```
**********************************************************
    1308 Sat Dec 20 17:42:54 2014
new/usr/src/cmd/utmpd/Makefile
3244 utmpd.c: fix uninitialized variable, ret_val and other gcc warnings
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
# Copyright 2014 Shruti V Sampat <shrutisampat@gmail.com>
#
# Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

PROG= utmpd
DEFAULTFILES= utmpd.dfl
MANIFEST= utmp.xml
SVCMETHOD= svc-utmpd

include ../Makefile.cmd

ROOTMANIFESTDIR = $(ROOTSVCSYSTEM)

FILEMODE = 555

CERRWARN += -_gcc=-Wno-extra

.KEEP_STATE:

all: $(PROG)

install: all $(DIRS) $(ROOTLIBPROG) $(ROOTETCDEFAULTFILES) $(ROOTMANIFEST) \
        $(ROOTSVCMETHOD)

check:  $(CHKMANIFEST)

clean:

lint:   lint_PROG

include ../Makefile.targ
```

```
**********************************************************
   25323 Sat Dec 20 17:42:54 2014
new/usr/src/cmd/utmpd/utmpd.c
3244 utmpd.c: fix uninitialized variable, ret_val and other gcc warnings
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2014 Shruti V Sampat <shrutisampat@gmail.com>
 */

/*
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * Portions of such source code were derived from Berkeley 4.3 BSD
 * under license from the Regents of the University of California.
 */

/*
 * utmpd        - utmp daemon
 *
 *              This program receives requests from  pututxline(3)
 *              via a named pipe to watch the process to make sure it cleans up
 *              its utmpx entry on termination.
 *              The program keeps a list of procs
 *              and uses poll() on their /proc files to detect termination.
 *              Also the  program periodically scans the /etc/utmpx file for
 *              processes that aren't in the table so they can be watched.
 *
 *              If utmpd doesn't hear back over the pipe from pututline(3) that
 *              the process has removed its entry it cleans the entry when the
 *              the process terminates.
 *              The AT&T Copyright above is there since we borrowed the pipe
 *              mechanism from init(1m).
 */


#include        <sys/types.h>
#include        <signal.h>
#include        <stdio.h>
#include        <stdio_ext.h>
#include        <unistd.h>
```



```
#include        <utmpx.h>
#include        <errno.h>
#include        <termio.h>
#include        <sys/termios.h>
#include        <sys/tty.h>
#include        <ctype.h>
#include        <sys/stat.h>
#include        <sys/statvfs.h>
#include        <fcntl.h>
#include        <time.h>
#include        <sys/stropts.h>
#include        <wait.h>
#include        <syslog.h>
#include        <stdlib.h>
#include        <string.h>
#include        <poll.h>
#include        <deflt.h>
#include        <procfs.h>
#include        <sys/resource.h>

#define dprintf(x)      if (Debug) (void) printf x

/*
 * Memory allocation keyed off MAX_FDS
 */
#define MAX_FDS         4064    /* Maximum # file descriptors */
#define EXTRA_MARGIN    32      /* Allocate this many more FDS over Max_Fds */
/*
 * MAX_POLLNV & RESETS - paranoia to cover an error case that might not exist
 */
#define MAX_POLL_ERRS   1024    /* Count of bad errors */
#define MAX_RESETS      1024    /* Maximum times to reload tables */
#define POLL_TIMEOUT    300     /* Default Timeout for poll() in seconds */
#define CLEANIT         1       /* Used by rem_pid() */
#define DONT_CLEAN      0       /* Used by rem_pid() */
#define UTMP_DEFAULT    "/etc/default/utmpd"
#define WARN_TIME       3600    /* seconds between utmp checks */
#define WTMPX_UFREQ     60      /* seconds between updating WTMPX's atime */


/*
 * The pidrec structure describes the data shipped down the pipe to
 * us from the pututxline() library in
 * lib/libc/port/gen/getutx.c
 */

/*
 * pd_type's
 */
#define ADDPID  1
#define REMPID  2

struct  pidrec {
        int     pd_type;                /* Command type */
        pid_t   pd_pid;                 /* pid to add or remove */
};
______unchanged_portion_omitted_

static struct pidentry *pidtable = NULL;

static pollfd_t *fdtable = NULL;

static int      pidcnt = 0;             /* Number of procs being watched */
static char     *prog_name;             /* To save the invocation name away */
static char     *UTMPPIPE_DIR = "/var/run";
static char     *UTMPPIPE = "/var/run/utmppipe";
```

```
141 static int      Pfd = -1;              /* File descriptor of named pipe */
142 static int      Poll_timeout = POLL_TIMEOUT;
143 static int      WTMPXfd = -1;          /* File descriptor of WTMPX_FILE */
144 static int      WTMPX_ufreq = WTMPX_UFREQ;
145 static int      Debug = 0;             /* Set by command line argument */
146 static int      Max_fds         = MAX_FDS;

148 /*
149  * This program has three main components plus utilities and debug routines
150  *      Receiver - receives the process ID or process for us to watch.
151  *                 (Uses a named pipe to get messages)
152  *      Watcher  - Use poll(2) to watch for processes to die so they
153  *                 can be cleaned up (get marked as DEAD_PROCESS)
154  *      Scanner  - periodically scans the utmpx file for stale entries
155  *                 or live entries that we don't know about.
156  */

158 static int wait_for_pids();     /* Watcher - uses poll */
159 static void scan_utmps();       /* Scanner, reads utmpx file */
160 static void drain_pipe();       /* Receiver - reads mesgs over UTMPPIPE */
161 static void setup_pipe();       /* For setting up receiver */

163 static void add_pid();          /* Adds a process to the table */
164 static void rem_pid();          /* Removes a process from the table */
165 static int find_pid();          /* Finds a process in the table */
166 static int proc_to_fd();        /* Takes a pid and returns an fd for its proc */
167 static void load_tables();      /* Loads up the tables the first time around */
168 static int pidcmp();            /* For sorting pids */

170 static void clean_entry();      /* Removes entry from our table and calls ... */
171 static void clean_utmpx_ent();  /* Cleans a utmpx entry */

173 static void fatal() __NORETURN; /* Prints error message and calls exit */
174 static void nonfatal();         /* Prints error message */
175 static void print_tables();     /* Prints out internal tables for Debug */
176 static int proc_is_alive(pid_t pid);    /* Check if a process is alive */
177 static void warn_utmp(void);

179 /*
180  * main()  - Main does basic setup and calls wait_for_pids() to do the work
181  */

183 int
184 main(int argc, char *argv[])
180 main(argc, argv)
181         char **argv;
185 {
186         char *defp;
187         struct rlimit rlim;
188         int i;
189         time_t curtime, now;

191         prog_name = argv[0];                    /* Save invocation name */

193         if (getuid() != 0)  {
194                 (void) fprintf(stderr,
195                     "You must be root to run this program\n");
196                 fatal("You must be root to run this program");
197         }

199         if (argc > 1) {
200                 if ((argc == 2 && (int)strlen(argv[1]) >= 2) &&
201                     (argv[1][0] == '-' && argv[1][1] == 'd')) {
202                         Debug = 1;
203                 } else {
204                         (void) fprintf(stderr,
```



```
205                             "%s: Wrong number of arguments\n", prog_name);
206                         (void) fprintf(stderr,
207                             "Usage: %s [-debug]\n", prog_name);
208                         exit(2);
209                 }
210         }

212         /*
213          * Read defaults file for poll timeout
214          */
215         if (defopen(UTMP_DEFAULT) == 0) {
216                 if ((defp = defread("SCAN_PERIOD=")) != NULL) {
217                         Poll_timeout = atol(defp);
218                         dprintf(("Poll timeout set to %d\n", Poll_timeout));
219                 }

221                 if ((defp = defread("WTMPX_UPDATE_FREQ=")) != NULL) {
222                         WTMPX_ufreq = atol(defp);
223                         dprintf(("WTMPX update frequency set to %d\n",
224                             WTMPX_ufreq));
225                 }

227                 /*
228                  * Paranoia - if polling on large number of FDs is expensive /
229                  * buggy the number can be set lower in the field.
230                  */
231                 if ((defp = defread("MAX_FDS=")) != NULL) {
232                         Max_fds = atol(defp);
233                         dprintf(("Max_fds set to %d\n", Max_fds));
234                 }
235                 (void) defopen((char *)NULL);
236         }

238         if (Debug == 0) {
239                 /*
240                  * Daemonize ourselves
241                  */
242                 if (fork()) {
243                         exit(0);
244                 }
245                 (void) close(0);
246                 (void) close(1);
247                 (void) close(2);
248                 /*
249                  * We open these to avoid accidentally writing to a proc file
250                  */
251                 (void) open("/dev/null", O_RDONLY);
252                 (void) open("/dev/null", O_WRONLY);
253                 (void) open("/dev/null", O_WRONLY);
254                 (void) setsid();                /* release process from tty */
255         }

257         openlog(prog_name, LOG_PID, LOG_DAEMON);        /* For error messages */
258         warn_utmp();    /* check to see if utmp came back by accident */

260         /*
261          * Allocate the pidtable and fdtable.  An earlier version did
262          * this as we go, but this is simpler.
263          */
264         if ((pidtable = malloc(Max_fds * sizeof (struct pidentry))) == NULL)
265                 fatal("Malloc failed");
266         if ((fdtable = malloc(Max_fds * sizeof (pollfd_t))) == NULL)
267                 fatal("Malloc failed");

269         /*
270          * Up the limit on FDs
```

```
271          */
272         if (getrlimit(RLIMIT_NOFILE, &rlim) == 0) {
273                 rlim.rlim_cur = Max_fds + EXTRA_MARGIN + 1;
274                 rlim.rlim_max = Max_fds + EXTRA_MARGIN + 1;
275                 if (setrlimit(RLIMIT_NOFILE, &rlim) != 0) {
276                         fatal("Out of File Descriptors");
277                 }
278         } else
279                 fatal("getrlimit returned failure");

281         (void) enable_extended_FILE_stdio(-1, -1);

283         if ((WTMPXfd = open(WTMPX_FILE, O_RDONLY)) < 0)
284                 nonfatal("WARNING: unable to open " WTMPX_FILE "for update.");

286         /*
287          * Loop here scanning the utmpx file and waiting for processes
288          * to terminate.  Most of the activity is directed out of wait_for_pids.
289          * If wait_for_pids fails we reload the table and try again.
290          */

292         curtime = time(NULL);
293         dprintf(("utmp warning timer set to %d seconds\n", WARN_TIME));

295         for (i = 0; i < MAX_RESETS; i++) {
296                 load_tables();
297                 while (wait_for_pids() == 1) {
298                         now = time(NULL);
299                         if ((now - curtime) >= WARN_TIME) {
300                                 dprintf(("utmp warning timer expired\n"));
301                                 warn_utmp();
302                                 curtime = now;
303                         }
304                 }
305         }

307         (void) close(WTMPXfd);

309         /*
310          * We only get here if we had a bunch of resets - so give up
311          */
312         fatal("Too many resets, giving up");
313         return (1);
314 }
```

unchanged_portion_omitted_

```
342 /*
343  *                      *** The Watcher ***
344  *
345  * Wait_for_pids        - wait for the termination of a process in the table.
346  *                        Returns 1 on normal exist, 0 on failure.
347  */

349 static int
350 wait_for_pids()
351 {
352         register struct pollfd *pfd;
353         register int i;
354         pid_t pid;
355         int ret_val = 0;
352         int ret_val;
356         int timeout;
357         static time_t last_timeout  = 0;
358         static int bad_error  = 0;      /* Count of POLL errors */
```



```
360         /*
361          * First time through we initialize last_timeout to now.
362          */
363         if (last_timeout == 0)
364                 last_timeout = time(NULL);

366         /*
367          * Recalculate timeout - checking to see if time expired.
368          */

370         if ((timeout = Poll_timeout - (time(NULL) - last_timeout)) <= 0) {
371                 timeout = Poll_timeout;
372                 last_timeout = time(NULL);
373                 scan_utmps();
374         }

376         fdtable[0].events = POLLRDNORM;

378         for (i = 0; i < (timeout / WTMPX_ufreq); i++) {

380                 /*
381                  * Loop here while getting EAGAIN
382                  */

384                 while ((ret_val = poll(fdtable, pidcnt, WTMPX_ufreq*1000)) < 0)
385                         if (errno == EAGAIN)
386                                 (void) sleep(2);
387                         else
388                                 fatal("poll");
389                 /*
390                  * The results of pread(2) are discarded; we only want
391                  * to update the access time of WTMPX_FILE.
392                  * Periodically touching WTMPX helps determine when the
393                  * OS became unavailable when the OS boots again .
394                  * See PSARC 2004/462 for more information.
395                  */

397                 (void) pread(WTMPXfd, (void *)&pid, sizeof (pid), 0);

399                 if (ret_val)            /* file descriptor(s) need attention */
400                         break;
401         }

403         /*
404          * If ret_val == 0 the poll timed out - reset last_time and
405          * call scan_utmps
406          */
407         if (ret_val == 0) {
408                 last_timeout = time(NULL);
409                 scan_utmps();
410                 return (1);
411         }

413         /*
414          * Check the pipe file descriptor
415          */
416         if (fdtable[0].revents & POLLRDNORM) {
417                 drain_pipe();
418                 fdtable[0].revents = 0;
419                 ret_val--;
420         }

422         (void) sleep(5);        /* Give parents time to cleanup children */

424         /*
425          * We got here because the status of one of the pids that
```

```
426          * we are polling on has changed, so search the table looking
427          * for the entry.
428          *
429          * The table is scanned backwards so that entries can be removed
430          * while we go since the table is compacted from high down to low
431          */
432         for (i = pidcnt - 1; i > 0; i--) {
433                 /*
434                  * Break out of the loop if we've processed all the entries.
435                  */
436                 if (ret_val == 0)
437                         break;

439                 pfd = &fdtable[i];

441                 if (pfd->fd < 0) {
442                         rem_pid((pid_t)0, i, DONT_CLEAN);
443                         continue;
444                 }
445                 /*
446                  * POLLHUP      - Process terminated
447                  */
448                 if (pfd->revents & POLLHUP) {
449                         psinfo_t psinfo;

451                         if (pread(pfd->fd, &psinfo, sizeof (psinfo), (off_t)0)
452                             != sizeof (psinfo)) {
453                                 dprintf(("! %d: terminated, status 0x%.4x\n", \
454                                     (int)pidtable[i].pl_pid, psinfo.pr_wstat));
455                                 pidtable[i].pl_status = psinfo.pr_wstat;

457                         } else {
458                                 dprintf(("! %d: terminated\n", \
459                                     (int)pidtable[i].pl_pid));
460                                 pidtable[i].pl_status = 0;
461                         }
462                         /*
463                          * PID gets removed when terminated only
464                          */
465                         rem_pid((pid_t)0, i, CLEANIT);
466                         ret_val--;
467                         continue;
468                 }
469                 /*
470                  * POLLNVAL and POLLERR
471                  *      These error's shouldn't occurr but until their fixed
472                  *      we perform some simple error recovery.
473                  */
474                 if (pfd->revents & (POLLNVAL|POLLERR)) {
475                         dprintf(("Poll Err = %d pid = %d i = %d\n", \
476                             pfd->revents, (int)pidtable[i].pl_pid, i));

478                         pid = pidtable[i].pl_pid; /* Save pid for below */
479                         /*
480                          * If its POLLNVAL we just remove the process for
481                          * now, it will get picked up in the next scan.
482                          * POLLERR pids get re-added after being deleted.
483                          */
484                         if (pfd->revents & POLLNVAL) {
485                                 rem_pid((pid_t)0, i, DONT_CLEAN);
486                         } else {                        /* Else... POLLERR */
487                                 rem_pid((pid_t)0, i, DONT_CLEAN);
488                                 add_pid(pid);
489                         }

491                         if (bad_error++ > MAX_POLL_ERRS) {
```



```
492                                 bad_error = 0;
493                                 return (0);     /* 0 Indicates severe error */
494                         }
495                         ret_val--;
496                         continue;
497                 }

499                 /*
500                  * No more bits should be set in revents but check anyway
501                  */
502                 if (pfd->revents != 0) {
503                         dprintf(("%d: unknown err %d\n", \
504                             (int)pidtable[i].pl_pid, pfd->revents));

506                         rem_pid((pid_t)0, i, DONT_CLEAN);
507                         ret_val--;

509                         if (bad_error++ > MAX_POLL_ERRS) {
510                                 bad_error = 0;
511                                 return (0);     /* 0 Indicates severe error */
512                         }
513                         return (1);
514                 }
515         }
516         return (1);                     /* 1 Indicates Everything okay */
517 }
```

_____***unchanged_portion_omitted_***