```c
/* This process is the father (mother) of all Minix user processes.  When
 * Minix comes up, this is process number 2, and has a pid of 1.  It
 * executes the /etc/rc shell file, and then reads the /etc/ttytab file to
 * determine which terminals need a login process.
 *
 * If the files /usr/adm/wtmp and /etc/utmp exist and are writable, init
 * (with help from login) will maintain login accounting.  Sending a
 * signal 1 (SIGHUP) to init will cause it to rescan /etc/ttytab and start
 * up new shell processes if necessary.  It will not, however, kill off
 * login processes for lines that have been turned off; do this manually.
 * Signal 15 (SIGTERM) makes init stop spawning new processes, this is
 * used by shutdown and friends when they are about to close the system
 * down.
 */

#include <minix/type.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/svrctl.h>
#include <ttyent.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <utmp.h>

/* Command to execute as a response to the three finger salute. */
char *REBOOT_CMD[] =    { "shutdown", "now", "CTRL−ALT−DEL", NULL };

/* Associated fake ttytab entry. */
struct ttyent TT_REBOOT = { "console", "−", REBOOT_CMD, NULL };

char PATH_UTMP[] = "/etc/utmp";           /* current logins */
char PATH_WTMP[] = "/usr/adm/wtmp";       /* login/logout history */

#define PIDSLOTS        32              /* first this many ttys can be on */

struct slotent {
  int errct;                    /* error count */
  pid_t pid;                    /* pid of login process for this tty line */
};

#define ERRCT_DISABLE   10      /* disable after this many errors */
#define NO_PID  0               /* pid value indicating no process */

struct slotent slots[PIDSLOTS]; /* init table of ttys and pids */

int gothup = 0;                 /* flag, showing signal 1 was received */
int gotabrt = 0;                /* flag, showing signal 6 was received */
int spawn = 1;                  /* flag, spawn processes only when set */

void tell(int fd, char *s);
void report(int fd, char *label);
void wtmp(int type, int linenr, char *line, pid_t pid);
void startup(int linenr, struct ttyent *ttyp);
int execute(char **cmd);
void onhup(int sig);
void onterm(int sig);
void onabrt(int sig);

int main(void)
{
  pid_t pid;                    /* pid of child process */
  int fd;                       /* generally useful */
  int linenr;                   /* loop variable */
  int check;                    /* check if a new process must be spawned */
  struct slotent *slotp;        /* slots[] pointer */
  struct ttyent *ttyp;          /* ttytab entry */
```



```c
  struct sigaction sa;
  struct stat stb;

  if (fstat(0, &stb) < 0) {
        /* Open standard input, output & error. */
        (void) open("/dev/null", O_RDONLY);
        (void) open("/dev/log", O_WRONLY);
        dup(1);
  }

  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;

  /* Hangup: Reexamine /etc/ttytab for newly enabled terminal lines. */
  sa.sa_handler = onhup;
  sigaction(SIGHUP, &sa, NULL);

  /* Terminate: Stop spawning login processes, shutdown is near. */
  sa.sa_handler = onterm;
  sigaction(SIGTERM, &sa, NULL);

  /* Abort: Sent by the kernel on CTRL−ALT−DEL; shut the system down. */
  sa.sa_handler = onabrt;
  sigaction(SIGABRT, &sa, NULL);

  /* Execute the /etc/rc file. */
  if ((pid = fork()) != 0) {
        /* Parent just waits. */
        while (wait(NULL) != pid) {
                if (gotabrt) reboot(RBT_HALT);
        }
  } else {
#if ! SYS_GETKENV
        struct sysgetenv sysgetenv;
#endif
        char bootopts[16];
        static char *rc_command[] = { "sh", "/etc/rc", NULL, NULL, NULL };
        char **rcp = rc_command + 2;

        /* Get the boot options from the boot environment. */
        sysgetenv.key = "bootopts";
        sysgetenv.keylen = 8+1;
        sysgetenv.val = bootopts;
        sysgetenv.vallen = sizeof(bootopts);
        if (svrctl(MMGETPARAM, &sysgetenv) == 0) *rcp++ = bootopts;
        *rcp = "start";

        execute(rc_command);
        report(2, "sh /etc/rc");
        _exit(1);       /* impossible, we hope */
  }

  /* Clear /etc/utmp if it exists. */
  if ((fd = open(PATH_UTMP, O_WRONLY | O_TRUNC)) >= 0) close(fd);

  /* Log system reboot. */
  wtmp(BOOT_TIME, 0, NULL, 0);

  /* Main loop. If login processes have already been started up, wait for one
   * to terminate, or for a HUP signal to arrive. Start up new login processes
   * for all ttys which don't have them. Note that wait() also returns when
   * somebody's orphan dies, in which case ignore it.  If the TERM signal is
   * sent then stop spawning processes, shutdown time is near.
   */

  check = 1;
  while (1) {
        while ((pid = waitpid(−1, NULL, check ? WNOHANG : 0)) > 0) {
                /* Search to see which line terminated. */
                for (linenr = 0; linenr < PIDSLOTS; linenr++) {
                        slotp = &slots[linenr];
                        if (slotp->pid == pid) {
                                /* Record process exiting. */
```

```
147                                 wtmp(DEAD_PROCESS, linenr, NULL, pid);
148                                 slotp->pid = NO_PID;
149                                 check = 1;
150                         }
151                 }
152         }
153
154         /* If a signal 1 (SIGHUP) is received, simply reset error counts. */
155         if (gothup) {
156                 gothup = 0;
157                 for (linenr = 0; linenr < PIDSLOTS; linenr++) {
158                         slots[linenr].errct = 0;
159                 }
160                 check = 1;
161         }
162
163         /* Shut down on signal 6 (SIGABRT). */
164         if (gotabrt) {
165                 gotabrt = 0;
166                 startup(0, &TT_REBOOT);
167         }
168
169         if (spawn && check) {
170                 /* See which lines need a login process started up. */
171                 for (linenr = 0; linenr < PIDSLOTS; linenr++) {
172                         slotp = &slots[linenr];
173                         if ((ttyp = getttyent()) == NULL) break;
174
175                         if (ttyp->ty_getty != NULL
176                                 && ttyp->ty_getty[0] != NULL
177                                 && slotp->pid == NO_PID
178                                 && slotp->errct < ERRCT_DISABLE)
179                         {
180                                 startup(linenr, ttyp);
181                         }
182                 }
183                 endttyent();
184         }
185         check = 0;
186   }
187 }
188
189 void onhup(int sig)
190 {
191   gothup = 1;
192   spawn = 1;
193 }
194
195 void onterm(int sig)
196 {
197   spawn = 0;
198 }
199
200 void onabrt(int sig)
201 {
202   static int count;
203
204   if (++count == 2) reboot(RBT_HALT);
205   gotabrt = 1;
206 }
207
208 void startup(int linenr, struct ttyent *ttyp)
209 {
210   /* Fork off a process for the indicated line. */
211
212   struct slotent *slotp;                /* pointer to ttyslot */
213   pid_t pid;                            /* new pid */
214   int err[2];                           /* error reporting pipe */
215   char line[32];                        /* tty device name */
216   int status;
217
218   slotp = &slots[linenr];
219
```



```
220   /* Error channel for between fork and exec. */
221   if (pipe(err) < 0) err[0] = err[1] = -1;
222
223   if ((pid = fork()) == -1 ) {
224         report(2, "fork()");
225         sleep(10);
226         return;
227   }
228
229   if (pid == 0) {
230         /* Child */
231         close(err[0]);
232         fcntl(err[1], F_SETFD, fcntl(err[1], F_GETFD) | FD_CLOEXEC);
233
234         /* A new session. */
235         setsid();
236
237         /* Construct device name. */
238         strcpy(line, "/dev/");
239         strncat(line, ttyp->ty_name, sizeof(line) - 6);
240
241         /* Open the line for standard input and output. */
242         close(0);
243         close(1);
244         if (open(line, O_RDWR) < 0 || dup(0) < 0) {
245                 write(err[1], &errno, sizeof(errno));
246                 _exit(1);
247         }
248
249         if (ttyp->ty_init != NULL && ttyp->ty_init[0] != NULL) {
250                 /* Execute a command to initialize the terminal line. */
251
252                 if ((pid = fork()) == -1) {
253                         report(2, "fork()");
254                         errno= 0;
255                         write(err[1], &errno, sizeof(errno));
256                         _exit(1);
257                 }
258
259                 if (pid == 0) {
260                         alarm(10);
261                         execute(ttyp->ty_init);
262                         report(2, ttyp->ty_init[0]);
263                         _exit(1);
264                 }
265
266                 while (waitpid(pid, &status, 0) != pid) {}
267                 if (status != 0) {
268                         tell(2, "init: ");
269                         tell(2, ttyp->ty_name);
270                         tell(2, ": ");
271                         tell(2, ttyp->ty_init[0]);
272                         tell(2, ": bad exit status\n");
273                         errno = 0;
274                         write(err[1], &errno, sizeof(errno));
275                         _exit(1);
276                 }
277         }
278
279         /* Redirect standard error too. */
280         dup2(0, 2);
281
282         /* Execute the getty process. */
283         execute(ttyp->ty_getty);
284
285         /* Oops, disaster strikes. */
286         fcntl(2, F_SETFL, fcntl(2, F_GETFL) | O_NONBLOCK);
287         if (linenr != 0) report(2, ttyp->ty_getty[0]);
288         write(err[1], &errno, sizeof(errno));
289         _exit(1);
290   }
291
292   /* Parent */
```

```
293   if (ttyp != &TT_REBOOT) slotp->pid = pid;
294
295   close(err[1]);
296   if (read(err[0], &errno, sizeof(errno)) != 0) {
297         /* If an errno value goes down the error pipe: Problems. */
298
299         switch (errno) {
300         case ENOENT:
301         case ENODEV:
302         case ENXIO:
303                 /* Device nonexistent, no driver, or no minor device. */
304                 slotp->errct = ERRCT_DISABLE;
305                 close(err[0]);
306                 return;
307         case 0:
308                 /* Error already reported. */
309                 break;
310         default:
311                 /* Any other error on the line. */
312                 report(2, ttyp->ty_name);
313         }
314         close(err[0]);
315
316         if (++slotp->errct >= ERRCT_DISABLE) {
317                 tell(2, "init: ");
318                 tell(2, ttyp->ty_name);
319                 tell(2, ": excessive errors, shutting down\n");
320         } else {
321                 sleep(5);
322         }
323         return;
324   }
325   close(err[0]);
326
327   if (ttyp != &TT_REBOOT) wtmp(LOGIN_PROCESS, linenr, ttyp->ty_name, pid);
328   slotp->errct = 0;
329 }
330
331 int execute(char **cmd)
332 {
333   /* Execute a command with a path search along /sbin:/bin:/usr/sbin:/usr/bin.
334    */
335   static char *nullenv[] = { NULL };
336   char command[128];
337   char *path[] = { "/sbin", "/bin", "/usr/sbin", "/usr/bin" };
338   int i;
339
340   if (cmd[0][0] == '/') {
341         /* A full path. */
342         return execve(cmd[0], cmd, nullenv);
343   }
344
345   /* Path search. */
346   for (i = 0; i < 4; i++) {
347         if (strlen(path[i]) + 1 + strlen(cmd[0]) + 1 > sizeof(command)) {
348                 errno= ENAMETOOLONG;
349                 return -1;
350         }
351         strcpy(command, path[i]);
352         strcat(command, "/");
353         strcat(command, cmd[0]);
354         execve(command, cmd, nullenv);
355         if (errno != ENOENT) break;
356   }
357   return -1;
358 }
359
360 void wtmp(type, linenr, line, pid)
361 int type;                       /* type of entry */
362 int linenr;                     /* line number in ttytab */
363 char *line;                     /* tty name (only good on login) */
364 pid_t pid;                      /* pid of process */
365 {
```



```
366 /* Log an event into the UTMP and WTMP files. */
367
368   struct utmp utmp;             /* UTMP/WTMP User Accounting */
369   int fd;
370
371   /* Clear the utmp record. */
372   memset((void *) &utmp, 0, sizeof(utmp));
373
374   /* Fill in utmp. */
375   switch (type) {
376   case BOOT_TIME:
377         /* Make a special reboot record. */
378         strcpy(utmp.ut_name, "reboot");
379         strcpy(utmp.ut_line, "~");
380         break;
381
382   case LOGIN_PROCESS:
383         /* A new login, fill in line name. */
384         strncpy(utmp.ut_line, line, sizeof(utmp.ut_line));
385         break;
386
387   case DEAD_PROCESS:
388         /* A logout.  Use the current utmp entry, but make sure it is a
389          * user process exiting, and not getty or login giving up.
390          */
391         if ((fd = open(PATH_UTMP, O_RDONLY)) < 0) {
392                 if (errno != ENOENT) report(2, PATH_UTMP);
393                 return;
394         }
395         if (lseek(fd, (off_t) (linenr+1) * sizeof(utmp), SEEK_SET) == -1
396                 || read(fd, &utmp, sizeof(utmp)) == -1
397         ) {
398                 report(2, PATH_UTMP);
399                 close(fd);
400                 return;
401         }
402         close(fd);
403         if (utmp.ut_type != USER_PROCESS) return;
404         strncpy(utmp.ut_name, "", sizeof(utmp.ut_name));
405         break;
406   }
407
408   /* Finish new utmp entry. */
409   utmp.ut_pid = pid;
410   utmp.ut_type = type;
411   utmp.ut_time = time((time_t *) 0);
412
413   switch (type) {
414   case LOGIN_PROCESS:
415   case DEAD_PROCESS:
416         /* Write new entry to utmp. */
417         if ((fd = open(PATH_UTMP, O_WRONLY)) < 0
418                 || lseek(fd, (off_t) (linenr+1) * sizeof(utmp), SEEK_SET) == -1
419                 || write(fd, &utmp, sizeof(utmp)) == -1
420         ) {
421                 if (errno != ENOENT) report(2, PATH_UTMP);
422         }
423         if (fd != -1) close(fd);
424         break;
425   }
426
427   switch (type) {
428   case BOOT_TIME:
429   case DEAD_PROCESS:
430         /* Add new wtmp entry. */
431         if ((fd = open(PATH_WTMP, O_WRONLY | O_APPEND)) < 0
432                   || write(fd, &utmp, sizeof(utmp)) == -1
433         ) {
434                 if (errno != ENOENT) report(2, PATH_WTMP);
435         }
436         if (fd != -1) close(fd);
437         break;
438   }
```

```
439 }
440
441 void tell(fd, s)
442 int fd;
443 char *s;
444 {
445         write(fd, s, strlen(s));
446 }
447
448 void report(fd, label)
449 int fd;
450 char *label;
451 {
452         int err = errno;
453
454         tell(fd, "init: ");
455         tell(fd, label);
456         tell(fd, ": ");
457         tell(fd, strerror(err));
458         tell(fd, "\n");
459         errno= err;
460 }
```