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 "dbug_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         return 1;
00124     }
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
00138     return 1;
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 //äåìîíèçàöèÿ
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
all
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()) &&
(!sb_daemon_is_started_by_inetd())) {
00229
00230         // fork() so the parent can exit, this returns control to
the
00231         // command line or shell invoking your program. This step
is required
00232         // so that the new process is guaranteed not to be a
process group
00233         // leader. The next step, setsid(), fails if you're a
process group
00234         // leader.
00235         pid = fork();
00236         if (pid == -1) {
00237             return -1;
00238         }
00239         if (pid != 0) {
00240             exit(EXIT_SUCCESS);
00241         }
00242
00243         // setsid() to become a process group and session group
```

```
00244         // leader. Since a controlling terminal is associated with
a session,
00245         // and this new session has not yet acquired a controlling
terminal
00246         // our process now has no controlling terminal, which is a
Good Thing
00247         // for daemons.
00248         setsid();
00249
00250         // Ignore SIGHUP. When session leader terminates, it will
send a SIGHUP
00251         // signal to all child processes ("to us"). Your daemon
should handle
00252         // this signal after daemonizing has completed.
00253         signal(SIGHUP, SIG_IGN);
00254
00255         // fork() again so the parent, (the session group leader),
can exit.
00256         // This means that we, as a non-session group leader, can
never regain
00257         // a controlling terminal.
00258         pid = fork();
00259         if (pid == -1) {
00260             return -1;
00261         }
00262         if (pid != 0) {
00263             exit(EXIT_SUCCESS);
00264         }
00265  ensure that our process doesn't keep any directory
00269     // in use. Failure to do this could make it so that an
administrator
00270     // couldn't unmount a filesystem, because it was our current
directory.
00271     //
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 maximun 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 //èíèöèàëèçàöèÿ äåìîíà
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             return -1;
00375         }
00376     }
00377 
00378     return 0;
00379 }
00380 
```

```
00381
00382
00383 int
00384 sb_daemon_is_running(const char *pidfile)
00385 {
00386     int fd = -1;
00387
00388     // check for parameter and set errno appropriately
00389     if (pidfile == NULL) {
00390         return ERRNO(EINVAL);
00391     }
00392
00393     // try to get pidfile
00394     if ((fd = sb_daemon_pidfile_create_and_lock(pidfile)) == -1) {
00395         if ((errno == EACCES) || (errno == EAGAIN)) {
00396             // locked -> another daemon is running
00397             return 1;
00398         }
00399         // error
00400         return -1;
00401     }
00402
00403     // not locked -> remove old pidfile
00404     sb_file_close(fd);
00405     sb_file_remove(pidfile);
00406
00407     return 0;
00408 }
00409
00410
00411
00412 int
00413 sb_daemon_destroy(const char *pidfile)
00414 {
00415     // remove pidfile
00416     sb_file_remove(pidfile);
00417
00418     return 0;
00419 }
00420
00421
00422
00423 int
00424 sb_daemon_pidfile_create_and_lock(const char *pidfile)
00425 {
00426     int fd = -1;
00427
00428     // O_EXCL is broken on NFS file systems, programs which rely
00429     // on it for performing locking tasks will contain a race
condition.
00430     // see also 'man 2 open'
00431
00432     // try to open pidfile (will fail, if already existing)
00433     if ((fd = sb_file_open_ext(pidfile,
```

```
00434                               O_RDWR | O_SYNC | O_CREAT | O_EXCL,
00435                               S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH))
== -1)
00436     {
00437         // return error, if not EEXIST
00438         if (errno != EEXIST) {
00439             return -1;
00440         }
00441
00442         // try to open file. This will fail, if another daemon is
running
00443         // and holds the lock.
00444         if ((fd = sb_file_open(pidfile, O_RDWR | O_SYNC)) == -1) {
00445             return -1;
00446         }
00447     }
00448
00449     // try to get the lock for pidfile
00450     if (sb_file_lock_acquire_write(fd) == -1) {
00451         return -1;
00452     }
00453
00454     return fd;
00455 }
00456
00457
00458
00459 int
00460 sb_daemon_pidfile_write(int fd)
00461 {
00462     ssize_t written = 0;
00463     char data[MAX_PIDDATA_LENGTH];
00464
00465     // write pid to pidfile
00466     memset(data, 0, sizeof(data));
00467     snprintf(data, MAX_PIDDATA_LENGTH, "%d\n", (int)getpid());
00468     data[MAX_PIDDATA_LENGTH-1] = '\0';
00469
00470     // seek to begin
00471     lseek(fd, 0, SEEK_SET);
00472
00473     // write data
00474     if ((written = write(fd, data, strlen(data))) == -1) {
00475         return -1;
00476     }
00477     if ((size_t)written < strlen(data)) {
00478         return -1;
00479     }
00480
00481     // sync out data
00482     fsync(fd);
00483     sync();
00484
00485     return 0;
```

```
00486 }
00487
00488
00489
00490 pid_t
00491 sb_daemon_pidfile_read(const char *pidfile)
00492 {
00493     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 }
```