```
**********************************************************
    1307 Mon Jan 12 23:37:07 2015
new/usr/src/cmd/utmpd/svc-utmpd
5375 utmpd(1M) core dumps when WTMPX_UPDATE_FREQ is zero
**********************************************************
#!/sbin/sh
#
# 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 2007 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# ident "%Z%%M% %I%     %E% SMI"

. /lib/svc/share/smf_include.sh

[ ! -x /usr/lib/utmpd ] && exit $SMF_EXIT_ERR_CONFIG

# If a utmppipe exists, check for a utmpd process and exit
# if the daemon is already running.

if [ -p /var/run/utmppipe ]; then
        if /usr/bin/pgrep -x -u 0 -z `smf_zonename` utmpd >/dev/null 2>&1; then
                echo "$0: utmpd is already running"
                exit 1
        fi
fi

/usr/bin/rm -f /var/run/utmppipe
/usr/lib/utmpd &
```

****************************************************************
***26383 Mon Jan 12 23:37:07 2015***
***new/usr/src/cmd/utmpd/utmpd.c***
***5375 utmpd(1M) core dumps when WTMPX_UPDATE_FREQ is zero***
****************************************************************

```
/*
 * 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, 2015 Shruti V Sampat <shrutisampat@gmail.com>
 * 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>
#include        <limits.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";
static int      Pfd = -1;               /* File descriptor of named pipe */
static int      Poll_timeout = POLL_TIMEOUT;
static int      WTMPXfd = -1;           /* File descriptor of WTMPX_FILE */
static int      WTMPX_ufreq = WTMPX_UFREQ;
static int      Debug = 0;              /* Set by command line argument */
static int      Max_fds         = MAX_FDS;

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

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

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

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

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

/* Validate defaults from file and assign */
static int validate_default(char *defp, int *flag);

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

int
main(int argc, char *argv[])
{
        char *defp;
        struct rlimit rlim;
        int i;
        time_t curtime, now;
        char msg[256];

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

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

        if (argc > 1) {
                if ((argc == 2 && (int)strlen(argv[1]) >= 2) &&
```

```
                    (argv[1][0] == '-' && argv[1][1] == 'd')) {
                        Debug = 1;
                } else {
                        (void) fprintf(stderr,
                            "%s: Wrong number of arguments\n", prog_name);
                        (void) fprintf(stderr,
                            "Usage: %s [-debug]\n", prog_name);
                        exit(2);
                }
        }

        /*
         * Read defaults file for poll timeout, WTMPX update frequency
         * and maximum number of processes to monitor.
         * Read defaults file for poll timeout
         */
        if (defopen(UTMP_DEFAULT) == 0) {
                if ((defp = defread("SCAN_PERIOD=")) != NULL)
                        if (validate_default(defp, &Poll_timeout) == -1) {
                                (void) snprintf(msg, sizeof (msg), "SCAN_PERIOD"
                                    " should be a positive integer, found %s",
                                    defp);
                                nonfatal(msg);
                        }
                if ((defp = defread("SCAN_PERIOD=")) != NULL) {
                        Poll_timeout = atol(defp);
                dprintf(("Poll timeout set to %d\n", Poll_timeout));
                }

                if ((defp = defread("WTMPX_UPDATE_FREQ=")) != NULL)
                        if (validate_default(defp, &WTMPX_ufreq) == -1) {
                                (void) snprintf(msg, sizeof (msg),
                                    "WTMPX_UPDATE_FREQ should be a positive "
                                    "integer, found %s", defp);
                                nonfatal(msg);
                if ((defp = defread("WTMPX_UPDATE_FREQ=")) != NULL) {
                        WTMPX_ufreq = atol(defp);
                        dprintf(("WTMPX update frequency set to %d\n",
                            WTMPX_ufreq));
                        }
                dprintf(("WTMPX update frequency set to %d\n", WTMPX_ufreq));

                /*
                 * Paranoia - if polling on large number of FDs is expensive /
                 * buggy the number can be set lower in the field.
                 */
                if ((defp = defread("MAX_FDS=")) != NULL)
                        if (validate_default(defp, &Max_fds) == -1) {
                                (void) snprintf(msg, sizeof (msg), "MAX_FDS "
                                    "should be a positive integer, found %s",
                                    defp);
                                nonfatal(msg);
                if ((defp = defread("MAX_FDS=")) != NULL) {
                        Max_fds = atol(defp);
                        dprintf(("Max_fds set to %d\n", Max_fds));
                        }
                dprintf(("Max fds set to %d\n", Max_fds));
                (void) defopen((char *)NULL);
        }

        if (Debug == 0) {
                /*
                 * Daemonize ourselves
                 */
                if (fork()) {
                        exit(0);
                }
                (void) close(0);
                (void) close(1);
                (void) close(2);
                /*
                 * We open these to avoid accidentally writing to a proc file
                 */
                (void) open("/dev/null", O_RDONLY);
                (void) open("/dev/null", O_WRONLY);
                (void) open("/dev/null", O_WRONLY);
                (void) setsid();                /* release process from tty */
        }

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

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

        /*
         * Up the limit on FDs
         */
        if (getrlimit(RLIMIT_NOFILE, &rlim) == 0) {
                rlim.rlim_cur = Max_fds + EXTRA_MARGIN + 1;
                rlim.rlim_max = Max_fds + EXTRA_MARGIN + 1;
                if (setrlimit(RLIMIT_NOFILE, &rlim) != 0) {
                        fatal("Out of File Descriptors");
                }
        } else
                fatal("getrlimit returned failure");

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

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

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

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

        for (i = 0; i < MAX_RESETS; i++) {
                load_tables();
                while (wait_for_pids() == 1) {
                        now = time(NULL);
                        if ((now - curtime) >= WARN_TIME) {
                                dprintf(("utmp warning timer expired\n"));
                                warn_utmp();
                                curtime = now;
                        }
                }
        }

        (void) close(WTMPXfd);
```

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

```
/*
 *              *** Utilities for add and removing entries in the tables ***
 */

/*
 * add_pid      - add a pid to the fd table and the pidtable.
 *                these tables are sorted tables for quick lookups.
 *
 */
static void
add_pid(pid_t pid)
add_pid(pid)
        pid_t pid;
{
        int fd = 0;
        int i = 0, move_amt;
        int j;
        static int first_time = 1;

        /*
         * Check to see if the pid is already in our table, or being passed
         * pid zero.
         */
        if (pidcnt != 0 && (find_pid(pid, &j) == 1 || pid == 0))
                return;

        if (pidcnt >= Max_fds) {
                if (first_time == 1) {
                        /*
                         * Print this error only once
                         */
                        nonfatal("File Descriptor limit exceeded");
                        first_time = 0;
                }
                return;
        }
        /*
         * Open the /proc file checking if there's still a valid proc file.
         */
        if (pid != 0 && (fd = proc_to_fd(pid)) == -1) {
                /*
                 * No so the process died before we got to watch for him
                 */
                return;
        }

        /*
         * We only do this code if we're not putting in the first element
         * Which we know will be for proc zero which is used by setup_pipe
         * for its pipe fd.
         */
        if (pidcnt != 0) {
                for (i = 0; i < pidcnt; i++) {
                        if (pid <= pidtable[i].pl_pid)
                                break;
                }
                /*
                 * Handle the case where we’re not sticking our entry on the
                 * the end, or overwriting an existing entry.
                 */
                if (i != pidcnt && pid != pidtable[i].pl_pid) {

                        move_amt = pidcnt - i;
                        /*
                         * Move table down
                         */
                        if (move_amt != 0) {
                                (void) memmove(&pidtable[i+1], &pidtable[i],
                                    move_amt * sizeof (struct pidentry));
                                (void) memmove(&fdtable[i+1], &fdtable[i],
                                    move_amt * sizeof (pollfd_t));
                        }
                }
        }

        /*
         * Fill in the events field for poll and copy the entry into the array
         */
        fdtable[i].events = 0;
        fdtable[i].revents = 0;
        fdtable[i].fd = fd;

        /*
         * Likewise, setup pid field and pointer (index) to the fdtable entry
         */
        pidtable[i].pl_pid = pid;

        pidcnt++;                       /* Bump the pid count */
        dprintf(("  add_pid: pid = %d fd = %d index = %d pidcnt = %d\n",
            (int)pid, fd, i, pidcnt));
}


/*
 * rem_pid      - Remove an entry from the table and check to see if its
 *                not in the utmpx file.
 *                If i != -1 don’t look up the pid, use i as index
 *
 * pid          - Pid of process to clean or 0 if we don’t know it
 *
 * i            - Index into table or -1 if we need to look it up
 *
 * clean_it     - Clean the entry, or just remove from table?
 */

static void
rem_pid(pid_t pid, int i, int clean_it)
rem_pid(pid, i, clean_it)
        pid_t pid;      /* Pid of process to clean or 0 if we don’t know it */
        int i;          /* Index into table or -1 if we need to look it up */
        int clean_it;   /* Clean the entry, or just remove from table? */
{
        int move_amt;

        dprintf(("  rem_pid: pid = %d i = %d", (int)pid, i));

        /*
         * Don’t allow slot 0 in the table to be removed - utmppipe fd
         */
        if ((i == -1 && pid == 0) || (i == 0))  {
                dprintf((" - attempted to remove proc 0\n"));
                return;
```

```
        }

        if (i != -1 || find_pid(pid, &i) == 1) {        /* Found the entry */
                (void) close(fdtable[i].fd);    /* We’re done with the fd */

                dprintf((" fd = %d\n", fdtable[i].fd));

                if (clean_it == CLEANIT)
                        clean_entry(i);

                move_amt = (pidcnt - i) - 1;
                /*
                 * Remove entries from the tables.
                 */
                (void) memmove(&pidtable[i], &pidtable[i+1],
                    move_amt * sizeof (struct pidentry));

                (void) memmove(&fdtable[i], &fdtable[i+1],
                    move_amt * sizeof (pollfd_t));

                /*
                 * decrement the pid count - one less pid to worry about
                 */
                pidcnt--;
        }
        if (i == -1)
                dprintf((" - entry not found \n"));
}


/*
 * find_pid     - Returns an index into the pidtable of the specifed pid,
 *                else -1 if not found
 */

static int
find_pid(pid_t pid, int *i)
find_pid(pid, i)
        pid_t pid;
        int *i;
{
        struct pidentry pe;
        struct pidentry *p;

        pe.pl_pid = pid;
        p = bsearch(&pe, pidtable, pidcnt, sizeof (struct pidentry), pidcmp);

        if (p == NULL)
                return (0);
        else {
                *i = p - (struct pidentry *)pidtable;
                return (1);
        }
}


/*
 * Pidcmp - Used by besearch for sorting and finding  process IDs.
 */

static int
pidcmp(struct pidentry *a, struct pidentry *b)
pidcmp(a, b)
        struct pidentry *a, *b;
{
        if (b == NULL || a == NULL)
```

```
                return (0);
        return (a->pl_pid - b->pl_pid);
}


/*
 * proc_to_fd   - Take a process ID and return an open file descriptor to the
 *                /proc file for the specified process.
 */
static int
proc_to_fd(pid_t pid)
proc_to_fd(pid)
        pid_t pid;
{
        char procname[64];
        int fd, dfd;

        (void) sprintf(procname, "/proc/%d/psinfo", (int)pid);

        if ((fd = open(procname, O_RDONLY)) >= 0) {
                /*
                 * dup the fd above the low order values to assure
                 * stdio works for other fds - paranoia.
                 */
                if (fd < EXTRA_MARGIN) {
                        dfd = fcntl(fd, F_DUPFD, EXTRA_MARGIN);
                        if (dfd > 0) {
                                (void) close(fd);
                                fd = dfd;
                        }
                }
                /*
                 * More paranoia - set the close on exec flag
                 */
                (void) fcntl(fd, F_SETFD, 1);
                return (fd);
        }
        if (errno == ENOENT)
                return (-1);

        if (errno == EMFILE) {
                /*
                 * This is fatal, since libc won't be able to allocate
                 * any fds for the pututxline() routines
                 */
                fatal("Out of file descriptors");
        }
        fatal(procname);                /* Only get here on error */
        return (-1);
}


/*
 *              *** Utmpx Cleaning Utilities ***
 */

/*
 * Clean_entry  - Cleans the specified entry - where i is an index
 *                into the pid_table.
 */
static void
clean_entry(int i)
clean_entry(i)
        int i;
{
        struct utmpx *u;
```

```
        if (pidcnt == 0)
                return;

        dprintf(("    Cleaning %d\n", (int)pidtable[i].pl_pid));

        /*
         * Double check if the process is dead.
         */
        if (proc_is_alive(pidtable[i].pl_pid)) {
                dprintf(("      Bad attempt to clean %d\n",
                dprintf(("      Bad attempt to clean %d\n", \
                    (int)pidtable[i].pl_pid));
                return;
        }

        /*
         * Find the entry that corresponds to this pid.
         * Do nothing if entry not found in utmpx file.
         */
        setutxent();
        while ((u = getutxent()) != NULL) {
                if (u->ut_pid == pidtable[i].pl_pid) {
                        if (u->ut_type == USER_PROCESS) {
                                clean_utmpx_ent(u);
                        }
                }
        }
        endutxent();
}


/*
 * clean_utmpx_ent      - Clean a utmpx entry
 */

static void
clean_utmpx_ent(struct utmpx *u)
clean_utmpx_ent(u)
        struct utmpx *u;
{
        dprintf(("      clean_utmpx_ent: %d\n", (int)u->ut_pid));
        u->ut_type = DEAD_PROCESS;
        (void) time(&u->ut_xtime);
        (void) pututxline(u);
        updwtmpx(WTMPX_FILE, u);
        /*
         * XXX update wtmp for ! nonuserx entries?
         */
}
______unchanged_portion_omitted_

/*
 * proc_is_alive        - Check to see if a process is alive AND its
 *                        not a zombie.  Returns 1 if process is alive
 *                        and zero if it is dead or a zombie.
 */

static int
proc_is_alive(pid_t pid)
proc_is_alive(pid)
        pid_t pid;
{
        char psinfoname[64];
        int fd;
        psinfo_t psinfo;
        if (kill(pid, 0) != 0)
                return (0);             /* Kill failed - no process */

        /*
         * The process exists, so check if it's a zombie.
         */
        (void) sprintf(psinfoname, "/proc/%d/psinfo", (int)pid);

        if ((fd = open(psinfoname, O_RDONLY)) < 0 ||
            read(fd, &psinfo, sizeof (psinfo)) != sizeof (psinfo)) {
                /*
                 * We either couldn't open the proc, or we did but the
                 * read of the psinfo file failed, so pid is nonexistent.
                 */
                psinfo.pr_nlwp = 0;
        }
        if (fd >= 0)
                (void) close(fd);

        /* if pr_nlwp == 0, process is a zombie */
        return (psinfo.pr_nlwp != 0);
}

/*
 * warn_utmp -  /var/adm/utmp has been deprecated. It should no longer
 *              be used.  Applications that try to directly manipulate
 *              it may cause problems. Since the file is no longer
 *              shipped, if it appears on a system it's because an
 *              old application created it.  We'll have utmpd
 *              complain about it periodically.
 */

static void
warn_utmp()
{
        struct stat s;

        if (lstat(UTMP_FILE, &s) == 0 &&
            s.st_size % sizeof (struct utmp) == 0) {
                nonfatal("WARNING: /var/adm/utmp exists!\nSee "
                    "utmp(4) for more information");
        }
}

/*
 * validate_default - validate and assign defaults.
 */

static int
validate_default(char *defp, int *flag)
{
        long lval;
        char *endptr;

        errno = 0;
        lval = strtol(defp, &endptr, 10);

        if (errno != 0 || lval > INT_MAX || lval <= 0)
                return (-1);

        while (isspace(*endptr) != 0)
                endptr++;

        if (*endptr != '\0')
                return (-1);
```

```
        *flag = lval;
        return (0);
}
______unchanged_portion_omitted_
```

***********************************************************
***118 Mon Jan 12 23:37:07 2015***
***new/usr/src/cmd/utmpd/utmpd.dfl***
***5375 utmpd(1M) core dumps when WTMPX_UPDATE_FREQ is zero***
***********************************************************

```
#
# Copyright 1994 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
#pragma ident   "%Z%%M% %I%     %E% SMI"

SCAN_PERIOD=300
```

```
**********************************************************
    4580 Mon Jan 12 23:37:08 2015
new/usr/src/man/man1m/utmpd.1m
5375 utmpd(1M) core dumps when WTMPX_UPDATE_FREQ is zero
**********************************************************
'\" te
.\" Copyright 2015 Shruti V Sampat <shrutisampat@gmail.com>
.\" Copyright (c) 2004, Sun Microsystems, Inc.  All Rights Reserved
.\" Copyright 1989 AT&T
.\" The contents of this file are subject to the terms of the Common Development
.\" You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE or http:
.\" When distributing Covered Code, include this CDDL HEADER in each file and in
.TH UTMPD 1M "Jan 01, 2015"
.TH UTMPD 1M "Jun 4, 2008"
.SH NAME
utmpd \- utmpx monitoring daemon
.SH SYNOPSIS
.LP
.nf
\fButmpd\fR [\fB-debug\fR]
.fi

.SH DESCRIPTION
.sp
.LP
The \fButmpd\fR daemon monitors the \fB/var/adm/utmpx\fR file. See
\fButmpx\fR(4) (and \fButmp\fR(4) for historical information).
.sp
.LP
\fButmpd\fR receives requests from \fBpututxline\fR(3C) by way of a named pipe.
It maintains a table of processes and uses \fBpoll\fR(2) on \fB/proc\fR files
to detect process termination. When \fButmpd\fR detects that a process has
terminated, it checks that the process has removed its \fButmpx\fR entry from
\fB/var/adm/utmpx\fR. If the process' \fButmpx\fR entry has not been removed,
\fButmpd\fR removes the entry. By periodically scanning the
\fB/var/adm/utmpx\fR file, \fButmpd\fR also monitors processes that are not in
its table.
.SH OPTIONS
.sp
.ne 2
.na
\fB\fB-debug\fR\fR
.ad
.sp .6
.RS 4n
Run in debug mode, leaving the process connected to the controlling terminal.
Write debugging information to standard output.
.RE

.SH EXIT STATUS
.sp
.LP
The following exit values are returned:
.sp
.ne 2
.na
\fB\fB0\fR\fR
.ad
.sp .6
.RS 4n
Successful completion.
.RE

.sp
.ne 2
.na
```

```
\fB\fB>0\fR\fR
.ad
.sp .6
.RS 4n
An error occurred.
.RE

.SH FILES
.sp
.ne 2
.na
\fB\fB/etc/default/utmpd\fR\fR
.ad
.sp .6
.RS 4n
You can set default values for the flags listed below. For example:
\fBSCAN_PERIOD=600\fR
.sp
The values for these flags should be greater than 0. If values read
from the file are found to be less than or equal to 0, or containing
invalid characters, the default values mentioned below are retained.
.sp
.ne 2
.na
\fB\fBSCAN_PERIOD\fR\fR
.ad
.sp .6
.RS 4n
The number of seconds that \fButmpd\fR sleeps between checks of \fB/proc\fR to
see if monitored processes are still alive. The default is 300.
.RE

.sp
.ne 2
.na
\fB\fBMAX_FDS\fR\fR
.ad
.sp .6
.RS 4n
The maximum number of processes that \fButmpd\fR attempts to monitor. The
default value is 4096.
.RE

.sp
.ne 2
.na
\fB\fBWTMPX_UPDATE_FREQ\fR\fR
.ad
.sp .6
.RS 4n
The number of seconds that \fButmpd\fR sleeps between read accesses of the
\fBwtmpx\fR file. The \fBwtmpx\fR file's last access time is used by
\fBinit\fR(1M) on reboot to determine when the operating system became
unavailable. The default is 60.
.RE

.RE

.sp
.ne 2
.na
\fB\fB/var/adm/utmpx\fR\fR
.ad
.sp .6
.RS 4n
File containing user and accounting information for commands such as
```

```
\fBwho\fR(1), \fBwrite\fR(1), and \fBlogin\fR(1).
.RE

.sp
.ne 2
.na
\fB\fB/proc\fR\fR
.ad
.sp .6
.RS 4n
Directory containing files for processes whose \fButmpx\fR entries are being
monitored.
.RE

.SH SEE ALSO
.sp
.LP
\fBsvcs\fR(1), \fBinit\fR(1M), \fBsvcadm\fR(1M), \fBpoll\fR(2),
\fBpututxline\fR(3C), \fBproc\fR(4), \fButmp\fR(4), \fButmpx\fR(4),
\fBattributes\fR(5), \fBsmf\fR(5)
.SH NOTES
.sp
.LP
If the filesystem holding \fB/var/adm/wtmpx\fR is mounted with options which
inhibit or defer access time updates, an unknown amount of error will be
introduced into the \fButmp\fR \fBDOWN_TIME\fR record's timestamp in the event
of an uncontrolled shutdown (for example, a crash or loss of power ).
Controlled shutdowns will update the modify time of \fB/var/adm/wtmpx\fR, which
will be used on the next boot to determine when the previous shutdown ocurred,
regardless of access time deferral or inhibition.
.sp
.LP
The \fButmpd\fR service is managed by the service management facility,
\fBsmf\fR(5), under the service identifier:
.sp
.in +2
.nf
svc:/system/utmp:default
.fi
.in -2
.sp

.sp
.LP
Administrative actions on this service, such as enabling, disabling, or
requesting restart, can be performed using \fBsvcadm\fR(1M). The service's
status can be queried using the \fBsvcs\fR(1) command.
```