Please mind that shared memory is natural when working with threads in place of tasks. Threading is frequently used as a quick quirk when shared memory is wanted. Similarly, programming with shared memory requires many of the programmer attentions discussed for threads: usually shared memory is used in conjunction with some form of synchronization, often semaphores.
A segment of memory mapped into a process' address space will never be detached unless the `munmap()` is called. A posix shared memory object will persist unless the `shm_unlink()` function is used (or manually removed from the filesystem, where possible).

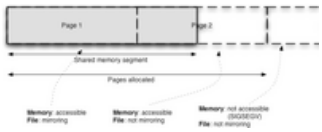

Figure 1

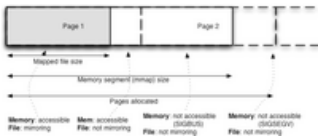

Figure 2

Further notes on mapping:
The mmap function maps objects into the process' address space. Mmap does two things: it causes the kernel to allocate and attach to the process address space a suitable number of page for covering the size requested by the user, and it establish the mapping between the object content and the memory content.
When mmap is called, the kernel allocates a suitable number of pages to cover the size requested by the user. Being memory allocation quantized, the real memory allocated into the process address space is rarely large what the user asked for. In contrast, files on disk can be stretched to whatever size byte by byte. Several effects raise from this fact. Figure 1 illustrates what happens when the memory segment size and mapped file size requested are equal, but not multiple of the page size. Figure 2 shows what happens when the size of the mapped file is shorter than the size requested for the memory segment.

Further notes on anonymous mapping:
Anonymous mapping is used share memory segment without mapping it to a persistent object (posix object, or file). It is the shared memory counterpart of unnamed semaphores, pipes etc. With this role, it is clear that anonymous mapping can only occur between related processes. From this mechanism, shared memory benefits in terms of performance. The developer is also advantaged not having to handle any persistent object.
There are 2 ways to get anonymous memory mapping. BSD provides a special flag (MAP_ANON) for `mmap()` to state this wish explicitly. Linux implements the same feature since version 2.4, but in general this is not a portable way to carry out the job.
System V (SVR4) doesn't provided such method, but the `/dev/zero` file can be used for the same purpose. Upon opening, the memory segment is initialized with the content read from the file (which is always 0s for this special file). Hereafter, whatever the kernel will write on the special file for mirroring from memory will be just discarded by the kernel.

Summary:

- what to #include: `sys/types.h` , `sys/mman.h` , possibly `fcntl.h` for O_* macros
- functions:
  1. `int` **`shm_open`**`(const char *name, int flags, mode_t mode)`
     open (or create) a shared memory object with the given POSIX name. The `flags` argument instructs on how to open the object: the most relevant ones are O_RDONLY xor O_RDWR for access type, and O_CREAT for create if object doesn't exist. `mode` is only used when the object is to be created, and specifies its access permission (umask style).
     Returns a file descriptor if successful, otherwise `-1`.
  2. `void *`**`mmap`**`(void *start_addr, size_t len, int protection, int flags, int fd, off_t offset)`
     maps the file `fd` into memory, starting from `offset` for a segment long `len`. `len` is preferably a multiple of the system page size; the actual size alloced is the smallest number of pages needed for containing the length requested. Notably, if the file is shorter than `offset` + `len`, the orphaned portion is still available for accessing, but what's written to it is not persistent.
     `protection` specifies the rules for accessing the segment: PROT_READ , PROT_WRITE, PROT_EXEC to be ORed appropriately. `flags` sets some details on how the segment will be mapped, the most relevant ones being private (MAP_PRIVATE) or shared (MAP_SHARED, default) and, on the systems supporting it, if it is an anonymous segment (MAP_ANON), in which case `fd` is good to be -1. `fd` is simply the file descriptor as returned by shm_open.
  3. `int` **`shm_unlink`**`(const char *name)`
     removes a shared memory object specified by `name`. If some process is still using the shared memory segment associated to `name`, the segment is not removed until all references to it have been closed. Commonly, `shm_unlink` is called right after a successful `shm_create`, in order to assure the memory object will be freed as soon as it's no longer used.
- *anonymous mapping* is useful when the persistence of the shared segment isn't requested, but like all unnamed mechanisms it only works between related processes. Two methods are available throughout the different OSes:
  1. BSD4.4 anonymous memory mapping: the mmap function is passed -1 as `fd` and an ored MAP_ANON in `flags`. The offset argument is ignored. The filesystem isn't touched at all.
  2. System V anonymous mapping: the `/dev/zero` file is opened to be passed to mmap. Then, the segment content is initially set to 0, and everything written to it is lost. This machanism also works with systems supporting the BSD way, of course.

Examples:
1) Two processes: a server creates a mapped memory segment and produces a message on it, then terminates. The client opens the persistent object mapping that memory and reads back the message left. The mapped object acts like a persistent memory image in this example.

```
/*
 *  shm_msgserver.c
 *
 *  Illustrates memory mapping and persistence, with POSIX objects.
 *  This process produces a message leaving it in a shared segment.
 *  The segment is mapped in a persistent object meant to be subsequently
 *  open by a shared memory "client".
 *
 *
```

```
 *  Created by Mij <mij@bitchx.it> on 27/08/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unixprg/
 *
 */


#include <stdio.h>
/* shm_* stuff, and mmap() */
#include <sys/mman.h>
#include <sys/types.h>
/* exit() etc */
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
/* for random() stuff */
#include <stdlib.h>
#include <time.h>


/* Posix IPC object name [system dependant] - see
http://mij.oltrelinux.com/devel/unixprg/index2.html#ipc__posix_objects */
#define SHMOBJ_PATH         "/foo1423"
/* maximum length of the content of the message */
#define MAX_MSG_LENGTH      50
/* how many types of messages we recognize (fantasy) */
#define TYPES               8

/* message structure for messages in the shared segment */
struct msg_s {
    int type;
    char content[MAX_MSG_LENGTH];
};


int main(int argc, char *argv[]) {
    int shmfd;
    int shared_seg_size = (1 * sizeof(struct msg_s));   /* want shared segment capable of storing 1 message */
    struct msg_s *shared_msg;      /* the shared segment, and head of the messages list */


    /* creating the shared memory object    --  shm_open()  */
    shmfd = shm_open(SHMOBJ_PATH, O_CREAT | O_EXCL | O_RDWR, S_IRWXU | S_IRWXG);
    if (shmfd < 0) {
        perror("In shm_open()");
        exit(1);
    }
    fprintf(stderr, "Created shared memory object %s\n", SHMOBJ_PATH);

    /* adjusting mapped file size (make room for the whole segment to map)      --  ftruncate() */
    ftruncate(shmfd, shared_seg_size);

    /* requesting the shared segment    --  mmap() */
    shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);
    if (shared_msg == NULL) {
        perror("In mmap()");
        exit(1);
    }
    fprintf(stderr, "Shared memory segment allocated correctly (%d bytes).\n", shared_seg_size);


    srandom(time(NULL));
    /* producing a message on the shared segment */
    shared_msg->type = random() % TYPES;
    snprintf(shared_msg->content, MAX_MSG_LENGTH, "My message, type %d, num %ld", shared_msg->type, random());


    /* [uncomment if you wish] requesting the removal of the shm object     --  shm_unlink() */
/*
    if (shm_unlink(SHMOBJ_PATH) != 0) {
        perror("In shm_unlink()");
        exit(1);
    }
*/

    return 0;
}
```

```
/*
 *  shm_msgclient.c
 *
 *  Illustrates memory mapping and persistence, with POSIX objects.
 *  This process reads and displays a message left it in "memory segment
 *  image", a file been mapped from a memory segment.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 27/08/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unixprg/
 *
 */

#include <stdio.h>
/* exit() etc */
#include <unistd.h>
/* shm_* stuff, and mmap() */
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
/* for random() stuff */
#include <stdlib.h>
#include <time.h>


/* Posix IPC object name [system dependant] - see
http://mij.oltrelinux.com/devel/unixprg/index2.html#ipc__posix_objects */
#define SHMOBJ_PATH         "/foo1423"
/* maximum length of the content of the message */
```

```c
#define MAX_MSG_LENGTH      50
/* how many types of messages we recognize (fantasy) */
#define TYPES               8

/* message structure for messages in the shared segment */
struct msg_s {
    int type;
    char content[MAX_MSG_LENGTH];
};


int main(int argc, char *argv[]) {
    int shmfd;
    int shared_seg_size = (1 * sizeof(struct msg_s));   /* want shared segment capable of storing 1 message */
    struct msg_s *shared_msg;      /* the shared segment, and head of the messages list */


    /* creating the shared memory object    --  shm_open()  */
    shmfd = shm_open(SHMOBJ_PATH, O_RDWR, S_IRWXU | S_IRWXG);
    if (shmfd < 0) {
        perror("In shm_open()");
        exit(1);
    }
    printf("Created shared memory object %s\n", SHMOBJ_PATH);

    /* requesting the shared segment    --  mmap() */
    shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);
    if (shared_msg == NULL) {
        perror("In mmap()");
        exit(1);
    }
    printf("Shared memory segment allocated correctly (%d bytes).\n", shared_seg_size);


    printf("Message type is %d, content is: %s\n", shared_msg->type, shared_msg->content);

    return 0;
}
```

You can download the original ascii source files with these links: shm_msgclient.c and shm_msgserver.c.
Compile with:
`gcc --ansi --Wall -o shm_msgserver shm_msgserver.c` and
`gcc --ansi --Wall -o shm_msgclient shm_msgclient.c`
(some systems wrap these functions into a posix realtime library: add `-lrt` for them -- e.g. Linux does)
Run the following:
`./shm_msgserver`
`./shm_msgclient`

---

2) Example on BSD asonymous memory mapping. The process gets a semaphore, establishes an anonymous memory mapping and forks (so the data space get independent). Both the parent and the child, in mutex, update the value of the shared segment by random quantities.

```c
/*
 *  shm_anon_bsd.c
 *
 *  Anonymous shared memory via BSD's MAP_ANON.
 *  Create a semaphore, create an anonymous memory segment with the MAP_ANON
 *  BSD flag and loop updating the segment content (increment casually) with
 *  short intervals.
 *
 *
 *  Created by Mij <mij@bitchx.it> on 29/08/05.
 *  Original source file available at http://mij.oltrelinux.com/devel/unixprg/
 *
 */

#include <stdio.h>
/* for shm_* and mmap() */
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
/* for getpid() */
#include <unistd.h>
/* exit() */
#include <stdlib.h>
/* for sem_* functions */
#include <sys/stat.h>
#include <semaphore.h>
/* for seeding time() */
#include <time.h>

/* name of the semaphore */
#define     SEMOBJ_NAME         "/semshm"
/* maximum number of seconds to sleep between each loop operation */
#define     MAX_SLEEP_SECS      3
/* maximum value to increment the counter by */
#define     MAX_INC_VALUE       10


int main(int argc, char *argv[]) {
    int shmfd;
    int shared_seg_size = 2 * sizeof(int);
    int *shared_values;     /* this will be a (shared) array of 2 elements */
    sem_t *sem_shmsegment;  /* semaphore controlling access to the shared segment */
    pid_t mypid;


    /* getting a new semaphore for the shared segment       -- sem_open()   */
    sem_shmsegment = sem_open(SEMOBJ_NAME, O_CREAT | O_EXCL, S_IRWXU | S_IRWXG, 1);
    if ((int)sem_shmsegment == SEM_FAILED) {
        perror("In sem_open()");
```

```
        exit(1);
    }
    /* requesting the semaphore not to be held when completely unreferenced */
    sem_unlink(SEMOBJ_NAME);

    /* requesting the shared segment    --  mmap() */
    shared_values = (int *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0);
    if ((int)shared_values == -1) {
        perror("In mmap()");
        exit(1);
    }
    fprintf(stderr, "Shared memory segment allocated correctly (%d bytes) at %u.\n", shared_seg_size, (unsigned int)shared_values);
    close(shmfd);


    /* dupping the process */
    if (! fork() )
        /* the child waits 2 seconds for better seeding srandom() */
        sleep(2);

    /* seeding the random number generator (% x for better seeding when child executes close) */
    srandom(time(NULL));

    /* getting my pid, and introducing myself */
    mypid = getpid();
    printf("My pid is %d\n", mypid);

    /*
       main loop:
        - pause
        - print the old value
        - choose (and store) a random quantity
        - increment the segment by that
    */
    do {
        sleep(random() % (MAX_SLEEP_SECS+1));       /* pausing for at most MAX_SLEEP_SECS seconds */

        sem_wait(sem_shmsegment);
        /* entered the critical region */

        printf("process %d. Former value %d.", mypid, shared_values[0]);

        shared_values[1] = random() % (MAX_INC_VALUE+1);            /* choose a random value up to MAX_INC_VALUE */
        shared_values[0] += shared_values[1];   /* and increment the first cell by this value */

        printf(" Incrementing by %d.\n", shared_values[1]);

        /* leaving the critical region */
        sem_post(sem_shmsegment);
    } while (1);

    /* freeing the reference to the semaphore */
    sem_close(sem_shmsegment);


    return 0;
}
```

You can download the original ascii source with this link: shm_anon_bsd.c.
Compile with:
`gcc --Wall -lrt -o shm_anon_bsd shm_anon_bsd.c`
Run the following: `./shm_anon_bsd`

IPC: notes on POSIX objects -- the POSIX objects mess

- *posix message queues*
- *posix named semaphores*
- *posix shared memory*

Notes: The POSIX API makes frequent use of "posix objects". Posix objects are for instance semaphores, shared memory segments or message queues. From the programmer point of view, they are simply referenced by an identificator, which is a string: the POSIX object name. POSIX objects come in play when persistence is required. In the semaphore example, the value of the semaphore needs to be preserved across system reboots. A program can create those objects and recall them during a later execution by their name.
The problem while using POSIX objects is actually their naming. Posix.1 (1003.1-1996) rules the following about posix object names:

- names must comply the OS rules for pathnames (eg length)
- names must begin with a "/" for different calls to reference the same object
- the behavior on further slashes "/" is implementation specific

For Posix, as you see, object names and (filesystem) pathnames are friend (naming rules and heading slash). But mapping them in the actual filesystem is not pointed out anywhere. The effect is that several different approaches to this mapping evolved until today, and the whole mechanism itself is non-portable. For example, FreeBSD 5 maps these objects directly into the file system (the semaphore named "/tmp/foo" is the actual file "foo" in /tmp), while Linux maps them usually under "/dev/shm" (message queue "foo" is file /dev/shm/foo).
Using only one slash, heading, complies to the rules but does not help for implementations which maps directly to the filesystem, in which the process wouldn't probably have write permission to the root directory where the object would be stored. The whole standard covering IPC objects naming is useless, there's no portable solution; the developer might use tools like autoconf to decide what name to choose at compile time, on the basis of the OS for which the code is compiled. A simple solution is to use a "template" name, and possibly prefixing it on architectures that map posix object names directly to the filesystem.
