

âMain Page | Modules | Data Structures | File List | Data Fields |
Related Pages

mntd

http://mntd.bambach.biz/daemonize_8c-source.html

daemonize.c

00001

```
/******
```

```
***
```

```
00002  * CVSID: $Id: daemonize.c,v 1.18 2004/08/07 15:10:47 stefanb Exp $
```

```
00003  *
```

```
00004  * daemonize.c : Daemonize handler
```

```
00005  *
```

```
00006  * Copyright (C) 2004 Stefan Bambach, <sbambach@gmx.net>
```

```
00007  *
```

```
00008  * Licensed under the GNU General Public License 2.0
```

```
00009  *
```

```
00010  * This program is free software; you can redistribute it and/or
```

```
modify
```

```
00011  * it under the terms of the GNU General Public License as
```

```
published by
```

```
00012  * the Free Software Foundation; either version 2 of the License,
```

```
or
```

```
00013  * (at your option) any later version.
```

```
00014  *
```

```
00015  * This program is distributed in the hope that it will be useful,
```

```
00016  * but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
00017  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
```

```
00018  * GNU General Public License for more details.
```

```
00019  *
```

```
00020  * You should have received a copy of the GNU General Public
```

```
License
```

```
00021  * along with this program; if not, write to the Free Software
```

```
00022  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-
```

```
1307 USA
```

```
00023  *
```

```
00024
```

```
*****
```

```
*/
```

```
00025
```

```
00026 #ifdef HAVE_CONFIG_H
```

```
00027 # include <config.h>
```

```
00028 #endif
```

```
00029
```

```
00030 #include <sys/types.h>
```

```
00031 #include <pwd.h>
```

```
00032 #include <grp.h>
```

```
00033 #include <sys/time.h>
```

```
00034 #include <sys/resource.h>
```

```
00035 #include <unistd.h>
```

```
00036 #include <sys/socket.h>
```

```
00037 #include <sys/stat.h>
```

```
00038 #include <fcntl.h>
```

```
00039 #include <errno.h>
```

```
00040 #include <syslog.h>
```

```
00041
```

```
00042 #include "file.h"
```

```
00043 #include "errmanager.h"
```

```
00044 #include "daemonize.h"
```

```
00045 #include "debug_mem.h"
```

```
00046
```

```
00047
```

```

00048
00049 extern int errno;
00050
00051
00052
00053 int
00054 sb_daemon_drop_privileges(void)
00055 {
00056     uid_t uid = getuid();
00057     gid_t gid = getgid();
00058     uid_t euid = geteuid();
00059     gid_t egid = getegid();
00060
00061     // set egid=gid
00062     if (egid != gid && (setgid(gid) == -1 || getegid() !=
getgid())) {
00063         return -1;
00064     }
00065
00066     // set euid=uid
00067     if (euid != uid && (setuid(uid) == -1 || geteuid() !=
getuid())) {
00068         return -1;
00069     }
00070
00071     // closes the /etc/passwd file
00072     endpwent();
00073     // closes the /etc/group file
00074     endgrent();
00075
00076     return 0;
00077 }
00078
00079
00080
00081 int
00082 sb_daemon_no_corefile(void)
00083 {
00084     struct rlimit limit[1] = {{ 0, 0 }};
00085
00086     if (getrlimit(RLIMIT_CORE, limit) == -1) {
00087         return -1;
00088     }
00089
00090     limit->rlim_cur = 0;
00091
00092     if (setrlimit(RLIMIT_CORE, limit) != 0) {
00093         return -1;
00094     }
00095
00096     return 0;
00097 }
00098
00099
00100
00101 int
00102 sb_daemon_is_started_by_init(void)
00103 {
00104     // get parents pid (init is the first process)
00105     if (getppid() == 1) {
00106         return 1;

```

```

00107     }
00108
00109     return 0;
00110 }
00111
00112
00113
00114 int
00115 sb_daemon_is_started_by_inetd(void)
00116 {
00117     size_t param_length = sizeof(int);
00118     int socket_type;
00119
00120     // check for type of stdin socket (inetd will set it
appropriate)
00121     if (getsockopt(STDIN_FILENO, SOL_SOCKET, SO_TYPE,
00122                     &socket_type, (void *)&param_length) == 0)
00123     {
00124         return 1;
00125     }
00126     return 0;
00127 }
00128
00129
00130
00131 int
00132 sb_daemon_is_started_by_root(void)
00133 {
00134     if (getuid() != 0) {
00135         return 0;
00136     }
00137     return 1;
00138 }
00139
00140
00141
00142
00143 int
00144 sb_daemon_is_started_suid(void)
00145 {
00146     // started suid, if effective and real IDs doesn't match
00147
00148     // check for sgid
00149     if (getgid() != getegid()) {
00150         return 1;
00151     }
00152
00153     // check for suid
00154     if (getuid() != geteuid()) {
00155         return 1;
00156     }
00157
00158     return 0;
00159 }
00160
00161
00162
00163 int
00164 sb_daemon_create_fd(int fd, int mode)
00165 {

```

```

00166     int i;
00167
00168     // create fd
00169     if ((i = open("/dev/null", mode)) == -1) {
00170         return -1;
00171     }
00172     // check, if it is correct file descriptor
00173     if (i != fd) {
00174         if (dup2(i, fd) == -1) {
00175             return -1;
00176         }
00177         close(i);
00178     }
00179
00180     return fd;
00181 }
00182
00183
00184
00185 int
00186 sb_daemon_change_user(uid_t uid, gid_t gid)
00187 {
00188     // daemon started by root ?
00189     if (sb_daemon_is_started_by_root() == 0) {
00190         return -1;
00191     }
00192
00193     // set gid, and check if gid and egid are set appropriate
00194     if ((setgid(gid) == -1) || (getgid() != gid) || (getegid() !=
gid)) {
00195         return -1;
00196     }
00197
00198     // set uid, and check if uid and euid are set appropriate
00199     if ((setuid(uid) == -1) || (getuid() != uid) || (geteuid() !=
uid)) {
00200         return -1;
00201     }
00202
00203     return 0;
00204 }
00205
00206
00207 //äâîîîèçàöèÿ
00208 int
00209 sb_daemon_detach(const char *rundir)
00210 {
00211     long i;
00212     pid_t pid;
00213
00214     // check for startup as root user (needed for change user, ...)
00215     if (sb_daemon_is_started_by_root() == 0) {
00216         return -1;
00217     }
00218
00219     //
00220     // see also http://www.unixguide.net/unix/programming/1.7.shtml
00221     //
00222     // Almost none of this is necessary (or advisable) if your
daemon is being

```

```

00223      // started by inetd. In that case, stdin, stdout and stderr are
00224      // set up for you to refer to the network connection, and the
00225      // fork()s and session manipulation should not be done (to
00226      // avoid confusing inetd). Only the chdir() and
00227      // umask() steps remain as useful.
00228      if ((!sb_daemon_is_started_by_init()) &&
00229          (!sb_daemon_is_started_by_inetd())) {
00230          // fork() so the parent can exit, this returns control to
00231          // the
00232          // command line or shell invoking your program. This step
00233          // is required
00234          // so that the new process is guaranteed not to be a
00235          // process group
00236          // leader. The next step, setsid(), fails if you're a
00237          // process group
00238          // leader.
00239          pid = fork();
00240          if (pid == -1) {
00241              return -1;
00242          }
00243          if (pid != 0) {
00244              exit(EXIT_SUCCESS);
00245          }
00246          // setsid() to become a process group and session group
00247          // leader. Since a controlling terminal is associated with
00248          // a session,
00249          // and this new session has not yet acquired a controlling
00250          // terminal
00251          // our process now has no controlling terminal, which is a
00252          // Good Thing
00253          // for daemons.
00254          setsid();
00255          // Ignore SIGHUP. When session leader terminates, it will
00256          // send a SIGHUP
00257          // signal to all child processes ("to us"). Your daemon
00258          // should handle
00259          // this signal after daemonizing has completed.
00260          signal(SIGHUP, SIG_IGN);
00261          // fork() again so the parent, (the session group leader),
00262          // can exit.
00263          // This means that we, as a non-session group leader, can
00264          // never regain
00265          // a controlling terminal.
00266          pid = fork();
00267          if (pid == -1) {
00268              return -1;
00269          }
00270          if (pid != 0) {
00271              exit(EXIT_SUCCESS);
00272          }
00273          // ensure that our process doesn't keep any directory
00274          // in use. Failure to do this could make it so that an
00275          // administrator
00276          // couldn't unmount a filesystem, because it was our current
00277          // directory.
00278          //

```

```

00272      // [Equivalently, we could change to any directory containing
files
00273      // important to the daemon's operation.]
00274      if (rundir != NULL) {
00275          struct stat st;
00276          if ((stat(rundir, &st) == 0) && (S_ISDIR(st.st_mode))) {
00277              chdir(rundir);
00278          } else {
00279              return -1;
00280          }
00281      } else {
00282          chdir("/");
00283      }
00284
00285      // umask(0) so that we have complete control over the
permissions of
00286      // anything we write. We don't know what umask we may have
inherited.
00287      umask(0);
00288
00289      // close() fds 0, 1, and 2. This releases the standard in, out,
and
00290      // error we inherited from our parent process. We have no way
of knowing
00291      // where these fds might have been redirected to. Note that
many daemons
00292      // use sysconf() to determine the limit _SC_OPEN_MAX.
00293      // _SC_OPEN_MAX tells you the maximum open files/process. Then
in a
00294      // loop, the daemon can close all possible file descriptors.
You have to
00295      // decide if you need to do this or not. If you think that
there might be
00296      // file-descriptors open you should close them, since there's a
limit on
00297      // number of concurrent file descriptors.
00298      if (sb_daemon_is_started_by_inetd() == 1) {
00299          // started by inetd: close all fds, but stdin, stdout and
stderr
00300          for (i = 0; i < sysconf(_SC_OPEN_MAX); i++) {
00301              if ((i!=STDIN_FILENO) && (i!=STDOUT_FILENO) &&
(i!=STDERR_FILENO)) {
00302                  close(i);
00303              }
00304          }
00305      } else {
00306          // started normal: close all fds
00307          for (i = 0; i < sysconf(_SC_OPEN_MAX); i++) {
00308              close(i);
00309          }
00310      }
00311
00312      // Establish new open descriptors for stdin, stdout and stderr.
Even if you
00313      // don't plan to use them, it is still a good idea to have them
open. The
00314      // precise handling of these is a matter of taste; if you have
a logfile,
00315      // for example, you might wish to open it as stdout or stderr,
and open
00316      // '/dev/null' as stdin; alternatively, you could open

```

```

00317     // '/dev/console' as stderr and/or stdout, and '/dev/null' as
00318     // stdin, or any other combination that makes sense for your
particular
00319     // daemon.
00320     if (sb_daemon_is_started_by_inetd() == 0) {
00321         // NOT started by inetd, so open stdin, stdout and stderr
again
00322
00323         // stdin
00324         if (sb_daemon_create_fd(STDIN_FILENO, O_RDONLY) == -1) {
00325             return -1;
00326         }
00327
00328         // stdout
00329         if (sb_daemon_create_fd(STDOUT_FILENO, O_WRONLY) == -1) {
00330             return -1;
00331         }
00332
00333         // stderr
00334         if (sb_daemon_create_fd(STDERR_FILENO, O_WRONLY) == -1) {
00335             return -1;
00336         }
00337     }
00338
00339     return 0;
00340 }
00341
00342
00343 //èíèòèàèèèçàöèý ääîíà
00344 int
00345 sb_daemon_init(const char *name,
00346                const char *rundir,
00347                const char *pidfile)
00348 {
00349     int fd = -1;
00350
00351     if (sb_daemon_is_running(pidfile) == 1) {
00352         return -1;
00353     }
00354
00355     // close logging before detaching (syslog wrapper)
00356     emClose();
00357
00358     // daemonize process
00359     if (sb_daemon_detach(rundir) != 0) {
00360         // initialize error manager again
00361         emInit(LOG_DEBUG, EM_TYPE_SYSLOG, NULL, NULL, NULL, name);
00362         return -1;
00363     }
00364
00365     // initialize error manager again
00366     emInit(LOG_DEBUG, EM_TYPE_SYSLOG, NULL, NULL, NULL, name);
00367
00368     // generate pidfile
00369     if (pidfile != NULL) {
00370         if ((fd = sb_daemon_pidfile_create_and_lock(pidfile)) == -
1) {
00371             return -1;
00372         }
00373         if (sb_daemon_pidfile_write(fd) == -1) {
00374

```

[illegible]


```

00435                                     S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH))
00436 == -1)
00437 {
00438     // return error, if not EEXIST
00439     if (errno != EEXIST) {
00440         return -1;
00441     }
00442     // try to open file. This will fail, if another daemon is
00443     running
00444     // and holds the lock.
00445     if ((fd = sb_file_open(pidfile, O_RDWR | O_SYNC)) == -1) {
00446         return -1;
00447     }
00448     // try to get the lock for pidfile
00449     if (sb_file_lock_acquire_write(fd) == -1) {
00450         return -1;
00451     }
00452     return fd;
00453 }
00454
00455 int
00456 sb_daemon_pidfile_write(int fd)
00457 {
00458     ssize_t written = 0;
00459     char data[MAX_PIDDATA_LENGTH];
00460
00461     // write pid to pidfile
00462     memset(data, 0, sizeof(data));
00463     snprintf(data, MAX_PIDDATA_LENGTH, "%d\n", (int)getpid());
00464     data[MAX_PIDDATA_LENGTH-1] = '\0';
00465
00466     // seek to begin
00467     lseek(fd, 0, SEEK_SET);
00468
00469     // write data
00470     if ((written = write(fd, data, strlen(data))) == -1) {
00471         return -1;
00472     }
00473     if ((size_t)written < strlen(data)) {
00474         return -1;
00475     }
00476
00477     // sync out data
00478     fsync(fd);
00479     sync();
00480
00481     return 0;
00482 }
00483
00484 pid_t
00485 sb_daemon_pidfile_read(const char *pidfile)
00486 {
00487     int fd = -1;

```

```
00494     char data[MAX_PIDDATA_LENGTH];
00495     int pid = -1;
00496
00497     if ((fd = sb_file_open(pidfile, O_RDONLY)) == -1) {
00498         return -1;
00499     }
00500
00501     memset(data, 0, sizeof(data));
00502     if (read(fd, &data, MAX_PIDDATA_LENGTH) == -1) {
00503         return -1;
00504     }
00505
00506     if (sscanf(data, "%d", &pid) != 1) {
00507         return -1;
00508     }
00509
00510     return pid;
00511 }
```

Generated on Wed Mar 30 13:43:26 2005 for Mntd by 1.3.9.1